# PHYS 319 Final Project: PWM-Based Synthesizer

Ashtan Mistal

April 2022

# Contents

# 1 Abstract

In this project, a 3 oscillator, MIDI compatible synthesizer was built, along with a self-complete PWM based synthesizer. The PWM-based synthesizer features mostly consistent waveforms, a low I/O pin count, and is powered directly through the microprocessor. The 3 oscillator version unfortunately suffered from numerous hardware and software limitations, mainly due to the speed of the microprocessor, the number of I/O available, and the number of ADCs available for use. Although it was concluded that the 3 oscillator synthesizer was not in working enough condition due to these limitations, the PWM-based synthesizer was playable. The 3 oscillator version functioned well as a noise generator and experimental synthesizer. Both versions of the synthesizer would offer more use as a low frequency oscillator due to such limitations, but nonetheless hold use within an analog synthesizer setup.

# 2 Introduction

Exploration into the possibility of creating a user-controlled, MIDI[1]-compatible fully featured synthesizer has held a personal interest shortly after I became interested in music production itself. Motivation for this project to encompass this personal interest stemmed from first using pulse width modulation to change the brightness of a bulb during an earlier lab. Since my more recent interest in analog audio equipment itself, I was motivated to do an audio-related project. The purpose of this project, as a result, was to help me gain a further understanding into the inner workings of oscillators and envelope generators, gain an appreciation for the MIDI standard, and of course to have a working, hand-made synthesizer for future music production related use. All three areas of knowledge mentioned above would aid in future audio-related projects, both in software as well as in hardware, and help inspire for future related projects. This project builds off of my previous related experience in music production, which provided me with a background on audio synthesis and processing.

[1] musical instrument digital interface

# 3 Theory

In the following subsections, some basics regarding audio synthesis is discussed, as well as basics regarding the conversion methods needed for analog synthesizers. These basics provide a sufficient enough understanding behind the components necessary for both the 3 oscillator, MIDI compatible synthesizer, as well as the PWM-based synthesizer.

## 3.1 Digital - Analog Converter

The primary goal of a digital to analog converter is to convert data bits (strictly HIGH or LOW), and convert to an analog signal with voltage varying on the values of the data bits. This can be done either by receiving all data pins at once, or sending the data pins individually along with a clock bit, which is then read by a segment in the integrated circuit that decodes the data. Receiving all data pins at once does not require the use of an integrated circuit, and can be done using resistor ladders [1]. More details regarding resistor ladders will be shown in Section 4.2. Nonetheless, the DAC allows us to input a certain number of bits, in the case of this project, 10 bits, and convert to an analog signal (thereby giving us 1024 different values from 0 volts up until the maximum voltage). This is essentially "receiving what output voltage to emit in binary", so that 1111111111, all pins on, corresponds to the maximum voltage, and 0000000000 (all pins off) correspond to minimum voltage, and something like 1101100110 till be $\frac{870}{1023}$ of the maximum.

## 3.2 Analog - Digital Converter

An analog to digital works in the exact opposite way as a digital to analog converter. By taking in an input voltage (as a proportion of the maximum voltage), an analog to digital converter will set some data value to be equal to that input voltage as a binary proportion to the maximum (or reference) voltage. In the case of the microprocessor, this reference will be 3.3 volts. The usage of this in the project is to be able to perform certain actions based on the value of this input pin, i.e. control the frequency based on what notes are being pressed.

## 3.3 Low Pass Filter

The theory behind an analog low pass filter is relatively simple; the signal is sent through an RC circuit (see Figure 1), which, due to the charging and discharging properties of a capacitor, allows us to block frequencies past a certain cutoff frequency. This cutoff frequency is given by the following formula:

$$f_{cutoff} = \frac{1}{2\pi RC},$$

where $R$ is the resistance of the resistor in the circuit, and $C$ is the capacitance of the capacitor.

So, in order to reduce unnecessary signal noise, we can apply a low pass filter to block frequencies past the human range of hearing, as well as attenuate higher pitched distortion coming from the circuit.

In the specific case of this project, a 9.55 nF capacitor was made available. Using the formula mentioned above, we can set the cutoff frequency to 20 kHz, therefore using a resistor of 800 Ohms.

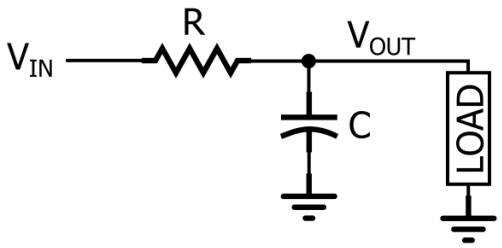


Figure 1: A basic circuit diagram of a low pass filter [2].

We can graph what the resultant signal graph will be post-LPF by using the following formula. This formula is derived from the formula for a voltage divider circuit [2].

$$V_{out} = V_{in}\left(\frac{X_C}{\sqrt{R_1^2 + X_C^2}}\right), \quad X_C = \frac{1}{2\pi fC}$$

Hence, assuming $V_{in}$ is kept constant at 3.3 volts, we can view the output voltage as a

function of the frequency, with $C = 9.55 * 10^{-9}$ F, and $R_1 = 800\Omega$.

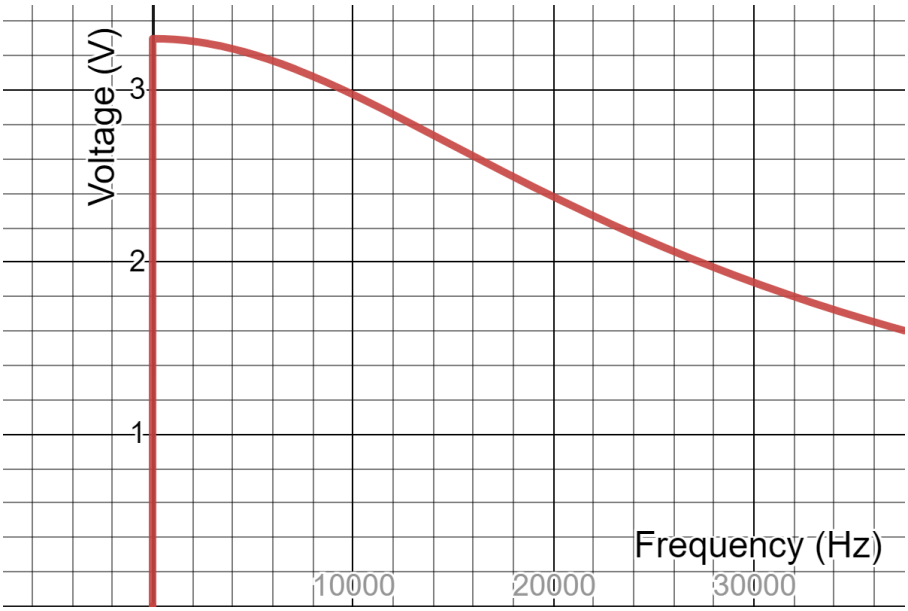


Figure 2: Plot of voltage versus frequency (linear scale) for an RC low pass filter.

## 3.4 MIDI Standard

Next, we delve further into details regarding the MIDI standard. Note that this is strictly regarding the standard, and details regarding receiving MIDI data on the microprocessor will be described in Section 4.4.

Sending MIDI data works by sending a stream of bytes at a given baud rate[2]. [3] This data is then decoded and processed, and instrument-related actions are taken accordingly.

MIDI data is sent in either 2 or 3 bytes of information, depending on what the first byte is. The first byte is always a command byte, as we see in Table 1. [4]

The latter half-byte in the command is what channel the command is being sent on (in this case, they are all being sent on channel 1. With specific regards to this project, MIDI commands were just being sent through Channel 1 for simplicity's sake.

[2]The rate at which information is transferred in a communication channel.

| | |
|---|---|
| 10000001 | note off |
| 10010001 | note on |
| 10100001 | aftertouch |
| 10110001 | continuous controller |
| 11000001 | patch change |
| 11010001 | channel pressure |
| 11100001 | pitch bend |
| 11110001 | non-musical commands |

Table 1: MIDI Command Bytes

If the command is a Note On, Note Off, or aftertouch, the first byte that is sent after the command is what note was pressed, and the second is the velocity value or aftertouch value, respectively. If the command is a controlled change or similar, the first byte that is sent after the command is what controller number was changed, and its new value. We are ignoring aftertouch, patch change, channel pressure, pitch bend, and non-musical commands in this project.

