ECE 4760 Lab 1 Synthesizing Birdsongs using RP2040

Ridhwan Ahmed and Nikita Dolgopolov

Lab Group 2

February 19th, 2025

High Level Overview:

The goal of Lab 1 Synthesizing Birdsong project is to build a system that can play the birdsong of Northern Cardinal - specifically, two of its parts. The system is also ought to be able to record a sequence of sounds for playback.

The hardware in use for this project are RP2040 microcontroller, a reset button, a Digital-to-Analog Converter (DAC), a speaker, and a keypad with relevant circuitry and wiring.

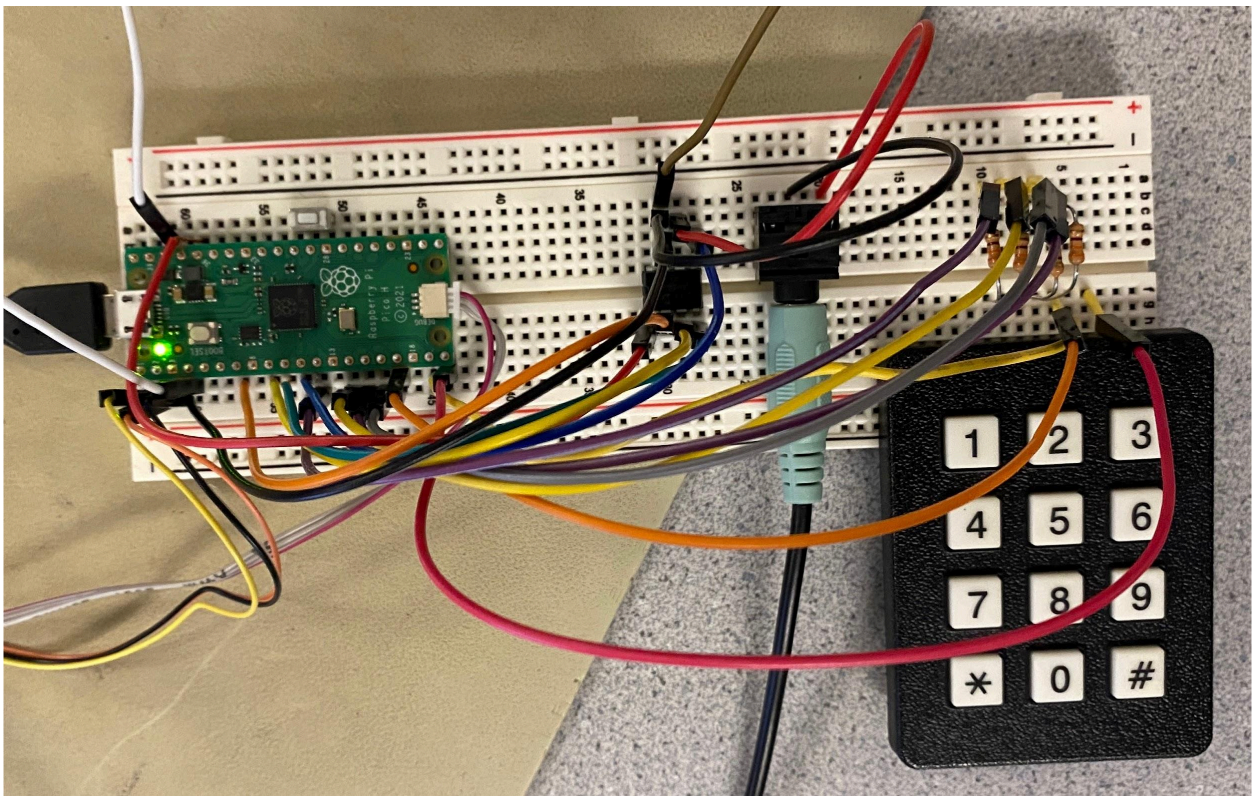

Figure 1: Actual Tested Circuit

The Software side of the project includes the following blocks: DAC communication via SPI, debouncing logic for the keypad key presses and masking, timing and ISR, use of multiple photothreads, value look up in pregenerated table, states and semaphores, fixed point decimal arithmetics for speed, and mathematical characterization of the birdsong.

Software Logic Overview

Reading keypad & Debouncing

The Keypad is connected to the microcontroller in the following way: 4 row pins are connected through resistors to GPIO outputs and the 3 column pins are connected to GPIO inputs and pulled low. The row pins are individually supplied voltage pulses and the column pins are polled, which produces a binary number with each digit for each GPIO pin. The result is masked with the keys to determine which GPIO pins are high and, if only one is pressed, return the number of the keypad key associated with it. If no keys are pressed or more than 1 are pressed, the key press is considered invalid and -1 is returned to communicate this to the debouncing machine.

