Sebastian Saner, 4A

BUILDING AN ANALOG SYNTHESIZER

SUPERVISION Christian Freiburghaus

MATURA PAPER 2020

Table of Contents

1 Abstract

My Matura project is about building an analog synthesizer. This paper will briefly cover the history of the invention and the use in music of the synthesizer. In the main section, I will explain my planning and building. Additionally, I will explain all the electronics and the code I wrote. Going into this project, I already had some experience in analog electronics and synthesizers, altough not on a scale this large. At the end, I will talk about the difficulties and errors which showed up while building. I will cover which ideas were worthwhile and which could be done differently in the future.

2 Introduction

2.1 History & Inspiration

Since the first electrical synthesized sounds by the Theremin in 1920 the world has seen many ways of making sounds out of voltages and currents. Different methods span from additive to subtractive and even abstracter methods like frequency modulation synthesis.

Synthesizers mainly synthesize sound waves. If a sound is inspected on the frequency spectrum, it consists of a base frequency and its overtones with higher frequencies. Besides the main frequency which gives the sound its pitch, the things that describe the overall timbre or colour, are its overtones also called harmonics. Those higher frequencies in the sound are mostly integer multiples of the base frequency. The different amplitudes of those overtones appear as different sounding timbres. This is also the reason why for example the sound of a violin can be distinguished from a saxophone. In additive synthesis different sine waves are added up to produce a specific sound. On an electrical level this is hard to accomplish. In electronic organs like the Hammond Organ from 1935 this was done by using close to a hundred spinning mechanical tonewheels, each producing a single sinewave. Those were then added up to simulate overtones in a played note.

A different kind of synthesis called FM Synthesis or frequency modulation synthesis also became popular half a century later with the first digital synthesizers like the DX7 in 1983. The DX7 used operators (sine wave oscillators) which modulated eachothers phase. This type of distortion created glassy of bell-like overtones in the sound.

Subtractive synthesis approaches overtones from another point of view. Instead of adding up single frequencies, we start out with a lot of overtones and then remove some of them using filters. Most early subtractive synthesizers were big studio sized machines often using vacuum tubes to generate sound. All of this changed in 1964 when the American Engineer Robert Moog invented his Moog Modular system. The Moog Modular was smaller and more affordable than its predecessors. It also allowed the user to connect a keyboard, which made the Moog Modular very attractive for musicians.

Picture 2.1 A Minimoog Model D

The Moog Modular was still very expensive and only accessable for a few individuals. This changed when Moog released the Minimoog Model D in 1970, which quickly became one of the most popular synthesizers around the world. The Minimoog was essentially a miniature version of the Moog Modular. It featured the basic subtractive signal flow and a built-in keyboard. It also was the first synthesizer to feature a so-called pitch-wheel to bend notes like on a guitar. It was constructed in suitcase-like design which made it convenient for live shows. The Minimoog's popularity was huge because it was not only used by the new upcoming progressive rock bands but also in every other genre. The Minimoog defined the sound of experimental bands like Tangerine Dream or Kraftwerk. Its simplicity and its warm fuzzy sounds made the Minimoog a success.

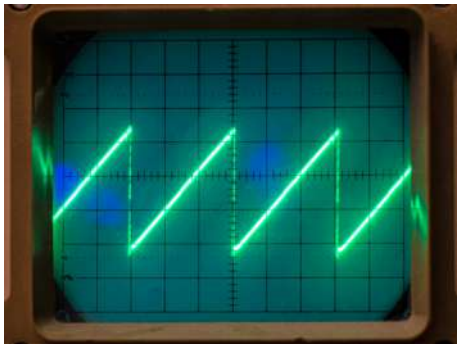
A downside of synthesizers back then was monophony. Because there was only one signal path, only one note could be played at a time. Later, when electronics got smaller and more stable, the first polyphonic synthesizers were made, with the most famous of them being the Sequential Circuits Prophet-5 from 1978. The Prophet-5 featured five voices which were identical copies of a monophonic subtractive synthesizer. This allowed five notes to be played at once. This and the ability to save patches to memory was revolutionizing and inspired many other polyphonic synthesizers later.

As synthesizers became more complicated and incorporated more digital circuits, there was a need for communication between different instruments. Back then, every company used their own way of communicating between devices. In 1981, different synthesizer companies collaborated to invented the Musical Instrument Digital Interface or short MIDI. MIDI allowed to send note and track information from one instrument to another. To this day MIDI is still being used and because it was well thought through and made open source from the beginning, it has never been updated. This makes it possible to connect a synthesizer from the 80s to a computer from today using MIDI, without having to struggle with versions and updates.

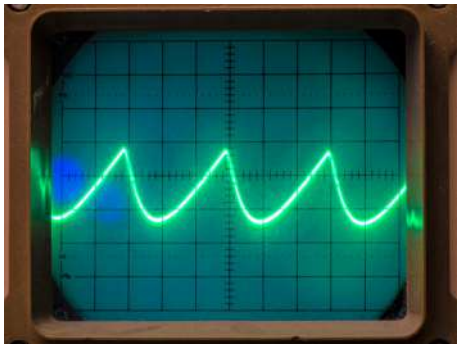
2.2 Subtractive Synthesis

In subtractive synthesis, the main sources of sound are circuits called oscillators. They produce alternating electrical voltages which would result in sound if connected to a speaker. The frequency, i.e. also the pitch of the soundwave is controlled by voltages making this section of the synthesizer the Voltage Controlled Oscillator or VCO for short. The VCOs produce waveforms which are very rich in overtones, like square- or saw-shaped waveforms, which are very easy to synthesise using basic electronics.

Because the overtone rich waveforms of the VCO are too harsh for most styles of music, a filter is used. An audio filter lets overtones through or attenuates them depending on their frequency. There are many types of filters used in electronics, however the most popular one used in synthesizers is the low-pass filter or LPF. A low-pass filter will start to filter away frequencies above a certain threshold called the cutoff frequency. No analog filter is ideal, therefore frequencies close to the cutoff get attenuated less. How strongly a filter attenuates the higher frequencies, is called its fequency roll-off. Around synthesizers, the rolloff is commonly given as dB/Octave. The frequency responce of a low-pass filter can be seen in **Picture 2.6** on the next page. The low-pass filter probably gained its popularity, because of its similarity to the opening and closing of the mouth while speaking and therefore sounds very natural. However, there are also other popular types of filters. There is the High-Pass (HP) which passes frequencies higher than the cutoff frequency and there is the Band-Pass (BP) which passes only a narrow band of frequencies around the cutoff and cuts high and low frequencies. In synthesizers, filters can also be voltage controlled, making them Voltage Controlled Filters or VCFs. Some VCFs have another feature called resonance. If a filter is fed back into itself, it starts to resonate. Similar to mechanical resonance, electrical resonance has the form of a sine wave centered around the cutoff frequency of the filter, like in **Picture 2.6**. Resonance can enhance the sound and also exaggerate the effect of the filter.

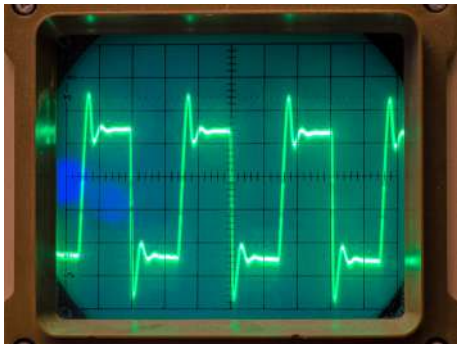


Picture 2.2 A sawtooth wave from an oscillator of my synthesizer.



Picture 2.3 The same sawtooth wave passing through the filter in lowpass mode.

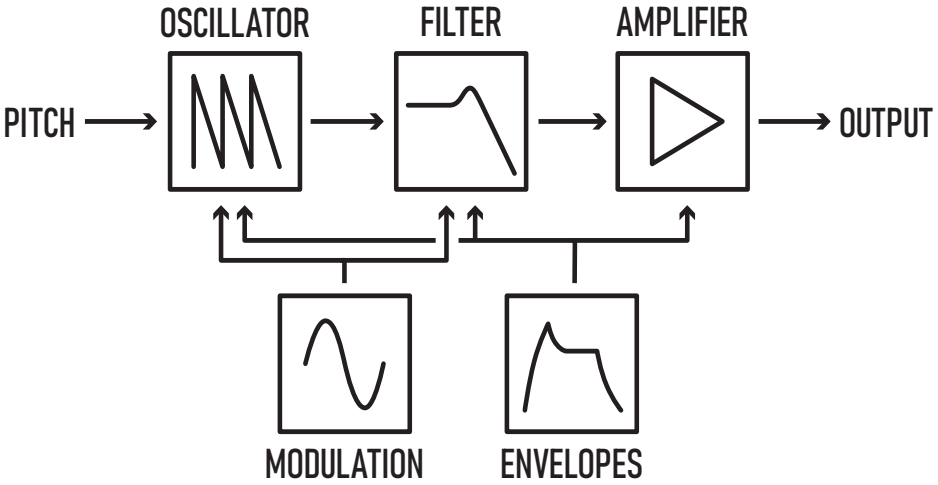
At this point our synthesizer has some oscillators and a filter. However, some important parts are still missing. As said earlier, the VCF can be used to simulate the opening and closing of the mouth while speaking. Therefore, we need something to change the cutoff of the filter over time. This is commonly done by an Envelope Generator. An envelope does not consist of paper, but rather of a voltage changing over time. An ADSR evelope can be seen in **Picture 2.7**. If a key is pressed on the keyboard, the envelope starts to rise to a maximum level. The time this process takes is called attack time. After that, the decay time shows how long it takes the voltage to drop to a sustain level. The envelope stays on this voltage until the key is released. Then the release time shows how long the voltage has to again reach the lowest level. There are also other types of envelopes in synthesizers but the ADSR (Attack, Decay, Sustain, Release) is the most common one and follows the dynamics of most instruments very well. By using an ADSR to modulate the cutoff frequency of the filter, we can give the filter a large amount of expression. Now it is possible to simulate the short plucking sounds of a guitar but also the slow attacking sounds of strings.



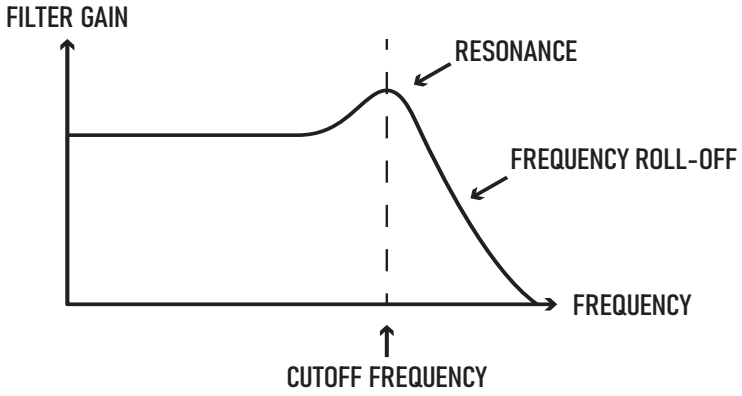
Picture 2.4 A square wave with filter resonance turned up. The sharp edges of the square wave produce these oscillations, which we hear as the resonant frequency.

In addition to the filter, it is common in most synthesizers to have an amplifier which changes the loudness of the sound. The amplifier is also voltage controlled, making it the Voltage Controlled Amplifier or VCA. The VCA is also modulated with an additional ADSR. This allows the note to only be audible when a key is pressed, because oscillators just continuously oscillate.

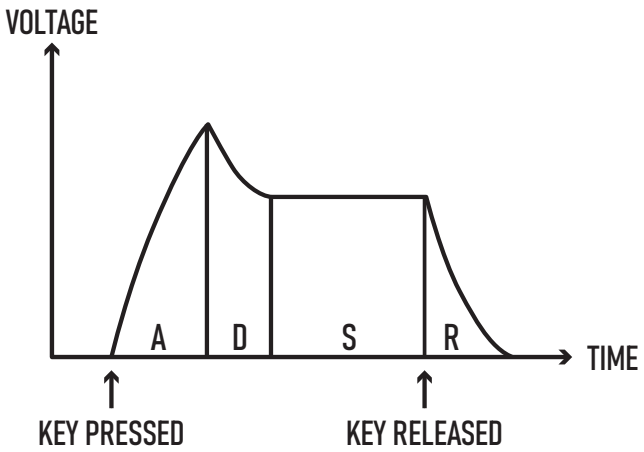
Aside from those necessary features, most synthesizers have additional features to modulate pitch or cutoff to shape the sound further. The most common feature is the LFO or low frequency oscillator. This oscillator is very similar to the VCOs, but the frequency is below the audible range (<20Hz) and its signal is not used as a sound source. When the voltage of the LFO is used to modulate the pitch of the oscillators, the effect is very similar to vibrato on a stringed instrument. When applied to the filter cutoff, the resulting effect is a tremolo.



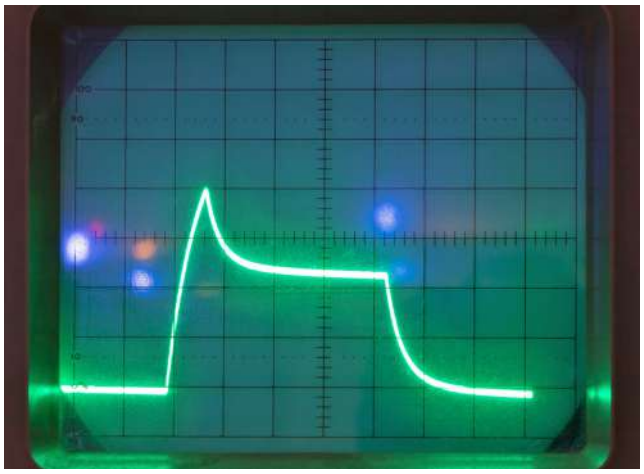
Picture 2.5 The basic signal flow of a substractive synthesizer.