Hence, we can read the first byte, quickly decode it, and read the second byte and third byte accordingly. The C implementations are in Appendix B. When reading a note on / off command, the second byte corresponds to what note is being played. There is a direct conversion from the MIDI note to the frequency, through the following formula:

$$f_{note} = 440 \times \frac{2^{(n-69)/12}}{1},$$

where $n$ is the MIDI note number. This frequency is then converted into the period of the note, and then converted into microseconds. Because this formula is expensive to calculate, we used a lookup table to store the values of the frequencies. This lookup table is in Appendix A.

## 3.5 PWM-Based Synthesis

PWM-based synthesis generates a square waveform from a given frequency, and a given amplitude. The waveform generator is built into the microprocessor, and is controlled by the period and duty cycle of the waveform.

The theory behind this is relatively straightforward. Below is a diagram of the PWM waveform.

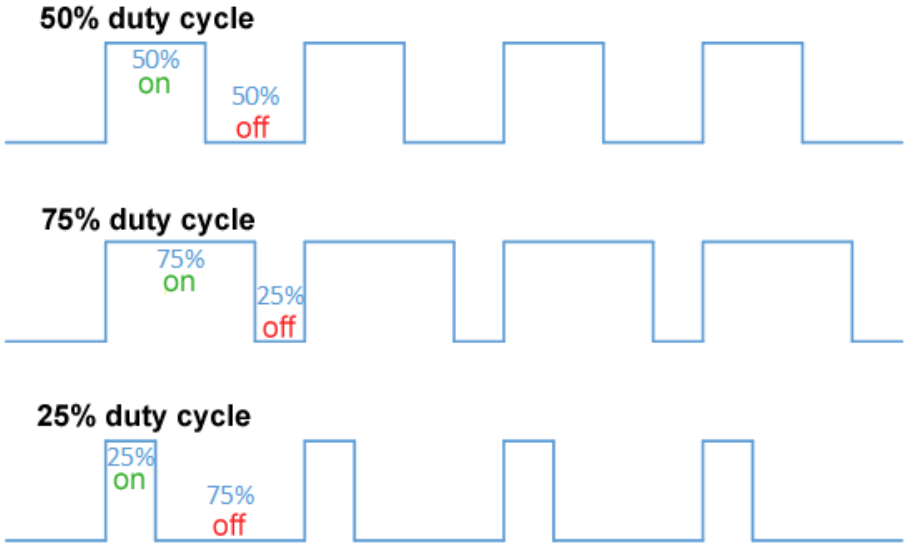


Figure 3: PWM waveform [5]

For this project, we used the PWM generator to generate a square waveform which was then outputted to an AUX output.

The PWM was used once it was determined that multi-oscillation was too computationally expensive (more discussed in Section 4.1).

## 3.6 ADSR Envelope Generation

Modern synthesizers do not just have a note ON / OFF transition when playing. An envelope is what allows one oscillator to be played like a pluck, to a pad, to numerous other types of sounds. An ADSR envelope is controlled by 4 crucial parts:

1. Attack phase. The attack phase controls how long it takes from the initial note press to when the amplitude is at its maximum. When the attack is short (like we'd want on a pluck), the maximum amplitude is reached very quickly, whereas when the attack is long (like we might want on a pad or string-type instrument), the maximum amplitude is reached much more slowly.

2. Decay Phase. Once the note reaches its maximum amplitude from the attack stage and the note is still on, the decay phase controls the time it takes for the note to decay from the attack phase to the sustain level (explained below). For a pluck-like instrument, we will want this decay time to be relatively short, for example, as this shapes the sound to only be heard for a short amount of time regardless of the time that the note is played for.

3. Sustain Phase. So long as the note is still on, this is the amplitude of the output (as a fraction of the input amplitude, so a note with a lower velocity will have a sustained amplitude of the same fraction, but not the same volume, as a note with a higher velocity).

4. Release Phase. As soon as the note is released (regardless of what phase it is in), the envelope will go into the Release phase, where the rate at which the output level decays from its initial volume before the release phase to its final level (0 volume, or -$\infty$ dB) is controlled. For a slow release, the release would be long (likely useful for pads or other slow-decaying instruments).

Due to debugging other issues, the envelope generation for this project did not work as anticipated (and was a low priority issue, compared to getting the waveform generators to work). Nonetheless, the code for the envelope generator in progress is shared in Appendix C.

# 4 Apparatus

## 4.1 PWM Based Synthesis Apparatus

The actual PWM was generated on the MSP430G2553 microcontroller. There was no need to send the signal to a DAC, as the PWM signal was directly sent to the low pass filter. As was discussed in Section 3.3, the low pass filter was used to remove any unintended high frequency noise caused by the synthesizer and also to smooth out DAC volume changes. The low pass filter also allows the note OFF implementation to work: When the voltage inputted into the

microprocessor was at its "default" value (both the default in the if statements in the code as well as when the voltage is at the level when no buttons are pressed), the period of the PWM was set to 2, and hence the duty cycle set to be 1. This frequency is high - 500000 Hz to be exact - and so to avoid unnecessary output interference and block short-term voltage fluctuations, this is blocked by the low pass filter. The LPF is also necessary as the MSP430 has a 32-kHz watch crystal - meaning the Nyquist frequency is 16 kHz. It may have actually been a good idea to increase the resistance such that the cutoff frequency on the LPF was below the 16 kHz mark to further avoid aliasing, but nonetheless frequencies at and above this range are still somewhat attenuated with the cutoff frequency being set to 20 kHz as we saw in Figure 2.

This LPF was then connected to an AUX output. This AUX output was then connected to a speaker, however the goal of having an AUX output was to be able to connect to anything – which is indeed the case. A speaker was used for demonstration purposes, but for measurement purposes as we see in Section 6, this AUX output was connected to an audio interface to be able to be measured. This allowed us to analyze the signal on a computer, instead of having to use a microphone to record the signal which would lead to a less accurate representation of the signal. The actual circuit diagram for the output is very simple (as it's just a low pass filter and a speaker), and will be included in Section 4.2.

## 4.2 Apparatus Details

The final rendition of this project is relatively simple in terms of electronics. We begin by sending a signal of the same voltage to 10 different buttons, which act as the user control for the frequency of the wave emitted from the PWM. The project is intended to be used similarly to a piano, with the 10 buttons available corresponding to various keys on a piano, ranging from A4 to C6, excluding sharps and flats in the note range. When these buttons are pressed, the corresponding amplitude of the input signal is changed by sending the bits to a 10 bit DAC, which was done using a resistor ladder based on the R-2R architecture, where the resultant signal is read through a 10-bit ADC (thus allowing for the same level of precision

for the DAC as is available for the ADC) that is built into the MSP430G2553 microprocessor. When the ADC decodes the input, the period of the PWM signal accordingly by modifying a global variable. The duty cycle is then set to always be 50%, by bit shifting the value of the global variable to the right by 1 bit. Finally, the signal is outputted, and heard by the user.

The code for this implementation is in Appendix D, and the circuit diagram is shown in Figure 4. It was discovered that a USB power supply to the switches was not necessary, and the whole project could be powered by connecting the ground of the MSP430 to the switches itself as we see in the diagram. While this *works*, this method may also be the rationale behind the unexpected voltage fluctuation (hence frequency fluctuation) observed in the output. It would have been a smart idea to separate the grounds and power the inputs separately.

## 4.3 How the project is used

The use of the project is relatively simple: The user plugs in the MSP430 to a power source, and plugs in headphones or a 3.5mm TRS cable to the output port. Once the synthesizer is powered on, the user plays the keys like a piano, where A4 is the leftmost key (assuming orientation of synthesizer is such that the headphone jack is facing the user, and the low pass filter is to the right of the buttons), and C6 us the uppermost key.