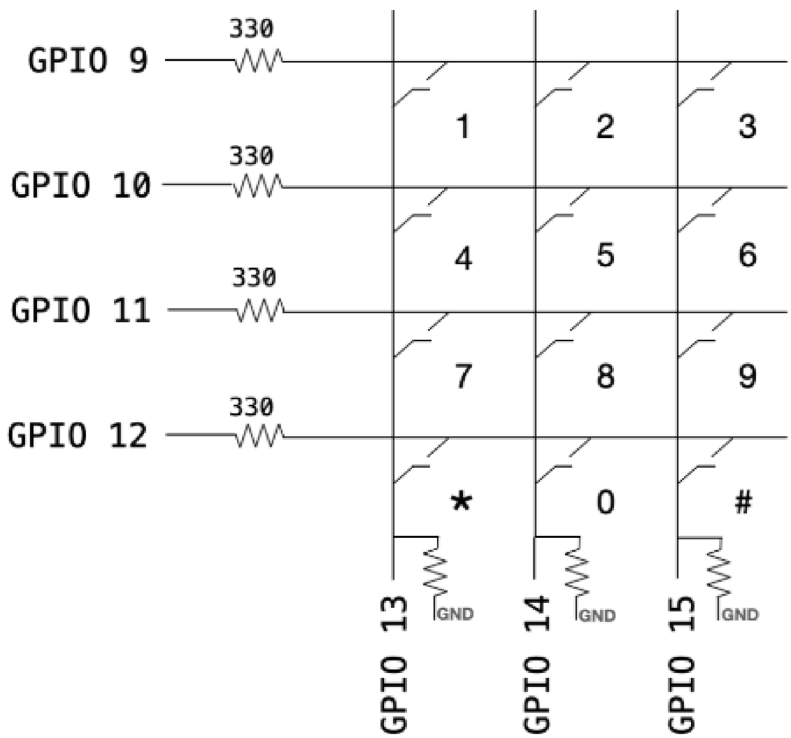

Figure 2: Keypad Schematic Diagram

However, interpreting these key presses directly is problematic: during a press from human, the keypad will read the key as high level multiple times because of physical limitations on the speed of releasing the key; additionally, the keys being spring-mass-damper systems are prone to bounce, which causes positive key readings after the keys were released.
To deal with this, a debouncing algorithm is required that compares the current and previous values of the keys and manages a state machine, as presented below. The transitions between states happen based on the scan of keypad keys.
There are four main states:

0) Not pressed/invalid press
1) Maybe pressed
2) Pressed
3) Maybe not pressed

The transition logic is as follows:

0) Not pressed/invalid press
   - If some exactly one key is pressed, transition to state 1
   - Else, stay in state 0

1) Maybe pressed
   - If current key pressed is the same as last key pressed, transition to state 2
   - If the key is different or no key is pressed, transition to state 0

2) Pressed
   - If current key pressed is the same as last key pressed, stay in state 2
   - Else, transition to state 3

3) Maybe not pressed
   - If current key pressed is the same as last key pressed, transition to state 2
   - If the key is different or no key is pressed, transition to state 0

