

Duale Hochschule Baden-Württemberg Mannheim

Studienarbeit

Konzeption und Bau eines selbstspielenden Klaviers

Studiengang Informatik

Studienrichtung Angewandte Informatik

Verfasser(in):	Jakob Kautz, Olivier Stenzel, Valentin Richter
Matrikelnummer:	5101173, 6918933, 8439777
Kurs:	TINF21AI1
Studiengangsleiter:	Prof. Dr. Holger D. Hofmann
Wissenschaftliche(r) Betreuer(in):	Prof. Dr. Eckhardt Kruse
Bearbeitungszeitraum:	01.10.2023 – 16.04.2024

Ehrenwörtliche Erklärung

Wir versichern hiermit, dass wir die vorliegende Arbeit mit dem Titel „*Konzeption und Bau eines selbstspielenden Klaviers*“ selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Mannheim, 15.04.2024

Ort, Datum



The image shows three handwritten signatures. The top signature is in red ink and reads "Stenzel". Below it is a black signature that appears to read "Jakob P. Kautz". To the right of these is a blue signature that appears to read "VR". A horizontal line extends from the end of the "Kautz" signature to the right, under the "VR" signature.

Jakob Kautz, Olivier Stenzel, Val Richter

Inhaltsverzeichnis

Titelseite	i
Ehrenwörtliche Erklärung	ii
Inhaltsverzeichnis	iii
Abbildungsverzeichnis	iv
Tabellenverzeichnis	v
Quelltextverzeichnis	vi
Abkürzungsverzeichnis	vii
Kurzfassung (Abstract)	viii
1 Einleitung	1
2 Ziele und Anforderungen	3
3 Konzeption - Hardware	5
3.1 Mechanik	6
3.1.1 Auswahl des Klaviers	6
3.1.2 Ansteuerungskonzept	6
3.2 Elektronik	10
3.2.1 Mikrocontroller	10
3.2.2 Pulsweitenmodulation	12
3.2.3 Vermehrung der Ausgänge	13
3.2.4 Transistor	18
3.2.5 Aktuator	19
3.2.6 Schaltplan	24
4 Umsetzung - Hardware	29
4.1 Materialien	29
4.2 Prototypenbau	31
4.2.1 Verbindung Tasten und Aktuatoren	32
4.2.2 Klangdämpfung der Aktuatoren	38

4.2.3 Klavieranbau	40
4.2.4 Ergebnisse des Prototypen	40
5 Konzeption - Software	43
5.1 Datenformat für Musikstücke	44
5.2 Logik des Mikrocontrollers	49
5.3 Kommunikation	52
5.3.1 Fehlererkennung	55
5.3.2 „Music“-Nachrichten	58
5.4 Desktop Anwendung	61
5.4.1 UI Design	62
5.4.2 Architektur	64
6 Umsetzung - Software	69
6.1 Desktop Anwendung	70
6.2 Kommunikation	73
6.2.1 Kommunikation auf dem Mikrocontroller	73
6.2.2 Kommunikation auf der Desktop Anwendung	75
6.3 Arduino-Programmierung	77
7 Ergebnisse	79
7.1 Tests & Messungen	80
7.1.1 Hardware-Tests	80
7.1.2 Software-Tests	84
7.2 Limitationen & Erweiterungsmöglichkeiten	86
8 Zusammenfassung	89
8.1 Fazit	89
8.2 Ausblick	91
Anhang	
A Anhang: Demovideo	92
B Anhang: Quellcode	93
C Anhang: Format-Spezifikationen	94
C.1 PIDI-Format	94
C.2 PDIL-Format	96
C.3 SPPP-Protocoll	97
Literaturverzeichnis	101

Abbildungsverzeichnis

3.1	Arduino Uno	10
3.2	Raspberry Pi	11
3.3	Pulsweitenmodulation	13
3.4	Schematik Schieberegister	14
3.5	Beispielhafte Darstellung einer LED-Matrix	16
3.6	Aktuator-Matrix ohne Multiplexer	16
3.7	Matrix mit Multiplexing	17
3.8	Aufbau eines Rotors	20
3.9	Spule mit Magnetfeld	20
3.10	Schrittmotor	21
3.11	Linearer Servomotor	21
3.12	Aufbau eines Linearen Servomotors	22
3.13	Hubmagnet	23
3.14	Hubmagnet Aufbau	23
3.15	Schaltung der Schieberegister	26
3.16	Schaltung der Mosfets	26
3.17	Beispielhaftes Schema des Schaltplans	27
3.18	Überblick Schaltplan	28
4.1	Tastenbohrung	32
4.2	Bohrung durch das Tastenbrett	33
4.3	Befestigung der Hubmagnete	34
4.4	Taste locker ohne Umlenkung	35
4.5	Taste gezogen ohne Umlenkung	35
4.6	Taste mit Umlenkung	36
4.7	Fußraum des Klaviers	37
4.8	Hubmagnet: Dämpfung mit Schaumstoff	39
4.9	Hubmagnet: Dämpfung mit Seil	40
5.1	Komponenten-Diagramm mit Datenfluss	43
6.1	Screenshot der Startseite in der Desktop-Anwendung	71
6.2	Screenshot der Import-Seite in der Desktop-Anwendung	71
6.3	Screenshot der Umbenennung beim Import in der Desktop-Anwendung	72
6.4	Screenshots der Wiedergabe in der Desktop-Anwendung	72

Tabellenverzeichnis

2.1	Anforderungen an das selbstspielende Klavier	4
4.1	Übersicht der Bauteile	30
4.2	Übersicht der Kosten	31
5.1	SPPP-Nachrichten	56
7.1	Hardware-Tests	80
7.2	Software-Tests	85
8.1	Ergebnisse der Anforderungen	89

Quelltextverzeichnis

5.1	Definition eines PIDI-Kommand	48
5.2	Nutzung der <code>PidiCmd</code> -Struktur	50
7.1	Lautstärke-Skalierung	83

Abkürzungsverzeichnis

UI	User Interface
μC	Mikrocontroller
IoT	Internet of Things
PWM	Pulsweitenmodulation
V_{cc}	Voltage at the common collector
PIDI	Piano Digital Interface Format
PDIL	Piano Digital Interface Library Format
SPPP	Self Playing Piano Protocol
MIDI	Musical Instrument Digital Interface
I₂C	Inter-Integrated-Circuit
SPC	Serial Peripheral Interface
UART	Universal Asynchronous Receiver-Transmitter
SPPP	Self-Playing Piano Protocol
SAM	Self-Applying Musician
HTTP	Hypertext Transfer Protocol
FFI	Foreign Function Interface
I/O	Input/Output
GPIO	General Purpose Input/Output
WMP	Windows Media Player
FPS	Frames Per Second

Kurzfassung (Abstract)

Jakob Kautz

Deutsch: Diese Studienarbeit beschäftigt sich mit der Konzeption eines selbstspielenden Klaviers, das über einen Arduino gesteuert wird. Das Hauptziel besteht darin, eine Benutzeroberfläche zu entwickeln, über die Benutzerinnen und Benutzer Lieder aus einer Bibliothek auswählen oder eigene MIDI-Dateien einspielen können. Die MIDI-Dateien werden angepasst und in Signale umgewandelt, die dem Arduino und somit den Aktuatoren, die das Klavier betätigen, übermittelt werden. Der praktische Teil der Arbeit umfasst die Umsetzung von mindestens einer Oktave - 12 Tönen. Die Arbeit konzentriert sich hauptsächlich auf die erste Konzeption. Themen wie mechanische Belastbarkeit oder erweiterter Brandschutz werden nicht behandelt.

Englisch: This study focuses on the conception of a self-playing piano controlled via an Arduino. The main objective is to develop a user interface through which users can select songs from a library or input their own MIDI files. The MIDI files are adapted and converted into signals that are transmitted to the Arduino and thus to the actuators that operate the piano. The practical part of the work involves implementing at least one octave - 12 notes. The focus of the work is primarily on the initial conception. Topics such as mechanical robustness or extended fire protection are not considered in this work.

1 Einleitung

Olivier Stenzel, Valentin Richter

Diese Studienarbeit befasst sich mit der Frage, welchen Technik- und Kostenaufwand die Eigenentwicklung eines selbstspielenden Klaviers mit sich bringt. Professionell hergestellte, selbstspielende Klaviere existieren zwar bereits, jedoch nur mit Preisen, die leicht über 10.000 Euro gehen können [Pia24], was für Privatpersonen oft nicht bezahlbar ist. Hier soll dagegen versucht werden, mit einem deutlich geringerem Budget von maximal 2000€, ein Klavier zu entwickeln, welches die selben Funktionen bereitstellt, wie das der großen Hersteller.

Der Fokus dieser Arbeit liegt dabei in der Konzeption dieses Instruments. Über die Konzeption hinaus soll jedoch auch ein Prototyp umgesetzt werden. Der Prototyp soll hierbei nicht nur über die Technik zum automatischen Anspielen des Klaviers verfügen, sondern auch über eine Computer-Anwendung, von der aus das Piano angesteuert werden kann. Damit soll erkundet werden, wie funktional die Konzeption in der Praxis ist und mit wie viel Aufwand die Implementierung verbunden ist. Die erstellte Software des Prototypen soll dabei möglichst portabel sein, um für Eigenentwicklungen Anderer mit möglichst wenigen Änderungen verwendbar zu sein.

Ähnlich wie die professionellen, selbstspielenden Klaviere, soll der Prototyp dieser Arbeit sowohl manuell von Musiker:innen als auch automatisiert durch einen Computer bespielt werden können. Das automatisierte Anspielen des Pianos sollte über die Eingabe breit verwendeter Dateien möglich sein. Logische Kandidaten wären hier „Musical Instrument Digital Interface (MIDI)“-Dateien, aber auch generelle Audio-Dateien oder eingescannte bzw. digital vorliegende Notensheets wären hier naheliegend.

Im folgenden Kapitel werden die Anforderungen an den zu entwickelnden Prototypen genauer herausgearbeitet.

Die darauf folgenden Kapitel „Konzeption - Hardware“ und „Umsetzung - Hardware“ behandeln die Hardware, bevor dann in den Kapiteln „Konzeption - Software“ und „Umsetzung - Software“ der Software-Teil der Arbeit betrachtet wird. Dabei wird jeweils zuerst die Planung durchgeführt und dann auf die Umsetzung des Prototypen eingegangen.

In Kapitel 7 wird dann der erstellte Prototyp dann zusammengefasst. Außerdem werden Tests der Hardware und Software des Prototypen durchgeführt und Limitationen des Projekts aufgezeigt.

Abschließend zieht Kapitel 8 dann ein Fazit über diese Arbeit und fasst zusammen, welche der gestellten Anforderungen erfüllt werden konnten.

2 Ziele und Anforderungen

Olivier Stenzel

Das Kernanliegen dieser Arbeit ist die Entwicklung eines multifunktionalen Klaviers, das sowohl manuell von Musiker:innen als auch automatisiert durch einen Computer bespielt werden kann.

Da es wünschenswert ist, dass Nutzer:innen auf eine breite Palette fertiger Stücke zugreifen und diese Palette selbstständig erweitern können, wäre eine Steuerung über den PC sinnvoll. Neben der Auswahl von Musikstücken soll die Anwendung außerdem die Möglichkeit bieten, die Wiedergabegeschwindigkeit und Lautstärke individuell anzupassen.

In Anlehnung an die Funktionsweise populärer Musikplayer wird angestrebt, dass die Wiedergabe nicht nur pausiert, sondern auch innerhalb eines Stücks frei navigiert werden kann.

In Tabelle 2.1 befinden sich die allgemeinen Anforderungen, die an dieses Projekt gestellt wurden. Basierend auf den Anforderungen und deren Priorisierungen kann die Arbeit in kleinere Arbeitspakete aufgeteilt werden. Zuerst sollen die Anforderungen mit hoher Priorität umgesetzt werden und im Anschluss jene mit einer niedrigeren. Im weiteren Verlauf dieser Arbeit wird nicht weiter auf die Priorisierung eingegangen, da nur Anforderungen mit niedriger Priorität nicht umgesetzt wurden.

Wie an den genannten Anforderungen erkenntlich wird, lässt sich diese Arbeit entlang der Software-/Hardware-Grenze teilen. Das ist insoweit sinnvoll, dass die Planung und Umsetzung der beiden Teile dann parallel getan werden kann.

ID	Kategorie	Anforderung	Priorität
A1	Allgemein	Flexibilität: Das Klavier muss sowohl manuell als auch automatisch bespielbar sein	Hoch
A2		Benutzerinterface: Entwicklung eines intuitiven Interfaces	Hoch
A3		Budget: Gesamtkosten < 2000€	Mittel
A4		Portabilität: Die Anwendung muss auf verschiedenen Computern mit Windows 10 und 11 laufen	Mittel
A5		Performanz: Die Performanz soll mindestens die von professionellen Pianist:innen erreichen, Genauer bedeutet dies, dass im Vollbetrieb maximal 70ms zwischen zwei nacheinanderfolgenden Tastenanschlägen liegen sollen ¹	Niedrig
A6		Spielbarkeit: Der Aufbau sollte dazu in der Lage sein, mindestens 10 Tasten gleichzeitig anspielen zu können	Mittel
A7	Hardware	Tastenbetätigung: Die Hardware muss in der Lage sein, Klaviertasten autonom zu bedienen	Hoch
A8		Anpassbarkeit: Hardware- & Software-Komponenten müssen auf unterschiedliche Klaviertypen anpassbar sein	Mittel
A9	Software	Musikstück-Auswahl: Musikstücke sind über eine Desktop-Anwendung wählbar	Hoch
A10		Wiedergabe-Kontrolle: Anpassung der Wiedergabegeschwindigkeit und Lautstärke	Mittel
A11		Navigation: Möglichkeit, die Wiedergabe zu pausieren und innerhalb des Stücks zu navigieren	Mittel
A12		MIDI-Integration: System muss MIDI-Dateien für die Erweiterung des Katalogs unterstützen	Mittel
A13		Musikverwaltung: Musikstücke im Katalog müssen jederzeit umbenannt und entfernt werden können	Niedrig

Tabelle 2.1: Anforderungen an das selbstspielende Klavier

3 Konzeption - Hardware

Jakob Kautz, Olivier Stenzel

Für die Umsetzung eines selbstspielenden Klaviers wurden verschiedene technische Überlegungen angestellt. Die Grundfragen, welche bei der Konstruktion auftreten, betreffen die Methode des Anspielens der Klaviertasten, die Signalübertragung des Mikrocontroller (μ C)s, sowie die Konzeption der Schaltung.

Um die Tasten des Klaviers via Hardware bedienen zu können, werden Aktuatoren - also Bauteile, welche elektrische Impulse/Signale in mechanische (o.ä.) Bewegungen umsetzen - benötigt um die erforderliche Bewegung der Tasten zu ermöglichen. Zusätzlich müssen die Informationen wann welcher Aktuator betätigt werden muss an diese weitergegeben werden. Dafür wird eine Schnittstelle zwischen Anwendung (Software) und den Aktuatoren genutzt. Für diese Schnittstelle werden in der Arbeit verschiedene μ Cs betrachtet.

Folgende Anforderungen muss die Konzeption abdecken:

1. **Ansteuerungskonzept:** Die Tasten müssen möglichst präzise und effizient von der Hardware angespielt werden können; dies beinhaltet Positionierung der Aktuatoren und die Verbindung zwischen Hardware und Klavier
2. **Auswahl der Aktuatoren:** Die Aktuatoren müssen eine präzise Steuerung der Tasten ermöglichen, ohne dass Genauigkeit und Geschwindigkeit des Anspielens der Tasten vernachlässigt wird
3. **Auswahl des μ C:** Die Aktuatoren benötigen zur Ansteuerung der Klaviertasten die Signale der Software, wobei die Kommunikation über einen μ C geschieht

Die Überlegungen und Entscheidungen für die genutzte Hardware sowie den Aufbau dieser werden in diesem Kapitel erläutert. Die Hardware-Komponente des Projektes wird im folgenden auch als „Piano Player“ bezeichnet.

3.1 Mechanik

3.1.1 Auswahl des Klaviers

Olivier Stenzel

Da sich dieses Projekt nicht auf die theoretische Konzeption beschränkt, muss ein entsprechendes Testobjekt - ein reales Klavier - gefunden werden. Dieses muss einige Voraussetzungen erfüllen:

1. Der Preis muss im Rahmen des selbst gestellten Budgets (< 2000€) liegen
2. Es muss leicht auseinander zu bauen sein, damit die Mechanik zugänglich ist
3. Es muss vollständig sein (alle 88 Tasten)
4. Es muss stimmbar sein
5. Der Transport muss einfach durchzuführen sein

Mittels dieser Anforderungen wurde online ein passendes Klavier für 100€ ersteigert.

3.1.2 Ansteuerungskonzept

Jakob Kautz

Das Ansteuerungskonzept bezieht sich auf die Problemstellung, dass die Saiten des Klaviers auf irgendeine Art automatisiert zum Schwingen gebracht werden müssen. Dafür ist eine Vorrichtung hilfreich, die elektrische Signale in Bewegung umwandeln kann. Dies ist die Eigenschaft von Aktuatoren, wie z.B. Motoren.

In diesem Kapitel wird untersucht, wie genau man das Anspielen der Saiten technisch umsetzen kann. Dabei ist eine grundlegende Überlegung, ob man die Saiten direkt anschlägt, oder die bestehenden Tasten dafür nutzt.

Wenn die Aktuatoren direkt die Saiten des Klaviers anspielen, erfordert dies weniger Kraft im Vergleich zum Anspielen der Tasten. Das liegt daran, dass die Tasten eine größere mechanische Übersetzung bieten, um die Saiten anzuschlagen. Wenn die Aktuatoren direkt auf die Saiten wirken, müssen sie nur die erforderliche Kraft aufbringen, um die Saiten in

Schwingung zu versetzen, was im Allgemeinen weniger Kraft erfordert als das Drücken einer Taste mit ausreichend Kraft, um den Hammer gegen die Saiten zu schlagen.

Die Verwendung von weniger kraftaufwendigen Aktuatoren kann die Gesamtkosten des Systems senken. Aktuatoren, die weniger Kraft erzeugen müssen, sind oft einfacher und kostengünstiger herzustellen (und zu kaufen) und erfordern möglicherweise weniger energieintensive Komponenten. Darüber hinaus kann die Verwendung von leistungsschwächeren Aktuatoren die Größe und das Gewicht des Systems verringern, was zusätzliche Vorteile hinsichtlich Kosten, Transport und Montage bieten kann.

Allerdings geht dabei die gesamte Mechanik des Klaviers verloren, was bedeutet, dass Aspekte wie Dämpfung nicht genutzt werden können, was wiederum zu einem weniger ansprechenden Klang führt. Zusätzlich müsste das Klavier permanent geöffnet bleiben, und die Aktuatoren müssten äußerst präzise die Saiten anschlagen, um akzeptable Ergebnisse zu erzielen. Da die Erzeugung eines ansprechenden Klangs für dieses Projekt eine höhere Priorität annimmt, fiel die Entscheidung darauf, die Tasten anzuspielen. Somit kann die Klaviermechanik genutzt werden, was für einen authentischeren Klang sorgt.

Nun stellt sich also noch die Frage, wie genau die Aktuatoren mit den Tasten des Klaviers verbunden werden.

Eine Möglichkeit besteht darin, die Aktuatoren oben über den Tasten anzubringen, entweder in Form einer nachgebildeten „Klavierhand“ mit zehn „Fingern“ oder in Form einer Schiene mit 88 Aktuatoren auf den Tasten. Beide Lösungen bringen allerdings das Problem mit sich, dass sie gegen die in Kapitel 2 definierte Anforderung A6 der Flexibilität verstößen, nämlich dass das Klavier sowohl automatisch als auch manuell bespielbar sein soll. Außerdem würde die Klavierhand-Option - die dem tatsächlichen Klavierspiel ähnlich sieht - aufgrund der Bewegung und Präzision eine komplexere Logik und Montage erfordern. Die Schienenooption bietet hier eine viel einfachere Ansteuerung, benötigt jedoch einen Aktuator für jede Taste. Außerdem steht diese Variante im Gegenspruch zur Anforderung A8, dass der Aufbau für unterschiedliche Klaviere anpassbar sein soll. Das liegt daran, dass diese Schiene eine fest vorgegebene Länge hätte, welche für Klaviere mit anderen Größen nicht mehr passen würde.

Eine Alternative hierzu besteht darin, die Aktuatoren unter den Tasten anzubringen, wodurch die Anforderung der Flexibilität nicht beeinflusst wird.

Das Klavier kann also sowohl vom „Piano Player“ als auch von einem menschlichen Spieler gleichzeitig bedient werden. Da diese Anforderung eine hohe Priorität hat, wird die An-

steuerung von untern erfolgen.

Grundsätzlich gibt es hierbei zwei Möglichkeiten für die Ansteuerung:

1. Ziehen der Tasten
2. Drücken der Tasten

Bei beiden Varianten wird jede Taste mit einem Aktuator ausgestattet, es werden also 88 Aktuatoren benötigt, was die Kosten im Gegensatz zu der von oben Spielenden „Klavierhand“ erhöht. Generell ist die Drückoption ästhetisch ansprechender, da die Hardware sehr einfach versteckt werden könnte. Somit würde die Illusion entstehen, dass die Tasten sich von alleine bewegen. Die Option erfordert jedoch eine hohe Präzision und könnte die Tastenempfindlichkeit beeinträchtigen. Außerdem wäre aufgrund der präzisen Montage der Aktuatoren die Anforderung der Anpassbarkeit (A8) aufwendiger zu erfüllen.

Für das Drücken der Tasten fallen noch weitere Probleme an. Um die Tasten drücken zu können, muss ein möglichst großer Hebel aufgebracht werden, damit die Aktuatoren möglichst wenig Kraft für das anspielen aufbrauchen müssen. Dieser Punkt ist besonders wichtig, wenn der „Piano Player“ mit unterschiedlichen Lautstärken und Dynamiken spielen können soll (siehe Anforderung A10). Dies führt dazu, dass die Aktuatoren soweit hinten wie möglich angebracht werden müssen, um einen größeren Hebeleffekt zu erhalten.

Hierbei tritt nun das folgende Problem auf: Der Holraum, der sich im hinteren Teil des ausgewählten Klaviers befindet, ist nicht durchgängig. Daher ist es erforderlich, Löcher einzubauen. Diese müssen räumlich so platziert werden, dass sie keine wichtigen Teile der Klaviermechanik stören. Aufgrund des Klavierbaus bedeutet dies, dass die Löcher etwa im mittleren Drittel der Taste angebracht werden müssen. Dies führt zu einer verringerten Hebelkraft. Das Problem betrifft nur etwa ein Viertel der Tasten. Die restlichen könnten mit voller Hebelkraft von ganz hinten angespielt werden. Allerdings würde dies dazu führen, dass die Tasten

1. unterschiedlich stark angespielt werden und der Klang somit seltsam ist,
2. der Anstoß der Tasten genormt wird und somit nicht die gesamte Dynamik der hinten angebrachten Aktuatoren genutzt wird, damit alle gleich klingen. Dies würde eine komplexere Software erfordern, da dies berücksichtigt werden muss, um eine konsistente Leistung zu gewährleisten.

Zur Vereinfachung wäre es daher sinnvoller, alle Tasten von der gleichen Höhe aus anzuspielen. Dies bedeutet, dass alle Tasten von weiter vorne angespielt werden und somit ein

Teil der möglichen Dynamik verloren geht.

Diese Probleme fallen beim Ziehen der Tasten weg bzw. sind weniger ausgeprägt. Bei der Präzision liegt der Grund dafür darin, dass die Taste nicht direkt getroffen werden muss. Stattdessen wird lediglich ein Seil (siehe Kapitel 4.2.1) gezogen, das den Aktuator dazu veranlasst, die Taste gerade nach unten zu ziehen. Da das Ziehen des Seils weniger präzise Kontrolle erfordert und keine direkte Interaktion mit der Tastenmechanik erfordert, kann es auch die Tastenempfindlichkeit weniger beeinträchtigen als das Drücken der Tasten. Es ist allerdings zu beachten, dass das ziehen der Seile - im Gegensatz zum drücken - keine optimalen Ergebnisse im Bereich des präzisen Spielens bringen. Dies hat mehrere Gründe.

1. **Verzögerung:** Wenn das Seil sich über die Zeit lockert, kann es sein, dass die Taste nicht direkt mit der Bewegung des Aktuators betätigt wird. Außerdem würde die Taste so nicht komplett gedrückt werden.
2. **Verschleiß:** Die Bewegung über Führsysteme oder Umlenkrollen, welche für eine gerade Bewegung der Seile benötigt werden, können zu Reibungsverlust führen. Außerdem könnten die Seile sich durch verschleiß lockern oder verschieben.

Letztendlich handelt es sich bei beiden Problemen um Gebrauchserscheinungen, welche durch eine gute Auswahl von Seilen und Wartungen am Klavier, wenn es öfter in Benutzung ist, minimiert werden können. Die Hebelkraft ist ebenfalls ein geringeres Problem, da die Aktuatoren problemlos am vordersten Punkt der Tasten befestigt werden können, da hier - im Gegensatz zum hinteren Teil des Klaviers - nichts im Weg ist.

Insgesamt fiel die Entscheidung deshalb auf die Strategie des Ziehens von unten. Bei diesem Ansatz wird die geringste mechanische Präzision benötigt und das Klavier ist trotz Montage der Aktuatoren noch manuell bespielbar. Das Problem, dass es bei den Tasten verschiedene „Mechanik-Gewichte“ gibt, wird bei dem Ansteuerungs-Konzept vernachlässigt. An sich geht es hierbei darum, dass die tieferen Tasten mehr Kraftaufwand zum Spielen erfordern als die der höheren Töne. Technisch gesehen könnte der Kraftaufwand berechnet werden und die Aktuatoren mit den entsprechenden Hebeln angebracht werden, dass die Unterschiede im mechanischen Gewicht ausgeglichen werden. Dies schien in der Konzeption allerdings nicht wichtig genug und kann - wenn die späteren Tests zeigen dass die Unterschiede zu klanglichen Problemen führen - immernoch in der Software angepasst werden.

3.2 Elektronik

3.2.1 Mikrocontroller

Jakob Kautz

Um die Signale des Programmes an die Aktuatoren weitergeben zu können, ist ein Mikrocontroller (μ C) gut geeignet. Auf dem Markt steht eine hohe Anzahl an μ Cs zur Verfügung, wobei für diese Arbeit nur ein Arduino und ein Raspberry Pi¹ betrachtet werden sollen.

Arduino Ein Arduino unterstützt die Entwicklung von elektronischen Prototypen. Er besteht aus einem μ C-Board, das mit verschiedenen Sensoren, Aktuatoren und anderen elektronischen Komponenten verbunden werden kann. Der Arduino verfügt über digitale und analoge Ein- und Ausgangspins, die für die Interaktion mit Geräten verwendet werden können [vgl. Aut24].



Abbildung 3.1: Arduino Uno

Quelle: [Aut24]

¹Genau genommen handelt es sich bei einem Raspberry Pi nicht um einen μ C, sondern um einen Einplatinencontroller [vgl. 22].

Raspberry Pi Ein Raspberry Pi ist ein Einplatinencontroller, der auf einem ARM-Prozessor basiert. Er ist dafür konzipiert, eine breite Palette von Anwendungen zu unterstützen, vom Prototypenbau bis zu Internet of Things (IoT)-Geräten. Im Gegensatz zum Arduino ist er besonders für komplexe und rechenlastige Projekte geeignet [vgl. 22].



Abbildung 3.2: Raspberry Pi

Quelle: [Okd24]

Entscheidungsfindung Es wurde sich letztendlich für den Arduino entschieden. Der Raspberry Pi stand vor allem aufgrund seiner höheren Speicherkapazität und Rechenleistung zur Diskussion, wodurch eine komplexere Ansteuerungslogik möglich wäre. Allerdings ist er aufgrund seines Betriebssystems nicht für Echtzeit-Anwendungen bzw. geschwindigkeitskritische Anwendungen ausgelegt. Im Gegensatz dazu bietet der Arduino eine Echtzeitverarbeitung mit geringer Latenz. Außerdem verfügt ein Arduino über eine recht simple Hardware-Interaktion - er ist darauf spezialisiert, die Hardware direkt anzusprechen - und ist somit besser für Projekte geeignet, die eine möglichst Latenzfreie Ansteuerung von Aktuatoren benötigen [vgl. 22]. Zudem sollte hier keine besonders komplexe Ansteuerungslogik benötigt werden, weshalb ein Arduino genug Rechenleistung aufweisen sollte.

Für das Projekt wird spezifisch ein Arduino UNO R3 verwendet [siehe Ard24c, für die Spezifikation]. Bei der Auswahl des spezifischen Arduinos wurden der Arduino UNO Mini, Nano und R3 betrachtet. Im Prinzip wäre jedes der genannten Modelle für die Anwendung möglich, allerdings verfügt der R3 im Gegensatz zu den anderen beiden Modellen über mehr Speicherkapazitäten (256KB Flash Speicher gegenüber der 32KB vom Arduino

UNO Mini und den 30KB, die auf dem Arduino Nano verfügbar sind [vgl. Ard24c; Ard24b; Ard24a]). Der Arduino UNO R3 verfügt zwar über mehr Ports, allerdings bringt dies keinen Mehrwert, da die Anzahl der benötigten Ausgänge auch bei einem R3 nicht erreicht werden (siehe Kapitel 3.2.3). Der Arduino R3 bietet - wie die anderen beiden Modelle auch - Pulsweitenmodulation (PWM)-Pins, die im Rahmen des Projekts genutzt werden können.