Picture 2.6 A frequency responce diagram (also called bode plot) of a low-pass filter. Notice the resonance which increases the amplitude around the cutoff frequency.



Picture 2.7 A diagram of an ADSR envelope. Here you can see the attack, decay and release time and the sustain level.



Picture 2.8 This is a photo of the actual envelope from my synthesizer being displayed on the oscilloscope. The envelope changes in an exponential manner, because a capacitor is being charged or discharged.

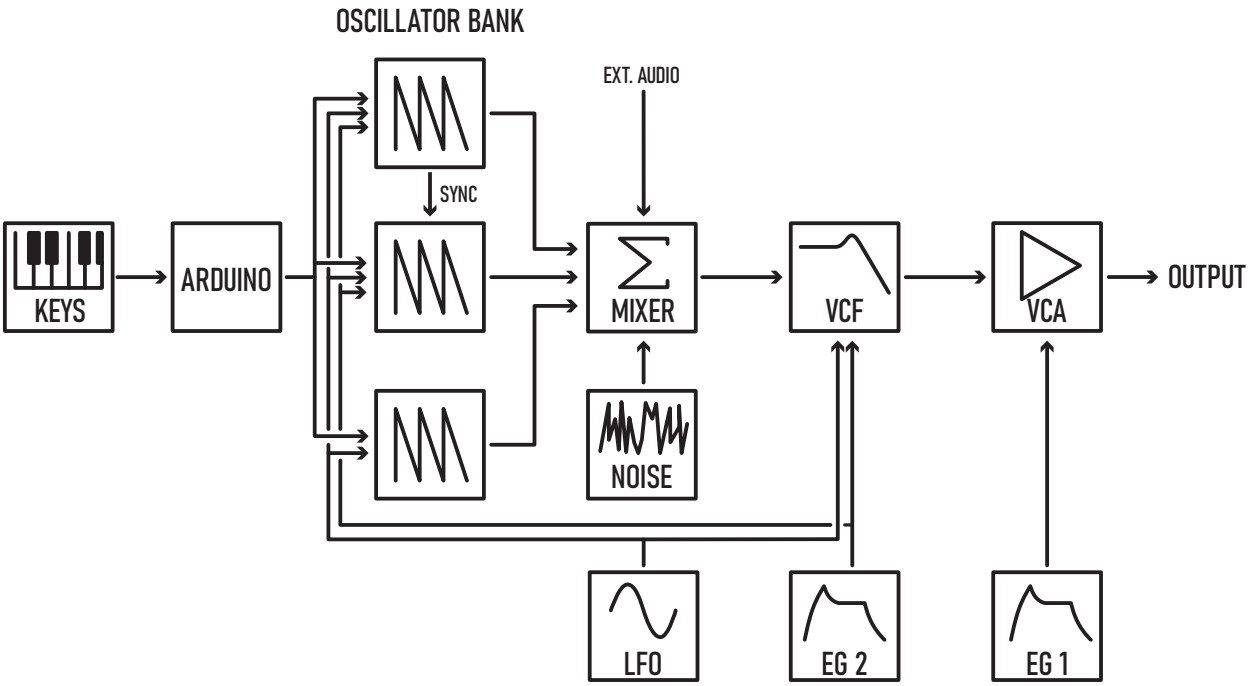
3 Main Section

3.1 Planning

I was always inspired by the Minimoog and the old Moog Modular Synthesizers. Therefore, I decided to recreate something similar. Because I did not want to make an exact copy of the Model D, I took inspiration from the design and construction more so than from the electronics. The Minimoog was constructed out of wood and had its main panel mounted on a smaller metal cabinet, which could be rotated upwards. The main front panel was black, and all labels and dials are written on in white.

For the electronics, I wanted to add a lot of features to allow more possibilities. One problem in old synthesizers was tuning instability. Some components like transistors and capacitors strongly drift in value when the temperature changes. This could change the pitch of the oscillators slightly, causing instability. To completely avoid this problem, I planned to use CEM3340 chips (the same used in the Prophet synthesizers), which should be largely temperature compensated and therefore more stable. Next up was the choice of filter. The first thought would be to use the classic Moog Ladder Filter, which was used in the Model D. However, the Ladder Filter loses a lot of the original signal when turning the resonance up. Another alternative was the filter from the Steiner-Parker Synthacon, which was built in 1975. I settled on choosing the Synthacon filter for my synthesizer. I planned my synthesizer to feature ADSRs to allow more versatility and ease of use. One to control the VCA and one for the filter or additional features like oscillator pitch. I also planned that the LFO should be able to control the pitch of the oscillators and the filter cutoff.

To build a synthesizer like the Model D, I obviously needed a keyboard. I found an old MIDI Keyboard second-hand for a low price. From this point, I started modelling my synthesizer in the 3D design software Blender. There was no real reason to use Blender for this step, I am just very comfortable using it and it allows me to preview the final product with the right materials. My synthesizer is mostly built out of wood. The main panel, the back panel and the one with the pitch and modulation wheel are cut out of aluminium. Aluminium is perfect for building synthesizer panels. It is easy to drill and paint afterwards.



Picture 3.1 This is the signal flow which I planned for my synthesizer.

3.2 Keyboard & Digital Section

To simplify a lot of the keyboard control and to allow MIDI messages to be received and sent, I chose a microcontroller to handle most of the digital logic. A microcontroller is like a very small computer which can be programmed to do tasks. For my project I chose the Arduino Nano. The Nano carries the same specs as the popular Arduino Uno, but is smaller and more practical for being placed onto circuit boards.

3.2.1 Keyboard

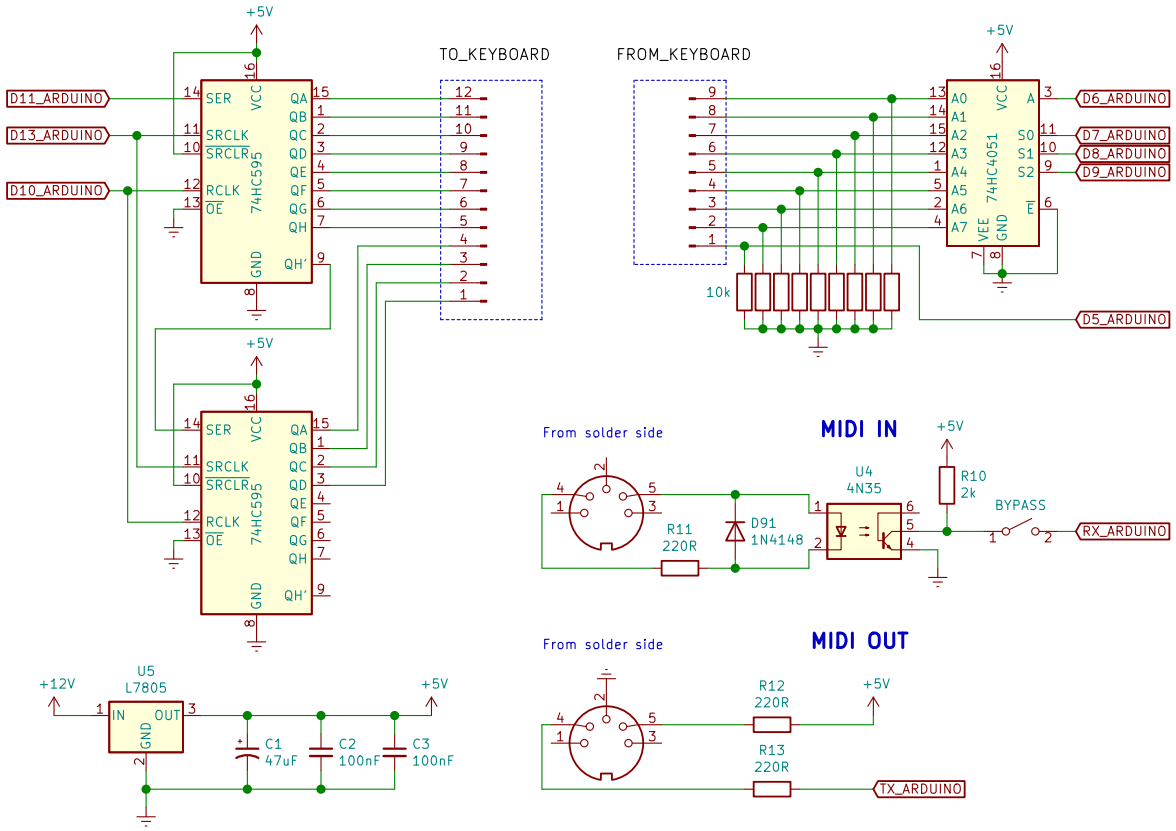
As said earlier, I scavenged my keyboard out of an old MIDI keyboard, with 49 velocity-sensitive keys. Velocity on keyboards is a way of telling how fast a key is pressed. Digital keyboards measure the time of the keys initial press and the time of the key fully being pressed down. This very short time difference can be used to calculate the velocity of the keypress.

After removing the old circuit board from the keyboard, I was left with two connectors. One with 9 pins and one with 12. After some researching and testing, I found out that the two connectors act as rows and columns for a button matrix. The matrix of my keyboard can be seen in the Appendix on page 22. The button matrix consists of a grid of buttons. Columns and rows of buttons are connected. Each button, when pressed, connects a single line from the 12-pin connector to the 9-pin connector. The rows and columns are individually multiplexed using a microcontroller. This means the microcontroller quickly checks all combination of lines. If there is a connection, the position of the button can be calculated. Notice how there is a small diode after every button. This avoids shorts when more than one button is pressed at once.

This button matrix consists of 98 buttons, which makes up two buttons per key. As explained earlier, two buttons are used per key to detect velocity. The last column only consists of two buttons, which makes the code a little more complicated. If the manufacturer had chosen a matrix which was 7 by 14, it would have made the code a little cleaner.

However, why could you not connect each key to a separate pin on the Arduino? The answer is saving space. There are not a lot of pins on a microcontroller and space must be used efficiently. With this setup we use only 21 IO-Pins on the Arduino. Otherwise we would have used 98. Still, the Arduino does not have 21 IO-Pins. Therefore, we use multiplexing and registers. For my synth, I used two 74HC595 shift-registers. Those registers can be hooked up to the Arduino using only 3 Pins. The Arduino sends the data on a serial line to the registers. Each register can output 8 bits. I needed two of them to connect to the 12 input lines on the matrix.

For the output of the matrix I used a 4051 8-bit multiplexer. This chip can be used to select one address out of all outputs using binary. In this case, I used 3 bits to select one of the 8 output addresses. Because there are actually 9 outputs, I just connected the last one directly to an IO-Pin on the Arduino. Before going into the multiplexer, the pins need to be pulled down using a 10kΩ resistor on each line, so that they show ground potential when they are not connected. Otherwise the pins would be floating and showing wrong outputs. Altogether, I used 8 Pins of the Arduino for the keyboard circuitry.



Picture 3.2 All of the digital electronics

3.2.2 MIDI

Because I wanted to add full compatibility to my synthesizer, I needed MIDI connections. Building a MIDI input for an Arduino is straight-forward and a library can be used to simplify the coding process. MIDI works on a slave-master principle. This means, if a MIDI-cable is plugged from the MIDI-out of one instrument to the MIDI-in of another, the first instrument has full control over the other.

Therefore, I needed to use two different circuits for input and output. A MIDI output is simple to build and uses only two resistors. However, on the MIDI input circuit, I needed a few additional components. To prevent ground-loops between instruments, I needed to electrically isolate the signal in the circuit. This is done by the 4n35 optocoupler. An optocoupler uses a combination of an LED and a phototransistor on its inside, to transmit a signal using light. This isolates the two keyboards from each other.

The input and output circuits use the two serial pins RX and TX on the Arduino. Because these pins are connected to the USB interface and are used by the programmer, the Arduino cannot be programmed while the MIDI input circuit is connected. Therefore, I added a small switch on my circuit board to disconnect the MIDI-input while the Arduino is being programmed.

3.2.3 DAC

My synthesizer is mostly analog, but the keyboard and the Arduino are storing notes digitally. This meant I needed a way of converting digital numbers to analog voltages. Therefore, I used a digital to analog converter or DAC, which does exactly that. There is a big range of different DACs for different tasks. For my purpose I used an MCP-4728 breakout board. This board has 4 channels which could simultaneously send out voltages to my synth. I only needed two separate signals for pitch and velocity, but I still went for the 4-channel option. The Arduino communicates to the board using I^2C. I^2C is another protocol used by microcontrollers to communicate with each other. To connect my DAC, I needed to connect pins A4 and A5 on the Arduino to the breakout board.

3.2.4 Code

Programming the Arduino to do all the abovementioned tasks was a challenge. In total, my Arduino sketch stretched over 500 lines long and featured multiple classes. The entirety of my code can be found in the appendix. First, I started to interact with the MIDI keyboard I salvaged. The first lines I wrote were checking notes I pressed on the keyboard. This was to make sure the keyboard and digital logic all worked. This was done using two nested for-loops to step through the matrix and read the state of each button.

After a lot of debugging and troubleshooting, I had the Arduino display values on the screen. I feared that my digital electronics would not be reliable enough and miss pressed notes, but the way I wrote the code made the measuring very reliable and it worked surprisingly well. I put the whole loop into a function to be used later.

Now it was time to start working on a main loop in the program to step though all different tasks the Arduino has to manage. After some planning, I decided to switch between three modes in the code. MIDI-input, normal play and arpeggiator. During MIDI-input mode the keyboard should not respond to any keypresses. Only incoming MIDI messages should trigger notes. I decided this would be optimal, because it simplified the code and reduces possible bugs. During normal play, the keyboard responds to pressed keys. It also acts like a MIDI keyboard, which sends note-on and note-off messages. Pitch bend and modulation wheels are also scanned and sent out using MIDI.