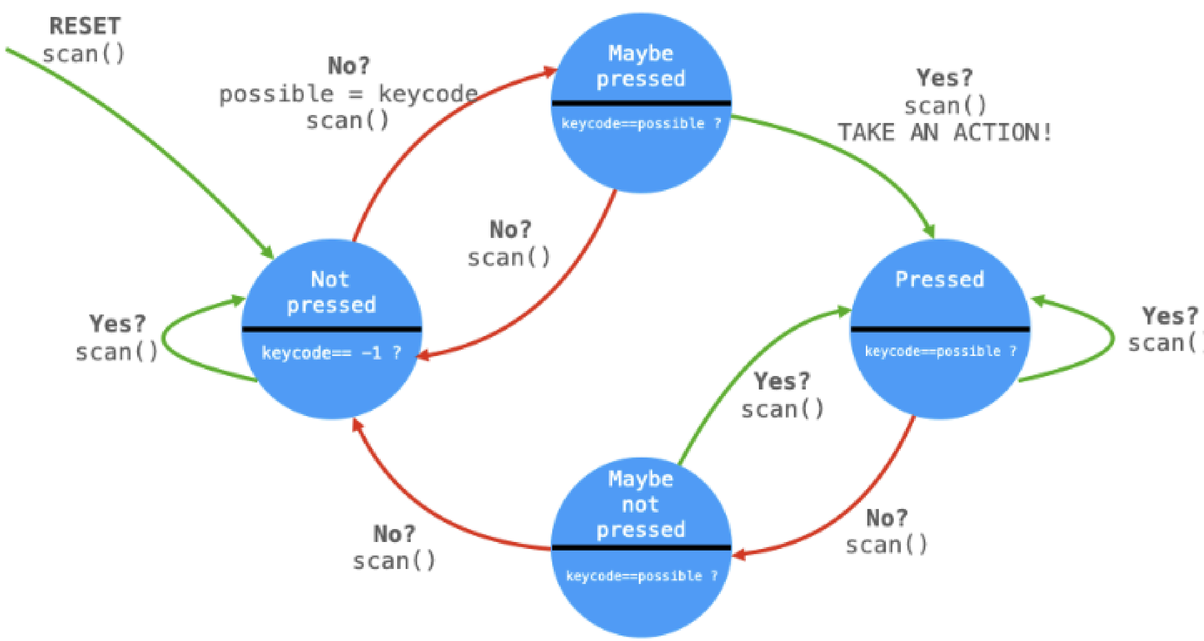

Figure 3: FSM for Button Press Debouncing Logic

Modes

There are three modes in the program: sandbox mode (pressing a sound key plays it), record mode (pressing a sound key plays the sound and appends the sound sequence in memory), and playback mode (no key presses are accepted and the sound sequence is played back).

The debouncing logic for the keypad and logic for transitioning between modes are implemented inside of the main protothread of the program protothread_core_0.

The sound is produced by the speaker connected to a DAC and the communication with the DAC is implemented in the ISR via SPI.
The playback mode is implemented in a separate protothread called protothread_playback, which significantly simplifies the logic of the program with the use of semaphores.
The following flags are used:

- trigger - indicates that sound should be played right now and permits writing to DAC
- is_playing_back - indicates the playback state
- is_recording - indicates the recording state

The following global variables are shared:

- soundnumber - the number of the sound to be played
- recorded_count - number of sounds in the current sequence
- recorded_sounds - an array that holds the sequence of sounds for playback

The transition logic is happening at the transition from “maybe pressed” state to “pressed” state and works the following way:

First of all, the state machine does not accept any key presses if the semaphore is_playing_back is set to 1 (when the sound sequence is simply being played) or unless the pressed key is the record key - a forceful way to end the playback.

If the pressed key is the record key and the current mode is not recording (is_recording flag is “false”), the is_recording flag is set to “true” and the recorded_count is reset to 0 to prepare for recording. If the is_recording flag was “true”, the is_recording flag is set to “false” and the is_playing_back flag is set to “true”. The is_playing_back flag is an indicator for the playback thread to start playing the sound, because the user just finished recording the sequence. This is the transition to playback mode.

If the key press is not the record key and the is_recording flag is “true”, the key press (only if it is 1, 2, or 3 for swoop, chirp, or silence) is placed at the position recorded_count+1. If the flag is false, the soundnumber obtains the value of the key number and the trigger is set to 1 - an indication of the sandbox mode and a semaphore for ISR to play the respective sound.

As mentioned, the playback thread is always active in a while(1) loop and is waiting for semaphore is_playing_back to be set to 1 to start playing the sound sequence. When that happens, the program iterates through the recorded_sounds array: assigns the soundnumber the value of the next sound, sets trigger to 1 to indicate ISR to play the next sound, and waits 130ms for sound to be played. On the last sound in the array (at index recorded_count - 1), the soundnumber is reset to -1 and the is_playing_back flag is reset to 0 to indicate the end of the playing sequence. This indicates the return to the sandbox mode.

Timing and Optimization

The ISR frequency is set to 50kHz, which implies that the routine should be able to compute sound frequency and amplitude and communicate those to DAC within 20 microseconds.
The key time optimization points are using lookup tables for sine wave values (which take a long time to compute), using fixed decimal point arithmetic, and using constants.

Due to the sound duration of 130ms, it takes 6500 ISR calls to play each sound. To track time, a counter count_0 is introduced in the ISR and is incremented by one with each call; on every iteration, ISR sends to the DAC the amplitude and frequency of the desired signal. When reaching a value of 6500, the counter is reset to 0 and the communication with DAC stops. For convenience, the semaphore trigger stays 1 until it is reset to 0 together with the ISR counter - this provides an easy check for whether a sound is currently being played.

The frequency of the sound is calculated based on the value of the counter as an indicator of time and is described by a different equation for each sound. The sound consists of a sine-shaped peak - swoop - followed by a quadratic rise - chirp, as in Figure X. While calculating a value of parabola directly is sufficiently fast, computing the value of sine wave is a very time-expensive process. To deal with this and complete the timing requirements, the value of the sine wave is looked up from a table that is constructed during initialization of the program - sin_table_swoop. This table is calculated for sine argument values between 0 and 2 pi and amplitude of 260 - to precompute even this small bit, and it consists of 6500 entries. We only need the first half of them, so we divide count_0 by 2 (so it is between 0 and 6500/2) when using it as an index to retrieve the sine wave value from the table.

The amplitude envelope of the sound is a trapezoid with a ramp up and ramp down; this is needed to ensure smooth derivative of the sound amplitude and make the sound cleaner. Without ramp up, the speaker would have to jump between 0 and full amplitude, which would cause it to make a loud crack sound.

A similar lookup table is used to convert the accumulated phase of the signal to the format required by DAC; the frequency and amplitude values are then sent over SPI.

