Fachhochschule Kiel
*University of Applied Sciences*

Faculty of Computer Science and Electrical Engineering
Bachelor Thesis in Information Technology

# Development of a complex Eurorack Sequencer Module

by Felix Kriegsmann - 931240

03.04.2023

supervised by
Prof. Dr. Robert Manzke
Prof. Dr. Steffen Prochnow

## Abstract

This bachelor thesis is about the development of a Eurorack synthesizer module that allows the creation of complex melody patterns. These sequences are created and edited with a flowchart like user interface.

One of modular synthesis unique selling points is its variety. While there is a real plethora of different modules available, strangely sequencers often times are “just” simple step sequencers, or in this context more precisely, linear sequencers. Sequencers in general are modules giving other modules two important pieces of information: (a) what they should do and (b) when they should do it. What exactly results from this is up to the second module, but in the most common case of a synthesizer voice this would be note pitch and note timing. This project aims to expand these two typical step node parameters by the selection of the next step and its condition allowing the creation of the familiar linear step sequencer, but also branching, tree-like structures and simple algorithms with the help of a flowchart like user interface.

# Contents

# 1. Introduction

## 1.1 Motivation

Modular synthesis, in this case more specifically the Eurorack format, is a world of wonders filled with ideas and possibilities. It has been a significantly growing topic in the last years, which is especially true for Eurorack, which is now regarded as the dominant format for modular hardware synthesizers. Currently, over 1000 manufactures produce over 15,000 different modules. At least that is what the Eurorack web site modulargrid [17] has listed. These modules range from DIY kits and boutique one-man businesses all the way up to well-established and known manufactures like Roland or Moog, mass-producing modules being only a part of their product line-up. This makes it even stranger that in the realm of sequencers more often than not we see sequencers that are “just” simple step sequencers, or in the context of this project more accurately, linear sequencers.

Sequencers are in general, not only in the Eurorack realm modules, that “sequence” other modules. This means that sequencers give other modules two important pieces of information: (a) what they should do and (b) when they should do it. What exactly results from this information is up to the second module in question, but in the most common case it is a synthesizer voice. For this use case the two information pieces are pitch and whether or not the sound should be on or not.

For the first part Eurorack modules use the Volt per Octave Standard (commonly abbreviated to V/Oct). This standard defines that the musical interval of one octave (a doubling of the frequency) equals exactly one Volt difference in the control voltage. Not defined in this standard is which tone exactly that is. These signals are analog and have typically a peak-to-peak amplitude of 5V (from -2.5V to +2.5V)

The second type of signal is a gate signal. It is digital by principle and equals either one or zero. In the real world all values below one volt are counted as zero and all above three as one. Typically these values are zero and five volts respectively.

“Linear sequence” refers to a sequence that only runs in one level. The typical 8 step sequencer has eight steps, which run, one by one, from one to eight. Some more complex sequencers expand on this concept by allowing trigger probabilities, different length steps or a possibility to influence the order of steps, but they still run “in line”. The goal of this project then is to allow the creation of more complex patterns. For this a tool is to be employed that comes from the realm of planning rather than the realm of music: the flow chart. The idea goes like this: by using a web interface the user is able to create a diagram that expands a typical step node by two parameters: the selection of the next step and its condition. This allows the creation of the familiar linear course of events, but also branching, tree-like structures and simple algorithms.

To achieve this goal a micro-controller is to be used which performs two functions: First, it directly controls the sequence and its output, second it serves the web interface used to configure said sequence. An ESP32 micro controller will be used for this. It features the necessary GPIOs, WiFi communication for the web server and an SPI Bus for the required DAC and ADC chips.

## 1.2 Approach

During the course of this thesis, the following steps are taken:

- An induction and research phase, in which hardware components and software libraries are selected
- A planning phase that defines key features in a requirements analysis and splits the work into logical blocks
- The actual implementation of the project in hardware and software
- A review and result interpretation phase

# 2. Technical Backgrounds

This section serves to explain key concepts and definitions, before used hardware and software components are briefly introduced.

## 2.1 Definitions

### 2.1.1 Control Voltage

Electronic devices such as Synthesizers, are internally controlled by electric signals. In analog, modular synthesizers these control signals are divided into two categories: Control voltage and gate signals. Both of them are analog signals, encoding certain information in their voltage.

**Gate, Trigger and Clock** A gate, or more precisely gate signal, is a type of control signal that has two states: “High” or “Low”. In the context of sound synthesis “High” usually means On, but inverted uses are also possible. It is a timing signal that specifies a certain window for an event to occur.

A trigger signal can be considered a subcategory of gate signals, in that it has the same binary states, but is by design a short pulse rather than a sustained state. It consist of a short burst of high voltage followed by a return to the passive low voltage state. In the sound synthesis realm it is often used to synchronise signals (then referred to as Clock) or "trigger" one-shot sounds like drums or other percussive elements.

**Volt per Octave** Control Voltage is the other kind of signal used in synthesizers. Here, in contrast to the binary nature of the gate signals, each possible voltage is mapped to a certain value. How exactly the information is encoded depends on the particular standard used. Eurorack, for example, uses the "Volt per Octave" standard for pitch specification. The phrase "One Volt per octave" refers to the fact that this analog synthesizer standard maps one Volt to one musical octave. This means the difference of on Volt in the Control Voltage equals a musical interval of one octave, which in turn equals a doubling in frequency. The standard was popularized by Bob Moog[1] in the 1960s and is now widely adapted, not only in the Eurorack system.

**MIDI** The "Musical Instrument Digital Interface" is another technical standard describing communication between musical devices. It in part supersedes the analog control voltage and gate schemes described previously.

MIDI in and of itself is a bidirectional digital protocol. Published in 1981 by Dave Smith[2] and Chet Wood at the Audio Engineering Society conference, it allows for the interconnection of a wide variety of different devices. It

[1]Robert "Bob" Moog was the founder of the now well-known company Moog Music and a pioneer of electronic music and especially synthesizers.

[2]David "Dave" Smith was an American engineer and founder of the synthesizer company Sequential. He is best know for his work on the MIDI standard and the creation of the Prophet-5.

enables up to sixteen separate channels, readily supports polyphony and can be exported to specific MIDI-files. It is also internally timed using the MIDI clock, a timing signal, "´[s]ent 24 times per quarter note"[3].

The most common types of MIDI Messages are "Note On", "Note Off" and "Control Change". Their simplified[3] message code can bee seen in Table 2.1. [3]

| | | | | | | |
|---|---|---|---|---|---|---|
| Note Off | 1 | 0 | 0 | 0 | Note | Velocity |
| Note On | 1 | 0 | 0 | 1 | Note | Velocity |
| Control Change | 1 | 0 | 1 | 0 | Index | Data |

Table 2.1: MIDI Messages

The Note byte encodes musical notes by simply counting them, centered around the concert pitch of A4 = 440 Hz [4] equal temperament, the standard for modern western music. A4 equals MIDI Note Number 69. The Note a half step above that then is A#4 or MIDI Note Number 70. This data is encoded in 8 bits leading to a note range of 128 discrete values (from 8.18 Hz up to 13289.75 Hz)

MIDI Velocity is one of the features that adds some dynamic to MIDI Notes. It is a way to transmit the Keyboard expression, to give a more naturally play feel. MIDI velocity encodes how fast (or how hard) a key was pressed, ranging from a soft touch up to a hard strike. This data is again encoded in 8 bits, leading to a value range from 0 to 127.