3.2.2 Pulsweitenmodulation

Jakob Kautz

Zur Vollständigkeit wird in diesem Abschnitt das Prinzip der Pulsweitenmodulation (PWM) erläutert. Zur Vollständigkeit wird in diesem Abschnitt das Prinzip der Pulsweitenmodulation erläutert [vgl. Hir22]. Oftmals ist bei der Ansteuerung der Aktuatoren nicht die gesamte Versorgungsspannung erwünscht bzw. benötigt. In diesen Fällen muss die anliegende Spannung variiert werden, damit die Spannung am Ziel dem gewünschten Wert entspricht. Angenommen es ist eine Spannung von 2.5V erwünscht, wobei die Vollversorgungsspannung 5V beträgt. Das Signal kann dafür die Hälfte der Zeit ausgeschaltet werden, womit zwischen 0V und den vollen 5V durchschnittlich insgesamt 2.5V anliegen. Je höher die Frequenz zwischen An- und Ausschalten des Signals eingestellt wird, desto weniger wird diese „künstlich“ simulierte Halbierung wahrgenommen.

PWM bedient sich im Grunde genau dieser Technik. Dabei wird die Zeitspanne eines digitalen Signals variiert, um einen durchschnittlichen Wert zu erzeugen. Bei den PWM-Ausgängen wird die Pulsbreite - die Dauer der Einschaltzeit - des Signals angepasst, um die gewünschte Spannung zu erreichen.

Spezifischer ausgedrückt passiert folgendes: Eine digitale Steuerung wird verwendet, um eine Rechteckwelle - ein Signal, das zwischen HIGH und LOW umgeschaltet wird - zu erzeugen. Dieses HIGH-LOW-Muster kann Spannungen zwischen der vollen Versorgungsspannung und 0V simulieren. Dabei wird der Anteil der Zeit geändert, für den das Signal auf HIGH geschaltet ist, relativ zur Zeit, in der das Signal auf LOW geschaltet ist. Um unterschiedliche, analoge Werte zu erhalten, wird die Pulsbreite moduliert. Wenn dieses HIGH-LOW-Muster zum Beispiel schnell genug mit einer LED wiederholt wird, resultiert daraus eine konstante Spannung zwischen 0 und Voltage at the common collector (Vcc), die die Helligkeit der LED steuert.

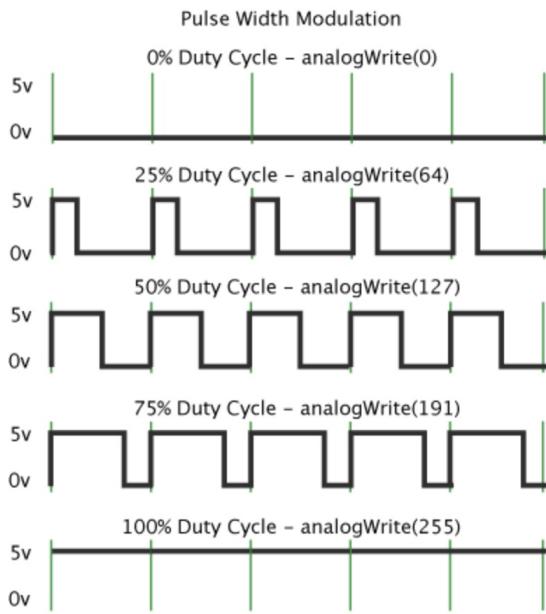


Abbildung 3.3: Pulsweitenmodulation

Quelle: [Hir22]

Auf den Arduino bezogen sieht die Umsetzung eines PWM Signals wie folgt aus: Ein Taktsignal gelangt in die entsprechende Clock. Die Clock stellt den entsprechenden PWM-Modus ein. Dabei werden zwei wichtige Werte gesetzt: Der erste bestimmt, wann das Signal von HIGH auf LOW umschaltet, während der zweite bestimmt, wann es zurückkommt. Das Verhältnis zwischen HIGH und LOW wird als Tastverhältnis bezeichnet und bestimmt die Helligkeit der LED bzw. die Stärke, mit der der Aktuator anschlägt. Je länger die Ausgabe im HIGH-Zustand bleibt, desto schneller erfolgt entsprechend der Tastenanschlag.

Neben dem Tastverhältnis, welches oft in Prozent ausgedrückt wird, ist auch die Auflösung ein variierbarer Parameter. Die Auflösung bezieht sich auf die Anzahl der möglichen diskreten Werte, die das Signal annehmen kann.

3.2.3 Vermehrung der Ausgänge

Jakob Kautz

Da ein Klavier über 88 Tasten verfügt, müssen 88 Aktuatoren angesteuert werden. Der gewählte Arduino hat keine 88 PWM-Ports, daher müssen die Signale über eine Erwei-

terung der Ausgänge an die Aktuatoren weitergegeben werden. Dafür gibt es mehrere Möglichkeiten, wobei in dieser Arbeit 2 im Detail betrachtet werden:

1. Schieberegister
2. Aktuator-Matrix

Schieberegister

Ein Schieberegister ist ein integrierter Schaltkreis, der zur Speicherung und sequenziellen Verschiebung von Datenbits verwendet wird [vgl. Ele22].

Um 88 Ausgänge mit Hilfe von Schieberegistern, die von nur 3 PWM-Pins des Arduinos gesteuert werden, umzusetzen, können 11 8-Bit-Schieberegister - also Schieberegister mit 8 Ausgängen - verwendet werden.

74HC959 Schieberegister In dieser Arbeit wird spezifisch ein 74HC959 Schieberegister betrachtet [siehe Inc18, für die Spezifikation]. Dieses hat folgenden Aufbau: Ein Nachteil

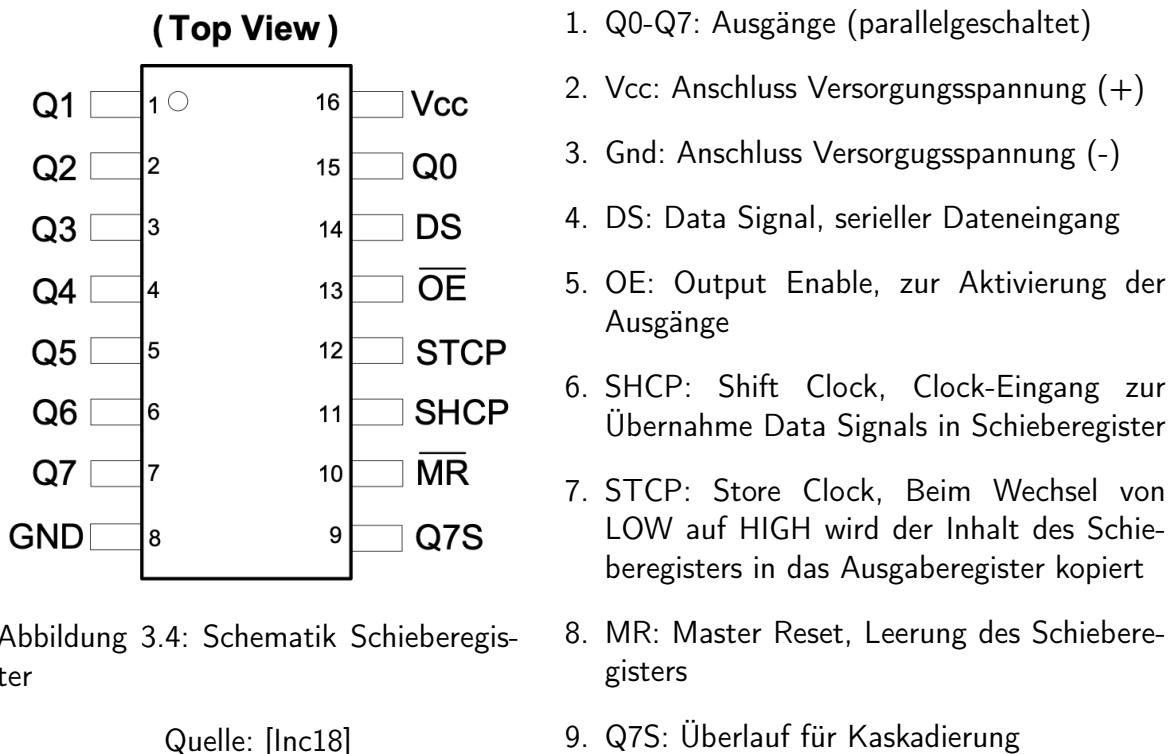


Abbildung 3.4: Schematik Schieberegister

Quelle: [Inc18]

des Schieberegisters besteht darin, dass Latenzen auftreten.

Aktuator-Matrix

Die Aktuator-Matrix ist einer LED-Matrix nachgeahmt. In einer Matrix, werden zwei Reihen nach folgendem Muster an Ports angeschlossen:

$$\begin{pmatrix} (11) & (12) & (13) & (14) & (15) & (16) & (17) & (18) \\ (21) & (22) & (23) & (24) & (25) & (26) & (27) & (28) \\ (31) & (32) & (33) & (34) & (35) & (36) & (37) & (38) \\ (41) & (42) & (43) & (44) & (45) & (46) & (47) & (48) \\ (51) & (52) & (53) & (54) & (55) & (56) & (57) & (58) \\ (61) & (62) & (63) & (64) & (65) & (66) & (67) & (68) \\ (71) & (72) & (73) & (74) & (75) & (76) & (77) & (78) \\ (81) & (82) & (83) & (84) & (85) & (86) & (87) & (88) \end{pmatrix}$$

Eine LED-Matrix besteht aus einer Anordnung von LEDs in Zeilen und Spalten. Jede LED kann unabhängig von den anderen ein- oder ausgeschaltet werden. Die Steuerung der Matrix kann sowohl mit als auch ohne Multiplexing erfolgen. Prinzipiell wird jede Zeile der Matrix nacheinander aktiviert, während die entsprechenden LEDs in den Spalten gleichzeitig eingeschaltet werden. Durch schnelles Wechseln zwischen den Zeilen mithilfe von PWM-Signalen erscheint es den Betrachter:nnen, als ob alle LEDs gleichzeitig leuchten würden, obwohl sie tatsächlich nacheinander aktiviert werden [vgl. Fal+23].

Matrix ohne Multiplexing Bei einer LED-Matrix ohne Multiplexing werden die LEDs direkt über die General Purpose Input/Output (GPIO)-Pins des μ Cs angesteuert. Jede LED ist einzeln mit einem Pin des μ Cs verbunden. Um die LEDs anzusteuern, muss der μ C jeden Pin einzeln aktivieren oder deaktivieren, um die entsprechende LED ein- oder auszuschalten. Hierbei werden viele Pins benötigt, um die gesamte Matrix anzusteuern, was besonders bei größeren Matrizen unpraktisch sein kann. Für dieses Projekt wird, wie oben aufgeführt, eine 8x8-Matrix benötigt. Es werden also 16 PWM-Fähige Pins benötigt. Der hier verwendete μ C verfügt allerdings nur über 6 von diesen Pins.

Abbildung 3.6 zeigt den Aufbau einer Aktuator-Matrix ohne Multiplexing. Dargestellt, ist eine 3x3 Matrix, mit externer Stromversorgung (Da das Tool nur maximal 9V Batterien zur

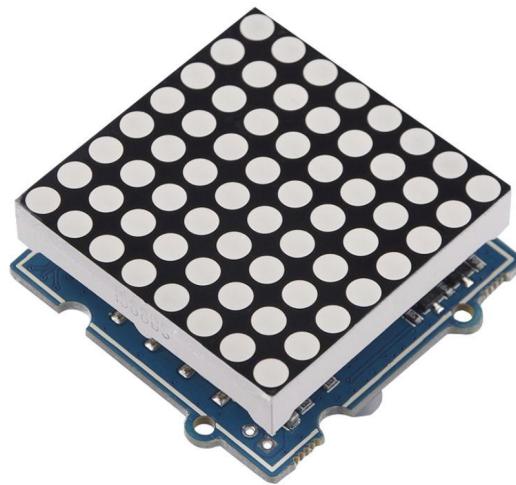


Abbildung 3.5: Beispielhafte Darstellung einer LED-Matrix

Quelle: [Far24]

auswahl bietet, wurden für eine Simulation mehrere davon in Reihe geschaltet). Statt Aktuatoren wurden zum Testen Messgeräte angeschlossen. Die eigentliche Matrix-Schaltung ist auf den Breadboards zu sehen. Dort ist dargestellt, wie die Signale verteilt sind.

Abbildung 3.6: Aktuator-Matrix ohne Multiplexer

Matrix mit Multiplexing Ein Multiplexer (bzw. ein Demultiplexer) ermöglicht das Kombinieren mehrerer Signale zu einem bzw. dem Trennen eines einzelnen Signals zu mehreren, unterschiedlichen Signalen. Bei einer LED-Matrix erfolgt das Prinzip durch die Verwendung von Schieberegistern und Transistoren. Somit kann damit die Notwendigkeit zum Verbinden jedes individuellen Pins mit einem GPIO-Pin des μ Cs eliminiert werden.

16

Entsprechend werden weniger Pins benötigt, um die Matrix anzusteuern. Die Nutzung von Multiplexern bringt somit eine effizientere Ressourcennutzung mit sich, kommt allerdings auch mit einer komplexeren Schaltung und Latenzen durch die Schieberegister einher. Wo bei die Latenzen der Schieberegister in diesem Projekt vernachlässigt werden können.

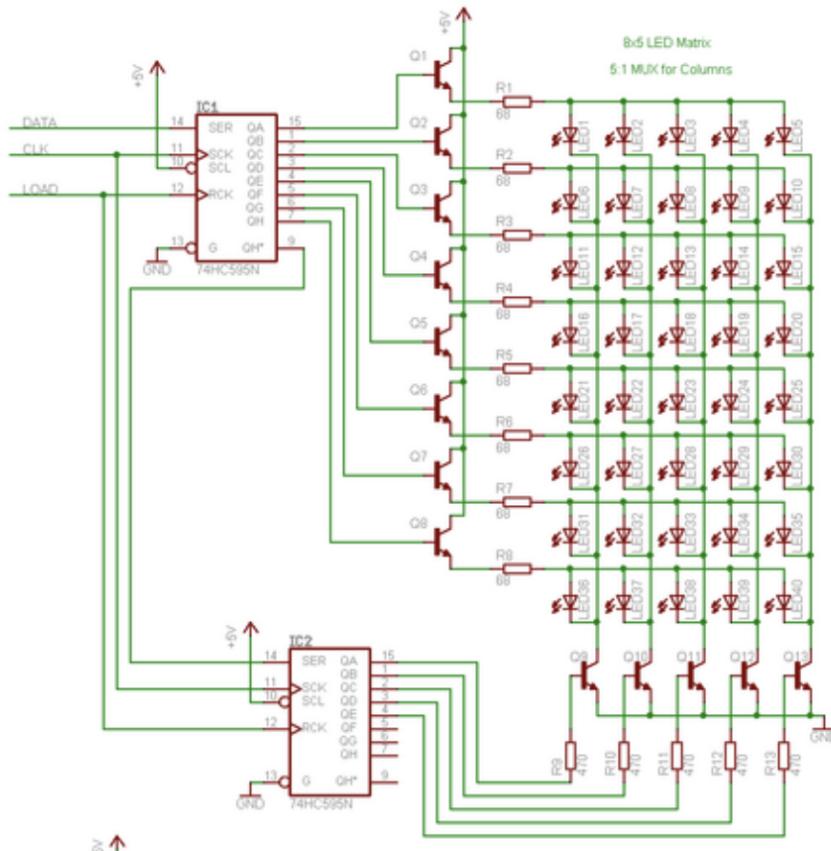


Abbildung 3.7: Matrix mit Multiplexing

Quelle: [Fal+23]

Entscheidungsfindung

Letztendlich fiel die Entscheidung auf die Verwendung von Schieberegistern. Dies liegt an mehreren Gründen:

- Funktionalität:** Bei der Verwendung einer Aktuatoren-Matrix über PWM kann es zu Interferenzen kommen, die die Präzision der Steuerung beeinträchtigen. Da die PWM-Signale zeilenweise vom Multiplexer kombiniert werden, kann es vorkommen,

dass die PWM-Signale nicht mit der benötigten Stärke für jeden Aktuator liefert werden. Das liegt daran, dass verschiedene Aktuatoren in derselben Zeile unterschiedliche PWM-Signale benötigen können, was zu ungenauer Steuerung führt. Dies kann zu Kompromissen bei der Präzision und Leistung der Aktuatoren führen.

2. **Komplexität:** Die Implementierung einer Aktuator-Matrix über eine LED-Matrix hinaus wäre technisch anspruchsvoller und erfordert eine komplexere Schaltung. Im Gegensatz dazu ist die Verwendung von Schieberegistern eine einfachere Methode zur Ansteuerung der Aktuatoren.
3. **Anzahl der Pins:** Wie bereits erklärt, benötigt die Matrix ohne Multiplexing eine Vielzahl an Pins, welche der μ C nicht liefert. Da die Matrix verwendet werden sollte um eben dieses Problem zu lösen, liefert die Matrix keine gute Lösung. Durch die Kombination von Schieberegistern und Matrixaufbau wäre dieses Problem gelöst. Es würden insgesamt 2 Schieberegister benötigt werden, also 6 Pins die diese ansteuern, womit die Anzahl der verfügbaren Pins ausreichen würde. Die Entscheidung von dieser Variante abzusehen, ist in den anderen hier aufgeführten Punkten begründet.
4. **Ressourcen:** Des Weiteren basiert die Entscheidung auf unzureichenden Ressourcen für die Matrix. Im Internet gab es keine ausreichenden Anleitungen/Ressourcen, die die Implementierung einer LED-Matrix mit Aktuatoren ausreichend unterstützen würden. Zusätzlich dazu lieferten die Simulationen via Tinkercad² keine befriedigenden Ergebnisse für die Aktuator-Matrix. Im Gegensatz dazu sind Schieberegister gut dokumentiert und es gibt ausreichende Ressourcen, um ihre Verwendung für die Ansteuerung von Aktuatoren zu verstehen und umzusetzen.

3.2.4 Transistor

Jakob Kautz

Ein Transistor ist ein elektronisches Bauteil, das in der Lage ist, den Stromfluss zwischen zwei seiner Anschlüsse (den sogenannten Source und Drain) mithilfe eines dritten Anschlusses (der Gate) zu steuern [Sch24, vgl.].

Verwendung von Transistoren in Aktuatoren-Schaltungen: Transistoren werden in Aktuatoren-Schaltungen verwendet, um die Aktuatoren zu steuern. Sie dienen als Schalter, der den

²Tinkercad: <https://www.tinkercad.com>

Stromfluss zu den Aktuatoren regelt. Dabei ist es wichtig, auf die Durchlassspannung und Mindestspannung des MOSFETs zu achten. Die Durchlassspannung ist die minimale Spannung, die an das Gate angelegt werden muss, um den MOSFET in den leitenden Zustand zu versetzen. Die Mindestspannung ist die minimale Spannung, die zwischen Source und Gate anliegen muss, um den MOSFET zuverlässig zu sperren. Es ist entscheidend, sicherzustellen, dass die Spannungen in der Schaltung diese Anforderungen erfüllen, um eine ordnungsgemäße Funktion des MOSFETs sicherzustellen und Schäden zu vermeiden.

3.2.5 Aktuator

Jakob Kautz

Die Aktuatoren werden für die Steuerung der Tasten benötigt. Bei den möglichen Aktuatoren für die Ansteuerung der Klaviertasten wurden drei Möglichkeiten betrachtet:

1. Elektromotor: Schrittmotor
2. Elektromotor: linearer Servomotor
3. Hubmagnet

An sich funktionieren alle 3 Aktuatoren über das selbe Prinzip: elektromagnetische Induktion. Der Hauptunterschied, der in dieser Arbeit betrachtet wird, ist einmal der Unterschied zwischen einer rotierenden und linearen Bewegung der Aktuatoren, und im Späteren noch wie die Aktuatoren bezüglich Präzision, Kraft, etc. abschneiden.

Elektromotor Zu Beginn soll erst einmal das Prinzip hinter Elektromotoren betrachtet werden, da dies hinter allen gewählten Aktuatoren ähnlich ist.

Grundlegend handelt es sich bei einem Elektromotor um ein Bauteil, welches elektrische Leistung in mechanische umwandelt. Er besteht aus einem von Statoren erzeugten Magnetfeld und einem sich drehenden Magneten (Rotor) im Inneren [vgl. Ele23].

Die in Abbildung 3.8 dargestellten Nummern markieren die folgenden Komponenten:

1. Dauermagnet
2. Drehbares Eisenteil
3. Spule

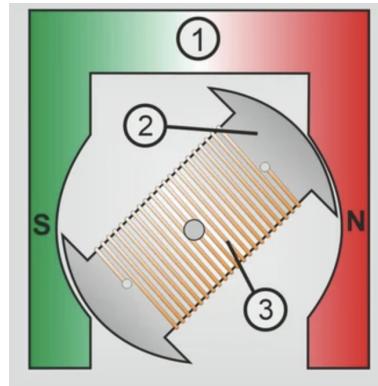


Abbildung 3.8: Aufbau eines Rotors

Quelle: [Ele23]

Wird ein Gleichstrom angelegt, fließt dieser durch die Spule, welche so ein Magnetfeld (siehe Abbildung 3.9) erzeugt.

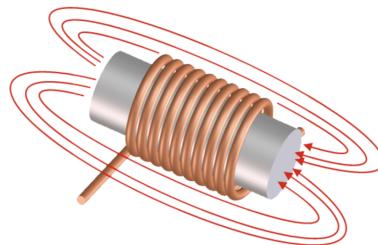


Abbildung 3.9: Spule mit Magnetfeld

Quelle: [Ele23]

Der Eisenkern wird nun zum Elektromagneten. Die Polung des Magneten hängt davon ab, in welche Richtung der Strom fließt. Um die Drehbewegung zu ermöglichen, ist die Positionierung des Magneten so, dass der Süd- am Nordpol und der Nord- am Süd-Pol sind. Die gewünschte Rotierbewegung erfolgt durch das umpolen der magnetischen Ausrichtung. Dafür ist ein sogenannter Kollektor zuständig, welcher die Stromrichtung in der Spule umschaltet [vgl. Ele23].

Schrittmotor Ein Schrittmotor übt eine Rotierende Bewegung aus und ist im Wesentlichen ein bürstenloser Elektromotor, der intern so konstruiert ist, dass er Drehbewegungen in genau definierten Schritten ausführt, typischerweise mit einem Winkel von 1,8 Grad oder weniger. Dabei sind bürstenloser Elektromotoren Drehstrommotoren. Als Innenläufer

sind die Wicklungen des Stators außerhalb im festen Gehäuse, während der Rotor sich im inneren dreht. Die Energieversorgung erfolgt über Gleichspannung, die in einer festgelegten Reihenfolge auf die Spulen des Motors geschaltet wird, um die Bewegung zu steuern [vgl. Ele23]. Schrittmotoren finden eine hohe Anwendung in der Steuerungstechnik, was einer der Gründe ist, warum dieser Motor in diesem Projekt betrachtet wird. Durch ihre bürstenlose Bauweise verursachen sie keine Störungen durch Bürstenfeuer [vgl. Ele23], was bei einem Holzklavier durchaus sinnvoll ist.

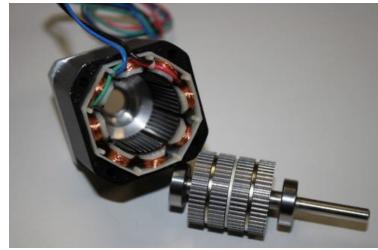


Abbildung 3.10: Schrittmotor

Quelle: [Ele23]

Linearer Servomotor Im Gegensatz zum Schrittmotor, wird die elektrische Leistung hier in eine lineare Bewegung umgewandelt. Er basiert auf der Verwendung permanentmagnetischer Felder. Im Wesentlichen verhält sich ein linearer Servomotor ähnlich wie ein rotierender Servomotor, jedoch ist er in seiner Form gestreckt und linear ausgelegt [vgl. Kol21].



Abbildung 3.11: Linearer Servomotor

Quelle: [Rei24]

Im wesentlichen besteht der Motor aus 5 verschiedenen Komponenten [vgl. Kol21]:

1. Spule/Kolben: Für die eigentliche Umsetzung von Energie in mechanische Leistung zuständig

2. magnetische Welle (durch Rotor): Schwingung aus elektrischen und magnetischen Feldern, die sich räumlich ausbreitet
3. Rückführsystem: Meldet relevante Parameter des Zustands an den Motor zurück
4. Linearlager und- führung: Ermöglichen Bewegung entlang einer geraden Achse
5. Servoverstärker/Steuerungseinheit: Schnittstelle Steuerung und Motor

Diese Elemente ermöglichen eine direkte lineare Bewegung, wobei der Servoregler den Motor mit Strom versorgt und die Rückmeldungen mit den Sollparametern vergleicht, um die Leistung zu optimieren.

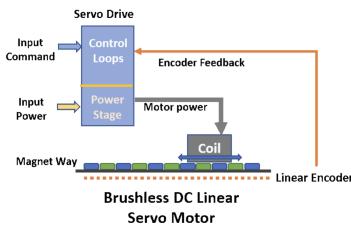


Abbildung 3.12: Aufbau eines Linearen Servomotors

Quelle: [Kol21]

Für eine präzise Bewegungssteuerung werden integrierte Servoregelkreise eingesetzt, die Strom-, Geschwindigkeits- und Positionsregelung umfassen. Der Stromkreis reguliert den Strom, um die Kraft für Beschleunigung oder Schub zu liefern, während die Geschwindigkeitsschleife die Geschwindigkeit proportional zur Spannung steuert. Die Positionsregelschleife übernimmt den Sollwert und passt die Bewegung entsprechend an [vgl. Kol21].

Hubmagnet Hubmagnete sind ebenfalls eine Form von Elektromotoren [vgl. SL15, Kapitel 1-2].

Hubmagnete verwenden eine lange Drahtschleife, die um einen metallischen Kern (Plunger) gewickelt ist, und erzeugen ein Magnetfeld, um die lineare Bewegung des Plungers herbeizuführen, wenn ein elektrischer Strom durch die Drahtspule fließt. Lineare Solenoids können einseitig oder bidirektional (Drücken und Ziehen) sein und haben oft eine Rückholfeder, um den Plunger in die Ausgangsposition zurückzubringen [vgl. SL15, Kapitel 1-2].

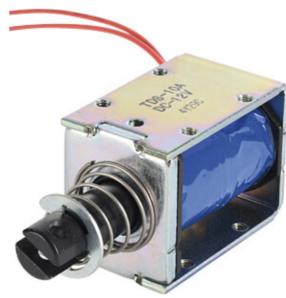


Abbildung 3.13: Hubmagnet

Quelle: [Rei24]

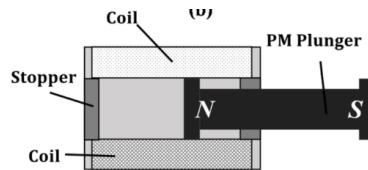


Abbildung 3.14: Hubmagnet Aufbau

Quelle: [SL15]

Entscheidungsfindung Prinzipiell kann jeder der genannten Aktuatoren für das Projekt verwendet werden. Die Entscheidung fiel aufgrund mehrerer Vorteile allerdings auf den Hubmagneten:

1. **Schrittmotor:** Im Laufe der Arbeit wurde sich gegen eine rotierende Bewegung des Aktuators entschieden, da mehrere Umleitungen der Seile dann notwendig wären, um die Tasten gerade nach unten ziehen zu können. Außerdem bringt es mit den in Kapitel 2 definierten Anforderungen keinen Mehrwert, die Bewegung des Aktuators in „Schritte“ aufzuteilen. Der Hauptgrund der Betrachtung dieses Aktuators lag darin, dass das Projekt dann insoweit erweitert werden könnte, dass auch am Klavier gespielte Stücke gespeichert und später automatisch wiedergegeben werden könnten, da die Position von Schrittmotoren einfach gespeichert und ausgelesen werden kann.
2. **Linearer Servomotor:** Prinzipiell gibt es drei Gründe warum die Entscheidung nicht auf den linearen Servomotor fiel. Der Hauptgrund liegt im Preis, der sehr viel höher ist als bei Hubmagneten, da die Steuerungssysteme in den Motoren komplexer sind. Aus dem gleichen Grund haben Hubmagnete in der Regel eine höhere Reaktionsgeschwindigkeit, was beim Spielen von Musikstücken eine große Rolle spielt. Außerdem

ist die Montage von Hubmagneten - insbesondere wenn es an Platz mangelt - einfacher, da diese tendenziell kleiner sind.

3. **Hubmagnete:** Hubmagnete³ bieten Vorteile wie eine schnelle Reaktionsgeschwindigkeit, einfache Steuerung und geringen Kosten im Vergleich zu den anderen Aktuatoren. Die verwendeten Hubmagnete können eine Kraft von 25N ausüben, was dazu genügt, eine Taste zu spielen und gleichzeitig Platz für dynamische Anpassungen von Lautstärke und Tastanschlag lassen.

3.2.6 Schaltplan

Olivier Stenzel, Jakob Kautz

Der Schaltplan besteht im Großteil aus den bereits aufgeführten Komponenten:

1. Arduino
2. Schieberegister
3. Transistoren
4. Hubmagneten

In diesem Kapitel wird erläutert, wie diese Komponenten miteinander verbunden werden um eine Funktionsfähige Schaltung zu erstellen.

Arduino

Im Zentrum der Schaltung steht der μ C (hier: Arduino Uno R3). Dieser erhält Daten und Strom über den integrierten USB-Anschluss, welcher mit dem Computer verbunden wird. Da der Arduino nur eine begrenzte Anzahl von PWM-fähigen Ausgängen bereitstellt, werden Schieberegister (hier: 74HC595) zur Vermehrung der Ausgänge verwendet (siehe Abschnitt 3.2.3). Mit jedem „in Reihe“ geschaltetem Schieberegister kann die Anzahl PWM-fähiger Ausgänge um 8 erweitert werden.