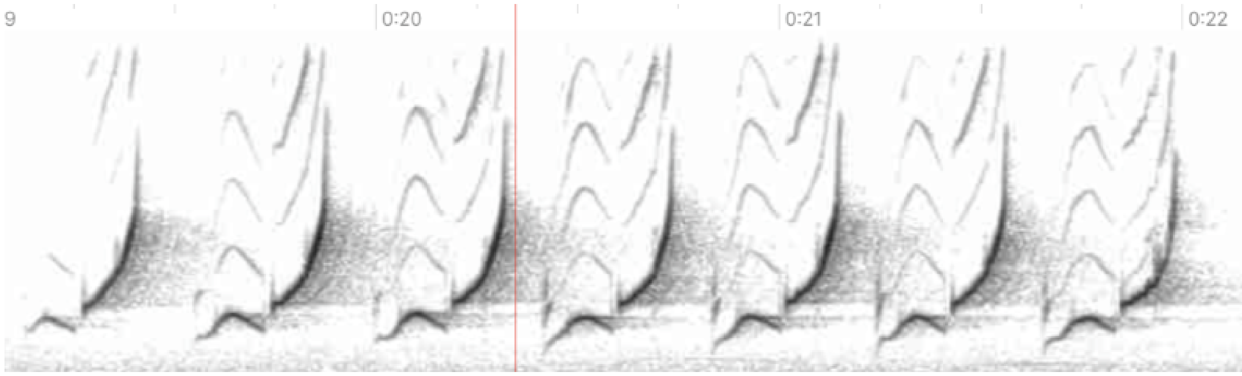

Figure 4: Spectrogram of several swoops and chirps

Another major optimization technique in use is the fixed decimal point numbers. Specifically, the sine tables are all computed in fix15 format and are multiplied by the amplitude profile using a specially designed multiplication function for this type. It makes use of the binary representation of the numbers - which makes them ints - and inexpensive register shifts. The initial 32-bit numbers are converted to long long format, which makes them 64 bits long with addition of zeros. This is needed to prevent overflow during multiplication. When done, the final number is shifted right 15 bits to cut off least significant bits and is converted back to the 32-bit fix15 type number.

The following are the durations of ISRs given different soundnumber values as measured by the oscilloscope: the swoop sound ISR takes ~10 microseconds and chirp sound ISR takes ~12.3 micro seconds.

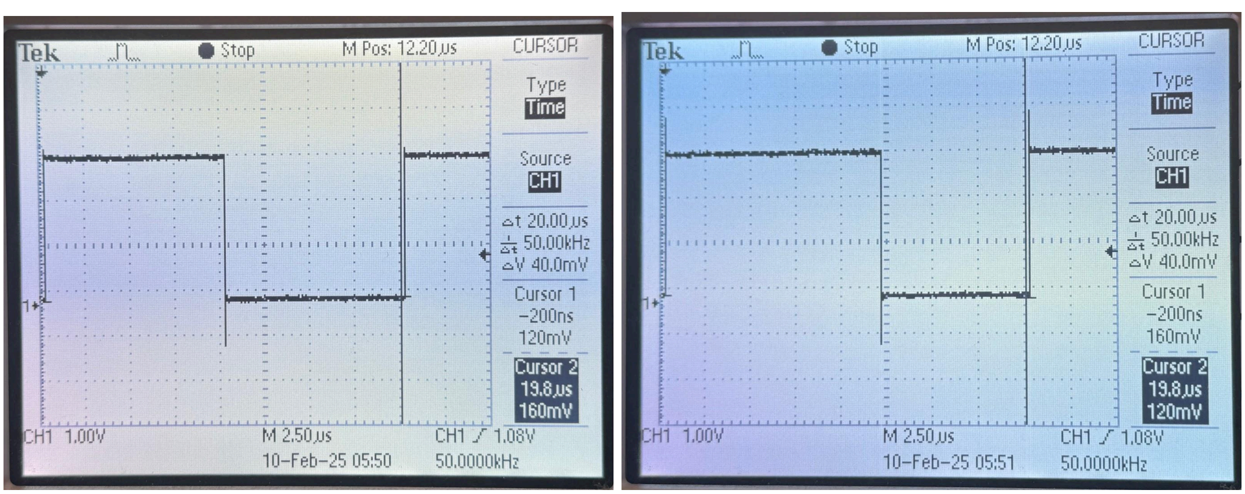

Figure 5: Waveform Screenshots from the Oscilloscope

Debugging and Testing Strategies

In this particular lab, we found the overall integration and test aspect of the code to be the one that challenged us the most. When trying to integrate the sample code for the keypad with beeping logic, the most important thing was defining constraints, understanding the keypad wiring, as well as making sure the correct frequency could be reflected for the chirps and swoops. I will touch on each of these debugging problems in more detail.

