Generation of Audio signals using a Raspberry Pi Pico

**Jack D. Morrison**

**ID: 190009927**

**AC40001 Honours Project**

**BSc (Hons) Applied Computing**

**University of Dundee, 2023**

**Supervisor: Dr. Andrew Cobley**

***Abstract* –**

*The aim of this project is to investigate the methods that can be used to build a Digitally controlled Oscillator using a Raspberry Pi Pico and integrate it into a euro rack synthesiser format. The model must be designed to generate an oscillating voltage that oscillates at an audio rate frequency*. *This requires a method for controlling a voltage output, this requires use of the Raspberry Pi Pico’s PWM methods which can generate voltage pulses of a width that can be altered on the micro controller. If these pulses of variable width, charge a capacitor then it will result in a variable voltage since the pulses will charge the capacitor more, the longer the width of the pulse and release a higher voltage. This means that we can oscillate voltage by alternating the pulse widths outputted by the Pico. There are a few methods of doing this, the first that I investigated was table lookup, this requires a table containing several integer values. The micro controller reads the table and uses the integer to determine the width of the pulse. This allows all the processing of complex audio waveforms to be done before the execution on the Pico. The second method that was investigated was calculating the width during execution. This proved to be more efficient on square waves but other waves such as triangular was very demanding. But this method allowed easier alteration to the length of the square waves and the angles on the triangle.*

# Introduction

The objective of this project is to develop a Digitally controlled oscillator capable of producing audio rate frequencies using a Raspberry Pi Pico. This report will detail the extent to which the inexpensive Raspberry pi Pico can be used as an audio synthesiser and then integrate the results of my findings in a euro rack synthesiser format. The outcome of this project should be that the full capabilities of Raspberry Pi Pico as an audio synthesiser. The methods researched in this project could be used to create audio synthesis devices that would be commercially viable for example it could be integrated into a toy keyboard for kids to learn how to play piano where they don’t have enough money or space for a full-sized keyboard.

The Raspberry Pi Pico is a microcontroller which can generate digital outputs which poses a problem since the outcome of this project is to generate an analogue audio output. This requires us to convert our binary 1 or 0 output into a continous output. The method that is used in this project is pulse width modulation where voltage size is controlled by modulating the duration of the pulse also referred to as width. These pulses then go through a transformation circuit containing a capacitor routed to ground which stores the voltage from the pulse for the duration of the pulse. The resultant voltage from the capacitor that has a magnitude that corresponds to the duration of the pulse. This allows us to generate a controlled analogue voltage which can be altered by changing the width of the pulses going into it.

The pulses are outputted by the Pico at a rate that can be altered by changing the time between each pulse. This means that given a certain pulse rate and a desired frequency the number of pulses in a single period can be calculated by dividing the pulse rate by the desired frequency. This is the start to generating a frequency since it is known how many pulses are in a single wave, the wave shape can be controlled by altering the widths of pulses from 0 to x number of pulses.

Two methods that will be investigated in this report are using runtime calculations to find width and the other is generating the pulses first, then storing them in a table and then using table look up to find the pulse width at runtime.

Both methods have their own use cases and limitations. The main benefit of runtime calculated width is that it is possible to alter the wave forms profile during runtime but is limited to simplistic calculations since large complex calculations will take a large amount of time to compute and may reduce the frequency of the wave if it becomes backlogged with too many large calculations. On the other hand, the table look up method allows for much more complex waveforms and a much more stable frequency although this limits it in its ability to alter the wave profile. This project will also investigate how a combination of both methods can be used to create unique sounds.

One of the largest requirements of this project is to deliver a synthesiser that can be integrated within a Euro rack synthesiser format. This means that the device will need to be able to take an input from a synthesiser sequencer. A sequencer is a device that allows the user to construct sequences of musical notes by sending a control voltage that is indicative of the note it is trying to signal to the synthesiser to generate. This means that the synthesiser will be able to take this voltage as an input and read the value so that it can produce the intended result. Of course, this also means that the synthesiser needs means of changing the frequency of the sound that it produces. In this project the methods for changing the frequency will also be researched and tested. One method that will be researched in this project is storing different frequencies in wave tables and changing the table to the desired frequencies table when required. Alternatively, instead of holding the frequencies in wavetable it is possible to use only one wave table that hold the pulse widths for a single frequency and use it as a base frequency and modulate the rate that the widths are outputted so that it decreases the Frequency of the waveform. This allows for a much wider range of frequencies that the system can produce and doesn’t require as much memory usage. The freed-up memory space also allows for more memory that could be used for other wavetables with different effects. On the other hand, it does require more calculations to alter the rate since you need to calculate how much the rate needs to be altered to achieve the frequency.

The integration with the sequencer also requires a method to stop the notes from playing. This is usually controlled by a ‘gate’ which generates an output voltage when the note is required to play and stops when the voltage is not require. The solution to this is taking the voltage and using a PIN to check if the voltage is present or not and then output if the PIN has a voltage or stop when it doesn’t. Alternatively, the analogue solution is using a voltage-controlled switch to control the output of the PWM circuit where if a voltage is present the switch stops the output.

# Background

PWM on the Pico can be integrated on any of the 30 GPIO pins which are divided into 8 slices. GPIO pins 0-15 correspond to slices 0-7 and GPIO pins 16-29 correspond with 0-6. Each slice has 2 output channels (A/B) which have 2 pins associated with it as shown in the table below.