## 4.4 Previous Apparatus Renditions

As the initial intention of this project was to include MIDI compatibility, it is important to also include circuit diagrams for this, given it offered different methods of user interaction. The rendition of the project that allowed for MIDI input was the 3 oscillator version (which, as will be discussed in the Results section, did not work properly due to hardware limitations). Thus, the apparatus diagram below will reflect the legacy oscillators. The MIDI compatible circuit diagram is shown in Figure 5.

Note that the sending of the output through the 8 separate pins instead of the single output pin is due to sending the direct digital signal to an 8 bit DAC, as using just a PWM does not

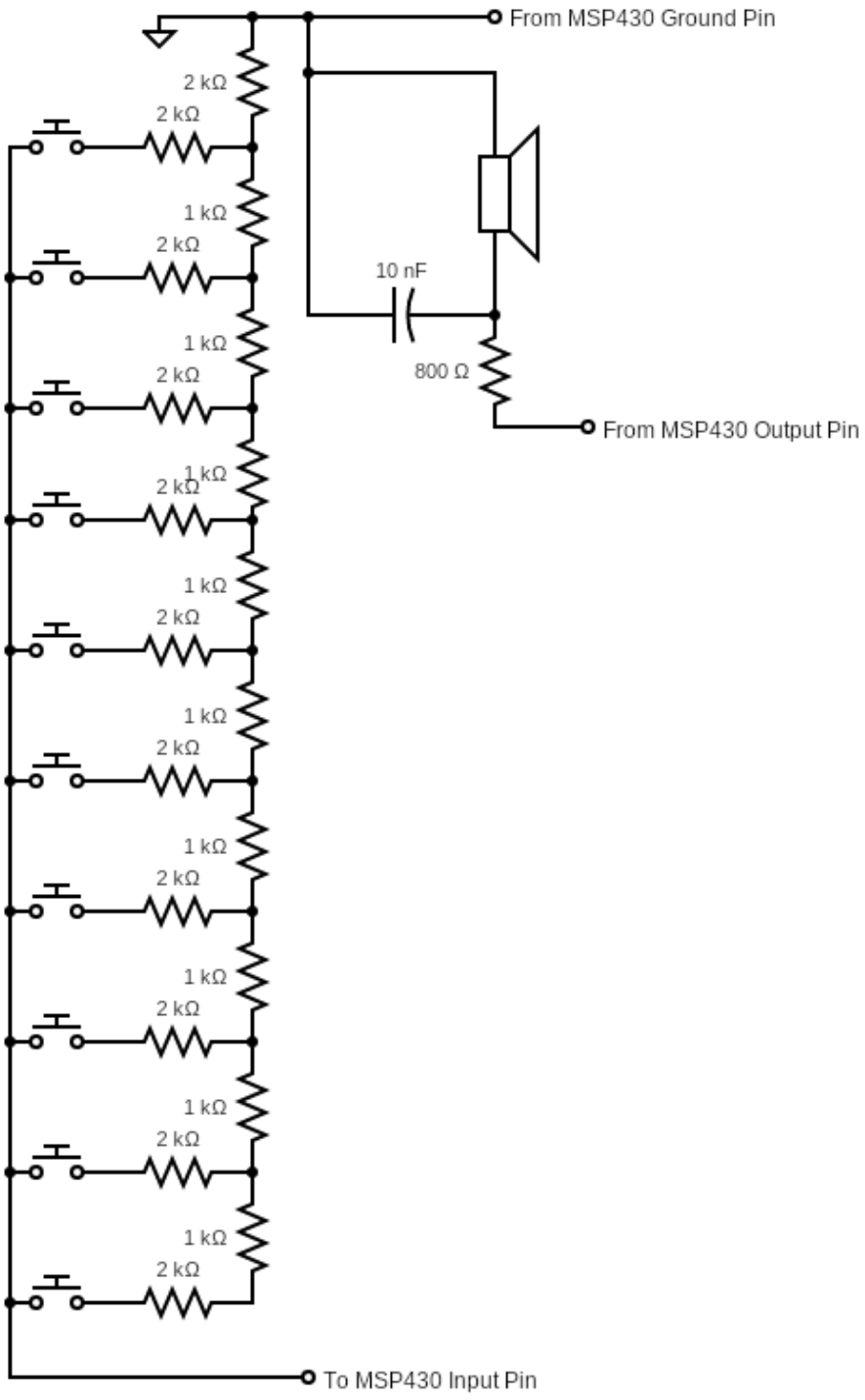


Figure 4: Circuit diagram for PWM-based synthesizer. Self-powered through the MSP430. Note that the speaker is just denoting the existence of an audio output, and is actually the AUX output and not necessarily an actual speaker.

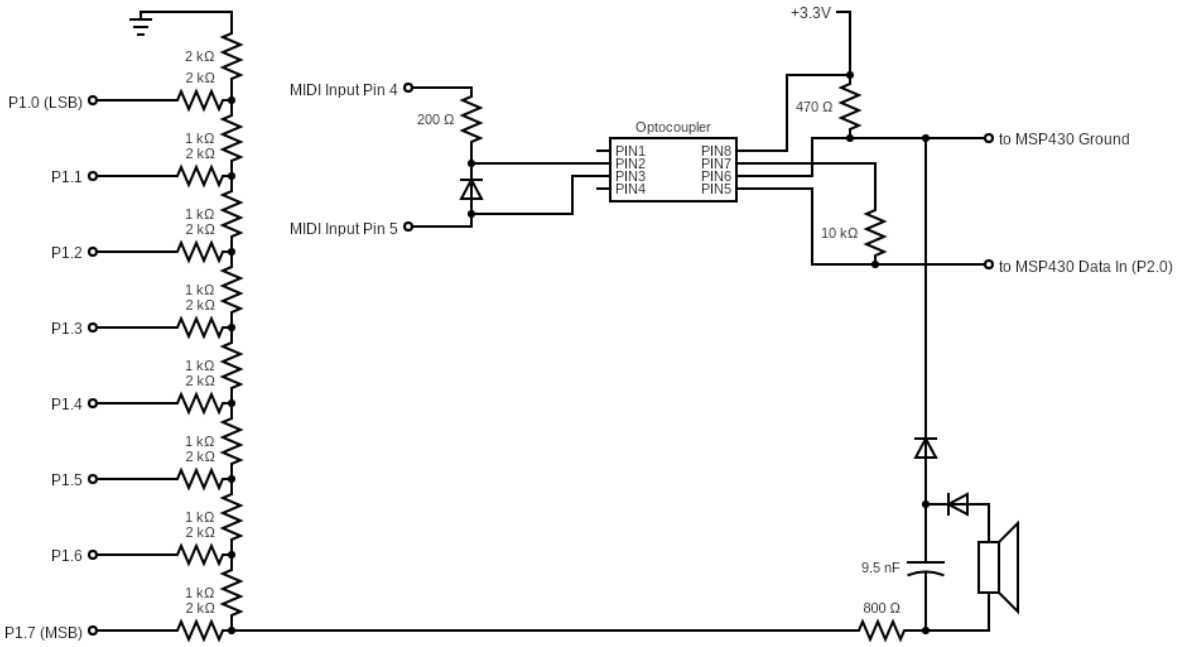


Figure 5: Circuit diagram for the MIDI compatible, 3 oscillator version. Note that the speaker is just denoting the existence of an audio output, and is actually the AUX output and not necessarily an actual speaker.

require the use of a DAC (unlike adding the 3 waveforms together). Adding the 3 waveforms together would certainly be *possible* using a PWM, but would be jsut as computationally expensive. The reason that this circuit diagram is still being shown and discussed is because it was the original intention of the project, and is significantly more complicated, thorough, and playable than the PWM version - if it worked (limitations and reasons why are discussed in Section 5. The MIDI was being sent by an Edirol UA 25-EX Audio Interface, controlled by Presonus Studio One.

