Multi-tool Music Synthesizer

Alex Kim

5/11/2024

1 Introduction

For my Physics 245 final project, inspired by the suggested metronome idea, I sought to develop a device that encompasses my passion for music. When we began learning about filters, amplifiers, and digital analog conversion, I was curious about how principles of electronics connect to the audio production tools I learned about in Music 270. Thus, my proposed project was to build a music synthesizer with multiple instruments of different qualities capable of playing the spectrum of notes on a standard 88 key piano. The synthesizer would use hardware on the digital designer to form a drum pad, switch between instruments, turn the speaker on and off and control volume. This project combined physical components such as Op-Amps and FETs with an Adafruit Feather arduino, and a CircuitPython program to build the final product.

2 Circuit Components and Design

In this section I will touch upon the circuit schematic (Figure 1, Appendix 5.1), and what each component contributes to the overall project, and why I chose to use them specifically. First, note that the digital input box has been labeled "Serial Port" to represent inputs and outputs being sent and received between the Feather and the computer via UART connection. The serial port pins have been labeled as I used them in my physical circuit with the corresponding RX and TX pins that send and receive data. I opted for the computer keyboard and serial port over just physical switches because this approach provided access beyond the basic 8-bit logic switches available on the digital designer. A single octave chromatic scale has 12 notes and additionally, I needed two other keys to control the octave, and clear the notes being played. Moreover, I chose the specific letter keys to somewhat resemble where the black and white keys are on a typical piano instead of just putting each key in a line. The rest of the physical inputs were built using the logic switches and push buttons on the digital designer. In Figure 1, S5, S1, and S2 refer to the logic switches while S3,S4 represent the push buttons for the "drum pad".

2.1 S5 (synthesizer wave type switch)

S5 was the most straightforward switch, connected to the A5 digitalio pin on the Feather, which corresponds to the digitalio input variable *wave_type*. A 2kΩ resistor in between S5 and the A5 pin so when the switch was turned on high, the Feather would not

crash due to an overload of current. The result was a switch whose value would be checked in the *play_chord*() function to determine if the square or triangle wave instrument would be used.

2.2 S1 (speaker on-off switch)

S1 functions similarly. When combined with an n-type MOSFET (M1), effectively turns the speaker on and off. In Figure 1, the output of S1 is connected to the gate of the MOSFET, while the source of the MOSFET is connected directly to the output of the Op-Amp. The drain of the MOSFET is connected to the input pin of the speaker, and the ground pin of the speaker is connected to the common ground of the system. When S1 activates, the gate voltage input allows the MOSFET to conduct, establishing a path from the Op-Amp through the MOSFET to the speaker. This configuration enables the Op-Amp to drive the speaker by modulating the voltage at the source of the MOSFET, which in turn controls the current flow through the speaker. When the gate voltage is below the threshold, the MOSFET turns off, interrupting this current flow and effectively silencing the speaker. This arrangement ensures that the speaker's operation is precisely controlled by the voltage signal from the Op-Amp, using the MOSFET as a power-controlling switch.

2.3 S2 (instrument switch), S3, and S4 (drum inputs)

Unlike the other two logic switches, S2 splits its output between the n-type MOSFET M2 and digitalio pin D10 on the Feather. Referring to the code appendix, pin D10 corresponds to the digitalio input object *switch_instrument* which is then checked by the function *check_instrument*() to return a boolean value. If *check_instrument*() sees that *switch_instrument* is high, it returns True back to the main while loop which then calls the *drums*() function to switch from the keyboard instrument to the drum pad instrument. However, the drum pad instrument is driven by PWM rather than DAC values since A0 was the only functioning analogio port on the Feather, so to trouble shoot this S2 was tied to the gate of the M2 MOSFET whose source came from the PWM output pin D11. Similarly to the M1 MOSFET, when S2 provided high input to the gate, M2 allowed current the flow from the source to the drain. In other words, the PWM output could then provide input to the Op-Amp to drive the drum pad instrument. With this functionality, the push buttons S3 and S4 could send their input through their 2kΩ resistors (like with S5) which would then be read by the *drums*() function to set the PWM output to the Op-Amp and then to the speaker.