A picture containing text, crossword puzzle

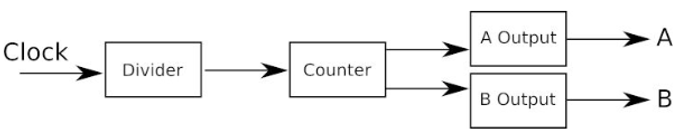
Description automatically generated

Each GPIO PIN has a slice associated with it, the PIN shares a slice with 1 other PIN and the hardware identifies each PIN with its slice number and the channel which could be either A or B depending on which PIN is required. These variable can be defined by using the pwm\_gpio\_to\_slice\_num(uint gpio) function which returns the slice number and pwm\_gpio\_to\_channel(uint gpio) function returns 0 or 1 for channel A or B.

To enable pwm on a pin it needs to be initialise first by using the gpio\_set\_function(int GPIO PIN, GPIO\_FUNC\_PWM); so the hardware know that the function of the PIN is for PWM only.

To turn PWM off or on you can use the pwm\_set\_enabled(uint slice\_num, bool enabled) function or if you have multiple signals that need turned off or on you can use pwm\_set\_mask\_enabled(uint32\_t mask). The mask represents the eight PWM slices as the first eight bits.

This also requires the target\_link\_libraries(hardware\_pwm) inserted int the CMakeLists.txt file. Programming pico in C book



PWM generation in the pico work by stepping using the clockspeed which means that the steps occur every seconds. You can increase the time between the steps by increasing the clock divider which is initially set to 1 which can be substituted into the formula so that where is the time between steps and is the clock frequency. The steps work with a 16-bit counter meaning that the can be stepped on a maximum of times before wrapping around. The number of steps before it wraps can be configured to be any value between 1 and by changing the “wrap” value in the PWM configuration. This higher the value you set the wrap to be the lower the output frequency of the PWM signal will become likewise the lower the value the higher the frequency but the lower the value you make the wrap the lower the granularity of the duty cycle you set can becomes. This is because in the pico the duty cycle is set to a value between 0 and the wrap. This is the case because the when the pico steps through the counter it outputs a the IOVDD voltage until it reaches the limit value.

In the documentation is lays out how to change the wrap size using the PWM\_hardware library. The functions were pwm\_set\_wrap(uint slice\_num, uint16\_t wrap). The wrap value controls how high the counter will count up to. In the normal mode it is explained that after this the counter counts back up to the value from 0. On the other hand there is a second mode that can be used called “phase-correct” mode which instead counts back down to 0 then up to the count. The normal mode mimics a sawtooth wave as you can see from the diagrams from the documentation:

Diagram, line chart

Description automatically generated

And the Phase correct mode is a triangular wave:

Diagram, line chart

Description automatically generated

This subsequently means that the frequency is halved in phase correct mode, but it also means that the pulses produced are active for double the time duration. As shown by the diagram the phase is corrected which means that pulses are centred on the 0 count of the counter. The documentation states that it can be used by implementing the pwm\_set\_phase\_correct(uint slice\_num, bool phase\_correct) method from the hardware\_pwm library.

Then documentation also explains that to set the duty cycle of the pwm output you must set the channel’s level. There are 3 methods for doing this, first is using the pwm\_set\_channel\_level(uint slice\_num, uint chan, uint16\_t level) which takes in the slice of the channel, the channel (A or B) and the level. Secondarily you can set the both channels using the pwm\_set\_both\_levels(uint slice\_num, uint16\_t level\_a, uint16\_t level\_b) which has separate level parameters for channel a and b of a given slice. The third is the helper function pwm\_set\_GPIO\_level(uint GPIO, uint16\_t level) which will set the level without knowing the slice or channel number.

The Book Programming the Raspberry Pi Pico In C By Harry Fairhead explains a couple of helpful equations that can be used to find the frequency of the PWM output based on the clock frequency () and the wrap. The equation looks like for non-phase correct mode hence the wrap to produce a frequency can be found by the equation . Fairhead also states an equation for finding the level of a duty cycle as , with duty being the duty cycle as a fraction.

He also specifies the same equations for phase correct mode as and to account for the decrease in frequency.

This can also be modified by the clock divider which divides the value in the previous equations. The documentation has 2 methods for doing this which are pwm\_set\_clkdiv\_int\_frac(uint slice\_num, uint8\_t integer, uint8\_t fraction) and helper function pwm\_set\_clkdiv(uint slice\_num, float divider). Fairhead states in his book that “The divider is decomposed into an 8-bit integer and a 4-bit fractional part which specifies the fraction as fract/16.” Hence you can find the new clock frequency by dividing by the integer, then dividing by fract/16.

Research began by looking into PWM with the Raspberry pi Pico which led to finding the “Hardware design with RP2040” Datasheet which contains a circuit diagram for use with a PWM audio source see figure 1 in appendix. The circuit from the datasheet has 2 PWM outputs from the Raspberry Pico which go into a small logic buffer. Then each signal passes through a 220ohm resistor, then past a 100 nano farad resistor and then a 100-ohm resistor, both connected to ground. Each signal is then connected to a 47 micro farad capacitor and then passed a 1.8k ohm resistor connected to ground. The effect this would have on the pulses outputted from the Pico would be that the logic buffer would clean up the signal with an external 3.3V supply before going into the rest of the circuit. The rest of the circuit stores the pulses inside the capacitors and release them when the pulse ends meaning that the longer the pulse the higher the voltage released from the capacitors.