# 5 Results

Unfortunately, the 3 oscillator version did not work properly due to software limitations. When iterating through the different waveforms, the period set in the for loop was wildly inaccurate. This was due to largely hardware-based fluctuations, as I did not use a timer to ensure accuracy of the period being set. Hence, the actual period was based on the number of CPU cycles that

were executed, which changed based on the period due to numerous factors. Some of these factors include the use of multiplication, division, and modulo, which were not available in the MSP430G2553 microprocessor on a hardware level (no hardware multiplier, for example) and thus were computationally expensive. Because there was this large computational overhead, not only was the waveform itself inaccurate, but the MIDI data was unable to be processed in a timely manner given it required quick turnaround time due to a fast baud rate of 31.25 k bits per seconds. The MIDI also did not work due to accidental frying of the MIDI Out port on the Edirol UA 25-EX, where it was treated as a MIDI In port and was being sent 5 volts when it should not have. This lead to spotty USB connection[3]. The 3 oscillator version was therefore only able to act as a noise generator, and was not able to be used as a semi-realistic synthesizer due to its inability to hold higher frequency synthesis. But, changing the period every iteration did lead to interesting results:

```
1
2  void main() {
3      // [setup code here, omitted for redundancy]
4      period = 2; // modifiable value
5      while(1) {
6          period += 1; // modifiable value
7          period %= 200; // modifiable value
8          // [synthesis code / for loop here, omitted for redundancy]
9      }
10 }
```

This lead to more "experimental" sounding synthesis, similar to what would be found in some Eurorack modules. Without MIDI support (as discussed directly above), however, and not enough ADCs available on the microprocessor for parameter changes, there would not have been nearly enough user interaction to make something that is enjoyable to use without going in and editing the C code, a task which is obviously not friendly to an end user, and inconvenient

[3]This may have instead been due to transportation of the audio interface to the lab and back.

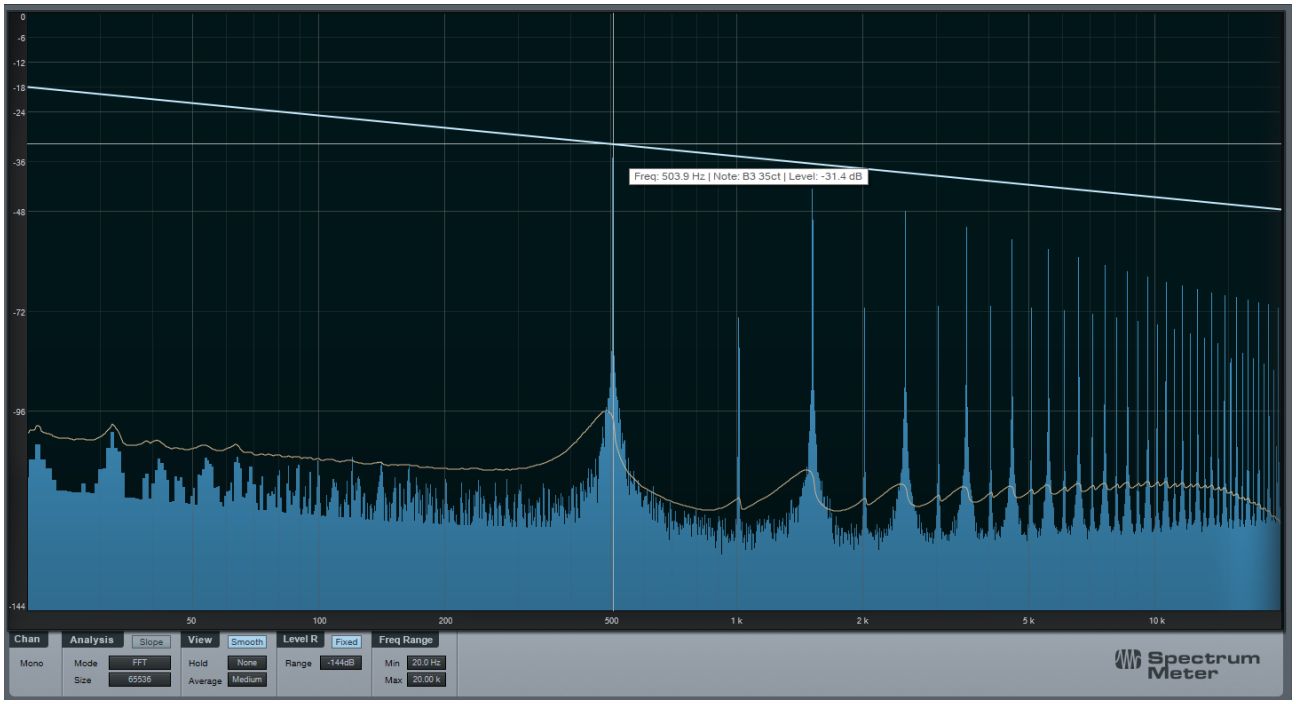


Figure 6: FFT transform of the PWM synthesizer, playing at A4.

at best.

Nonetheless, the PWM based, single oscillator version was much more accurate than the 3 oscillator version. The PWM based version was able to accurately reproduce a square wave with a given period, mainly due to the fact that the PWM is based on a timer, which is able to accurately set the period and is also not based on the number of CPU cycles.

We can observe the accuracy and fluctuation of the PWM waveform by connecting the output of our synthesizer to the input of an audio interface, which allows the input to be observed through signal analysis software. Presonus Studio One's Spectrum Analyzer was used to analyze the waveform.

Firstly, let's visualize A4 (at what should be 440 Hz), in Figure 6.

Given the output is a square wave, we can compare this to a square wave generated using a software tone generator (also built into Studio One), both with and without antialiasing (figure 7 and 8 respectively).

From what we can see here, the low pass filter did a good job at performing rudimentary

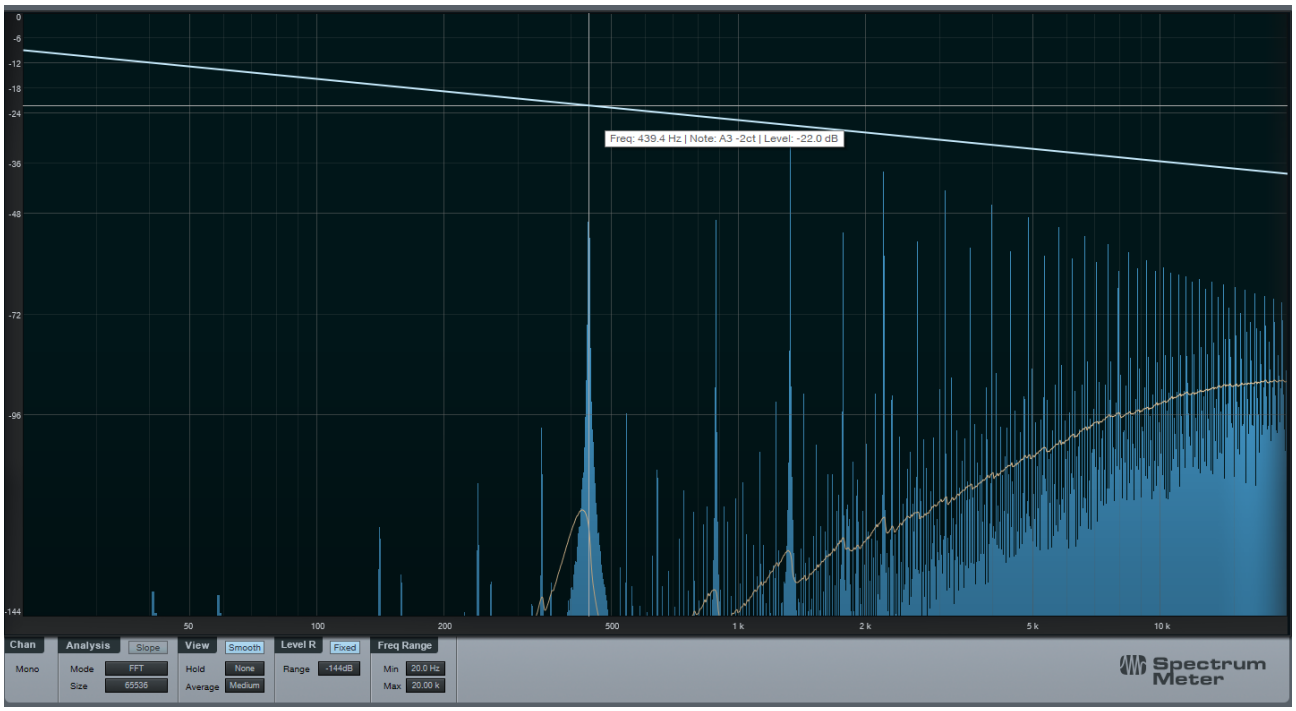


Figure 7: FFT transform of a square wave at 440 Hz with antialiasing.

