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: Mr. Andrew Cobley**

***Abstract* –**

*The aim of this project is to develop a Voltage controlled Oscillator using a Raspberry Pi Pico and integrate it into a euro rack synthesiser format. The product uses pulse width modulation to create an analogue oscillating voltage using a transformation circuit. This required investigating 2 methodologies for determining the duty cycle of the pulse width modulation circuit, using wave tables to determine the level or calculating the duty cycle at runtime. Further development went into both methods, developing different waveforms, with emphasis on the wave table solution that made its way into the final product. The Product was developed so that Oscillating frequency was determined by a Korg SQ-1 sequencer and integrated within a euro rack format. The product was implemented with an additional Oscillating output and testing concluded that the frequencies that were produced were accurate within ±5% of the desired frequency. The waveforms were also produced accurately and the overall the project was successful.*

# 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 was developed to become a voltage-controlled oscillator and how it was integrated into a euro rack synthesiser format. The outcomes of this project will be a final product that should be an inexpensive commercially viable voltage-controlled oscillator that can change the shape of the output wave form. The methods researched in this project could additionally be integrated into other commercially viable audio synthesis devices for example it could be integrated into a toy keyboard for kids to learn how to play piano or into a synthesiser pedal for electric guitar.

The Raspberry Pi Pico is a microcontroller which can generate digital outputs which means that the output voltage can only be on or off meaning that the only wave form it can produce is square wave. This poses a problem when generating an oscillating voltage with analogue waveforms such as triangle and sinusoidal. Hence a method of converting the digital output into analogue is required. This problem requires pulse width modulation to convert the digital output to analogue. Pulse width modulation is a method where analogue voltage size is determined by modulating the duration of a voltage pulse. The 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 voltage from the capacitor will rise to the voltage of the voltage supply over time, the voltage across the capacitor after a t time period is given by the equation where is the voltage from the Pico, e is the Euler irrational constant ≈ 2.7182, t is the time of the pulse, and RC is the resistance and capacitance of the circuit in ohms and Farads. Hence the resultant voltage from the capacitor that has a magnitude that corresponds to the duration of the pulse. This is used to generate a controlled analogue voltage which can be modulated by changing the duration of the pulses going into the capacitor.

The duration of pulses from the Pico is relative to the rate that pulses are repeated and is referred to as the duty cycle. The period of the repeated rate can then be used to determine the period of the output wave. The oscillating frequency can then be translated into the number of pulses that is required to reach the size of the period of the wave by the formula . 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 duty cycle of the pulses from 0 to x number of pulses.

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

Both methods have their own use cases and limitations. The main benefit of runtime calculated duty cycle is that it is possible to alter the wave forms profile during runtime but is potentially limited to simplistic calculations since large complex calculations could 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 could allow for much more complex waveforms and a more stable frequency although this limits it in its ability to alter the wave profile. This project will investigate how both methods can be used to create unique sounds.

This report will also document how the frequency of the output is controlled by an input voltage. Which will document the development of the Pico to take an input from a Korg SQ-1 sequencer. A sequencer is a device that allows the user to construct sequences of musical notes by sending a control voltage to indicate a desired frequency. 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. This project will document means of changing the frequency of the sound in an accurate way. One method that will be researched in this project is storing different frequencies in an array and looking up the frequency when required. This requires taking a base frequency and modulate the rate that the duty cycles are outputted to change the Frequency of the waveform. This allows for a wide range of frequencies that the system can produce and doesn’t require much memory usage.

# Similar Projects

The has been several similar projects that generate audio using the Raspberry Pi Pico. Turi Scandurra has produced a few of these products, one of which is the Dodepan that is a Lo-fi electronic version of a marimba or handpan that was implemented using the Raspberry Pi Pico and includes features such as multiple tunings, pitch bending and Midi out. Another project that aligns itself with this project is Ben Everard’s Pico Pico synth project that has an built-in sequencer, waveforms and ADSR which is very impressive. Another Project that relates to the outcomes of this project is Rory Allen’s EuroPi project is a fully user reprogrammable module based on the Pico which allows users to process inputs and controls to produce outputs based on python code. There are a number of modules that are available from allensynthesis.co.uk ranging from £28 for just the front panel and PCB to the full DIY Kit for £106

# Background

## Programming the Raspberry Pi Pico in C

To research how PWM is integrated into the Pico in the C programming language, a section of the Book, ‘Programming the Raspberry Pi Pico in C’ By Harry Fairhead was researched. This book contains knowledge of the Pico PWM hardware which is explained in this section.

PWM 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 based on Harry Fairhead’s explanation. (See figure 1.)

A picture containing text, crossword puzzle

Description automatically generated

Figure 1. table showing the slice and channel allocations to the GPIO pins.

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 for a specific slice the pwm\_set\_enabled(uint slice\_num, bool enabled) function can be used or if there are multiple signals that need turned off or on the pwm\_set\_mask\_enabled(uint32\_t mask) function can be used. 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.