Another place of research was Robin Grosset’s pico-pwm-audio github repository which contains an example of PWM using table look up. Inside the repository Grosset includes a C program that generates a PWM output when compiled into a UF2 file using the raspberry Pi Pico C build configuration in VS code. The C program uses the Pico standard library along with the interrupt, PWM and sync library to accomplish this. The C file defines a PWM audio PIN (28) and includes a sample.h file which has a macro called WAV\_DATA\_LENGTH and an unsigned int array called WAV\_DATA this is the table that is looked up in his method. It contains values ranging from 0 to 255 to represent the pulse widths used to replicate the sample audio signal. The code also overclocks the Raspberry Pico to 176000Mhz and initialises the pin with a function from the PWM library that sets up the PIN in PWM mode. It then creates an int value called audio\_pin\_slice which is set to the PWM slice which a value attached to the GPIO PIN which tells it when to fire. It then clears the PWM channel interrupt on the PIN and Enables a single PWM instance interrupt and sets the handler of the interrupt to a function defined before main () called pwm\_interrupt\_handler which checks its wave position is less than the wav data length multiplied by 2^3, utilising bit shifting to increase efficiency. This is because each value is repeated 8 cycles. It then sets the value of the pulse width and increases the wav\_position. Also includes an else to reset the wav\_position to start.

In the final part of the main it sets a pwm\_config to the default config provided by the hardware/pwm.h file and sets the clock divide to 8.0 which adjusts the rate of events seen on the PWM PIN before passing them on to the PWM counter. This value changes the rate at which events fire in other words. It then sets the wrap to 250. The wrap sets the highest value the counter can reach before turning to 0. It then initialises the pin with the configuration and start parameter set to true. After initialisation the GPIO 28 PIN’s level is set to 0 and starts an infinite loop that waits for interrupts to wake up the core and execute the handler function.

The grosset repository contains a great approach to PWM and it is certainly one way of doing it and wasn’t particularly well explained so a lot of the knowledge gained was using the Raspberry pi pico library documentation to find out what the methods were doing but was a bit vague in explaining the actual functioning of the methods.

# Specification

The final product of this project must use the Raspberry Pi Pico to produce an oscillating voltage using PWM so that it can be controlled by the user. The product must also allow the user to control the frequency and tone of the output voltage. The method for controlling frequency in this project is a Korg SQ1 pitch sequencer so the product must be able to interface with the pitch sequencer. This requires the Pico to take in a control voltage to determine the frequency that is requested, and that the Pico must also read the Gate output to determine when to play notes. The product should also have button inputs so that the different tone options can be switched to easily. The prototype must be able to output different waveforms such as Sine waves, square waves, triangle waves and parabola. Additionally, it could combine waveforms in runtime this could be done with different notes to make chords or add harmonics to simulate plucked notes.

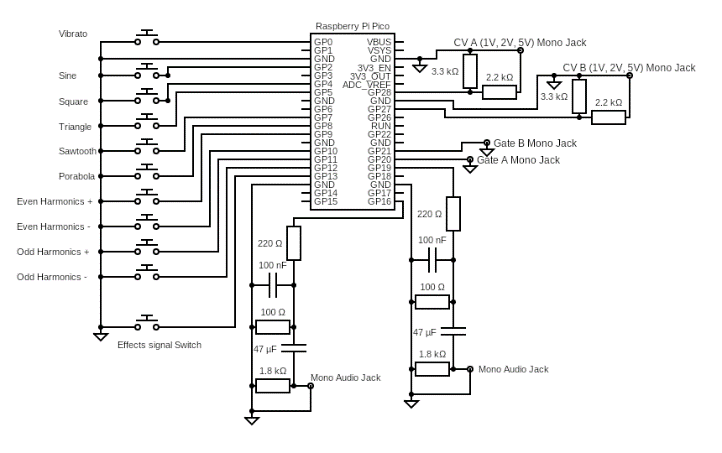
# Design

The Design stage began with PWM the circuit which was inspired by the diagram from the “Hardware design with RP2040” Datasheet by Raspberry Pi.

Diagram, schematic

Description automatically generated

Figure 1. The schematics for an Analogue PWM circuit from “Hardware design with RP2040”

The PWM circuit is instead using a single PWM pin that passes through a 220ohm resistor which reduces the current going through the capacitors. It then contains a 100 nano farad resistor connected to ground, this allows the pulses to be stored and then released. Next past a 100-ohm resistor connected to ground, the signal is then flows into a 47 micro farad capacitor and then continues past a 1.8k ohm resistor connected to ground. Then the signal goes into a mono audio jack and the ground connection of the audio jack to the ground of the circuit.

# Implementation and Testing

## Overview

The first step of implementation was creating a working circuit that could take a PWM output from the Raspberry pi Pico and output a oscillating analogue voltage. Next was to develop methods of generating a consistent and accurate frequency so that an output frequency could be determined programmatically ideally so that an integer value in the Program corresponds to the value in Hz out from the circuit.

The next stage of implementation to reach the final design specification is to take an analogue voltage and convert it into a digital value and use it to determine the frequency output.

Once the frequency can be altered by an input voltage the next specification to address is the alternate waveforms these consist of sinusoidal, triangle, sawtooth, reverse sawtooth, and parabolic waves. This required a method for generating wavetables, and a method for alternating between them.

With the wavetable solution addressed the next step is to find a runtime solution to produce these waves, and then introduce a controlled voltage input to change the frequency. The wave profiles of some waves can then be modified to take in another controlled voltage to alter the duty cycle of a square wave and the angle of a triangular wave.

The next stage is integrating sinusoidal harmonics into the waveforms at runtime. This also means implementing another method to add or remove harmonics from the wave.

After the waveforms have been fully explored the next step in implementation is taking the results and integrating it into a euro rack synthesiser.