[3]MIDI allows for the specification of hierarchy data like the aforementioned channels or message types. Additionally in later MIDI Specifications Attributes are added to many messages.

[4]In German Music Notation this would be the Kammerton a'.

The "Note On" and "Note Off" messages are very similar to the control voltage and gate signals examined before. In fact sending eight consecutive "Note On" messages (leaving a sensible amount of time in between) would yield the same result as an 8 Step Sequencer.

It is true, that MIDI has many advantages over the Control Voltage and Gate configuration, but there are some drawbacks. The first one is the matter of precision. MIDI being a digital standard means that it comes with a limited resolution. Take for example the MIDI Note Values mentioned earlier: there are exactly 128 of them, mapped to half steps of a fixed tuning. While it could be argued that microtonality[5] or the usage of a different concert pitch are less widely used practices, a slide - a smooth, uninterrupted musical transition between two note (values) - would also benefit from having access to intermediate values. Though Slides are much more common, this issue is even more pronounced with what MIDI calls Control Changes. These allow the musician or sequencer to access secondary parameters, like the sounds timbre or audio effects. These too would suffer from the limited resolution. While the resolution could, and in fact has been increased between MIDI standards, seeing a rise from 8 bit in MIDI 1.0 up to four times that in MIDI 2.0 [3], it would still be lacking the near infinite solution of analog control voltage signals.

The second major disadvantage of MIDI in comparison with control voltage is its complexity: a control voltage can be transmitted by a single wire. While MIDI can be transmitted via a wide range of cable and wire configurations,

[5]The practice of using intervals smaller than a semitone.

it still necessitates coding and decoding at both ends requiring dedicated equipment. This added complexity rises component count, cost and the learning curve for producing your own schematic designs. Moreover, in the world of modular synthesis the additional features are often times simply not required. It is seen as more beneficial to have a dedicated timing, pitch and expression signal than a blended, digital MIDI signal.

### 2.1.2 Eurorack and Standards

Unfortunately, Eurorack standards are not strictly enforced.

This can be explained by the fact that Eurorack as a format has historically grown quite organically. Doepfer started releasing the first modules in the late 90s, first only inter-compatible with other modules produced by Doepfer [10].

Analogue Systems, a UK based synthesizer company, developed a similar format in the same time that would later be adapted to offer Eurorack compatibility. Other manufactures followed in a similar manner. However the synthesizer format also spoke to a growing DIY community, so that a lot of hobbyists, small time entrepreneurs and start-ups started offering their own modules [17].

Doepfer then specifies three types of signal for his A-100:

- Audio
- Control

- Trigger

The Trigger signals, or more precisely logic signal (including not only Trigger, but also Gate and Clock signals), are the most clearly defined signals of the three and are "rectangle shaped signals with typical voltage levels of 0/+5 V. In case of a trigger application normally the rising edge of the signal used to trigger the event" [9]. However, this is immediately followed by "[b]ut all A-100 modules will withstand gate/trigger/clock signals up to +12V" [9], hinting to the fact that is good practice to protect module inputs against "wrong" inputs.

Audio signals are defined as "Signals [...] produced by [...] sound source Modules [...] and [...] typically in the 10Vpp range (from -5 V to +5 V)" [9]. Note again the use of the word "typically" (in the original publishing even underlined for its importance).

The Control Voltage section takes it a step further by stating "[Control Voltages] are typically from -2.5 V to +2.5 V (5 Vpp) for LFOs, and from 0 V to +8 V for ADSRs" [9] (typically again being underlined).

All three definitions follow this up with the statement: "In addition the signal levels are usually specified in the module description." [9]

When looking at a different company's publications, for example the Canadian manufacturer Intellijel, one can see a different specification for Control Voltage: "[..] modular synthesizers use a format called Control Voltage. This is an analog electrical signal ranging from negative 10 volts to positive 10 volts." [8]

All of this is further complicated by the experimental, modular nature of the Eurorack format. In practice "all of the modules produce voltages, and can be used as control voltages or triggers, thus blurring the distinction between the various types." [9]

For the purposes of this project the gate and trigger inputs are excepted to be either zero or five Volts but should build them in a way that they can handle up to 12 Volts. Logic level outputs will output zero or five volts. CV outputs will range from -5 up to +5 Volts. Input expects the same range but should be prepared to handle -10V to +10V.

### 2.1.3 Step Sequencers

While these signals can be created by a human with an adequate interface device like a keyboard, more often than not a sequencer is used. Step sequencers utilise steps, fixed length time intervals to organize the musical pattern [22].

A step is a certain defined point in a step sequencers sequence. It consist of some musical data, information to identify this step and how to reach the next step.

In the case of a simple, 8-step linear step sequencer this information may be as simple as a number (ranging from 1 to 8) and the knowledge that the next step is always the next higher numbered one, the notable exception of course being the last step number 8. After this step the sequencer will "loop-around" to step 1 [21].

The data part of the equation can also be fairly different from one sequencer implementation to the next. Usually it consist of pitch information and a gate signal, determining if the note is supposed to be on or not.

### 2.1.4 Unipolar/Bipolar

Unipolar and Bipolar are two words describing the behavior of Voltages. Examine Figure 2.1, Unipolar voltage oscillates between its ground terminal, equal to 0v and its positive terminal of - here - 5V.

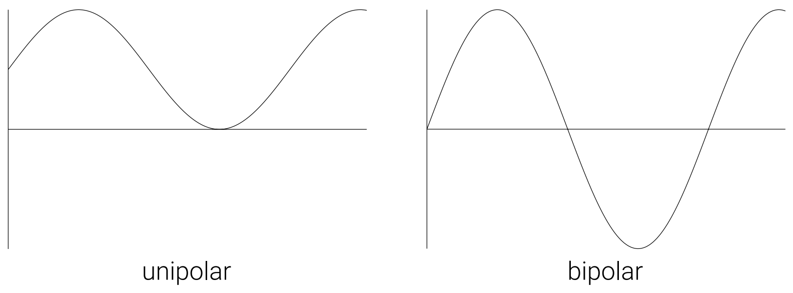


Figure 2.1: Unipolar/Bipolar

Bipolar Voltages swing past the 0V Point into a negative terminal, in this example -5V.

## 2.2 Previous Work

This project is in part based on similar projects, drawing from their schematics.

**MidiFox** MidiFox is a small Eurorack module previously developed by the author. "It takes a MIDI-Signal via Micro-USB from a Computer and turns it into Eurorack usable Note Value, MIDI Velocity and Gate CV." [16]

The CV Output schematic has been adapted from this project.

**CTAG TBD** The CTAG TBD is a multipurpose open-source Eurorack audio module. It is also based on the ESP32 [15].

The Logic Input schematic has been adapted from this.

## 2.3 Hardware

### 2.3.1 ESP32

ESP32 is a family of micro-controllers featuring low-cost and low-power system-on-chip micro-controllers with integrated Wi-Fi and Bluetooth capabilities. It is developed by the Chinese manufacturer Espressif Systems and was announced in 2016.

