Re-RoutaSynth: Hybrid Analog/Digital Modular Synthesizer

By

Adam Willis

Connor Jones

Ishaan Datta

Final Report for ECE 445, Senior Design, Fall 2020

TA: Dean Biskup

09 December 2020

Project No. 32

Abstract

In this report, we discuss the design of a hybrid analog/digital modular synthesizer. This synthesizer allows the user to control the parameters of up to 8 audio or control signals using a computer interface implemented in Max/MSP. This interface also provides a means for routing these signals in a highly customizable fashion through 4 analog voltage-controlled filter/amplifier boards. The virtualization of signal routing allows for greater ease of use and reproducibility for the consumer. Although the product is not fully functional, individual modules have been completed and will be presented.

Contents

1. Introduction

1.1 Problem and Solution Overview

The audio market was once dominated by the use of bulky, analog modular synthesizers. Their unique audio quality and a lack of an alternative allowed them to exist as the industry standard, with music producers and sound researchers struggling to garner the necessary funding to acquire such expensive instruments. However, over the past few decades, technological advancements have allowed digital synthesizers to generate a greater public response. Due to their easy upkeep and relatively low cost, these synthesizers became heavily sought after by the industry, resulting in the plethora of digital synthesizers available in digital audio workstations (DAWs) such as Logic Pro X, FL Studio and Pro Tools.

Despite the accessibility of digital synthesizers, the public's demand for their analog counterparts has re-emerged over the past two decades. Most consumers, musicians especially, find the sound quality of analog devices to be superior to digitally filtered sounds[1]. However, these synthesizers aren't always digitally-compatible, are expensive and require patching cables which can easily clutter as in figure 1.

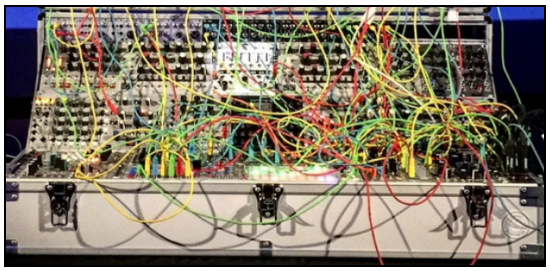


Figure 1. Analog Modular Synthesizer with cable-based patching.

Our solution offers a hybrid synthesizer, primarily for use by musicians and audio researchers, that marries the flexibility of digital signal processing with the superior audio quality of analog hardware, like voltage-controlled filters and amplifiers[14]. This analog/digital synthesizer allows for both digital inputs, MIDI inputs and analog outputs to be routed through a number of channels and filters with greater ease and reproducibility than provided by cable-based patching.

1.2 Project Summary

Ultimately, we were unable to get the synthesizer fully functioning. Specific modules have been tested and finalized, but the main source of error has arisen from our inability to trigger the proper System Exclusive (SysEx) and serial messages to be sent to our system from the user interface. Nevertheless, our report will cover the advancements that were made; primarily with regard to the rigging of our analog components and our micropython support design.

Once fully implemented, our synthesizer would allow the user to craft and output audio waves through an analog routing matrix that eliminates cable-based patching and is more portable than its analog predecessors. The synthesizer's price would fall somewhere in the range of $500-700, depending on manufacturing methods, providing a highly flexible and cost-effective device with more options for sound design and synthesis than the more expensive options currently available on the market.

2 Power Supply

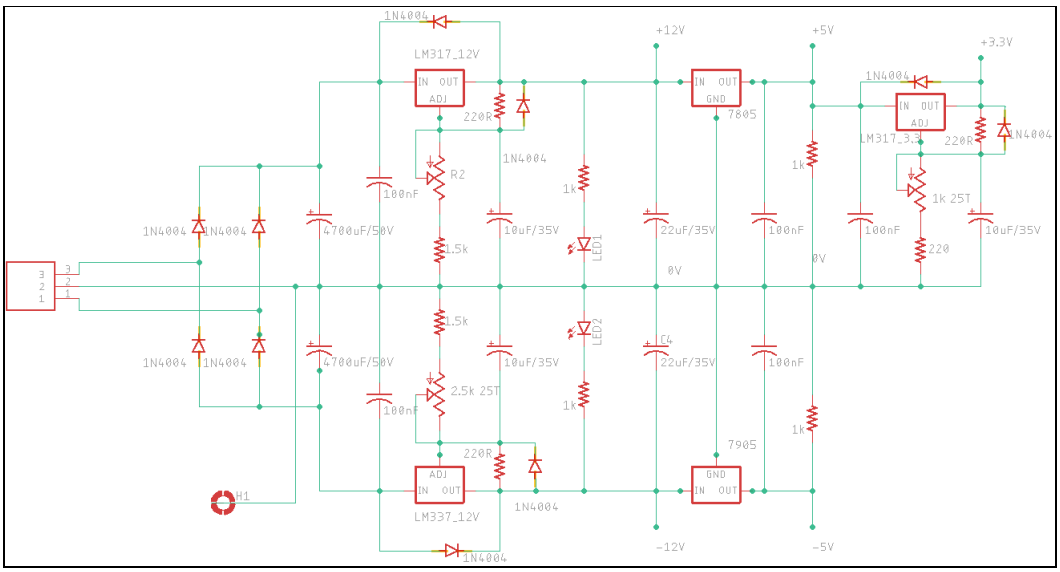


Figure 2. Core Power Supply Schematic

2.1 Design Procedure and Details

The power supply was based around Ken Stone's CGS66 Power Supply which is intended as a linear power supply for DIY modular synthesizers, and as such provides the proper rectification and filtering to ensure that a consistent voltage is read by all components[1]. This is critical, as the human ear is very sensitive to pitch and any fluctuation in power delivery would affect aspects of the filters (such as the cutoff frequency) and could lead to intermittent power for the PyBoard. Not pictured in the schematic above is the Hammond 167K30 transformer, which accepts wall voltage from the grounded power inlet and provides a 30V, 1.5A center-tapped AC output. The center-tap is connected to the inlet labelled "2" on the port to the far left of figure 2, with the other two wires attached to "1" and "3." A bridge rectifier converts this to positive and negative DC which is filtered first through two 4700 µF capacitors and subsequently sent to an LM317 positive voltage regulator and an LM337 negative voltage regulator, respectively. These are connected as per the application notes in the datasheets with the output voltage set, using the LM317 as an example, by adjusting the value of R2. The variable resistors are all of the high-precision 25-turn variety, and the output voltage can be found, as per the datasheet[2],[3], through the formula

$$V_O \approx 1.25\ V\left(1 + \frac{R2 + 1.5\ k\Omega}{220\ \Omega}\right) \qquad \textbf{(2.1)}$$

The +/- 12 V outputs serve as a power supply for the various amplifiers at the inputs and outputs of the filters, the outputs of the DACs, and the overall output mixer. While most signal outputs will never exceed a maximum of approximately 3.3 V (the reference voltage for our DAC), powering the amplifiers through a 12 V bipolar supply ensures that input voltages to amplifiers never come too close to their power-rail values and are thus never distorted by the amplifiers.

The bipolar 12V output is sent to LM7905 and LM7805 voltage regulators, again implemented as per their respective datasheets[4],[5] , which provide both the analog power supply for the NJM2069 filter chips and the digital supplies for the ADG2128 switchpoint matrices and the MAX537 DAC chips. As such, further rectification is provided at the power inputs for each board. Ferrite beads in series with the power rails prevent high-frequency noise, due to digital components, from appearing at the power-supply inputs for the NJM2069. A 10 µF capacitor between each power supply rail and ground provides the same protection on the low-frequency end of the spectrum. All applicable integrated circuits are also equipped with 100nF metal film decoupling capacitors at their power supply inputs, as is typically recommended for noise suppression. These can be seen in the full schematics provided in Appendix B.

The 3.3 V supply features most of the same filtration and regulation as is used in the 12V regulator circuit, but uses a highly regulated supply as its input. As this is used as a reference voltage for the digital-to-analog converters, extreme precision is ideal and was achieved; the output was accurate to the thousandth of a volt, varying slightly between 3.299 V and 3.3 V.

Given that extremely linear power is critical for audio applications, it was determined that a linear power supply similar to the one above was critical to our needs. Alternatives to the Hammond transformer were proposed, and included using two 15VDC wall adapters. However, for consumer applications, a power supply which takes up two outlets is somewhat unheard-of, and thus the above configuration was adopted.

2.2 Verification

As verification for the power supply, all four VCA/VCF boards were attached with the same signal at all four signal inputs (supplied through an FG-7002C sweep-function generator). The output of each power rail was measured over the course of 5 minutes, and any variation greater than 5% of its total output would be deemed a failure. The test was successful, with no power supply rail deviating by more than 0.5%.

3 User Interface

3.1 Design Procedure and Details

3.1.1 Design Procedure

The user interface is the primary way in which the user interacts with the synthesizer. It offers a window with clickable options and menus where the output of each generator and routing matrix can be selected, controlled and modified. Max/MSP was ultimately chosen as the tool for constructing this interface as it offers simplified tools to perform tasks for data processing and parsing, as well as an object which allows for serial messages to be sent through the computer's USB connection. These messages are then processed by the isolated USB-to-UART converter and subsequently sent to a UART input on the PyBoard.

The initial intention was for an independent program to be designed from scratch using a GUI library in a higher-level language such as Python, Java or C++, but the abstractions already available in Max streamlined the process considerably. We wanted to allow the user control of the signal routing and generator information through a simplified visual mechanism, and Max/MSP ultimately offered this compatibility.

All options selected by the user in the Generator window are encoded numerically as a string of binary data. The MIDI channel is always set to zero. This value is followed by is the generator number, and a 0 or 1 to indicate whether it is currently on or off. Next is the waveform type, encoded as a 3-bit binary value, followed by the input source for the amplitude and the frequency, each encoded as 2-bit binary values. The options for the formula-input modes for the amplitude and the frequency are given next, with the reading of these values skipped over by the program running on the PyBoard if the user has not opted to set either of these values through formula. Finally, the last segment of the message is a list of points that the user has selected in the amplitude window, encoding the overall volume of the selected waveform.

3.1.2 Design Details

In figure 3 below, the interface through which the user interacts with the generator is shown, as well as a small window that the user can double-click to access the interface for interacting with the routing matrices.

Figure 3. Generator Interface, User-End

Dropdown menus allow the user to choose a generator number, waveform type, frequency source and amplitude source. Selecting “Constant Value” for either the frequency or the amplitude source causes the display to change to feature a single window in which the user can enter a floating-point value, while selecting “Formula” replaces that single window with another allowing the user to select between the various formula options. Buttons allow the user to clear entered points from the amplitude envelope window, while clicking anywhere in the window will create a new point that will change the overall way in which the volume or amplitude of the output changes over time. The “Total Envelope Time” option selects the maximum value of the x-axis for the amplitude envelope, with the x-values of all other points being adjusted accordingly in proportion to this maximum value. This allows for the total duration of a sound to be changed without the user needing to adjust each point individually.