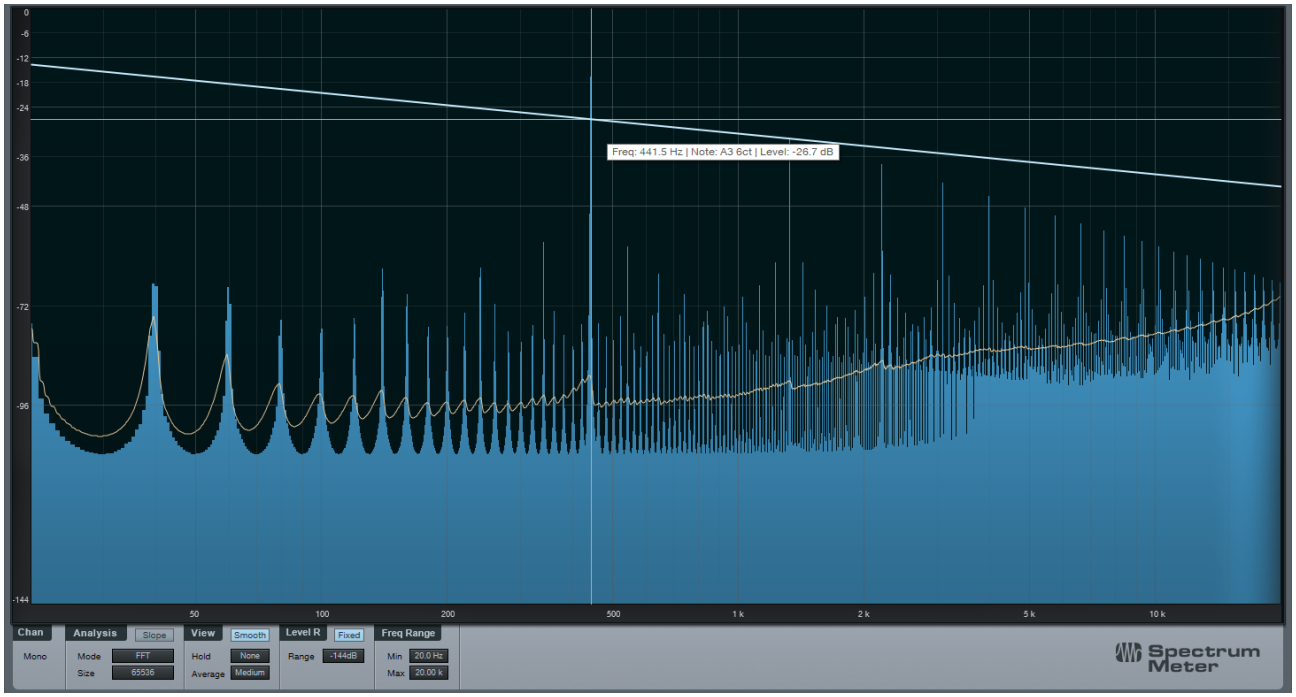


Figure 8: FFT transform of a square wave at 440 Hz with no antialiasing.

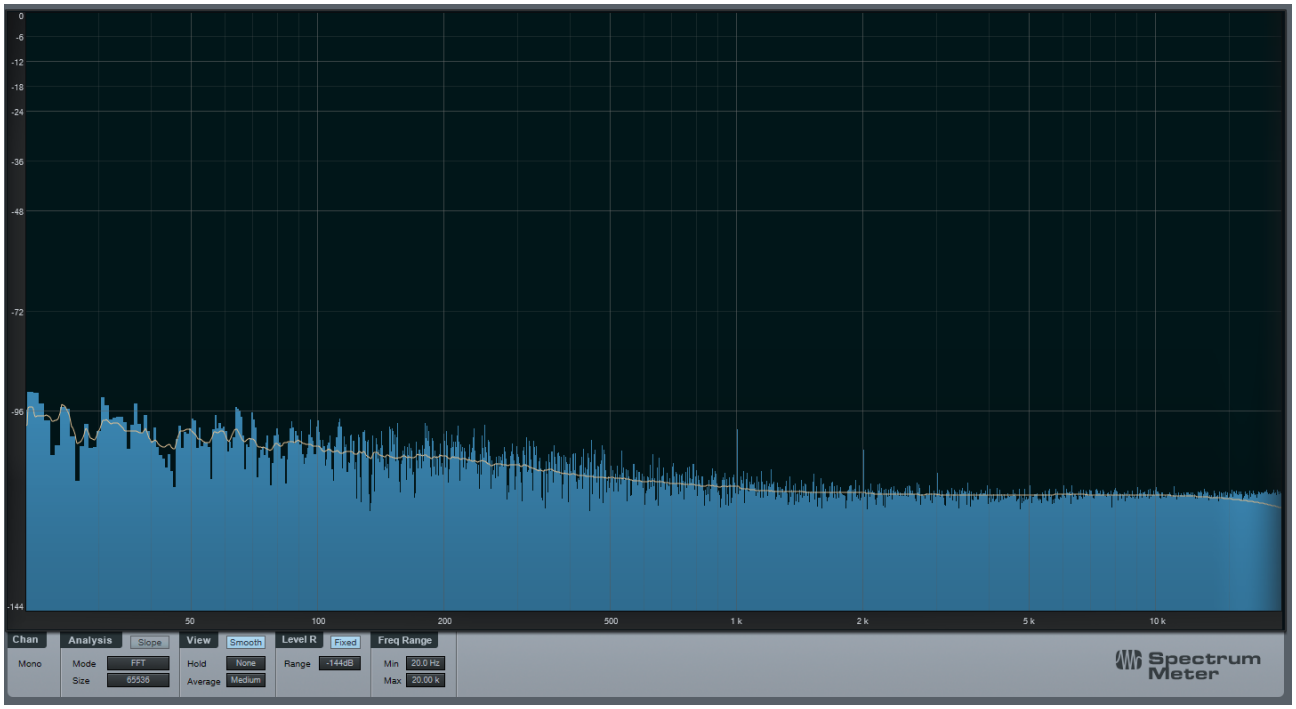


Figure 9: FFT transform of the noise floor coming from the PWM synthesizer.

antialiasing. However, the PWM period is actually off – Even though 440 Hz (2273 $\mu$s) was selected as the period, as we see in the for loop in Section D, the actual frequency is at approximately 504 Hz, which is closer to B rather than A.

Note that the signal to noise ratio is much higher on the software tone generator, so on the PWM square wave, there is a much more visible noise floor, but nonetheless the signal can still be analyzed accordingly. In fact, we can directly view the noise floor coming from the synthesizer in Figure 9.

The distortion of the PWM wave did change based on the note that was being played, as an inherent property of not using precision resistors and going through numerous conversions before turning into an audible output. A4, when played, did not have much distortion - this is juxtaposed with B4, which, as we will see in Figure 10, has a much more distorted signal and did not sound as pleasant when played. The [software] tone generated signal for B4 is very similar to that of A4, and will be omitted for space's sake.