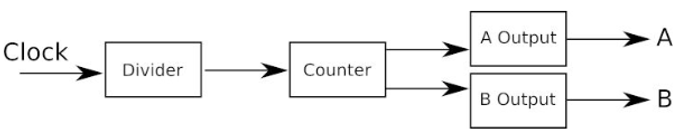


Figure 2. diagram showing a pipeline of the Pico’s PWM circuit

PWM generation in the pico work by iterating the counter at the clock frequency which means that the steps occur every seconds. Increasing 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 that the wrap is set to, 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 the wrap the lower the granularity of the duty cycle. This is due because 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). In the normal mode it is explained that once the counter reaches the wrap value it starts from 0 again. On the other hand, there is a second mode that can be used called “phase-correct” mode which instead counts back down to 0 once the wrap is achieved then up to the count again. The normal mode mimics a sawtooth wave as which can be seen from these diagrams from the documentation. (See figure 3.)

Diagram, line chart

Description automatically generated

Figure 3. Diagram of counter increasing in non-phase shift mode

And the Phase correct mode is a triangular wave. (See figure 4.)

Diagram, line chart

Description automatically generated

Figure 4. Diagram of counter increasing in phase shift mode

This means that in phase shift mode the frequency is halved, 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.

The book also explains that to set the duty cycle of the pwm output the channel’s level must be set. 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 both channels can be set 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 method is a helper function pwm\_set\_GPIO\_level(uint GPIO, uint16\_t level) which will set the level on the specified GPIO pin.

The Book also 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 the new clock frequency can be found by dividing by the integer, then dividing by fract/16.

## Circuit research

Research into a transformation circuit by looking into the “Hardware design with RP2040” Datasheet which contains a circuit diagram for use with a PWM audio source. (see figure 5.)

Diagram, schematic

Description automatically generated

Figure 5. Circuit Diagram showing stereo PWM output.

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 voltage inside the capacitors, the longer the pulse the higher the voltage released from the capacitors up to the IOVDD voltage.

## Robin Grosset’s pico-pwm-audio project

Another place of research was Robin Grosset’s pico-pwm-audio project which contains an example of PWM using table look up. Grosset’s github repository includes a C program that generates a PWM output when compiled into a UF2 file using the raspberry Pi Pico C SDK. The C program utilises the Pico standard library and the interrupt, PWM and sync library to accomplish PWM output. 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 based on the GPIO PIN. It then clears the PWM channel interrupt on the PIN, then Enables PWM interrupts so that it interrupts after the wrap has completed 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 used to repeat each value for 2^3 cycles. It then sets the level on the pin from the WAV\_DATA at wav\_position array, bit shifted to compensate for the 2^3 extra cycles and iterates the wav\_position up. Then it includes an else to reset the wav\_position to start once the end of the array has been achieved.

In the final part of the main grosset sets a pwm\_config to the default config provided by the hardware/pwm.h file and sets the clock divider to 8.0. It then sets the wrap to 250, then then initialises the pin with the updated 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.

## Jake Rosoman’s pico-button.c and pico-gpio-interrupt.c

Another place of research was Jake Rosoman’s pico-button.c and pico-gpio-interrupt.c which both have MIT licenses. These C programs provides helper structs to initialise multiple button inputs and handlers for interrupts.

The pico-gpio-interrupt.c program first defines a pointer to a handler function type with a pointer to argument as a parameter. Then it defines a closure\_t struct which is holds handler information for each GPIO pin. It contains 2 members, a pointer called argument and a handler function called fn using the pointer-to-function type defined above. Rosoman then defines a 28-item array of closure\_t structs for each gpio which is set initially null. Then it defines a handle\_interupt function that is called when an interrupt is created hence is contains uint gpio and uint32\_t events parameter. In this handler the gpio pin is used to find the closure\_t item from the array defined above and then uses it to call the fn function of the struct using the structs argument member as a parameter. Finally, a listen function is created which takes in a pin number, condition (which is the event that causes an interrupt), a fn pointer-to-function handler type, and a pointer to an argument. This function initialises the gpio pin and pulls up the pin, then enables interrupts with callback for the pin using the condition parameter to set the event and set handle\_interupt function as the callback function. Then a closure\_t object is created with the fn and argument members set using the parameters of the function. Then the closure\_t array is updated so that the gpio pin number’s item in the array is set to a pointer to the newly created closure\_t structure.

The button.c program includes the gpio-interrupt.c program and uses the listen function to initialise the button pin. The program first defines a struct called button\_t which has 3 members, a uint8\_t member for the button pin, a bool member for the state of the button which is used to determine if the state of a button has changed in the handle button alarm function and a pointer to a function with parameter of a button\_t structure. Next the program defines a function that called handle\_button\_alarm which takes in a void pointer p which is then converted into a button\_t pointer which calls the onchange function if the button state has changed. This is used to add an alarm so that the button’s onchange function is delayed 200 microseconds. Then a create\_button function is added that returns a pointer to a newly allocated button\_t structure. The function takes the pin number and a pointer to the onchange handler function as parameters. Then allocates a button\_t structure in memory. Then it uses the listen() function to initialise the button pin with the RISE and FALL events as well as the handle\_button\_interrupt function, which is set to the call back function. It also sets the argument parameter as the newly allocated button\_t structure so that the handler\_button\_interupt function uses it as the parameter. Then the struct members of the structure are initialised with the parameters. In the button.c repository it contains an example.c program which shows how this can then be implemented into a C program, by including Button.c, defining a onchange function and initialising a button\_t struct pointer with the create\_button function with the pin number and onchange function as parameters.