The third mode is arpeggiator mode and was the hardest to implement. In an attempt to tidy up my program, I put all arpeggiator related things into a class. In a nutshell, the arpeggiator class receives the notes currently pressed. The program then steps through the sequence at a fixed time period and outputs the notes. The hold function of the arpeggiator was also difficult to implement, but I still decided to do it like it was done in synthesizers like the Roland Juno, where a chord can be shortly pressed to set the arpeggiator instead of having to hold the notes down continuously. When activated, the hold function keeps track of the amount of notes pressed at any time. Every time the arpeggiator senses more notes than earlier, it saves them in memory. If all notes are released, the arpeggiator will start playing the maximum amount of notes it saved. Working out the arpeggiator logic took the most revisions and troubleshooting.

3.3 Wooden Frame

I started out the construction of my synthesizer with making the wooden frame first. Therefore, I designed the pieces, like I wanted them to look, in the software Blender3d, where I could preview the overall construction. Then I bought a $1m^2$ piece of 1.2mm thick plywood and marked the shapes I wanted to cut. I decided it would be best to cut the pieces on a big circular saw, because the small jig saw I had at home would give bad results. With the help of my father, who is a teacher at the local school, I could use the big circular saw there to cut my wood to shape. Then, I started to screw the pieces together using wood screws. I decided not to glue the parts, so I could take the construction apart later. With some small wooden blocks, I screwed the keyboard in place. The piece which housed the main electronics should be able to rotate up and down.

Therefore, I mounted it to the rest of the frame using long piano hinges. To prevent over-rotating, I connected the rotating piece to the main frame with a small cloth on both sides.

To be able to rotate the housing for the electronics is not only convenient for use of the synthesizer, but it was very helpful when designing the electronics. When working on the insides I could just rotate the back part upwards, then when testing I could rotate it back down, without any problems.

In the end, after I installed all the electronics, I took the synthesizer apart again for a paint job. The basic plywood I was using was very bright and did not look great on the synthesizer. Therefore, I bought mahogany coloured wood stain and clearcoat. After sanding down the wood, I applied the stain, which could be done in one coat. After drying I added layers of clearcoat and sanded down each one to get a smooth surface.

3.4 Aluminium Panels

3.4.1 Cutting and drilling

My synthesizer uses three aluminium panels. There is the main panel with, all the switches and dials, then there is another one, which houses the pitch bend and modulation wheels, and the last one is on the back, where the inputs and outputs are connected. For every panel, I used around 1.5mm thick aluminium sheet. For the main panel I needed to find a big piece, because the panel is 75cm long. I cut the three pieces to size using my jig saw. After punching points to mark the position of the holes, I used the drill press again at my local school to cut the holes. The places where I had to cut out shapes, like the holes for pitch and modulation wheels, I started by drilling a hole in the centre and then cut out the rest using my jig saw. The three panels are mounted to the wooden body using screws.

3.4.2 Modulation & Pitch Wheels

The MIDI-keyboard I used also featured pitch and modulation wheels. Those wheels were made of ugly gray plastic, therefore I decided to create my own. Fortunately, the metal bracket and the potentiometer from the old wheels could be reused for my own ones.

I decided to 3d-Print the new wheels. Therefore, I first designed wheels with the proper dimensions in Blender3d. My classmate Nico Lenz ownes a 3d-Printer, so he kindly printed out the two wheels out of black ABS plastic.

Picture 3.3 The fully assembled synthesizer. The main panel and the modulation and pitch wheel panel are both visible.

More pictures of the complete synthesizer can be found in the appendix.

3.4.3 Screen Printing

To achieve a vintage-look on my panel, I wanted to somehow print all lines and labels using white on black. The problem is I am not able to just run the aluminium through a printer, obviously. There are different methods of printing onto aluminium, but the one I chose was screen printing. I have done screen printing in the past, so I had some experience doing it. Also, my local secondary school had a lot of screen-printing equipment ready.

I asked my old art teacher Sasha Christener if I could screen print at the school. He agreed and helped me through the process of printing the panels.

In screen printing, the design is firstly printed on a transparent sheet. The parts which where printed black should block UV light from passing and therefore act as a mask.

All my designs where made in Adobe Illustrator, where I could design all the icons, text and lines.

The next step is to find an appropriate screen to print with. The screen has a small grid, which lets paint through, but the grid also holds the emulsion which blocks some areas from passing the paint. Important is the size of the screen, which must be wide enough for the panel it is used for. There are also different grid sizes for screens. Finer grids are often used for high resolution images, which was not necessary in my case, because I printed only black on white text.

In a dark room, the screen is then coated in a photo-emulsion. This liquid hardens when exposed to ultraviolet light. After this emulsion has dried, the transparent sheet with the design is placed on top of the screen and exposed to light. The light should contain UV-light, so it is best to use something like a metal-halide bulb. The sun can also be used to expose the emulsion, but timing can get difficult, because the strength varies depending on weather and time.

The exposure time is very important for the quality of the screen. When washing the unexposed emulsion off later, it should come off easily and the exposed emulsion should stay in its place. Exposing too briefly makes the exposed parts come off and exposing for too long lets the UV seep under the transparency or reflect from the screen and expose the parts, which should stay unexposed. Finding the right exposure time can not really be calculated and must be done experimentally. A black paper can be used to sequentially block parts of the screen during an exposure to determine the best exposure time. In my case, I overexposed far too much on my first tries. After some more attempts, I settled on 45s exposure time, which is rather short for screen printing.

After the unexposed parts are washed off, the screen can be dried. The screen now acts as a mask for printing.

Before printing on my panel, I sanded it down well and sprayed it black using spray paint. Then I printed all my panels using white screen-printing paint. The screen is filled with paint and placed over the substrate. Then the paint is pressed through the screen onto the aluminium using a squeegee. After drying, I finished the surface off using glossy clearcoat out of a spray can.

Picture 3.4 The screen being exposed under UV-light.

Picture 3.5 The finished screen is ready for printing the labels for the wheels-panel.

3.5 Analog Electronics

When electronics get mass produced in factories, printed circuit boards are specifically made for every circuit. This would not be necessary for my circuits because I am only building one of each. Also, prototyping would become more difficult because a lot of changes must be made all the time. Therefore, I am using stripboard for all my circuits. Stripboard has holes spaced in a grid, where components can be placed. On the backside are strips of copper, which span horizontally over a board. If a connection must be removed, a drill bit can be used to scrape away the copper. I built all the analog circuitry on stripboard and used jumper cables if the connections on the back side were not enough. It is always good practice to be able to remove the boards from the synthesizer. Therefore, I did not just solder all cables directly to the board, I instead used connectors for every connection. I crimped female Dupont headers onto the cable and soldered male headers to the circuit boards.

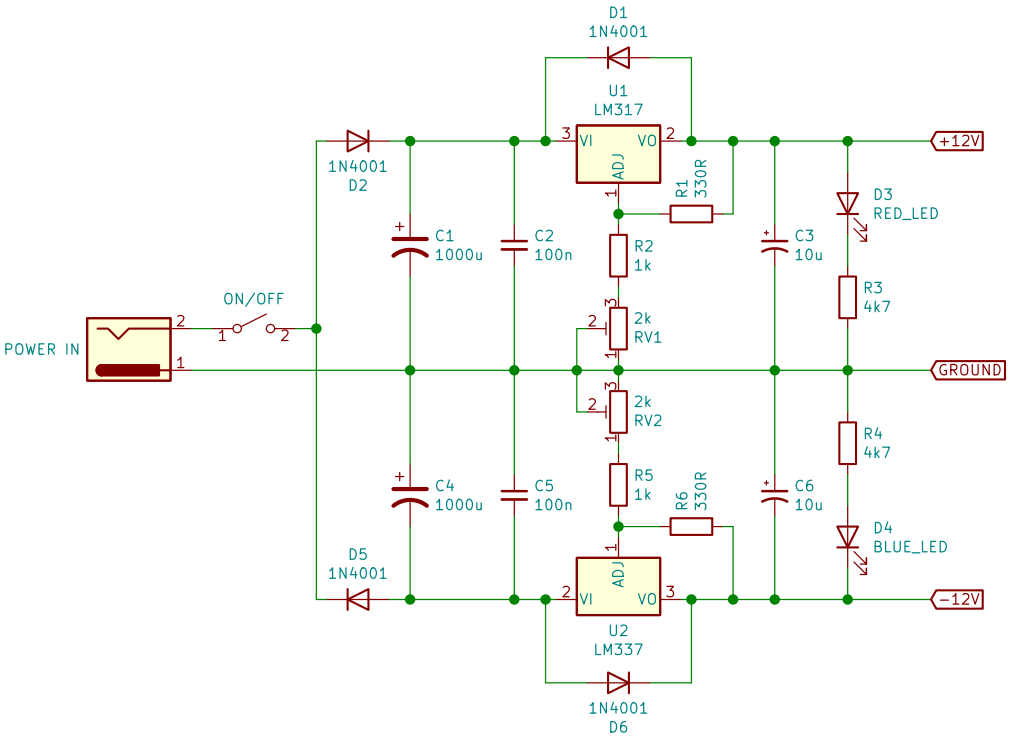
I mounted the boards onto the backside of the aluminium panel using small Nylon standoffs, which could be easily removed, if a change had to be made.

3.5.1 Power Supply

Every electronic project needs a power supply. In synthesizers, power supplies often have a positive and a negative rail, e.g. +12 volts and -12 volts. Those dual rails are mostly used because of operational amplifiers. They amplify signals which are centred around 0v, this means they alternate between negative and positive voltages. The op-amp therefore needs the negative rail to be able to amplify a negative voltage.

For my power supply, I roughly followed a design I found online from syntherjack.net. This design uses a transformer to reduce the 230 Volts AC to 12VAC. Then the AC is converted to DC and stored using capacitors. The rough DC is smoothed out using voltage regulators. Most other power supplies use 7812 and 7912 IC as positive and negative 12 Volt regulators. In Syntherjacks design, they use LM317 and LM337 ICs, on which the voltage can be exactly adjusted using a trimmer.

When planning my synthesizer, I wanted to avoid working with high and potentially lethal voltages. Therefore, I used an external 12 Volt AC wall adapted, which I found on eBay. This made me reduce the full bridge rectifier to a half wave rectifier, because I only had a single AC input and not the dual secondary coil from a transformer. The rest stayed mostly similar to Syntherjacks design. Because I did not have fitting heatsinks for the two regulators, I cut off a big piece of aluminium from an old heatsink and mounted it to the frame. I build my circuit on stripboard and added female headers for the outputs. The AC input is a small jack on the back panel, which is connected in series with an on-off switch.



Picture 3.6 The schematic of my power supply. The two regulators are mounted on a big heatsink for protection. The voltages can be accurately set using the two trimpots. The two LEDs are built onto the pcb and are only used for prototyping. In case of a short the LEDs will go out, which makes it easy to spot.

3.5.2 Oscillator Bank

Oscillators are the main source of sound in a synthesizer. The main input for an oscillator is pitch, the output is always a waveform. In a synthesizer, oscillators are voltage controlled, making them Voltage Controlled Oscillators or VCOs. Signals and values are always represented in voltages. Pitch as voltage is often given using Volts/Octave. 1 Volt difference in input makes up one musical octave, which also represents multiplying the frequency by 2. The advantage of Volts/Octave is its exponential behaviour. Our ears perceive frequencies in a exponential manner. Measured in Hertz, differences in higher frequencies are less noticeable than in lower frequencies. Some Korg and Yamaha synthesizers used another representation of pitch called Hertz/Volt, which was less popular due to its linearity.

On an electrical level, a VCO is a basic relaxation oscillator: A capacitor is charged up by a current. If the voltage reaches a certain level, the capacitor is discharged and the process restarts. This results in a sawtooth waveform, which will later be shaped into other waveforms. The rate of this oscillation is controlled by the current charging the capacitor. Because this rate is proportional to the current and the notes should follow an exponential manner, the linear input voltage needs to be converted. This is usually done using an exponential converter circuit. The converter takes in the volt/octave inputs and outputs a current, which charges the main capacitor.

The different output waveforms are made using some op-amps configurations. For example, a square-wave can be created by comparing the sawtooth to a constant reference voltage. If the saw wave is higher in voltage than the constant one, the output of the comparator is high, otherwise it is low. This results in a square wave whose duty cycle can be set by changing the reference voltage. A triangle wave can also be easily made from a sawtooth. By taking an inversion from the original sawtooth wave and combining it with the original in the right way will result in a triangle wave.

These types of analog oscillators are often the biggest cause of problems in synthesizers. Mainly because the human ear is very well trained in hearing musical pitch. Even a small difference in pitch can be heard. The problem with analog circuitry is component values drifting, which happens if the temperature changes. Therefore, most oscillators feature some sort of temperature compensation, which can be difficult to achieve, but it can almost completely remove pitch drift.

To avoid all the above-mentioned problems, I decided to work with the complete VCO inside a chip, the Curtis CEM3340. Curtis chips were first introduced in the production of the Prophet synthesizers. The advantage of the CEM3340 is the very easy to achieve temperature stability. Additionally, the chip takes up far less space than any other analog oscillator.

The CEM3340 features a Volt/Octave summer, two oscillator-sync inputs and a linear FM input. The three output waveforms are triangle, sawtooth and pulse with an additional pulse-width input. The CEM3340s datasheet shows the basic arrangement with all the features. For my three oscillators, I removed the linear FM input and the sync inputs. I instead used the special sync circuit, which is shown in the datasheet to achieve a more classic hard-sync sound. The sync only goes from oscillator 1 to oscillator 2. The volt/octave summing note acts like the virtual ground on an op-amp. It allows to connect all volt/octave inputs using 100kΩ resistors.

For the output waveforms, I used different resistors as voltage dividers. This allowed me to adjust the level of each waveform, so that all sounded equally loud. I did this through testing on a breadboard.