We see that there are 2 conflicting notes that are being played (that are very close to each

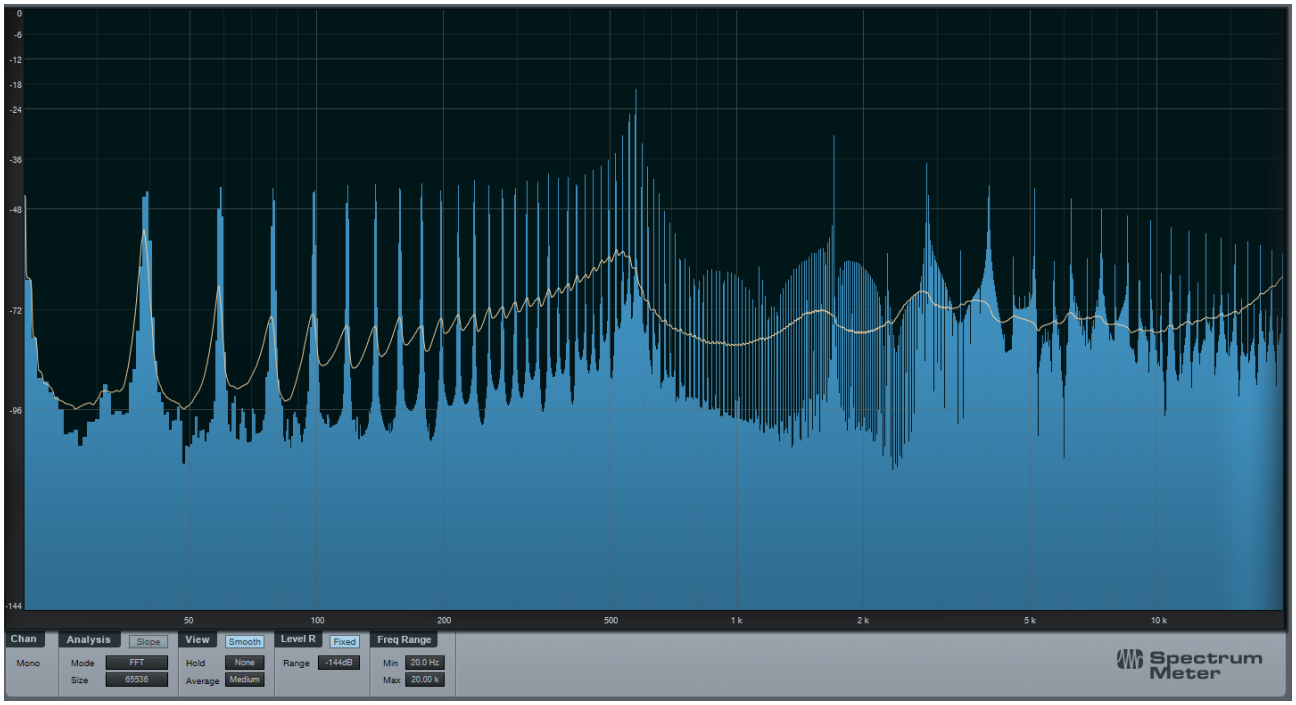


Figure 10: FFT transform of the PWM synthesizer, playing at B4.

other). B4 was not always a poorly synthesized note: Dependent on placement of the synthesizer and resistor placement, the voltage level fluctuated repeatedly between the different bounds for a given note, hence why we see the fluctuating voltage here. This is an inherent flaw in the buttons to DAC to ADC to period translation.

# 6 Discussion

It would have been a much better idea to use 3 PWM signals instead of 3 iterative waveforms, and modify the period and duty cycle accordingly to match the given waveform being played. This would have allowed for a much more accurate representation of the waveform being played, and would have allowed for the use of a timer to accurately set the duty cycle. Of course, this may have had its own limitations as well, as we would have to be accurately changing the duty cycle of the PWM signal, and this would have required another timer to be used, along with numerous interrupts (in order to use the PWM as a non-square wave). This may have been a better idea, but still is at risk of being too computationally expensive due to the

constant changing of the duty cycle with a semi-unknown period (due to the fact that we have so many notes that could be played). Early analog synthesizers combated this problem, and simultaneously allowed for polyphony, by using a separate waveform generator for each note. But alas, this is too expensive for our microprocessor, and the 3 oscillator version was not able to be used as a real-time synthesizer, and the PWM based version was not "interesting" enough for real-world use (albeit was much more accurate and actually playable).

The primary source of inaccuracy for the PWM based version was the DAC being used to convert the notes being pressed to an analog signal. Precision resistors were not used, and so a lot of fluctuation was observed in the analog input signal. This is especially observed in Figure 10. This could have also been mitigated somewhat by using an integrated circuit instead of a resistor ladder DAC, as the resistor ladder was very reactive to external conditions. During the in-class demonstration, for example, B4 was just fine in its frequency consistency, whereas D5 was hugely fluctuating and distorted, which interfered somewhat with the demonstration.

A possible improvement that could be made is to use this module - either the 3 oscillator version or the PWM version - as not a synthesizer, but instead an LFO (low frequency oscillator), which could have been used as a module in another synthesizer application, or even as a Eurorack module. Other improvements would be to use precision resistors to ensure a more consistent frequency input, or an integrated circuit DAC, as discussed above. It would also lead to a more interesting output if the output was split, with one split being smoothed using a capacitor, and the other as-is, and mixing the two together using a variable resistor (thus having two somewhat different waveforms to play with). And of course, having proper MIDI compatibility - perhaps by doing the decoding on another microprocessor - would have made the synthesizer more playable.

# 7 Conclusions

The most important thing that was learned is that the MSP430 is not powerful enough to be used for accurate audio reproduction. Hardware limitations were often overlooked in the design

of the synthesizer, leading to numerous issues and inaccuracies as described in Sections 5 and 6. This project was certainly a learning experience, and being aware of such limitations will make a future design of a synthesizer a more successful endeavour. As such, the project was certainly worth constructing: the design of this project lead to more knowledge in decoding MIDI data, envelope and waveform generation, and circuit design. The 3 oscillator synthesizer still has use as an LFO, and with some tuning of the output frequency, could be used as such so long as MIDI data is decoded elsewhere.

# References

[1] D. L. Terrell, "Chapter eight - digital-to-analog and analog-to-digital conversion," in *Op Amps (Second Edition)* (D. L. Terrell, ed.), pp. 337–359, Burlington: Newnes, second edition ed., 1996.

[2] R. Keim, "What is a low pass filter? a tutorial on the basics of passive rc filters - technical articles." `https://www.allaboutcircuits.com/technical-articles/low-pass-filter-tutorial-basics-passive-RC-filter`, May 2019.

[3] N. Phongchit, "What is baud rate & why is it important?." `https://www.setra.com/blog/what-is-baud-rate-and-what-cable-length-is-required-1`, Jan 2016.

[4] A. Ghassaei and Instructables, "Send and receive midi with arduino." `https://www.instructables.com/Send-and-Receive-MIDI-with-Arduino/#discuss`, Oct 2017.

[5] Thewrightstuff, "Examples of 50%, 75%, and 25% duty cycles." `https://en.wikipedia.org/wiki/Pulse-width_modulation#/media/File:Duty_Cycle_Examples.png`, Sept. 2018.

# A Appendix: Table of Notes to Frequency Values

| Note | Frequency (Hz) | Period ($\mu$s) |
|---|---|---|
| A4 | 440.00 | 2273 |
| B4 | 493.89 | 2025 |
| C5 | 523.25 | 1911 |
| D5 | 587.33 | 1703 |
| E5 | 659.25 | 1517 |
| F5 | 698.46 | 1432 |
| G5 | 783.99 | 1276 |
| A5 | 880.00 | 1136 |
| B5 | 987.77 | 1012 |
| C6 | 1046.50 | 956 |