2.4 Op-Amp

Although the switches were key in terms of controlling the circuit, the most important piece to the keyboard's functionality was the non-inverting Op-Amp. As the name would suggest, the Op-Amp was responsible for regulating volume level of the speaker output. This was accomplished by connecting the output 10kΩ(R1) resistor to the B1M petentiometer(R2) to the inverting input of the Op-Amp. By applying the voltage divider equation to the Op-Amp equation, the output voltage can be determined by

$$V_{out} = \frac{R1 + R2}{R1} \cdot V_{in}.$$

With a maximum resistance of 1MΩ, the gain of the Op-Amp ranges from 1 to 101. Then by substituting in the maximum gain,

$$\text{Max db} = 20\log(\frac{V_{out}}{V_{in}}) = 20\log(101) = 40.09$$

which is loud enough to be easily and annoyingly audible without destroying anybody's ear drums. The full 0-40 db range suffices for solid pianissimo to mezzo-forte volume range. The final output is then sent from the Op-Amp to MOSFET M1 which then delivers current to the speaker depending on the state of S1.

3 Code and Feather Output

On the software side, a combination of functions within a continuous operational loop manages and manipulates sound in real time before outputting it to the physical components that produce the sound. The cornerstone of the system's functionality is its suite of functions, each dedicated to a specific aspect of sound control. For example, the *change_octave*() function is pivotal, allowing dynamic adjustment of the musical pitch. This function responds to user input to either double or halve the frequencies of all notes in the scale, enabling instantaneous octave shifts. This feature significantly broadens the instrument's musical range, to that of a typical piano as outlined in the introduction.

3.1 Instrument Switching and Wave Generation

As mentioned earlier, the *check_instruments*() function determines if the sound output comes from the keyboard or drum pad instruments. When S1 triggers *check_instrument*() to return True, the *drums*() function ceases all DAC outputs and then checks for the values of the push button inputs to determine the PWM output. When this isn't the case, the generation of specific wave forms is handled by *generate_waveform*() and *triangle_wave*() functions.

The former creates a square wave by rapidly toggling the DAC output between its maximum and minimum voltage levels, while the latter modulates the DAC output linearly between the low and high using addition of DAC intervals. The values for each frequency is calculated by dividing the frequency by the resolution of the serial data before multiplying by the general run time of the program to generate a milder triangle wave. Additionally, the *sin_wave*() function enhances the instrument's sound quality by generating sine waves, for acoustic instrument tuning when "b" is typed in the serial window.

3.2 Note Output and Real Time Handling

Through my favorite function and feature, *play_chord*(), polyphonic capabilities are enabled, which processes frequencies stored in the *active_notes* set. This set represents the notes currently being played, with each note's sound generated according to the selected waveform. The waveform type is determined by the user via the *wave_type* input, which toggles between a square wave and a triangle wave depending on physical input. This capability allows the instrument to produce a variety of sounds, from sharp, distinct square waves to smoother, softer triangle waves.

The operational backbone of the project is its main loop, which actively monitors and processes user inputs via the UART serial connection. This loop is crucial for ensuring that the instrument reacts promptly to user interactions, such as activating specific notes, adjusting octaves, or switching instruments. Commands received through the serial window are decoded and used to update the *active_notes* set, either adding new notes to start their sound production or removing them to cease playback. The loop continuously checks this set, and as long as it contains notes, the *play_chord* function is called to output sound. Further details and comments can be found in Appendix 5.2.

4 Discussion

In conclusion, the Physics 245 synthesizer keyboard project successfully integrated complex electronic components and software to create a multi-functional musical instrument. While the project aimed to emulate the extensive range of an 88-key piano and incorporate features such as a drum pad, distinct instrument switching, and dynamic volume control, certain limitations hindered the authenticity and depth of what the instrument could produce.

The synthesizer was adept at generating triangle and square waves, which were effectively utilized as distinct audio output modes for the keyboard. This achievement showcased a proficient use of digital-to-analog conversion techniques and waveform synthesis, crucial for electronic music applications. However, the project encountered significant hurdles with sine wave generation of variable frequency, a limitation that restricted the synthesizer's ability to produce sounds resembling those of acoustic instruments. This was primarily a programming issue where allowing variable frequency inputs caused odd gaps in the wave form. This made it rather unpleasant to listen to. As a result, the *sine_wave*() function was adapted to the role of a tuning tool rather than a feature for musical output so the sine wave functionality of the instrument would not be totally lost.

The drum kit component of the synthesizer did not replicate the sound quality a kick drum and snare drum for similar reasons. The sounds produced were distinctly electronic (square wave like); however, the drum kit accomplished its tactile goal of simulating a physical drum pad by using the digital designer push buttons. These buttons only triggered the *drum_kit*() function when pressed down similar to the pad that would be found on a midi keyboard.

Ultimately, this project achieved most of its designated goals. Particularly, the generation and switching between of triangle and square waves was successful, providing a solid base for sonic expression; however, the inability to produce variable sine waves and the electronic-sounding drum kit highlighted the limitations of the current setup. These elements, while functional, lacked the depth and authenticity that true sophisticated synthesizers offer. Regardless, the main goals and core functionality were achieved for this original project idea.