# Specification

The final product of this project must use the Raspberry Pi Pico to produce an oscillating voltage using PWM and the frequency of the output must be externally controlled by a controlled voltage. The product must be integrated with a Korg SQ-1 pitch sequencer which requires that it uses the Korg SQ-1’s CV A to control the frequency and could use the GATE-OUT to determine when the frequency is to be read. The product must use the 1V output option from the SQ-1 to modulate the frequency in a 1-octave range. The product should be able to use the 2V output option from the SQ-1 to modulate the frequency in a 2-octave range. The product could be able to use the 5V output option from the SQ-1 to modulate the frequency in a 5-octave range. The product must be integrated into a Euro Rack synthesiser format. This may be integrated using 2 ADSR modules attenuating the volume of a voltage controlled low pass filter and amplifier modules. The product should be able to output different waveforms which could be Sine waves, square waves, triangle waves and parabola. The product should also have button inputs so that the different Wave Form options can be selected. Additionally, the product could combine waveforms in runtime this could be done with different notes to make chords or add harmonics to simulate plucked notes. The product could additionally implement vibrato functionality that modulates the frequency up and down a semitone. The product could integrate another oscillating voltage using PWM, the frequency of which could be determined by the CV-B output from the Korg SQ-1. This could also lead to a secondary euro rack integration with a ADSR and amplifier modules. The Secondary could have a secondary waveform selection allowing for independent control of wave form for each output.

# Design

## Language for Pico

From the outset of the project the Coding language that was chosen was C. This was chosen because C is much faster than the other option, micro-python because micro-python is converted into C then compiled, the result is much slower than compiling the C code into the UF2 binary.

## SDK

The SDK used to compile the code was the Raspberry Pi Pico C/C++ SDK for Linux operating system. An Ubuntu VM was used for the SDK because the windows version seemed to have a lot less support and the installation process was a lot more streamlined. This SDK was used due to its integration of the Pico standard libraries and its CMake compiler’s ability to compile C programs into UF2 files that are used on the Pico.

## Circuit design

A tool called circuit-diagram.org was used to design the circuit diagram due to its built in Raspberry Pi Pico component. The PWM circuit was inspired by the diagram from the “Hardware design with RP2040” Datasheet by Raspberry Pi. The input buttons were allocated to GPIO pins 0-13, allowing for all the inputs to be along the same side with the CV and gate inputs on the opposite side which allowed for a separation between the user inputs and the external inputs. The PWM circuit was designed to use GPIO 16 and 18 due to them use separate slices but are both located at the bottom of the Pico which allowed the PWM output to be out of the way of the inputs. (see figure 6.)

Diagram, schematic

Description automatically generated

Figure 6. Circuit diagram showing the inputs and PWM outputs.

## Python Script

A script was required to create a header file based on some configuration settings that are inputted at runtime. Python was the chosen language for the script due to the developer’s technical proficiency in python. Python also suited the mathematically technical scripting involved with generating the arrays that were placed in the header file.

# Implementation and Testing

## Overview

Implementation started with creating a working circuit that takes a PWM output from the Raspberry Pi Pico and output an oscillating analogue voltage. The next step 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 specifications requirement was to take an input voltage and convert it into a digital value so that it could be used to determine the frequency of the output.

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

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

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

After the waveforms had been fully explored the next step in implementation was to take the prototype and integrate it into a euro rack synthesiser.

This first involved using a Korg SQ1 sequencer to modulat the input voltage. This step also involved modifying the prototype to convert the voltage into an octave per volt format. This was completed by modifying the prototype to accept a 5V signal that allowed the Pico’s output frequency to span 5 Octaves and accepting a gate input which tells the Pico when to play notes. The next step was then to integrate a voltage-controlled low pass filter and a voltage-controlled amplifier module. This also required integrating 2 ASDR module which allowed for modulation of the attack, decay, sustain and release of the output by modulating the level of the low pass filter and the amplifier independently.

The next developmental goal that was achieved was implementing a method of generating vibrato where the frequency alternates up and down by half step intervals. This meant that the frequency had to increase up a semitone then down to the original frequency and then down a semitone then back to the frequency and then repeat.

Once the Pico was fully integrated with the Euro rack modules the final stage was to integrate a second PWM signal. This involved taking integrating another ADC input and another gate input as well as another PWM circuit. This was also integrated within the euro rack and the waveforms were integrated so that each signal’s waveforms could be altered independently.

## PWM Circuit

The first step of implementation was creating a circuit that took a PWM output from the Pico and transform it into an oscillating voltage. The circuit for this first prototype was designed after the “Hardware design with RP2040” data sheet. (see figure 7.)