Table 2: Table of Notes to Frequency Values

# B Appendix: C Code for Decoding MIDI Data

Please note that functions regarding the actual waveform generation were ignored here for space and organization purposes.

```
1  //
2  // Created by Ashtan Mistal on 2022-03-24.
3  //
4
5  #include "msp430.h"
6  #include "msp430g2553.h"
7
8  unsigned long frequency;
9  unsigned long amplitude;
10 unsigned long noteon;
```

```
11 unsigned long square_wave_volume;
12 unsigned long noise_wave_volume;
13 unsigned long triangle_wave_volume;
14 unsigned long sawtooth_wave_volume;
15 unsigned long attack_rate;
16 unsigned long decay_rate;
17 unsigned long sustain_level;
18 unsigned long release_rate;
19
20 void send_output(int input) {
21     P1OUT = input;
22 }
23
24 // take in MIDI note and convert to frequency
25 // midi note is a number between 0 and 127
26 int midi_to_frequency(int midi_note) {
27     // this is a slow function because it declares a variable every time it is
           called
28     // will be better to declare a global variable and use that instead
29     int midi_to_frequency_table[128] = {
30         8, 8, 9, 9, 10, 10, 11, 12, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 23, 24,
              25, 27, 29, 30, 32, 34, 36, 38, 41, 43, 46, 48, 51, 55, 58, 61, 65, 69,
              73, 77, 82, 87, 92, 97, 103, 110, 116, 123, 130, 138, 146, 155, 164,
              174, 184, 195, 207, 220, 233, 246, 261, 277, 293, 311, 329, 349, 369,
              391, 415, 440, 466, 493, 523, 554, 587, 622, 659, 698, 739, 783, 830,
              880, 932, 987, 1046, 1108, 1174, 1244, 1318, 1396, 1479, 1567, 1661,
              1760, 1864, 1975, 2093, 2217, 2349, 2489, 2637, 2793, 2959, 3135, 3322,
              3520, 3729, 3951, 4186, 4434, 4698, 4978, 5274, 5587, 5919, 6271, 6644,
              7040, 7458, 7902, 8372, 8869, 9397, 9956, 10548, 11175, 11840, 12544};
```

```
31      return midi_to_frequency_table[midi_note];
32  }
33
34
35  // midi data is sent in as serial peripheral data on port P2.0
36  // convert the serial port input to an 8-bit integer
37  // this is the MIDI data
38  int serial_to_midi() {
39      int i;
40      int midi_data = 0;
41      for (i = 0; i < 8; i++) {
42          midi_data |= (P2IN & 0x01) << i; // assuming P2.0 is a 1 or 0
43          // may have to delay cycles here based on baud rate -- or may have to remove
                for loop if baud rate cannot be changed and number of cycles is too high
44      }
45      return midi_data;
46  }
47
48
49  // decode the midi data to determine if it is a note on or a continuous controller
        change
50  // return if it is a note on or a continuous controller change
51  // if it is a note on, return 1
52  // if it is a continuous controller change, return 2
53  // else return -1
54  int decode_midi_data(int midi_data) {
55      if ((midi_data & 0xF0) == 0x90) {
56          return 1;
57      } else if ((midi_data & 0xF0) == 0xB0) {
```

```
58          return 2;
59      } else if ((midi_data & 0xF0) == 0x80) { // note off
60          return 3;
61      } else {
62          return -1;
63      }
64  }
65
66  void handle_note_on(int midi_note, int velocity) {
67      frequency = midi_to_frequency(midi_note);
68      amplitude = velocity;
69  }
70
71  void handle_note_off() {
72      noteon = 0;
73      amplitude = 0; // set amplitude to 0 to turn off the note; remove this if we
            are using ADSR
74      // amplitude will be handled by the release phase UNLESS we are removing the
            ADSR envelope
75
76  }
77
78  void handle_continuous_controller_change(int controller_number, int
        controller_value) {
79  // case numbers are arbitrary and based on what input MIDI controller we're using
80      switch(controller_number) {
81          case 1:
82              square_wave_volume = controller_value;
83              break;
```

```
84          case 2:
85              noise_wave_volume = controller_value;
86              break;
87          case 3:
88              triangle_wave_volume = controller_value;
89              break;
90          case 4:
91              sawtooth_wave_volume = controller_value;
92              break;
93          case 5:
94              attack_rate = controller_value;
95              break;
96          case 6:
97              decay_rate = controller_value;
98              break;
99          case 7:
100             sustain_level = controller_value;
101             break;
102         case 8:
103             release_rate = controller_value;
104             break;
105         default:
106             break;
107     }
108 }
109
110
111 /*
112  * Steps to decode MIDI data:
```

```
113  * 1. Read the first byte
114  * 2. If the first byte is a note on, read the second and third byte
115  * 3. If the first byte is a continuous controller change, read the second and
         third byte
116  * 4. If the first byte is neither, return -1 and ignore the rest of the bytes
117  */
118
119 // this is the interrupt service routine
120 void decoder() {
121     int midi_data = serial_to_midi();
122     int midi_data_type = decode_midi_data(midi_data);
123     if (midi_data_type == 1) {
124         int midi_note = serial_to_midi();
125         int velocity = serial_to_midi();
126         handle_note_on(midi_note, velocity);
127     } else if (midi_data_type == 2) {
128         int controller_number = serial_to_midi();
129         int controller_value = serial_to_midi();
130         handle_continuous_controller_change(controller_number, controller_value);
131     } else if (midi_data_type == 3) {
132         int midi_note = serial_to_midi();
133         int velocity = serial_to_midi();
134         handle_note_off();
135     }
136 }
137
138
139
140
```

```
141 // enable an interrupt service routine based on the midi clock pin
142 void enable_interrupt() {
143     P2IE |= 0x01; // enable interrupt on P2.0
144     P2IES |= 0x01; // set interrupt to trigger on falling edge
145     P2IFG &= ~0x01; // clear interrupt flag
146     P2IE |= 0x01; // enable interrupt on P2.0
147     _BIS_SR(GIE); // enable global interrupts
148 }
149
150 void __attribute__((interrupt(PORT2_VECTOR))) PORT2_ISR(void) {
151     decoder(); // run ISR
152     P2IFG &= ~0x01; // clear interrupt flag
153 }
154
155 int main(void) {
156     P1DIR = 0b11111111; // set all pins on P1 to output
157     P1OUT &= 0b00000000; // set all pins on P1 to 0
158     // set up P2 input pins
159     P2DIR = 0b00000000; // set all pins on P2 to input
160     P2REN |= 0b11111110; // enable pull-up resistors on all pins on P2 except P2.0
161
162     //// NOTE TO SELF that there is not in fact a clock pin on the MIDI cable.
163
164     // initialize variables to some default values
165     frequency = 2;
166     amplitude = 255;
167     noteon = 1; // can change to 0 if we do not want our synthesizer to be making
        noise when we turn it on
168     square_wave_volume = 255;
```

```
169     noise_wave_volume = 0;
170     triangle_wave_volume = 255;
171     sawtooth_wave_volume = 255;
172     attack_rate = 2;
173     decay_rate = 40;
174     sustain_level = 127;
175     release_rate = 127;
176
177
178
179     // set up P2 interrupt
180     enable_interrupt();
181     unsigned long period = 1;
182
183     while (1) {
184         period = 1000000 / frequency; // period converted to microseconds
185         unsigned long i;
186         for (i = 0; i < period; i++) {
187             unsigned long output = square_wave_volume *
                    square_wave_iteration(period, i);
188             output += noise_wave_volume * noise_wave_iteration(period, i)/4096;
189             output += triangle_wave_volume * triangle_wave_iteration(period,
                    i)/period;
190             output += sawtooth_wave_volume * sawtooth_wave_iteration(period,
                    i)/period;
191             output = output / 4;
192             output = output * amplitude / 128;
193 //            output = ADSR(output, timestep_counter);
194 //            serial_output(output);
```

```
            send_output((int) output);
        }
}
```

# C Appendix: ADSR Envelope Code

```
int ADSR(int output, int timestep) {
    if (noteon) {
        if (timestep < attack_rate) {
            // attack phase
            output = timestep * amplitude / attack_rate;
        } else if (timestep < decay_rate + attack_rate) {
            // decay phase
            output = amplitude - (timestep - attack_rate) * amplitude / decay_rate;
        } else output = sustain_level; // sustain phase
    } else {
        // release phase
        if (timestep < release_rate + decay_rate + attack_rate) {
            output = max(output - (timestep - attack_rate - decay_rate) * output /
                release_rate, 0); // this might be wrong
        } else output = 0;
    }
    return output;
}
```