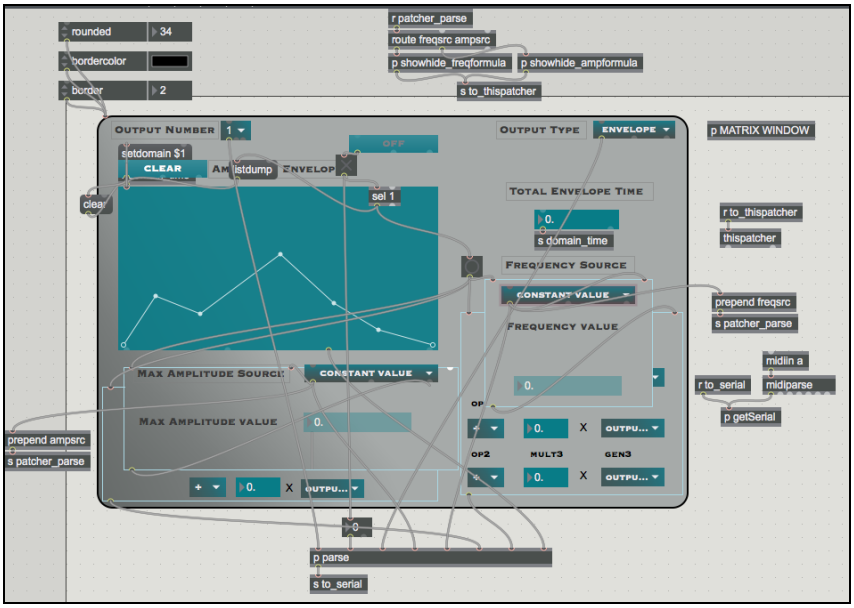


Figure 4. Generator Interface, Control-End showing Hidden Objects

In figure 4 above, the same window is shown; however, the objects parsing the user input data, normally hidden to the user, are visible. Some values, such as the output number, output type, and on/off status, are sent directly to an object called “p parse,” which is a subpatcher designed to prepare the message for serial output. The “Constant Value” or “Formula Input” inputs for the frequency and amplitude sources, will appear based on what source the user sets for both attributes.

The inner workings of the “p parse” object and the “p getSerial” object are shown in figure 5. The parse object simply collects the values the user has selected, modifies them and only outputs the values when the “ON/OFF” button has been toggled to “ON”. It achieves this by using “i” and “zl.reg” objects to store integers or lists, respectively, until a trigger is received from a “t b i” or a “t b l” object. The “zl.group” object collects these and only sends the message when the last piece of data has been received. The “getSerial” object is simply a Max serial object with some debugging features available.

While messages were successfully sent from the serial object to the USB-to-UART converter and were received and detected at the UART input of the PyBoard, difficulties with trivialities such as, e.g. the correct way to process floating-point values to conform to the encoding scheme shown above ultimately prevented us from operating the synthesizer as intended through the Max interface. However, given the successful receipt of messages by the PyBoard from Max, these issues could undoubtedly be resolved given further time.

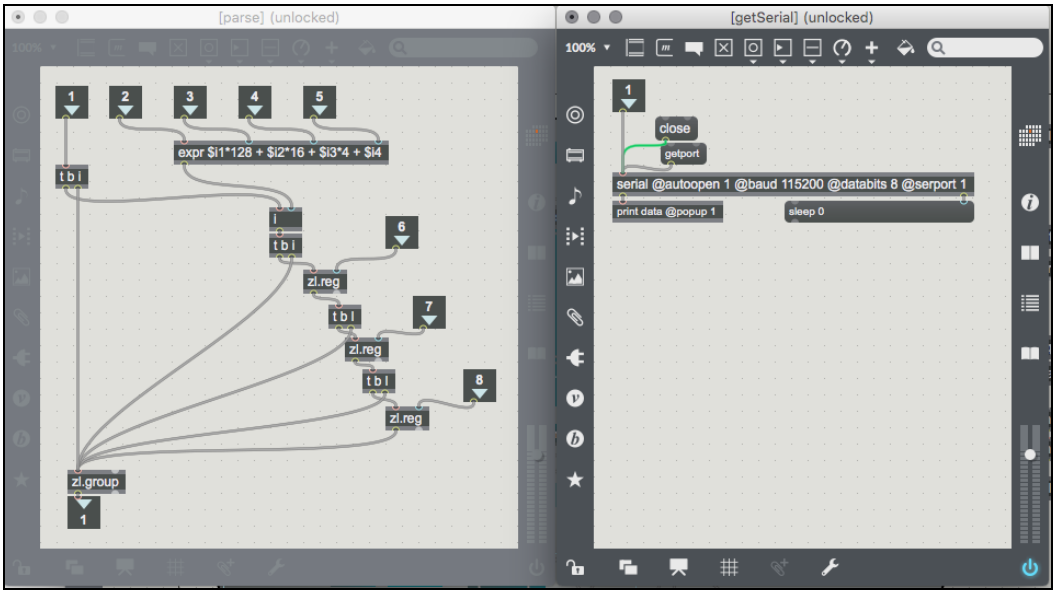


Figure 5. The ‘p parse’ and ‘p getSerial’ Objects

Similar principles are at work in the matrix window which controls the routing, the interface for which is shown in Figure 6 below.

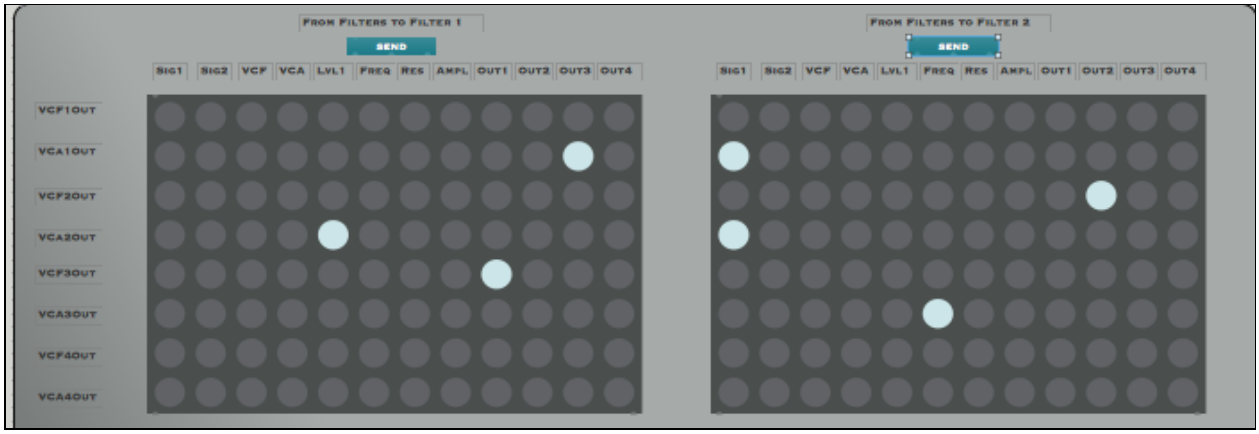


Figure 6. Matrices from the Matrix Control Window, Routes from FIlter Outputs to Other FIlter Inputs)

4 Analog/Digital Board

The analog/digital board serves as the bridge between the Max-based user interface and the analog voltages expected at the inputs to the VCA/Filter boards. It also houses the PyBoard and the mixed-signal integrated circuits; such as the MAX537 digital-to-analog converters and the ADG2128 switchpoint matrices. The schematic for this board can be found in figure 8 of appendix B.

The primary difficulties with our project were caused by a series of issues involving this board: First, it was found that if the PyBoard is connected through USB to the computer and the power supply simultaneously, it will receive power through the USB; the first draft of this board had the PyBoard sharing a ground with all other components, and provided neither a provision for an alternative power source nor a way to receive serial data from the computer aside from the dedicated USB port. This would have resulted in serious ground issues which would likely have completely compromised any noise-proofing at the power supply or by the ferrite beads and decoupling capacitors on the boards themselves. It was determined that an isolated USB-to-UART converter could be used to direct serial input to the UART ports of the PyBoard while simultaneously isolating the power supply and its ground from that of the USB port, and so this scheme was adopted and the board was redesigned. Second, the revised board, shown in the schematics below and ordered from Elecrow, never arrived. Last minute compensatory measures were taken, such as using breakout-boards for the switchpoint matrices, but failures and mistakes in component sourcing resulted in delays that hindered our progress.

Alternatives to the use of the ADG2128 switchpoint matrices included the use of a very large number of multiplexers, but the complexity of controlling these as well as the simplicity of the I^2C protocol utilized by the ADG2128 chips led to their integration into the project.

4.1 Design Procedure and Details

4.1.1 Digital-to-Analog Converters

The MAX537 DACs in figure 9 of appendix B take the SPI outputs from the PyBoard and use them, alongside a 3.3 V reference voltage, to produce a continuous analog voltage at each of the outputs with a magnitude of 3.3 Vpp[6]. This is far higher than the intended 0.5 Vpp expected at the inputs of the filter/VCA boards, so four AD8672 chips are used as inverting attenuators to bring the output to;

$$\frac{15\ k\Omega}{100\ k\Omega} * 3.3\ V\,pp\ =\ 0.495\ V\,pp \tag{4.1}$$

The intended DAC was changed numerous times during the course of the design, beginning with the 24-bit 192kHz ADAU1962A DAC. Unfortunately, this DAC operates through I^2S communication, which utilizes both the SPI and the I^2C ports to output audio signals. In theory, I^2S is fully achievable using an STM32F767 chip as featured on the PyBoard; however, the firmware implementation of the Micropython language on the PyBoard has yet to meet the full potential of its main microprocessor. It was subsequently discovered that I^2S has yet to be implemented fully on any PyBoard, which has been an open issue in the Micropython community since around 2016.

The MAX537 DACs featured in the schematics were ultimately chosen, as a very-late-stage design switch, due to their general simplicity and ease of use. Only 8 of the intended 12 generators are

featured in the schematics, corresponding to the two available (functional) SPI ports on the PyBoard and the four available inputs and outputs on each board. These chips can theoretically be daisy-chained, such that multiple chips run on the same SPI port, but we thought it best to limit ourselves as the deadline was fast approaching.

4.1.2 Routing Matrices

An excerpt of the ADG2128 switchpoint matrix array is shown in figure 10 of appendix B; The full array can be seen in the schematics, but there are four total in the left column and four total in the right, with each group of four attached to a separate I^2C input port. The 2.2 kΩ I^2C pullup resistors (four total) are only present on the topmost ADG2128 in each column, while a different number or configuration of resistors is attached to ports A2, A1 and A0, allowing their I^2C addresses to be hardcoded. Each ADG2128 in the column on the left receives eight inputs (Y0 through Y7) from the outputs of the four AD8672 operational amplifiers from the previous section. The outputs of this group of four matrices are sent to the four VCA/VCF filter boards via the ports labelled TO_FILTER0 through TO_FILTER3. Meanwhile, each ADG2128 in the column on the right accepts the outputs of the VCFs and VCAs as inputs (two outputs per board and four boards for a total of eight inputs, applied to outputs Y0 through Y7). Each chip then connects these outputs to the inputs of one of the four filter boards through the ports labelled TO_FILTER4 through TO_FILTER7 (connected to X0 through X7), and allows for four signals to be sent to the final four output jacks on the front panel, through the ports labelled TO_OUTPUT1 through TO_OUTPUT4 (connected to X8 through X11).