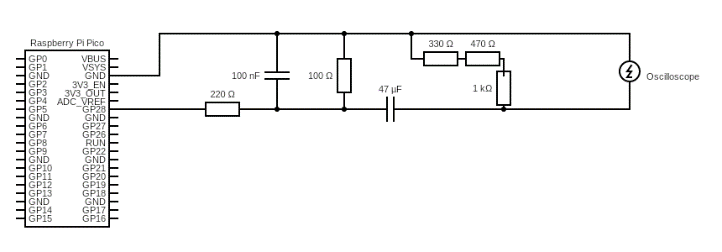
This first involved using a Korg SQ1 sequencer to alter the voltage. This step also involves modifying the prototype to take a voltage per octave format and accepting a 5V signal that allows the Pico to span 5 Octaves and a gate which tells the Pico when to play notes. The next step is then routing the signal through a voltage-controlled filter and then through a voltage-controlled amplifier. These also contain an effects loop which allows for modulation of the attack, decay, sustain and release of the Wave form produced by the Pico.

The next developmental goal to achieve was implementing a method of generating vibrato where the frequency alternates up and down by half step intervals. This meaning that the frequency goes up a semitone then down to the frequency and then down a semitone then back to the frequency and then repeats.

Once the pico is fully integrated within the circuit the final stage is to have 2 PWM signals produced by the Pico. This involved taking integrating another ADC input and another gate input as well as another PWM circuit. This was also integrated within the waveforms were integrated so that each signal’s waveforms can be altered independently.

## PWM Circuit

The first step of implementation was creating a circuit that can take a PWM output from the Pico and transform it into an oscillating voltage. The circuit for this first prototype looked very similar to the RGrosset implementation as shown by the diagram.



This implementation used the same GPIO 28 as grosset but instead of using an Audio Jack the output went straight into an oscilloscope. Another slight change was the use of a 330Ω, 470Ω and 1KΩ resistor in parallel instead of a dedicated 1.8KΩ resistor.

This prototype was first tested using RGrosset’s pico-pwm-audio code from his GitHub and was compiled on an Ubuntu virtual machine with the Raspberry Pi Pico C/C++ SDK for VS code installed. The compiled UF2 file was then transferred onto the Pico for testing. The results of the first prototype showed several different frequencies which indicated that the circuit had been implemented correctly.

## Accurate Frequency

After the circuit was established, the next step was to generate a single frequency which is generated so that it produces a specific frequency and tested to ensure the frequency is generated with the expected frequency. This was achieved by first forking the code developed by RGrosset and removing the the sample.h, ring.h, that’s\_cool.h, circuit diagram.png and the Raspberry Pi Pico PWM Wave File Converter.ipynb. The pico-pwm-audio.c file was also updated to use a new .h file created by a python program.

The python program was Implemented to take in a frequency and sample rate as an input and then generate an array that is written to the new.h file and used to modify the duty cycle of the pwm output.

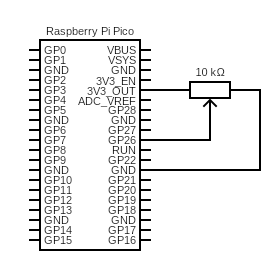
The first iteration of the python program generates Pulse widths for a single sine wave. This was achieved by defining a variable for the period of the wave equal to 1/Frequency then create a multiplier variable = 2 times pi over the period of the wave, which is referred to as B in the program, this variable, B, is used to find the duty cycle to equal the amplitude at each point along waves cycle. When the wave is reproduced, pulses are generated at a certain rate which depends on the clock frequency and clock divider hence it is also of use to find the interval size(1/samplerate) which is the size between 8 calls of the handler function in RGrosset’s C program which are called at a rate of clock frequency/clock divider. The interval size was used to iterate until it has reached the end of one cycle of the wave. At each iteration the duty cycle at the point is found by using a sine function of the variable B multiplied by the size of the intervals iterated through. With the duty cycle at that point found we need to create a value that the pico uses to determine the duty cycle. The sine wave function alternates between -1 and 1 but this is no use for PWM because the value needs to be a positive integer from 0 to the size of the PWM wrap. Which requires addition of one to the value to make it positive then multiplied by the size of the PWM wrap to justify it for the program. It then writes the data as an C array to a header file which is included in the C program.

The RGrosset Program was then modified to use the Python generated array. It was then compiled and tested through the oscilloscope by checking that the period of the waves are constant and of equal length to the period defined in the python program.

## Voltage Controlled Frequency

The next step in development was the use of a controlled voltage to determine the frequency of the PWM output. This involved implementing 3 parts, creating a controlled voltage to alter the frequency, converting the analogue voltage input into a digital form, and changing the frequency of the signal with digital value.

### Controlled Voltage

The controlled voltage used to change the frequency of the output was implemented by using a 10kΩ potentiometer and the 36th PIN (3V3(OUT)). The Vcc pin of the potentiometer was connected to the 3V3 OUT PIN and the output pin connected to the GPIO 26 PIN with the ground pin of the potentiometer connected to a ground pin on the Pico as shown by the diagram below. 

### Analogue to digital conversion

The analogue to digital conversion was implemented using the “hardware/adc.h” library. This was used to initialise the GPIO 26 pin for use in adc using the adc\_gpio\_init() method in the main function. Then using adc\_select(0) to select the GPIO 26 PIN. We then set a constant called conversion factor in the global scope which converts the adc\_read() value to an integer representation of the voltage.

Then, inside the handler function we define the adc\_value to equal the adc\_read() multiplied by the conversion factor each time the wav\_position has equalled the size of the waves array length.