5 Appendix

5.1 Circuit Schematic

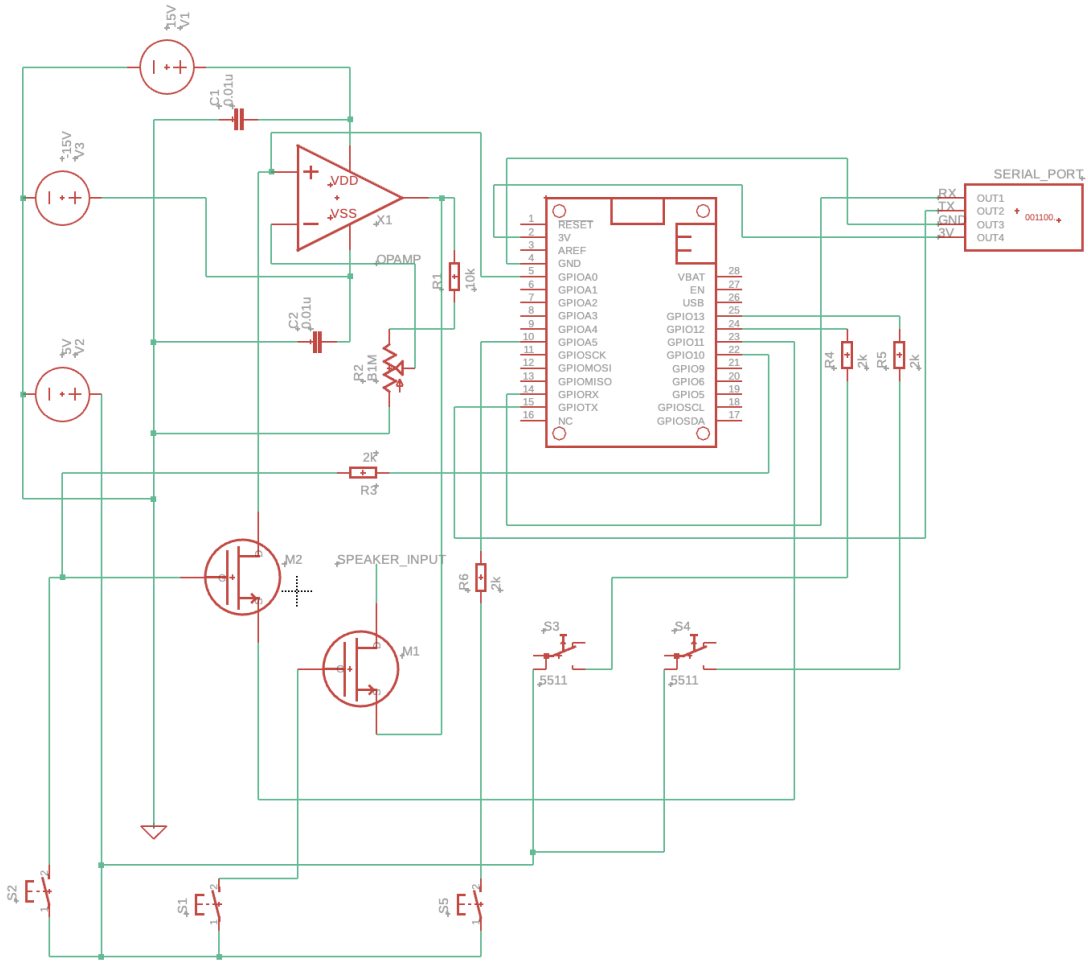

Figure 1: Schematic of Synthesizer Components

5.2 Python Code

```
# Import required libraries for interfacing hardware and performing calculations
import time
import board
import busio
import pwmio
import digitalio
import analogio
import math

# Initialize Digital to Analog Converter (DAC) on pin A0 for audio output.
# This is used to send analog voltage signals corresponding to sound waves.
analog_out = analogio.AnalogOut(board.A0)

# Setup UART (Universal Asynchronous Receiver/Transmitter) for serial communication.
# This enables data reception through TX and RX pins at a baud rate of 9600, with no timeout.
ser = busio.UART(board.TX, board.RX, baudrate=9600, timeout=0)

# Initialize digital input on pin A5 to select the type of waveform to be generated.
# This input determines whether to generate a triangle wave or another waveform.
wave_type = digitalio.DigitalInOut(board.A5)
wave_type.direction = digitalio.Direction.INPUT

# Setup digital input pins for drum triggers. These inputs control drum sounds.
kick_drum = digitalio.DigitalInOut(board.D13)
kick_drum.direction = digitalio.Direction.INPUT
high_drum = digitalio.DigitalInOut(board.D12)
high_drum.direction = digitalio.Direction.INPUT

# Initialize Pulse Width Modulation (PWM) on pin D11 to control drum frequencies.
# The duty cycle is set to 50% (middle of the 0-65535 range), initial frequency is 50Hz.
pwm_pin = pwmio.PWMOut(board.D11, duty_cycle=32767, frequency=50, variable_frequency=True)

# Setup a digital input to switch between different instruments.
switch_instruments = digitalio.DigitalInOut(board.D10)
switch_instruments.direction = digitalio.Direction.INPUT

# Constants defining the maximum DAC output voltage and resolution.
max_voltage = 3.3  # Max output voltage of DAC in volts.
num_steps = 1024  # Resolution of DAC (10-bit resolution).

# List of frequencies corresponding to musical notes C4 to B4 in Hertz.
notes = [262, 277, 294, 311, 330, 349, 370, 392, 415, 440, 466, 494]

# Set to store currently active musical notes.
active_notes = set()
```

```
46
47 def change_octave(user_input):
48     """
49     Adjusts the octave of the musical notes.
50     'o' increases the octave, doubling the frequency of all notes.
51     'l' decreases the octave, halving the frequency of all notes.
52     """
53     global notes
54     if user_input == b'o':
55         notes = [note * 2 for note in notes]
56     elif user_input == b'l':
57         notes = [int(note / 2) for note in notes]
58
59 def play_chord(active_notes):
60     """
61     Plays a chord composed of the frequencies in active_notes.
62     If wave_type is set to 1, it generates a triangle wave for each
    frequency.
63     Otherwise, it generates the default waveform.
64     """
65     for note_freq in active_notes:
66         if wave_type.value == 1:
67             triangle_wave(note_freq)
68         else:
69             generate_waveform(note_freq)
70
71 def generate_waveform(frequency):
72     """
73     Generates a basic square waveform for the given frequency by
    toggling
74     the DAC value between its max and min states at the frequency's
    period.
75     """
76     period = 1 / frequency
77     num_steps = int((1024 * period) * 73)
78     for i in range(num_steps):
79         dac_value = 0 if i < num_steps / 2 else 65535
80         analog_out.value = dac_value
81
82 def sin_wave(freq):
83     """
84     Generates a sine wave at the specified frequency by calculating the
     sine
85     value for each point in one cycle and outputting corresponding DAC
    values.
86     """
87     delta_t = 1 / freq / num_steps
88     for i in range(num_steps):
89         voltage = 0.5 * max_voltage * (1 + math.sin(2 * math.pi * i /
    num_steps))
90         dac_value = int(voltage / max_voltage * (num_steps - 1))
91         analog_out.value = dac_value
92         time.sleep(delta_t)
93
94 def triangle_wave(freq):
95     """
96     Generates a triangle wave at the specified frequency by linearly
    increasing
```

```
    and then decreasing the DAC value within the voltage range.
    """
    period = 1 / freq
    increment = (freq / 1024) * 2700
    dac_value = 1
    while dac_value < 65535:
        analog_out.value = int(dac_value)
        dac_value += increment
    while dac_value > 0:
        analog_out.value = int(dac_value)
        dac_value -= increment

def drums():
    """
    Controls the frequency of drum sounds based on digital inputs.
    Kick drum is set to 50 Hz, and high drum is set to 210 Hz.
    """
    if kick_drum.value == 1:
        pwm_pin.frequency = 50
    if high_drum.value == 1:
        pwm_pin.frequency = 210

def check_instrument():
    """
    Checks if the instrument switch is activated.
    Returns True if switch is active.
    """
    return switch_instruments.value == 1

# Main loop handles real-time instrument interaction and sound
   generation based on user inputs.
while True:
    if check_instrument():
        drums()  # Check and activate drums based on switch and
   triggers.
        pass
    user_input = ser.read(1)  # Read serial input from UART.
    if user_input:
        print(user_input)
        if user_input in [b'o', b'l']:
            change_octave(user_input)
        elif user_input in b'awsedftgyhujol':  # Note keys handling.
            note_index = b'awsedftgyhuj'.index(user_input)
            note_freq = notes[note_index]
            if note_freq not in active_notes:
                active_notes.add(note_freq)
        if user_input == b'z':
            active_notes.clear()  # Stop all notes.
        if user_input == b'b':
            sin_wave(440)  # Generate tuning tone at 440 Hz.
    if active_notes:
        play_chord(active_notes)  # Play active notes.
```