³siehe [Hes] für genauere Informationen

Schieberegister

Der Arduino wird an fünf Stellen mit dem ersten Schieberegister verbunden:

Arduinoport D2 <-> Serial (SER) Input

Über diese Verbindung werden serielle Daten hier bitweise in das Register geschoben.

Arduinoport D3 <-> SHCP (Shift Register Clock Input)

Dieser Pin wird verwendet, um den Takt für das Verschieben der Daten innerhalb des Schieberegisters anzulegen. Bei jedem Taktimpuls auf diesem Pin wird das Bit am seriellen Dateneingang in das Register verschoben. Das bedeutet, dass bei jeder steigenden Flanke des Taktsignals das Datenbit, das am Eingang anliegt, in das Schieberegister übernommen und alle vorhandenen Daten um eine Position verschoben werden.

Arduinoport D4 <-> STCP (Storage Register Clock Input)

Nachdem die Daten in das Schieberegister eingelesen wurden, wird dieser Pin verwendet, um die im Schieberegister vorhandenen Daten in das Ausgangsregister zu übertragen. Ein Taktimpuls auf diesem Pin bewirkt, dass die Daten vom Schieberegister ins Ausgangsregister übernommen werden, sodass alle Ausgänge gleichzeitig aktualisiert werden. Das ist besonders relevant, da sonst unter Umständen alle Ausgänge von einer Änderung im letzten Schieberegister betroffen wären.

Arduino GND <-> Ground, Output Enable (OE)

Der OE-Pin wird genutzt, um die Ausgänge des Schieberegisters global zu aktivieren oder zu deaktivieren, ohne die Daten selbst zu beeinflussen. Da das Schieberegister zu keiner Zeit deaktiviert sein soll, wird dieser Pin dauerhaft mit dem GND-Pin verbunden.

Arduino Vcc 5V <-> Vcc, \overline{SRCLR} (Reset)

Um das Schiebregister mit den benötigten 5V zu betreiben, wird der entsprechende Pin mit dem 5V Output des Arduino verbunden. Zusätzlich wird der SRCLR Port des Schieberegisters, welcher ein Reset ermöglicht dauerhaft mit 5V verbunden.

Jedes weitere Schieberegister greift die oben genannten Signale ab. Der einzige Unterschied befindet sich am Serial (SER) Input Port. Das Schieberegister an Position $i + 1$ wird mit dem seriellen Output des Schieberegisters an Position i verbunden ($\forall i = 0, \dots, 10$).

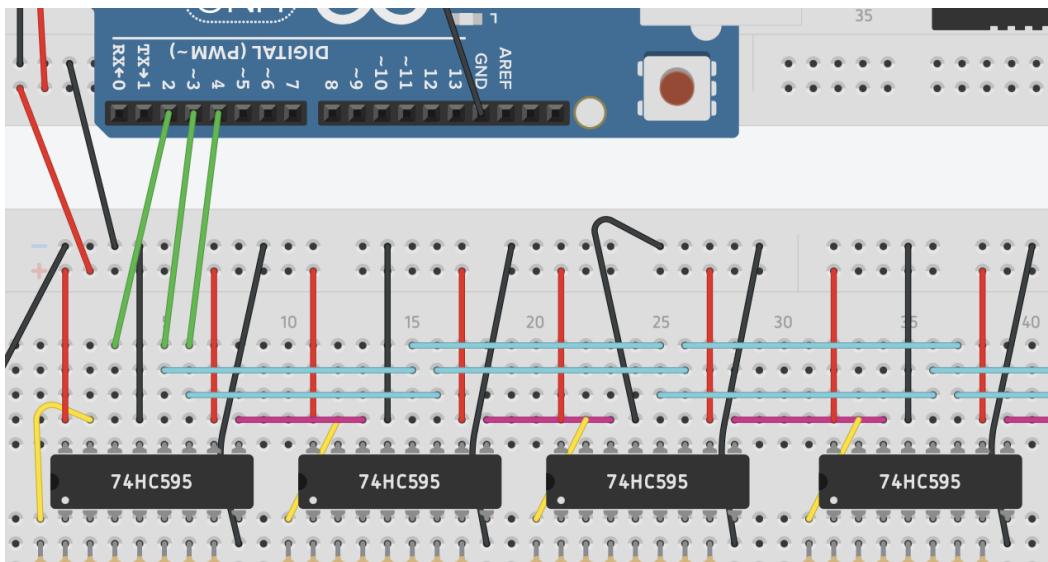


Abbildung 3.15: Schaltung der Schieberegister

Transistoren

Die Hubmagnete werden jeweils mit 24V und mit bis zu 400mA betrieben. Um einen hohen Stromfluss zu steuern, können Transistoren verwendet werden. Für hohe Spannungen und schnelle Schaltvorgänge eignen sich besonders MOSFETs. Im Folgenden werden speziell n-MOSFETs verwendet, die mit einem Signal zwischen 0V (nicht leitend) und +5V (voll leitend) angesteuert werden können.

Der folgende Aufbau ist für die insgesamt 88 Ausgänge der 11 Schieberegister identisch, da jeder Ausgang für die Ansteuerung genau eines Motors zuständig ist.

Der GATE-Pin des MOSFETs erhält das Signal, das die „Durchlässigkeit“ steuert, aus einem der Outputs des Schieberegisters. Der SOURCE-Pin wird mit Ground des gesamten Systems verbunden. Der DRAIN-Pin wird direkt mit dem entsprechenden Kontakt am Hubmagneten verbunden.

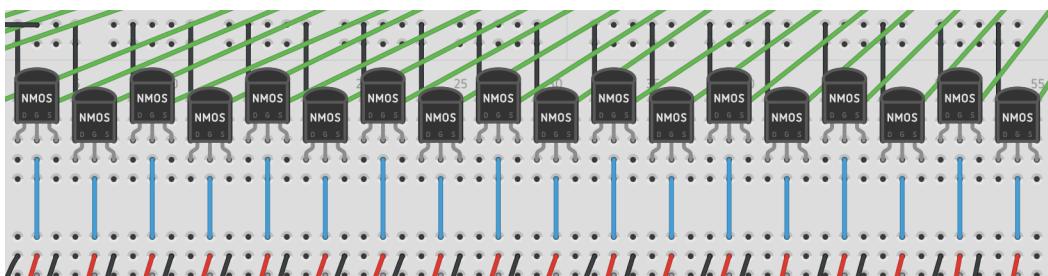


Abbildung 3.16: Schaltung der Mosfets

Hubmagnet

Um den Stromkreis zu schließen wird der andere Kontakt des Hubmagnets mit dem +24V verbunden. Bei Hubmagneten ist es in der Regel egal, welcher Anschluss an Plus und welcher an Minus angeschlossen wird [vgl. SL15].

Testen

Um die Fehlersuche zu erleichtern, werden LEDs in den Schaltplan mit eingebaut. Diese werden jeweils mit einem entsprechenden $1k\Omega$ Widerstand parallel zu den Motoren angeschlossen. So kann anhand der Helligkeit der LED die Intensität abgelesen werden, mit der eine Taste gespielt wird.

Für insgesamt 4 Hubmagnete - wobei in dem Schema Motoren gewählt wurden, da das Programm keine Hubmagnete als Komponenten zur Verfügung stellt - und 2 Schieberegister wurde der Schaltplan in Abbildung 3.17 dargestellt.

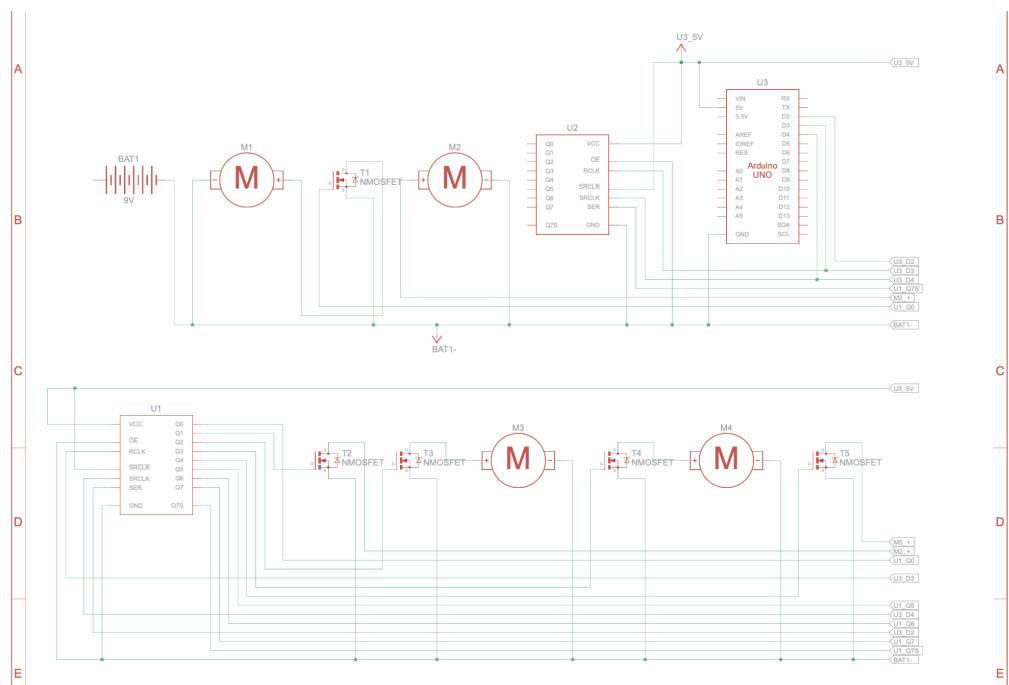


Abbildung 3.17: Beispielhaftes Schema des Schaltplans

Die gesamte Schaltung mit allen 88 Hubmagneten wird wie folgt aussehen:

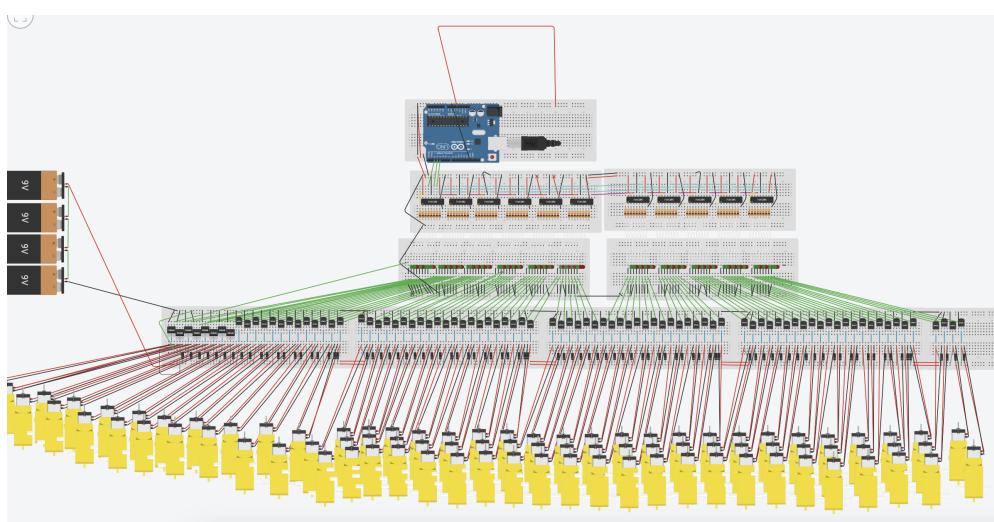


Abbildung 3.18: Überblick Schaltplan

4 Umsetzung - Hardware

Jakob Kautz, Olivier Stenzel

Nachdem die Planungsphase abgeschlossen wurde, mussten die Überlegungen umgesetzt werden. Im Rahmen dieser Arbeit wurde ein Prototyp gebaut, mit dem Ziel möglichst viele der 88 Tasten anzuspielen. Aufgrund der begrenzten Zeit umfasst der in dieser Arbeit beschriebene Prototyp nur 8 spielbare Tasten, wäre allerdings ohne weitere Überlegungen auf alle restlichen erweiterbar. Zusätzlich wurde die Elektronik mit weiteren 32 LEDs so erweitert, sodass das Anspielen von 40 Tasten simuliert werden kann.

4.1 Materialien

Jakob Kautz

Die Bauteile, die für die Umsetzung des spezifizierten Schaltplans benötigt und besorgt wurden, sind in Tabelle 4.1 aufgelistet.

Aufgrund der relativ hohen Kosten für eine Studienarbeit, wurde bei einer der betreuenden Firmen angefragt, ob diese die Kosten für das Projekt übernehmen würde. Damit dies möglich war, wurde ein Kostenvoranschlag gestellt, in welchem die benötigten Materialien mit den geschätzten Kosten aufgeführt wurden.

Dieser Kostenvoranschlag ist in Tabelle 4.2 wiedergegeben.

Der Kostenanschlag erwies sich im Laufe des Projektes als (teils) unrealistisch. Dies lag insbesondere an einer Anforderung der Firma. Die Schätzung der Kosten basierte auf Anbietern, bei welchen die Materialien möglichst günstig zu kaufen sind. Aufgrund von Firmenregelungen mussten diese allerdings alle bei Conrad¹ oder Reichelt² gekauft werden. Diese Anbieter verkaufen die Materialien für sehr viel mehr Geld. Die Kostenschätzung wäre also passend gewesen, wenn die Anbieter frei wählbar gewesen wären.

¹<https://www.conrad.de/>

²<https://www.reichelt.de/>

Bauteil	Anzahl	Begründung
Hubmagnete	88	jede Taste braucht einen Hubmagneten um angespielt zu werden (Für den Prototypen werden nur 8 verwendet)
Stromversorgung	1	Externe Stromversorgung für die Aktuatoren, da diese mehr als die 5V Vcc des Arduinos brauchen
Arduino	1	Kommunikation
Breadboard	10	Erweiterung der Ausgänge
Schaltplatine	5	Elektronik für die Aktuatoren
Schieberegister	11	Weitergabe Signal
Kabel (10cm)	352stck (bzw. 9 · 40 in Packs)	Verbindungen in der Schaltung mit Schätzung 4 Kabel pro Hubmagnet
Kabel (20cm)	176stck (bzw. 5 · 40 in Packs)	Verbindungen zu den Hubmagneten
LEDs	88	Tests
1kOhm Widerstände	88	Sicherheit
MOSFET	88	Steuerung Strom
Feste Anschlussblöcke	88	Anschluss von Schaltplatine zu Hubmagnet
Angelschnur (1m)	88	Verbindung Hubmagnet und Taste

Tabelle 4.1: Übersicht der Bauteile

Komponente	Angesetzte Kosten in €	Tatsächliche Kosten in €
Klavier	200€	180€
Arduino	30€	24€
Hubmagnete	5€ -> 440€	13,91€ -> 1224.08€
Stromversorgung	50€	kostenlos von der Uni bereitgestellt
Breadboard	3€ -> 30€	3.43€ -> 34.3€
Schaltplatine	2€ -> 10€	2€ -> 10€
Schieberegister	0.3€ 3.3€->	8€, da nur in Paket erhältlich
Kabel insgesamt	0.12€ -> 67€	0.12€ -> 67€
LEDs	Bei der Kostenschätzung nicht mit aufgenommen	0.10€ -> 8.8€
1kOhm Widerstände	0.10€ -> 8.8€	0.10€ -> 8.8€
MOSFET	0.9€ -> 79.2€	0.66€ -> 58,08€
Feste Anschlussblöcke	Bei der Kostenschätzung nicht mit aufgenommen	10€
Verbindung gesamt	100€	120€

Tabelle 4.2: Übersicht der Kosten

4.2 Prototypenbau

Olivier Stenzel

Ursprünglich sollte der Aufbau des in Kapitel 3.2.6 spezifizierten Schaltplans mit Steckbrettern und Jumperkabeln umgesetzt werden. Während der Umsetzung ist allerdings aufgefallen, dass die gewählten Platinen den benötigten Stromfluss nicht aushalten.

Jeder aktiver Aktuator - also jede gespielte Taste - zieht einen Strom von 0.7A. Wie in Anforderung A6 spezifiziert, sollen mindestens 10 Tasten gleichzeitig gespielt werden können, was bedeutet, dass der Aufbau mindestens 7.0A Stromfluss problemlos ausstehen muss.

Aus diesem Grund wurde die gesamte Schaltung, die nach dem Schieberegister kommt, auf einer Lochrasterplatine fest gelötet. Hierfür wurde ein 3mm starker Draht für die Stromversorgung verwendet.

Es wurde bei den Kabeln und Steckplatten damit gerechnet, dass maximal 20 Aktuatoren gleichzeitig gespielt werden. Da ein Großteil der Stücke für einzelne Pianist:innen geschrieben wurde, kann davon ausgegangen werden, dass generell nicht mehr als 10 Tasten gleichzeitig gespielt werden müssen - tendenziell weniger. Die restlichen 10 Tastendrücke

die dazu gerechnet wurden, sind lediglich eine Versicherung.

4.2.1 Verbindung Tasten und Aktuatoren

Wie in Kapitel 3.1.2 beschrieben, wurde sich für ein Ziehen der Tasten entschieden. Dieses Ziehen wird technisch mittels Hubmagneten umgesetzt.

Dieses Kapitel beschreibt wie und wo die Hubmagnete am Klavier und den Tasten befestigt wurden. Zusätzlich wird der Aufbau im Verlauf des Kapitels bezüglich Reibung und Genauigkeit beim Ansteuern der Tasten weiter verbessert.

Die Grundidee ist, mittels einer Schnur, oder Ähnlichem die Tasten mit den Hubmagneten zu verbinden. Diese sollen dann an der Schnur, und somit an der Taste ziehen, welche wiederum den Hammer an die Saite schlägt.

Die Schnur wird mittels eines waagerechten Loches in der Taste an dieser befestigt. Die Position an der Taste kann nicht frei gewählt werden. Es muss nämlich darauf geachtet werden, dass das Loch möglichst am Ende der Taste angebracht wird, um einen größeren Hebeleffekt zu erzeugen. Zusätzlich ist es wichtig, dass das Tastenbrett unter der Taste an der Position gut durchzubohren ist. Eine ästhetische und nicht notwendige Anforderung wäre, dass das Seil beim spielen nicht von oben gesehen wird. In Abbildung 4.1 wird gezeigt, wie die Montage umgesetzt wurde. Mit rot ist das gebohrte Loch gekennzeichnet, durch das ein Seil (in grün dargestellt) geführt wird.

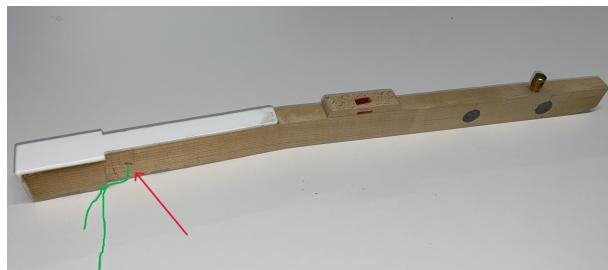


Abbildung 4.1: Tastenbohrung

Anschließend wird jede Schnur einzeln in den Fußraum geführt. Dafür werden senkrechte Löcher (rot in Abbildung 4.2) in den Klaviaturbalken (bzw. das Tastenbrett), worauf die Tasten liegen, gebohrt.



Abbildung 4.2: Bohrung durch das Tastenbrett

Anordnung der Aktuatoren

Die Idee ist, die Hubmagnete parallel zur Klaviatur zu befestigen, um die Tasten möglichst senkrecht nach unten ziehen zu können. Hierfür müssen folgende Aspekte berücksichtigt werden:

1. Breite der Hubmagnete
2. Hitzeentwicklung
3. Seilführung
4. Stabilität
5. Modularität

Ein Hubmagnet hat die Maße 2,5cm x 6cm. Das Tastenbrett für 88 Tasten ist allerdings nur 140cm breit, wodurch pro Tastenansteuerung nur ca. 1,6cm zur Verfügung stehen. Wenn zwei Hubmagnetreihen übereinander gebildet werden, hat jede Tastenansteuerung 3,20cm Platz.

Durch die 7mm Abstand ist eine Wärmeabfuhr über die Luft in geringem Ausmaß möglich. Alle Seile werden pro Hubmagnetreihe, parallel zwischen Tasten und Hubmagneten verbunden.

Die Hubmagnete direkt unter dem Tastenbrett zu befestigen wäre eine triviale Lösung, würde allerdings auf Kosten der Spielbarkeit (siehe Anforderung A1 in Kapitel 2) gehen. Um also die Beinfreiheit der Pianist:in zu gewährleisten werden die Hubmagnete am Korpus des Klaviers (untere Frontplatte) befestigt.

Um die Austauschbarkeit der Komponenten zu verbessern, werden die Hubmagnete nicht direkt an der unteren Frontplatte, sondern auf zwei Pressspanplatten (ca. 70cm x 25cm) befestigt, welche an vier Punkten mit dem Klavier verschraubt werden können (siehe Abbildung 4.3). In diesem Prototyp wird die Verschraubung an das Klavier nicht genutzt, um an Flexibilität zu gewinnen.



Abbildung 4.3: Befestigung der Hubmagnete

Seilführung

Würden die Seile von den Tasten direkt zu den Hubmagneten geführt werden, gäb es deutliche Hub-Verluste, sodass unter Umständen die Tasten nicht mehr ausreichend stark betätigt werden könnten, um einen Ton zu erzeugen.

In den folgenden Abbildungen (4.4 bis 4.6) ist die seitliche Ansicht des Klaviers gezeichnet. Dabei repräsentieren der schwarze bzw. weiße Block jeweils eine schwarze bzw. weiße Taste. Der grüne und pinke Strich stehen für die Seile, die den Hubmagneten mit dem Loch in der Taste verbinden. Unten links sind die zwei übereinanderliegenden Solenoid-Reihen abgebildet. Der schwarze Strich repräsentiert den Stab im Hubmagneten.

Abbildung 4.4 zeigt den intuitiven Aufbau mit ausgeschaltetem Solenoid und entsprechend entspanntem Seil und. Abbildung 4.5 zeigt den selben Aufbau mit dem Unterschied, dass der Solenoid hier angezogen ist.

Mit diesem intuitiven Aufbau funktioniert das Betätigen der Tasten nicht, da diese nicht weit genug heruntergezogen werden können. Um den gesamten Hub des Magneten zu Nutzen und an die Taste weiterzugeben, müssen Umlenkungen eingebaut werden. Mittels dieser Umlenkung, die mit einem PVC-Rohr umgesetzt werden kann, werden die Seile so geführt, das sie senkrecht auf die Hubmagnete fallen. Somit wird die Tiefe, mit der die

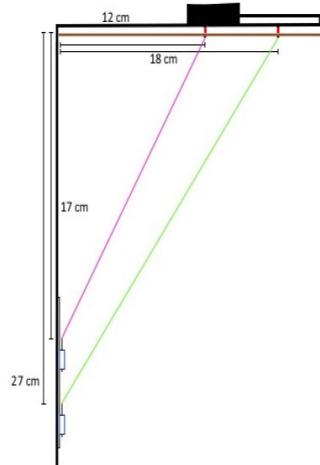


Abbildung 4.4: Taste locker ohne Umlenkung

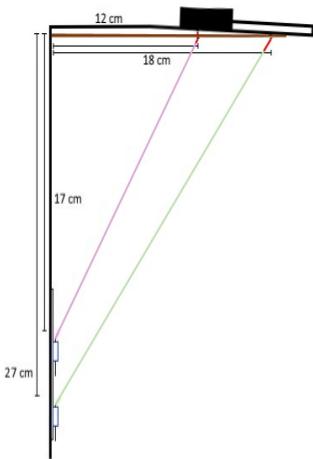


Abbildung 4.5: Taste gezogen ohne Umlenkung

Tasten gedrückt werden, wieder auf annähernd 1cm erhöht, was dem Hub des Magneten entspricht..

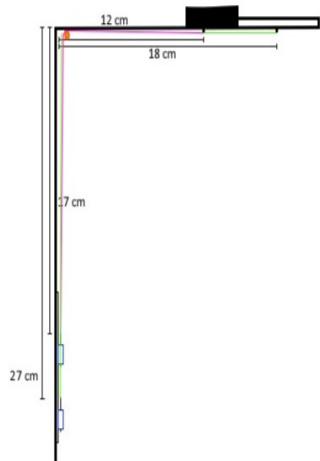


Abbildung 4.6: Taste mit Umlenkung

Dies kann auch mathematisch bewiesen werden:

geg.:

Distanz zwischen dem Tastenbrett und der unteren Reihe Hubmagnete $h = 27\text{cm}$

Distanz zwischen der unteren Frontplatte und dem zum Loch im Tastenbrett $t = 18\text{cm}$

Gesucht ist die Differenz der Längen zwischen dem Tastenbrett und dem Hubmagneten, wenn die Tasten (nicht-)gedrückt sind. Diese sind in Zeichnung „Taste locker ohne Umlenkung“ und „Taste gezogen ohne Umlenkung“ rot gekennzeichnet.

$ht_{entspannt}$ beschreibt die Seillänge zum Tastenbrett im entspannten Zustand

$$ht_{entspannt} = \sqrt{h^2 + t^2} = 32.45\text{cm}$$

$$ht_{gespannt} = \sqrt{(h+1)^2 + t^2} = 33.29\text{cm}$$

Die Differenz zwischen $ht_{gespannt}$ und $ht_{entspannt}$ (bzw. ht_2) beschreibt die Tiefe, die eine Taste gedrückt werden kann, wenn der Hub des Magnetens 1cm beträgt.

Diese beträgt also ohne Umlenkung nur 0.84cm. Mit der Umlenkung (siehe Abbildung 4.6) kann diese wieder nahezu auf den vollen Hub angehoben werden.

Um die Intensität des Tastendrucks besser steuern zu können und um die Haltbarkeit des Materials zu verlängern, sollte die Reibung am Seil möglichst gering gehalten werden. Dazu können mittels Rohren, welche waagerecht unter dem Klaviaturbalken entlang der Löcher montiert werden können (siehe Abbildung 4.7), die Seile umgelenkt und entlang der Verkleidung zur unteren Frontplatte des Klaviers geführt werden. Das hat den positiven Nebeneffekt, dass die Pianist:innen nicht durch Platzmangel im Beinbereich eingeschränkt sind.



Abbildung 4.7: Fußraum des Klaviers

Auswahl des Seils

Wie eben beschrieben wird das Ziehen durch ein Seil ermöglicht. Dafür wird ein leichtes, formbares, unelastisches und dehnungsresistenteres Material benötigt, welches die Hubmagnete mit den Tasten verbindet. Je leichter das Material ist, desto weniger geht die Genauigkeit der Kraftübertragung zwischen den Magneten und den Tasten verloren. Durch die Formbarkeit kann der Knoten sehr eng an der Taste geschnürt werden, wodurch das Ansprechverhalten schneller erfolgen kann. Da die Anschläge ruckartige Bewegungen sind, ist es wichtig, ein unelastisches Seil zu verwenden. Um häufiges Nachspannen oder Austauschen des Seils zu vermeiden, sollte es so dehnungsresistent wie möglich sein.

Nähgarn Als Erstes wurde Nähgarn, welches zur Hand war, getestet. Auch doppelt verlegt hielt es der ruckartigen Ziehbewegung (mit 25N) des Hubmagnetens nicht stand. Vier Fäden funktionierten zu Beginn gut, leierten allerdings schnell aus.

Nylonsaiten Als Nächstes wurde die g-Saite einer Gitarre verwendet. Durch den höheren Durchmesser und das stärkere Material riss und leierte die Saite nicht aus. Da die Saite kaum Flexibilität liefert, war die Befestigung an der Taste jedoch äußerst schwierig. Beim Ziehen des Magnetens wurde erst die lockere Schlaufe an der Taste gestreckt, wodurch nicht der vollständige Hub des Magnetens auf die Taste übertragen wurde.

Angelschnur Nach einer kurzen Auseinandersetzung mit Angelzubehör konnte schnell eine geeigneter Schnur gefunden werden. Genauer handelt es sich hier um eine geflochtene Schnur aus Polyethylen. Mit einem Durchmesser von 1.6mm ist sie nicht nur sehr dünn und flexibel, sondern kann auch bis zu 7kg standhalten. Nach ausgiebigem Testen ist eindeutig, dass die Angelschnur die Anforderungen erfüllt. Damit ist nun auch das Spielen des Forellenquintetts von Schubert ein Leichtes.

4.2.2 Klangdämpfung der Aktuatoren

Die Hubmagnete machen beim Anschlagen laute „Klack“ Geräusche, welche von der Melodie des Klaviers ablenken. Genauer handelt es sich um den Metall-Anker, der gegen das Ende des Metall-Gehäuses stößt. Auch hierfür gab es mehrere Ideen und Tests, um das Geräusch zu dämpfen:

Isolierfolie Die erste Überlegung war die Auskleidung des Innenraums der Hubmagnete mit Isolierfolie. Diese Idee wurde wieder verworfen, da das Wissen über die Hitzeentwicklung zu diesem Zeitpunkt noch zu gering war, um sicherzustellen, dass die Isolierung ihr standhält.

Gummi-Stopper Die nächste Idee war das Limitieren des Schlags durch Dichtungsringe aus Gummi zwischen Anker und dem äußeren Gehäuse. Da für die Befestigung keine zufriedenstellende Lösung gefunden wurde, wurde auch diese Idee verworfen.

Schaumstoff Wie im Abbildung 4.8 zu sehen ist, wurde die „Gummi-Stopper“-Idee durch den Einsatz von Schaumstoff leicht modifiziert.

- + Klopferäusch wird vollständig verhindert
- Durch den Schaumstoff wird 1mm des Hubs nicht verwendet, weshalb nur noch 9mm übrig bleiben
- Der Aufbau ist optisch nicht besonders ansprechend



Abbildung 4.8: Hubmagnet: Dämpfung mit Schaumstoff

Seil (2mm Durchmesser) um den Anker Um das „Problem“ der schlechten Ästhetik beim Schaumstoff zu beseitigen, wurde ein 2mm dickes Seil im Inneren des Gehäuses um den Anker gewickelt.