### Altering the frequency

The adc\_value variable is then used to alter the frequency of the output. This was done by altering the clock divider which means the core frequency is divided into parts. The original clock divider used was 2.0 meaning the 176Mhz core clock frequency is divided by 2 for creating an effective frequency of 88Mhz which if we use the formula knowing that we have a wrap of 250. We get an output frequency of 352,000Hz which is iterated through 8 times to iterate through a position in the wave so effectively we have an output frequency of 44,000Hz. If we increase or decrease the by changing the clock speed we can increase or decrease the rate at which the values are outputted from the Pico. Which means that if we increase the clock divider we can increase the frequency likewise if we decrease the clock divider we decrease the output frequency. The next step is to take create a function that returns the clock divider necessary to generate a voltage. This was implemented by taking the new frequency as a float parameter and using 2 globally defined constants called WAV\_FREQUENCY which is the frequency of the output with the original clock divider and clkDiv which is the original clock divider. The function divides the WAV\_FREQUENCY constant by the new frequency to create a coefficient of the difference between the frequencies, then multiplies the coefficient by clkDiv to give the clock divider needed to achieve the new frequency. This value is then returned out of the function. This function was implemented to alter the frequency by first multiplying the WAV\_FREQUENCY by the adc\_value to create a newFrequency so that the frequency will increase as the potentiometer is turned. Then I used the function to find the new clock divider and then set the new clock divider using the set\_pwm\_clkdiv(int slice, float clkDiv) method with use of the slice variable used in the initialisation of the PWM pin, which put into the global scope so it could be accessed in the handler function, and the new clockdivider.

### Testing

This was then tested using an oscilloscope by turning the potentiometer knob around and testing the if the frequency was altered. Then I used the oscilloscope to determine the voltage from the potentiometer with a slight turn of the knob and then calculating the what the WAV\_FREQUENCY multiplied by the voltage would give. Then I attached the oscilloscope back onto the PWM circuit and read the frequency of the output to see if it corresponded with the value.

## Alternate Waveforms

### Overview

Like the sinusoidal waveform the other input arrays for the alternate waveforms were implemented using a python program to generate the values then writing them to the header file. The program was also altered so that all the alternate wave forms were generated from methods of a class named ‘wave’ which has fields of frequency, amplitude, sample rate, period, B, interval, and wrap. These values were used in the methods which return a string of the input array used to reproduce the wavicles. All the fields are set in the constructor which takes in frequency, amplitude, sample rate, and wrap as parameters and calculates the other fields using them as described in the accurate frequency section.

### Square Wave

The square wave was implemented using a while loop that iterates through using a variable called x which increase by 1 interval each loop until it has reached half of the period length + one interval. Before the loop a variable called vals is set to equal the wrap field but converted into a string. Inside the while loop, the ‘vals’ variable is concatenated with ‘, ’ then concatenated with the wrap field converted to a string. Another while loop was then implemented which again iterates through using x which increases by 1 interval each cycle but until it equals the period field + 1 interval. The while loop similarly concatenates ‘, ’ to the ‘vals’ variable but instead concatenates “0” after. Then the vals variable is returned from the method.

### Triangle Wave

Like the square wave implementation, the triangle waveform was implemented by looping through in different fractions of the wave. The method starts with defining the tan\_theta variable which is used to find the duty cycle at each point along the period of the waveform. This is set to 4 times the amplitude field divided by the period field. This is because a triangular wave can be split into 4 triangles with a height of 1 amplitude and width of ¼ of the period of the wave or 2 triangles of 2 amplitudes of the wave and ½ the period. The quarter wave implementation was used so that the wave starts with a duty cycle of 50% just like the sinusoidal waveform. A variable called x is used again to iterate through the wave and is used set the first value which is x multiplied by tan\_theta to give the amplitude at 0 which is set to variable v. This was then used to initialise the vals string by taking the v variable adding it to the amplitude field and dividing it by 2 times the amplitude field to justify it to start in the middle and then multiplying that by the wrap field to get the duty cycle for the program. Then a while loop is used to iterate through the first quarter of the waves period using x where it adds the values to the string using the tan\_theta variable and multiplying it by the x value. Then we iterate through the next 2 quarters of the period again using x. In this loop x times tan\_theta is used to subtract from 2 times the amplitude to give the amplitude relative to the height from 0 at that point, then that value is justified the same as in the first quarter then multiplied by the wrap to give the duty cycle at that point.

The final while loop iterates through the last quarter of the period again using x. Where it finds the value of x times theta and justifies it instead to be aligned from the bottom of the wave by subtracting the value by three amplitudes then dividing it by 2 amplitudes. Then it returns the vals string which are the values found by iterating through the wave concatenated as strings and separated by commas.

### Sawtooth Wave

The sawtooth wave was implemented much in the same way as the triangle wave, instead of using the 4 subsections instead I used just 2 sections this meant that the tan\_theta value changed to 2 times the amplitude divided by the period field. Similarly, to the last wave it iterates variable x from 0 to half of the period + interval. We then multiply the value x by the tan theta to get the value and then concatenate the string with the value plus amplitude divided by 2 amplitudes, multiplied by the wrap and turned into a string. Then I added another while loop to iterate the x variable further until it reaches the period plus the interval in another while loop that finds the value but instead concatenates the value minus the amplitude divided by 2 times the amplitude. This means that the value is justified to the lowest duty cycle. Then it returns the string of the values separated by commas.

#### Reverse Sawtooth Wave

This was created by copying the previous function but instead the value was calculated by subtracting the x times tan\_theta value from the amplitude when its concatenated in the first loop and in the second subtracting it from 3 times the amplitude when concatenating it.

### Parabolic Waves

The method for generating the parabolic wave input array was implemented again by iterating through using the x value from 0 to the period plus one interval. The value at each iteration was found by squaring x minus half of the period divided by half of the period. This can be used because as x rises toward half of the period the fraction of increases towards 0 from -1 and since it is then squared the value decreases exponentially until zero. Then from 0 to 1 it increases exponentially. The value is then multiplied by the wrap and concatenated to the vals string and returned from the method.