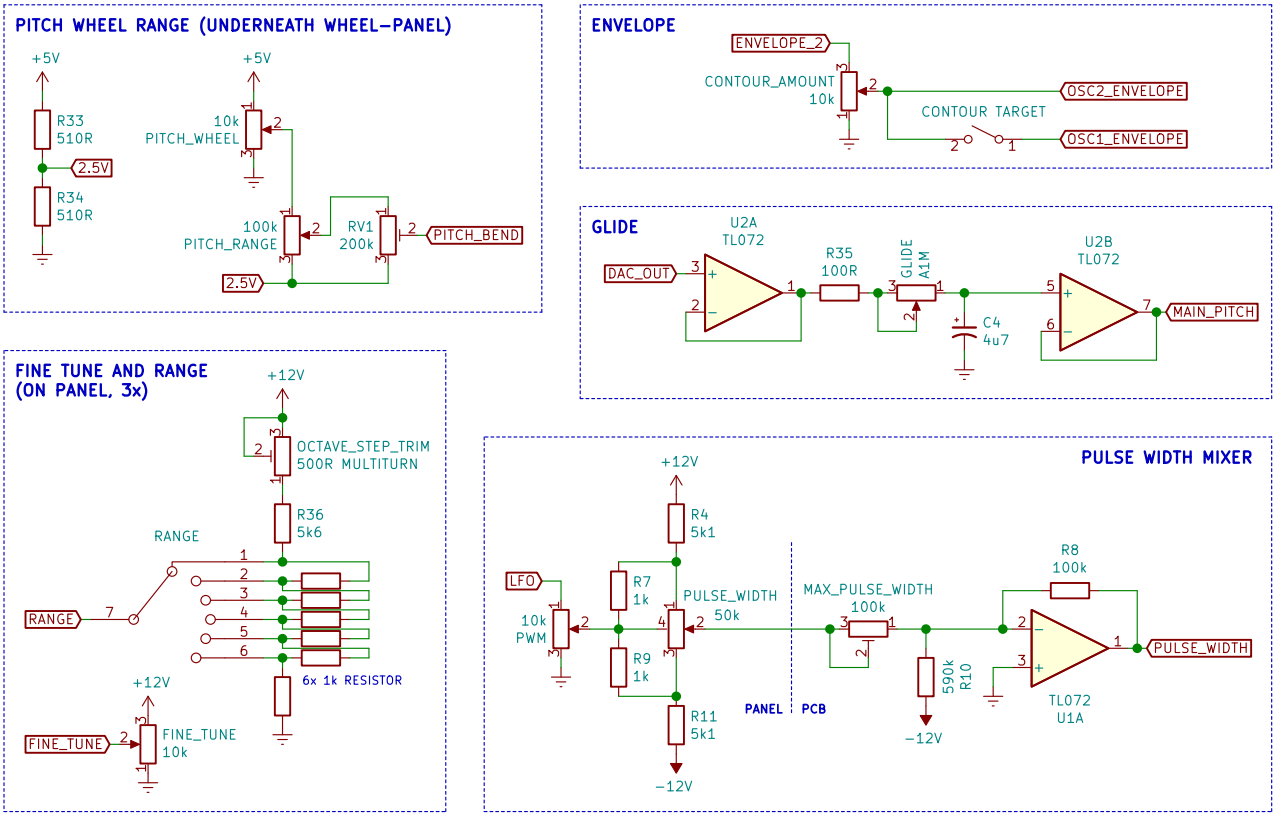
All pitch modulation and pulse width inputs can be adjusted or turned off on the front panel. The sync can also be turned on or off using a switch.

Each oscillator has a rotary switch labelled as range. This switch is used to offset each oscillator by one or more octaves. A simple resistor ladder and a trimmer gives voltage references from 0v to 5v, which can be selected with the switch. There is also an additional tuning pot. It can be used to slightly offset the oscillator to adjust tune or also offset a single oscillator by a few semitones. Each oscillator also has a rotary switch to select the waveform. For the first and the second oscillator, I used the three waveforms and an additional waveform consisting out of a mix of sawtooth and triangle. This waveform was also used in the original Minimoog. On the third oscillator, I did not use this triangle-saw mix, however, I used an op-amp and two diodes to convert the triangle to a sinusoidal waveform.

Another popular feature in synthesizers is glide (also called portamento). Instead of hopping directly to the new pitch if a key is pressed, glide makes the pitch slowly transition to the new note. This glide circuit can be easily built using a dual op-amp and a small RC low-pass filter. A capacitor is charged to the pitch voltage through a potentiometer. The variable resistance will change the charging speed of the capacitor.

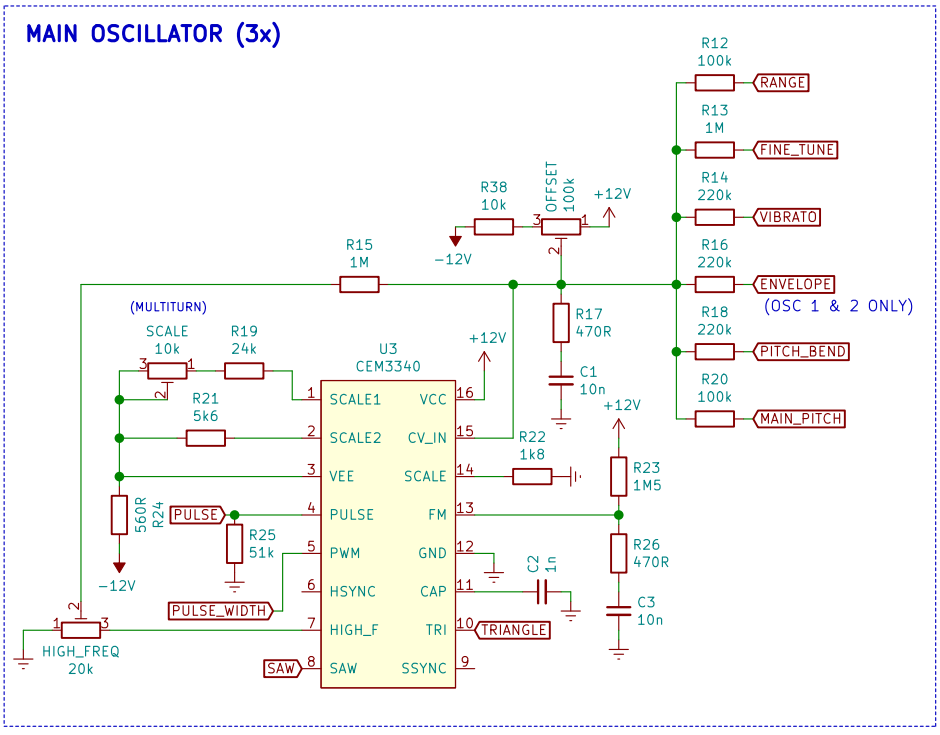
Mixing together the pulse-width voltage with the pulse-width-modulation is approached differently in my synthesizer than in others. Usually the two signals are summed together. The pulse-width potentiometer in my synthesizer is a 4 pin potentiometer. This means there is a center tap on the resistive track. On this center tap the pulse-width-modulation signal is added. I decided this would be handy, because it would stop the oscillator from going silent like it usually does when pw and pwm signals mixed get too large and the duty cycle goes to either 0% or 100%.

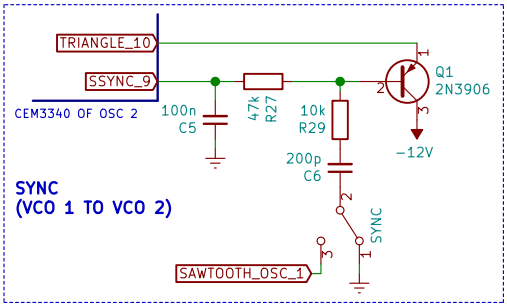
On the panel for the oscillator bank, there is one small error. The switch for the contour target has options 1 or 1&2. This is wrong, because the correct label would be 2 or 1&2. I made this mistake because at the time I was unsure how oscillator synchronisation works. If oscillator 1 synchronizes oscillator 2, the pitch of oscillator 1 should be constant, while the pitch of oscillator 2 changes to get the nice hard-sync effect. But now I cannot change it, because the panel is already printed.

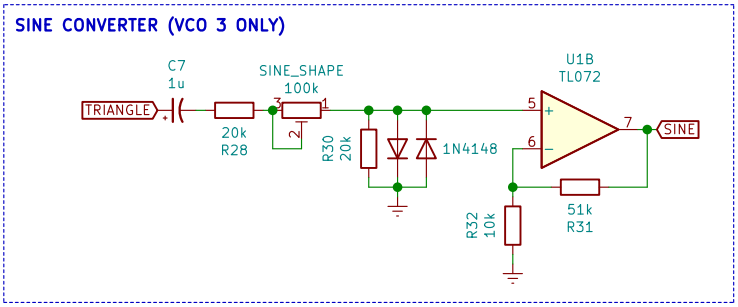


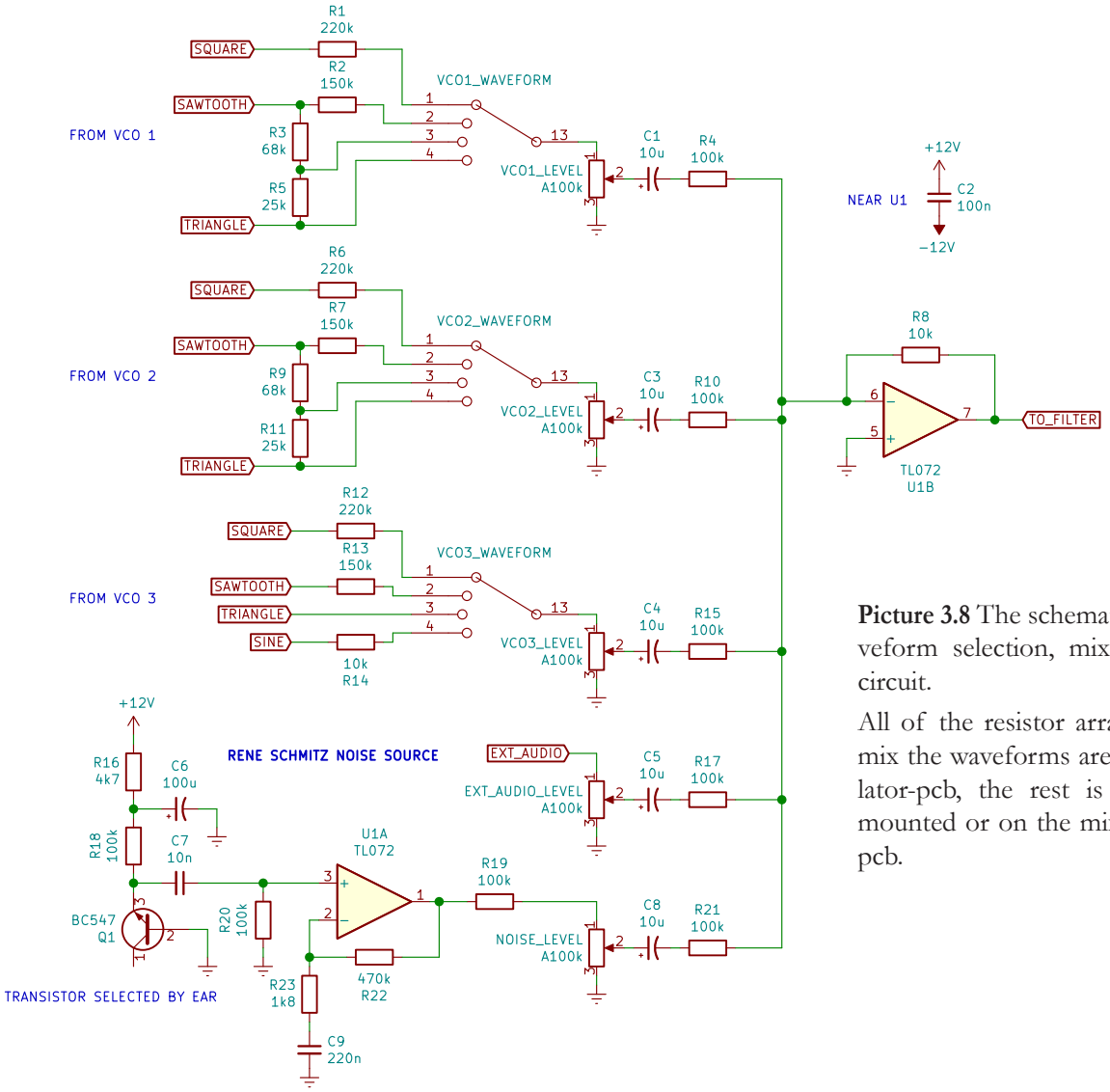
Picture 3.7 These are all schematics, which are related to the oscillator bank. Some features are panel-mounted (directly soldered to the potentiometers). For the range switch, I soldered all resistors directly onto the rotary switch. The trimpot was soldered to a small piece of stripboard and stuck onto the rotary switch with two leads.

This circuit board was the largest hassle to build, because it featured so many circuits. This pcb houses 13 different connectors, which made up a total of 36 cables entering the pcb.









Picture 3.8 The schematic of the waveform selection, mixer and noise circuit.

All of the resistor arrangements to mix the waveforms are on the oscillator-pcb, the rest is either panel mounted or on the mixer and noise pcb.

3.5.6 Mixer & Noise

Before going into the filter, the three oscillator outputs, noise and external audio must go through the mixer. This allows adjusting the volume of each oscillator and lets you turn of an oscillator completely. The change in level can also be useful to add some distortion to the signal, if the sum of the three oscillator waveforms overload the circuitry of the filter.