- + Auch hier konnte das Klopfen vollständig beseitigt werden
- + Von außen ist keine Veränderung zu sehen
- + Das Seil scheint nach ausführlichem Testen der Hitzeentwicklung gut Stand zu halten
- Durch die Dicke des Seils, werden 2mm des Hubs nicht verwendet, weshalb nur noch 8mm übrig bleiben.



Abbildung 4.9: Hubmagnet: Dämpfung mit Seil

4.2.3 Klavieranbau

Im Rahmen des Protoypen wurde die Elektrik nicht fest am Klavier verschraubt. Das Brett mit den Hubmagneten wurde stattdessen nur an das Klavier angelehnt, während die Angelschnüre die Verbindung zu den Tasten herstellen. Gespannt werden die Seile durch eine Schrauben-Mutter Konstruktion (siehe auch Abschnitt 4.2.1).

4.2.4 Ergebnisse des Prototypen

Jakob Kautz

Der Prototyp des selbstspielenden Klaviers zeigt erfolgreiche Ergebnisse in mehreren Bereichen. Die Implementierung von 8 Tasten funktioniert zuverlässig, wodurch das Klavier in der Lage ist, Töne korrekt wiederzugeben. Seitens der fertig gelöteten Schaltung könnten 40 Tasten angespielt werden, allerdings haben die Tests sich auf 8 beschränkt. Ein Großteil der in Kapitel 2 definierten Anforderungen konnten von dem Prototypen erfüllt werden. Zusätzlich erweist sich die Klangdämpfung, dort wo sie eingesetzt wird, als effektiv. Ein großer Teil, der fehlt, ist ein Sicherheitskonzept, weshalb es nicht unbeaufsichtigt betrieben werden sollte. Auch fehlt die Fertigstellung der Schaltung für die zweite Hälfte der Tasten.

Für die mögliche Fertigstellung des Prototypen, muss der Zeitaufwand betrachtet werden. Von der Umsetzung her ist nur die Hälfte des Klaviers anspielbar. Speziell wären die folgenden Aufgaben noch zu tun.

- Löten der Schaltung für 48 Solenoids

- Löten der Anschlussblöcke: 76 Stück
- Isolierung von 48 Hubmagneten
- Fertigstellung des Brettes für die Befestigung der Hubmagnete (Bohren)
- Verbindung zwischen 76 Hubmagneten und Tasten
- Klangdämpfung für 86 Hubmagneten

Die Zeiteinschätzung kann gut auf den bisherigen Arbeiten basieren. Das Löten und isolieren der Schaltung für 40 Hubmagnete, benötigte etwa 40 Stunden. Die Befestigung der Magnete und die Verbindung mit den Tasten wird auf mindestens zwei Stunden Arbeit herauslaufen. Zusätzlich wird mindestens eine Stunde (tendenziell eher 2) für die Klangdämpfung benötigt werden.

Damit beläuft sich die Fertigstellung des Prototypen auf mindestens weitere 44 Stunden. Dazu kommt, dass der Prototyp noch ausführlich getestet werden muss und gegebenenfalls Fehler die beim Löten o.ä. aufkommen behoben werden müssen.

Der Prototyp wird aus diesem Grund nicht komplett fertiggestellt.

Während der Entwicklung des Prototypen traten außerdem unerwartete Herausforderungen auf, die den Projektverlauf beeinflussten. Eine der ersten Hürden war die Stromstärke, die anders als ursprünglich angenommen ausfiel. Anfangs wurde mit einem Verbrauch von 0,4 Ampere pro Hubmagnet kalkuliert, doch letztendlich betrug die tatsächliche Stromstärke 0,7 Ampere pro Hubmagnet. Die zu Beginn vorgesehenen Breadboards können jedoch nur maximal 0,5 Ampere pro Stecker verarbeiten. Somit musste der Teil der Schaltung, der mehr Strom benötigte - alles nach den Dioden, da ab diesem Punkt ein externer Stromanschluss für den Betrieb der Hubmagnete hinzukam - auf Steckplatten gelötet werden. Dies führte zu Verzögerungen im Projekt, da auf die Platinen gewartet werden musste, zusätzlicher Arbeitsbelastung, da nun die gesamte Schaltung gelötet wurde, und höheren Kosten, da zusätzlich zu den bereits bestellten Breadboards Platinen gekauft wurden. Die Breadboards konnten jedoch in der Ausbildungsabteilung der Firma weiterverwendet werden, weshalb sie nicht in die Kostenübersicht aufgenommen wurden.

Eine weitere unerwartete Herausforderung im Zusammenhang mit der Stromstärke bestand darin, dass die verwendeten Jumper-Kabel nicht dick genug waren, um den Stromfluss der externen Stromversorgung zu bewältigen. Für eine stabilere Verbindung waren daher dickere Kabel für den Plus- und Minuspol erforderlich. Es wurden keine neuen Kabel bestellen, da

die dickeren Kabel für den externen Stromanschluss von der Uni bereitgestellt wurden, während die Jumper-Kabel für den Rest der Schaltung ausreichten.

Die letzte unerwartete Herausforderung ergab sich aus den unterschiedlich großen Bohrlöchern in den Hubmagneten, die zur Befestigung an dem Brett dienen sollten. Dies erforderte den Kauf verschieden großer Schrauben und teilweise auch Muttern, um sicherzustellen, dass sich die Magnete nicht lösen. Obwohl diese Lösung funktioniert, ist es ratsam, die Schrauben gelegentlich nachzuziehen, um zu verhindern, dass sich während des Spielens ein Hubmagnet löst.

5 Konzeption - Software

Valentin Richter

Wie in Kapitel 2 beschrieben, soll eine Desktop-Anwendung für die Nutzer-Interaktion mit dem „Piano Player“ erstellt werden. Die Anwendung soll im Folgenden auch als „Self-Applying Musician (SAM)“ bezeichnet werden. Die fundamentale Aufgabe der Anwendung liegt im Verwalten und Spielen von Musikstücken auf dem selbstspielenden Klavier. Zum Anspielen des Klaviers wird ein Mikrocontroller (μ C) benötigt (siehe Kapitel 3.2.1). Da die Auswahl des Musikstücks dynamisch während der Laufzeit über Self-Applying Musician (SAM) gehen soll, muss SAM mit dem μ C kommunizieren können. In diesem Kapitel soll die Software beider Komponenten und deren Kommunikation konzipiert werden.

Abbildung 5.1 zeigt diese unterschiedlichen Komponenten, sowie den Datenfluss zwischen diesen. Nutzer:innen werden in der Abbildung auch dargestellt, um eindeutig zu machen, dass Nutzer-Eingaben nur an SAM gehen. Der μ C erhält Daten zum abzuspielenden Musikstück von SAM und schickt darauf basierend in regelmäßigen Intervallen PWM-Signale an den „Piano Player“.

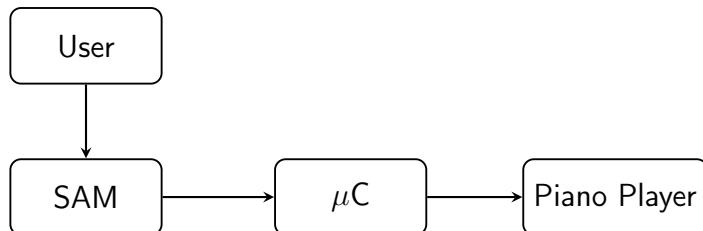


Abbildung 5.1: Komponenten-Diagramm mit Datenfluss

Da die Hardware-Komponente in vorherigen Kapiteln bereits behandelt wurde, soll hier nur die Architektur der Desktop-Anwendung (in Abschnitt 5.4) und des μ Cs (in Abschnitt 5.2) erarbeitet werden. Weiterhin soll noch die Kommunikation zwischen diesen beiden Komponenten in Abschnitt 5.3 betrachtet und spezifiziert werden. Ein Ziel der Kommunikations-Spezifikation ist eine leichte Anpassbarkeit der einzelnen Komponenten, solange sie sich weiterhin an das selbe Kommunikations-Protokoll halten. Damit soll der Anforderung A8 gerecht werden.

Da die Konzeption aller Teile jedoch vom Format, in dem Musikstücke auf dem μ C gespeichert werden, abhängen, soll dieses zuerst entworfen werden (siehe Abschnitt 5.1). Da der μ C eine möglichst hohe Geschwindigkeit für die Ansteuerung der Motoren erbringen soll, muss das Format eine performante Nutzung ermöglichen. Auch muss das Format möglichst wenig Speicherplatz aufbrauchen, damit möglichst viele Daten auf einmal auf dem μ C gespeichert werden können. Je mehr Speicherplatz das Format nämlich benötigt, desto öfter muss die User Interface (UI) Daten an den μ C schicken, was aufgrund relativ hoher Latenzen in der Kommunikation nicht wünschenswert ist.

5.1 Datenformat für Musikstücke

Es gibt bereits viele Datenformate, die zum Speichern von Musikstücken verwendet werden können. Dabei lassen sich solche Formate generell in zwei fundamental verschiedenen Designs unterscheiden.

Zum Einen gibt es generelle Audioformate. Dazu gehören solche Formate wie MP3,¹ WAV² oder FLAC.³ Diese Formate speichern für jeden Zeitpunkt des Audios die Information über das Geräusch, das zu diesem Zeitpunkt abzuspielen ist.

Zum Anderen gibt es Formate, die speziell für Musikstücke entwickelt wurden. Das Vorzeige-Beispiel wäre hier das MIDI-Format. Bei diesen Formaten werden statt Geräusch-Daten Informationen über Noten, die für eine gewisse Zeit zu spielen sind, gespeichert. Das genaue Geräusch, das am Ende abgespielt werden sollte, wird dabei noch von anderen Faktoren beeinflusst. So könnten beispielsweise mehrere Noten gleichzeitig zu spielen sein. Auch könnte das Instrument von Nutzer:innen dynamisch änderbar sein.

Die zweite Art an Formaten ist im Kontext dieses Projekts zu bevorzugen, da es dem Format der Ausgabe näher kommt. Zum Anspielen eines Klaviers werden nämlich zu jedem Zeitpunkt die Information über derzeit zu spielende Tasten benötigt. Formate der zweiten Art speichern genau jene Informationen und erfordern damit keine weiteren Informations-Umwandlungen mehr.

¹formal „MPEG-1 Audio Layer III“ bzw. „MPEG-2 Audio Layer III“; siehe [ISO93] für die Spezifikation des Formats

²wird auch „WAVE“-Format genannt; siehe [Kab22] für die Spezifikation des Formats

³formal „Free Lossless Audio Codec“; siehe [BW24] für die Spezifikation des Formats

Im Gegensatz dazu sind die erstgenannten Formate speziell für digitale Lautsprecher entworfen worden und speichern entsprechend Informationen über die vom Lautsprecher zu spielenden Geräusche.

Wenn hier von Informations-Umwandlung gesprochen wird, meint dies nicht eine simple Datentransformation. Daten sind eine Enkodierung von Informationen und somit können unterschiedliche Daten die selben Informationen darstellen. Die Wandlung eines Datensatzes in eine andere Kodierung kann zwar rechenaufwendig werden, ist aber ohne Verlust von Informationen möglich. Je nach Kontext können Datentransformationen auch ausgelassen oder stark optimiert werden. Falls grundlegend andere Informationen benötigt werden, ist eine Änderung der Daten nicht nur notwendig, sondern oftmals auch vergleichsweise aufwendig.

Es ist also festzuhalten, dass die Verwendung eines Formats wie MIDI hier sinnvoll ist. Bevor MIDI selbst gewählt wird, sollten jedoch noch weitere Aspekte beachtet werden.

Eine Anforderung dieses Projekts ist eine höhere Performanz als die schnellsten menschlichen Pianist:innen zu erreichen (siehe Tabelle 2.1). Entsprechend sollte das gewählte Datenformat der Musikstücke auch möglichst schnell in das Format umgeformt werden können, das für die Ausgabe an die Aktuatoren benötigt wird. Am performantesten wäre es hierbei, eine Liste jener Daten zu speichern, die zu jedem Zeitpunkt an die Aktuatoren ausgegeben werden müssen, da dann gar keine Datentransformation mehr nötig ist.

Allerdings muss das Datenformat auch möglichst wenig Speicherplatz verbrauchen. Da der Code auf einem μ C laufen muss, ist Speicherplatz relativ begrenzt. Der μ C muss dabei nicht das gesamte Musikstück auf einmal gespeichert haben, da restliche Teile des Musikstücks auch später von SAM nachgereicht werden können. Das verringert die Wichtigkeit des Speicherplatzes jedoch nur zu Teilen, da ein solches Streamen der Daten mehr Kommunikation zwischen SAM und MC benötigt. Da die Latenzen der Kommunikation relativ hoch sein können, gefährdet dies wiederum das Ziel der Performanz.⁴

MIDI schneidet hierbei an sich sehr gut ab, da Noten sehr kompakt abgespeichert werden. Dafür wird eine Enkodierung variabler Länge verwendet [vgl. Ass96, S. 131]. Da MIDI allerdings ein generelles Format ist, das eine Vielzahl von digitalen Instrumenten und Synthesizers unterstützen möchte, beinhaltet der MIDI-Standard eine Vielzahl optionaler

⁴Während der Erstellung der Konzeption wurde experimentell herausgefunden, dass die Latenzen nicht zu vernachlässigen sind. Konkrete Zahlen wurden jedoch erst bei den Tests am Ende der Entwicklung ermittelt und finden sich in Tabelle 7.2

Metadaten [siehe VA20a, für die gesamte Liste], die für dieses Projekt irrelevant sind. Entsprechend würde nur ein Subset des MIDI-Standards zum Speichern der Musikdaten in Frage kommen.

Nichtsdestotrotz kann das Format für diesen eingeschränkten Anwendungsfall noch optimiert werden. Um diese Verbesserungen zu motivieren, soll zuerst die Enkodierung von Noten in MIDI erläutert werden.

MIDI speichert die Notenfolgen in sogenannten „Nachrichten“. Nachrichten können auch andere Daten tragen [siehe VA20b, für die gesamte Liste], hier sollen aber nur die Nachrichten zum Spielen von Tönen betrachtet werden. Jede dieser Nachrichten folgt einer relativen Zeitangabe, im Standard „delta time“ genannt. „delta time“ gibt an, wie viele Schläge der virtuellen MIDI-Uhr zwischen der letzten Nachricht und dieser vergehen sollen. Die MIDI-Uhr kann direkt in Angaben einer Wand-Uhr umgerechnet werden. Hier ist jedoch wichtig zu bemerken, dass diese MIDI-Uhr sehr viel genauer als eine Angabe in Millisekunden ist. Auch ist zu erwähnen, dass diese „delta time“ eine variable Länge hat, mindestens aber immer ein Byte verwendet [vgl. Ass96, S. 135].

Nach der „delta time“ folgt dann genau ein Byte für den „Ereignis-Code“ der Nachricht. Dieser gibt in jeweils 4 Bits den „Status“ und den „Kanal“ des Ereignisses an. Während der Status angibt, was für eine Aktion ausgeführt werden soll und damit auch wie viele Datenbytes für diese Nachricht noch folgen, besagt der Kanal, für welches Instrument dieses Ereignis gilt. Beim Anspielen und Aufhören des Spielens einer Note folgen jeweils zwei Bytes, die respektiv die Note und Spielstärke angeben [vgl. Van12, Kapitel 2].

Hierbei fallen einige Daten auf, die in diesem Projekt unnötig sind. So gibt es hier zum Beispiel nur ein einziges Instrument, weswegen die Notwendigkeit unterschiedlicher Kanäle wegfällt. Auch ist die sehr hohe Auflösung der „delta time“ hier nicht benötigt. Stattdessen würde eine Auflösung in Millisekunden oder Centisekunden für dieses Projekt ausreichen.

Weiterhin fällt auf, dass für das Spielen jedes Tons in MIDI mindestens zwei Nachrichten benötigt werden. Einmal eine Nachricht zum Starten des Ton-Spielens und einmal eine Nachricht zum Aufhören. Diese zweite Nachricht wäre nicht nötig, wenn zusammen mit der ersten Nachricht eine Länge, mit der der Ton gespielt werden soll, gespeichert würde. Diese Speicher-Optimierung bringt allerdings auch Nachteile mit sich. So verkompliziert es die Transformation der Musikdaten, bevor sie zum Ansteuern der Aktuatoren verwendet werden können. Auch müssen die zeitlichen Längen, für die die zurzeit spielenden Töne noch zu spielen sind, gespeichert werden. Dies könnte im Generellen einiges an Speicherplatz verbrauchen. In diesem speziellen Anwendungsfall lässt sich allerdings annehmen, dass

Noten in der Regel nur für relativ kurze Zeiten gespielt werden (in der Regel unter 500ms) und dass nur relativ wenige Noten auf einmal gespielt werden (in der Regel weniger als 10). Da wie in Kapitel 4.2 erklärt, auch im Allgemeinen höchstens 10 Aktuatoren auf einmal aktiviert werden sollen, kann die Menge zusätzlich benötigten Speichers hier sehr gering gehalten werden.

Aufgrund all dieser möglichen Verbesserungen des Formats, wurde sich entschieden, ein eigenes Datenformat für diesen Anwendungsfall zu entwickeln, welches im Folgenden „Piano Digital Interface Format (PIDI)“ genannt wird. Abgesehen von einer effizienten Speicher Nutzung, soll das PIDI-Format auch performant zum Ansteuern der Aktuatoren transformiert werden können und möglichst simpel zu benutzen sein. Außerdem soll es trotz der starken Einschränkung auf diesen Anwendungsfall die Variabilität besitzen, für unterschiedliche Konfigurationen von Klavieren verwendbar zu sein, um der Anforderung A8 gerecht zu werden.

Listing 5.1 zeigt die endgültige Datenstruktur, die für ein individuellen `PidiCmd` gewählt wurde, in Form eines C-Bitfields. Die vollständige Spezifikation des Formats lässt sich in Anhang C.1 finden.

```

1 typedef enum {
2     PIANO_KEY_C,
3     PIANO_KEY_CS,
4     PIANO_KEY_D,
5     PIANO_KEY_DS,
6     PIANO_KEY_E,
7     PIANO_KEY_F,
8     PIANO_KEY_FS,
9     PIANO_KEY_G,
10    PIANO_KEY_GS,
11    PIANO_KEY_A,
12    PIANO_KEY_AS,
13    PIANO_KEY_B,
14 } PianoKey;
15
16 typedef struct PidiCmd {
17     uint32_t dt : 12,
18     velocity : 4,
19     len : 8,
20     octave : 4, // needs to be sign-extended when used, because it's a signed 4-Bit integer
21     key : 4;
22 } PidiCmd;

```

Quelltext 5.1: Definition eines PIDI-Kommand

Ein `PidiCmd` stellt dabei das Anspielen eines Tons dar. Die Töne selbst wurden als ein Paar aus Oktave und Taste in der Oktave dargestellt, da die Anzahl Tasten in einer Oktave auf praktisch allen Klavieren gleich ist. Die nullte Oktave bezeichnet hierbei die mittlere Oktave des Pianos, während positive und negative Zahlen in Relation zur mittleren Oktave zu verstehen sind. Mit jeweils 4 Bits können alle 12 Tasten in einer Oktave und alle Oktaven, die auch auf großen Klavieren Platz finden, verwendet werden.

Die „velocity“ eines `PidiCmds` gibt die Anschlagsstärke an. Um Speicherplatz zu sparen und da genauere Präzision beim Anspielen sich bereits auf Hardware-Seiten als Schwierigkeit ergeben hat (siehe Kapitel 3.1.2), wurde sich entschieden, nur 4 Bits für die „velocity“ bereitzustellen. Zuletzt gibt es dann noch die `dt` und `len`, die respektiv jeweils die Zeit seit der zuletzt gespielten Note und die Spiellänge der Note darstellen.

Während in MIDI eine sehr hohe und dynamisch veränderbare Auflösung für die „delta time“ verwendet wurde, ist `dt` hier immer in Millisekunden enkodiert. Mit den dafür allo-

kierten 12 Bits können $\frac{2^{12}}{1000} = 4,096$ Sekunden ohne Anspielen eines neuen Tons enkodiert werden.

Da die Anspiellänge zusätzlich noch für jeden zurzeit gespielten Ton gespeichert werden muss, wurde entschieden genau ein Byte dafür zu verwenden. Dies erlaubt eine leichte und speicher-effiziente Nutzung außerhalb der PidiCmd-Datenstruktur. Da 8 Bits nur Spiellängen von bis zu $\frac{2^8}{1000} = 0,256$ Sekunden abspeichern können, wurde entschieden hier stattdessen eine Enkodierung in Centisekunden zu benutzen. Damit können bis zu $\frac{2^8}{100} = 2,56$ Sekunden abgespeichert werden.

Insgesamt benötigt ein PidiCmd dann genau 4 Byte. Damit passt die Datenstruktur gut in CPU-Register, was eine schnellere Benutzung ermöglicht. Wenn davon ausgegangen wird, dass die meisten Töne mit einer Anspielstärke gespielt werden und ausklingen bevor sie mit einer anderen oder derselben Stärke wieder gespielt werden, dann benötigt MIDI für jeden Ton 8 Bytes - 4 Bytes zum initialen Anspielen und 4 Bytes zum Ausklingen des Tons. PIDI benötigt dagegen immer genau 4 Bytes für jeden Ton und ist somit doppelt so speicher-effizient.

5.2 Logik des Mikrocontrollers

Der μ C hat grundlegend zwei Aufgaben. Einerseits muss der μ C Daten an die Aktuatoren senden, um das Klavier richtig anzuspielen. Andererseits muss der μ C auch mit SAM kommunizieren können. Da der μ C nur einen einzelnen CPU-Kern hat, können diese Aufgaben nur abwechselnd nacheinander und nicht parallel bearbeitet werden. Während die Anspiellogik in diesem Kapitel erläutert wird, soll die Kommunikation mit SAM im nächsten Kapitel 5.3 dargelegt werden.

Um die richtigen Werte zu den Aktuatoren zu senden, müssen nacheinander alle Werte an die Schieberegister gegeben werden. Der Wert des letzten Schieberegister-Ausgangs muss dabei zuerst gesendet werden, da es dann als erstes komplett durchgeschoben wurde. Sobald alle Werte ausgesandt wurden, müssen die Ausgänge der Schieberegister gesperrt werden, damit diese nicht ständig fluktuieren. All das kann beim Arduino UNO R3, der in Kapitel 3.2.1 als μ C gewählt wurde, über das Ansprechen spezieller PWM-Pins getan werden.

Wann immer die Signale für die Aktuatoren ausgesandt werden sollen, müssen diese Werte vorher bereitstehen. Da 88 Aktuatoren angesteuert werden sollen, muss also eine Liste von

88 PWM-Werten befüllt werden. Im Folgenden soll diese Liste piano genannt werden. Um piano zu befüllen müssen die Informationen des Musikstücks verwendet werden, die wie in Abschnitt 5.1 beschrieben, als eine Liste von PidiCmds gespeichert werden.

Die PidiCmd-Struktur ist nicht nur sehr speicher-effizient, sondern erlaubt auch eine performante Anpassung der piano Liste. Da die PidiCmds zeitlich sortiert sein müssen, reicht ein laufender Index in die Liste. Um zu entscheiden, ob eine Note gespielt werden soll, wird außerdem die Zeit benötigt, die seit Spielen des letzten PidiCmds vergangen ist. Außerdem muss überprüft werden, ob einige der zurzeit spielenden Töne wieder ausgeschaltet werden sollten. Um dies schnell berechnen zu können, wird eine Liste mit allen zurzeit gespielten Noten gehalten. Für jede Note wird dabei die Zeit gespeichert, für die sie weiterhin noch gespielt werden sollte. Listing 5.2 stellt diese Logik in Pseudocode dar.

```

1 PIDICmds *cmds; // assume this array to be initialized already
2 u8 piano[88]; // list of values to send to magnets
3 u32 index = 0;
4 u32 elapsed_time = 0;
5 while (true) { // Endless Loop of Microcontroller
6     u32 start_time = currentTime(); // time in milliseconds
7     update_played_keys(elapsed_time);
8     while (index < length(cmds) && cmds[index].dt <= elapsed_time) {
9         applyCmd(piano, cmds[index]);
10        add_to_played_keys(cmds[index].dt);
11        elapsed_time -= cmds[index].dt;
12        index++;
13    }
14    elapsed_time += getElapsedTimeSince(start_time);
15 }
```

Quelltext 5.2: Nutzung der PidiCmd-Struktur

Die in Listing 5.2 erwähnten Funktionen `currentTime` und `getElapsedTimeSince` sind über die eingebaute Uhr des μ C umsetzbar. Die Funktion `applyCmd` setzt die Stärke des Tastenan-schlags an der richtigen Stelle in der piano Liste. Da die PidiCmd-Struktur die Anordnung und Anzahl verfügbarer Aktuatoren ignoriert, muss diese Umrechnung von Oktave und Ton in den genauen Aktuator, der diese Taste betätigt, hier geschehen.

Zuletzt verwendet das Listing noch die Funktionen `update_played_keys` und `add_to_played_keys`. Diese benötigen eine Liste der zurzeit gespielten Töne. Weil nie mehr als 10 Aktuatoren zu einem Zeitpunkt aktiviert sein dürfen (siehe Kapitel 4.2), kann diese Liste klein gehalten

werden.

Sollte ein Musikstück diese Obergrenze gleichzeitig zu spielender Noten überschreiten, müssen einige Töne früher beendet werden als eigentlich erwünscht.

Hier gibt es mehrere Strategien, die jeweils eigene Vor- und Nachteile mit sich bringen. Die am leichtesten zu implementierende Idee, würde neue Noten einfach nicht spielen lassen. Entweder könnten diese neuen Noten übersprungen werden, oder es könnte gewartet werden, bis eine der zurzeit spielenden Noten wieder ausklingen sollen, bevor weitere PidiCmds überhaupt betrachtet werden. Letztere Strategie würde quasi die Geschwindigkeit des Musikstücks kurzzeitig anpassen und eher ungewollte Anpassungen an der Natur des Stücks vornehmen. Um den Nachteil der ersten Strategie aufzuzeigen, soll ein Beispiel verwendet werden.

Angenommen es dürfen nur 5 Aktuatoren auf einmal aktiviert werden. Nun werden gerade C, D, E, F und G der mittleren Oktave gespielt. Das C soll nur noch für ein paar Millisekunden gespielt werden. Davor soll aber nun ein B gespielt werden. Dieses B soll nun für eine ganze Sekunde gespielt werden. Optimal wäre es hier, das C, das sowieso bald ausklingen soll, ein paar Millisekunden früher auszuschalten, damit das B zum richtigen Zeitpunkt schon gespielt werden kann. Die zuerst genannte Strategie würde dagegen das B einfach überhaupt nicht spielen.

Aus diesem Beispiel lässt sich eine alternative Strategie entwickeln. So lassen sich die Längen, für die die bereits gespielten Töne und den neu zu spielenden Ton vergleichen. Die Töne, die am frühesten wieder ausklingen sollen, können dann bereits ein wenig früher abgebrochen werden. Da die Liste bereits gespielter Töne kurz sein muss, ist eine Iteration über alle bereits gespielten Töne nicht besonders zeitaufwendig. Nichtsdestotrotz ist dieses Vorgehen komplizierter und rechenaufwendiger als die beiden erstgenannten Ideen.

Eine weitere Möglichkeit wäre statt der Länge die Anschlagstärken zu vergleichen. Es kann davon ausgegangen werden, dass das Fehlen stärker zu spielender Töne eher negativ auffallen würde als das Fehlen leiserer Töne.

Es ließe sich auch eine Heuristik erstellen, die sowohl die Anschlagsstärke als auch die noch verbleibende Spiellänge mit einander vermischt. Ohne Testen an realen Musikstücken, ist es jedoch schwer zu sagen, wie eine solche Heuristik aussehen sollte. Hier ist es also sinnvoll eine leicht anpassbare Heuristik zu verwenden und während des Testens dann zu analysieren, welche Heuristik die am besten klingende Resultate mit sich bringt.

Die benötigte Logik zum Anspielen des Klaviers ist damit fast vollständig besprochen. Es

fehlt jedoch noch die Logik zum dynamischen Anhalten und Anpassen der Lautstärke und Spielgeschwindigkeit (siehe Anforderungen A10 und A11).

Das Anhalten kann mithilfe eines einfachen Bool'schen Wert gelöst werden. Wenn angehalten wurde, soll statt den Werten aus der piano Liste, eine Null an jeden Aktuator gesendet werden. Auch sollen keine weiteren PidiCmds angewandt werden und die `elapsed_time` soll nicht weiter inkrementiert werden.

Die Lautstärke ist über Erhöhen der Anschlagsstärke umsetzbar. Da PidiCmds die Anschlagsstärke in einem abstrakten Wert von 0 bis 15 darstellen, kann dieser Wert über einen Lautstärke-Faktor einfach skaliert werden.

Die Geschwindigkeit kann ebenfalls über einen Faktor implementiert werden. Dieser Faktor skaliert dabei jedoch die Messung der vergangenen Zeit. Hierbei ist aufzupassen, dass nicht die in Listing 5.2 verwendete Variable `elapsed_time` skaliert wird, sondern der Term `getElapsedTimeSince`, der zur `elapsed_time` hinzu addiert wird.

Ansosten würde nämlich die akkumulierte Zeit der Vergangenheit angepasst werden, statt die Geschwindigkeit der zurzeit laufenden Zeit zu verändern.

5.3 Kommunikation

Den in Kapitel 2 gestellten Anforderungen nach, soll über SAM das automatisch gespielte Musikstück dynamisch wählbar sein (Anforderung A9). Das bringt mit sich, dass SAM Daten an den μ C senden können muss.

Abgesehen von den zu spielenden Noten, muss SAM dem μ C auch mitteilen können, ob angehalten, oder die Lautstärke bzw. Geschwindigkeit angepasst werden sollen (Anforderungen A10 und A11).