This project uses the ESP32-WROOM-32 Development Kit “a powerful, generic Wi-Fi + Bluetooth + Bluetooth LE MCU module that targets a wide variety of applications” [11].

It features two individually controllable CPU cores, the aforementioned Wi-Fi and Bluetooth capabilities and a range of hardware interfaces like SPI (Serial Peripheral Interface, see section 2.4.2). It includes 4 MB integrated flash memory, 25 usable GPIO-Pins (General Purpose Input/Output) and runs on an operating voltage of 3.0 to 3.6 V (nominal 3.3V) [11].

### 2.3.2 DAC

The MCP4922 is a "dual [...] 12-bit buffered voltage output Digital-to-Analog Converter[...]" [2]. It operates "from a single 2.7V to 5.5V supply" [2], communicates via SPI and has a user configurable Gain Selection Option [2].

### 2.3.3 ADC

The MCP3202 is a dual channel "12-bit analog-to-digital (A/D) converter" [1]. It is "compatible with the SPI protocol"[1] and "operates over a broad voltage range, 2.7V to 5.5V" [1].

## 2.4 Software

### 2.4.1 Arduino

The ESP32 Family supports different firmware and programming approaches. Espressif officially supports Arduino and FreeRTOS. FreeRTOS is an Open Source Real-Time operating system for micro-controller. Arduino is an open-source hardware and software platform. With a focus on learning and the community, the Arduino platform simplifies working with programmable micro-controllers. It features a wide range of hardware solutions in the form of micro-controllers and development kits. Additionally it provides the Arduino Library, a collection of C/C++ libraries that allow for simpler programming of compatible micro-controllers.

### 2.4.2 SPI

SPI or the Serial Peripheral Interface is a synchronous serial port that allows communication via the controller/periphery model between two or more hardware components. It uses three data lines for this:

1. SLK (sometimes referred to as SCK) for Serial Clock, to control the timing
2. POCI (MISO) the peripheral output and controller input line
3. PICO (MOSI) the peripheral input and controller output data line

These three lines can be shared by all bus participants. In addition to this one line called Chip Select is needed for each participating periphery component. **[spi_patent]**

Transmission begins when the controller drives the Chip Select Line of a component low. There are a number of different transmission modes with slightly different timings, but generally SPI transmits one Bit per Cycle. This transmission, however, is bi-directional or full-duplex. The transmission is ended when the controller sets the Chip Select Channel to high again.

Please note that sometimes other names for the lines are in use, for clarity this thesis refers to the stated names. In addition, sometimes other wire configurations are possible, which are also not covered here. [4]

### 2.4.3 JSON

The JavaScript Object Notation is a human readable data format. It is "lightweight, text-based [and] language-independent" [23]. It was originally specified by Douglas Crockford in 1997. Today it is specified in the RFC 8259 [6] and the identical ECMA-404 [24]. It uses key-value pairs, can be nested and allows for a variety of different data types. Most programming languages have libraries for dealing with serialization and deserialzation, making JSON a very useful format for transmitting data from one language to the next. Figure 2.2 shows an exemplary JSON Object.

```
{
   "id":1,
   "a":"-4.723400702605518",
   "b":"-2.192637650019835",
   "gate":true,
   "trigger":true,
   "type":"1",
   "nextNodes":[
      "2",
      "3"
   ]
}
```

Figure 2.2: Structure and syntax of a JSON Object

## 2.5 Libraries

This project uses the following C++ libraries: Arduino, ESPAsyncWebServer, AsyncTCP, SPIFFS, SPI and ArduinoJson. Additionally, it uses the following front-end libraries: Boostrap 5, jQuery 2.6.2 and FontAwesome

### 2.5.1 C++

**Arduino** see 2.4.1

**ESPAsyncWebServer** The ESPAsyncWebServer provides an Async HTTP and WebSocket Server for ESP8266 and ESP32 Arduino. It requires the AsyncTCP Library to work for ESP32. [20] Asynchronous communication allows for exchanging data without the need to reload the web page.

**AsyncTCP** AsyncTCP "is a fully asynchronous TCP library, aimed at enabling trouble-free, multi-connection network environment for Espressif's ESP32 MCUs." [19]

**SPIFFS** Library for the Serial Peripheral Interface Flash File System, the file system of the ESP32. "SPIFFS is a file system intended for SPI NOR flash devices on embedded targets. It supports wear levelling, file system consistency checks, and more." [12] The file system holds the HTML-Page and other Server-Side Data.

**SPI** SPI Library for ESP Systems, part of the Arduino Core. It is used for inter chip communication between the ESP32, the ADC and DAC respectively. For more information on SPI, see section 2.4.2.

**WiFi** A Library to enable WiFi communication for ESP. It is based on the Arduino WiFi library.

**ArduinoJson** ArduinoJson allows JSON serialization and deserialization on Arduino compatible devices.

### 2.5.2 Front-end

**Boostrap** "Powerful, extensible, and feature-packed frontend toolkit." [5] It is used to simplify the building of the web-page and provides basic responsiveness.

**jQuery** "jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation, event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers." [14] jQuery is used to simplify and shorten parts of the JavaScript code.

**FontAwesome** "Font Awesome is the Internet's icon library and toolkit". [13] It provides the icons for the project.

# 3. Requirements analysis

In the conception phase it is easy to come up with a plethora of ideas and "nice-to-have" features. Looking at the number of sequencers out there does not make this easier, as each boasts with its own unique features. As a baseline this sequencer should fill the minimum requirement of being to able to act as an 8-step sequencer. Specifically the "Baby 8".

**Baby 8** The Baby 8 Step Sequencer is a circuit originally published in "Massive Magazine" in the 1990's as part of a column called "Captain's Analog". Here the original design specifies 10 steps, but the last two steps are often omitted to fit a more musically grid of multiples of two. This is done simply by resetting the counter earlier.

The Baby 8 is, as the name implies a step sequencer featuring 8 steps. It is based around the CMOS 4017, a decade counter first introduced by the American company RCA in 1968. [18] A counter, like the 4017 is a simple Integrated Circuit. Its truth table looks like binary count up, which is unsurprising since that is its only job.

Looking at the schematic and article, the following functionality can be determine as the base line sequencer:

- 8 steps of musical data

- timed by an external trigger and reset

where each step has

- an editable CV Value and
- a pulse (or trigger) output

The schematic also calls for an LED as a form of visualization for the active step. Furthermore the article also gives ideas for the first additional modification: a second row of knobs, allowing for a second CV output. Formalizing these leads to the requirements table A, as shown in 3.1:

| Nr. | Required |
|---|---|
| A1 | A minimum of 8 Steps |
| A2 | An editable CV per Step |
| A3 | A Trigger Output per Step |
| A4 | A step trigger input |
| A5 | A reset trigger intput |

| Nr. | Recommended |
|---|---|
| A1 | A visualization for the active step |
| A2 | A second editable CV per Step |

Table 3.1: Requirements table A

**Hardware Inputs** As discussed in section 2.1.2 some protection against "wrong" inputs is to be built into the module. However, note that the values defined as "expected" are a must-have. This leads to the additional requirements shown in table 3.2

| Nr. | Required |
|---|---|
| B1 | The Logic inputs works with logic levels of 0V and 5V |
| B2 | The CV inputs work with -5V to +5V |

| Nr. | Recommended |
|---|---|
| B1 | The Logic inputs work with up to 10V |
| B2 | The Logic Levels are protected against negative voltage |
| B3 | The CV Levels are protected against voltages up to -10V/+10V |

Table 3.2: Requirements table B

**Flowchart** When looking at what separates a simple step from a flowchart node another core feature emerges: the dynamic reassignment, which step comes next. This alone, however, does not make the sequence any less "linear". One could rearrange the sequence, changing its order and would still have a line. So additionally a form of branching is required. Two simple conditionals come to mind: a simple mathematical comparison and a probability-based program flow. From the fact that a conditional has a true and a false output the need for two different next node variables arises. Mathematical comparisons like smaller, greater and equals should be made possible with a CV input or internally variable defined via the user interface. For clarification this internal variable gets the name "N".