### Harmonic waveforms

The harmonic waveforms were implemented much later in the program’s development. The harmonic waveforms were created by copying the sine wave method but instead iterating using a variable called h from 1 to 12, the h variable was used in creating a new B variable that multiplies the frequency field by h+1 and multiplying that by 2 times pi. Then I use that variable to find the values of the harmonic’s sine wave input array and concatenates it to a string then adds the string to an array of input array strings and then returns the array with the values from each of the harmonics.

### Writing to the header

The python file was then modified to create a new wave class using variable that were found using pythons ‘input()’ method that allows easy changes to the frequency and sample rate which is useful when testing the project. These values were then used to initialise the wave object which is then used to define variables for each waveform by calling the specific generate waveform method. Python’s open() method was then used to open the header file which required assigning a variable f as the file object and called the write() method from it to write to the header. A first thing to write to the header file was 2 macros the WAV\_FREQUENCY macro used to alter the clock divider and the WAV\_DATA\_LENGTH. This was done by writing “#define WAV\_FREQUENCY” then converting the int value from the input method into a string and writing that to the file then writing ‘\n’ to the file to break the line. The WAV\_DATA\_LENGTH was done in the same way except it found the value of the macro by dividing the sample rate by the frequency. Then I wrote a constant of the new clock divider found by dividing 88,000 by the new sample rate 88000 was found by taking the clock frequency and dividing it by the wrap and number of times it repeats a value in the pwm handler (8). The waveform input array strings are then written to the header file by writing "uint8\_t \*wave name\*\_WAV\_DATA[] = {\n" to the file then writing the string input array variable to the file followed by “};\n”. The harmonics shared the similar format except that it iterates through the array outputted by the wave object in the format of "uint8\_t HARMONIC\*number\*\_WAV\_DATA[] = {\n"

### Testing

To test the wave forms a button was connected to GPIO PIN 10 and the 3V3OUT PIN meaning that when the button is pressed the GPIO 10 PIN gets a HIGH reading. The PIN was set up in the C program using a macro called WAVEBUTTON set to 10 and in the main function gpio\_init() was used to initailise the pin and gpio\_set\_dir(WAVEBUTTON,GPIO\_IN) was used to set the direction. The PIN was implemented to use a handler function which was enabled with gpio\_set\_irq\_enabled(WAVEBUTTON,GPIO\_IRQ\_EDGE\_RISE ,true). This also sets the interrupt to be initialise when the voltage across the pin is raised up. The handler function was then assigned to the pin using gpio\_add\_raw\_irq\_handler\_masked(( 0x01 << WAVEBUTTON),&rawHandler1) function which take in a bit mask of the integer value of the pin which is converted in this case by bitshifting the 0x01 value up by the magnitude of the WAVEBUTTON macro. Then we link to the memory address of the rawhandler.

The raw handler first checks to the event mask of the interrupt are from the WAVEBUTTON pin and is from a rise event. If so then it acknowledges the interrupt and iterates an integer variable declared in the global scope called button between 0 and 5. This variable is then used in the pwm interrupt handler function in a switch case that has 8 cases for each value 0 to 5, each corresponding to a waveform, the switch case is situated where the pwm\_set\_gpio\_level is called. In each case of the switch statement the pwm\_set\_gpio level is set to equal the corresponding input array value at the given wave position.

This means that when the button is pressed it changes the input array and subsequently, the wave form that is produced. This was then tested using an oscilloscope to ensure that the frequency of waves stayed constant, and that the amplitude of the oscillating voltage stayed constant independent of the waveform. The waveforms also had to conform to the specific shape of the wave form.

## Runtime solutions

### Overview

The runtime solution was implemented using a wavelength variable of a frequency which isn’t the distance wavelength but instead is the number of wraps cycles needed to achieve a frequency. This can be calculated as the clock frequency divided by the wrap size, the frequency, and the clock divider. This variable is the same as the array length of the input array method. In the implementation of the runtime solution this was divided a further 8 times because each value was repeated 8 times in the handler function. This was then used in runtime as an analogue to the period of the wave. Hence an integer variable named wav\_position was created to iterate from 0 to the ‘wavelength’ with each call of the PWM handler function and resets once the variable has exceeded the ‘wavelength’. This was created so that the duty cycle at a given point in a waves cycle could be calculated. Different methods were then implemented for each waveform to calculate the duty cycle using this variable.

### Square wave

The square wave duty cycles were implemented by checking if the wav\_position variable was less than half of the “wavelength” and if true output a 100% duty cycle by setting the GPIO level to the size of the wrap. Else it will output a 0% duty cycle by setting the GPIO level to 0.

### Triangle wave

The Triangle wave was implemented by splitting it into 2 halves, the “up” half where the duty cycle increase from 0 to 100% in the span of half the wavelength and the down section that goes from 100% to 0. This meant the duty cycle was calculated taking the wav\_position and dividing it by half of the “wavelength” then multiplying by the wrap to find the level to generate the required duty cycle for the up part. The down part was then implemented by first finding the fraction that the wav\_position is between half the wavelength and the wavelength. This involved subtracting the wav\_postion by half the wavelength and then dividing by half the wavelength. That value was then used to subtract 1 to give the invert the direction then multiplying by the wrap to give the level needed to produce the desired duty cycle.

### Porabola wave

The parabola waveform drops at an decreasing rate until the halfway point then rises at an increasing. This means that the duty cycle of the wave drops from 255 increasingly slow rate as wav\_position increases at the same rate it decreased.

This method was implemented by first squaring the wav\_position subtracted by the half of the wavelength then dividing by half of the wavelength. This was implemented in this way so that as the wav\_position increases towards the halfway point it decreases in duty cycle gradient because the fraction becomes closer to 0 at an exponentially smaller rate and then the fraction becomes negative and continues to decrease until its -1 which is exponentially rising until the end of the loop. The value was then multiplied it by the wrap to give the level for the GPIO.