Hierbei sollten mehrere Fragen auftreten:

1. Geht die Kommunikation in beide Richtungen oder nur von SAM zum μ C?
2. Welche Medien sollen für die Kommunikation unterstützt werden?
3. Soll die Kommunikation seriell oder parallel ablaufen?
4. Über welches Protokoll werden die Daten encodiert?
5. Soll die Kommunikation synchron, asynchron oder isochron ablaufen?

6. Wie wird Datenkonsistenz sichergestellt?
7. Wie wird Datenvollständigkeit sichergestellt?
8. Wie wird mit Fehlern umgegangen?

Zuerst mag eine einseitige Kommunikation ausreichend erscheinen. Allerdings erlaubt eine zweiseitige Kommunikation bessere Portabilität der Anwendung. So muss SAM wissen, über welchen Port der μ C angesprochen werden kann. Da nicht alle Maschinen die gleichen Ports zur Verfügung stehen haben, sollte dieser Port dynamisch wählbar sein.

Um den korrekten Port entsprechend auch dynamisch finden zu können, ist es hilfreich, wenn der μ C ein einzigartiges Signal über den Port zu SAM schicken kann. Hier kann beispielsweise ein spezielles „PING“-Signal über jeden verfügbaren Port geschickt werden. Während der „PING“ von anderen Ports ignoriert wird, kann der μ C mit dem entsprechenden „PONG“-Signal antworten. Somit kann SAM jederzeit die Verfügbarkeit des μ Cs und dessen Port prüfen.

Mögliche Übertragungsmedien für die Kommunikation lassen sich in die Kategorien „Kabel“ und „Kabellos“ einteilen. Kabellose Medien wie WLAN oder Bluetooth benötigen bei dem in Kapitel 3.2.1 gewählten μ C ein weiteres Modul. Das würde die Kosten weiter erhöhen und damit entgegen dem Ziel einer möglichst preisgünstigen Entwicklung sprechen. Nichtsdestotrotz sollte die Kommunikation erweiterbar konzipiert werden, um auch leicht kabellos umgesetzt werden zu können.

Da es beim gewählten μ C jedoch schon eingebaut ist und damit die preisgünstigste Möglichkeit darstellt, soll hier eine Verbindung über ein USB-Kabel zwischen μ C und SAM angenommen werden.

Damit wird auch die nächste Frage bereits beantwortet, da über das USB-Kabel nur eine serielle Kommunikation möglich ist. Der gewählte μ C schränkt auch die möglichen Protokolle zur Übertragung von Daten ein. So unterstützt der Arduino Uno R3 nur die folgenden drei Übertragungsprotokolle:

1. Inter-Integrated-Circuit (I2C)
2. Serial Peripheral Interface (SPC)
3. Universal Asynchronous Receiver-Transmitter (UART)

I2C und SPC sind beides Protokolle, die für die Kommunikation mit Periphergeräten gedacht wurden [vgl. ZSH24; AS23]. UART wird dagegen von der Arduino Dokumentation für

generelle Kommunikation mit einem PC empfohlen [vgl. Sie23]. Da die Desktop-Anwendung das selbe Protokoll sprechen können muss, ist die Nutzung von UART auch insoweit sinnvoll, dass Windows nativ die Nutzung des UART-Protokolls über sogenannte „COM“-Ports ermöglicht [vgl. Mar22].

Da UART verwendet wird, steht auch fest, dass die Übertragung von Daten asynchron abläuft.

Das UART-Protokoll bringt auf Level der Bit-Übertragung bereits Mechanismen zum Sicherstellen der Datenkonsistenz und Vollständigkeit mit sich. Bis jetzt wurde jedoch nur geklärt wie die Bits der Nachrichten übertragen werden, nicht jedoch wie diese Nachrichten aussehen sollen. Das benötigt ein weiteres Protokoll, das diesmal jedoch über Software in den Anwendungen des Arduinos und Desktops implementiert werden muss. Im Folgenden soll dieses Protokoll „Self-Playing Piano Protocol (SPPP)“ genannt werden.

Es wäre hierbei möglich ein bereits existierendes, generelles Kommunikationsprotokoll zu verwenden. Zum Beispiel könnten die Daten über das Hypertext Transfer Protocol (HTTP) gesendet werden. Jedoch muss selbst dann noch ein spezielles Schema spezifiziert werden, in dem die Daten angeordnet werden.

Entsprechend ist es sinnvoll hier ein eigenes, einfaches Protokoll zu entwerfen, dass alle Anforderungen an die Kommunikation abdecken kann. Dabei muss das SPPP Protokoll die folgenden Anforderungen erfüllen, welche von den in Kapitel 2 gestellten Anforderungen abgeleitet wurden.

1. SAM muss einen „Ping“ senden können, auf die der μ C mit einem „Pong“ antwortet, damit der genaue Port, über dem der Arduino verbunden ist, identifiziert werden kann
2. SAM muss dem μ C sagen können, dass die Musik angehalten oder fortgesetzt werden soll
3. Die Lautstärke muss von SAM an den μ C gesendet werden können
4. Die Geschwindigkeit muss von SAM an den μ C gesendet werden können
5. Musikdaten in Form von PidiCmds müssen von SAM an den μ C gesendet werden können
6. SAM muss dem μ C sagen können, dass zu einem beliebigen Zeitpunkt im Musikstück gesprungen werden soll

7. Fehler in der Kommunikation sollten erkannt werden, damit SAM und μ C diese entsprechend behandeln können
8. Die Latenz beim Senden von Nachrichten sollte möglichst gering und mindestens unter einer Sekunde sein
9. Das Protokoll soll erweiterbar sein, sodass leicht weitere Versionen erstellt werden können, ohne Backwards-Compatibility zu brechen

Aus den Anforderungen wird klar, dass es unterschiedliche Arten an Nachrichten gibt, die vom Protokoll unterstützt werden müssen. Ein entsprechend simpler Aufbau von SPPP Nachrichten besteht aus einer ID, die den Typ der Nachricht identifiziert, und einem Datensatz, der je nach Nachrichten-Typ unterschiedlich spezifiziert ist.

Damit lässt sich das Protokoll auch leicht erweitern, da neue Nachrichten-Typen einfach hinzugefügt werden können. Es empfiehlt sich außerdem jede Nachricht mit bestimmten „Magic Bytes“ zu starten, um keine arbiträren Datenströme als Nachrichten zu interpretieren.

Um die Latenz der Nachrichten unabhängig technischer Gegebenheiten gering zu halten, empfiehlt es sich, die Anzahl zu übertragener Bytes möglichst gering zu halten. Entsprechend wurde entschieden, die IDs der Nachrichten-Typen auf jeweils ein einzelnes Byte zu reduzieren. Zur leichteren Identifizierung werden in Nachrichten-IDs nur ASCII-enkodierte Buchstaben verwendet. Weiterhin wird hier die Konvention eingeführt, dass Großbuchstaben eine Nachricht von SAM und Kleinbuchstaben eine Nachricht des μ Cs spezifizieren.

Die Magic Bytes des Protokolls werden als die 3-Byte Sequenz „SPP“ definiert. Ein einzelnes Byte wäre zu unsicher, da jedes zufällig empfangene Byte eine $\frac{1}{256} = 0,39\%$ Chance hat das Magic Byte zu sein.

Die meisten Nachrichten müssen jeweils nur eine einzelne Zahl übersenden. Tabelle 5.1 listet alle Nachrichtentypen mit ihren jeweils zu übertragenen Daten auf. Die vollständige Spezifikation des Protokolls findet sich in Anhang C.3.

5.3.1 Fehlererkennung

Wie in den oben gelisteten Anforderungen erwähnt, benötigt es jedoch noch eine Möglichkeit Fehler in der Kommunikation zu erkennen. Es gibt hierbei grundsätzlich drei Arten an potenziellen Fehlern:

ID	Beschreibung	Payload
„P“	„Ping“-Nachricht	-
„p“	„Pong“-Nachricht - Antwort vom μ C auf „Ping“-Nachricht	Maximal-Anzahl von PidiCmds pro „Music“-Nachricht als 16-Bit Zahl
„C“	„Continue“-Nachricht, zum Anhalten bzw. Fortsetzen der Musik	1 Byte; Nur bei 0 soll die Musik angehalten werden
„V“	„Volume“-Nachricht - setzt die Lautstärke	32-bit Floating-Point Wert
„S“	„Speed“-Nachricht - setzt die Abspielgeschwindigkeit	32-bit Floating-Point Wert
„s“	„Success“-Nachricht - Antwort vom μ C auf jede erfolgreich erhaltene Nachricht	-
„M“	„Music“-Nachricht	siehe Abschnitt 5.3.2
„N“	„New-Music“-Nachricht	siehe Abschnitt 5.3.2
„r“	„Request“-Nachricht - vom μ C geschickt (siehe Abschnitt 5.3.2)	-

Tabelle 5.1: SPPP-Nachrichten

1. Inkorrekte Daten werden empfangen
2. Die Nachricht wird nicht empfangen
3. Teile der Nachricht werden nicht empfangen

Die erste Art an Fehlern soll hier erstmal ignoriert werden, da sie dank der von USB und UART versicherten Datenkonsistenz ein relativ geringes Eintrittsrisiko darstellt. Nichtsdestotrotz ist dies eine mögliche Fehlerquelle, die in Erweiterungen an diese Arbeit ausgebessert werden sollte. Eine mögliche Sicherstellung der Datenkonsistenz würde beispielsweise über das Mitsenden eines Kontroll-Bytes funktionieren.

Der zweite Fehler kann sowohl durch Fehler in der Übertragung, als auch durch einen Fehler bei der Software des Empfängers entstehen. Eine einfache Lösung dieses Problems, lässt den Empfänger eine Empfangsbestätigung zurücksenden. Das selbe Prinzip wird bei der oben erwähnten „Ping“ und „Pong“-Nachricht verwendet. Ein mögliches Problem beim Senden einer Empfangsbestätigung ist die Ungewissheit, ob die Empfangsbestätigung selbst empfangen wurde.

Dann müsste nämlich eine Empfangsbestätigung für die Empfangsbestätigung gesendet werden, welche ebenfalls wieder als empfangen bestätigt werden müsste, bis nur noch Empfangsbestätigungen hin und her geschickt würden.

Es stellt sich jedoch heraus, dass das hier kein Problem ist. Sollte eine Empfangsbestätigung nicht empfangen werden, soll der Sender die selbe Nachricht einfach nochmal senden. Solange das zweimalige Erhalten der selben Nachricht genau die selben Auswirkungen hat wie das einmalige Erhalten der Nachricht, wird keine Empfangsbestätigung für die Empfangsbestätigung selbst benötigt, da die Nachricht ohne Risiken einfach nochmal geschickt werden kann. Wenn die in Tabelle 5.1 gelisteten Nachrichtentypen betrachtet werden, fällt außerdem auf, dass nur die Nachrichten, die von SAM gesendet werden, eine Empfangsbestätigung benötigen. Diese Bestätigung soll über die „Success“-Nachricht gegeben werden, welche entsprechend nur vom μ C geschickt werden muss.

Ein weiteres Problem, das mit Empfangsbestätigungen auftreten kann, ist das Zuordnen der Nachricht, die nun als empfangen bestätigt wurde.

Sollte SAM beispielsweise zwei Nachrichten direkt nacheinander schicken und dann nur eine einzelne „Success“-Nachricht zurück erhalten, ist unklar welche der beiden Nachrichten nun angekommen ist und welche nicht. Dies kann gelöst werden, indem jeder Nachricht eine einzigartige Nummer zugeordnet wird und die „Success“-Nachricht diese Nummer referenziert.

Alternativ kann auch spezifiziert werden, dass eine neue Nachricht nur gesendet werden darf, wenn eine „Success“-Nachricht erhalten wurde oder eine gewisse Zeit vergangen ist. Diese Zeit wird auch als „Timeout“ benannt.

Die zweite Methode bringt mit sich, dass das SPPP Protokoll synchron wird und deshalb weniger flexibel als die asynchrone, erste Lösungsvariante erscheinen mag. Allerdings erleichtert die Synchronisierung des Protokolls auch den Umgang mit dem 3. Fehler, wie nachher gezeigt wird. Entsprechend wird sich für diese Lösungsmöglichkeit entschieden.

Weiterhin sollte die Frage auftreten, warum statt einer eigenen „Pong“-Nachricht nicht einfach die selbe „Success“-Nachricht gesendet werden kann, wenn der Austausch von „Ping“- und „Pong“-Nachrichten genauso dem Prinzip der Empfangsbestätigung folgt.

Der Grund dafür liegt in den „Music“-Nachrichten, die eine variable Länge haben. Da dieses Protokoll - wie auch das in Kapitel 5.1 spezifizierte PIDI-Format - portabel für verschiedene Hardware-Voraussetzungen sein soll, kann SAM nicht die Speicherkapazitäten des μ Cs voraussetzen.

Entsprechend muss der μ C SAM irgendwie mitteilen, was die akzeptierte Maximallänge für die „Music“-Nachricht ist. Da die „Ping“-Nachricht im Regelfall sowieso die zuerst gesendete Nachricht wäre, kann der μ C deren Empfangsbestätigung nutzen, um SAM dieses Maximum mitzuteilen. Damit ist die „Ping“-Nachricht nun notwendigerweise die zuerst

gesendete Nachricht in diesem Protokoll.

Es bleibt nun noch der 3. Fehler zu behandeln. Wenn ein Teil der gesendeten Nachricht ausfällt, gibt es zwei Möglichkeiten:

Entweder werden weitere Bytes empfangen, sodass eine scheinbar vollständige, wenn auch inkorrekte Nachricht erhalten wird. Das ist im Endeffekt wieder das 1. Problem und wird ebenso im Rahmen dieser Arbeit ignoriert. Weitere Versionen des Protokolls sollten diese Quellen möglicher Dateninkonsistenz jedoch behandeln.

Es ist allerdings auch möglich, dass keine weiteren Bytes empfangen werden und die Nachricht unvollständig bleibt. Da das Protokoll synchron ist und keine weiteren Nachrichten gesendet werden sollten, bevor eine Empfangsbestätigung zurückgeschickt wurde, ist dies tatsächlich der sehr viel wahrscheinlichere Fall. Da für die Behandlung des 2. Problems sowieso schon ein Timeout festgelegt werden muss, kann dieser Timeout hier ebenfalls verwendet werden. Sobald der Timeout abgelaufen ist und die Nachricht noch nicht vollständig empfangen wurde, soll diese einfach verworfen werden. Sollten die fehlenden Bytes nach Ablauf desTimeouts eintreffen, werden diese ignoriert, da sie sehr recht wahrscheinlich nicht die Magic-Bytes des SPPP Protokolls beinhalten.

In den oben genannten Anforderungen wurde die maximale Latenz für Nachrichten auf 1 Sekunde gesetzt. Diese Anforderung kann eins-zu-eins als Wert des Timeouts von SPPP-Nachrichten übernommen werden. SAM muss dabei jedoch den doppelten „Timeout“ als Wert nehmen, da sie zusätzlich noch auf den Eingang der Empfangsbestätigung warten muss.

5.3.2 „Music“-Nachrichten

Die grundlegende Aufgabe, für die die Kommunikation zwischen SAM und μ C benötigt wird, ist das Transferieren der zu spielenden Musikdaten. Wie in Kapitel 5.2 beschrieben, verwendet der μ C das selbst-erstellte PIDI-Format zum Speichern der Musikdaten. Entsprechend ist es sinnvoll, diese Daten direkt im gleichen Format an den μ C zu schicken.

Da der Speicherplatz des μ Cs jedoch vergleichsweise stark begrenzt ist, können viele Songs nicht komplett gespeichert werden. Auch wenn diese Limitation mit Erweiterungen der Hardware gelöst werden könnten, würden diese die Kosten weiter heben und die Portabilität verringern. Um dem Ziel der Arbeit gerecht zu werden, darf kein großer Speicherplatz vorausgesetzt werden.

Wenn also nicht davon auszugehen ist, dass das gesamte Musikstück auf einmal auf dem

μ C gespeichert werden kann, müssen die Musikdaten in Teilen nach einander geschickt werden. In anderen Worten: es ist ein Streaming der Musikdaten notwendig.

Eine erste Problematik, die hierbei aufkommt, ist die Latenz der Nachrichten. Wenn nach Spielen eines Teils des Songs immer eine längere Wartezeit folgt, bevor die nächsten Töne gespielt werden können, macht es das selbst-spielende Klavier praktisch unnutzbar. Diese Problematik ist auch als „Buffering“ bekannt [vgl. Clo21].

Die einfachste Art dem Vorzeubugen ist durch die Verwendung von zwei Buffern für die PidiCmds. Ein Buffer hält die derzeit zu spielenden PidiCmds, während der zweite Buffer mit den danach zu spielenden PidiCmds befüllt werden kann. Sobald der erste Buffer fertig gespielt wurde, können die beiden Buffer effizient mit einem Tausch ihrer Pointer gewechselt werden.

Das Problem des Bufferings kann dann nur noch vorkommen, wenn der erste Buffer fertig gespielt wurde, bevor der nächste Buffer aufgefüllt werden konnte. Eine einfache Verbesserung dieser Strategie ist die Nutzung von mehr als nur zwei Buffern. Wenn zum Beispiel drei Buffer genutzt werden, müssen zwei Buffer durchgespielt werden, bevor der nächste Buffer aufgefüllt werden muss. Auch können die Buffer bereits getauscht werden, bevor die Nachricht fertig gelesen wurde, solange zumindest ein PidiCmd bereits gelesen wurde.

Die nächste Schwierigkeit kommt daher, dass SAM wissen muss, wann der nächste Teil des Musikstücks benötigt wird.

SAM könnte versuchen selbst die vergangene Zeit zu zählen und die selben Berechnung wie der μ C zu machen, um zu wissen, wieweit der μ C bis jetzt gespielt hat. Falls hierbei jedoch ein Fehler auftritt, würden die Musikdaten zum falschen Zeitpunkt gesendet werden. Abgesehen davon wäre dies auch ein relativ großer Entwicklungsaufwand. Einfacher ist es, wenn der μ C eine bestimmte Nachricht schickt, um den nächsten Teil des Musikstücks anzufordern. In Tabelle 5.1 ist diese Nachricht als „Request“-Nachricht spezifiziert.

Die „Request“-Nachricht ist die einzige Nachricht, die der μ C schickt, ohne auf eine Nachricht von SAM zu antworten. Trotzdem muss SAM hier nicht mit einer Empfangsbestätigung antworten, da sie sowieso mit einer Nachricht antwortet. Nur ist die Antwort von SAM eine „Music“-Nachricht stattdessen.

Die „Request“-Nachricht enthält, wie in Tabelle 5.1 spezifiziert, keine weiteren Daten. Das heißt, dass SAM sich merken muss wie weit im Musikstück die letzte Nachricht fortgeschritten war. Ein alternativer Ansatz würde den μ C speichern lassen, wie viele PidiCmds

bereits gespielt wurden und würde diesen Index in der „Request“-Nachricht an SAM schicken. Beide Methoden geben sich nicht besonders viel, hier wurde sich jedoch für die erste Möglichkeit entschieden, da sie der Desktop-Anwendung mehr Flexibilität bietet und weniger Daten übertragen werden müssen.

Die soweit beschriebene Strategie funktioniert zum Spielen vollständiger Stücke nacheinander. Nun soll die Anwendung aber auch die Möglichkeit bieten, während des Spielen eines Stücks zu einem anderen zu wechseln. In diesem Fall muss dem μ C mitgeteilt werden, dass ein neues Stück anfängt und alle Aktuatoren zu Beginn wieder ausgeschaltet werden sollen.

Zusätzlich soll es den Nutzer:innen möglich sein, in einem zurzeit spielenden Stück zu einem anderem Zeitpunkt zu springen. In diesem Fall müssen die zurzeit aktiven Aktuatoren womöglich ebenfalls ausgeschaltet werden, da einige der in Zeitpunkt A zu spielenden Noten beim Zeitpunkt B vermutlich nicht gespielt werden sollen.

Jedoch ist es hier im Allgemeinen nicht der Fall, dass alle Aktuatoren ausgeschaltet werden sollten. Die Aktuatoren der Noten, die bis zu dem gewählten Zeitpunkt gespielt werden sollen, müssen nämlich angeschaltet werden.

In Kapitel 5.2 wurde beschrieben, wie der μ C eine Liste bereits gespielter Noten für die selbe Problematik verwendet. Es liegt entsprechend nahe, das selbe Konzept hier zu verwenden. So kann ebenjene Liste direkt auf Seiten SAMs erstellt werden und zum μ C geschickt werden.

Es müssen hier jedoch noch die Elemente innerhalb dieser Liste spezifiziert werden. Im Folgenden sollen diese Elemente als PlayedKeys bezeichnet werden. Wie in Listing 5.2 auch, müssen die PlayedKeys die verbleibende Länge, mit der der Ton gespielt werden soll, speichern. Weiterhin muss auch die Stärke, mit der der Ton gespielt werden soll, gespeichert werden. Die PlayedKey-Struktur besteht entsprechend aus einer Referenz auf den Ton, der Länge und Spielstärke. Wie im PIDI-Format wird die anzuspielende Taste über Oktave und Note definiert. Da Oktave, Note und Anspielstärke im PIDI-Format jeweils als 4-Bit Zahlen und die Länge als 8-Bit Zahl spezifiziert wurde, benötigt eine PlayedKey-Struktur 20 Bits. Das heißt auch, dass 3 Byte für die Struktur benötigt werden, wobei 4 Bits als Padding verwendet werden.

Auf Seiten des μ Cs ergibt es Sinn die Oktave und Note direkt in den Index für den zuständigen Aktuator zu transformieren, damit diese Berechnung nur einmal ausgeführt werden muss. Ein weiterer Unterschied zwischen der PlayedKeys Liste in der „Music“-Nachricht

und auf dem μ C ist in der Länge der Liste. Da der μ C nur eine bestimmte Anzahl gleichzeitig spielender Tasten haben darf, ist die Länge dessen Liste beschränkt. Da das SPPP Protokoll ebenjene Feinheiten der Hardware abstrahieren soll, ist diese Limitierung in der „Music“-Nachricht nicht gegeben. Sollten in der Nachricht also mehr PlayedKeys sein, als der μ C gleichzeitig spielen darf, muss der μ C das gleiche Verfahren zum Eliminieren einiger PlayedKeys anwenden, dass er auch sonst verwendet (siehe Kapitel 5.2).

Da diese Liste nur überendet werden muss, wenn ein neues Musikstück angefangen wird, oder zu einer anderen Stelle im spielende Stück gesprungen wird, gibt es zwei Möglichkeiten. Entweder wird ein Byte in der „Music“-Nachricht eingefügt, dass klarstellt, ob die Liste von PlayedKeys vorhanden ist oder nicht. Oder es wird ein anderer Nachrichten-Typ dafür verwendet. Letztere Strategie benötigt kein zusätzliches Byte in der Nachricht und wurde somit gewählt. Dafür wird nun die „New-Music“-Nachricht verwendet. Beim Eingang einer „New-Music“-Nachricht - anders als bei einer „Music“-Nachricht - ist es nicht erwünscht die zuvor eingegangenen PidiCmds noch fertig zu spielen.

Zuletzt mag noch die Frage auftreten, was passieren soll, wenn das Musikstück fertig gespielt wurde. Sollte es dafür ebenfalls einen eigenen Nachrichtentyp geben? Das wäre eine simple Lösung, würde das Protokoll aber ein wenig verkomplizieren. Sollte SAM stattdessen die eingehenden „Request“-Nachrichten einfach ignorieren? Dann würde der μ C endlos neue „Request“-Nachrichten schicken und somit unnötige Arbeit auf sich nehmen. Stattdessen wurde entschieden, am Ende des Musikstücks eine „Music“-Nachricht mit einer leeren Liste von PidiCmds zu schicken. Damit wird das SPPP Protokoll nicht unnötig erweitert und der μ C muss nicht endlos neue Nachrichten schicken. Der μ C sollte keine weiteren „Request“-Nachrichten schicken, sobald beide Buffer von PidiCmds leer sind. Auf den ersten Blick mag das nach weiterer Arbeit klingen, diese Logik sollte allerdings zu Start des μ Cs - bevor das erste Musikstück gespielt wurde - sowieso gelten. Entsprechend ist es kein weiterer Aufwand in der Ausführung oder Entwicklung des μ Cs.

5.4 Desktop Anwendung

SAM soll den Nutzer:innen eine Schnittstelle zum Anspielen des Pianos bieten. Den in Tabelle 2.1 genannten Anforderungen nach, soll es der Nutzer:in möglich sein Musikstücke aus einem Katalog auszuwählen (A9) und dem Katalog neue Stücke hinzuzufügen (A12). Weiterhin soll die Wiedergabe über die UI steuerbar sein (siehe Anforderungen A10 und

A11).

In diesem Kapitel SAM sowohl in Hinblick auf Code-Architektur (Abschnitt 5.4.2) als auch auf das UI Design (Abschnitt 5.4.1) ein Konzept entwickelt werden.

5.4.1 UI Design

Aus Sicht der Nutzer:innen hat SAM zwei Aufgaben:

1. Musik-Verwaltung
2. Musik-Wiedergabe

Die Musik-Verwaltung beinhaltet das Hinzufügen neuer Musikstücke und die Darstellung des Katalogs. Um auch die Nutzung eines großen Katalogs leicht zu ermöglichen, soll es eine Möglichkeit zum Suchen von Musikstücken nach Namen geben. Bei der Musik-Wiedergabe muss dagegen mit dem μ C und somit indirekt mit dem Piano kommuniziert werden. Auch sollen Kontrollmöglichkeiten bei der Wiedergabe zur Verfügung stehen.

Der Haupt-Use-Case besteht hierbei im Suchen und nachfolgendem Abspielen eines Musikstücks, während das Hinzufügen neuer Musik zum sofortigen oder späteren Abspielen einen zweiten Use-Case abbildet. Diese Rangfolge der Use Cases sollte auch im Design der UI wiedergespiegelt sein.

Ein Ansatz dafür wäre die Teilung der UI in zwei Seiten, wobei die erste, im Normalfall angezeigte Seite den Katalog darstellt, und die zweite Seite den Import von Musikstücken behandelt. Alternativ könnten beide Funktionalitäten auch auf einer einzelnen Seite untergebracht werden mit der Gefahr, die UI damit zu überladen.

Die selbe Frage stellt sich ebenso für die Wiedergabe. Diese kann entweder auf der selben Seite oder auf einer separaten Seite untergebracht werden.

Da die Anwendung möglichst intuitiv und leicht erlernbar sein soll (siehe Anforderung A2), ist es bei diesen Fragen sinnvoll, ein ähnliches Design zu populären Musik-Playern zu verwenden. Hier sollen vor allem der „Windows Media Player (WMP)⁵“ und „Spotify⁶“ zum Vergleich gezogen werden. Es ist davon auszugehen, dass die meisten Nutzer:innen mit mindestens einer der beiden Anwendungen vertraut sind.

⁵<https://support.microsoft.com/en-us/windows/get-windows-media-player-81718e0d-cfce-25b1-aee3-94>

⁶<https://www.spotify.com>

Sowohl WMP als auch Spotify bieten die Möglichkeit zum Betrachten von Katalogen - sogenannten Playlists - an. Beide bieten auch die Möglichkeit an, mehrere Playlists zu erstellen. Im Rahmen dieser Arbeit soll eine solche Funktionalität nicht angeboten werden. Stattdessen soll SAM nur einen einzigen Katalog zur Verfügung stellen. WMP und Spotify bieten außerdem beide die Möglichkeit zum Abspielen von Musik.

Das Öffnen beider Anwendung liefert eine Startseite, der u.a. auf die unterschiedlichen, vorhandenen Playlists verlinkt. Da SAM nur einen Katalog anbietet, wäre dies unnötig und würde die Nutzung der Anwendung nur verlangsamen. Stattdessen soll auf der Startseite der Katalog direkt gezeigt werden.

Beim Abspielen der Musik soll wie bei WMP und Spotify auch eine Timeline des Musikstücks am unteren Fensterrand angezeigt werden. Dabei soll eine Navigation im Musikstück über Klicken auf der Timeline möglich sein. Die Kontroll-Elemente, die zum Erfüllen der Anforderungen A10 und A11 benötigt werden, sollen wie in Spotify oberhalb der Leiste dargestellt werden und ähnliche Symbole verwenden.

Anders als WMP oder Spotify soll SAM in der Lage sein, die Wiedergabegeschwindigkeit anzupassen (siehe Anforderung A10). Dafür könnte entweder ein Slider verwendet werden, wie es auch bei der Konfiguration der Lautstärke üblich ist, oder über Buttons, die die Geschwindigkeit um eine feste Größe de- bzw. inkrementiert, wie es beispielsweise im populären Multi-Media-Spieler „VLC media player“⁷ gelöst ist. Da anzunehmen ist, dass Buttons leichter als ein Slider zu implementieren sind, wurde sich im Rahmen des Prototypen entschieden, die Wiedergabegeschwindigkeit über Buttons steuern zu lassen.

Zuletzt ist nun noch das Hinzufügen von Musikstücken zu beachten. Da das Importieren neuer Musikstücke einen zweitrangigen Use-Case darstellt, wurde sich entschieden, diesen auf einer zweiten Seite umzusetzen. Viele bekannte Anwendungen, bei denen das Hochladen bzw. Importieren von Dateien ebenfalls nicht den primären Use-Case darstellt, lösen dies genauso. Auf der Startseite soll ein Button zum Importieren existieren, der dann zur zweiten Seite führt, wo die zu importierende Datei ausgewählt werden kann, und für den Import umbenannt werden kann.