The three rotary switches switch between the four available waveforms. Waveforms with a high amplitude coming out of the CEM3340 have their signal attenuated using resistors, so that every waveform has the same perceived loudness. The resistor forms a voltage divider using the 100kΩ resistance of the potentiometer. Additionally, oscillator one and two have a mix between sawtooth and triangle waves. This was inspired by the Minimoog Model D, which also features this waveform. The mixing part consists of five logarithmic potentiometers, which attenuate the input signals. DC offset gets removed using capacitors and all inputs are summed up using a inverting op-amp configuration.

Noise is an additional source of sound next to the main oscillators. The noise circuit produces white noise, which has an equal amount and level of all audible frequencies. The noise signal then passes through the mixer as a fourth channel to adjust its level. There is an additional fifth input on the pack panel, which can be used to route an external signal into the mixer.

When I first built the mixer and noise circuit, the noise leaked a lot of interference into other parts of the circuit. This was partly because the noise circuit, which I built after René Schmitz's design, produced far too loud a signal. I also did not use a summing op-amp after the mixer and the signal directly went into the filter. When I finally noticed that this circuit board contained all the noise problems, I completely scrapped it and started from scratch. This time I used and op-amp to sum the signals and adjusted some resistors to reduce the output level of the noise circuit.

3.5.3 Filter

No part inside the synthesizer influences the sound as much as the filter. This makes choosing the right filter an important step. Because I am building a Minimoog inspired synthesizer, the obvious choice would be the classic Moog Ladder filter, which gave the Minimoog its recognizable sound. However, as I already explained, I settled on the Steiner-Parker Synthacon filter. This filter is state variable, which means the filter can be switched between lowpass, bandpass, highpass and allpass mode. Allpass, instead of just adding resonance, also distorts the phase of different frequencies and therefore alters the waveform. The Synthacon filter is a type of diode ladder filter. Diode ladders are somewhat inspired from the original transistor ladder filter from the Minimoog but produce a harsher tone. This filter is very experimental, but it gained a lot of popularity lately, because it was featured in the Arturia Minibrute. A schematic of the filter can be found on the website yusynth.net which belongs to Yves Usson, one of the creators of the Minibrute. I have used the schematics from yusynth.com for a lot of my projects. They are explained well and work nicely.

The Synthacon filter can get very unstable when turning the resonance up all the way. The feedback makes the filter freak out completely, which can produce screaming and scratching sounds. I had to tweak some resistors and add a trimmer to adjust the resonance level, so that the screaming only happens, if the resonance is turned up all the way. This still lets you experiment with the weird behaviours without them being annoying.

I will not be adding a schematic, because my circuit is mostly identical the one on Usson‘s website.

3.5.4 Modulation

Modulation is used in a synth to change parameters over time. Usually this is done with a low frequency oscillator, or LFO. For the LFO, I originally planned to again use one of Yusynth’s schematics. However, I decided to experiment and build my own. So, I started to connect some resistors and capacitors and sketch out my circuit.

The schematic can be seen on the next page in **Picture 3.9**. The oscillation is made by two op-amps and a transconductance op-amp, the LM13700. U1B is configured in an integrating configuration. It integrates the constant current of the LM13700, which results in a rising slope. U1A acts as Schmitt-trigger and produces a square wave, which feeds into O1A. If U1A changes direction, the current from O1A also changes and the slope of U1B reverses. This results in a triangle wave on the output of U1B. The triangle wave is then fed into U4B, which converts it into a sinusoid. This triangle to sine conversion is copied from Yusynth’s LFO design. The advantage of this oscillator is that the output current of O1A can be electrically controlled. This results in a change of frequency of the oscillator. For both O1A and O1B, I_{ABC} is provided by the two voltage to current converters with U3A and U3B. The rate is controllable from the front panel, from the modulation wheels and from the AD-envelope which is built into the LFO. An Attack-Decay envelope rises if a key is triggered on the keyboard. Differently to an AR-envelope, an AD-envelope starts to fall immediately if the attack level is reached, instead of waiting for a key to be released. It consists out of a simple 555-timer configuration which charges a capacitor. The transistor in front of the 555-timer converts the gate signal to a short trigger. The state of the envelope is displayed by an LED on the front panel.

The other half of the dual transconductance op-amp is used for a small additional VCA just for the LFO signal. With the VCA-dry-wet knob, the mix between amplified or original signal can be controlled. This dry-wet knob allowed me to only use one knob on the front panel for the whole VCA part, because I was running a little short on space. The whole LFO now only uses 6 knobs in total.

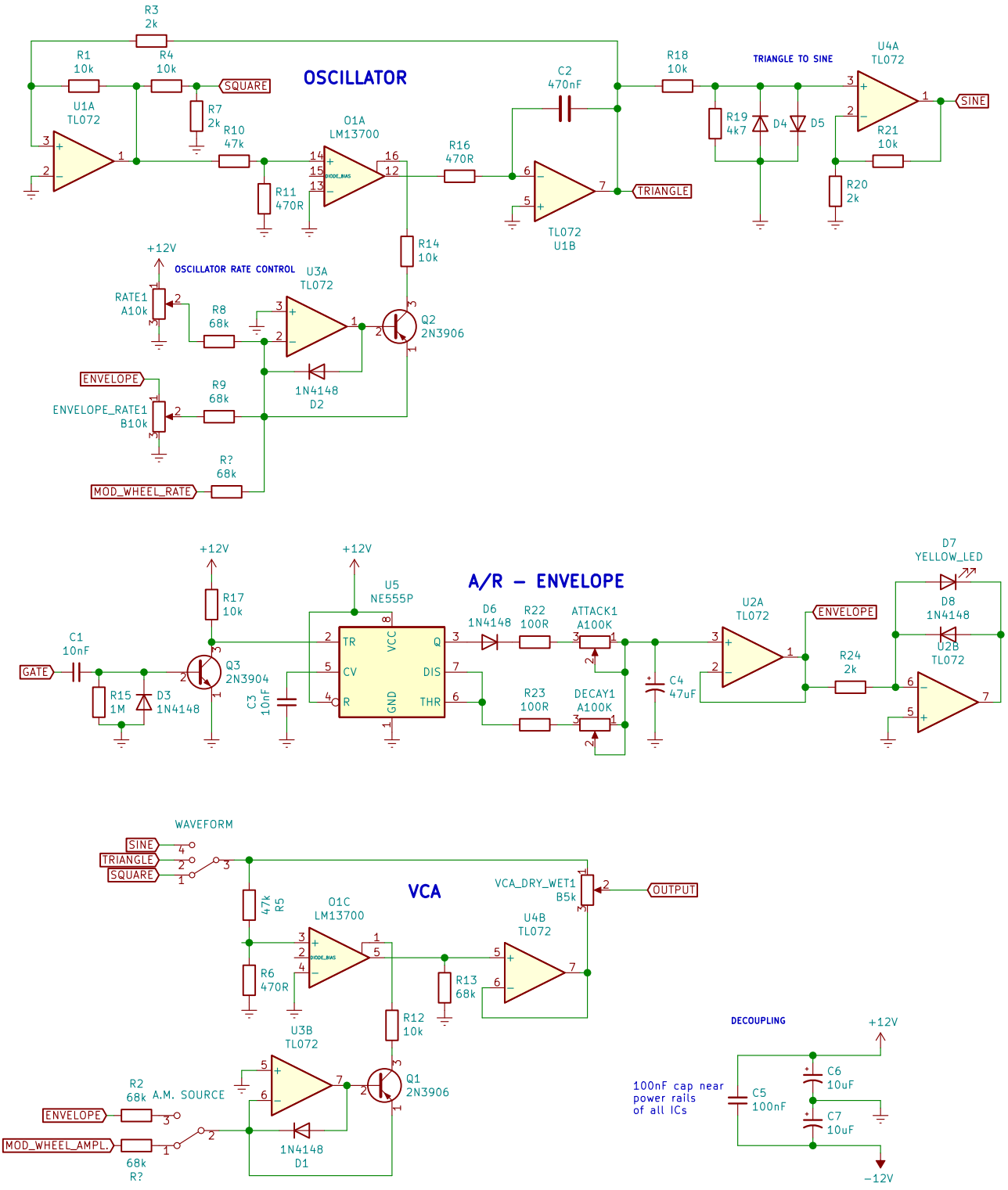
3.5.5 ADSR, VCA & Outputs

For my synthesizer, I planned on building two ADSR envelopes. One for controlling the main VCA, and one for the filter and additional features. Because I was running a little short on space for circuit boards, I decided to build the ADSRs, the VCA and the two outputs all on one stripboard. The schematics for this section can be found on page 16 in **Picture 3.10**.

For the ADSRs, I first thought that I could design my own, however they were much too complicated. Therefore, I used Yusynth’s ADSR design with some small tweaks to fit my design. I firstly reduced the input stage to two transistors instead of one. This resulted in some trouble with the envelope, so I had to change it back to three transistors. I still only had to use one input stage for both ADSRs, which made me save some space on the board.

For the VCA, I used René Schmitz’ design from his website schmitzbits.de. I chose this so that I could save even more space, because this design mostly uses transistors. After the VCA, there is a bypass switch. This switch allows the user to completely bypass the VCA, which can be useful in some uses. Therefore, U2A must amplify the signal to a level which gives the VCA unity gain, so the level does not change if the VCA is bypassed.

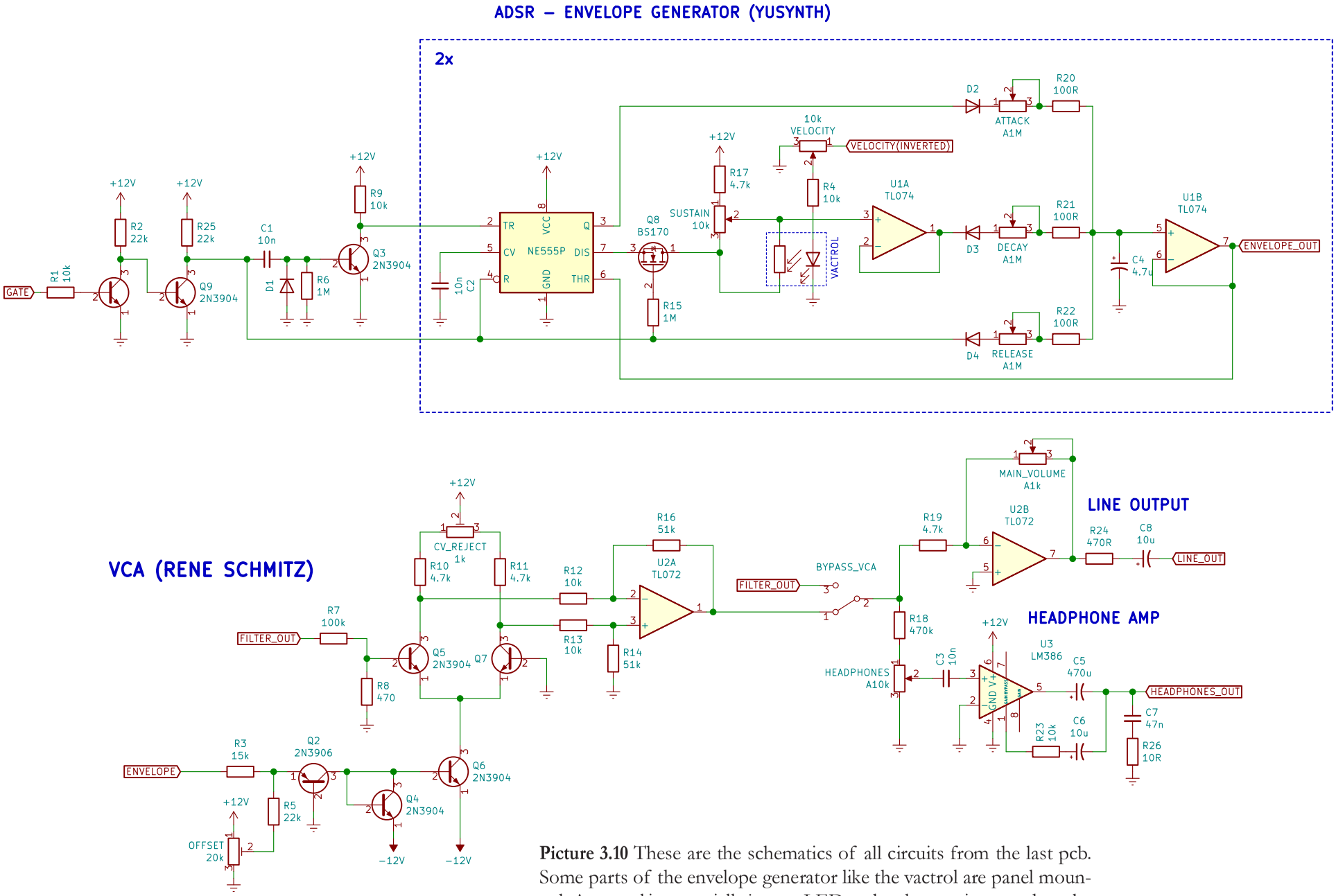
The line output lowers the signal to line level and lets you change the volume with the main volume knob. For the headphone amp, I used the LM386, which is an easy to use amplifier. I had to tweak some values to make it work correctly and not cause a lot of noise.



Picture 3.9 The schematic for the modulation section.

This circuit board was easy to build, because I already prototyped it on a breadboard. I put the LED into a 5mm hole on the panel and secured it with hot glue.

The amplitude source switch was implemented later, therefore I had to drill a hole below the other features on the front panel. It also is missing its label, which I wont be able to print on with everything already being installed.



Picture 3.10 These are the schematics of all circuits from the last pcb. Some parts of the envelope generator like the vactrol are panel mounted. A vactrol is essentially just an LED and a photoresistor, and can be made by putting the two parts face to face into a heatshrink.

3.6 User Manual

Due to the many features of my synthesizer and demand, I decided to create this user manual. I will go over the six main sections of the synth.