First, we had to carefully define constraints so that our microcontroller would remain responsive and stable. We decided to scan the keypad at intervals of roughly 130 ms. When it came to understanding the keypad presses and wiring, we placed a strong emphasis on validating keypad wiring because incorrect row or column assignments are a classic source of errors. We labeled every pin on the keypad and double-checked that it matched the Raspberry Pi Pico pins in our breadboard layout (using a MicroPython simulator). Not only did we use this simulator to understand the GPIO mapping in a high level manner, but we used print statements for the physical connections to test for correct output in the serial monitor coming from each press of the button. This did a great job in giving us concrete feedback for our key presses especially when checking our FSM Debouncing logic.

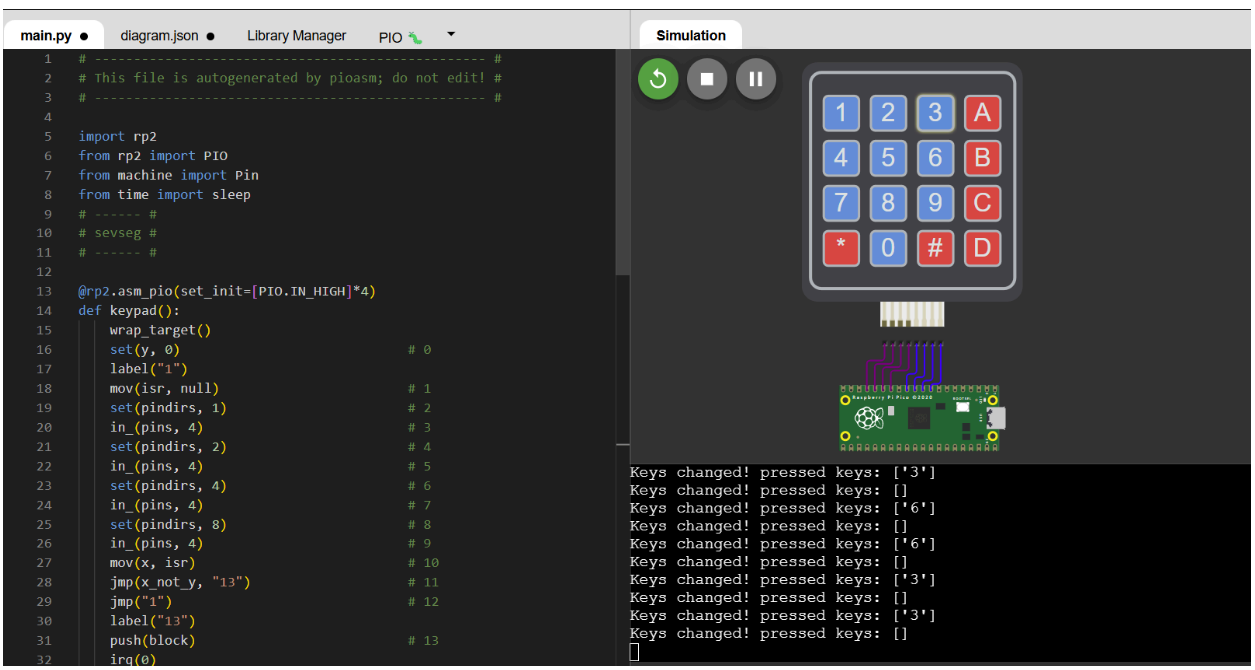

Figure 6: Understanding GPIO Mappings using MicroPython Simulations

Since our project used a FSM state machine, we needed a thorough understanding of how the transitions would be reflected based on the desired conditions. Again, we methodically tested by pressing each key from top to bottom and left to right, then comparing the console output to what we expected in our FSM diagram (using print statements to showcase the state number) . Finally, tuning the chirps and swoops required trial and error to find frequencies that were distinct, audible, and brief. We started initially calculating our Frequencies based on our 6500 samples. For swoops, we swept the frequency from Listening to the output to help us decide whether the beep was too long or too high‑pitched. Additionally, We tried to match the Sinusoidal curve as best as possible by modeling our frequency equation with a quadratic function, which did get very close to our desired frequency but not quite. Once we truly set up the correct sine table for the swoops on real hardware, an oscilloscope further confirmed whether the output was hitting the correct frequencies and whether multiple presses overlapped sounds.