Es soll der Nutzer:in generell die Möglichkeit gegeben werden, jedes Musikstück im Katalog zu jedem Zeitpunkt umzubenennen und auch wieder zu entfernen. Diese Änderungen sollten dabei durch Eingaben direkt beim dargestellten Musikstück möglich sein. Das hilft laut dem UI Design Pattern „Make it direct“ eine intuitivere Nutzung zu gestalten. Beim

⁷<https://www.videolan.org/vlc/>

Darstellen der Musikstücke sollte entsprechend der Name angezeigt werden, sowie eine Möglichkeit zum Abspielen, Anpassen des Namens und zum Löschen des Musikstücks.

Zuletzt ist beim Import noch zu beachten, dass die Nutzer:innen jederzeit den Import abbrechen können sollten. Da der Import die Nutzer:in auf eine neue Seite bringt, soll das Abbrechen über einen links-gerichteten Pfeil gekennzeichnet werden, welches ein gängiges Symbol zum Rückkehren auf die vorherige Seite darstellt.

5.4.2 Architektur

Um eine sinnvolle Architektur für die Desktop Anwendung zu erstellen, soll hier ein „Bottom Up“ Ansatz genutzt werden. Dabei wird beim Start der Anwendung angefangen und dann jeder „Codepath“, der aufgrund von Nutzer-Interaktionen oder sonstigen Effekten auftreten kann, Schritt für Schritt analysiert. Funktionalitäten die benötigt werden, werden dabei benannt und können zusammengeführt werden, wenn die selbe oder eine ähnliche Funktionalität an anderer Stelle wieder benötigt wird. Gleichzeitig wird auch notiert, welche Daten zu welchem Zeitpunkt vorliegen müssen und wie diese von einander abhängen. Dieser Ansatz wurde von Casey Muratori „Semantic Compression“ getauft [vgl. Mur14].⁸

Mit „Codepath“ ist hier eine endliche, serielle Liste von Instruktionen gemeint. Dabei ist zu beachten, dass auch eine mehrfach ausführende Schleife oder ein *if-else* Block eine Liste von Instruktionen erstellt, die von der CPU ausgeführt werden können [vgl. Fle23]. Eine Endlosschleife dagegen erzeugt keinen Codepath, da sie nicht terminiert. Solche Konstrukte werden stattdessen als „Codecycle“ bezeichnet [vgl. Fle23]. Nachdem die Definitionen nun geklärt wurden, kann die Analyse für die Architektur beginnen.

Eine Anwendung kann dabei durchaus aus mehreren Codecycles bestehen. Zwei Codecycles sind in der Regel relativ unabhängig und sollten zumindest nicht seriell abhängig von einander sein [vgl. Fle23]. Es empfiehlt sich oft zwei Codecycles in separaten Threads oder Prozessen ausführen zu lassen, dies ist aber nicht notwendig. Die Konzeption des μ Cs sieht beispielsweise zwei Codecycles vor - einen zum Lesen und Schreiben von SPPP-Nachrichten und einen zum Wiedergeben der zurzeit laufenden Musik - die nach einander ausgeführt werden (siehe Abschnitt 5.2).

⁸Muratori spricht von Semantic Compression speziell bei der Entwicklung, jedoch wird in seiner Erklärung klar, dass er keinen vorherigen Schritt für die Konzeption der Architektur als notwendig sieht. Hier soll zwar der selbe Ansatz verwendet werden, aber mit einer Trennung des Entwurfs von der Implementierung.

Da SAM auf modernen Desktops laufen soll, die in der Regel über mehrere Kerne verfügen und von Parallelisierung eher profitieren, sollen unterschiedliche Codecycles hier immer in separaten Threads ausgeführt werden.

Nach vermutlich notwendigen Initialisierungen, muss die Anwendung beim Starten in einen Codecycle zum Darstellen der UI übergehen. Dieser Codecycle soll fortgehend auch der „UI-Codecycle“ genannt werden, da er sich um das Rendern der UI kümmert.

Zu Beginn der Anwendung soll der vorhandene Musikkatalog angezeigt werden. Da der Katalog permanent gespeichert werden soll, muss er in einer oder mehreren Dateien gespeichert werden. Der Katalog besteht aus einer Liste von Musikstücken. Jedes Musikstück besteht dabei aus einem Namen sowie der Liste von PidiCmds, die zum Abspielen des Stücks benötigt werden (siehe Abschnitt 5.1).

Zum Darstellen des Katalogs werden dabei nur die Namen der Musikstücke benötigt. Die weiteren Daten werden erst beim Abspielen der Musik gebraucht. Entsprechend ist es sinnvoll, die Namen der Musikstücke in einer separaten Liste (und der Einfachheit halber somit auch einer eigenen Datei) zu speichern.

Dieses Format des Katalogs ist in Anhang C.2 genauer spezifiziert. Hier soll darauf aber nicht eingegangen werden, da es nicht weiter interessant ist.

Um die Startseite der UI anzeigen zu können, muss zuerst also zuerst der Katalog vom Dateisystem ausgelesen werden. Es gibt nun zwei Möglichkeiten zur Anordnung dieses Codepaths zum Auslesen des Katalogs:

1. Der Codepath wird vor dem UI-Codecycle ausgeführt
2. Der Codepath wird parallel zum UI-Codecycle ausgeführt

Der erste Ansatz vereinfacht den Code, der für den UI-Codecycle nötig ist, da davon ausgegangen werden kann, dass der Katalog von Beginn an zur Verfügung steht. Der zweite Ansatz erlaubt dagegen ein schnelleres Darstellen der UI, was die Usability der Anwendung erhöht. Da die zusätzliche Komplexität hier nicht besonders hoch ist, wurde hier die Usability priorisiert und der zweite Ansatz gewählt. Solange der Katalog noch nicht geladen wurde, soll dabei eine Lade-Animation statt des Katalogs angezeigt werden. Daraus folgt auch, dass der UI-Codecycle eine Fallunterscheidung benötigt und darauf basierend etwas anderes darstellt.

Sobald der Katalog geladen wurde, gibt es nun mehrere Möglichkeiten, wie die Nutzer:innen den Zustand der Anwendung ändern können sollen.

1. Suchen eines Musikstücks über eine Suchbox
2. Umbenennen eines Musikstücks über eine zugehörige Eingabebox
3. Löschen eines Musikstücks über einen zugehörigen Button
4. Hinzufügen eines Musikstücks durch Klicken auf den „Import“-Button
5. Wiedergabe eines Musikstücks über einen zugehörigen Button

Die ersten drei Möglichkeiten benötigen keine neuen Daten. Stattdessen wird hier nur der Katalog angepasst, bzw. bei der ersten Möglichkeit wird der Katalog nur gefiltert.

Bei allen drei Codepaths stellt sich wieder die Frage, ob sie parallel zum UI-Codecycle ausgeführt werden sollten oder nicht. Die Parallelisierung bringt jeweils wieder eine erhöhte Komplexität der Implementierung mit sich, stellt aber sicher, dass die UI nicht temporär einfriert. Da die zweite und dritte Aktion eine Interaktion mit dem Dateisystem voraussetzen, was relativ hohe Latenzen mit sich bringen kann, wurde sich entschieden diese parallel zum UI-Codecycle zu gestalten. Die Suche dagegen soll nicht parallel geschaltet werden.

Für das Hinzufügen eines Musikstücks soll, wie in Abschnitt 5.4.1 erwähnt, eine zweite Seite dargestellt werden. Dies benötigt entsprechend eine weitere Fallunterscheidung beim UI-Codecycle. Nachdem die Nutzer:innen eine Datei zum Hinzufügen gewählt haben, muss diese zum Katalog hinzugefügt werden. Da zum Abspielen der Musikstücke das PIDI-Format verwendet wird (siehe Abschnitt 5.1), muss es eine Transformation der eingegebenen Daten zum hier verwendeten Format geben. Je nach Länge des Musikstücks, muss davon ausgegangen werden, dass diese Transformation relativ lange brauchen kann. Es ist entsprechend sinnvoll, diese Transformation zu machen, bevor das Musikstück in den Katalog eingefügt wird. Dann kann ein Musikstück aus dem Katalog jederzeit sofort wiedergegeben werden.

Weiterhin ist aus dem selben Grund auch ratsam, diese Transformation parallel zum UI-Codecycle zu gestalten. Während die Nutzer:innen auf der zweiten Seite sind und den Namen des Musikstücks anpassen können, sollte parallel also der Import der Datei laufen. Das Ergebnis dieses Codepaths muss das Musikstück im richtigen Datenformat sein. Diese Daten müssen dann einerseits in einer Datei gespeichert werden und andererseits muss auch der Name des Musikstücks zum Katalog - sowohl in der Datei des Katalogs als auch im bereits verwendeten Speicher dafür - hinzugefügt werden. Sollten die Nutzer:innen nun aber den Import abbrechen, darf der Katalog noch nicht angepasst sein. Das heißt, dass dieser Codepath nicht die Anpassung des Katalogs selbst vornehmen sollte. Stattdessen sollte

das im UI-Codecycle getan werden, nachdem die Nutzer:innen den Import abgeschlossen haben und der Codepath erfolgreich fertig gelaufen ist. Andernfalls sollte das Ergebnis des Codepaths vom UI-Codecycle wieder verworfen werden.

Zuletzt gilt es noch die 5. Interaktion der Nutzer:innen zu betrachten: Die Wiedergabe eines Musikstücks.

Wie in Abschnitt 5.3 spezifiziert, wird das PPP-Protokoll verwendet, um die Musikdaten an den μ C zu schicken. Da ein Musikstück nicht komplett auf einmal zum μ C geschickt werden kann, wird eine dauerhafte Kommunikation benötigt. Es muss regelmäßig überprüft werden, ob Nachrichten vom μ C eingetroffen sind. Auch müssen zu sendene Nachrichten womöglich warten, bis eine „Success“-Nachricht vom μ C empfangen wurde. Diese Aufgaben sind unabhängig der Darstellung der UI auszuführen und stellen deshalb einen eigenen Codecycle dar.

Es gibt zwar Abhängigkeiten zwischen diesen beiden Codecycles, allerdings sind diese nicht seriell, da die UI nicht auf das Ergebnis der Kommunikation mit dem μ C warten soll, bevor der nächste Frame gezeichnet werden soll.

Die Abhängigkeiten, die existieren, bestehen in der Kontrolle der Wiedergabe. So sollen die Nutzer:innen in der Lage sein, die Wiedergabe zu pausieren und fortzufahren, die Lautstärke und Wiedergabegeschwindigkeit anzupassen, zu einem anderen Zeitpunkt im spielenden Musikstück zu springen und ein anderes Musikstück zu starten.

All diese Wünsche der Nutzer:innen müssen vom UI-Codecycle an den Wiedergabe-Codecycle weitergegeben werden.

Es soll nun betrachtet werden, wie der Wiedergabe-Codecycle genau aussehen sollte.

Zu Beginn ist noch nicht klar, an welchem Port der μ C zu finden ist. Solange dieser Port also unbekannt ist, sollte jeder verfügbare Port eine „Ping“-Nachricht zugesendet bekommen. Falls ein Port mit einer „Pong“-Nachricht antwortet, wurde der μ C gefunden. Sollten zu einem späteren Zeitpunkt keine Antworten über den Port mehr erhalten werden, sollte davon ausgegangen werden, dass der μ C nicht mehr verbunden ist und womöglich an einem anderen Port angeschlossen wird. Entsprechend soll in diesem Fall zum Suchen des Ports zurückgekehrt werden.

Wenn der Port gefunden wurde, müssen zwei Sachen im Wiedergabe-Codecycle passieren:

1. Lesen von potenziell eingegangenen Nachrichten
2. Potenzielles Schreiben ausstehender Nachrichten

Beim Lesen von Nachrichten gibt es vier Möglichkeiten, was glesen werden kann:

1. Keine Nachricht ist eingetroffen
2. „Pong“-Nachricht
3. „Success“-Nachricht
4. „Request“-Nachricht

Bei Fall 2 und 3 kann die zuletzt gesendete Nachricht als empfangen vermerkt werden, damit die nächste Nachricht verschickt werden kann. Bei Fall 4 dagegen müssen die nächsten PidiCmds des Musikstücks versandt werden. Allerdings muss hier darauf geachtet werden, dass die Nachricht nicht sofort geschickt werden darf, wenn eine andere Nachricht noch auf ihre Empfangsbestätigung wartet. Es wird also eine Warteschlange zum Senden von Nachrichten benötigt, in die auch die Antwort auf die „Request“-Nachricht eingesortiert werden kann.

Zum Schreiben ausstehender Nachrichten muss einfach das erste Element dieser Warteschlange genommen und abgeschickt werden. Es muss auch vermerkt werden, dass die Nachricht gerade verschickt ist, damit keine weiteren Nachrichten versandt werden, bis eine Empfangsbestätigung eingegangen ist.

Es steht nun nur noch offen, wie die Eingaben der Nutzer:innen zum Senden einer Nachricht beihelfen. Der UI-Codecycle verarbeitet die Nutzer-Eingaben und muss dem Wiedergabe-Codecycle mitteilen, dass eine neue Nachricht zu senden ist. Dafür muss diese neue Nachricht ans Ende der Warteschlange eingefügt werden.

Zuletzt kann die Verbindung der beiden Codecycle auch noch erweitert werden. So ist es beispielsweise sinnvoll anzugeben, ob eine Verbindung zum μ C gefunden wurde. Auch kann der Fortschritt auf der Wiedergabeleiste relativ genau angegeben werden. Auch kann dem UI-Codecycle über z.B. geteilte Variablen mitgeteilt werden, wann eine Nachricht erfolgreich vom μ C empfangen wurde, um die UI mit dem Stand des μ Cs synchron zu halten.

6 Umsetzung - Software

Valentin Richter

Nachdem alle Software-Komponenten spezifiziert waren, mussten diese umgesetzt werden. Die Entwicklung der Software lief größtenteils parallel zur Umsetzung des „Piano Player“s. Gegen Ende der Entwicklungsphase, als genug Teile der Schaltung standen, wurden die Ausgaben des μ Cs beim Spielen von Musikstücken über LEDs getestet. In dieser Phase wurden viele Fehler gefunden und behoben, als müsste die Wichtigkeit von Tests nochmal unterstrichen werden.

Bevor auf die Umsetzung der Komponenten speziell eingegangen wird, soll hier kurz die Wahl der Programmiersprache der Implementierung motiviert werden.

Es wurde sich entschieden, die gesamte Implementierung der Software in der Programmiersprache C¹ zu erstellen. Da der μ C vergleichsweise nur sehr begrenzten Speicherplatz zur Verfügung hat, ist die manuelle Speicherverwaltung von C hier sehr wünschenswert. Weiterhin ist die Sprache sehr simpel und elegant und erzwingt keinen Programmierstil, wie funktionale oder Objekt-orientierte Sprachen es oft tun.

Zuletzt erleichtert die Verwendung von C die direkte Nutzung der Windows-API, da die Verwendung eines Foreign Function Interface (FFI)s ausbleibt. Das hat sich vor allem bei der Entwicklung der Kommunikation (siehe Abschnitt 6.2) als sinnvoll erwiesen. Zuletzt ist auch noch anzumerken, dass ein substantiver Beweggrund für die Wahl von C in der hohen Erfahrung der Entwicklerin mit der Sprache lag.

Folgend sollen besonders interessante Teile der Entwicklung noch genauer betrachtet werden. Abschnitt 6.1 geht dabei auf die Entwicklung von SAM ein, während Abschnitt 6.2 Schwierigkeiten bei der Umsetzung der Kommunikation und Abschnitt 6.3 Herausforderungen bei der Programmierung des μ Cs beschreibt.

¹Streng genommen ist hier der ISO-Standard C99 gemeint; siehe [ISO99] für die Spezifikation. Der Einfachheit halber wird in dieser Arbeit jedoch nur von „C“ gesprochen.

6.1 Desktop Anwendung

Die Umsetzung der Anwendung SAM wurde anhand der in Kapitel 5.4.2 ausgearbeiteten Architektur erledigt. Die Anwendung konnte aufgrund der begrenzten Zeit leider nicht vollständig umgesetzt werden, ist für die beiden Haupt-Use-Cases aber funktionsfähig.

Das heißt, dass neue Musikstücke zum Katalog hinzugefügt werden können und alle Musikstücke im Katalog wiedergegeben werden können. Auch die Umsetzung der Kontroll-Elemente zum Pausieren/Fortfahren der Musik und der Anpassung der Lautstärke und Wiedergabegeschwindigkeit wurden implementiert und getestet. Zuletzt wurde sogar die Navigation im Musikstück umgesetzt.

Somit wurden alle Anforderungen an die Anwendung mit mittlerer oder hoher Priorität umgesetzt (Anforderungen A9-A12 in Tabelle 2.1). Nur die Anforderung A13 wurde nicht umgesetzt, das heißt, dass Musikstücke im Katalog zurzeit nicht umbenannt oder entfernt werden können.

Weiterhin gibt es bei den erfüllten Anforderungen zu Teilen Limitationen, die hier kurz benannt werden sollen.

Die MIDI-Integration, Anforderung A12, scheint noch einige Fehler aufzuweisen, auch wenn es bei den meiste getesteten MIDI-Dateien die erwarteten PIDI-Dateien ausgibt. Zum Testen der Integration, wurde ein Skript geschrieben, dass aus einer PIDI-Datei wieder eine MIDI-Datei erstellt. Da viele Metadaten absichtlich bei der Transformation ins PIDI-Format verloren gehen, kann die neue MIDI-Datei nicht einfach mit der alten MIDI-Datei verglichen werden. Stattdessen werden bei dem Test beide Dateien abgespielt, um zu hören, ob es Unterschiede in den Dateien gibt. Diese Methode findet subtile Fehler zwar nicht zuverlässig, das ist aber kein größeres Problem, da subtile Fehler auch die Qualität beim Spielen über den „Piano Player“ nur gering mindern.

Bei der MIDI-Integration wurde weiterhin eine Entscheidung getroffen, die den endgültigen Klang stark beeinflusst und weder unbedingt korrekt noch falsch ist.

MIDI-Dateien können mehrere Instrumente in einem Stück haben, wobei keins der Instrumente ein Klavier sein muss.

Da es den geringsten Aufwand mit sich brachte und sinnvoll schien, wurde sich bei der Entwicklung entschieden, alle Instrumente zusammen zu nehmen und auf dem Klavier spielen zu lassen, unabhängig davon, ob es originell von einem Klavier gespielt wurde. Wenn aus einem Orchester-Stück nur der Teil des Pianos vom „Piano Player“ gespielt werden sollte,

ist das hier nicht möglich.

Eine mögliche Verbesserung würde hier den Nutzer:innen mehrere Möglichkeiten anbieten, um nur bestimmte oder alle Instrumente aus der MIDI-Datei zu übernehmen.

Außerdem ist noch zu erwähnen, dass das UI-Design des Prototypen nicht besonders ansprechend ist. Die Abbildungen 6.1, 6.2, 6.3, 6.4 zeigen das derzeitige Aussehen von SAM.

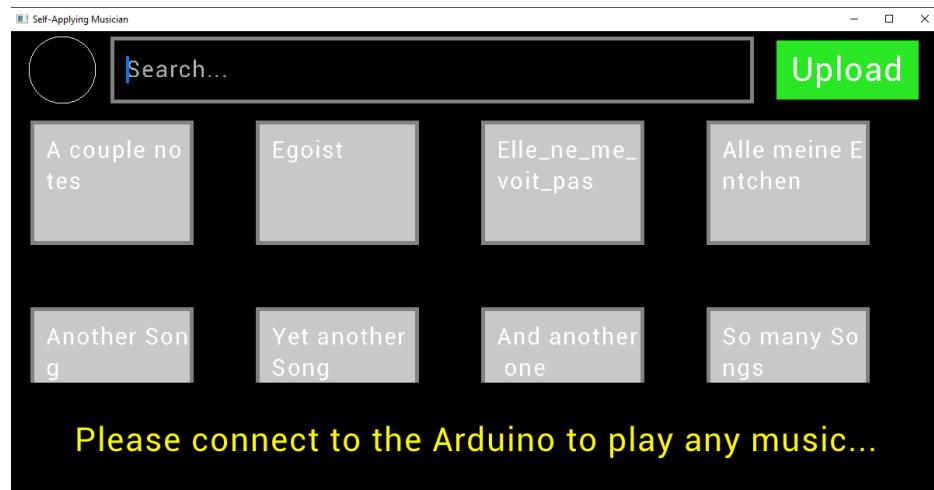


Abbildung 6.1: Screenshot der Startseite in der Desktop-Anwendung

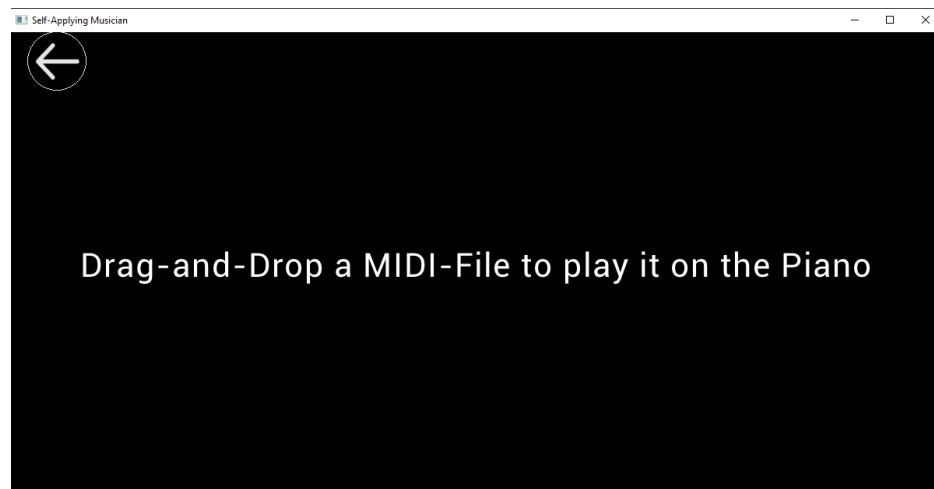


Abbildung 6.2: Screenshot der Import-Seite in der Desktop-Anwendung



Abbildung 6.3: Screenshot der Umbenennung beim Import in der Desktop-Anwendung

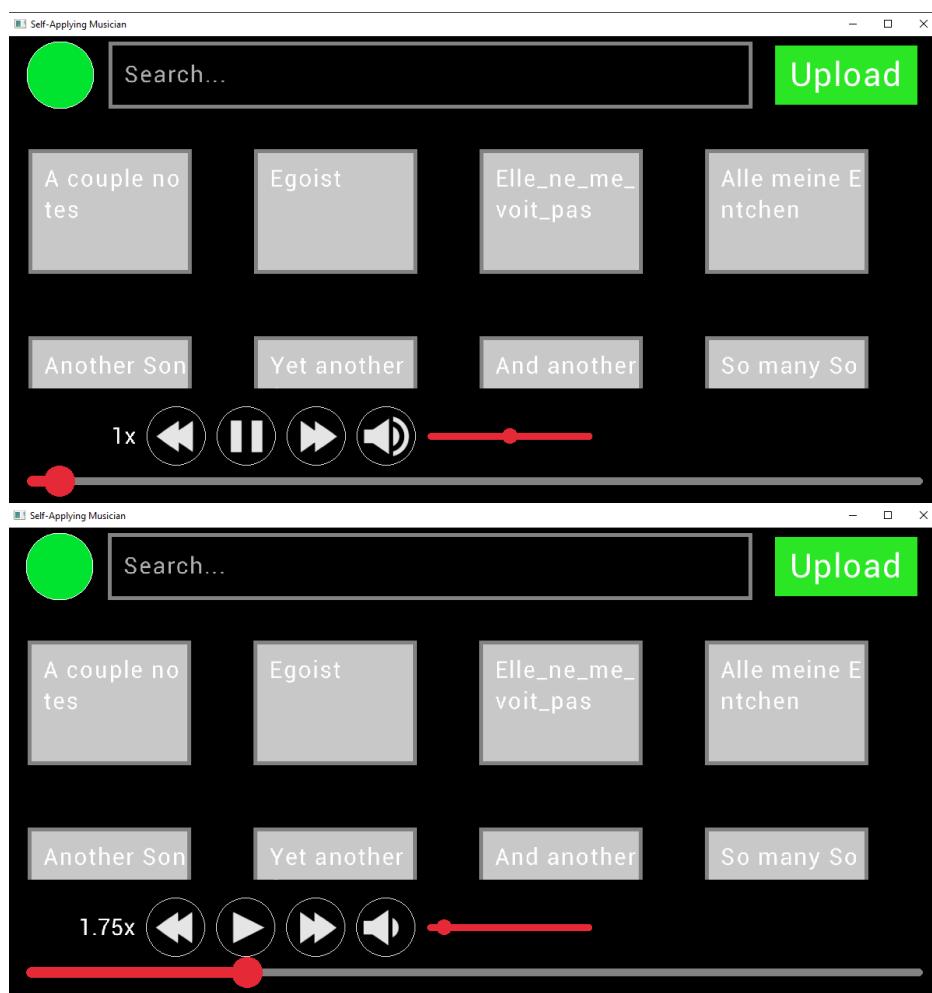


Abbildung 6.4: Screenshots der Wiedergabe in der Desktop-Anwendung

6.2 Kommunikation

Die Protokolle der Kommunikation wurden in Kapitel 5.3 detailliert spezifiziert. Die Umsetzung des Lesens und Schreibens von Nachrichten braucht allerdings sowohl auf Seiten der Desktop Anwendung als auch beim μ C mehr als nur eine Möglichkeit zum De- und Enkodieren der Nachrichten. Die Kommunikation muss nämlich in den Kontext beider Systeme eingebunden werden. Im Folgenden soll darauf eingegangen werden, welche Schwierigkeiten und Auswirkungen diese Kontexte auf die Umsetzung der Kommunikation hatten. Zuerst soll in Abschnitt 6.2.1 die Umsetzung auf dem μ C und dann in Abschnitt 6.2.2 die Umsetzung in SAM beschrieben werden. Trotz einiger Unterschiede gilt in beiden Fällen, dass die Kommunikation im Kontext eines Echtzeit-Systems implementiert werden muss.

6.2.1 Kommunikation auf dem Mikrocontroller

Der gewählte Arduino hat einen einzelnen Kern und kann keinen Code parallel laufen lassen. Das Spielen der Musik und das Lesen bzw. Schreiben von Nachrichten an SAM müssen entsprechend nach einander abgehandelt werden. Der Ablauf fürs Spielen der Musik wurde in Abschnitt 5.2 bereits dargestellt. Dem SPPP Protokoll nach muss der μ C einkommende Nachrichten lesen und je nach Nachricht eine Antwort senden. Auch kann es zu Zuständen kommen, in denen der μ C ohne Lesen einer einkommenden Nachricht von sich aus eine „Request“-Nachricht an SAM schicken muss.

Eine naive Implementierung dieser Logik tritt jedoch bald in Probleme. Da die Bytes kontinuierlich nacheinander eintreffen, müssen alle einkommenden Bytes in einem Buffer gespeichert werden, bevor sie als SPPP Nachricht interpretiert werden können. Die Standard-Bibliothek des Arduino Projekts bietet dieses Buffering für das Lesen von Daten über das UART Protokoll bereits an. Jedoch ist dieser Buffer nicht ausreichend, da er zu klein ist und nach gewisser Zeit wieder von vorne überschrieben wird.

Deshalb muss ein eigener Buffer erstellt werden, der groß genug für eine SPPP Nachricht und nicht überschrieben wird. Der Buffer sollte jede Iteration mit den neu gelesenen Bytes befüllt werden. Sobald genug Bytes im Buffer gesammelt wurden, kann versucht werden, diese als SPPP Nachricht zu interpretieren.

Hier kann vorkommen, dass eine korrekte Nachricht eingetroffen ist, aber dass direkt davor erst noch unsinnige Bytes angekommen sind. Es wäre schlecht, die gesamte Nachricht

deswegen wegzuschmeißen. Stattdessen sollten nur solange die vorderen Bytes ignoriert werden, bis die Magic Bytes des SPPP Protokolls gefunden wurden. Um effizient von vorne durch den Buffer zu gehen, während weiterhin jede Iteration neue Bytes hinten angehängt werden können, ist die Verwendung eines Ring-Buffers empfehlenswert.

Es wurde nun gesagt, dass die Nachricht geparst werden kann, sobald genug Bytes im Buffer gelandet sind. Hier lässt sich aber die Frage stellen, was genug Bytes sind. Eine Nachricht besteht aus mindestens 4 Bytes - 3 Magic Bytes und 1 Byte für die ID des Nachrichtentyps. Die längste mögliche Nachricht des Protokolls dagegen kann bis zu 262.727 Bytes enthalten.² Es ist dadurch nicht möglich zu warten, bis die maximale Anzahl von Bytes pro Nachricht angekommen sind. Einerseits hat der gewählte Arduino nicht genug Speicherplatz für einen Buffer dieser Länge und andererseits würde dies die Antwortzeit des μ Cs auf nicht akzeptable Höhen steigern.

Stattdessen muss nach dem Eintrffen von mindestens 4 Bytes geschaut werden, um welchen Nachrichtentyp es sich handelt. Je nach Typ ist die gesamte Nachricht dann schon gelesen. Bei Nachrichten einer fixen Länge kann gewartet werden, bis alle Bytes der Nachricht eingetroffen sind. Bei der „Music“- und „New-Music“-Nachricht dagegen, kann aufgrund geringen Speicherplatzes nicht unbedingt gewartet werden, bis die gesamte Nachricht eingetroffen ist. Eine alternative Lösung muss entsprechend „greedy“ sein und die empfangenen Bytes der PidiCmds und PlayedKeys direkt auslesen und in die zur Verfügung stehenden Buffer für die nächsten zu spielenden Noten einsortieren. Um dies umzusetzen, muss der derzeitige Zustand gespeichert werden, damit sich nach jeder Iteration gemerkt wird, ob die nächsten Bytes einen PidiCmd, einen PlayedKey, den Start einer neuen Nachricht oder etwas anderes darstellt.