At this point it also makes sense to change the nomenclature from "step" to "node". All of this can be seen in table 3.3.

**User Interface** Recommended feature C5 also really necessitates the graphical user interface. Here a user interface that allows editing of the step is needed. Meaning a way to enter a CV value and its next node or

| Nr. | Required |
|---|---|
| C1 | Dynamic reassignment of next Nodes |
| C2 | A possibility for Branching |

| Nr. | Recommended |
|---|---|
| C1 | A branching via mathematical comparison |
| C2 | A branching via probability |
| C3 | A CV Input |
| C4 | A second CV input |
| C5 | An internal variable |

Table 3.3: Requirements table C

nodes, additionally a way to organize the sequence. Lastly another must have feature is to hand over the sequence generated in the user interface to the actual sequencer.

The interface should also be easy to use, and aesthetically pleasing: two often-cited requirements that don't directly translate to a quantifiable metric.

Regarding the ease of use: A proper testing of these metrics requires a lot of time and a number of test users, that regrettably are beyond the scope of this thesis. Proper measurable metrics are for example the success rate, given as a percentage of how many test users can achieve a defined goal task, the time requires for them to fulfill this goal task, the number of mistakes they make in this time or a user's subjective satisfaction. It is therefore regarded as an optional feature. However, in the the realm of analog synthesizers there is the term of "knob-per-function", meaning that every available variable and setting of the synthesizer has its own dedicated user input. This is indeed a testable metric, and for that reason these pseudo ease of use can be regarded as a recommended feature. Of note here is that the practice of "menu-diving" would not count against the "knob-per-function" design philosophy. "Menu-diving" is the quite infamous practice that started with the invention and mass market adaptation of early micro controllers and text displays to "hide" options and parameters of complex electronic devices, like synthesizers, in sub-menu upon sub-menu. An example for this would be the the Yamaha DX7, featuring the, at the time new and "fairly complicated" [7], technology of FM synthesis. "Basically, in order to use [it], you have to forget everything you know [...] and go on a deep dive through a series of menus to get it to do what you want. This was a problem with the DX7 as the only way that you had to interact with it was through a tiny LCD display and a load of membrane keys." [7] This naturally has negative effects on the usability of these devices and is to be avoided.

Aesthetic is another wholly undefined and hard to measure feature. It again is best put in number by extensive testing with test users. It is therefore regarded here as an optional feature.

To promote accessibility the user interface should be as language independent as possible. To put this in a measurable metric it should suffice to have no more than ten language-only icons.

The User Interface requirements are shown in table 3.4.

| Nr. | Required |
|---|---|
| D1 | User inputs for CV output |
| D2 | User inputs for next node |
| D3 | User inputs for internal variable N |
| D4 | Allow user to organize sequence flow |
| D5 | A way to save the generated sequence |
| **Nr.** | **Recommended** |
| D1 | "Knob-per-function" Usability |
| D2 | Language Independence |
| **Nr.** | **Optional** |
| D1 | The user interface is aesthetically pleasing |

Table 3.4: Requirements table D

**Further considerations** Building on this another, final round of considerations is added: Allowing the gate and trigger outputs to be set by the user allows rests and different length notes, another feature expanding the variability of the sequence. The addition of a second CV input allows for greater variance in sequence without much additional effort, the corresponding hardware and software is simply duplicated. The added sequence complexity via branching justifies a enlarging of the minimum node count of 8 to 16. Visualization of logic input and output via LEDs is another nice to have, yet optional feature. The design of a PCB, and a proper enclosure/front plate were also briefly considered, but regarded as beyond the scope of this prototype. Finally, after the first week of implementation the functionality of the internal variable N was dropped. It was decided that this software-only feature can be added again later without much complication. This considerations are formalized in table 3.5.

| Nr. | Recommended |
|---|---|
| F1 | A node gate output |
| F2 | A node trigger output |
| F3 | A user interface for the node gate value |
| F4 | A user interface for the node trigger value |
| F5 | A Second CV input |

| Nr. | Future |
|---|---|
| F1 | An internal variable N |
| F2 | The design of a dedicated PCB |
| F3 | The design of a Front plate |

Table 3.5: Requirements table F

Combining all previous tables gives us tables 3.6 and 3.7.

Note: To avoid the duplicate use of the letter "R" the numbering is prefixed by the first letter of the verbs "must" for required, "shall" for recommend and "can" for optional features. The Future Features are listed with simple, unprefixed numbers, as they are only listed here for completion’s sake and are not mentioned further.

| Nr. | Required |
|---|---|
| M1 | A minimum of 16 Nodes |
| M2 | An editable CV per Step |
| M3 | A Trigger Output per Step |
| M4 | A step trigger input |
| M5 | A reset trigger input |
| M6 | The Logic inputs works with logic levels of 0V and 5V |
| M7 | The CV inputs work with -5V to +5V |
| M8 | Dynamic reassignment of next Nodes |
| M9 | A possibility for Branching |
| M10 | User inputs for CV output |
| M11 | User inputs for next node |
| M12 | Allow user to organize sequence flow |
| M13 | A way to save the generated sequence |

Table 3.6: Final Requirements Table

| Nr. | Recommended |
|---|---|
| S1 | A visualization for the active step |
| S2 | A second editable CV per Step |
| S3 | The Logic inputs work with up to 10V |
| S4 | The Logic Levels are protected against negative voltage |
| S5 | The CV Levels are protected against voltages up to -10V/+10V |
| S6 | Branching via mathematical comparison |
| S7 | Branching via probability |
| S8 | A CV Input |
| S9 | A second CV input |
| S10 | "Knob-per-function" Usability |
| S11 | Language independence |
| S12 | A node trigger output |
| S13 | A node gate output |
| S14 | A user interface for the node gate value |
| S15 | A user interface for the node trigger value |

| Nr. | Optional |
|---|---|
| C1 | The user interface is aesthetically pleasing |

| Nr. | Future |
|---|---|
| 1 | An internal variable N |
| 2 | The design of a dedicated PCB |
| 3 | The design of a Front plate |

Table 3.7: Final Requirements Table (cont'd)

# 4. Implementation

## 4.1 Hardware

### 4.1.1 Schematic Explanation

**CV Input** The CV Input stage translates two incoming voltages with expected values of -5V up to +5V into a Values the ESP can understand. Here this is done with the help of an ADC Chip, namely the MCP3202. This leads to two new challenges:

1. The MCP3202 expects unipolar input voltages, but the use of bipolar ones is desired.

2. To work with the 5V Voltage maximum, the ADC Chip needs to be supplied with 5V, but the used Micro-controller only accepts 3.3V.