While multiplexers could have been substituted for the switchpoint arrays, the ease of use of the ADG2128 chips through I^2C, as well as the close correspondence between the code used to turn ADG connections on and off and the output of Max/MSP 'matrixctrl' objects, led to this implementation.

4.1.3 PyBoard Interface and Output Ports

We wanted the analog/digital board to be as flexible as possible; at the time we weren't certain if all eight ADG2128 switchpoint matrices would be able to be run on a single I^2C port. This, among other software uncertainties, led to the use of a somewhat disconnected format, wherein each subsystem was furnished with input and output headers to allow the I^2C channel to be switched manually should that later prove desirable.

As such, the system in figure 11 of appendix B was devised. WBUS41-80 and WBUS1-40 (the leftmost 40-input bus) are Hirose DF40-series female mezzanine connectors which are compatible with the PyBoard's male mezzanine outputs[7], located on the underside of the board. These were placed 12.7mm apart (as per the pyBoard D-series documentation[8]) and are designed to house the board itself. A subset of the GPIO pins as well as the pins corresponding to the SPI and I^2C ports were sent to 0.1" male headers. Two further male headers, labeled J1 and J2 in figure 11 of appendix B, are intended to provide stabilization for the mezzanine connectors, which can be quite fragile if too few of the pins are connected to stable, mounted objects on the parent board.

4.2 Verification

It was important for our analog/digital board to be able to sustain output from all eight digital inputs at 4186 Hz (the highest note on a piano) for 20 seconds without audible distortion. We also intended to test to ensure the board would be able to pass eight simultaneous signals from the DAC to the switchpoint matrices, detect these signals at the inputs of the VCA/VCF filter board boards, and toggle these signals on and off once. These verifications were not performed as the board could not ultimately be constructed. However, the switchpoint matrices were tested to a certain extent (through the use of a breakout board) and were able to route and toggle on/off one sine wave at 4186 Hz, generated through a signal generator in the lab, from each input to each output. This was tested not through the Max/MSP interface but through the use of the PyBoard REPL prompt.

5 Micropython Support

5.1 Design Procedure and Details

In designing our micropython code, we had to first determine how we would handle user input from Max/MSP, as well as from MIDI. Each signal would have to allow the user to set the three main attributes of our audio signals; amplitude, frequency, and output number. Other audio elements such as attack, sustain, and delay time have been omitted for purposes of focusing our system's implementation. In addition, by utilizing amplitude and frequency as our primary attributes, we are better able to align our system for MIDI integration.

5.1.1 Max/MSP | Generators

Our Max/MSP Interface allows for the user to select from varied waveform parameters as per table 2 in Appendix D. The majority of our micropython code in this section is catered towards the different methods of setting the amplitude and frequency. For both attributes, they can be directly updated as a constant value or by inputting values for a formula through Max/MSP. Equation (5.1) depicts how the user input is handled when the formula option has been selected. The integer values are provided by the user for multipliers 1 through 3, and for the indices of generators X, Y, and Z. Operation (op) 1 and op 2 are selected via a drop-down arrow and trigger flags that properly select the appropriate equation to run.

$$\textbf{Frequency} = \left(\boxed{\text{Mult. 1}} \times \boxed{\text{Gen. X}}\right) \boxed{\text{Op 1}} \left(\boxed{\text{Mult. 2}} \times \boxed{\text{Gen. Y}}\right) \boxed{\text{Op 2}} \left(\boxed{\text{Mult. 3}} \times \boxed{\text{Gen. Z}}\right) \quad \textbf{(5.1)}$$

***Note: 0 Hz < **Frequency** < 20 kHz*

We also set upper and lower bounds for our frequency and amplitude values to ensure they are kept within a specific region to avoid system error. Frequency is set to range from 0 Hz to 20 kHz, and anything outside the bounds is corrected accordingly. This precautionary measure was taken to ensure that no frequency crosses the bound set by the Nyquist Limit. Because our system has a sample rate of 44100 Hz, the Nyquist limit is set at 22050 Hz. In order to avoid any potential alias-ing we decided to cap the system at 20 kHz. Given that humans can detect frequencies ranging from 20 Hz to 20 kHz (with the adult range dissipating around 15-17 kHz), cutting off the system at 20 kHz shouldn't overtly affect the user's playing and listening experience.

For amplitude we utilize a similar protocol in that our amplitude bounds are marked at values of 0 and 127. We then divide the calculated amplitude value by 127, normalizing the entry. Our amplitude value must be normalized in order to properly interact with our wavetable synthesis algorithm.

5.1.2 Max/MSP | Routing

In order to update the switches for the routing matrix we had to generate the appropriate hex values to be sent over the pyboard's I^2C bus. According to the datasheet for our ADG2128 routing chips, the value passed through to the system is an 8-bit, one byte, value. The larger of the four bits is dependent upon the input port's value, whereas the lesser of the four bits is dependent upon whether or not the input port is even-valued. The code for calculating this hex value is seen in figure 20 of Appendix C.

Once we've parsed the proper values to be passed to the chips, we first reset all of each chip's switches by pulsing the reset pin low. This pin is connected from X1 on our pyboard to pin 31 on all of our ADG2128 chips, as depicted in figure 17 of Appendix B. After resetting all of the switches, we create a byte array to which we then append the slave chip's address, the hex value of the switch to turn on and an additional byte determining whether or not there are additional switches to be parsed. The byte array then gets sent over the I^2C bus to the appropriate chip and the cycle is repeated if there are any remaining switches to handle.

5.1.3 MIDI

Incorporating MIDI into our system was somewhat easier than the code for updating the generators and routing switches. When a MIDI key is pressed, a serial message is generated that indicates the MIDI channel, note number and velocity. As seen in figure 21 of appendix C, our programming support makes use of numerous arrays to update the midi signal. The *midi_check* array is used to see if a midi note has already been designated to play on a specific output. This associated output index of the array is set to 1 when the note is first triggered, and then set to 0 upon the envelope's completion time. Currently, we have our envelope time set to a default value of 5 seconds. This was to ensure that sound could be played; however, proper MIDI implementation would use the velocity and the time the note is held to determine the note's attack, sustain and decay time.

The *amplitude* array is important for wavetable synthesis and contains the normalized version of that output's velocity/amplitude. The *frequency* array is the other primary array that must be set. Using the note number, which ranges from 21 to 108, we index into a table that contains the note number's associated frequency[9]. The final call in our MIDI interface, *update_wave*, decides which wave type will represent the MIDI note (e.g. sawtooth, sine wave, etc.) based upon the information that's already been parsed from the serial message. This code can be found in figure 22 of appendix C.

5.2 Verification

One of our primary concerns with this section came with ensuring our helper functions produced the appropriate hex and integer values. Particularly with the routing matrix, we ran the function independently of the system with various inputs and compared them to the values on our data sheets until we were certain the function was properly implemented. Similar methods were utilized in finalizing the code for parsing MIDI information and generator information from Max/MSP. However, we were unable to stabilize the connection between the Max/MSP and MIDI interface to the micropython code using SysEx (Generator/Routing) and Serial (MIDI) messages. Therefore, we were unable to update our system based on user input, an important aspect of our design.

6 Digital Signal Processing

6.1 Wavetable Synthesis and Implementation

Our wavetable synthesis algorithm serves as our actual mechanism for processing our signals and outputting them to the system. Through wavetable synthesis a single-cycle waveform is constructed and then stepped through at different intervals depending on the associated frequency. A general version of the cyclical nature of wavetable synthesis can be found in figure 18 of Appendix B.

For the initial implementation of our system, all single-cycle waveforms were calculated ahead of time. Each waveform consists of a table with 2048 entries. These entries range in data values from 0 to 4095, as our DAC, the Max 536/537, accepts this range of sample values and converts the value to its associated output voltage. As many audio signals are oriented with their maximum value at 1 and minimum value at -1, the signals should be reoriented so that their maximum is at 4095, minimum is at 0 and their mid point around which they oscillate is located at 2048. All of the waveforms were hardcoded into a single table. The only waveform not utilizing a single table from the aforementioned group was the sawtooth as different partials were introduced in different tables to avoid aliasing[10]. This resulted in the sawtooth waveform consisting of an array of 8 single cycle tables, where each table is associated with a particular octave.

$$\text{Phase\_Interval}[index] = \left(\left(\frac{\text{frequency}[index]}{\text{SAMPLE\_RATE\_IN\_HZ}} \times 2.0\right)\right) \times 1024 \quad \textbf{(6.1)}$$

***Note: SAMPLE_RATE_IN_HZ = 44100*

In order to step through this array, we calculated a phase interval value to increment the location of the sample to read in the table. This interval is depicted by equation (6.1)[10]. After we write a sample from the waveform to the DAC, we increment the current sample by the phase interval as depicted in figure 23 of Appendix C. A brief instance of this call is listed in figure 7. It's important to note that when calculating the phase_interval, it is calculated as a float value. Although only integers can be used to index into a table, by casting the sample location only when actually indexing into the table, we can ensure that the sample increment is properly handled, and not affected by rounding.

```
# 0x1[data] - DAC 1 (Zero-indexed)
    spi.write(int( data[0][int(curr_sample[0])] x amplitude[0] ) + 4096) # Adds 0x1000 (4096) to 12-bit waveform sample
    curr_sample[0] += phase_interval[0]
```

Figure 7. Writing waveform sample to DAC using SPI protocol and incrementing the sample index

In figure 24 of Appendix C, the micropython code for writing to each DAC is listed. One of the aspects of our DACs that we make use of is its ability to store each sample in an input shift register before being converted to it's analog output voltage, as seen in figure 19 of appendix B. After writing to each of the

eight outputs, we pulse the LDAC pin low. This pin is connected to both of the chips' LDAC ports and loads the DACs with the values from their associated input shift register to be immediately converted.

In our initial implementation of the system we also hardcoded for all of the audio signals to play for a duration of five seconds, defined within the *env_time* array. To update the sample and the playing time appropriately we utilized a series of nested if-loops as depicted in appendix C. The if loops checks to see if the wave_table has been fully stepped through, signalling once a complete cycle. If so, then the cycle count is decremented until it reaches zero, logically representing one second. For instance for the note A4 (440 Hz), the cycle would be set to 440, and once it reaches zero following numerous decrements, one second would be marked by decrementing the *env_time* array. Once the *env_time* value has reached zero, the audio signal is turned off as the envelope duration has been fulfilled.