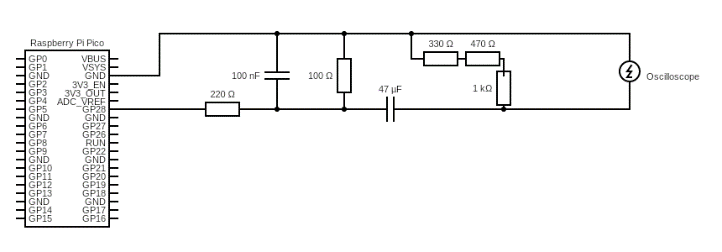


Figure 7. circuit diagram of the first PWM circuit

This implementation used just the GPIO 28 hence the circuit was cut down for a single output. Instead of using an Audio Jack the output went straight into an oscilloscope. It also used of a 330Ω, 470Ω and 1KΩ resistor in parallel was used 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 and tested.

## Accurate Frequency

After the circuit was established, the next step was to generate a single frequency which was generated so that it produced a specific frequency and tested to ensure the frequency was generated with the expected frequency. This was achieved by first forking the code developed by RGrosset and removing 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 header 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 was written to the new header file which was used to modify the duty cycle of the PWM output.

The first iteration of the python program generated duty cycles for a single sinusoidal wave and converted into level values by multiplying by the wrap. This was achieved by defining a variable for the period of the wave equal to 1/Frequency then create a multiplier variable set to 2 times pi over the period of the wave, which was referred to as B in the program, this variable, B, was used to find the duty cycle which equalled the amplitude at each point along waves cycle. When the wave was reproduced, pulses are generated at a certain rate called the repeated rate hence it was also of use to find the interval size (1/repeated rate) the values were repeated 8 times so this value was divided by 8 to compensate. The interval size was used to iterate until it had reached the size of the wave’s period. At each iteration the duty cycle at the point was found by using a sine function of the variable B multiplied by the size of the x number of intervals. With the duty cycle at that point found then the level value was calculated so that the Pico produces the correct duty cycle. At this point the sine wave function alternates between -1 and 1 so this was converted to a positive fraction, less than 1, and multiplied by the wrap. Which required addition of one amplitude to make it positive then divided by 2 amplitudes to make it a fraction. The data was then written as a C array to a header file which was included in the C program using #include.

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, 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 connected to 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 (see figure 8.)

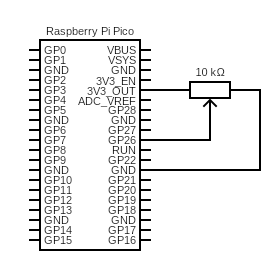


Figure 8. potentiometer connected to the Pico for ADC read

### 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. The adc\_select(0) was used to select the GPIO 26 PIN(ADC 0) for the read. A constant was then set 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 the adc\_value was defined to equal the adc\_read() value multiplied by the conversion factor each time the wav\_position had reached the end of the array.

### Altering the frequency

The adc\_value variable was then used to alter the frequency of the output. This was done by altering the clock divider which meant the core frequency was divided which modulates the operation of the clock speed. The original clock divider used was 2.0 which meant the 176Mhz core clock frequency was divided by 2 which created an operational frequency of 88Mhz which meant that using the formula ,with a wrap of 25, the repeated rate was 352,000Hz and with the levels repeated 8 times the rate at which samples were cycled through was 44,000Hz. When the operational was modulated by changing the clock divider then the rate at which the values were outputted from the Pico changed translates to the frequency of the analogue output changing. The next step was to create a function that returns the clock divider necessary to generate a frequency. This was implemented by taking the new frequency as a float parameter and using 2 globally defined constants called WAV\_FREQUENCY which was the frequency of the output with the original clock divider and the clkDiv constant which was the original clock divider. The function divided the WAV\_FREQUENCY constant by the new frequency to create a fraction of the difference between the frequencies, then multiplied the fraction by clkDiv to give the clock divider needed to achieve the new frequency. This value was 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 new frequency so that the frequency increased as the potentiometer was turned. Then the find\_clkdiv function was used 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 was put into the global scope so it could be accessed in the handler function, and the new clock divider. (see figure 9.)

Text

Description automatically generated

Figure 9. Code to alter the output frequency based on the adc\_read() value

### Testing

This was then tested using an oscilloscope by turning the potentiometer knob around and checking if the frequency was altered. The oscilloscope was used 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 the oscilloscope was attached 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 level values representative of the duty cycle at points in the waves cycle and 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 had 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 were 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. (See figure 10.)

Graphical user interface, text, application

Description automatically generated

Figure 10. Wave class definition and constructor.

### Square Wave

The square wave was implemented using a while loop that iterated using a variable called x which increased by 1 interval each loop until it had reached half of the period length + one interval. Before the loop a variable called vals was set to equal the wrap field which was converted into a string. Inside the while loop, the ‘vals’ variable was concatenated with ‘, ’ then concatenated with the wrap field, converted to a string. Another while loop was then implemented which again iterated through using x but instead looped until it equalled the period + 1 interval. The while loop similarly concatenated ‘, ’ to the ‘vals’ variable but instead concatenated “0” after. Then the vals variable was returned from the method. (See figure 11.)