Problem number two is actually the easier one of the two to fix: the operation to be done here is called a logic level shift and is common enough to merit a dedicated part: the level shifter. Here the TXB0104 is used. It works in both communication directions and is fast enough to be compatible with the SPI specification. [25]

The first challenge is solved with the help of an Operational amplifier. The schematic is referred to as a bipolar to unipolar converter.

It is required to translate the voltage range from -5V to +5V into 0V to +5V, meaning:

If $V_{in} = -5V$, then $V_{out} = 0V$ and

if $V_{in} = +5V$, then $V_{out} = +5V$.

This is a linear function and can be expressed with the formula 4.1 for linear circuits:

$$V_{out} = V_{in} \cdot Gain + V_{offset} \tag{4.1}$$

The gain in this case is 0.5, because the input range has a difference of 10 between its highest and lowest amplitude and the difference of the outputs range is 5. Applying only this would get an output range from -2.5V to +2.5V, which is still somewhat off from the target of 0V to 5V by a constant factor of 2.5, which would be the offset $V_{offset}$. Now equation 4.1 can be re-written as equation 4.2:

$$V_{out} = V_{in} \cdot 0.5 + 2.5 \tag{4.2}$$

If re-written further, this would become equation 4.3:

$$V_{out} = \left(V_{in} \cdot \frac{1}{2} + 5 \cdot \frac{1}{2}\right) \cdot 1 + 0 \tag{4.3}$$

This is the equation for a summing amplifier, whose general formula and schematic look like equation 4.4 and figure 4.1:

$$V_{out} = \left( V_{in_1} \cdot \frac{R2}{R1 + R2} + V_{in_2} \cdot \frac{R1}{R1 + R2} \right) \cdot \left( 1 + \frac{R4}{R3} \right) \tag{4.4}$$

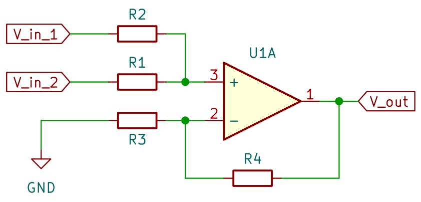


Figure 4.1: Summing Amplifier

The output voltage `V_out` is a sum of both input voltages `V_in_1` and `V_in_2`. However, these voltages are not summed up directly but are weighted by the Resistors R1 and R2. The Resistors R3 and R4 provide another although inverted factor.

Since $\frac{R4}{R3} = 0$, it can be omitted them from the design.
(R3 is technically infinite here, meaning disconnected.)

In addition, since

$$\frac{R2}{R1 + R2} = \frac{R1}{R1 + R2} \implies R1 = R2$$

, any practical value for R1 and R2 can be chosen, here 10kΩ.

For the Op-Amp it is best to use an Amplifier that provides "Rail-to-rail" output, which means that the output of the actual part comes as close to ideal maximum and minimum output values as possible. These theoretical values are the positive and negative power supply or rail - in this case 0V and 5V. The MCP600X Series features such chips, and since there are two inputs that need this value translation, the dual channel MCP6002 is used.

The addition of all these components leads to the completed CV Input schematic as seen in Figure 4.2.

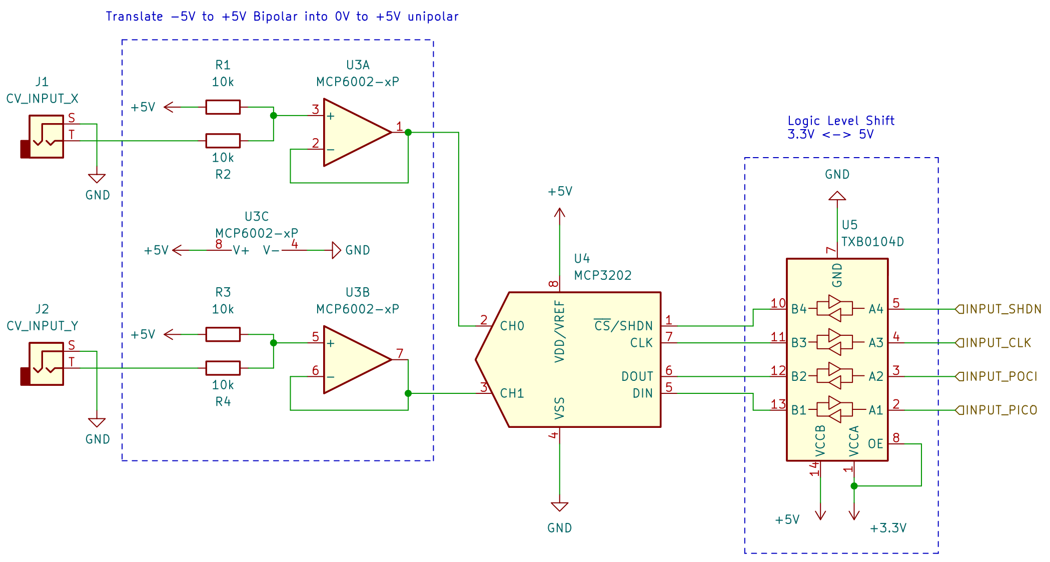


Figure 4.2: CV Input

**Logic Inputs** For the logic inputs, namely the trigger and step input there is only a need to cap any incoming Voltage above 3.3V to 3.3V.

For this a basic transistor, here the SS8050, is used as a switch. A 10kΩ resistor is added on the collector of the transistor to prevent the transistor from shortening the output. Then a basic switching diode (here the 1N4148) is added as protection against negative voltages. Then an optional Push switch is added that allows for manual triggering. This leads to the schematic shown in Figure 4.3.

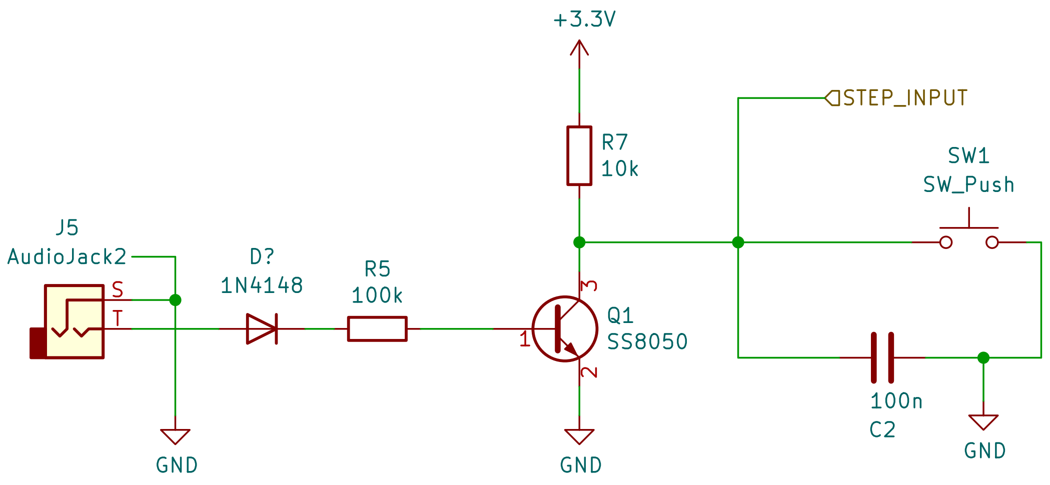


Figure 4.3: Logic Input

**CV Outputs** The CV Output is done simply with the help of a DAC (Digital Analog Converter). Here the MCP412 is used. It features two outputs with 12 bits of resolution. Communication is again done via SPI and works in a similar manner as the ADC mentioned above. However since the communication here only goes into one direction (the PICO line remains unconnected), the chip can be driven directly with 5V leading to the desired output range.