6.2 Verification

One of our primary concerns with our system is the handling of our single-cycle waveform tables during processing. Each time a sample is written to the DAC port, we multiply the value stored in the table by the amplitude. We didn't expect for our code to alter our wave table, but we didn't want to take the risk of there being an error buried in the syntax that could ultimately cause such a side effect. Therefore, we ran a test that cycled through our table 100 times. We created a temporary table equal to the original waveform table and used that table as the basis to our synthesis algorithm. At the end of the 100 seconds, we compared the original waveform table to the test table and found that there was no difference between the two. Ensuring us that our code would not affect the single-cycle waveforms embedded in our microcontroller.

Aside from this success, we were not successfully able to test if the SPI protocol would properly send the data samples to the DACs. We had originally experimented with I^2S protocols; however, we found that although the pyboard could handle the communication method, the SPI protocol was a better option as it was built into our version of the pyboard. We were, however, able to pass values through the DAC embedded on the pyboard microprocessor, which gives us hope that the signals can be properly passed from the pyboard to the Max 536 DAC once the system is fully rigged.

7 VCA/VCF Board

7.1 Design Procedure and Details

7.1.1 VCA/VCF Filter Core

The VCA/VCF board, figure 16 in appendix B, was designed around the VCA/VCF Filter core, figure 15 in appendix B. As our system's VCA/VCF filter chip, the NJM2069, was once a proprietary product of Korg, no datasheet exists to facilitate its use. To circumvent this issue, measurements of a working synthesizer which utilizes the chips were taken. It was found that, at the inputs of certain surrounding resistors, maximum and minimum values of the control voltages were of equivalent magnitudes. These magnitudes fell approximately between either 0V and 375mV for most control voltages, -375mV and 0V for the VCA control voltage, and -375mV and 375mV in the case of the frequency control voltage, with -375mV corresponding to the highest cutoff frequency. The block including the filter and these resistors was closely implemented from the schematic of the keyboard in question (the Korg DW-8000)[11].

As signal input and output voltages fell in a rather large range between ~225mVpp (at the SIG1 and SIG2 inputs and amplified VCA output) and ~750mVpp (at the VCF_{IN} input and at the -12dB and -24dB outputs of the VCF), the control voltages were used to devise a standard for the amplification of inputs and outputs. A 500mVpp input voltage was assumed for all inputs (based on the typical output level from the VCF); precision rectifiers (shown in section 7.2) were used wherever the input voltage was intended to stay between 0V and 375mV (or between -375mV and 0V), and amplifiers were used to raise or lower the voltage as necessary. These values are detailed in their respective subsections.

7.1.2 Control Voltage Processors/Amplifiers

The control voltage (CV) input processors, depicted in figure 12 of appendix B, were designed to act as buffers to negate the somewhat variable output resistance of the switchpoint matrices, as rectifiers where control voltages are intended to fall exclusively to one side of ground, and as amplifiers to normalize the input level

Each of the single-member input ports at the top of the schematic, labelled F_FREQ, VCA_LVL, F_RES, and SIG1_LVL, is a header intended to be connected to the washer of a potentiometer, with the assumption that the potentiometers will be held between the appropriate power-rail values (0V and 5V or -5V and 5V in the case of the filter frequency) at the chassis. The potentiometers are designed to provide DC offsets to time-varying voltages, or to hold control voltages at constant levels.

The CTRL_IN port receives, from left to right, the filter frequency CV, the VCA level CV, the filter resonance CV, and the LVL1 CV (which controls one of the filter's two internal VCAs, capable of altering the level of SIG1). The filter frequency control voltage was found to be both inverted and bipolar, falling between approximately -375mV with the filter fully open, and 375mV with the filter fully closed. Hence, a single inverting amplifier with an input resistance R_{In} = 75 kΩ and a feedback resistor R_F = 100 kΩ, resulting in an amplification of an amplification of $\frac{100\ k\Omega}{75\ k\Omega} \approx 1.33$, which would bring a 500mVpp input to 750mVpp, precisely the desired range, while also inverting it. A 1.5 MΩ resistor is connected from the negative input to the potentiometer, which would result in a gain of $\frac{100\ k\Omega}{1.5\ M\Omega} \approx 0.066$, bringing the

maximum/minimum +/-5 V down to -/+333 mV, not exactly the full range but close enough for our purposes.

The precision rectifier is a circuit which was borrowed from an article by Rod Elliot[12]. It acts as a rectifier even when input voltages are smaller than a diode's "on" voltage. We ultimately chose this rectifier for our system as its input is an inverting amplifier rendering it capable of gain. The gain is determined by the ratio of the internal resistors (e.g. The four 68 kΩ resistors R55-R58 in the 2 op-amp circuit connected to the VCA_LVL input and R40 in figure 12 in appendix B, which should be kept equal), and the input resistor.

This also means that the rectifier can act as a summing amplifier, therefore a similar gain calculation can be used for the three precision rectifiers as it was used for the filter frequency amplifier. The gain values for each are identical, bringing a 500 mV input to 375 mV and a 5 V signal from the potentiometers to approximately 333 mV, but the rectifier connected to the VCA control voltage has a unity-gain inverter at the output to account for the expected inverted input.

7.1.3 Audio Input Processors/Amplifiers

A 2.2 µF capacitor is included at each of the inputs for the audio input processors, figure 13 of appendix B, to account for any DC component which may be present. Unity gain buffers are also utilized at the VCA and VCF inputs, as a 500 mV input voltage corresponds to a medium-volume signal for both. The SIG1 and SIG2 inputs expect a voltage between 225 mV and 275mV for an approximately medium-gain signal at the VCF output, and are therefore connected to amplifiers with a gain of 0.5.

7.1.4 Output Processors/Amplifiers

The output of the VCA is connected to a 100 kΩ audio-taper potentiometer on the front of the chassis through the VCA_VOL port. A 27 kΩ feedback resistor was found to produce an output signal with a magnitude of approximately 225 mVpp, and hence a 100 kΩ potentiometer is used in place of the feedback resistor to allow the user to vary the gain between 0 (no output) and just under 750 mVpp. As the VCAs on the NJM2069 were found to invert the signal (including the VCAs internal to the VCF which modulate the level of SIG1 and SIG2, hence the inverting amplifiers at their input), a single inverting amplifier is all that's needed. The schematic for the processors can be found in figure 14 of appendix B.

Meanwhile, the -12dB and -24dB outputs of the VCF are sent to SPDT switches on the front-panel to allow the user to vary the slope of the filter by hand. The output of the filter was found to be around 500mV on average (with 250 mVpp signals applied to both SIG1 and SIG2, typical operation conditions on the DW-8000), but an inverting amplifier with an input resistor of 47 kΩ and a feedback resistor connected to a 100 kΩ audio-taper potentiometer on the front panel allow for gain from 0 to approximately 1 Vpp.

8. Costs

8.1 Parts

Table 1 Parts Costs

Part	Quantity	Manufacturer	Retail Cost ($)	Bulk Purchase Cost ($)	Actual Cost ($)
D-Series Pyboard	1	George Robotics Limited / Micropython	84.1500	70.1300	84.15
ADG2128 8x12 Analog Switch Array	8	Analog Devices	12.4100	8.5600	99.18
TLO74 Low-Noise, JFET-Input op-amps	22	Texas Instruments	0.4400	0.1500	15.40
MAX537BEWE+ 12-bit Output Voltage DAC	2	Maxim Integrated	26.5800	22.3500	82.42
AD8672 Precision Low Input Bias op-amp	4	Analog Devices	2.0400	1.6200	8.16
1/4W 1% Through-Hole Resistors	132	Stackpole Electronics	0.06800	0.0085	8.98
Ceramic SMD 47pF Capacitor	40	Kemet	0.1000	0.0360	4.00
NJM2069 analog filter/amplifier chips	4	Korg	22.4900	22.4900	89.96
1N4148 Diodes	24	Semtech Electronics LTD	0.1000	0.1000	2.40
100 kΩ, 1/20W Resistor	8	Yageo	0.0970	0.0034	0.78
20 kΩ, 1/16W Resistor	8	Yageo	0.3900	0.0450	3.12
2.2kΩ, 1/10W Resistor	4	Yageo	0.0970	0.0034	0.39
10 kΩ, 1/10W Resistor	12	Yageo	0.1700	0.0220	2.04
50 kΩ, 1/5W Trimmer	20	Bourns Inc.	0.4400	0.2100	9.68
100 kΩ, 1/4W potentiometers	36	Alpha	1.5000	1.3500	54.00
M/F Pin Connector Kit	1	Glarks (via Amazon)	13.9900	13.9900	13.99
SPDT miniature panel-mount toggle	4	TAIWAY	1.9500	1.2500	7.80
4-Pin Extension Strip	1	AOTOINK (via Amazon)	8.9900	8.9900	8.99
LM317 Voltage Regulator	4	Texas Instruments	1.5900	0.5290	6.36
Connector Receptacle	2	Hirose Electric Co Ltd	1.1000	0.7560	2.20
HS350-ND - Heatsinks	2	Aavid	1.3200	0.9833	2.64
ED11093-ND - 0.4” DIP	4	Mill-Max Manufacturing	1.0800	0.5900	4.32
0.1µF, 63V capacitors	56	Kemet	0.1460	0.0790	8.18
Total					**519.14**

8.2 Labor

Based on average ECE salary

- BS Computer Engineering Average Salary: $84.25k
- BS Electrical Engineering Average Salary: $67k

40 hours per week over 6 weeks = 240 hours

Per Partner:
($30/hour) x 2.5 x 240 hours = $18,000

Team Cost: $54,000.00

Total Cost: $54,519.14

9. Conclusion

9.1 Accomplishments

While we were unable to get a fully functional prototype of our synthesizer we were able to get multiple modules implemented. Our micropython support allows for the user to handle the signals to parse out the appropriate information and update the waveforms accordingly. In addition, we were able to test some of our components and found that they were functional and operating as expected. These both serve as solid steps towards actualizing the hybrid analog/digital synthesizer we have designed. Our primary concern is with finalizing the connection point between the user interface and the pyboard and from the pyboard to the DAC. Establishing the connection between the user interface and the PyBoard through SysEx and serial messages is a necessary step to full product functionality. In addition, once we can solidify the connection between the micropython code and the chips through I^2C communication and SPI protocol we will be able to output our audio signals as intended.

9.2 Ethical Considerations

We could think of very few ethical considerations for the project, largely due to the nature of the product. The synthesizer is a specialized tool for audio creation, a design that doesn't carry immediate implications towards an individual's privacy or security.

It should be noted that in the event of mass production, NJM2069 chips are no longer produced, which could cause potential issues. In light of this, it must be noted that not many other chips are on the market, aside from the recently re-released Roland 80017A VCF / VCA JUNO-106 Voice Chip Filter IC[14]. These chips were originally released in 1984 and were eventually discontinued. Newer versions of this chip are likely to be of better quality than their original iteration in 1984, however, the unreliability of the chips isn't something we would want in our future. It's also worth considering that the NJM2069A may be put back into production as the 80017A due to the rise in demand for these powerful audio chips.