Text

Description automatically generated

Figure 11. python method to generate string of level values for square wave

### Triangle Wave

Similarly, the triangle wave implementation was implemented by looping through in different fractions of the wave. The method started with defining the tan\_theta variable which was used to find the duty cycle at each point along the period of the waveform. This was set to 4 times the amplitude field divided by the period field because the triangular wave in this implantation was split into 4 triangles with a height of 1 amplitude with a width of ¼ of the period of the wave. The quarter wave implementation was used so that the wave started with a duty cycle of 50% just like the sinusoidal waveform. A variable named x was used to iterate through the wave again and was used set the first value which was x multiplied by tan\_theta and then the amplitude which was added and then divided by 2 amplitudes to give the duty cycle which was set to variable v. Then the vals string was initialised by taking the v variable and multiplying it by the wrap to find the equivalent level. Then a while loop was used to iterate through the first quarter of the waves period using x which added the values, calculated using tan theta, to the string. Then it iterated through the next 2 quarters of the period again using x. In this loop x times tan\_theta was used to subtract from 3 times the amplitude and divided by 2 times the amplitude to give the duty cycle of the descending part of the wave then multiplied by the wrap to give the level at each point.

The final while loop iterated through the last quarter of the period again using x. Where it finds the value of x times theta and justifies it to give the duty cycle by subtracting three amplitudes then dividing it by 2 amplitudes. Then it returned the vals string containing the levels, concatenated as strings, and separated by commas. (See figure 12.)

Text

Description automatically generated

Figure 12. python method to generate string of level values for triangle wave.

### Sawtooth Wave

The sawtooth wave was implemented much in the same way as the triangle wave, instead of using the 4 subsections instead just 2 sections were used which meant that the tan\_theta value changed to 2 times the amplitude divided by the period field. This solution iterated the variable x from 0 to half of the period + interval. then multiplied the value x by the tan theta to get the value and then concatenated the string with the value plus amplitude divided by 2 amplitudes, multiplied by the wrap to give the level as a string. Then another while loop was added to iterate the x variable further until it reaches the period plus the interval that found the value but instead concatenated the value minus the amplitude divided by 2 times the amplitude. This meant that the value was justified to a 0% duty cycle. Then it returns the string of the values separated by commas. (See figure 13.)

Text

Description automatically generated

Figure 13. python method to generate string of level values for sawtooth wave.

#### Reverse Sawtooth Wave

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

### Parabolic Wave

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 was then squared the value decreases exponentially until zero. Then from 0 to 1 it increases exponentially. The value was then multiplied by the wrap and concatenated to the vals string and returned from the method. (see figure 14.)

Graphical user interface, text, application

Description automatically generated

Figure 14. python method to generate string of level values for parabolic wave.

### 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 the variable was used to find the values of the harmonic’s sine wave input array and concatenated it to a string then added the string to an array of input array strings and then returns the array with the values from each of the harmonics. (See figure 15.)

Graphical user interface, text, application

Description automatically generated

Figure 15. python method to generate strings of level values for a series of harmonic waves.

### Writing to the header

Once the python program was modified to create a new wave class the fields were found using pythons ‘input()’ method. These values were then used to initialise the wave object which was then used to define variables for each waveform by calling the specific generate waveform method. Python’s open() method was used to open the header file which required assigning a variable f as the file object and used to call the write() method to write to the header. The first items to write to the header file were 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 the int value from the input method was converted into a string and written to the file, then ‘\n’ was written 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 repeated rate by the frequency. Then a constant of the clock divider was found by dividing 88,000 initially which 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). This was adapted to calculate it from an additional input for the clock speed. The waveform input array strings was 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 a 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". (See figure 16.)

Text

Description automatically generated

Text

Description automatically generated

Figure 16. python method to write data to the header file.

### Testing/Input method

To test the wave forms a button was connected to GPIO PIN 10 and the 3V3OUT PIN meaning that when the button was 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 initialise 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 was raised up. The handler function was then assigned to the pin using gpio\_add\_raw\_irq\_handler\_masked(( 0x01 << WAVEBUTTON),&rawHandler1) function which takes in a bit mask of the integer value of the pin which was converted in this case by bitshifting the 0x01 value up by the magnitude of the WAVEBUTTON macro. Then the memory address of the rawhandler was set as the parameter. (See figure 17.)

Text

Description automatically generated

Figure 17. Initialisation of Wave change button

The raw handler first checks the event mask of the interrupt were from the WAVEBUTTON pin and was from a rise event. If so then it acknowledged the interrupt and iterates an integer variable declared in the global scope called button between 0 and 5. This variable was then used in the pwm interrupt handler function in a switch case that has 6 cases for each value 0 to 5, each corresponded to a waveform, the switch case was situated in the PWM handler where the level was set. In each case of the switch statement the pwm\_set\_gpio\_level was set to equal the corresponding input array value at the given wave position.

This means that when the button was pressed it changes the input array and subsequently, the wave form that was produced. This was then tested using an oscilloscope to ensure that the frequency of waves stayed constant, and that the peak 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. (See figure 18/19.)

Text

Description automatically generated

Figure 18. The if statement situated in the Wave buttons raw handler.

A picture containing text

Description automatically generated

Figure 19. The switch case situated in the PWM handler.