**Logic Outputs** The Logic Outputs use two GPIO Pins of the ESP32. It would have been more ideal to bring them up to a 5V Level, but this feature was cut in the interest of time.

## 4.2 Software ESP32

### 4.2.1 node.h

The node class holds the data as it was input from the web interface. It consists of a `struct` containing the musical data, an array holding the next node data, a node type, an id and finally a probability value. The `struct` contains two doubles for the output control voltages and two booleans representing the output trigger and gate states. The node type is an enumeration differentiating the three node types Simple, Conditional and Probability as mentioned in Chapter 3. The Constructor method and mutator methods for the value struct are trivial and are therefore not further explained.

The only function of note then is the `getNextNode()`-function. Here, depending on the node type, the next sequencer node is determined. If the node type is Simple Step, an implementation of the simple step sequencer introduced in section 2.1.3, the next node of a node is predetermined and `getNextNode()` achieves the same result as any ordinary getter function. In the case of a Probability Node, the `getRandom()`-helper function provides a random value that leads to a random choice between one of two possible next nodes. The Conditional type is to evaluate a given condition and determines its next node based on the result of that evaluation.

### 4.2.2 main.cpp

**Outputs** The functions `scaleValue()`, `buildWord()` and `sendWord()` provide the communication interface for the Digital Analog Converter. `scaleValue()` clamps the value to the expected value range of -5 to +5, then translates that to the value range of the DAC from 0 to the max range, depending on the resolution. In the case of 12 bit resolution 4095. `buildWord()` combines this with the configuration flags for the DAC found in its data sheet. The relevant configuration consists of 4 bits:

$\overline{A}/B$ DAC Channel A and B Selection Bit

$BUF$ Buffered and Unbuffered Input Selection Bit

$\overline{GA}$ Output Gain Selection Bit

$\overline{SHDN}$ Shutdown Selection Bit

The Gain can be either $V_{REF}$ or twice that. The $\overline{GA}$ flag is set to 0, allowing for 10V Peak to Peak Amplitude. $\overline{SHDN}$ selects between Active operation mode and shutdown mode, which disables the selected DAC channel. [2] This means the $\overline{SHDN}$ flag will be 1. Buffered Inputs are selected, so $BUF$ will be 1. Lastly, $\overline{A}/B$ will depend on which channel the data is to be sent to. Finally, `sendWord()` sends this data using SPI as discussed in section 2.4.2.

`outputData()` uses `digitalWrite()` to set the gate and trigger output, then calls the aforementioned functions to output the control voltage data.

**Interrupts** The step and reset inputs use hardware interrupts. These are attached in the `setup()`-function. Since the input transistor schematic reversed the signal polarity, they are attached to the falling edge of a trigger input. When a falling edge is detected on the specified pi,n normal program execution is halted, and the interrupt service routine is called. Since the remaining program is waiting for the interrupt to terminate, it is best practice to keep them as short as possible, often just long enough to set a single flag, to be acted on at a later stage. This is also the case here, but the code is expanded to include "debouncing".

"Bouncing" is the practice of physical buttons or similar inputs to not settle into their new state in an instance, but taking some, although small, amount of time. While this time usually cannot be perceived by humans, for micro controllers, who execute millions of instructions per second, this is a notable difference. This would lead them to count a single input any number of times. "Debouncing" prevents that, here by disregarding any new supposed inputs for a short time frame after the first one has been detected.

The main loop function checks if the flags for reset, step or load have been set. If they have, it calls the according functions and clears the flag.

**Loading** The `load()`-function handles the loading of node data from the given JSON-String. First, a buffer is created, then deserialization function is called. After that, the function goes through the JSON Array and creates a

`Node` object for each entry, saving them in a Node array using their ids as an index. After this, the start of the sequence is determined and the reset function is called.

**Node flow** After loading, the sequence is started by calling the `reset()`-function. It sets the current Node to be the start node. The micro controller then sets the outputs using the `outputData()`-function. After that, it preloads the next node to the `seqNextNode`-variable. The `step()`-function follows a very similar process, the only difference being that instead of re-setting to the start node the sequence gets advanced to the next node.

**CV Input** The CV inputs work in a similar manner to the Outputs discussed above. A certain bit flag configuration is prepared and then sent to the ADC. However, unlike with the DAC, the communication with the ADC is bi-directional, so after sending the configuration data out the code reads the response from the ADC, which contains the sampled voltage. This data is then processed and saved to global input variables for access inside the rest of the program.

There are three configuration bits for the MCP3202:

$SGL/\overline{DIFF}$ Mode selection bit

$ODD/\overline{SIGN}$ Channel/polarity bit mode

$MSBF$ Bit order selection bit

"$SGL/\overline{DIFF}$ is used to select Single-Ended or Pseudo-Differential mode."

[1] Since two inputs are needed, Single-Ended Mode is chosen and this flag is set. "The $ODD/\overline{SIGN}$ bit selects which channel is used in Single-Ended mode, and is used to determine polarity in Pseudo-Differential mode." Since Single-Ended Mode is used, this is the channel select bit, and the configuration is sent once with the bit set and once without [1]. The $MSBF$ bit decides the bit order between Most Significant Bit First (MSBF) or Least Significant Bit First (LSBF) [1]. Here MSBF is chosen.

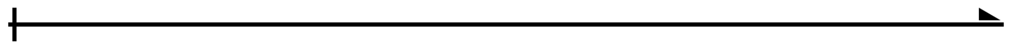
## 4.3 Web component

**index.html** The HTML-File contains the basic structure of the web interface. It defines a top menu bar consisting of a single row of buttons. Additionally, the main input form is defined here. It consists of a structured HTML-Form and provides a graphical interface for editing the values of the data holding classes as described in 4.2.1.

### 4.3.1 svgDraw.js

This JavaScript file takes care of all the line drawing inside the svg tag.