This does bring up some potential ethical issues with regard to the environment. When companies decide to start producing any component or device, it requires factory space, machines and either human or artificial labor. In a world where factories contribute heavily to the changing of the climate, the repercussions of producing such a component must be considered, as per the IEEE Code of Ethics[13].

9.3 Future Work

This synthesizer design presents a number of novel ideas. That having been said, there are plenty of ideas that would be great additions to the system that already exist. For example, implementing an attack, sustain, and decay feature for the audio signals would definitely help create an experience the user is expecting. Likewise, MIDI note implementation should be re-configured to allow for notes of varied lengths, depending on how the user plays the keyboard. This would also mean that the note-off signal needs to be handled in the micropython code. Beyond these factors and finalizing the connection points between the system modules, it would also be interesting to see the amplitude envelope feature be fully implemented so that users can also program the amplitude over time of their audio.

References

[1] Ken Stone, '*Power Supply for Music Synthesizers*', CGS Synthesizers. Accessed on: Oct. 2, 2020. [Online]. Available: http://www.synthpanel.com/modules/cgs66_psu.html

[2] '*LMx37 3-Terminal Adjustable Regulators (Rev. L)',* Texas Instruments. Accessed on: Dec. 8, 2020. [Online]. Available: https://www.ti.com/document-viewer/LM337/datasheet

[3] '*LMx17 3-Terminal Adjustable Regulators (Rev. Y)',* Texas Instruments. Accessed on: Dec. 8, 2020. [Online]. Available: https://www.ti.com/document-viewer/LM317/datasheet

[4] '*LM78XX/LM78XXA 3-Terminal 1A Positive Voltage Regulator',* Mouser. Accessed on: Dec 8, 2020. [Online]. Available: https://www.mouser.com/datasheet/2/308/LM7805-1301244.pdf

[5] *'LM79XX 3-Terminal Negative Regulators (Rev C)'*, Texas Instruments. Accessed on: Dec. 8, 2020. [Online]. Available: https://www.ti.com/product/LM79

[6] *'MAX537 Calibrated, Quad, 12-Bit Voltage-Output DACs with Serial Interface',* Maxim Integrated. Accessed on: Nov. 22, 2020. [Online]. Available: https://www.maximintegrated.com/en/products/analog/data-converters/digital-to-analog-converters/MAX537.html

[7] User 'alustig3', '*PYBD WBUS Connector',* Micropython.org, Jun. 5, 2018. Accessed on: Dec 8, 2020. [Online Forum Comment]. Available: https://forum.micropython.org/viewtopic.php?t=6164

[8] '*Pyboard D-Series Reference',* pybd.io. May 11, 2020. Accessed on: Dec 8, 2020. [Online]. Available: https://pybd.io/hw/pybd_sfxw.html

[9] Joe Wolfe, '*Note Names, MIDI Numbers and Frequencies*', UNSW. Accessed on: Dec. 8, 2020. [Online]. Available: https://newt.phys.unsw.edu.au/jw/notes.html

[10] Phil Burk, "Bandlimited Oscillators Using Wave Table Synthesis," in *Audio Anecdotes: Tools, Tips and Techniques for Digital Audio,* vol. 2. R Barzel and K. Greenbaum, Eds. Wellesley, MA: A. K. Peters, 2004, ch. 7, pp. 37–52.

[11] '*Korg DW-8000 Service Manual',* manualslib. Accessed on: Sept. 28, 2020. [Online]. Available: https://www.manualslib.com/manual/997976/Korg-Dw-8000.html

[12] Rod Elliot, "Precision Rectifiers", ESP. Accessed on Oct. 4, 2020. [Online]. Available: https://sound-au.com/appnotes/an001.htm

[13] '*IEEE Code of Ethics',* IEEE. Accessed on: Dec. 8, 2020. [Online]. Available: https://www.ieee.org/about/corporate/governance/p7-8.html

[14] "Roland 80017A", Polynomial.com, n.d. Accessed on: Oct. 30, 2020. [Online]. Available: https://www.polynominal.com/roland-80017a/

Appendix A Requirements and Verifications Table

Software Interface

Requirement	Verification	Status
Display menus, interfaces and user input fields appear on the screen, without error and can be updated appropriately.	1. Input values into each user input field. 2. Set generators using these input values 3. Print out the generator values to ensure the values are being properly parsed by the code and stored in the designated buffer.	Yes
All menus must be togglable and feature each option allocated by the design.	1. Click on each option in the drop-down menus on the interface. 2. Ensure all features are listed within the drop down function. 3. For drop-down functions that alter the screen, select these functions and ensure that the screen makes the necessary alterations. 4. For each drop down function, set the generator. Include random sample values for other parameters. 5. Print contents of the updated generator to ensure each feature selection is linked to the proper variable in the code that gets stored.	Yes

Microcontroller

Requirement	Verification	Status
Produces no aliasing for any oscillator waveform and all notes up to C8 - highest note on a grand piano - with frequency 4186.01 Hz	1. Display waveform with frequency >= 4186.01 on oscilloscope. 2. Rotate the horizontal scale knob, or otherwise change the horizontal scaling of the oscilloscope. If the waveform changes drastically, aliasing is present	No

Wavetable Synthesis

Requirement	Verification	Status
The wavetable synthesis program must not alter the single-pass waveform stored in the buffer. While not altering what's stored in the buffer, it also must use the buffer and properly process the waveform at the desired frequency to produce the correct audio signal.	1. Store signal of single amplitude in the wave_form buffer. 2. Pass through the wavetable synthesis algorithm at a single frequency. 3. After passing the wave_form buffer through the algorithm 1000 times, check the buffer has not been altered 4. Ensure that the output signal is a single-tone audio output of a single-level. If fluctuations in pitch or frequency are detected, there is an error.	Yes (not no.4)

Digital-to-Analog Interface

Requirement	Verification	Points
3.3Vp-p (+/- 0.3V error) digital waveforms at the input to the DAC during operation.	1. Using an oscilloscope, ensure a digital version of the intended waveform is present at the input pin and conforms to the desired standard	Yes
3.3Vp-p output (+/- 5% error) continuous Analog Output signal at DAC output pins	1. Set to ready/inactive (through the GUI or underlying program if the GUI is not yet debugged) a simple output waveform to the GPIO connected to DAC1 2. Connect the I2C_SCL (PB10) and I2C_SDA (PB11) outputs of the pyBoard to the corresponding I2C inputs on the LMP92001 DAC 3. Ensure, through LMP92001 address bits 31:32, and by setting the I2C configurations in the underlying program to mirror the address of 31:32, that you are communicating with the proper DAC 4. Measure output at the corresponding output pin using a digital oscilloscope and ensure it conforms to the given requirement 5. Repeat for all DACs	No
Must properly convert any input digital signals to audio or DC envelope signals.	1. Pass a single-tone audio signal into the DAC. 2. Test the audio it produces digitally 3. Connect the output of the DAC to an analog speaker. 4. The audio signal produced by the analog speaker must match that digitally produced during the aforementioned test. 5. Repeat with varied audio signals.	No

Routing Matrix

Requirement	Verification	Points
Verify that ADG2128 Chips Pass Signal Through All Connection Points	Test chips individually. 1. Manually verify that test DC voltages of +/-3.3V are passed through each of the 96 ports when operated individually on a +/-5V power supply. 2. Manually verify that a +/- 3.3V sine wave passes through all ports when operated on a +/- 5V power supply.	Yes
Verify that signals on separate ADG2128 chips carrying the same buffered output to different buffered inputs have no interaction with one another, or that signal interaction occurs at a low enough level as to be negligible for the listener/user (difference in peak-to-peak voltage of less than 2% from unconnected waveform)	1. View signals on an oscilloscope individually, without the other connected. Measure peak-to-peak voltages. 2. View signals on the oscilloscope while both signals are connected. Ensure minimal or no change	No
Verify that user input into the routing matrix via Max/MSP produces the correct output hex and the correct number of output hexes	1. Randomize switches that user sets in Max/MSP 2. Send serial message containing number of switches to be set and the matching I/O ports into the python code to be parsed 3. Print out the message to be passed over the SDA line by I^2C for each switch to be set. 4. Verify that signals match the corresponding values in user datasheet 5. Run test minimum 5 times with varied number of switches, ensure 100% accuracy	Yes