3.6.1 Mode

The Mode section is used to control the digital section. Using the rotary switch in the middle, one of three playing modes can be selected. In the first mode, called MIDI, the synth only respondes to incoming MIDI notes. The second mode lets you play normally, using the keyboard. The synthesizer will send out MIDI note information from the keys pressed on the keyboard. Additionally, CC commands for pitchbend and modulation are sent, if the wheels are used. The third mode consists of four different Arpeggiator sequences (UP, DOWN, UP-DOWN & RANDOM), whose rate can be set using the rate knob. Using the HOLD switch, notes which where pressed will be remembered and will arpeggiate. Whilst the arpeggiator is running, clock signals are sent out from the clock output on the back panel. If an external clock is connected to the clock-in socket, the incoming signal will overwrite the internal clock if the rate is higher. This means the synth automatically chooses the faster clock. Therefore, if an external clock is wanted, the internal clock rate can be turned to zero using the rate knob. The gate-signal of the synth is also available on the back panel.

3.6.2 Oscillator Bank

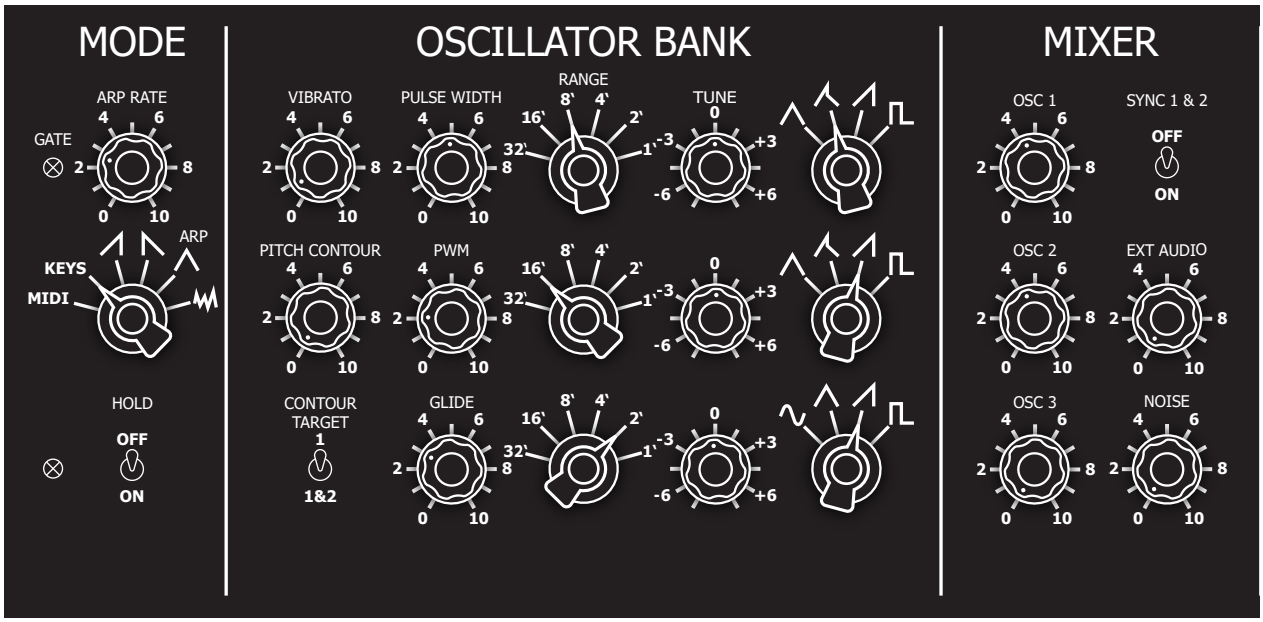
The oscillator bank makes up the largest section of the synthesizer. The main three signal sources, called oscillators, can be set, tuned and modulated here. The middle three rotary knobs, called RANGE, are used to set the octave of each oscillator. By turning the knob a step, the pitch is shifted by an octave at a time. The three tune knob to the right are used to fine tune each oscillator. Rotating the pots will shift an oscillator by approximately ±7 semitones. Further to the right, the three rotary knobs are used to set the output waveform of each oscillator. The top two feature triangle, moog-saw, sawtooth and pulse waveforms. The third oscillator contains a sinusoid instead of the moog-saw.

There are different ways to modulate pitch and shape of the three oscillators. Vibrato will alter the pitch of the oscillators using the LFO signal from the modulation section. The signal can be attenuated using the knob. PULSE WIDTH will vary the duty-cycle of the pulse waveform. PWM, which stands for Pulse-Width-Modulation, will modulate the duty-cycle of the pulse-waveforms using the LFO. Using the CONTOUR TARGET switch and the PITCH CONTOUR pot, the pitch of selected oscillators can be modulated using the filter contour on the right side of the synthesizer. The amount is controlled with the knob. The last feature, called glide or sometimes portamento, will slew the incoming pitch signal from the keyboard. If GLIDE is turned all the way up, the oscillators will take longer to reach a new note.

3.6.3 Mixer

The mixer is used to mix various audio signals before they pass into the filter. The level of the three oscillators can be adjusted using the first three knobs. An external audio-signal can be added using the knob and the input-jack on the back panel. The mixer features a white noise source, which can also be added to the mix.

The SYNC 1&2 switch is used to synchronize oscillator one to two. This means oscillatore one will periodically reset oscillator two, which can create an interesting new timbre. This feature is best used with the PITCH CONTOUR to create nice hard-sync leads.



Picture 3.11

3.6.4 Filter

The Steiner-Parker diode ladder filter is used to filter frequencies from the incoming audio signal. The filter has four states, which can be selected using the rotary knob: Lowpass, Bandpass, Highpass and Allpass. The cutoff frequency, where the filter starts attenuating, can be adjusted using the CUTOFF knob. The filter can also be fed back into itself using the RESONANCE knob. **WARNING**: The Steiner-Parker filter can be very unpredictable, if the resonance is turned up. When entering the red region, the filter can produce sounds which go from squeals and screams to loud high pitched oscillations, which can easily hurt your ears on high volume.

The filter features three different modulation sources, which all alter the cutoff frequency. The COUNTOUR knob can be used to modulate the filter cutoff using the filter-envelope on the second last section. The VELOCITY knob will alter the envelope, therefore I will explain it in the Contour & Amplifier section. The MODULATION knob will alter the cutoff with the LFO signal.

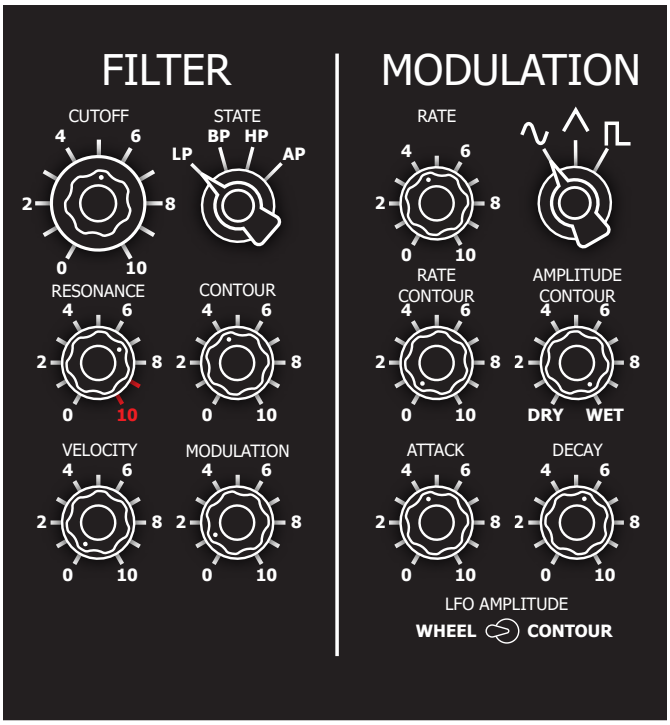
3.6.5 Modulation Section and Wheels

The modulation section is the most complicated one to understand, which is coincidentally also the one I designed myself. In a nutshell, this section consists of an LFO or Low-Frequency-Oscillator, which is modulated using a simple Attack-Decay-Envelope. The LFO is an oscillator at a rate below audible frequency. It is used to modulate different parameters of the synthesizer like pitch or filter-cutoff. With the first knob, the rate can be adjusted. The rotary switch to the right is used to switch the waveform of the oscillator between a sinusoidal, triangle or square waveform.

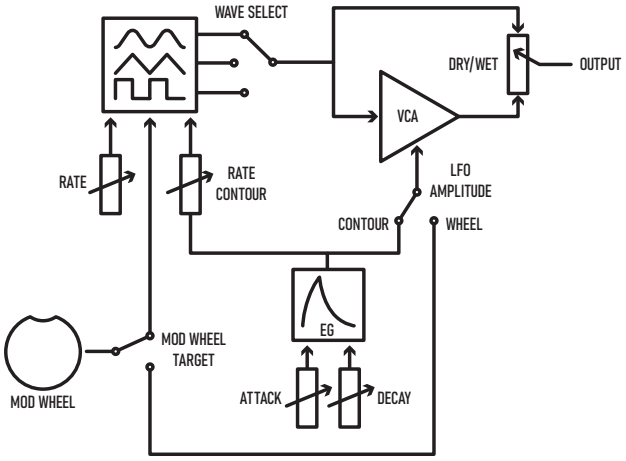
The lowest two knobs control the attack and decay time of the small AD-Envelope, which restarts every time a key is pressed. The first of the middle knobs, called RATE CONTOUR, controls how much the envelope will affect the rate of the LFO. This will make the LFO speed up or slow down when a key is pressed on the board. The second knob is used to adjust how much the envelope will affect the amplitude of the LFO. Unlike the rate, which just adds up the two voltages, this DRY-WET knob will let you adjust between 0% effect on the amplitude and 100% effect on it. Therefore, if the knob is all the way on DRY, the amplitude will always be 100% and not depend on the envelope. But if the knob is all the way on WET, the amplitude of the LFO will directly correspond to the contour.

After realizing the importance of using the modulation-wheel on the left to control the amount of LFO, I added and extra switch on the bottom. This switch can be used to overwrite the signal of the envelope with the one from the modulation wheel. This lets you choose between wheel or envelope.

However, the modulation wheel can also be used to affect the rate of the LFO. The switch on the wheel-panel can be used to switch the modulation-wheel between rate or amplitude of the LFO. The potentiometer next to it also lets you attenuate the amount of modulation wheel before entering the switch. The last pot on the wheel-panel lets you add the modulation-wheel voltage to the filter cutoff. Turning the wheel will therefore raise the filter cutoff.



Picture 3.12



Picture 3.13 To visualize the rather complex signal flow of the modulation section, I made this diagram. It shows the signal flow of all control voltages which interact with the LFO.

3.6.6 Contour, Amplifier & Output

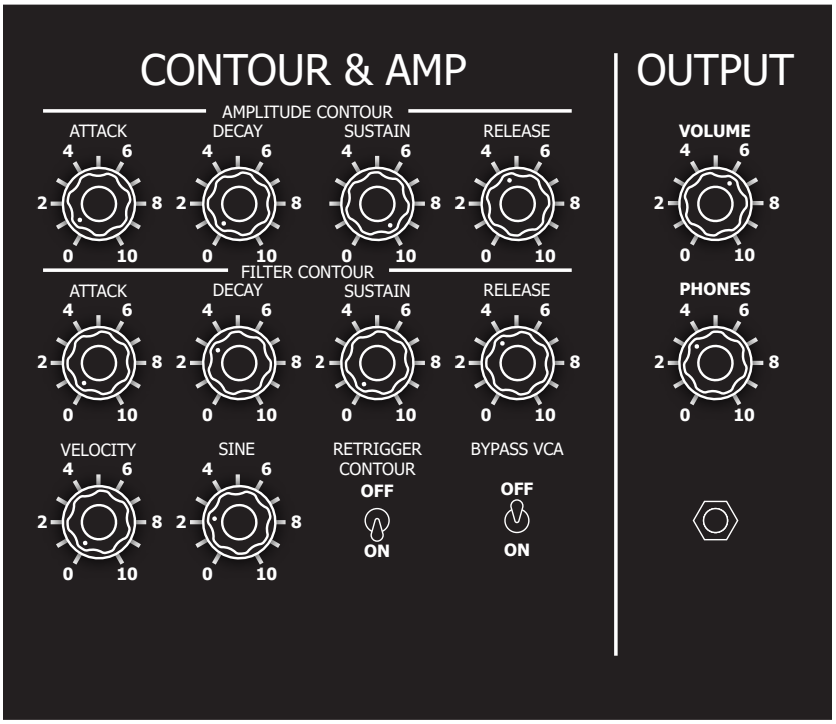
This section contains the two main envelope generators and the voltage controlled amplifier. Both envelopes consist of standard ADSR envelopes. Attack, decay and release time can be adjusted with the appropriate knobs, as well as the sustain level. The upper envelope routs to the VCA and is used to control the change in level of the sound. Therefore, the synthesizer remains silent until a key is pressed, which will trigger the envelope. If the VCA and envelope is not required, the VCA can be completely bypassed using the BYPASS VCA switch on the bottom.

The second envelope has two uses. The first is to control the filter cutoff using the envelope. This is done using the CONTOUR knob in the filter section. The second use is controlling the pitch of oscillators as explained earlier.

The two VELOCITY knobs, one being placed in the filter section and one here, influence the sustain level of their corresponding envelope. If a velocity knob is turned up, the speed of a keystroke will have effect on the amount of sustain on the envelope. The shortest possible velocity time will correspond to a sustain value similar to the one set on the panel. Longer velocity times will reduce the sustain value. The longest possible time will result in zero sustain if the velocity knob is turned all the way up.

The additional SINE knob is used to bypass oscillator 3 past the filter. This signal will always be sinusoidal, independent from the waveform oscillator 3 is currently set to. This SINE features is added to give certain sounds a stronger low end. Especially if the filter is in bandpass or highpass mode, where most of the bass is filtered out.