**Setup** This explanation starts by looking at the prerequisites in the other files. Firstly, there is a `<svg>` in the HTML file. It features an id for future reference. All children are added at a later stage through code. The only points of interest here are the two `<marker>`-tags. They specify two markers or arrowheads. These can later be drawn at the end of svg path elements (and a number of other elements that are of no relevance here). One of them is drawn using a simple line and takes the shape of a crossbar, the other is a polyline and is drawn as a triangular arrowhead. This leads to the following arrow decoration:



Now a look at the svgDraw,js itself is taken. It starts by declaring an array called `connectionsList`. Here the ids from which HTML element to which HTML element a connecting line needs to be drawn is saved. In a populated state it might look like this:

```
[{from: '1', to: '2'},
{from: '2', to: '3'},
{from: '3', to: '4'},
{from: '4', to: '5'},
{from: '5', to: '1'}]
```

Although JavaScript is a weakly typed language that does not support access modifiers, this variable is "private" in nature. It is modified only by the functions `addConnection()` and `deleteConnections()`.

**addConnection()/deleteConnection()** In these functions new connections can be added to or deleted from `connectionsList`. They are "public" in spirit.

First, the array is checked for an existing connection. If such a connection is found, the work is already done, and the function ends. If such a connection is not found, a new one needs to be added. The `deleteConnection()`-function simply deletes all connections from a given id using the inbuilt `filter()` function.

**connectAll()** `connectAll` is the main function of this file. It determines if the number of existing paths equals the number of needed paths and adjust accordingly (using the `createPath()` and `createPaths()` helper-functions) before looping over `connectionsList` and calling `connectElements()` for each connection.

This function also adds the event handler for the initial `document.ready()` and `window.resize()` callbacks.

**connectElements()** This function takes the start and end element of the connection and calculates starting and end coordinates from it, then calls `drawPath()` for the actual drawing with them.

**drawPath()** In this function the actual drawing takes place. It determines the relative position of the starting and end point and draws a connecting line between them. This line can take different shapes depending on the positing of start and end. Figure 4.4 shows the variants for the conditions below.

a.) $X_{Start} < X_{Start} \wedge Y_{Start} = Y_{Start}$

b.) $X_{Start} < X_{Start} \wedge Y_{Start} > Y_{Start}$

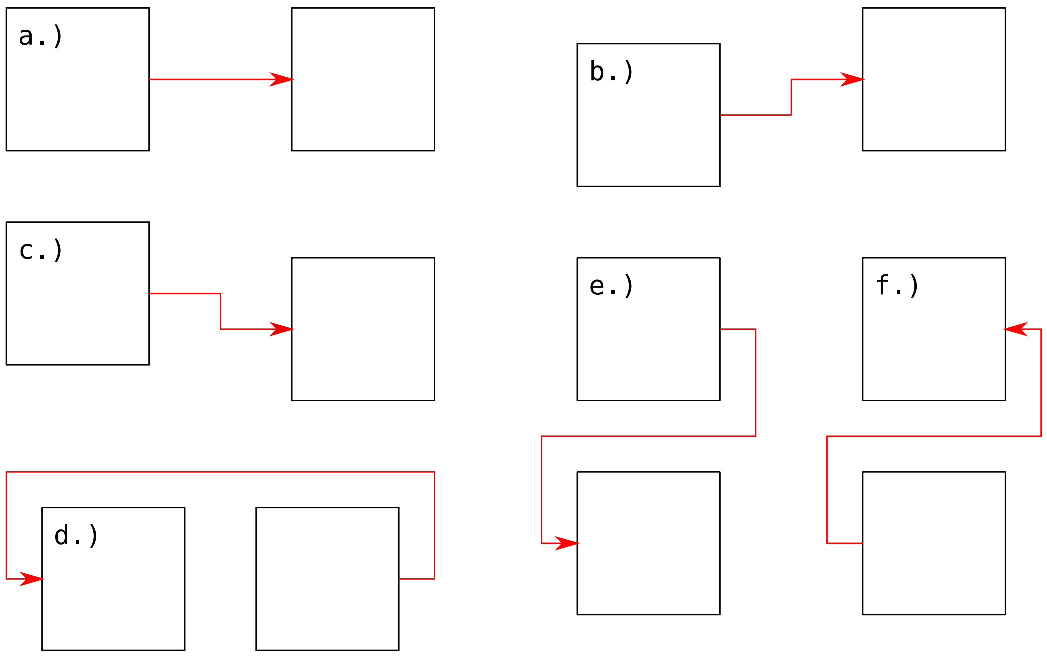


Figure 4.4: path variants

c.) $X_{Start} < X_{Start} \land Y_{Start} < Y_{Start}$

d.) $X_{Start} > X_{Start} \land Y_{Start} = Y_{Start}$

e.) $X_{Start} > X_{Start} \land Y_{Start} < Y_{Start}$

f.) $X_{Start} > X_{Start} \land Y_{Start} > Y_{Start}$

Case d.) is also used if an item references itself.

### 4.3.2 script.js

This file contains the main script of the project.

**Helper functions** This code section contains all helper functions. Most of them are trivial in what they do:

**linkSliderToNumberInput()** links a slider input with it's corresponding number input to make them both show the same value.

**translateBoolToIcon()** and **formatNumber()** format the output for nicer looking printing.

**hideMenu()**, **showMenu()** and **toggleMenu()** deal with the visibility status of the main editing menu.

**switchEditWindowNode()** updates the Editor Menu to show the current nodes data.

**randomint()**, **randomInRange()**, **randomBoolean()** and **randomColor()** generate random values. The number values are mainly used for the randomisation button, the color is used for node card identification.

**modeSelected()** toggles the visibility of the second "next node" input depending on context.

**Event Handlers** The event handlers allow drag'n'drop functionality for the node cards. On dropping the card, a redrawing of the connecting svg-arrows is triggered.

**createNewNode()** creates a new Sequencer Node. **createNodeCard()** then adds a new node card used as the User Interface for it utilizing the

JavaScript HTML DOM (Document Object Model). It adds all relevant styling and event listeners.

**createGrid()** creates the background grid which contains drop targets. The grid is created by using two nested `for`-loops.

**dataSubmit()** saves the data from the visualisation to the `nodes` array. If a node with the id already exits, it is updated, otherwise created. It also triggers the redrawing of connecting arrows.

**writeToDevice()** encodes the `nodes` array using JSON, then uses HTML POST to send it to the server via AJAX.

**randomizeValues()** randomizes the values of the node within certain parameters.

### 4.3.3 Data Transfer

The data transfer uses a JSON String.

The JSONs structure is an array of node objects, the node objects are structured as the example seen in figure 4.5.

```
{
    "id":1,
    "a":"-4.723400702605518",
    "b":"-2.192637650019835",
    "gate":true,
    "trigger":true,
    "type":"1",
    "nextNodes":[
        "2",
        "3"
    ]
}
```

Figure 4.5: JSON syntax example

# 5. Results

## 5.1 Current State

Here, the final state of the project at the end of the thesis time frame is stated and compared against the targets set in chapter 3. An overview is presented in the tables 5.1 and 5.2. Overall, the sequencer can only be considered as a prototype. While most of the major features are implemented, some are not and even more are untested or have other issues.

M3 is only partly implemented, as the sequencer gives out two gate signals instead of a gate and a trigger signal. The correct functionality could possibly be implemented by using a software side timer or by a hardware side AND logic gate, taking this gate signal and the originally step trigger as inputs.

The state of the CV input (M7 and S5) as well as the input protection of the logic inputs (S3 S4) could not be tested.