### Button.c Integration

The final solution was implemented using Jake Rosoman’s pico-button.c and gpio-interrupt.c code from jkroso’s github repositories. To implement his methods, the code from these files were integrated into a single c program named button.c which was included in the pico-pwm-audio.c program. The code from the raw handler was removed and the code from the main which initialised the button was removed. The circuit was also modified to include 9 buttons with one side of the buttons connected to ground and the other side connected to GPIO pins. 5 buttons corresponded to the 5 waveform, sine, square, triangle, sawtooth/reverse sawtooth and parabola and the other 4 were to add and subtract harmonics (even and odd separately). (see Figure 20.)Diagram, schematic

Description automatically generated

Figure 20. circuit diagram of buttons integrated with pico

To integrate them in the C program macros were defined for each of the GPIO pins numbers that the buttons were connected to and a void function was added that takes a button\_t struct pointer as a parameter. Then a local button\_t object was then created with the use of this pointer. The local button object was then used in a switch statement to determine the pin number using its pin member. 9 cases were added to the switch for each of the button’s macros and the buttonNum for the five waveforms were assigned under the corresponding case. For the even and odd harmonic’s up and down buttons 2 new global integer variables were added called evenHarmonics and oddHarmonics which contain the number of harmonics to be added to the waveform at runtime. This meant that inside the cases for the harmonic even up the evenHarmonics integer was increased by one until it reaches 6 and for the harmonics down the evenHarmonics variable was decreased until 0. Similar logic was used for oddHarmonics but only iterates to 5 harmonics. (see Figure 21./22)

Text

Description automatically generated

Figure 21. Initialisation of buttons using button\_t struct.

Text

Description automatically generated

Figure 22. The switch case in the onchange function

## Runtime solutions

### Overview

The runtime solutions were implemented using a wavelength variable of a frequency which isn’t the distance wavelength but instead was 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 was 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. (see figure 23.)

Text

Description automatically generated

Figure 23. PWM handler for Runtime square wave solution with duty cycle modulation

### 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 was 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.(see figure 24.)

Text

Description automatically generated

Figure 24. PWM handler for Runtime triangle wave solution with duty cycle modulation

### 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 was exponentially rising until the end of the loop. The value was then multiplied it by the wrap to give the level for the GPIO. (see figure 25.)

Text

Description automatically generated

Figure 25. PWM handler for Runtime parabola wave solution with frequency alteration

### 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 was the length that the square wave was 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() was called, then that value modified 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 was 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 was 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 was 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 musical frequency 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 was one quarter the size of the base frequency. Then the frequencies were found for the following 5 octaves and written to the file as a constant integer array. (see figure 26.)

Text

Description automatically generated with low confidence

Figure 26. python function to add the semitone’s frequency of an octave to a file in array format.

### 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 was 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 was 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 was 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 was off or on. This meant updating the raw handler for the gate to have an additional if statement to check if the interrupt was from the gate and if the event was 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 was true. This was then tested through the speaker to ensure that output occurs when the gate was not active. (See figure 27.)

Graphical user interface, text

Description automatically generated

Figure 27. Final handler function for gate functionality.

### 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 was in the 5th octave, then it multiplies the new clock divider by 4 since it was repeating 4 times less than the base frequency. Then a few else if statements were included to justify the clock dividers hence the frequencies in the 4th octave were multiplied by 2, the frequencies in the 1st octave were divided by 4 and the frequencies in the 2nd octave were divided by 2. Then an else was added for frequencies in the 3rd octave. This solution meant that the clock divider’s lowest value was 8/15 times the clock divider for just tuning since the highest frequency in an octave was the Major 7th which is 15/8 the size of the perfect 1st in just tuning and this method meant that each perfect 1st of an octave will have the same clock divider. This was because the frequency of the output was reduced when the duty cycle input was repeated. The final solution achieves this using a val global variable to store the bit shift that was used to increase the data length and reduce the wave position so that values get repeated 2^val number of times. The change clock divider function was then updated with the val variable when a frequency change was made. (See figure 28. &29.)

Text

Description automatically generated

Figure 28. PWM handler using val to increase the number of repeated values.

Text

Description automatically generated

Figure 29. Final update clock divider function which determines the number bit shift value and clock divider for each octave

## 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 that means that from the lower semi tone to the upper semitone it takes roughly 1/3 of a second for each note. This was achieved by setting the rate to the upper semitone frequency subtracted by the lower and divided by the current frequency/3.

A button was added to the circuit in the same set up as the Wave shape button this also meant adding a case to the switch statement of the onchange function to change the vibrato variable from true to false and false to true.

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 was set back to 0. This was modified by adding an if statement that checks if the vibrato Boolean value was 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 was true the Frequency was checked against the upper semitone frequency and if its less than then it increases by the vibchange variable and if its larger than the vibUp variable becomes false. When the vib up variable was false it then checks if the currentF variable was 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 was found so that each value gets updated for the new frequency. (See figure 30. &31.)

Graphical user interface, text, application

Description automatically generated

Figure 30. Case in onchange’s switch statement for Vibrato pin.

Graphical user interface, text, application

Description automatically generated

Figure 31. if statement in the PWM handler which handles vibrato after each wave

## Euro Rack integration

### ADSR

The ADSR stands for Attack, Decay, Sustain and Release and the way it works in the euro rack was that it takes in a Gate input which tells it how long to play a note for and returns a controlled voltage that was 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 was 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 was the level that was maintained when the gate is on, and release which was 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.