Appendix B Schematics/Figures

Figure 8. Analog/Digital Board Schematic

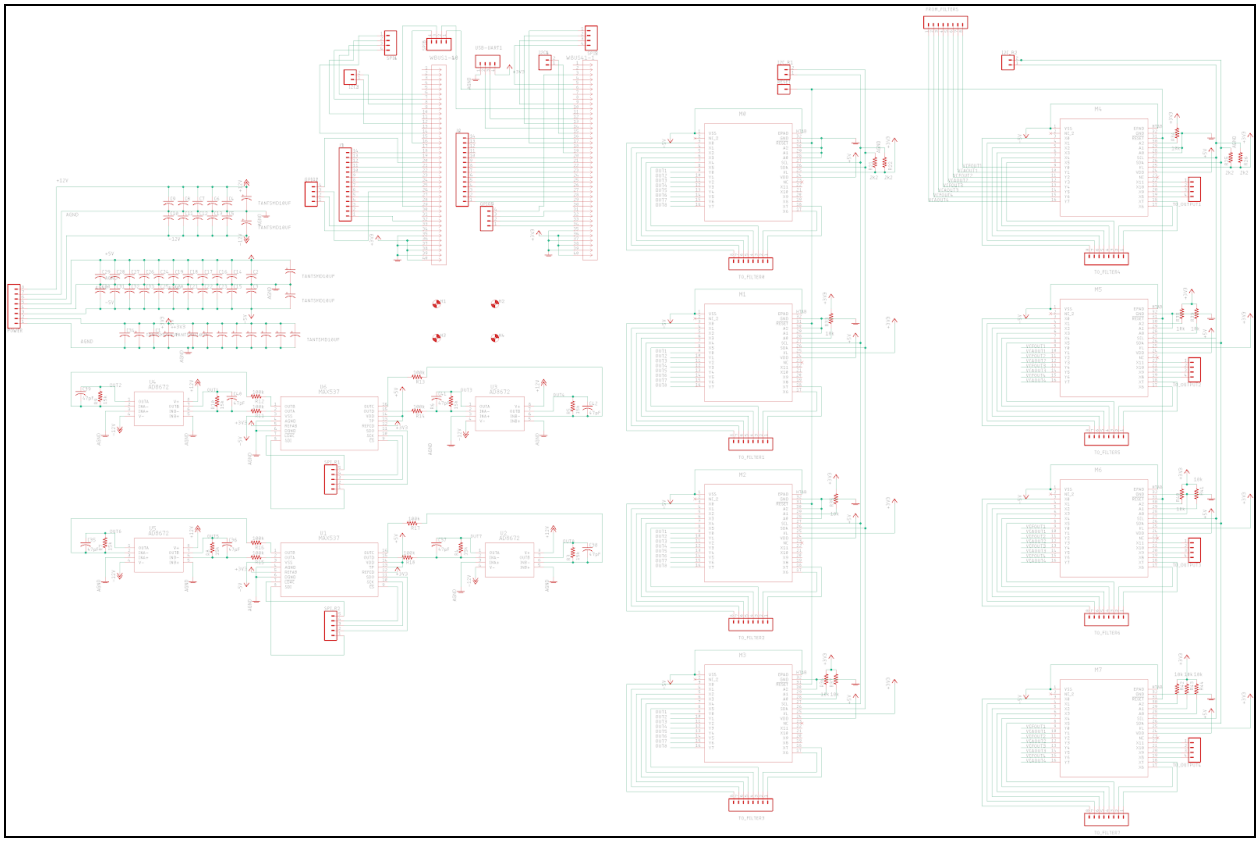


Figure 9. DAC Chips and Output Amplifiers

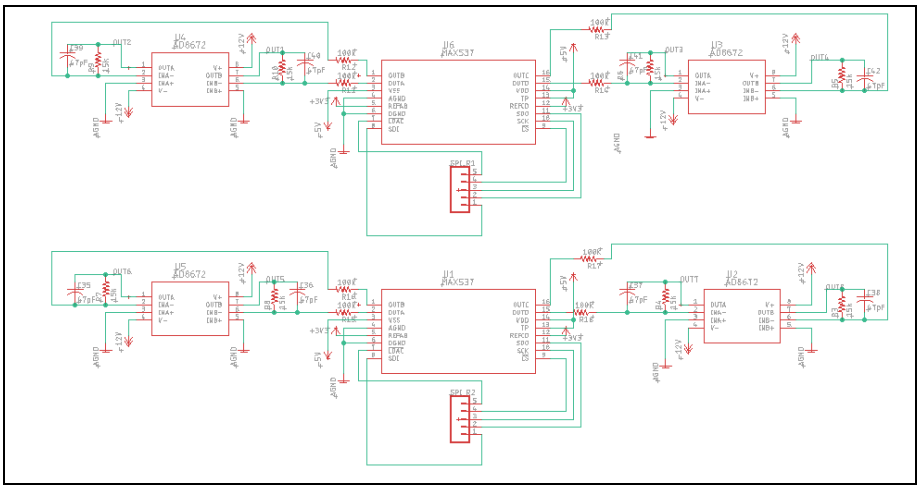


Figure 10. Routing Matrix Detailed

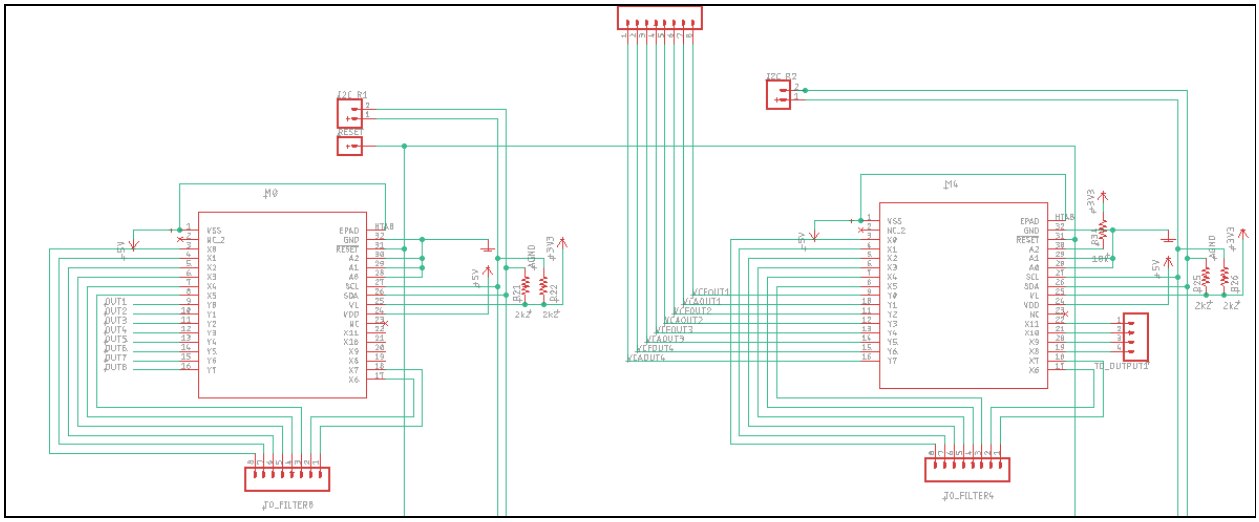


Figure 11. PyBoard Mezzanine Input Ports and Corresponding Output Ports

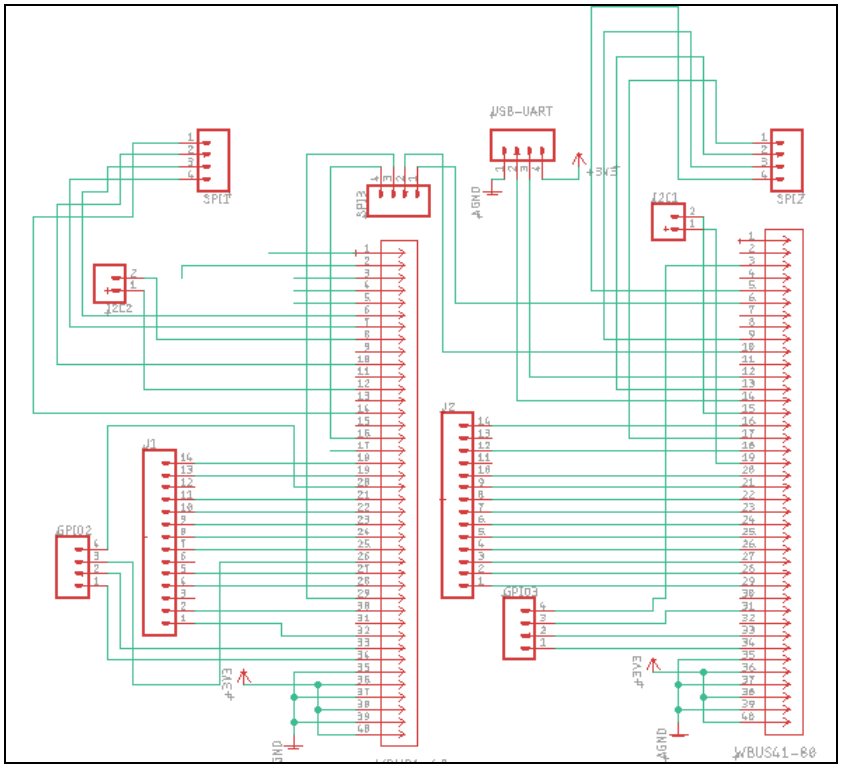


Figure 12. CV Input Processors

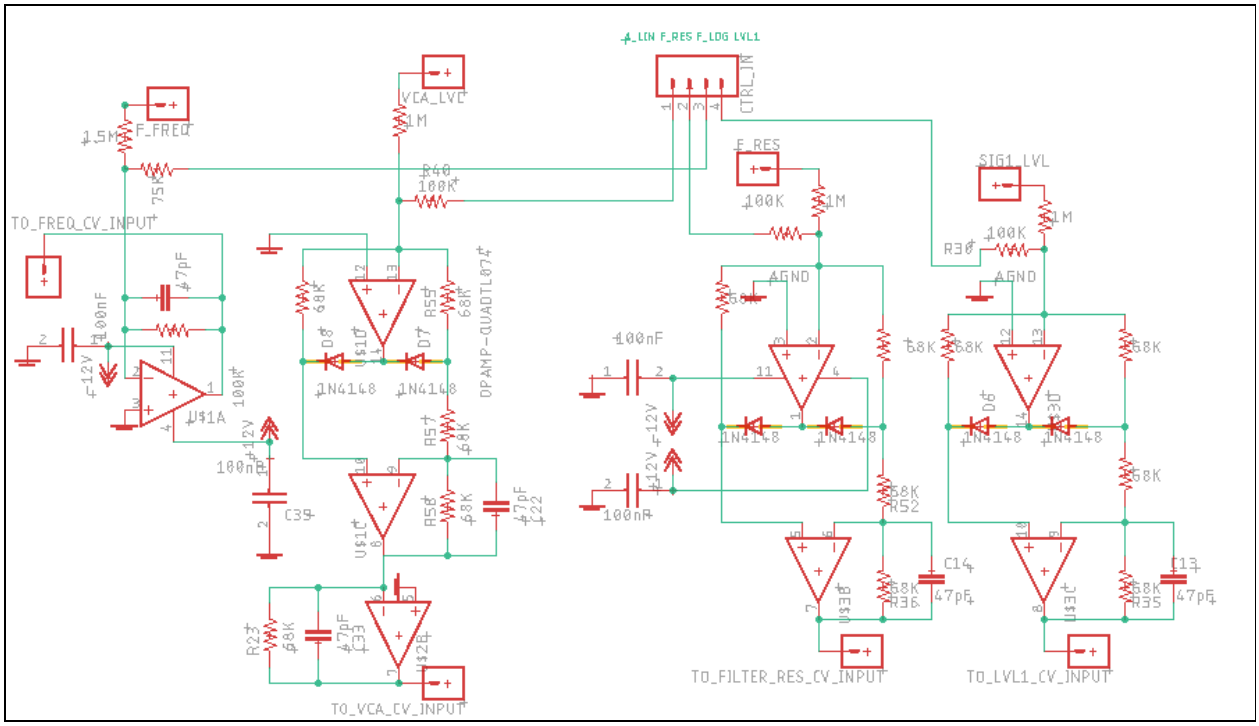


Figure 13. Audio Input Processors

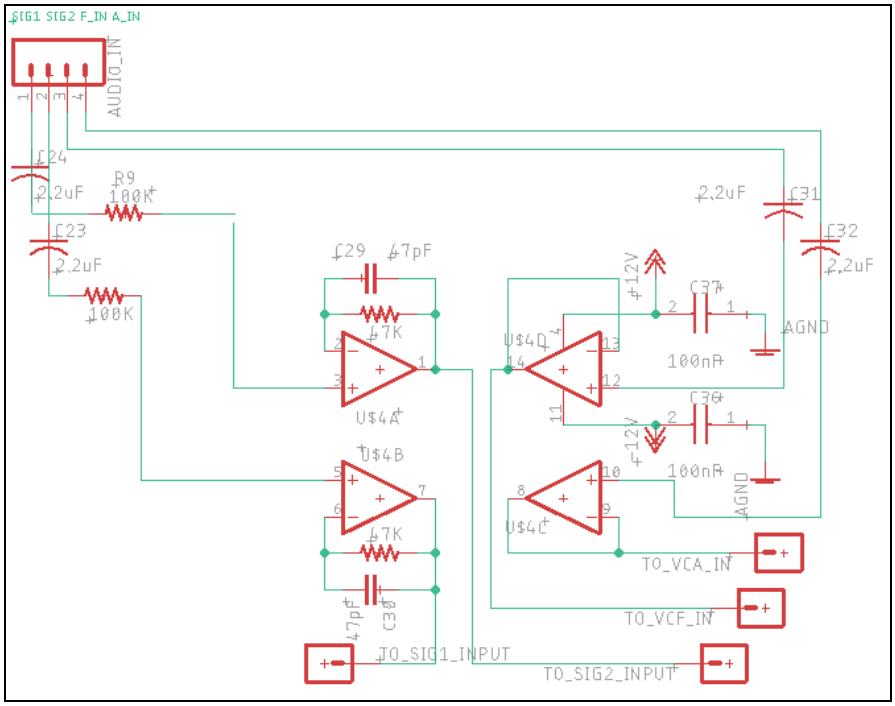


Figure 14. Output Processors

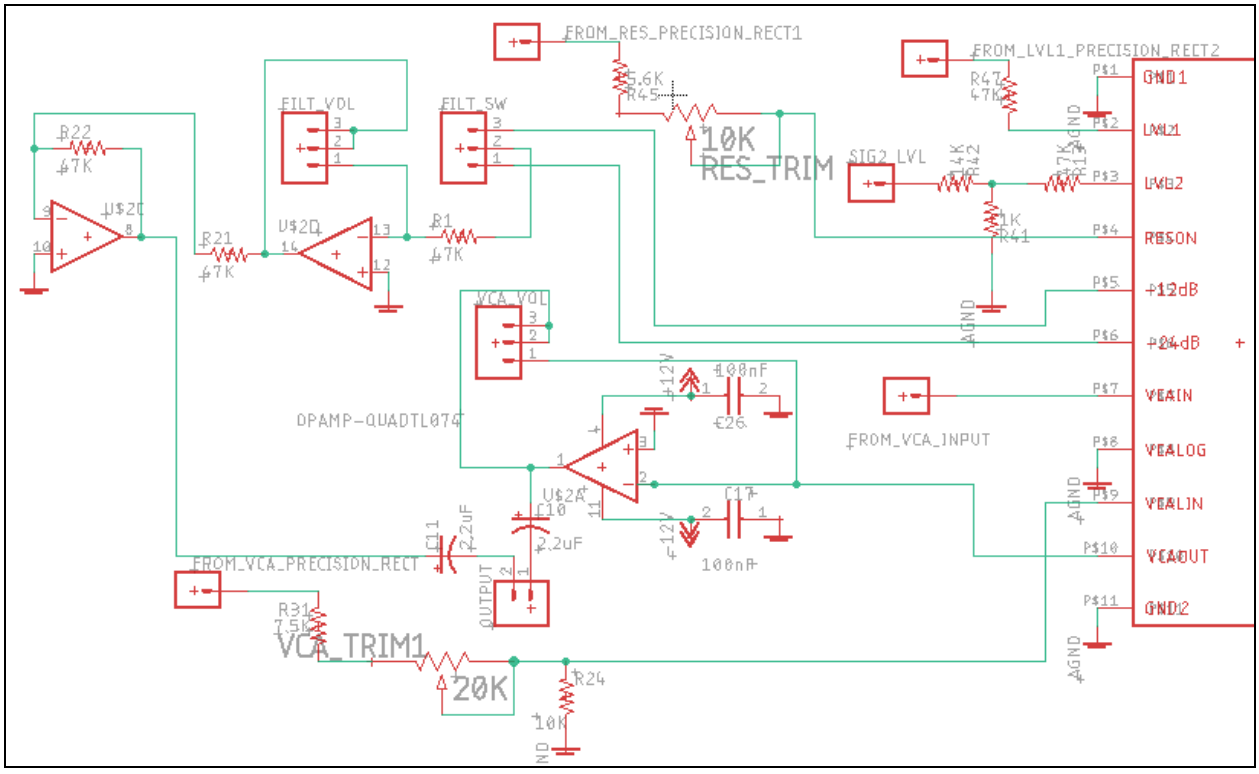


Figure 15. VCA/VCF Filter Core

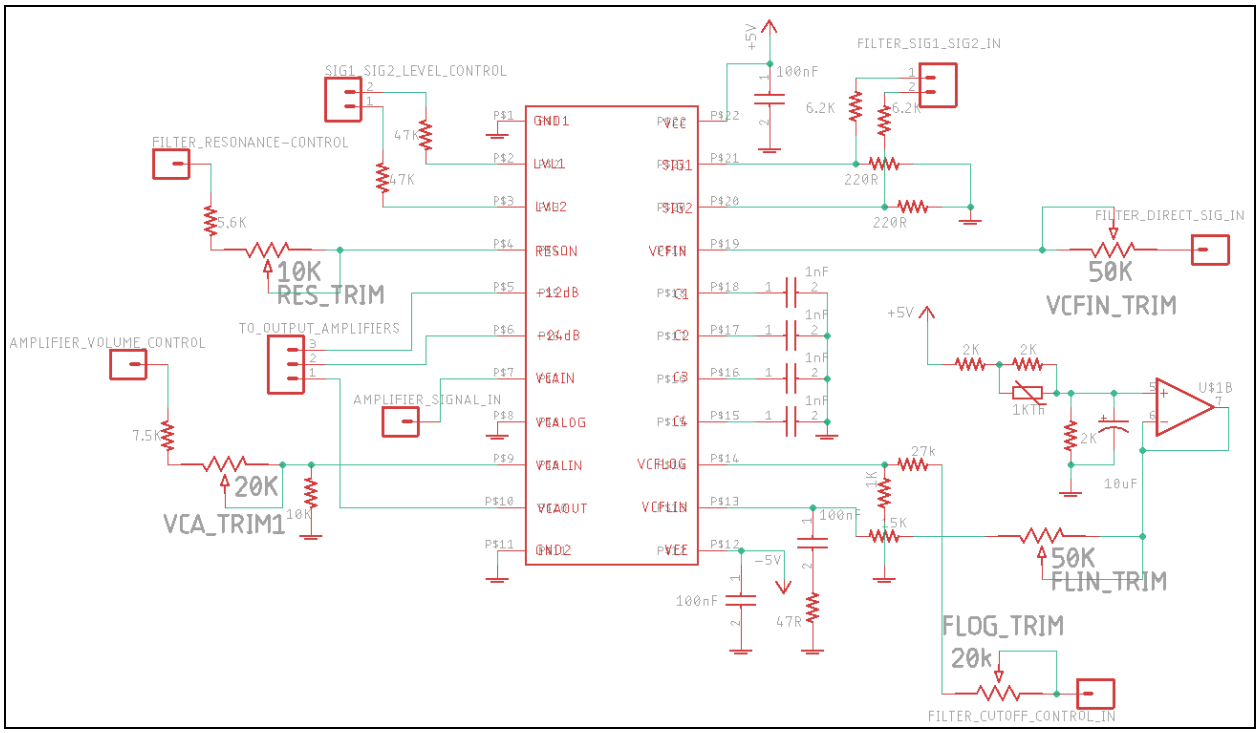


Figure 16. VCA/VCF Filter Board Schematic

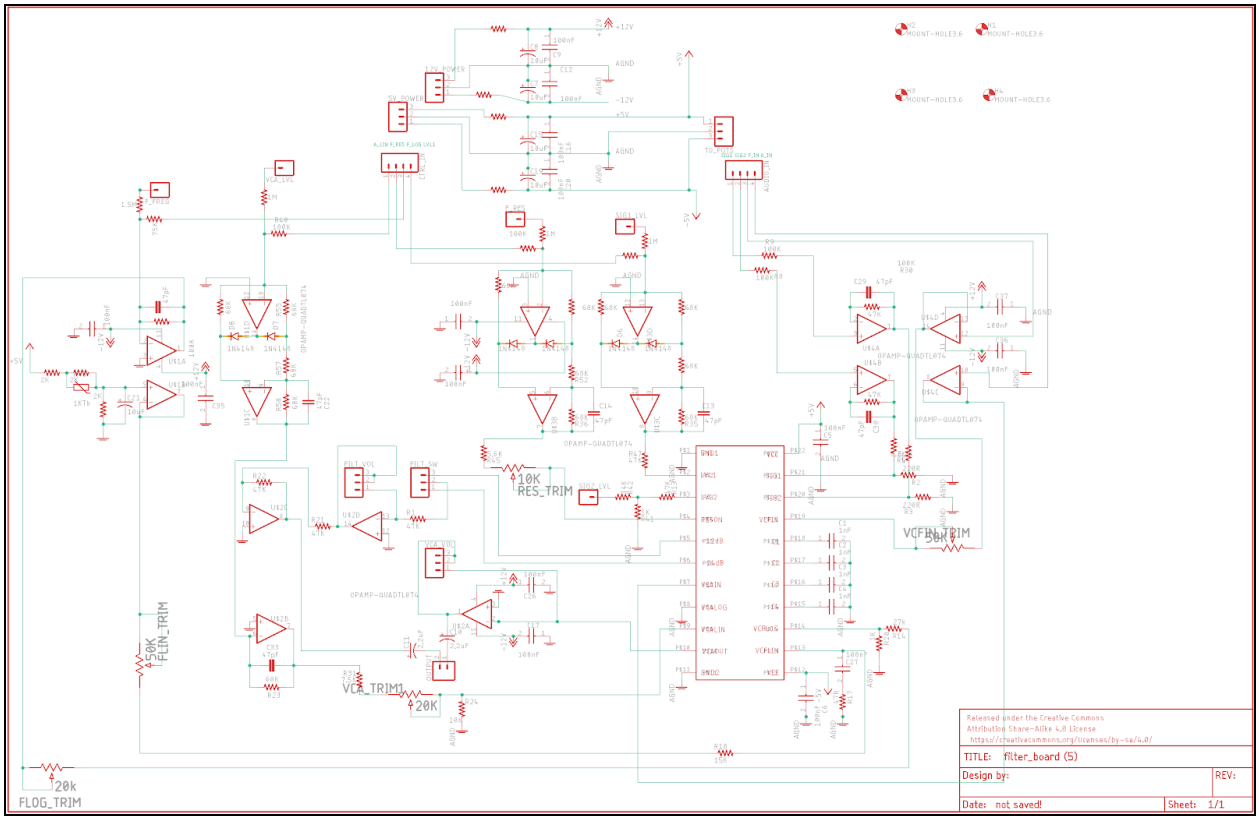


Figure 17. ADG2128 Pin Layout

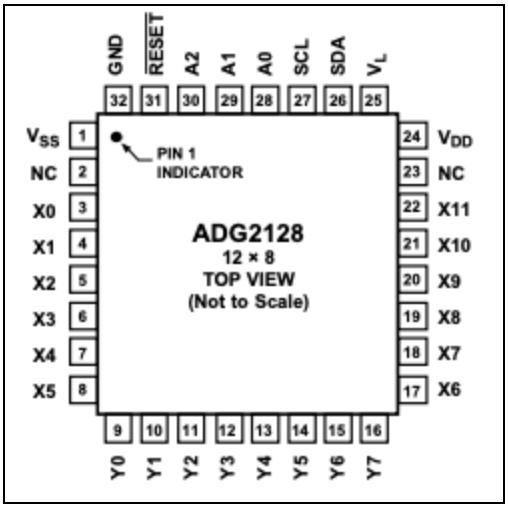


Figure 18. Oscillator Schematic

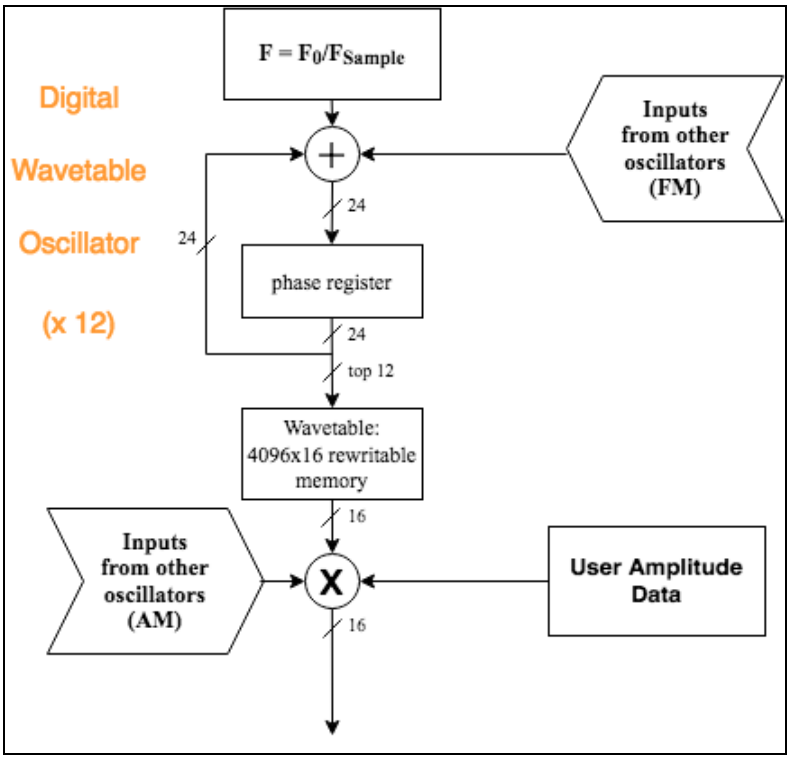
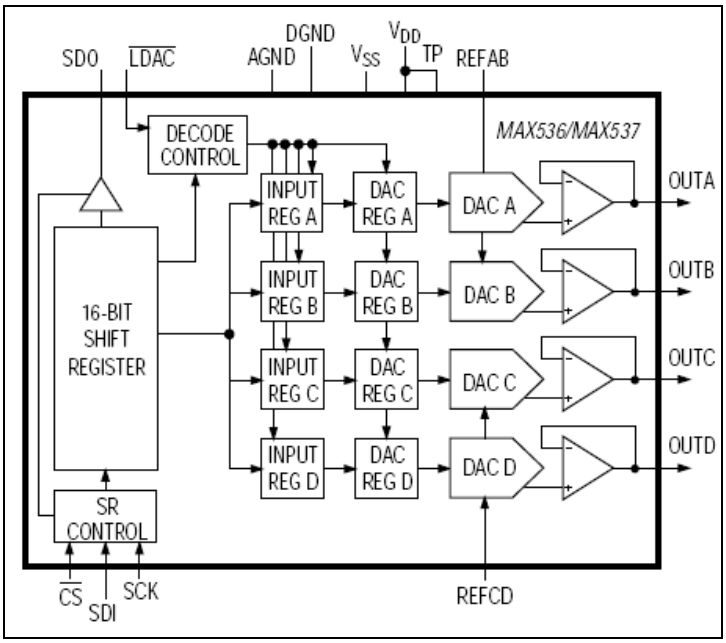


Figure 19. MAX 536/537 (DAC) Schematic



Appendix C Micropython Code

Figure 20. Routing Update - *get_hex*

Function: Returns the hex value to activate the switch between a particular input and output.

```
# Get hex value according to x and y values passed
# in of matrices I/O and whether to switch on/off
def get_hex(x,y):
    output = 0
    if(x == 0 or x == 1):
        output = 128 # 0x80
    elif(x == 2 or x == 3):
        output = 144 # 0x90
    elif(x == 4 or x == 5):
        output = 160 # 0xA0
    elif(x == 6 or x == 7):
        output = 192 # 0xC0
    elif(x == 8 or x == 9):
        output = 208 # 0xD0
    elif(x == 10 or x == 11):
        output = 224 # 0xE0
```

```
# If x is even
if(x%2 == 0):
    if(y == 0):
        output += 0 # 0x00
    elif(y == 1):
        output += 1 # 0x01
    elif(y == 2):
        output += 2 # 0x02
    elif(y == 3):
        output += 3 # 0x03
    elif(y == 4):
        output += 4 # 0x04
    elif(y == 5):
        output += 5 # 0x05
    elif(y == 6):
        output += 6 # 0x06
    elif(y == 7):
        output += 7 # 0x07
```

```
else:
    if(y == 0):
        output += 8 # 0x08
    elif(y == 1):
        output += 9 # 0x09
    elif(y == 2):
        output += 10 # 0x0A
    elif(y == 3):
        output += 11 # 0x0B
    elif(y == 4):
        output += 12 # 0x0C
    elif(y == 5):
        output += 13 # 0x0D
    elif(y == 6):