Lastly M13 is not working. The selected library for the ESP32 server side POST handling was not working as documented, leading to a broken feature. Unfortunately this feature was tested to late in the development process to implement another way. The web interface, however, can generate a sequence JSON string that, if given to the micro controller by manually

| ID | Desc. | State | Comment |
|---|---|---|---|
| M1 | 16 Nodes | Y | |
| M2 | CV out | Y | |
| M3 | Trigger out | Y | Implemented as gate |
| M4 | Step in | Y | |
| M5 | Reset in | Y | |
| M6 | 5V logic in | Y | |
| M7 | CV out | ? | Untested |
| M8 | Node reass. | Y | |
| M9 | Branching | Y | Only random |
| M10 | UI CV out | Y | |
| M11 | UI next | Y | |
| M12 | Organize | Y | |
| M13 | Saving | N | JSON transmission not working |

Table 5.1: Completed features overview

setting a constant in the code, before uploading it to the ESP32, produces the desired result.

While branching via mathematical comparison could not be implemented in time, branching via probability is possible, but lacks a user interface. Manual editing of JSON as before, however, is working.

The output of control voltage is also only partly working. The documentation of the DAC specifies a maximum output gain of twice the reference voltage that could not be achieved. The DAC has also not been calibrated or tuned, so the resulting output voltage may contain systematic errors. Additionally, there was an oversight that this output voltage should be bipolar, limiting the control voltages use cases.

| ID | Desc. | State | Comment |
|---|---|---|---|
| S1 | Step visual | N | |
| S2 | 2. CV | Y | |
| S3 | 10V logic in | ? | Untested |
| S4 | -V logic in | Y | Untested |
| S5 | 10V CV in | ? | Untested |
| S6 | Comparison | N | |
| S7 | Probability | Y | No UI |
| S8 | CV in | Y | Untested |
| S9 | 2. CV in | Y | Untested |
| S10 | Usability | | |
| S11 | Language | | |
| S12 | Trig. out | Y | |
| S13 | Gate out | Y | |
| S14 | UI trig | Y | |
| S15 | UI gate | Y | |
| C1 | UI Aesthetic | ? | Unquantifiable |
| F1 | N | N | |
| F2 | PCB | N | |
| F3 | Front plate | N | |

Table 5.2: Completed features overview (cont’d)

## 5.2 Outlook

The developed prototype shows clearly that the concept is viable. In the future, more of the missing features could be implemented. Adding to that the future requirements mentioned in chapter 3 could be executed given more development time.

# 6. Conclusion

On the onset of this thesis the question was raised whether a complex Eurorack sequencer module could be developed. Then existing components and software resources were shown, that in sum could achieve this goal. While these could in theory simply have been added together, in reality there are many hurdles. Sometimes parts do not behave as expected, other times initial development time estimations turn out to be wrong. This did not only impact the number of finished features but also their quality, leaving some features in a less polished, less desired and untested end state.

As hinted in the previous chapter the non-linear sequencer is a viable concept, but for now remains a concept and a prototype. Showing many features that could not be completed in the given time frame, leaving room for further development. But given more time and attention this prototype could be turned into a full working Eurorack sequencer module.

# Bibliography

[1] *2.7V Dual Channel 12-Bit A/D Converter with SPI Serial Interface*. Microchip. 2011.

[2] *8/10/12-Bit Dual Voltage Output Digital-to-Analog Converter with SPI Interface*. Microchip. 2010.

[3] MIDI Association. *Midi Specifications*. URL: `https://www.midi.org/specifications/midi1-specifications`.

[4] Open Source Hardware Association. *A Resolution to Redefine SPI Signal Names*. URL: `https://www.oshwa.org/a-resolution-to-redefine-spi-signal-names/` (visited on 03/20/2023).

[5] Boostrap. *Boostrap*. 2021. URL: `https://getbootstrap.com/` (visited on 03/02/2023).

[6] T. Bray. *The JavaScript Object Notation (JSON) Data Interchange Format*. Dec. 2017. URL: `https://www.rfc-editor.org/rfc/rfc8259` (visited on 03/23/2023).

[7] Nick Cooke. *Hardware focus: Yamaha DX7*. Dec. 2020. URL: `https://www.attackmagazine.com/technique/hardware-focus/yamaha-dx7/` (visited on 03/16/2023).

[8] doepfer. *Eurorack 101*. 2023. URL: `https://intellijel.com/support/eurorack-101/` (visited on 02/08/2023).

[9] doepfer. *Technical Details A-100*. 2023. URL: https://www.doepfer.de/a100%5C%5Fman/a100t%5C%5Fe.htm (visited on 02/08/2023).

[10] doepfer. *Zeit-Tabelle*. 1997. URL: https://www.doepfer.de/time.htm (visited on 02/26/2023).

[11] *ESP32 WROOM 32 Datasheet*. Version 3.3. Espressif Systems. 2022.

[12] Espressif. *SPIFFS*. 2021. URL: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/spiffs.html (visited on 03/02/2023).

[13] fontawesome. *fontawesome*. 2021. URL: https://fontawesome.com/ (visited on 03/02/2023).

[14] jquery. *jquery*. 2021. URL: https://jquery.com/ (visited on 03/02/2023).

[15] CTAG FH Kiel. *CTAG TBD*. 2021. URL: https://github.com/ctag-fh-kiel/ctag-tbd (visited on 02/08/2023).

[16] Felix Kriegsmann. *MidiFox. a Eurorack Module*. Jan. 2021.

[17] *modulargrid.net*. URL: https://www.modulargrid.net/ (visited on 03/16/2023).

[18] Computer History Museum. *1963: COMPLEMENTARY MOS CIRCUIT CONFIGURATION IS INVENTED*. URL: https://www.computerhistory.org/siliconengine/complementary-mos-circuit-configuration-is-invented/ (visited on 03/16/2023).

[19] me-no-dev. *AsynTCP*. 2021. URL: https://github.com/me-no-dev/AsyncTCP (visited on 03/02/2023).

[20] me-no-dev. *ESPAsyncWebServer*. 2021. URL: https://github.com/me-no-dev/ESPAsyncWebServer (visited on 03/02/2023).

[21] Martin Russ. *Sound Synthesis and Sampling*. CRC Press, Aug. 2012.

[22] Allen Strange. *electronic music. systems, techniques, and controls*. 2. Wm. C. Brown Company Publishers. URL: https://s3.amazonaws.com/arena-attachments/414578/f886cffe9297937b28dbf0da2a5b8a13.pdf.

[23] *The JavaScript Object Notation (JSON) Data Interchange Format*. URL: https://www.rfc-editor.org/rfc/rfc7159 (visited on 03/23/2023).

[24] *The JSON data interchange syntax*. Dec. 2017. URL: https://www.ecma-international.org/publications-and-standards/standards/ecma-404/ (visited on 03/23/2023).

[25] *TXB0104 4-Bit Bidirectional Voltage-Level Translator With Automatic Direction Sensing and ±15-kV ESD Protection*. Texas Instruments. 2006.

# Declaration of Authorship

I hereby declare and confirm that this project is entirely the result of my own original work. Where other sources of information have been used, they have been indicated as such and properly acknowledged. I further declare that this or similar work has not been submitted for credit elsewhere.

| | |
|---|---|
| Felix Kriegsmann | Date |