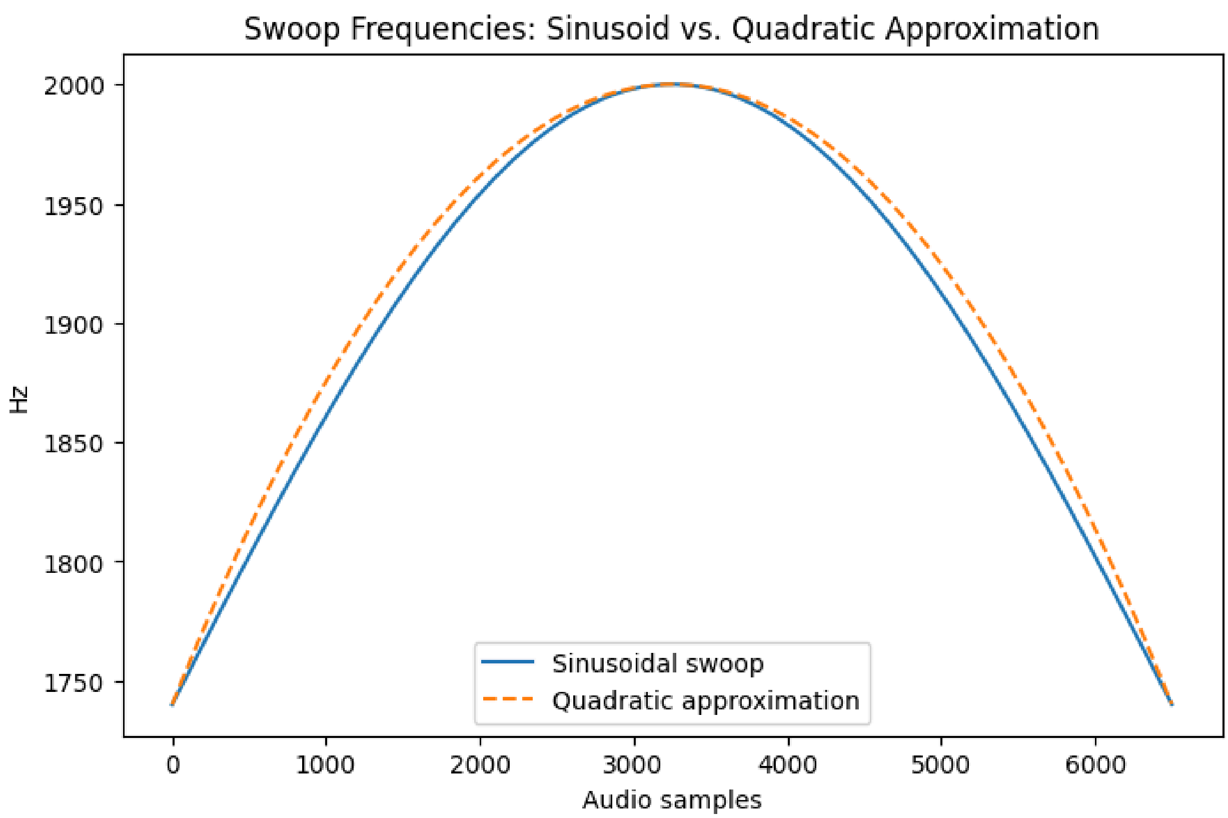

Figure 7: Quadratic Approximation vs. Sinusoid for Swoop Frequency

By methodically defining constraints, validating hardware connections, and fine‑tuning beep logic, we gradually built confidence in our code. Throughout each stage, we tested small sections of logic—first validating keypad scanning alone, then checking beep playback—and carefully merged them once each component worked reliably on its own. Printing status messages to the console gave continuous insight into system performance, timing, and potential issues. In the end, these debugging and testing practices helped us integrate the keypad and audio logic into a more robust and satisfying user interface experience.