        output += 14 # 0x0E
    elif(y == 7):
        output += 15 # 0x0F

return output
```

Figure 21. MIDI Update - *handle_midi_msg*

Function: Parses note number and velocity from serial message and updates oscillators using arrays, logical shifts and bit-masking

```
def handle_midi_msg(msg):
    midi_signal = 0
    midi_input = 0
    if(20 < ((msg >> 8) & 0xFF) < 109): # Check if MIDI note number is within range to be handled
        for x in range (8):
            # MIDI automatically overwrites pre-defined audio generators from Output 7 to Output 0
            if(midi_check[7 - x] == 0):
                midi_check[7 - x] = 1
                amplitude[7 - x] = (msg & 0xFF)/127 # Parse amplitude and normalize
                nn[7 - x] = (msg >> 8) & 0xFF # Parse note number and store
                env_time[7 - x] = 5 # Default play time of 5 seconds
                frequency[7 - x] = wave_forms.midi_frequency[nn[7-x] - 21] # Get frequency from table
                cycles[7 - x] = int(frequency[7 - x]) # Define number of cycles per second

                # Calculate interval between each sample point index for wave_table size 2048
                phase_update[7 - x] = (2.0 * (frequency[7 - x] / SAMPLE_RATE_IN_HZ)) * 1024
                # call function with current midi index, random wave type (excluding noise) and frequency
                update_wave((7-x), int(random.random() * 4), frequency[7 - x])
```

Figure 22. Waveform Update - *update_wave*

Function: Sets the *data* array to contain the single-cycle wave-form for the associated oscillator based off input type and frequency.

```
def update_wave(index, wave_type, freq_input):
    if(wave_type == 0):
        data[index] = wave_forms.sine_table # Set as default array
    elif(wave_type == 1):
        data[index] = wave_forms.saw_table[(int(0 - log((4 * (freq_input / SAMPLE_RATE_IN_HZ)), 2)) - 1)]
    elif(wave_type == 2):
        data[index] = wave_forms.square_table
    elif(wave_type == 3):
        data[index] = wave_forms.triangle_table
    elif(wave_type == 4):
        data[index] = []
        for x in range(2048):
            data[index].append(int(random.random() * 4095))
```

Figure 23. Digital Signal Processing - *Updating sample*

Function: Moves to next sample, and performs necessary checks to decrement cycle and envelope time. If envelope time reaches zero, the associated oscillator is “turned off.”