The output section features a volume knob to adjust the level of the main line output on the back panel. A stereo headphone out is provided with an additional volume knob.



Picture 3.14

4 Conclusion

During the building of my synthesizer, there were some difficulties. Some due to bad planning, not thinking everything through or just small errors which I had to find and remove. The biggest problem was not thinking the signal flow and the design of the LFO completely through. Using the LFO for modulation in my synthesizer is very unusual compared to others. To fix this problem I had to add an additional switch on the front panel to simplify the use of the modulation wheel in combination with the LFO. I believe those complications where due to wanting all possible modulation variations accessible on the front panel using only switches and knobs. For future projects, I will approach different modulation features more like they are done in monophonic synthesizers today, like the Moog Subharmonicon. Those synthesizers replace the inner routings with output jacks on the front panel. This allows the user to connect jacks using small patch cables. Additionally, the instrument can be connected to a modular synthesizer in Eurorack format, making the possibilities endless.

A problem which is hard to avoid while prototyping, is interference. Even if synthesizers usually use very simple electronics not susceptible to noise and interference, I still managed to get noise from the white-noise source to bleed into the audio path even if it was turned off. After a lot of experimenting and searching for the problem, I realized that the mixer used very weak signals to operate. This was due to not buffering the mixer and just passively mixing the signal and passing it on to the filter. To fix this problem, I completely replaced the circuit board which housed the noise source and mixer with a new and improved noise and mixer circuit.

A final thing I would approach differently in the future, is building each circuit on a separate strip board. Even though it makes it easy to take out and debug, I spent around half of my time just cutting and crimping the cables which connected board to board. Also, having tons of loose cables hanging around makes finding interference problems a lot harder. For my next big synthesizer project, I will design everything on a big circuit board. On such a large scale I will definitely save a lot of time and hassle.

5 Acknowledgements

First, I want to thank my supervisor Christian Freiburghaus. Due to his own experience in building synthesizers, he could help me a lot with planning and constructing my own. Next, I want to thank my former art teacher Sacha Christener for teaching me the process of screen printing and letting me use the equipment for it at my old school. Also a special thanks to my class mate Nico Lenz for 3D-printing the modulation and pitch wheels for my synthesizer. A huge thanks to Jonas Biedermann for proofreading my paper and helping me gain ideas. Lastly, I will thank my parents for the occasional financial and emotional support.

6 Appendix

6.1 Table of figures

Picture 2.1 Minimoog Model D, September 2020
https://community.mixedinkey.com/BlogPosts/gadgets-moog-synthesizer

All other pictures, illustrations and schematics were shot, designed or drawn by me. For schematics I used the free program KiCad. For illustrations I used Adobe Illustrator. 3d Models were designed in Blender3d. This paper was written in Adobe InDesign.

6.2 Table of sources

6.2.3 Webpages

Arturia Minibrute, September 2020
https://en.wikipedia.org/wiki/Arturia_MiniBrute

Steiner-Parker Synthacon, September 2020
https://en.wikipedia.org/wiki/Steiner-Parker_Synthacon

Minimoog Model D, September 2020
https://de.wikipedia.org/wiki/Moog_Minimoog

History of Minimoog, September 2020
https://www.youtube.com/watch?v=sLx_x5Fuzp4 History of Minimoog

MIDI, September 2020
https://de.wikipedia.org/wiki/Musical_Instrument_Digital_Interface

Adafruit MCP4728, September 2020
https://learn.adafruit.com/adafruit-mcp4728-i2c-quad-dac/arduino

Syntherjack Power Supply, September 2020
https://syntherjack.net/modular-synth-power-supply/

Steiner-Parker Filter & LFO Schematic, September 2020
https://yusynth.net/index_en.php

VCA & Noise Source Schematics, September 2020
https://www.schmitzbits.de/

Lots of inspiration and DIY knowledge, LOOKMUMNOCOMPUTER
https://www.lookmumnocomputer.com/

6.2.3 Books

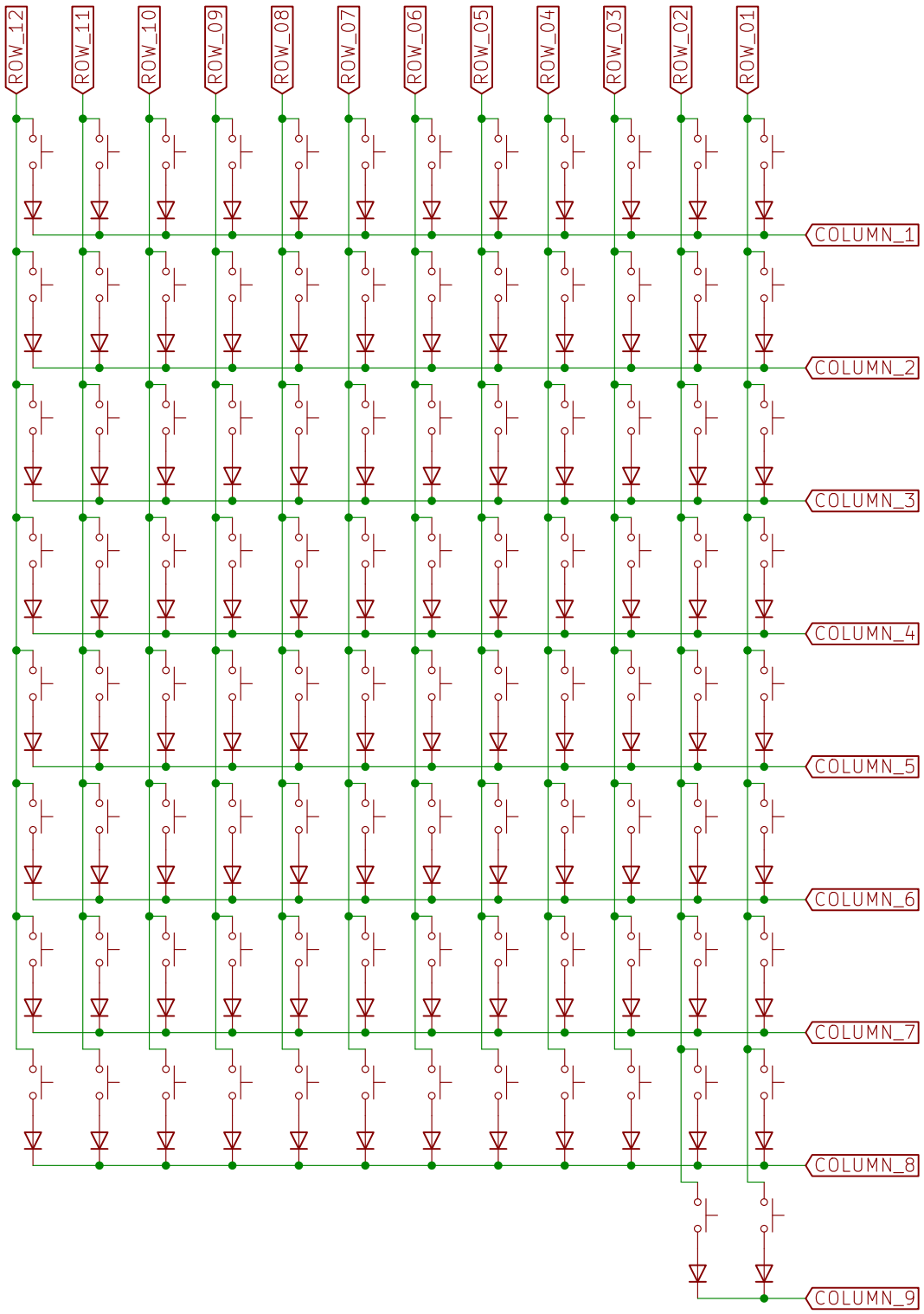
Florian Anwander, SYNTHESIZER, 2001

6.3 Additional Pictures

Picture 6.1 Another angle of the finished synthesizer. The back panel is visible in this picture.

Picture 6.2 This picture shows all electronics. Also, the heatsink on the top left and the stand on the right is visible. The ribbons on both sides stop the upper part from overrotating and destroying the hinge.

6.4 Additional Schematics



Picture 6.3 The keyboard button matrix

6.5 Arduino Code (C++)

```
#include <Wire.h>
#include <Adafruit_MCP4728.h>
Adafruit_MCP4728 mcp;
#include <SPI.h>
#include <MIDI.h>
MIDI_CREATE_DEFAULT_INSTANCE();

void setDac(int key, byte vel);

//////// IO PINS //////
// SHIFT REGISTERS
#define LATCH 10
#define MOSI 11
#define SCLK 13

//MULTIPLEXER
#define A 7
#define B 8
#define C 9

//MULTIPLEXED IN
#define IN1 6
#define IN2 5

#define MODULATION A2
#define PITCHBEND A3
#define RATE A6
#define MODESWITCH A7
#define HOLD 4

#define GATE 2
#define CLKOUT 3
#define CLKIN A1
////////////////////

#define MIDI_NOTE_OFFSET 24

#include „Note.h“
#include „Key.h“
#include „Arpeggiator.h“

int lastPB, lastMod, lastSwitchPos,
      lastPitch;

Key keys[49];

int midiNotesPlayed = 0;

Arpeggiator arp;

void setup()
{
  //Serial.begin(115200);
  mcp.begin();

  MIDI.begin(MIDI_CHANNEL_OMNI);
  MIDI.setHandleNoteOn(midiNoteOn);
```

This is the first line of my program. The Wire.h library is used for the communication with the DAC using I2C. Additionally the DAC needs its own library. The SPI library is used to transfer values to the shift registers very quickly. The MIDI library is imported and initialised.

This line declares the setDac() function. This has to be done in C++ if the function is used inside a different class, which is in this case the arpeggiator class. It must be called before the “Arpeggiator.h” class is imported.

I declare my IO-Pins all in caps-lock names. This makes it easier to change a pin later, if a connection has changed on the pcb.

This note offset must be used to send MIDI notes. The lowest note on my keyboard is a C, which does not correspond with the lowest possible MIDI note. This offset is simply added to the pitch.

My own classes are imported (There where four files originally. I have put them all together into the same text here).

The Arduino has its setup function, which runs once in the beginning. The DAC and MIDI are both started.

```
  MIDI.setHandleNoteOff(midiNoteOff);

  pinMode(MOSI, OUTPUT);
  pinMode(SCLK, OUTPUT);
  pinMode(LATCH, OUTPUT);

  pinMode(A, OUTPUT);
  pinMode(B, OUTPUT);
  pinMode(C, OUTPUT);

  pinMode(HOLD, INPUT_PULLUP);

  pinMode(GATE, OUTPUT);
  pinMode(CLKOUT, OUTPUT);

  SPI.setBitOrder(MSBFIRST);
  SPI.setDataMode(SPI_MODE0);
  SPI.setClockDivider(SPI_CLOCK_DIV8);
  SPI.begin();

  digitalWrite(LATCH, LOW);
  digitalWrite(GATE, LOW);
  digitalWrite(CLKOUT, LOW);

  for (int i = 0; i < 49; i++)
  {
    keys[i] = Key();
  }

  arp = Arpeggiator();
}

void setDac(int key, byte vel)
{
  int pitch = int(key * 1000L / 12L);

  mcp.setChannelValue(
      MCP4728_CHANNEL_B, pitch,
      MCP4728_VREF_INTERNAL,
      MCP4728_GAIN_2X);
  mcp.setChannelValue(
      MCP4728_CHANNEL_A, vel * 32,
      MCP4728_VREF_INTERNAL,
      MCP4728_GAIN_2X);
}

void midiNoteOn(
      byte channel, byte pitch,
      byte velocity)
{
  int p = pitch - MIDI_NOTE_OFFSET;
  if (p >= 0 && p < 49)
  {
    midiNotesPlayed++;
    digitalWrite(GATE, HIGH);
    setDac(p, velocity);
  }
}
```

Those two lines give the MIDI library a function to call, if a MIDI-message is received.

All pin modes are declared.

The INPUT_PULLUP internally pulls up a signal. This makes it easy to add a switch or button, because it only needs an external ground connection.

SPI is setup.

The keys[] array and the arpeggiator are initialized.

This function sets the voltages on the DAC.

The pitch is calculated to millivolt steps. Pitch as voltage is represented as volts per octave.
I copied those lines from adafruit.com.