Appendix:

```
/**
 * Hunter Adams (vha3@cornell.edu)
 *
 * Keypad Demo
 *
 * KEYPAD CONNECTIONS
 *  - GPIO 9   -->  330 ohms  --> Pin 1 (button row 1)
 *  - GPIO 10  -->  330 ohms  --> Pin 2 (button row 2)
 *  - GPIO 11  -->  330 ohms  --> Pin 3 (button row 3)
 *  - GPIO 12  -->  330 ohms  --> Pin 4 (button row 4)
 *  - GPIO 13  -->     Pin 5 (button col 1)
 *  - GPIO 14  -->     Pin 6 (button col 2)
 *  - GPIO 15  -->     Pin 7 (button col 3)
 *
 * VGA CONNECTIONS
 *  - GPIO 16 ---> VGA Hsync
 *  - GPIO 17 ---> VGA Vsync
 *  - GPIO 18 ---> 470 ohm resistor ---> VGA Green
 *  - GPIO 19 ---> 330 ohm resistor ---> VGA Green
 *  - GPIO 20 ---> 330 ohm resistor ---> VGA Blue
 *  - GPIO 21 ---> 330 ohm resistor ---> VGA Red
 *  - RP2040 GND ---> VGA GND
 *
 * SERIAL CONNECTIONS
 *  - GPIO 0        -->     UART RX (white)
 *  - GPIO 1        -->     UART TX (green)
 *  - RP2040 GND    -->     UART GND
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>

#include "pico/stdlib.h"
```

```
#include "pico/multicore.h"

#include "hardware/pio.h"
#include "hardware/dma.h"
#include "hardware/sync.h"
#include "hardware/spi.h"

// VGA graphics library
#include "vga16_graphics.h"
#include "pt_cornell_rp2040_v1_3.h"

// Low-level alarm infrastructure we'll be using
#define ALARM_NUM 0
#define ALARM_IRQ TIMER_IRQ_0

// Keypad Global flags for playback and recording functionality
static volatile bool is_recording = false;
static volatile bool is_playing_back = false;

// Storing logic to account sounds
#define MAX_SOUNDS 50 // max number of recorded sounds in a row
static int recorded_sounds[MAX_SOUNDS]; //array holding the sounds numbers
static int recorded_count = 0;  // counter for iterating through array

// Macros for fixed-point arithmetic (faster than floating point)
typedef signed int fix15 ;
#define multfix15(a,b) ((fix15)((((signed long long)(a))*((signed long
long)(b)))>>15))
#define float2fix15(a) ((fix15)((a)*32768.0))
#define fix2float15(a) ((float)(a)/32768.0)
#define absfix15(a) abs(a)
#define int2fix15(a) ((fix15)(a << 15))
#define fix2int15(a) ((int)(a >> 15))
#define char2fix15(a) (fix15)(((fix15)(a)) << 15)
#define divfix(a,b) (fix15)( (((signed long long)(a)) << 15) / (b))

//Direct Digital Synthesis (DDS) parameters
#define two32 4294967296.0  // 2^32 (a constant)
#define Fs 50000
#define DELAY 20 // 1/Fs (in microseconds)
```

```
// the DDS units - core 0
// Phase accumulator and phase increment. Increment sets output frequency.
volatile unsigned int phase_accum_main_0;
volatile unsigned int phase_incr_main_0 = (400.0*two32)/Fs ;

// DDS sine table (populated in main())
#define sine_table_size 256
#define sine_table_size_swoop 6500
fix15 sin_table[sine_table_size] ;
fix15 sin_table_swoop[sine_table_size_swoop] ;

// Values output to DAC
int DAC_output_0 ;
int DAC_output_1 ;

// Amplitude modulation parameters and variables
fix15 max_amplitude = int2fix15(1) ;     // maximum amplitude
fix15 attack_inc ;                       // rate at which sound ramps up
fix15 decay_inc ;                        // rate at which sound ramps down
fix15 current_amplitude_0 = 0 ;          // current amplitude (modified in
ISR)
fix15 current_amplitude_1 = 0 ;          // current amplitude (modified in
ISR)

// Timing parameters for beeps (units of interrupts)
#define ATTACK_TIME             250
#define DECAY_TIME              250
#define SUSTAIN_TIME            10000
#define BEEP_DURATION           6500
#define BEEP_REPEAT_INTERVAL    50000

// Frequency of sound
float freq = 0; //frequency

// SPI data
uint16_t DAC_data_1 ; // output value
uint16_t DAC_data_0 ; // output value

// DAC parameters (see the DAC datasheet)
```

```c
// A-channel, 1x, active
#define DAC_config_chan_A 0b0011000000000000
// B-channel, 1x, active
#define DAC_config_chan_B 0b1011000000000000

//SPI configurations (note these represent GPIO number, NOT pin number)
#define PIN_MISO 4
#define PIN_CS   5
#define PIN_SCK  6
#define PIN_MOSI 7
#define LDAC     8
#define LED      25
#define SPI_PORT spi0

//GPIO for timing the ISR
#define ISR_GPIO 2

// Keypad pin configurations
#define BASE_KEYPAD_PIN 9
#define KEYROWS         4
#define NUMKEYS         12
#define LED             25

unsigned int keycodes[12] = {   0x28, 0x11, 0x21, 0x41, 0x12,
                                0x22, 0x42, 0x14, 0x24, 0x44,
                                0x18, 0x48} ;
unsigned int scancodes[4] = {   0x01, 0x02, 0x04, 0x08} ;
unsigned int button = 0x70 ;

// trigger determines when to play sound or not
volatile int trigger = 0;

// State machine variables
volatile unsigned int STATE_0 = 0 ;
// 0 - not pressed
// 1 - maybe pressed
// 2 - pressed
// 3 - maybe not pressed
volatile unsigned int count_0 = 0 ;
```

```c
char keytext[40];
int prev_key = -1; // changed for core_0 protothread rewrite
int soundnumber = 0;


// key_press is used to determine the number of the key that is pressed
int key_press(uint32_t keypad, int i){
    // Scan the keypad!
        for (i=0; i<KEYROWS; i++) {
            // Set a row high
            gpio_put_masked((0xF << BASE_KEYPAD_PIN),
                            (scancodes[i] << BASE_KEYPAD_PIN)) ;
            // Small delay required
            sleep_us(1) ;
            // Read the keycode
            keypad = ((gpio_get_all() >> BASE_KEYPAD_PIN) & 0x7F) ;
            // Break if button(s) are pressed
            if (keypad & button) break ;
        }
        // If we found a button . . .
        if (keypad & button) {
            // Look for a valid keycode.