```
curr_sample[0] = curr_sample[0] + phase_interval[0] # Move to next sample point
if(curr_sample[0] > 2047): # Check if cycle has completed
    curr_sample[0] = curr_sample[0] % 2048 # Wrap sample location to start
    cycles[0] -= 1 # Decrement the number of cycles remaining
    if(cycles[0] <= 0): # Check if cycles has reached 0
        cycles[0] = frequency[0] # Reset number of cycles per second (frequency)
        env_time[0] -= 1 # Decrement remaining envelope time
        if(env_time[0] <= 0): # Check if envelope has been fully decremented
            data[0] = wave_forms.zeros_table # Essentially, turn off output port
            midi_check[0] = 0 # Essentially, turn off midi port and prepare port for next signal
```

Figure 24. Digital Signal Processing - *Writing to the DAC*

Function: Write data value to the DAC's input shift register using SPI protocol. Pulses LDAC low to update all DACS simultaneously.

```
# SPI signal to update all dacs
# SPI 1
cs_1.low()
# 0x1[data] - DAC 1
spi_1.write(int(data[0][int(curr_sample[0])] * amplitude[0]) + 4096) # Adds 0x1000 to data
# 0x5[data] - DAC 2
spi_1.write(int(data[1][int(curr_sample[1])] * amplitude[1]) + 20480) # Adds 0x5000 to data
# 0x9[data] - DAC 3
spi_1.write(int(data[2][int(curr_sample[2])] * amplitude[2]) + 36864) # Adds 0x9000 to data
# 0xD[data] - DAC 4
spi_1.write(int(data[3][int(curr_sample[3])] * amplitude[3]) + 53248) # Adds 0xD000 to data
cs_1.high()

# SPI 2
cs_2.low()
# 0x1[data] - DAC 4
spi_2.write(int(data[4][int(curr_sample[4])] * amplitude[4]) + 4096) # Adds 0x1000 to data
# 0x5[data] - DAC 5
spi_2.write(int(data[5][int(curr_sample[5])] * amplitude[5]) + 20480) # Adds 0x5000 to data
# 0x9[data] - DAC 6
spi_2.write(int(data[6][int(curr_sample[6])] * amplitude[6]) + 36864) # Adds 0x9000 to data
# 0xD[data] - DAC 7
spi_2.write(int(data[7][int(curr_sample[7])] * amplitude[7]) + 53248) # Adds 0xD000 to data
cs_2.high()

# send signal to active low LDAC
# should be connection on from master (43) to both slaves's (7)
ldac.low()
ldac.high()
```

Appendix D Tables

Table 2 Oscillator Configurations and Sources

Options	Oscillator Types	Frequency Sources	Amplitude Sources	Operations
	Sine	Constant	Constant	+
	Sawtooth	Formula	Formula	*
	Square	MIDI - NN	MIDI - Velocity	
	Triangle			
	Noise			