²Die längste mögliche SPPP-Nachricht kann mit einer „New-Music“-Nachricht erstellt werden. Diese Nachricht ist zwei variabel lange Listen begrenzt. Einerseits übersendet die Nachricht eine Liste von PidiCmds, welche jeweils 4 Byte einnehmen.. Die Länge dieser Liste wird in der „Pong“-Nachricht vom μ C begrenzt, wobei die Maximallänge als 16-Bit Zahl repräsentiert werden muss. Entsprechend können höchstens $2^{16} = 65.536$ PidiCmds pro Nachricht geschickt werden. Andererseits enthält die Nachricht auch eine variabel-lange Lise von PlayedKeys, welche jeweils in 3 Bytes repräsentiert werden. Da jede Taste nur einmal vorkommen darf, ist diese Länge über die Anzahl Tasten begrenzt, die im PIDI-Format repräsentiert werden können. Da es 12 Tasten pro Oktave und $2^4 = 16$ Oktaven im Format gibt, können bis zu $12 \cdot 2^4 = 192$ PlayedKeys übersendet werden. Zusätzlich besteht die „New-Music“-Nachricht noch aus den 3 Magic Bytes des SPPP Protokolls, 1 Byte für die ID des Nachrichtentyps, 1 Byte für die Länge der Liste von PlayedKeys und 2 Bytes für die Länge der Liste von PidiCmds. Insgesamt ergibt das dann $4 + 1 + 12 \cdot 2^4 \cdot 3 + 2 + 2^{16} \cdot 4 = 262.727$ Bytes.

6.2.2 Kommunikation auf der Desktop Anwendung

Es gibt keine standardisierte, Betriebssystem-übergreifende API zum Empfangen und Senden von Daten über das UART Protokoll. Da SAM den Anforderungen nach nur auf Windows laufen muss, wird hier auf die Windows-spezifische API dafür zurückgegriffen. Um dies stattdessen cross-platform zu entwickeln, müsste eine eigene Abstraktion über die Platform-spezifischen APIs für die UART Kommunikation gefunden werden. Fortführungen dieser Arbeit sollten den Aufwand dieser Erweiterung prüfung und ggf. umsetzen.

Auf Windows werden sogenannte COM Ports für die serielle Kommunikation über das UART Protokoll verwendet [vgl. Mar22]. Zum Lesen oder Schreiben werden diese Ports als Dateien betrachtet und die Funktionen „ReadFile“ und „WriteFile“ verwendet [vgl. Den10, Kapitel 4; Bay08, S. 3]. Alternativ bietet Windows auch asynchrone Versionen der Funktionen an, die den laufenden Thread nicht bis Abschluss der Input/Output (I/O)-Operation blockieren [vgl. Den10, Kapitel 4].

Das Lesen von COM Ports kann entweder in einer Endlosschleife wiederholt werden, um alle eingehenden Bytes zu erhalten, oder es kann ein sogenannter „I/O Completion Port“ verwendet werden, um über das Eintreffen von Bytes von Windows informiert zu werden [vgl. Ash22]. Da diese Funktionalität erst sehr spät in die Entwicklung gefunden wurde, wurde sie weder verwendet noch getestet.

Die naive Implementierung des Lesens von PPP-Nachrichten bringt mit hier ebenfalls Probleme mit sich. Aufruf zum Lesen von n Bytes blockt bis n Bytes gelesen wurden oder bricht mit einer Fehlermeldung ab. Angenommen der Aufruf der „ReadFile“ Funktion läuft in einem separaten Thread, sodass das Blockieren nicht die Interaktivität der UI beeinflusst, so bleibt trotzdem noch ein Problem. Angenommen es sollen genau 4 Bytes gelesen werden, da auf eine „Request“-Nachricht gewartet wird. Nun soll angenommen werden, dass zuerst ein zufälliges Byte gelesen wird, das nichts mit der Nachricht zu tun hat, bevor die Nachricht empfangen wird. Das erste Byte soll in diesem Fall ignoriert werden, damit die Nachricht gelesen werden kann. Wenn aber genau 4 Bytes gelesen werden, ist dies ohne das Zwischenspeichern der empfangenen Bytes in einem zusätzlichen Buffer nicht möglich.

Genau wie beim Arduino ist hier also die Verwendung eines Buffers zum Speichern der empfangenen Bytes unabdingbar. Aus den selben Effizienzgründen wird hier ebenfalls ein Ring-Buffer verwendet. Dem Ansatz der Semantic Compression nach, wurde hier der duplizierte Code wiederverwendbar zusammengeführt. Um Code zwischen Arduino und SAM

zu teilen, wurde eine Bibliothek erstellt, die von beiden inkludiert wird.

Zum Empfangen der Bytes muss weiterhin die „ReadFile“ Funktion verwendet werden. Hier kann sowohl die blockierende als auch die nicht blockierende Version der Funktion verwendet werden. Die nicht blockierende Version benötigt zusätzlichen Code zum regelmäßigen Nachschauen, ob Bytes angekommen ist. Die blockierende Version muss dagegen im einen separaten Thread laufen, um die restliche Anwendung nicht zu blockieren. Da nur dieser Thread den Ring-Buffer befüllt und der restliche Code nur aus dem Ring-Buffer liest ist eine Synchronisierung der Threads über Mutexe oder ähnliche Konstrukte nicht nötig. Aufgrund des geringeren Entwicklungsaufwands wurde sich für die Nutzung der blockierenden Version entschieden.

Bei der naiven Implementierung des Schreibens ist ebenfalls ein vorher unerwartetes Problem aufgetreten. Beim Schreiben kurzer Nachrichten hat ein einfacher Aufruf der „WriteFile“ Funktion mit der encodierten Nachricht ausgereicht. Beim Senden längerer Nachrichten - speziell bei „Music“ und „New-Music“ Nachrichten - wurden dagegen nicht alle Bytes vom Arduino empfangen. Es gibt mehrere Stellen, an denen der Fehler auftreten könnte. Es könnte sein, dass der μ C nicht schnell genug mit dem Lesen der einkommenden Bytes ist und deshalb einige Bytes überschreibt und überspringt. Es ist auch möglich, dass Windows ein Bug beim Senden längerer Byte Arrays hat oder eine andere, nicht dokumentierte Nutzung der „WriteFile“ Funktion erwartet.

Da dieses Problem erst relativ spät in der Entwicklung gefunden wurde, konnte die Quelle des Problems nicht mehr gefunden und beseitigt werden. Dies liegt zu großen Teilen daran, dass dieser Bug scheinbar nicht-deterministisch auftrat. Auch wenn nur große Nachrichten das Problem hatten, trat der Bug nicht bei allen längeren Nachrichten auf. Wäre noch genügend Zeit für Projekt übrig gewesen, wäre der nächste Schritt im Identifizieren des Problems gewesen. Um zu sehen, ob empfangene Bytes auf Seiten des Arduino überschrieben werden, könnte die verwendete Standard-Bibliothek um einen konditionalen Debug-Output erweitert werden. Sollte dies die Problemquelle sein, könnte sie behoben werden, indem zum Beispiel ein größerer Buffer zum Zwischenspeichern der eintreffenden Bytes verwendet wird. Sollte sich herausstellen, dass es auf Seiten des Arduinos keinen Fehler gibt, müsste stattdessen untersucht werden, was genau beim Aufruf der „WriteFile“ Funktion passiert. Da der Quellcode von Windows nicht frei verfügbar ist, würde dies einen hohen Aufwand mit sich bringen.

Wie bereits erwähnt, konnte diese Fehlersuche jedoch nicht vollzogen werden. Nichtsdestotrotz konnte das Problem noch gelöst werden. Experimentell wurde herausgefunden, dass

der Fehler nicht auftritt, wenn höchstens 16 Byte auf einmal mit der „WriteFile“ Funktion gesendet werden und ein paar Millisekunden gewartet wird, bevor die nächsten 16 Byte der Nachricht gesendet werden. Diese Lösung funktioniert zwar, ist aber nicht unbedingt stailö Da der Grund für den Fehler nicht bekannt ist, ist unklar warum diese Lösung genau funktioniert und unter welchen Umständen sie nicht mehr funktionieren würde. Fortsetzungen dieser Arbeit sollten deshalb die Fehlerquelle finden und sauberer lösen.

6.3 Arduino-Programmierung

Aufgrund der großen Unterschiede zwischen einem Desktop und einem μ C, war auch die Programmierung für die beiden Systeme relativ unterschiedlich. Der größte, spürbare Unterschied ist wahrscheinlich im stark begrenzten Speicherplatz zu finden. Während der Arduino ausreichend Flash-Speicher besitzt, ist der dynamische RAM-Speicher mit 2KB stark begrenzt. Mit der Verwendung der offiziellen Arduino IDE wird mitgeteilt, wie viel des dynamischen Speichers verwendet wird. Dabei kann jedoch nicht berechnet werden, wie viel Speicherplatz während der Laufzeit durch lokale Variablen noch benötigt wird. Es kann dadurch vorkommen, dass mehr Speicher benötigt wird als verfügbar ist. In diesen Fällen wird bereits verwandelter Speicher überschrieben, was zu viel nicht erwünschten und nicht erwartbaren Verhalten führt. Es ist mehrfach während der Entwicklung vorgekommen, dass unerklärliche Fehler aufgetreten sind, die mit einer verringerten Speichernutzung gefixt wurden. Um einen besseren Überblick über die Anzahl wirklich verwendeten Speichers zu erhalten, wurden viele lokale Variablen zu globalen Variablen geändert. Dadurch taucht deren Speicherplatz-Nutzung in der beim Kompilieren gezeigten Statistik verbleibenden Speichers mit auf.

Weiterhin ist die Programmierung des Arduinos durch das Ausbleiben eines Debuggers erschwert. Während es durchaus Debugger für den Arduino gibt, benötigen diese ebenfalls eine gewisse Menge dynamischen Speichers. Zum Finden von Problemen, die durch Speicherplatzmangel entstanden sind, sind die Debugger entsprechend ungeeignet. Aber auch beim Finden anderer Bugs sind sie nicht weiter hilfreich, da sie selbst den Speicher zum Überlaufen bringen können. Nichtsdestotrotz konnten Bugs über die Nutzung von Print-Statements gefunden und ausgemerzt werden.

Mit Ausnahme dieser teilweise schwer zu lösender Bugs, konnte das in Kapitel 5.2 beschriebene Konzept relativ leicht und nahezu 1-zu-1 in Code umgesetzt werden.

Es wurden hier auch mehrere Strategien für das Eliminieren von PlayedKeys ausprobiert. Hier hat sich herausgestellt, dass eine Eliminierung basierend auf der Spiellänge allein die besten Ergebnisse geliefert hat.

7 Ergebnisse

Jakob Kautz, Olivier Stenzel, Valentin Richter

Nachdem sowohl Hardware als auch Software konzipiert und zu großen Teilen umgestzt wurden, soll nun der entstandene Prototyp abgeschlossen werden. Dafür wird zunächst ein Kurzüberblick über den gesamten Prototypen gegeben. In Abschnitt 7.1 werden dann mehrere Tests und Messungen am Prototypen ausgeführt. Die Ergebnisse der Tests zeigen einerseits die Fähigkeiten und Limitationen des Prototypen auf, können zu Teilen aber auch verwendet werden, um bestehende Probleme der Umsetzung zu lösen. Zuletzt wird in Abschnitt 7.2 noch ein Überblick darüber gegeben, was beim Prototypen noch fehlt, um ihn zu einem vollständigen Produkt zu bringen. Hier werden nicht nur Limitationen der Umsetzung, sondern auch der Konzeption betrachtet, welche in Fortführungen oder Replikationen dieser Arbeit beachtet werden sollten.

Obwohl Software und Hardware getrennt von einander entwickelt wurden, war es aufgrund einer klar definierter Schnittstelle einfach, die beiden zusammen zu führen. Nichtsdestotrotz gab es bei der Zusammenführung einige Probleme, die in einem letzten Schritt der Implementierung ausgemerzt wurden. Speziell hat sich das Testen der Software des μ Cs ohne verfügbare Hardware als schwierig erwiesen. Diese Tests mussten entsprechend in diesem letzten Schritt durchgeführt werden und alle dabei auftretenden Fehler mussten gefunden und beseitigt werden. Der in Kapitel 6.2.1 vorgestellte Ansatz war dabei korrekt, jedoch gab es mehrere kleinere Bugs in den Details. Der in Anhang B beigelegte Quellcode hat beim Testen mit der prototyp-haften Hardware keine Probleme mehr aufgezeigt.

Der Prototyp besteht aus der Hardware, die von einer Desktop-Anwendung aus gesteuert werden kann, um Klaviertasten des Pianos anzuspielen. Aufgrund zeitlicher Limitationen war es nicht möglich, das gesamte Piano bespielbar zu gestalten. Nichtsdestotrotz ist der Prototyp mit diesem Ziel entwickelt worden und ist somit leicht erweiterbar, um das gesamte Klavier anzusteuernd. Die Anwendung SAM leidet ebenso an Defekten, vor allem im Bereich des UI Designs und der Usability. Allerdings ist die Anwendung funktional und kann Musikstücke aus einem Katalog - der von Nutzer:innen mit MIDI-Dateien erweitert werden kann - zum μ C schicken und abspielen lassen.

7.1 Tests & Messungen

Olivier Stenzel, Valentin Richter, Jakob Kautz

Mit der Fertigstellung des Prototypen, soll dieser nun getestet werden. Dabei sind die Messungen in zwei Teile geteilt. Zuerst werden Messungen der Hardware beschrieben (siehe Abschnitt 7.1.1). Deren Ergebnisse haben teilweise Auswirkungen auf die Software, um eine korrekte Funktionalität zu versichern. Daraufhin werden Messungen der Software in Abschnitt 7.1.2 dargestellt.

Es wurden auch Tests durchgeführt, um die Korrektheit des Prototypen zu versichern. Diese Tests liefen größtenteils jedoch informell durch einfaches Benutzen der Komponenten bzw. des gesamten Prototypen ab und sollen hier nicht genauer geschildert werden. Es sollte jedoch angemerkt werden, dass der Prototyp alle Tests auf Korrektheit bestanden hat.

7.1.1 Hardware-Tests

Olivier Stenzel, Valentin Richter, Jakob Kautz

Tabelle 7.1 listet die unterschiedlichen Messungen auf, die an der Hardware durchgeführt wurden. In den folgenden Abschnitten wird der Ablauf, die Bedeutung und das Ergebnis jeder Messung ausführlich beleuchtet. Anpassungen der Software, die aufgrund von Testergebnissen durchgeführt wurden, werden bei den jeweiligen Tests ebenfalls spezifiziert.

ID	Name
HW-T1	Min. Periodendauer eines Tastendrucks
HW-T2	Min. Lautstärke
HW-T3	Min. Genauigkeit

Tabelle 7.1: Hardware-Tests

HW-T1: Min. Periodendauer eines Tastendrucks

Erklärung: Mit der Periode eines Tastendrucks ist hier jene Zeit gemeint, die zwischen Loslassen und erneutem Anspielen einer Taste vergeht. Da der Hubmagnet, bevor

er einen erneuten Tastendruck initiieren kann, wieder in seinen Ausgangszustand zurückkehren muss, kann diese Periode nicht unendlich reduziert werden, da es sonst nur zu einem konstanten Spielen der Taste kommt. Es soll sichergestellt werden, dass alle Töne des Stücks erklingen, weshalb das Erfassen der minimalen Periodendauer bedeutsam ist. Software-seitig kann diese Periodendauer genutzt werden, um zu verhindern, dass ein solches „Verschmelzen“ von zwei Anschlägen zu einem konstanten Anspielen vorkommt.

Ablauf: Schrittweise wird software-seitig die Frequenz des Anspielens einer Taste erhöht. Es werden mehrere Tasten für diesen Test verwendet, wobei der Fokus, auf Grund des höchsten Mechanik-Gewichts, auf der tiefsten Taste - der A0 Taste - liegt. Während des Spielens wird eine Slow-Motion-Aufnahme des Hammers und der Klaviersaite angefertigt und im Anschluss analysiert. Wird die Saite angeschlagen, kann die Frequenz im nächsten Schritt erhöht werden. Sollte sie nicht angeschlagen werden, ist die maximal verwendbare Frequenz (und damit auch die minimale Periodendauer) gefunden und kann in den Code aufgenommen werden.

Ergebnis: Die minimale Periodendauer liegt durchschnittlich bei den Tasten bei ca. 30ms. Das höchste Minimum wurde wie erwartet bei der A0 Taste mit ca. 40ms gefunden. Software-seitig gibt es nun mehrere Möglichkeiten, dieses Minimum zu forcieren. Zum Beispiel könnte im PIDI-Format vorgegeben werden, dass zwischen zwei Anschlägen eines einzigen Tons immer min. 40ms liegen müssen. Da das PIDI-Format aber unabhängig der Hardware sein soll und diese Begrenzung speziell durch die verwendeten Komponenten aufkommt, ist dies eine eher unschöne Lösung.

Stattdessen wäre es besser wenn nur der μ C diese Limitierung beachten muss, da nur dieser Wissen über die spezifische Hardware-Umstzung verwenden soll. Der μ C muss entsprechend Tasten finden, die so lange gespielt werden sollen, dass ihr nächstes Anspielen weniger als 40ms später erfolgt. Wenn diese Tasten gefunden wurde, könnte entweder das nächste Anspielen der Taste später erfolgen oder die Spiellänge der Taste reduziert werden. Da das Anspielen einer Taste sehr klarer zu erkennen ist als das Weiterspielen einer Taste, beeinflusst die zweite Strategie den Klang des Musikstücks geringer.

Da der μ C allerdings immer nur Teile des Musikstücks erhält, kann nicht einfach zu Spielbeginn das gesamte Stück angepasst werden, um die Periodendauer von min. 40ms einzuhalten. Stattdessen müssen die PlayedKeys, die der μ C sich speichert, dynamisch während

Spielen des Musikstücks angepasst werden. Dafür wird durch die nächsten PidiCmds iteriert, die in den nächsten 40ms gespielt werden sollen. Falls eine dieser Tasten bereits gespielt wird, muss die Taste jetzt schon aufhören zu spielen.

Dieser Ansatz ist sowohl leicht zu implementieren, als auch performat in der Ausführung. Allerdings kommt diese Strategie auch mit Nachteilen. Soll eine Taste beispielsweise für 30ms gespielt werden und dann 5ms später wieder angespielt werden, so wird das erste Anspielen übersprungen. Ein alternativer Ansatz würde in diesen Fällen das Verschmelzen der Töne in Kauf nehmen. Das würde jedoch eine etwas kompliziertere Lösung mit sich bringen und es da diese Fälle in der Praxis eher unwahrscheinlich sind, wurde Einfachheit hier vorgezogen.

Eine größere Limitation dieser Lösung liegt dagegen darin, dass zurzeit nur die nächste PidiCmds im derzeit zu spielenden Buffer betrachtet werden. Der zweite Buffer von PidiCmds, der beim Lesen neuer „Music“-Nachrichten gefüllt wird, wird bei dieser Prüfung zurzeit ignoriert. Das bedeutet, dass die derzeitige Umsetzung des Ansatzes nicht vollständig korrekt ist, da das nächste Anschlagen eines Tons im nächsten Buffer liegen könnte. Allerdings wurde hier wieder die Einfachheit der Implementierung als wichtiger gesehen, da diese Fälle in den praktischen, getesteten Musikstücken nicht oder nur sehr selten aufgetreten sind.

HW-T2: Min. Lautstärke

Erklärung: Um den leisensten spielbaren Ton zu ermitteln, muss die minimale Spannung ermittelt werden, mit der eine Taste betätigt werden kann. Wie in Kapitel „Aktuator“ erklärt, benötigt ein Hubmagnet eine entsprechende Minimalspannung, um den Anker zu bewegen. Da diese Spannung im Datenblatt des Hubmagneten nicht ersichtlich ist und das Tastengewicht sowieso mit einzogen werden muss, muss diese Mindestspannung beim Klavier experimentell ermittelt werden.

Ablauf: Software-seitig wird der Wert, der die Intensität des Anschlags beschreibt, schrittweise soweit reduziert, bis die Taste keinen Ton mehr spielt. Dies wird an allen (mit Fokus auf den tieferen) Tasten durchgeführt, da sich die einzelnen Tasten in ihrem Ansprechverhalten teils deutlich unterscheiden. Anschließend wird wieder das Größte der einzelnen Minima in den Code mit aufgenommen.

Ergebnis: Das leisteste Anspielen der Tasten ist mit einer Spannung von 17,3V möglich. Als PWM-Wert ist dies eine 210 von 256 möglichen Werten. Der μ C hat beim Bespielen von Noten eine Anspielstärke als 4-Bit Zahl gegeben (siehe Listing 5.1). Um die Anschlagestärke auf die richtige Skalierung zu bringen kann eine lineare Interpolation verwendet werden. Listing 7.1 zeigt, wie das in Code aussehen könnte. Dabei werden im Listing auch der Lautstärke-Faktor beachtet und Anschlagsstärken von 0 korrekt auf 0 gesetzt.

```

1 #define DELTA 0.00001f
2 #define MAX_VELOCITY 16
3 #define MIN_PWM 210
4 #define MAX_PWM 255
5 unsigned char velocity; // is assumed to be given as a number between 0
   and 15
6 float volume_factor;    // is assumed to be given as some positive
   number
7 float t = volume_factor * velocity / (float)MAX_VELOCITY;
8 if (t <= DELTA) {
9     set(0); // Sets the specified key to receive a PWM value of 0
10 } else {
11     if (t > 1.0f) t = 1.0f;
12     set(MIN_PWM + t*(MAX_PWM - MIN_PWM));
13 }
```

Quelltext 7.1: Lautstärke-Skalierung

HW-T3: Min. Genauigkeit:

Erklärung: Der Zweck dieses Tests besteht darin sicherzustellen, dass der Tastenan- schlag des selbstspielenden Klaviers zum richtigen Zeitpunkt und in der korrekten Reihen- folge erfolgt. Dies gewährleistet die Genauigkeit und Zuverlässigkeit des Klaviers bei der Wiedergabe von Musikstücken.

Ablauf:

1. Es wird eine einfache Folge von Signalen erstellt, die repräsentativ für eine musikalische Passage ist.
2. Das selbstspielende Klavier wird mit den erstellten Signalen versorgt, und die Wiedergabe wird mittels einer Videoaufnahme dokumentiert.

3. Der Test wird mehrmals durchgeführt, um die Konsistenz der Ergebnisse sicherzustellen.
4. Während der Videoaufnahme wird eine Uhr verwendet, um den Zeitpunkt jedes gespielten Tons zu markieren.
5. Nach Abschluss des Tests wird das aufgezeichnete Video analysiert, um festzustellen, ob jeder Ton zur erwarteten Zeit gespielt wurde.
6. Es werden Abweichungen zwischen den erwarteten und tatsächlichen Spielzeiten dokumentiert und bewertet.

Ergebnis: Da der Test aufgrund fehlender Zeit nicht durchgeführt werden konnte, werden hier lediglich die Erfolgskriterien aufgezählt:

1. Jeder Ton wird zum erwarteten Zeitpunkt gespielt, innerhalb einer akzeptablen Toleranzschwelle von 5ms
2. Die Reihenfolge der gespielten Töne entspricht exakt der vorgegebenen Signalabfolge.
3. Die Abweichungen zwischen den erwarteten und tatsächlichen Spielzeiten (Dauer) liegen innerhalb einer Grenze von 5ms

7.1.2 Software-Tests

Valentin Richter

Tabelle 7.2 listet die unterschiedlichen Messungen, die an der Software durchgeführt wurden. Diese Tests wurden dabei alle auf dem gewählten Arduino UNO R3 bzw. einem Desktop mit Windows 10, einer Intel i5-10310U CPU und einer Intel UHD Grafikarte durchgeführt. Um einen Normalverbrauch zu simulieren, wurden bei den Tests der Desktop Anwendung mehrere Hintergrundanwendungen geöffnet, jedoch während der Testung nicht direkt verwendet. Die Bedeutung dieser Messergebnisse wird im Folgenden beleuchtet.

SW-T1 und SW-T2 haben den selben Test durchlaufen, jedoch jeweils mit unterschiedlichen Baud-Raten. Die Baud-Rate gibt beim UART-Protokoll die Anzahl Bits, die pro Sekunde geschickt werden, an [vgl. Bak03, S. 22]. Während die Baud-Rate den Durchsatz an Daten angibt, muss die Latenz separat berechnet werden. Nichtsdestotrotz bedingt auch

ID	Beschreibung	Ergebnis
SW-T1	Durchschnittliche Latenz bei der Kommunikation zwischen SAM & μ C bei einer Baud-Rate von 9600	ca. 1.75ms
SW-T2	Durchschnittliche Latenz bei der Kommunikation zwischen SAM & μ C bei einer Baud-Rate von 230400	ca. 0.9ms
SW-T3	Maximale Laufzeit einer Iteration des μ Cs während dem Abspielen eines Musikstücks	ca. 49ms
SW-T4	Minimale Laufzeit einer Iteration des μ Cs während dem Abspielen eines Musikstücks	ca. 12ms
SW-T5	Minimale Laufzeit einer Iteration des Wiedergabe-Codecycles in SAM während Abspielen eines Musikstücks	ca. 3ms
SW-T6	Maximale Laufzeit einer Iteration des Wiedergabe-Codecycles in SAM während Abspielen eines Musikstücks	ca. 31ms
SW-T7	Minimale Frames Per Second (FPS) von SAM während ausführlicher Nutzung	ca. 1000 FPS
SW-T8	Maximale FPS von SAM während ausführlicher Nutzung	ca. 52 FPS

Tabelle 7.2: Software-Tests

die Latenz eindeutig, da es die Geschwindigkeit, mit der Daten übertragen werden, beeinflusst.

Bei den Tests wurde jeweils wiederholt ein Byte vom Desktop zum Arduino geschickt und vom Arduino wieder zurück geschickt. Gemessen wurde dann die Zeit, die zwischen Senden und Wiedererhalten des Bytes auf Seiten des Desktops vergangen ist und durch zwei geteilt (da ja nur die Zeit zum Senden in eine Richtung erwünscht ist).

Es wurden hier auch mehrere Baud-Raten getestet, weil experimentell herausgefunden, dass es bei höheren Baud-Raten machmal zu Datenverlust kommt. Dieses Problem wurde in Kapitel 6.2.2 ein wenig genauer beleuchtet.

Da dieses Problem vor allem bei höheren Baud-Raten auftritt, ist stark zu vermuten, dass der Arduino an einem gewissen Punkt nicht mehr schnell genug mit dem Lesen von einkommenden Bytes ist und einige der zwischengespeicherten Bytes wieder überschreibt, bevor sie geparst werden konnten.

Ein weiteres Indiz dafür wird auch von den Ergebnissen der Tests SW-T3 und SW-T4 bereitgestellt. Hier wird nämlich gezeigt, dass der Arduino für eine einzelne Iteration seiner Endlosschleife bis zu 50ms braucht.

Während der μ C damit anscheinend zu langsam ist, um das vollständige Empfangen von SPPP-Nachrichten sicherzustellen, erreicht er damit immer noch die in Anforderung A5 genannten Performanz-Ziele zum Anspielen von Tönen.

Für eine Erweiterung des Prototypen sollte entsprechend überprüft werden, ob ein größerer Buffer zum Speichern einkommender Bytes das Problem bereits löst. Falls dies nicht der Fall sein sollte, müsste stattdessen versucht werden, die Performanz des μ Cs zu optimieren.

7.2 Limitationen & Erweiterungsmöglichkeiten

Valentin Richter, Jakob Kautz

Aufgrund limitierter Zeit konnte der hier erstellte Prototyp nicht vollständig umgesetzt werden. Aus den selben Gründen wurden auch bei den gestellten Anforderungen und der Konzeption des Projekts mehrere potenzielle Features ausgelassen. Auch während der Implementierung sind mehrere Probleme und Unzulänglichkeiten aufgetreten, die wegen der zeitlichen Begrenzung nicht mehr verbessert werden konnten. Zum Teil wurden diese Limitationen der Konzeption und prototypischen Umsetzung bereits erwähnt. Der Vollständigkeit halber sollen die größten Limitationen dieser Arbeit hier nochmal gelistet und kurz beschrieben werden.

Im Laufe der Konzeption traten mehrere Herausforderungen auf, welche aus Zeit- und Kosten-Gründen nicht weiter behandelt wurden.

Anzahl Aktuatoren: Während insgesamt 88 Aktuatoren für alle Klaviertasten bestellt wurden, hat die Zeit nur gereicht, um insgesamt 8 Hubmagnete mit der Schaltung zu verbinden. Für Testzwecke wurden allerdings bereits 40 von 88 LEDs mit der Schaltung verbunden. Das Hinzufügen der letzten 80 Aktuatoren ist trivial aber natürlich recht zeitaufwendig.

Maximum gleichzeitig spielender Tasten Die in dieser Arbeit verwendeten Hubmagnete benötigen jeweils etwa 0.7 Ampere. Das im Prototypen genutzte Netzteil ist auf 6 Ampere ausgelegt. Damit wäre es in diesem Aufbau möglich bis zu 8 Aktuatoren gleichzeitig anzusteuern. Da der Hauptanwendungsfall beim Spielen von Musik besteht, die für einzelne Pianos komponiert wurde, welche seltens mehr als 8 Tasten gleichzeitig bespielen müssen, sollte diese Limitation nicht oft in der Praxis relevant werden. Nichtsdestotrotz musste diese Limitation in der Software des μ Cs umgesetzt werden (siehe Abschnitt 5.2), um nicht zu viel Strom vom Netzteil zu ziehen.