# D Appendix: PWM Synthesizer Code

```
#include "msp430.h"
// translate an input voltage to period of the corresponding note
```

```
3  // notes are from A4 to C6
4  int translate_voltage_to_period() {
5      if (ADC10MEM > 0x03EF && ADC10MEM < 0x03F3) { // 0x03F1
6          return 2;
7
8      } else if (ADC10MEM > 0x0003 && ADC10MEM < 0x000F) { // 0x0005
9          // A4
10         // good now i think
11         return 2273;
12     } else if (ADC10MEM > 0x0120 && ADC10MEM < 0x0140) { // 0x012F
13         // B4
14         // good now
15         return 2025;
16     } else if (ADC10MEM > 0x01F0 && ADC10MEM < 0x0220) { // 0x0208
17         // C5
18         // not quite right but low priority
19         return 1911;
20     } else if (ADC10MEM > 0x0250 && ADC10MEM < 0x029E) { //0x0284
21         // D5
22         // sounds wack
23         return 1703;
24     } else if (ADC10MEM > 0x02B0 && ADC10MEM < 0x02DA) { // 0x02CF
25         // E5
26         // good, may need bound adjusting
27         return 1517;
28     } else if (ADC10MEM > 0x02E0 && ADC10MEM < 0x030F) { // 0x02FD
29         // F5
30         // good
31         return 1432;
```

```
32      } else if (ADC10MEM > 0x0310 && ADC10MEM < 0x0328) { // 0x0323
33          // G5
34          // good
35          return 1276;
36      } else if (ADC10MEM > 0x0328 && ADC10MEM < 0x0341) { // 0x0335
37          // A5
38          // good
39          return 1136;
40      } else if (ADC10MEM > 0x0341 && ADC10MEM < 0x0353) { // 0x034c
41          // B5
42          // really fuzzy
43          return 1012;
44      } else if (ADC10MEM > 0x0353 && ADC10MEM < 0x036D) {// 0x035F
45          // C6
46          // this is mostly good
47          return 956;
48      } else {
49          return 2; // I don't think the period can be zero, so this should do
50      }
51  }
52
53  void main(void)
54  { WDTCTL = WDTPW + WDTHOLD; // Stop WDT
55      ADC10CTL0 = ADC10SHT_2 + ADC10ON; // ADC10ON
56      ADC10CTL1 = INCH_1; // input A1
57      ADC10AE0 |= 2; // PA.1 ADC option select
58      P1DIR |= BIT2;                // P1.2 to output
59      P1SEL |= BIT2;                // P1.2 to TA0.1
60
```

```
61      CCR0 = 4;                // PWM Period
62      CCTL1 = OUTMOD_7;         // CCR1 reset/set
63      CCR1 = 2;               // CCR1 PWM duty cycle
64      TACTL = TASSEL_2 + MC_1;    // SMCLK, up mode
65
66      for (;;) {
67          ADC10CTL0 |= ENC + ADC10SC; // Sampling and conversion start
68          while (ADC10CTL1 &ADC10BUSY); // ADC10BUSY?
69          CCR0 = translate_voltage_to_period();
70          CCR1 = CCR0 >> 1; // always using half the period for the duty cycle
71      }
72  }
```

# E Appendix: Initial Design Sketches

Attached below are the initial design sketches for the synthesizer, as well as the design sketch for the PWM based version. Note that the initial design sketch did not take into account the hardware limitations of the MSP430 only having 2 ADCs available, and incorporated the use of multiple inputs.

With regards to the PWM design, the diodes ended up not being needed due to no longer splitting the output, and the note on/off was handled through software instead of going through a phototransistor NOT gate (not explicitly shown).

The sketches are included as reference only, and hence are not computer-generated. Necessary details regarding the circuit diagrams are in the corresponding figures in Section 4.

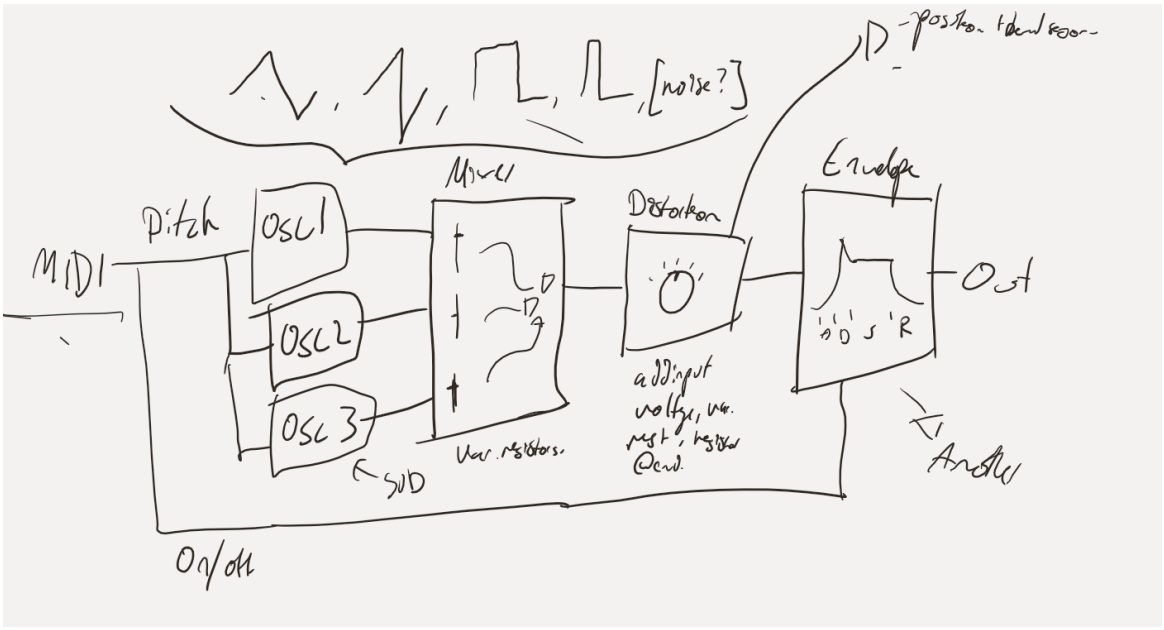


Figure 11: Initial design sketch for the 3 oscillator synthesizer. Layout of synthesizer inspired by the Moog Minimoog.

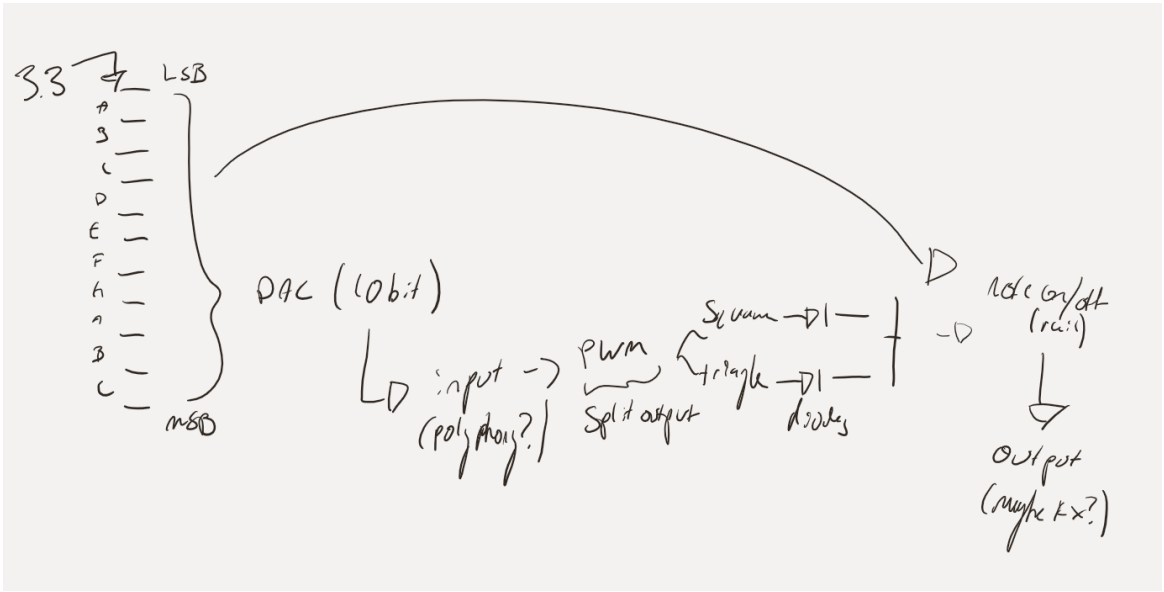


Figure 12: Initial design sketch for the PWM based synthesizer, with the 10 buttons being used as an input.