## Dual PWM implementation

### Overview

The second pwm output was set up on a separate slice and was implemented with a separate set of variables for the slice number, pin channel, wav\_position, vibrato variables, buttonNum, even harmonics, odd harmonics and val(for bit shifting in pwm handler). These were implemented into the program by first adding an if statement using a bitwise and operator to check if a variable irq set to interrupt status mask was equal to 1 bit shifted up by the slice number that was checked. The original code was then copied and put in the else if part of the statement that check for the second slice number. (See Figure 32.)Graphical user interface, text, application

Description automatically generated

Figure 32. PWM handler section for the second PWM.

### Second GATE-OUT

To take in a second GATE, secondary gate monojack was added and connected to GPIO 21. In the C program a macro was defined for the pin input and set up in the same way as the first GATE pin. This also meant adding an if statement to the raw handler that checks for GATE2 then the functionality from the first was copied into it and the new global variables were interchanged. (See figure 33.)

Text

Description automatically generated

Figure 33. Section in raw handler for the Second gate

### Second ADC

To achieve 2 ADC inputs another mono jack was added to the circuit with the same resistor configuration as in section 5.7.2. Then an additional macro was defined for the second ADC pin, this required changing the PWM pin to GPIO 18 and moving the original ADC pin to GPIO 28 and setting the second ADC to GPIO 27. The circuit was then updated to account for the changes. Then the Gate handler was updated so that it selects the correct ADC pin before the read for both GATEs so that ADC 2 was selected in gate 1 and ADC1 was selected for gate 2.

### Second PWM setup

The main was updated to include a secondary pwm initialisation for the new PWM pin. Which meant initialising the PWM config with clock divider and wrap then initialising the pin. (See figure 34.)

Text, letter

Description automatically generated

Figure 34. initialisation of secondary PWM pin

## Python script final version

The final version of the python was renamed to configuration.py and the header file name was changed to configuration.h. It was also adapted to contain logic to ask for base frequency, repeated rate, wrap and clock speed values as an input then checks to ensure that the values are valid. It tests these values by first checking if the clock speed was within the range of 125-220 Mhz and validates that the repeated rate, wrap and frequency are valid by ensuring the base clock divider was sufficiently to allow for the frequency to be modulated up to the highest value and down to the lowest value.(see Figure 35.)

Text

Description automatically generated

Figure 35. Input section of configuration.py to validate inputs.

# Evaluation / Testing

The first implementations of the accurate frequency showed a disparity of the frequency output through the oscilloscope due to a misunderstanding of the clock divider and wrap function in the PWM output. The mistakes were corrected, and the frequency output was read to be roughly accurate to the intended frequency, but it was difficult to read an accurate frequency with the oscilloscope as it had an inaccuracy of ±5% Hz.

Testing of the accurate frequency showed that if the repeated rate of the PWM output was set to around 11kHz then visible plateaus throughout the wave form but when increased past 20kHz the effect was removed. See figure 36..&37.)

A picture containing text, wall, clock, indoor

Description automatically generated

Figure 36. sine wave produced using 11khz repeated rate.

A screenshot of a computer

Description automatically generated with low confidence

Figure 37. sine wave produced using 352khz repeated rate.

The results from the testing wave form produced with wave tables showed that the square wave slopes a bit off instead of cutting straight to zero due to the capacitor taking time to discharge. The triangle waveforms were accurate with clear lines with a little rounding at the peaks of the triangle. The sawtooth, like the square wave has a slope off after the peak of the wave but was much less prominent and drops faster than the square wave. The parabola wave looked as expected. The results from adding the harmonic waveforms at runtime looked as expected. (see figure 38,39,40,41,42,43)

A picture containing text, indoor

Description automatically generated

Figure 38. Result of square wave test.

A picture containing text, wall

Description automatically generated

Figure 39. Result of triangle wave test.

A picture containing wall, indoor, clock, mounted

Description automatically generated

Figure 40. Result of sawtooth wave test.

A picture containing text, monitor

Description automatically generated

Figure 41. Result of porabola wave test.

A screen with writing on it

Description automatically generated with medium confidence

Figure 42. Result of harmonic wave test with 1 odd and 1 even.

A screen with writing on it

Description automatically generated with low confidence

Figure 43. Result of harmonic wave test with 5 even harmonics.

A picture containing text, wall, indoor, display

Description automatically generated

Figure 44. Result of harmonic wave test with 5 odd harmonics.

The Results from the runtime solutions were surprising since it the addition of the calculation didn’t seem to affect the wave shape or cause any delays in the output wave form even with the addition of the adc\_read() after every wave. The wave shape of the wave forms was as expected and the changes to the pulse width parameter had the expected result. Even with the parabola waves no delay was observed which shows that the Pico may be capable of doing even larger calculation.