The next two functions take care of incoming MIDI-messages. The pitch is offset, gate gets updated and the DAC gets set. The integer midiNotesPlayed keeps track on the number of notes pressed in MIDI mode. If This value turns to zero, the gate is set low.

```
void midiNoteOff(byte channel, byte pitch,
      byte velocity)
{
  int p = pitch - MIDI_NOTE_OFFSET;
  if (p >= 0 && p < 49)
  {
    midiNotesPlayed--;

    if (midiNotesPlayed <= 0)
    {
      midiNotesPlayed = 0;
      digitalWrite(GATE, LOW);
    }
  }
}
```

```
class Key
{
public:
  bool pad1, pad2;
  unsigned long initialPress;
  byte velocity;
  bool pressed;

  Key()
  {
    pad1 = pad2 = pressed = false;
  }
};
```

As mentioned, I used an array of this class to store the state of notes on the keyboard. I did not do this initially and changed it later. Storing every note very detailed has a lot of upsides, for example: I could physically represent a mechanical keyboard with high note priority and keyboard gate. Every loop I could check the most upper note and the number of notes played. This also simplified the arpeggiator drastically because I could just copy every pressed note into an array for later use.

```
class Note
{
public:
  int pitch;
  byte velocity;

  Note(){};

  Note(int pitch, byte velocity)
  {
    this->pitch = pitch;
    this->velocity = velocity;
  }
};
```

While coding, I needed to differentiate between a note and a key. I thought of a key as a physical representation of what is happening on the keyboard. Therefore, I used the "Key" class for the last step. For copying notes to the arpeggiator class, I used this "Note" class.

```
byte calcVel(unsigned long dt) //dt in ms
{
  // VELOCITY FIXPOINTS
  // C(   5 ms,   0 )
  // A(  20 ms,  63 )
  // B( 100 ms, 127 )

  int value;

  if (dt > 20)
    value = (4 * (int)dt + 235) / 5;
```

The previously mentioned "calcVel" function takes in the millisecond difference between the two button presses of each key. I used the program Geogebra to sketch out fixpoints. Those I could later use to calculate linear equations for a velocity curve. All of this could be replaced by a simple exponential equation, but I thought on my slow Arduino an exponential function could produce to much lag and delay, which could for example become apparent if the Arpeggiator would start lagging.

```
  else
    value = (21 * (int)dt - 105) / 5;

  value = 127 - value;

  if (value > 127) value = 127;
  else if (value < 0) value = 0;

  return value;
}

void detectKeys()
{
  for (int j = 0; j < 9; j++)
  {
    digitalWrite(A, (j >> 0) & 1);
    digitalWrite(B, (j >> 1) & 1);
    digitalWrite(C, (j >> 2) & 1);

    bool extrawurst = (j == 8);
    int loopLength = extrawurst ? 2 : 12;

    for (int i = 0; i < loopLength; i++)
    {
      byte sr1 = 1 << i;
      byte sr2 = i>=8 ? (1 << (i-8)) : 0;

      SPI.transfer(sr2);
      SPI.transfer(sr1);
      digitalWrite(LATCH, HIGH);
      digitalWrite(LATCH, LOW);

      delayMicroseconds(2);

      bool now;

      if (extrawurst)
      {
        now = digitalRead(IN2);
      }
      else
      {
        now = digitalRead(IN1);
      }

      int key = j * 6 + i / 2;

      bool last;
      if (i % 2 == 0)
      {
        last = keys[key].pad1;
        keys[key].pad1 = now;
      }
      else
      {
        last = keys[key].pad2;
        keys[key].pad2 = now;
      }
```

This function reads the keyboard. It starts out by looping over the 9 output lines of the keyboard matrix.

Then the multiplexers are set with the binary digits of index j.

The boolean “extrawurst” is used to show if the 9. row of the outputs is used, because then the additional input must be used.

Another for-loop is started with either a length of 12 or 2 depending on the “extrawurst”.

The bytes “sr1” and “sr2” are then sent out by the shift registers. They represent all zeros except for a single one. This is achieved by left-shifting 1 in binary by the index i.

Then the shift registers output is updated by triggering the RCLK pin.

To detect if a key is newly pressed or released the code keeps track of two variables: “now” and “last”. Now is set to the current output of the keyboard matrix.

The key number is calculated using i and j.

“last” represents the state of the button last time it was read. It is stored in an array of “Keys”.

```
      if (now && !last)
      {
        if (i % 2 == 0)
        {
          keys[key].initialPress =
                        millis();
        }
        else
        {
          keys[key].pressed = true;

          unsigned long t2 = millis();
          unsigned long t1 = keys[key].
                        initialPress;

          byte vel;
          if (t2 > t1)
          {
            vel = calcVel(t2 - t1);
          }

          keys[key].velocity = vel;

          if (!arp.enabled)
          {
            MIDI.sendNoteOn(MIDI_NOTE_
                  OFFSET + key, vel, 1);
          }
        }
      }

      else if (!now && last)
      {
        if (i % 2 == 1)
        {
          keys[key].pressed = false;

          if (!arp.enabled)
          {
            MIDI.sendNoteOff(MIDI_NOTE_
                  OFFSET + key, 0, 1);
          }
        }
      }
    }
  }
}


int switchPos()
{
  int sw = analogRead(MODESWITCH);

  if (sw < 102) //MIDI
    return 2;

  else if (sw > 102 && sw < 307) //KEYBOARD
    return 1;
```

If a button is newly pressed, now should be true but last should not.

The program needs to differentiate between the first and second button of every key. This is done by calculating if "i" is even or odd. If the first button is pressed, the system time is stored using "millis()".

If the second button is pressed, the key is now counted as pressed. System time is again used to calculate the time difference between the two buttons. The function "calcVel()" returns a byte with velocity values from 0 to 127.

If the synthesizer is not in arpeggiator-mode MIDI messages are sent to indicate if keys are pressed or released.

This part of the code is similar to the last part and is executed, if a key is released.

The switchPos function reads the state of the rotaty switch in the MODE section on the front panel. Every step of the switch is a voltage between 0v and 5v, which is in code represented by a 10-bit number between 0 and 1023.

```
  else if (sw > 307) //ARP
  {
    if (sw < 512)
      arp.mode = 0;  // up
    else if (sw > 512 && sw < 716)
      arp.mode = 1;  // down
    else if (sw > 716 && sw < 912)
      arp.mode = 2;  // up & down
    else if (sw > 912)
      arp.mode = 3;  // random

    return 0;
  }
}
```

If the arpeggiator is selected, additionally to the return value, there is also the arp.mode. This controls the order of notes on the arpeggiator.

```
void loop()
{
  int swchPos = switchPos();

  if (swchPos == 0 &&
      swchPos != lastSwitchPos)
  {
    arp.enabled = true;
  }
  else if (lastSwitchPos == 0 &&
      swchPos != lastSwitchPos)
  {
    arp.enabled = false;
    MIDI.sendNoteOff(MIDI_NOTE_OFFSET
            + arp.lastPitch, 0, 1);
  }
  else if (lastSwitchPos == 2
      && swchPos != lastSwitchPos)
  {
    //midi deselected
    midiNotesPlayed = 0;
    digitalWrite(GATE, LOW);
  }
  else if (lastSwitchPos == 1
      && swchPos != lastSwitchPos)
  {
    for (int i = 0; i < 49; i++)
    {
      if (keys[i].pressed)
      {
        MIDI.sendNoteOff(
            MIDI_NOTE_OFFSET + i, 0, 1);
      }
    }
    digitalWrite(GATE, LOW);
  }
  lastSwitchPos = swchPos;

  if (swchPos == 2)
  {
    //MIDI IN

    MIDI.read();
```

This is the main loop. The Arduino repeatitly calls this function after the setup.

Firstly, the switch position on the main panel is read. The function switchPos() returns an integer. 0 represents the arpeggiator, 1 is normal play and 2 stands for MIDI in.

Those first few if-statements detect if the switch position changed since the last loop. This is necessary, for example, when a note is pressed on the keyboard and the mode is switched to MIDI. The program must remove the note manually and reset the gate voltage. All those instructions are for purposes like this.

From here the main executions begin.

```
  if (!digitalRead(HOLD)) //safety reset
  {
    midiNotesPlayed = 0;
    digitalWrite(GATE, LOW);
  }
}
else
{
  detectKeys();

  int notesPlayed = 0;
  Note notes[16];
  for (int i = 0; i < 49; i++)
  {
    if (keys[i].pressed)
    {
      notes[notesPlayed] =
          Note(i, keys[i].velocity);
      notesPlayed++;
      if (notesPlayed >= 16)
      {
        break;
      }
    }
  }

  if (swchPos == 0)
  {
    //Arpeggiator

    arp.arpeggiate(notes, notesPlayed);
  }
  else
  {
    //Keyboard play

    if (notesPlayed > 0)
    {
      digitalWrite(GATE, HIGH);

      int pitch = notes[
          notesPlayed - 1].pitch;
      byte vel = notes[
          notesPlayed - 1].velocity;

      setDac(pitch, vel);
    }
    else
    {
      digitalWrite(GATE, LOW);
    }
  }

  //PITCHBEND
  int pbRead = analogRead(PITCHBEND);
  if (abs(lastPB - pbRead) > 1)
  {
    lastPB = pbRead;
    int value = (pbRead - 512) * 16;
```

If MIDI mode is selected, MIDI is read using the MIDI library. If messages are received, the MIDI library triggers midiNoteOn and midiNoteOff functions which we will see later. Also, the gate voltage is set depending on the notes pressed.

If the arpeggiator or the normal play is selected, some function are carried out.

The detectKeys() function from earlier updates the keys[] array.

The first 16 notes in the keys[] array are separated out and counted.

If the arpeggiator is selected, the arp.arpeggiate() function is called and the played notes are passed in.

Otherwise, if normal play is selected, gate is updated and the DAC is set using pitch and velocity values. The value of the most upper note in the notes[] array is used, making it high-note priority.

Lastly, pitchbend and modulation voltages are read. If the difference since the last value is greater than 1, a midi messages is sent. There is no need to send those values to the DAC, because the voltages on the wheels are directly sent into the VCOs etc. using analog circuity.

```
      MIDI.sendPitchBend(value, 1);
    }

    //MODULATION
    int modRead = analogRead(MODULATION);
    if (abs(lastMod - modRead) > 1)
    {
      lastMod = modRead;
      MIDI.sendControlChange(1, modRead /
                             8, 1);
    }
  }
}

class Arpeggiator
{
public:
  bool enabled = false;
  int mode = 0;
  Note heldNotes[16];
  int heldCount;
  int lastCount = 0;
  Note maxNotes[16];
  int maxCount = 0;
  unsigned long lastHalfStep = 0;
  int index = 0;
  bool state = false;
  unsigned long halfPeriod = 100;
  int dir = 1;
  bool lastHold = false;
  int lastPitch;

  Arpeggiator() {}
  void arpeggiate(Note notes[], int count);
  int calcArpPeriod();
};
```

All the arpeggiator logic is stored in this class. This made the code less messy.

There are a lot of different forms of notes and counts, mainly used for the hold-function. A note[] array stores notes and the corresponding count stores the number of notes in the array.

```
void Arpeggiator::arpeggiate(Note
newNotes[16], int newCount)
{
  Note *notes;
  int count;

  bool hold = !digitalRead(HOLD); //
electrically pulled up => false means hold
is active

  if (!lastHold && hold)
  {
    heldCount = 0;
  }
  lastHold = hold;

  if (hold)
  {
    if (newCount > lastCount)
    {
      // COPY NOTES
      maxCount = newCount;
```

The main function of the arpeggiator is called arpeggiate() and takes in the notes which are currently pressed on the keyboard. Those new notes have the prefix new.

The notes and the count without any prefix represent an empty array, which at the end of the function will be filled using the current notes that will be played by the arpeggiator.

The state of the hold switch is detected.

If hold is selected, the arpeggiator needs to detect if any new notes are added to the current notes. If the new count is larger than the last count the amount of pressed notes has increased and the maxNotes[] array is overwritten.

```
    for (int i = 0; i < newCount; i++)
    {
      maxNotes[i] = Note(newNotes[i].
          pitch, newNotes[i].velocity);
    }
  }

  if (newCount == 0 && lastCount != 0)
  {
    // COPY NOTES
    heldCount = maxCount;
    for (int i = 0; i < maxCount; i++)
    {
      heldNotes[i] = Note(maxNotes[i].
          pitch, maxNotes[i].velocity);
    }
  }

  notes = heldNotes;
  count = heldCount;
}
else
{
  notes = newNotes;
  count = newCount;
}

lastCount = newCount;

halfPeriod = calcArpPeriod();
if(lastHalfStep + halfPeriod < millis())
{
  if (count > 0)
  {
    if (state)
    {
      digitalWrite(GATE, HIGH);
      state = !state;

      if (mode == 0)   // up
      {
        index++;
      }
      else if (mode == 1) // down
      {
        index--;
      }
      else if (mode == 2) // up & down
      {
        index += dir;
        if (index==count-1) dir = -1;
        else if (index==0) dir = 1;
      }
      else  // random
      {
        index = random(count);
      }

      if (index < 0) index += count;
```

If the amount of pressed notes returns back to zero, those notes will be held, if no further note is pressed. The heldNotes[] array keeps track of the held notes.

The held notes are copied to the final notes.

If hold is not used, the new notes are copied to the final notes, as easy as that.

The main oscillation of the arpeggiator is done using this if-statement. Because the arpeggiator must update twice every note it plays to update the gate, I am using the variable halfPeriod. This time is calculated using the function calcArpPeriod().

The variable state indicates if the arpeggiator is in up or down position. The gate gets updated like expected.

Mode 0 lets the arpeggiator step upwards through the notes. This can be done by increasing the index.

Mode 1 works similar.

Mode 2 increases the index by the integer dir which is either a 1 or a -1. If the arpeggiator reaches a corner, the direction is switched.

Mode 3 is the simplest of all. It picks a random integer as the index.

```
        index %= count;

        int pitch = notes[index].pitch;
        byte vel = notes[index].velocity;
        setDac(pitch, vel);
        MIDI.sendNoteOn(MIDI_NOTE_OFFSET
            + pitch, vel, 1);
        lastPitch = pitch;
      }
      else
      {
        state = !state;
        digitalWrite(GATE, LOW);
        MIDI.sendNoteOff(MIDI_NOTE_OFFSET
            + lastPitch, 0, 1);
      }
    }
    else
    {
      digitalWrite(GATE, LOW);
      index = -1;
      MIDI.sendNoteOff(MIDI_NOTE_OFFSET
            + lastPitch, 0, 1);
    }

    lastHalfStep = millis();
  }
}

int Arpeggiator::calcArpPeriod()
{
  long rate = analogRead(RATE);

  // arp rate fixpoints (ms)
  // ( 0 / 2000)
  // ( 500 / 200 )
  // ( 1024 / 10 )

  if (rate > 500)
    return (int)
      ((-95L * rate + 99900L) / 262L );
  else
    return (int)
      ((-18L * rate + 10000L) / 5L );
}
```

If the arpeggiator reaches the beginning or the end of the array, the count is added or substracted.

The DAC is set and MIDI-note messages are sent.

If the arpeggiator is in the down state, the gate is set low and the last sent MIDI note is turned off again.

This function reads the potentiometer on the front panel. Similarly to the velocity-function, this fuction approximates an exponential function with two linear functions.

I chose the fixpoints so, that the longest period is 4 seconds and the shortest 20 milliseconds.