### Wave Shape control

The wave shape of the square and triangle waves was altered using the potentiometer input from the frequency alteration implementation. This meant that instead of taking the adc\_value in the C program and using it to change the clockdivider it was instead used to determine the point along the period where the calculations change. In the square wave this meant taking the wavelength and multiplying it by the adc value to find the value of the pulse length which is the length that the square wave is set to 100% duty cycle. Then setting a limit so that the pulse length doesn’t exceed the wavelength. This meant that where the adc\_read() is called that the value modifies a global variable named “pulselength” to equal the wavelength variable times the adc\_read value multiplied by the conversion factor. This variable was then implemented into the handler function so that it checks if the wav\_position is less than the pulse length then set level to equal the wrap.

## Korg SQ-1 sequencer integration

To integrate the Korg SQ-1 with the Raspberry Pi Pico it required reading the output voltage from the Korg SQ-1’s CV A output and using it to set the frequency of the output. It also required reading the OUT-GATE output and using it to determine when to update the frequency and when to play the notes when it is not connected to the ADSR.

This also required a integrating a different way of handling frequencies because the voltage from the SQ-1 didn’t translate to accurate notes when multiplied by the base frequency. This is because the SQ-1 uses the euro rack 1 volt/octave standard which means that every 1/12th of a volt corresponds to a semitone. Since the musical frequencies increasing logarithmically and this increases linearly there are some differences. This meant that the voltage had to be translated into which note it corresponds to. The next stage was implementing a tuning method to derive the different notes. First one to be implemented was just tuning.

### Just Tuning

Just Tuning was implemented by adding a method for generating semitone frequencies to the python file. This method was implemented in below the wave table generation. The method iterates through an integer value from 0 to 11, finding the value at each integer. It finds the frequency by multiplying the pitch ratio by the perfect first. The pitch ratios are as follows: m2(16:15), M2(9:8), m3(6:5), M3(5:4), P4(4:3), TT (45:32), P5(3:2), m6(8:5), M6(5:3), m7(7:4) and M7(15:8). It starts by finding the lowest octave’s frequencies which is one quarter the size of the base frequency. Then we find the frequencies for the following 5 octaves and write them to the file as constant integer array.

### 2 Volt Solution

The Korg SQ-1 has 3 volt/octave modes, 1V,2V and 5V with 1V being across 1 octave, 2 V being across 2 octaves and 5V having 5 Octaves. The first one to implement was the 2V solution which was implemented using analogue to digital conversion. The Pico took the input from a mono audio jack, that was connected to GPIO pin 26 and ground. The same methods as before were used for analogue to digital conversion to convert the voltage across the GPIO 26 pin into an integer representation. The adc\_value was read in the handler function after the wav\_position had been reset. The frequency representation of the voltage was then found, still inside the handler, using an incredibly arduous if statement tree that checked if the value was greater or less than 1 if so then check if its greater than 0.5 then if its greater than 0.25, then 0.125, then 0.0675, then corresponding else statements to differentiate between each 0.0675 volts where it can then determine which frequency it corresponds to from the array. Once the frequency was found the change clock divider function was used to alter the frequency. This was then tested with the oscilloscope and frequency values were checked.

### OUT-GATE

The OUT-GATE was implemented into the Pico using a mono jack connected to a GPIO pin and ground. The pin was then initialised using gpio\_init() and the direction was set using gpio\_set\_dir(). The GPIO pin was then pulled down to sets the pin low which for the gate means that the pin will shift to high when the gate is triggered. As with the button interrupts had to be enabled using gpio\_set\_irq\_enabled so that when the high reading happens the handler function is called. Then a handler was associated with the pin using the gpio\_add\_raw\_irq\_handler\_masked() function, using the same method as the button in the waveform solution to turn the macro into a bitmask. The same handler function was used but an additional if statement was added to check if the interrupt is from the gate and if the event that triggered it was a rise of voltage if so, then acknowledge the interrupt, take an ADC reading, convert it to the frequency and change the clock divider. The ADC functionality was then removed from the PWM handler function.­­­­­ After it was tested by attaching the PWM output to a speaker to see if the frequency changed when the gate changed. Then a Global Boolean variable added to switch the signal off and on when the gate is off or on. This meant updating the raw handler for the gate to have an additional if statement to check if the interrupt is from the gate and if the event is a fall of the voltage, if so, then change the Play Variable to false. This also meant changing the Variable to true first Gate statement. Then an if statement was added that wraps around the logic in the PWM handler function that checks if the play variable is true. This was then tested through the speaker to ensure that output occurs when the gate is not active.

### 5 Volt Solution

The 5V was implemented by first altering the input method, previously the CV-A was read in from an input mono jack wired directly to the GPIO 26 pin. If the output from the SQ-1 was set to 5V this would mean that the GPIO 26 pin voltage would exceed its stated capacity, hence the voltage had to be reduced to 3V. This required modifying the circuit so that the ADC pin was connected in the middle of 2 resistors that are connected from the mono audio jack output to ground. The resistance of the 2 resistors was based on the equation where the was chosen to be 2.2kΩ hence the other resistor was found to be 3.3kΩ by substituting in values as 5V and as 3V the following calculation were done. 3 = 5\*/2.2k+, 3(2.2k+)=5, 6.6k=5-3, =3kΩ.

The next step was updating the frequency finding algorithm to determine the frequency as 3V contains 5 octaves. This meant that each 0.05V differentiated a different frequency. Hence if statement tree was expanded to differentiate between each 0.05V which was implemented and tested before a better solution was found. The better solution divided the adc\_value by 3 to create a fraction that represents its relative size to the max voltage and then multiplied by 60 to find the position in the array of the frequency. This meant that the frequency could be found by assigning this value to the subscript of frequencyList array.