The results of the vibrato test showed the wave on the oscilloscope reducing the distance between the peaks as expected (see appendices section Vibrato\_Video)

# Description of the final product

The final product uses the Raspberry Pi Pico to produce an oscillating voltage using PWM with the frequency of the output externally controlled by the Korg SQ-1 pitch sequencer using the SQ-1’s CV A to control the frequency and uses the GATE-OUT to determine when the CV-A voltage is to be read. The product can use the 1V,2V,5V output options from the SQ-1 to modulate the frequency in a 1,2 and 5-octave ranges. The product has been integrated into a Euro Rack synthesiser format. This was integrated using 2 ADSR modules attenuating the volume of a voltage controlled low pass filter and amplifier modules. The product can output different waveforms which are Sine waves, square waves, triangle waves and parabola waves. The product uses multiple button inputs to select the different Wave Form options. The product can combine waveforms in runtime with harmonics to simulate plucked notes with inputs to add and subtract even and odd harmonics seperately. The product has an input to enable vibrato functionality that modulates the frequency up and down a semitone with an interval set so that it modulates at from each extreme every 1/3 of a second. The product has an additional oscillating voltage output using PWM, the frequency of which is determined by the CV-B output from the Korg SQ-1. This was integrated into the euro rack format by adding additional ADSR and amplifier modules. The Secondary has a secondary waveform selection allowing for independent control of wave form for each output with an additional input that switches the buttons to alter the secondary output waveform and vibrato.

# Appraisal

A lot of effort was wasted during this project from not fully understanding how PWM with the Pico worked but once that had a better understanding of the methods was acquired the project moved along more smoothly. There was a lot of time and effort put towards the wave table solutions and a lot less time on the runtime calculation. Someone were to attempt this project then more time should be spent looking into those solutions and implementing them into the final product.

A lot of effort was wasted was with not fully understanding the full scope of the project due to poor planning so that at the start some efforts were made that didn’t end up move the project along like when Ross Grosset’s tool for turning .wav files into wave tables was used to create a series of note wave tables from samples from the internet but this ended up being worthless to the project after the python program was developed and clock divider solution was implemented. While implementing multiple buttons, a lot of time was spent trying to implement a solution that was replaced using Jake Rosoman’s solution due to of a lack of knowledge of the C programming language, more specifically not knowing that pointers to functions could be members of structs and not understanding some of the uses of pointer that Jake Rosoman used.

It would also have also been beneficial to design the circuit near the start that reflects the outcomes of the project so that a PCB and front panel could have been created.

One thing that was really helpful was using an amplitude field while creating the python script since it made it a lot easier to visualise the wave form being generated. A lot more effort should have gone into implementing the runtime waveforms because they had a lot more potential for generating equally high fidelity low and high frequency waveforms due to its ability to increase the number of accurate levels that are used at runtime.

# Summary and Conclusions

In summary this report shows how pulse width modulation was in the development of an Oscillating voltage generated by the Raspberry Pi Pico. It contains reference to the methods that were implemented from the pico hardware library documentation. This report also documents the specification of the product and shows what design choices were made to fufil the requirements. The report contains a full walk through of all the steps that lead to the final products implementation from generating the first controlled frequency to then generating different wave shapes using the wave table method to read values from an array that specifies a series of level values that produce a specific wave form. Then it details the development of runtime solutions that calculate the level of the pwm output at runtime. The report details the integration of the prototype with the Korg SQ-1 sequencer and the development of 2V and 5V solutions. Then the development of vibrato functionality was documented aswell as the integration of an additional PWM output.

The report details the results of the tests that were carried out during the implementation.

# Future Work

Although the project meets all the requirements set out for this project, a lot more features could be added. One of which could be implementing an additional ADC component that is controlled by a potentiometer to alter the rate of change of the vibrato. This ADC could also be used to change the sample rate of the output which could produce a lower fidelity effect that could be useful in certain musical contexts. The runtime methods that were researched in this project could be integrated into the final product and the additional ADC could alter runtime wave forms. The product could also use the Additional ADC input modulate the level of harmonics, changing the representation of the harmonics in the output, this would also require an additional input to change the wave form buttons to select each harmonic then the potentiometer could change the level of the selected harmonic.

To make the product commercially viable a PCB would have to be created for the circuitry. Once a PCB is integrated then it could be integrated as a eurorack module by creating a front panel design, with room for the buttons and mono jack outputs. It would also need the buttons to be changed for larger ones that would be able to stick out of the front panel. It would also require attaching the PCB to the front panel in a way that the PCB conforms to the front panel. A method for adapting the euro rack power connector so that it could power the Pico would also be required this would be done by taking the 5V wire from the euro rack power cable and connecting it to pin 39 (VSYS) on the Pico and connecting the ground wire from the cable to pin 38 (GND). It would probably mean changing the circuit to use another ground pin for the CV A input as well. The IOVDD voltage could be increased which could allow for the higher notes to be represented more clearly.

## Acknowledgments

I would like to acknowledge Ross Grosset’s pico-pwm-audio project which supplied some of the foundational code that was used to develop the product. I would also like to acknowledge Jake Rosoman’s pico-button and pico-gpio-interrupt project which supplied the code in button.c which allowed for the integration of multiple buttons into the product.

# References

<https://github.com/jkroso/pico-button.c>

<https://github.com/jkroso/pico-gpio-interrupt.c>

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

# <https://datasheets.raspberrypi.com/pico/raspberry-pi-pico-c-sdk.pdf>

<https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf>

# Appendices