Pedalansteuerung Ein Klavier hat im Regelfall zwei bis drei Pedale, welche die Dynamik und den Klang des Klaviers beeinflussen. Trotz des klangtechnischen Mehrwehrts, den Pedale bringen, wurden sie hier ignoriert. Um zusätzlich Pedale anzusprechen, müssten einige Teile des Konzepts erweitert werden. So bräuchte es mindestens zwei weitere PWM-Ausgänge, die über ein zusätzliches Schieberegister oder weiter PWM-Ports des Arduinos zu verbinden wären. Weiterhin müssten leistungsfähigere Aktuatoren als die hier verwendeten Hubmagnete zum Anspielen der Pedale benutzt werden, da die Pedale schwerer als die Tasten des Klaviers sind. Eine Methodik zum Anspielen der Pedale müssten ebenfalls noch überlegt werden. Zuletzt müsste auch die Software dafür erweitert werden. Hier müsste entweder ein Algorithmus entwickelt werden, um anhand des PIDI-Formats zu erkennen, wann das Pedal wie stark gedrückt werden soll, oder das PIDI-Format müsste um Pedal-Informationen erweitert werden.

Wärmeabfuhr Da die Hitzeentwicklung der Hubmagnete mit der Dauer der Nutzung stark ansteigt, ist die Holzplatte bezogen auf die Wärmeabfuhr nicht geeignet. Besser würde sich hier eine Metallplatte anbieten, auf der die Aktuatoren verschraubt werden. Aus Kostengründen wurde das beim Prototypen hier noch nicht umgesetzt.

Sicherheitskonzept Es besteht bei der Konzeption deer Hardware kein Sicherheitskonzept. Die Wärmeabfuhr wurde für den Bereich Temperaturkontrolle bereits genannt. Dazu kommt, dass auch Bauteile wie die MOSFETS bei dauerhafter Nutzung erhitzten werden. Es wäre eine Möglichkeit, die Bauteile mit Kühlkörpern zu versehen um den Temperaturanstieg zu vermeiden.

Es besteht außerdem keine Konzeption von Gehäusen um die Hardware zu schützen. Somit kann diese beschädigt werden. Außerdem sind auch die Stromverbindungen zu dem externen Stromanschluss von 24V frei zugänglich. Hier sollten Gehäuse entworfen werden, die die Schaltung schützen und es sollte ein Konzept zur Isolation entworfen werden.

Auch der Stromschutz ist nicht gegeben, da technisch gesehen jedes Netzteil, egal mit welcher Stromstärke, angeschlossen werden könnte und es keine Mechanismen gibt um den Überstrom zu behandeln. Hier sollten Sicherungen oder Leistungsschalter eingebaut werden, um die Brandgefahr zu verringern.

Eingabedateien Die Desktop-Anwendung SAM unterstützt nur den Import von MIDI-Dateien. Die Untertützung weiterer Formate wäre hier wünschenswert. Speziell das Hin-

zufügen von Dateien, die grundlegend andere Informationen speichern, wie beispielsweise Bild-Dateien von Notensheets oder generelle Audio-Dateien, wären hier von besonderem Interesse. Die Forschungsfelder „Optical Music Recognition“ und „Musical Note Recognition“ bieten hier respektiv Einblicke, wie Noten-Informationen aus solchen Formaten gewonnen werden könnten.

Vollständige MIDI Unterstützung Sogar das MIDI-Format selbst wurde für den Prototypen nicht vollständig unterstützt. Bestimmte MIDI-Dateien, die eine alternative Kodierung der Zeit verwenden, können beispielsweise nicht korrekt geparsst werden. Auch bei einigen anderen Dateien, wurde ein fehlerhaftes Parsing festgestellt. Aufgrund zeitlicher Begrenzungen, konnte nicht untersucht werden, woher diese Fehler kommen.

UI Design Das Design der UI ist nicht besonders ansprechend. Während in der Konzeption mehrere Darstellungsmöglichkeiten des Musik-Katalogs bedacht wurden, wurde in der Implementierung nur eine sehr unschöne Version umgesetzt (siehe Abbildungen 6.1 bis 6.4).

Musikverwaltung SAM erlaubt zwar das Hinzufügen von Musikstücken mit selbstgewähltem Namen, das spätere Umbenennen oder Löschen der Musikstücke fehlt allerdings. Auch bietet das System der Musikverwaltung zurzeit nur einen einzelnen Katalog an. Es wäre denkbar und womöglich sinnvoll, der Nutzer:in die Erstellung und Verwaltung von Alben zu erlauben, ähnlich wie es bei bekannten Musikplayern (z.B. WMP und Spotify) auch möglich ist. Mit der Existenz von Alben würde auch das Abspielen eines gesamten Albums und der Kontrolle über zufällige und wiederholte Wiedergabe des Albums ein attraktives Feature sein.

Fehlende Fehlermeldungen SAM hat kein System, um Fehlermeldungen anzuzeigen. Stattdessen wird bei unerwarteten Zuständen die Anwendung gewaltsam abgestürzt.

Datenkonsistenz von SPPP-Nachrichten Das SPPP-Protokoll hat keine eingebauten Mechanismen zum Sicherstellen der Datenkonsistenz. Es muss entsprechend darauf vertraut werden, dass das Übertragungsmedium die Nachrichten vollständig und korrekt überträgt. Die Nutzung von Parity Bits oder Ähnlichem würde die Datenkonsistenz stärker versichern.

8 Zusammenfassung

Jakob Kautz

8.1 Fazit

Die Arbeit umfasst eine erfolgreiche Konzeption und Entwicklung des selbstspielenden Pianos, wobei ein Großteil der in der Zielstellung definierten Anforderungen mit mittlerer oder höherer Priorität erfüllt werden konnten. In diesem Kapitel soll Rückblickend der Erfolg des Projektes insbesondere in Bezug auf die Ausgangsfrage evaluiert werden.

Vorab die Betrachtung der in Kapitel 2 definierten Anforderungen:

ID	Name	Status
A1	Flexibilität	Erfüllt: Die Aktuatoren werden von manuellen Tastendrücken nicht beeinflusst
A2	Benutzerinterface	Erfüllt
A3	Budget	Erfüllt, liegt insgesamt bei 1753,78€
A4	Responsivität	Erfüllt
A5	Performanz	Erfüllt: liegt durchschnittlich bei 30ms
A6	Spielbarkeit	Teils Erfüllt, da nicht alle Hubmagnete angegeschlossen wurden.
A7	Tastenbetätigung	Teilweise erfüllt: Die verbundenen Tasten können erfolgreich betätigt werden, wobei die Anzahl der Tasten auf 8 reduziert werden musste
A8	Anpassbarkeit	Erfüllt: Die Elektronik und Mechanik sind nicht Klavier-abhängig, wobei der Fußraum zum Befestigen der Aktuatoren gegeben sein muss oder das Ansteuerungskonzept geändert werden müsste
A9	Musikstück-Auswahl	Erfüllt
A10	Wiedergabe-Kontrolle	Erfüllt
A11	Navigation	Erfüllt
A12	MIDI-Integration	Erfüllt
A13	Musikverwaltung	Nicht Erfüllt

Tabelle 8.1: Ergebnisse der Anforderungen

Nun zur Betrachtung der Ausgangsfrage: Kann ein selbstgebautes „Player-Piano“ mit den auf dem Markt erhältlichen mithalten und bietet der Selbstbau somit eine kostengünstigere aber dennoch funktionsfähige Alternative?

Hier sind mehrere Punkte von Bedeutung.

1. Funktionalität: dieser Punkt wurde teils schon in der Tabelle 8.1 beantwortet. Letztendlich ist es möglich, ein funktionierendes, selbst-spielendes Klavier selber zu entwickeln. Dieses Projekt kann im Bezug auf Qualität und Ästhetik allerdings noch nicht mit professionellen Klavieren mithalten. Was allerdings die Funktionalitäten, wie zum Beispiel die kostenfreie Erweiterung des Musikkatalogs, betrifft, die das Klavier bietet, sind diese beim Eigenbau deutlich flexibler erweiterbar. Dafür ist allerdings, zumindest bei diesem Prototypen, die dauerhafte Funktionsfähigkeit noch nicht getestet und somit nicht garantiert.
2. Kostenaufwand: Eine Anforderung für die erfolgreiche Beantwortung der Fragestellung ist, dass der Eigenbau kostengünstiger sein soll als die professionell hergestellten „Player-Pianos“. Dies ist der Fall. Die Kosten dieses Projektes beliefen sich wie in Tabelle 4.2 aufgeführt auf 1753,78€. Diese Kosten können noch stark reduziert werden, wenn die Komponenten über andere Anbieter bestellt werden, was in diesem Projekt aufgrund von Firmenregulationen nicht möglich war. Trotz dessen liegen die Kosten für die Entwicklung unter 2000€, was ein großer Unterschied zu den auf dem Markt erhältlichen Modellen ist [vgl. Pia24].
3. Zeitaufwand: Es stellt sich natürlich noch die Frage, ob sich der Aufwand für das „Player-Piano“ tatsächlich lohnt. Wie in Kapitel 4.2.4 aufgeführt ist, beläuft sich der Zeitaufwand - rein für die Montage und Schaltung nach der Planung - bei der Hälfte der Tasten auf ca. 44 Stunden. Das sind etwa 80 Stunden reine Fleißarbeit. Dazu kommt der Zeitaufwand für Reparaturen und Tests. Außerdem sind diese 80 Stunden nur ein Bruchteil dessen, was für die Konzeption des Klaviers und der Entwicklung der Software gebraucht wurde. Die Evaluation ob der Zeitaufwand sich lohnt, hängt daher sehr stark davon ab, wie viel Freude diese Arbeiten den Personen, die dieses Projekt umsetzen, bringen. Es steht fest, dass das Projekt, ein Player-Piano selber zu bauen, besonders für jene sinnvoll ist, die gerne ihr Wissen und ihre Erfahrungen sowohl im Informatik- als auch im Elektronik-Bereich erweitern wollen.

Die Ergebnisse zeigen, dass selbstgebaute Varianten im Vergleich zu käuflichen Produkten erhebliche Kosteneinsparungen ermöglichen, jedoch mit einem hohen Zeitaufwand und komplexeren Anpassungsmöglichkeiten einhergehen.

Für diese spezifische Studienarbeit gilt insgesamt, dass im Laufe des Projekts viele Erkenntnisse gewonnen werden konnten. Im Bereich der Hardware umfassen diese ein vertieftes Verständnis für Elektronik, zum Beispiel für die Bedeutung der Kabeldurchmesser und dessen Bedeutung für den maximal fließbaren Strom, der Stromversorgungsanforderungen. Positiv zu vermerken ist die erfolgreiche Funktionalität des Schaltplans nach einigen Anpassungen sowie die hilfreiche Nutzung von Tinkercad¹ als Simulationswerkzeug. Herausforderungen ergaben sich vor allem beim Testen der Soft- und Hardware. Da in diesem Projekt das Testen eines Teils sehr stark vom Funktionieren des anderen abhängig war, behinderten sie sich anfangs stark. Zusätzlich gab es anfangs ungeplante Verzögerungen, durch die hohen bürokratischen Hürden bezüglich der Bauteil-Bestellungen. Die Erfahrung aus dem Projekt bietet wertvolle Erkenntnisse im Projektmanagement, in der Recherche und im Fachwissen. Es ist jedoch zu beachten, dass der Bau eines selbstspielenden Klaviers mit erheblichem Arbeitsaufwand und nicht zu vernachlässigen Kosten verbunden ist.

8.2 Ausblick

Für zukünftige Entwicklungen bieten sich zahlreiche Möglichkeiten sowohl im Software- als auch im Hardwarebereich des Projekts an. Auf der Hardwareseite könnten ein verbessertes Sicherheitskonzept für Elektronik, die Fertigstellung der Schaltungen für die restlichen Aktuatoren und die Untersuchung von Schalldämmungstechniken in Betracht gezogen werden. In Bezug auf Softwareverbesserungen könnten das Design sowie die Usability der Desktop Anwendung um einiges verbessert werden. Zukünftig wäre auch die Unterstützung weiterer Dateiformate wie PDF neben MIDI wünschenswert. Es ist zu beachten, dass Projekte dieser Art selten abgeschlossen sind, da ständig neue Funktionen hinzugefügt und Verbesserungen vorgenommen werden können.

¹<https://www.tinkercad.com/>

A Anhang: Demovideo

Zum Abschluss des Projekts wurden mehrere Demo-Videos des laufenden Prototypens erstellt.

Diese Videos können über den folgenden Link angeschaut werden:

[https://drive.google.com/drive/folders/1xvHsqJGUpA3CCskspw0LUtP47AdV-Hd6?
usp=sharing](https://drive.google.com/drive/folders/1xvHsqJGUpA3CCskspw0LUtP47AdV-Hd6?usp=sharing)

B Anhang: Quellcode

Valentin Richter

Der vollständige Quellcode¹ kann über Github heruntergeladen werden. Die Software wurde dabei in drei Repositories aufgeteilt.

1. Code der Desktop-Anwendung SAM: <https://github.com/Piano-Playing-Bot/SAM/releases/tag/v1.0>
2. Code für den μ C: <https://github.com/Piano-Playing-Bot/Arduino/releases/tag/v1.0>
3. Code der von SAM und μ C verwendet wird: <https://github.com/Piano-Playing-Bot/common/releases/tag/v1.0>

Instruktionen zum Kompilieren und Ausführen des Codes finden sich dabei jeweils in der „README“-Datei, die in jedem der Repositories vorhanden ist.

Im Repository der Anwendung SAM ist außerdem bereits eine Zip-Datei mit ausführbarem, 64-bit, Windows Executable vorhanden.

¹Zur vollständige Attribution soll hier angemerkt werden, dass der gesamte Quellcode von Valentin Richter erstellt wurde

C Anhang: Format-Spezifikationen

Valentin Richter

Hier werden die vollständigen Spezifikationen aller Formate und Protokolle geführt, die in dieser Arbeit erstellt wurden. Die Spezifikationen sind jeweils in Englisch geführt.

Abschnitt C.1 beinhaltet das PIDI-Format. In Abschnitt C.2 wird dann das Piano Digital Interface Library Format (PDIL)-Format spezifiziert. Zuletzt findet sich in Abschnitt C.3 das SPPP-Protokoll.

C.1 PIDI-Format

This section specifies the format of PIDI-files. PIDI serves as a memory-efficient and simple-to-use format for storing all data required to play a song automatically on a piano.

PIDI stands for „Piano Digital Interface“.

A PIDI file follows the following format:

```
<Magic Bytes: 4 bytes> <Commands Count: 4 bytes> <Commands>
```

Unless otherwise specified, small-endian encoding is used.

1. **Magic Bytes:** The format's Magic bytes are: „PIDI“. It is always written in big-endian format.
2. **Commands Count:** The amount of commands is given as an unsigned 32-bit number. It should follow exactly as many commands as given here.
3. **Commands:** All commands are written here one after another. The amount of commands is given in Commands Count. A single command's format is given below.

Every command is encoded as a 32-bit number. The following shows the bit field that makes up these 32 bits:

```
delta time : 12 bits
velocity   : 4  bits
length     : 8  bits
octave     : 4  bits
key        : 4  bits
```

1. **delta time:** The delta time specifies the amount of milliseconds since the last command that the command should be applied at. This parameter implies that the commands are ordered by the time at which they should be played.
2. **velocity:** The velocity specifies the strength with which to play the note. Strength of playing a note is directly correlated with the volume of the played note.
3. **len:** The length specifies the amount of centiseconds for which the command should be active. A centisecond is 10 milliseconds.
4. **octave:** The octave gives the octave offset of the musical note that this chunk refers to. The key and octave together specify a specific key on a piano. The octave should be interpreted as a signed 4-bit integer using two's complement. 0 refers to the middle octave on a piano. Negative numbers refer to octaves below the middle one, while positive numbers refer to higher octaves.
5. **key:** The key gives the specific musical note that is referred to in this command. The key and octave together specify a specific key on a piano. The following keys are encoded as follows:

```
C  = 0
C# = 1
D  = 2
D# = 3
E  = 4
F  = 5
F# = 6
G  = 7
G# = 8
A  = 9
A# = 10
B  = 11
```

C.2 PDIL-Format

This section specifies the format of PDIL-files.

The idea behind PDIL files is to provide a way to permanently store a collection of PIDI files. The Desktop Application uses this library to provide an overview of the available songs.

PDIL stands for „Piano Digital Interface Library“.

A PDIL file follows the following format:

```
<Magic Bytes: 4 bytes> <Amount: 4 bytes> <Song-Infos>
```

Unless otherwise specified, small-endian encoding is used.

1. **Magic Bytes:** The format's Magic bytes are: „PDIL“. It is always written in big-endian format. It stands for „Piano Digital Interface Library“.
2. **Amount:** The amount of song infos is given as an unsigned 32-bit number. There are exactly as many song infos as given here.
3. **Song Infos:** All song infos are written here one after another. The amount of song infos is given in Amount. A single song info's format is given below.

Every Song Info follows the following format:

```
<Name Length: 4 bytes> <Song Length: 8 bytes> <Name>
```

1. **Name Length:** The Length of the song's name. This specifies the amount of bytes that Name takes up.
2. **Song Length:** The length of the given song in milliseconds. This number can also be calculated dynamically from a PIDI file if required.
3. **Name:** The filename of the PIDI file, that contains the data for the given song. It should be provided as a relative path from the folder that this PDIL file is in.

C.3 SPPP-Protocoll

This section outlines the protocol used for the communication between the UI and the microcontroller (MC), that plays the music. The UI and MC are also referred to as nodes in this specification.

SPPP stands for „Self-Playing-Piano Protocol“

Any message between the server and client has the following format:

<Magic Bytes: 3 bytes> <Message-Type: 1 byte> [<Payload: n bytes>]

Unless otherwise specified, small-endian encoding is used.

1. **Magic Bytes:** The protocol's Magic bytes are: „SPP“. It is always written in big-endian format. It stands for „Self-Playing-Piano“.
2. **Message-Type:** The different types of messages that exist, are listed below under Payload. Each message type has its own 1 character signature, by which it is defined. Should a node not recognize the provided message type, it should ignore all following bytes until it reads the Magic Bytes again. Version upgrades should easily stay backward-compatible that way. The protocol uses the convention to use ASCII-encoded characters for the Message Type's identifier. Uppercase characters are used for messages sent by the UI and lowercase characters for messages sent by the MC.
3. **Payload:** How to interpret the payload depends on the Message-Type. The payload's format for each specified message type is given below. The title of each section provides the message's human-readable name and its 1-byte identifier.

Ping: „P“ The Ping message type exists to check whether the MC is available. It can also be used by the UI to initially identify the port, on which the MC is connected.

The initial message sent from the UI to the MC must be a Ping message.

There is no payload in this message.

Pong: „p“ This message should only be sent by the MC as a reply to the Ping message.

The payload should be an unsigned 16-bit number representing the maximum amount of PidiCmds that may be sent in a Music or New-Music message.

Success: „s“ The Successs message is only sent by the MC as a response to the UI. It indicates successfully receiving and executing the UI's last message.

The UI should wait until receiving a Success message before sending the next message. When no Success message is received by the UI before the timeout is reached, the same message must be sent again.

There is no payload in this message.

Continue: „C“ The Continue message is sent by the UI to pause or continue playing the song.

The payload should consist of a single byte. When the byte is 0, the song should be paused and continued otherwise.

The MC must respond with a Success message.

Volume: „V“ The Volume message is sent by the UI to set the factor by which the MC multiplies each note's velocity and thus its loudness.

The payload should consist of a single 32-bit IEEE-754 floating point number.

The MC must respond with a Success message.

Speed: „S“ The Speed message is sent by the UI to set the factor by which the MC's timer is multiplied.

The payload should consist of a single 32-bit IEEE-754 floating point number.

The MC must respond with a Success message.

New-Music: „N“ The New-Music message is meant for initiating the start of a song. It can be sent at any time by the UI.

Subsequent chunks of the song should be sent via the Music message instead, however.

The payload should be encoded as follows:

```
<PlayedKeys-Count: 1 byte> [<PlayedKey: 3 bytes>]+ <Commands-Count: 2 bytes> [<Com
```

There should be exactly as many PlayedKeys as provided in PlayedKeys-Count and as many Commands as provided in the Commands-Count.

Each PlayedKey is encoded as follows:

len:	8 bits
octave:	4 bits (signed)
key:	4 bits
velocity:	4 bits
<padding>:	4 bits

where all its parameters have the same meaning as in the PIDI Format.

New songs should be started as soon as the entire message was successfully received.

The MC has to respond with a Success message.

Music: „M“ The Music message exists for sending a list of commands for playing a Song to the MC. The Commands are given in the same format as in PIDI files. Instead of sending the potentially very big PIDI file all at once, it is sent in chunks, thus effectively implementing a method of streaming the music.

A Music message may only be sent as a response to a Request message.

The payload should be encoded as follows:

```
<Commands-Count: 2 bytes> [<Command: 4 bytes>]+
```

There should be exactly as many Commands as provided in the Commands-Count.

Upon receiving the message, the MC must respond with a Success message.

Request: „r“ The Request message is used to tell the UI, that the MC is ready to receive the next chunk of the song.

There is no payload in this message.

The UI has to respond with a Music message. If the song is over, the Commands-Count in the Music message should be set to 0.

Literaturverzeichnis

- [22] „Arduino vs. Raspberry Pi: Mikrocontroller Und Einplatinencomputer Im Vergleich“. In: (7. Juli 2022). URL: <https://www.ionos.de/digitalguide/server/knowhow/arduino-vs-raspberry-pi/>.
- [Ard24a] Arduino. *Arduino Nano Datasheet*. datasheet, 3. Dez. 2024. URL: <https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf> (besucht am 04.12.2024).
- [Ard24b] Arduino. *Arduino UNO Mini Datasheet*. datasheet, 3. Dez. 2024. URL: <https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf> (besucht am 04.12.2024).
- [Ard24c] Arduino. *Arduino UNO R3 Datasheet*. datasheet, 3. Dez. 2024. URL: <https://docs.arduino.cc/resources/datasheets/A000066-datasheet.pdf> (besucht am 04.12.2024).
- [AS23] Arduino und Karl Söderby. *Arduino & Serial Peripheral Interface (SPI) | Arduino Documentation*. 6. Jan. 2023. URL: <https://docs.arduino.cc/learn/communication/spi/> (besucht am 15.04.2024).
- [Ash22] Alvin Ashcraft. *I/O Completion Ports - Win32 Apps*. 8. Aug. 2022. URL: <https://learn.microsoft.com/en-us/windows/win32/fileio/i-o-completion-ports> (besucht am 15.04.2024).
- [Ass96] MIDI Association. *The Complete MIDI 1.0 Detailed Specification*. 3rd. Association of Musical Electronics Industry AMEI, MIDI Manufacturers Association MMA, 1996. 334 S.
- [Aut24] Firgelli Automations. „Arduino“. In: (2024). URL: <https://www.firgelliauto.com/en-de/products/arduino-uno-r3-microcontroller> (besucht am 15.04.2024).
- [Bak03] Dr. Zachary K Baker. „Universal Asynchronous Receiver/Transmitter“. 2003. URL: <https://www.unm.edu/~zbaker/ece238/slides/UART.pdf> (besucht am 16.04.2024).
- [Bay08] Robertson Bayer. „Windows Serial Port Programming“. In: (30. März 2008).
- [BW24] Martijn van Beurden und Anew Weaver. *Free Lossless Audio Codec*. Internet Draft. 14. Jan. 2024. URL: <https://datatracker.ietf.org/doc/draft-ietf-cellcar-flac> (besucht am 14.04.2024).

- [Clo21] Cloudflare. *What Does Buffering Mean? / Buffering in Video Streaming*. 2021. URL: <https://www.cloudflare.com/learning/video/what-is-buffering/> (besucht am 15.04.2024).
- [Den10] Allen Denver. *Serial Communications*. 30. Juni 2010. URL: [https://learn.microsoft.com/en-us/previous-versions/ff802693\(v=msdn.10\)](https://learn.microsoft.com/en-us/previous-versions/ff802693(v=msdn.10)) (besucht am 15.04.2024).
- [Ele22] Conrad Electronics. „74HC595 Schieberegister » Aufbau Und Funktionsweise Anschaulich Erklärt“. In: (22. Juni 2022). URL: <https://www.conrad.de/de/ratgeber/industrie-40/elektronik-bauteile/74hc595.html>.
- [Ele23] Conrad Electronics. *Elektromotoren: Aufbau, Funktionsweisen und Arten einfach erklärt*. 17. Mai 2023. URL: <https://www.conrad.de/de/ratgeber/industrie-40/elektromotor.html>.
- [Fal+23] FalkB et al. „LED-Matrix“. In: *mikrocontroller.net* (15. Aug. 2023). URL: <https://www.mikrocontroller.net/articles/LED-Matrix>.
- [Far24] Farnell. „LED-Matrix Treiberplatine“. In: (2024). URL: <https://de.farnell.com/seeed-studio/105020074/led-matrix-treiberplatine-arduino/dp/4007829> (besucht am 12.04.2024).
- [Fle23] Ryan Fleury. *A Taxonomy Of Computation Shapes*. 17. Feb. 2023. URL: <https://www.rfleury.com/p/a-taxonomy-of-computation-shapes> (besucht am 22.02.2024).
- [Hes] Heschen. *Frame Solenoid Electromagnet (HS-1240S)*. datasheet.
- [Hir22] Timothy Hirzel. *Basics of PWM (Pulse Width Modulation)*. Arduino, 15. Dez. 2022. URL: <https://docs.arduino.cc/learn/microcontrollers/analog-output/>.
- [Inc18] Diodes Incorporated. *74HC595*. datasheet, Nov. 2018. URL: <https://www.diodes.com/assets/Datasheets/74HC595.pdf>.
- [ISO93] ISO/IEC. *ISO/IEC 11172-3:1993*. Aug. 1993. URL: <https://www.iso.org/standard/22412.html> (besucht am 02.03.2024).
- [ISO99] ISO/IEC. *ISO/IEC 9899:1999*. Version 2. Dez. 1999. URL: <https://www.iso.org/standard/29237.html> (besucht am 11.04.2024).
- [Kab22] Prof. Peter Kabal. *Wave File Specifications*. Wave File Specifications. 27. Sep. 2022. URL: <https://www.mmsp.ece.mcgill.ca/Documents/AudioFormats/WAVE/WAVE.html> (besucht am 14.04.2024).

- [Kol21] Kollmorgen. „Wie Funktioniert Ein Servo-motor Für Lineare Direkt-Antriebe?“ In: (22. Jan. 2021). URL: <https://www.kollmorgen.com/de-de/blogs/wie-funktioniert-ein-servomotor-lineare-direktantriebe#:~:text=Ein%20Aktuator%20f%C3%BCr%20lineare%20Direktantriebe,Bewegung%20entlang%20der%20angetriebenen%20Achse..>
- [Mar22] Don Marshall. *Configuration of COM Ports - Windows Drivers*. 26. Okt. 2022. URL: <https://learn.microsoft.com/en-us/windows-hardware/drivers/serports/configuration-of-com-ports> (besucht am 15.04.2024).
- [Mur14] Casey Muratori. *Semantic Compression*. Semantic Compression. 28. Mai 2014. URL: https://caseymuratori.com/blog_0015 (besucht am 22.02.2024).
- [Okd24] Okdo. „Raspberry Pi“. In: *RS Components GmbH* (2024). URL: <https://de.rs-online.com/web/p/raspberry-pi/2020644>.
- [Pia24] Pianelli. *Yamaha U1 Enspire ST - Disklavier Enspire Silent - in Weiß - Musikinstrumente Und Musikzubehör*. 2024. URL: <https://www.pianelli.de/yamaha-u1-enst-disklavier-enspire-silent-in-weiss.html> (besucht am 11.03.2024).
- [Pot22] Jordan Potter. „Meet Domingos-Antonio Gomes, the Fastest Pianist in the World“. In: *Far Out* (6. Aug. 2022). URL: <https://faroutmagazine.co.uk/domingos-antonio-gomes-fastest-pianist-in-world/> (besucht am 02.02.2024).
- [Rei24] Reichelt. „Reichelt, Linearer Servomotor“. In: (2024). URL: <https://www.reichelt.de/linear-servomotor-mightyzap-17-n-41-mm-irrb-17n-41mm-p249917.html>.
- [Sch24] Patrick Schnabel. „Bipolarer Transistor“. In: *Elektronik Kompendium* (6. März 2024). URL: <https://www.elektronik-kompendium.de/sites/bau/0201291.htm>.
- [Sie23] Hannes Siebeneicher. *Universal Asynchronous Receiver-Transmitter (UART) / Arduino Documentation*. 11. Jan. 2023. URL: <https://docs.arduino.cc/learn/communication/uart/> (besucht am 15.04.2024).
- [SL15] Chang-Woo Song und Seung-Yop Lee. „Design of a Solenoid Actuator with a Magnetic Plunger for Miniaturized Segment Robots“. In: (18. Sep. 2015). URL: <https://www.mdpi.com/2076-3417/5/3/595>.

- [VA20a] D. Valenti-Electronic Musician und MIDI Association. *Control Change Messages (Data Bytes)*. 2020. URL: <https://docs.google.com/viewer?url=https://midi.org/component/edocman/midi-1-0-control-change-messages-data-bytes-2/fdocument?Itemid=9999> (besucht am 10.10.2023).
- [VA20b] D. Valenti-Electronic Musician und MIDI Association. *Summary of MIDI 1.0 Messages*. 2020. URL: <https://midi.org/specifications-old/item/table-1-summary-of-midi-message> (besucht am 10.10.2023).
- [Van12] Dominique Vandenuecker. *MIDI Tutorial for Programmers*. 2012. URL: <http://www.music-software-development.com/midi-tutorial.html> (besucht am 02.11.2023).
- [ZSH24] Nicholas Zambetti, Karl Söderby und Jacob Hylén. *Inter-Integrated Circuit (I2C) Protocol / Arduino Documentation*. 16. Jan. 2024. URL: <https://docs.arduino.cc/learn/communication/wire/> (besucht am 15.04.2024).