This 5V solution caused some issues because when the frequency got higher than 1 octave then decreased below 1.0f and the capacitor got overloaded with current. Hence the solution had to be modified to fix this. This was mitigated by ensuring that the base frequency divided by the highest frequency and multiplied by the base clock divider was positive. This required changing the place of the base frequency in the array so that the base frequency was in the 3rd/4th octave, the wrap was also decreased to help increase the clock divider, but it then wasn’t producing enough current to fill up the capacitors in the low frequencies. Eventually the clock frequency was increased to 216Mhz which allowed for slightly less high frequency squealing but there was still an issue with the low notes not producing enough current. Once that was corrected by increasing the wrap or reducing the Base frequency to a lower octave, the higher notes did not balance well.

Hence the C program was then modified to repeat values just 1 time. This meant also changing the clock divider calculation to account for the 8 times increase in frequency. This had performed better than the previous methods and allowed for an increased wrap while still achieving good high frequency fidelity. The lowest notes still didn’t have enough current though.

The C program was later modified so that the base frequencies Octave’s wrap values were repeated 16 times in the lowest octave, 8 times in the 2nd Octave, 4 times in the 3rd Octave, twice in the 4th Octave and only once in the 5th Octave. The change clock divider function was then updated to check which Octave it’s in a return the correct clock divider, adjusted for the number of repeated values it has. The base frequency then became the perfect 1st in the 3rd Octave. The find clock divider function was then updated so that if the new frequency is in the 5th octave, then it multiplies the new clock divider by 4 since it is repeating 4times less than the base frequency. Then we include a few else if statements to justify the clock dividers hence the frequencies in the 4th octave are multiplied by 2, the frequencies in the 1st octave are divided by 4 and the frequencies in the 2nd octave and divided by to. Then an else was added for frequencies in the 3rd octave. This solution meant that the clock divider’s lowest value will be 8/15 times the clock divider for just tuning since the highest frequency in an octave is the Major 7th which is 15/8 the size of the perfect 1st and this method means that each perfect 1st of an octave will have the same clock divider. This is because the frequency of the output is reduced when the duty cycle input is repeated.

## Vibrato

The vibrato effect has the frequency of the output modulating between a semitone down and a semi tone up. The solution developed for this project has the frequency updating after every wave and changes at a rate of 1/24 times the semitone up subtracted by the semitone down from the current note being played.

This was achieved by input method…

Once the input method had been implemented the PWM handler function was then modified with to include an if statement when the wave cycle has completed and the wav\_position is set back to 0. This was modified by adding an if statement that checks if the vibrato Boolean value is true and if so, then modify the frequency. The frequency was modified by creating a temporary global frequency variable called currentF which holds the current frequency be that from the semitone below to the semitone above.

The direction of the vibrato depends on the Vibup Boolean variable that if true means the vibrato increases in frequency at every loop to the upper semitone else it descends to the lower semitone. When the vibup is true the Frequency is checked against the upper semitone frequency and if its less than then it increases by the vibchange variable and if its larger then the vibUp variable becomes false. When the vib up variable is false it then checks if the currentF variable is less than the lower semitone if its larger then it decreases the frequency by the vibchange parameter else the vibUp parameter becomes true.

The upper semitone variable, lower semitone variable and the vibchange variable are all updated when the gate handler gets called and after the new frequency is found so that each value gets updated for the new frequency.

### ADSR

The ADSR stands for Attack, Decay, Sustain and Release and the way it works in the euro rack is that it takes in a Gate input which tells it how long to play a note for and returns a controlled voltage that is used to modulate the volume of the synthesiser output. In this euro rack configuration 2 ADSR modules were added to the rack with one modulating the volume of the LFO and another modulating the amplifier module. The ADSR modulates the volume in 4 sections, attack which is the length for the volume to go from silent to maximum volume, decay which goes from peak of the attack to the sustain level, sustain which is the level that is maintained when the gate is on, and release which is the length of time after the gate has ended. This meant that the A channel GATE-OUT from the sequencer had to be removed from the Pico and the put through one of the ADSR modules and the B channel GATE-OUT put through the other. This required altering the code so that the adc\_read() functionality was put inside the PWM handler so that an the ADC pin’s voltage was read after every cycle through a wave. This solution caused some differentiating frequencies because when the voltage from the SQ-1 was not perfectly sustained throughout a note resulting in dips in frequency throughout the span of a note. This was the result from testing with a speaker and readings from an oscilloscope. Hence a new solution was made that duplicates the gate from the sequencer by adding an additional mono audio jack and wiring it to another mono audio jack. This meant that the Pico was then capable of using the previous GATE input code to read in the CV from the sequencer.

### Other modules

The eurorack setup includes 2 ADSR modules, 1 attenuates the amplifier and the other attenuates the low pass filter.

# Evaluation / Testing

The amplitude field isn’t necessary and could have been subbed in with the value 1 but I found it helped visualise the waveform a bit better. And the option is there to decrease the amplitude without altering the wrap size which could be potentially useful in niche situations. Write about how phase correct mode could have been implemented into the runtime solutions alongside removing the repeating values part.

# Description of the final product

# Appraisal

# Summary and Conclusions

# Future Work

## Acknowledgments

# References

<https://github.com/TuriSc/RP2040-Button>

<https://www.i-programmer.info/programming/hardware/14849-the-pico-in-c-basic-pwm.html>

# Appendices

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Slice | 0 | | 1 | | 2 | | 3 | | 4 | | 5 | | 6 | | 7 | |
| Channel | A | B | A | B | A | B | A | B | A | B | A | B | A | B | A | B |
| GPIO PIN | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| GPIO  PINS | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 | 28 | 29 |  | |