            for (i=0; i<NUMKEYS; i++) {
                if (keypad == keycodes[i]) break ;
            }
            // If we don't find one, report invalid keycode
            if (i==NUMKEYS) (i = -1) ;
        }
        // Otherwise, indicate invalid/non-pressed buttons
        else (i=-1) ;
        return i;
}

// --------------------- Playback Thread ---------------------

//replay the recorded buffer array with a separate thread

static PT_THREAD(protothread_playback(struct pt *pt))
{
```

```
    static int idx; //local iteration index
    PT_BEGIN(pt);

    while (1){
        if (is_playing_back == true){
            //For each recorded sound
            for(idx= 0; idx < recorded_count ; idx++){
                soundnumber = recorded_sounds[idx];

                trigger = 1; // indication to ISR to start playing it

                // ISR clears the 'trigger' when the beep ends, until then
wait for sound to finish playing
                while(trigger){
                    PT_YIELD_usec(130000);
                }

                //Done playing
                if (idx == recorded_count-1){
                    soundnumber = -1;
                    is_playing_back = false;
                }

            }
        }

    PT_YIELD_usec(30000) ;

    }

    PT_END(pt);
}

// This thread runs on core 0
static PT_THREAD (protothread_core_0(struct pt *pt))
{
    // Indicate thread beginning
    PT_BEGIN(pt) ;

    // Some variables
```

```
    static int i ;
    static uint32_t keypad ;

    while(1) {

        // GPIO for checking the loop
        gpio_put(LED, !gpio_get(LED)) ;
        //scan the keypad
        i = key_press(keypad, i);

        //change states only if we are not currently playing the sound
back
        if (!is_playing_back || i==10){
            //Debounce state machine, logic for state transitions
            switch (STATE_0){
                case 0: // when not pressed
                    if (i >= 0 ) STATE_0 = 1;
                    break;
                case 1:  // maybe pressed
                    if (i == prev_key &&  i >=0){
                        STATE_0 = 2; //definitely pressed
                        if(i == 10){
                            if(!is_recording){
                                //start recording
                                is_recording = true;
                                recorded_count =0;
                            }
                            else {
                                // Stop recording => begin playback when
thread is scheduled
                                is_recording = false;
                                is_playing_back = true; // switch flag to
true to indicate we are playing back
                            }
                        }
                        else {
                            // If not the record toggle key:
                            if (is_recording) {
                                // Save this key if it is a valid sound
key
```

```
                            if ((i == 1 || i==2 || i==3) &&
                                 recorded_count < MAX_SOUNDS) {
                                recorded_sounds[recorded_count++] = i;
                            }
                        }
                        // play key always when a valid sound key
pressed
                        if (i == 1 || i == 2 || i == 3) {
                            soundnumber = i;
                            trigger = 1;
                        }

                    }

                } else {
                    STATE_0 = 0; // not pressed
                }
                break;
            case 2: //pressed
                if (i!=prev_key){
                    STATE_0 = 3;  //maybe not pressed
                }
                break;

            case 3: // maybe not pressed
                if (i < 0){
                    STATE_0 = 0; // not pressed
                }
                else if (i == prev_key){
                    STATE_0 = 2; // pressed
                }
                else{
                    STATE_0 =0; // not pressed or invalid
                }
                break;
            }



        // Write key to VGA, if VGA is connected for debugging
```

```
        prev_key = i ;
        fillRect(250, 20, 176, 30, RED); // red box
        sprintf(keytext, "%d", i) ;
        setCursor(250, 20) ;
        setTextSize(2) ;
        writeString(keytext) ;;

        }

        PT_YIELD_usec(30000) ;
    }
    // Indicate thread end
    PT_END(pt) ;
}

// This timer ISR is called on core 0
static void alarm_irq(void) {

    // Assert a GPIO when we enter the interrupt
    gpio_put(ISR_GPIO, 1) ;

    // Clear the alarm irq
    hw_clear_bits(&timer_hw->intr, 1u << ALARM_NUM);

    // Reset the alarm register
    timer_hw->alarm[ALARM_NUM] = timer_hw->timerawl + DELAY ;

    // only play if we are allowed to play
    if (trigger == 1) {

        if (soundnumber == 1){ //swoop
            freq = 1740 + fix2float15( sin_table_swoop[count_0/2]);

        } else
        if (soundnumber == 2) { //chirp
            freq = 0.000118343 * (count_0) * (count_0) + 2000;
        } else

        if (soundnumber == 3) { //silence
            freq = 0;
```

```
        }

        // DDS phase and sine table lookup
        phase_incr_main_0 = (unsigned int)(freq*two32/Fs)  ;
        phase_accum_main_0 = phase_accum_main_0 + phase_incr_main_0;
        DAC_output_0 = fix2int15(multfix15(current_amplitude_0,
            sin_table[phase_accum_main_0>>24])) + 2048 ;

        // Ramp up amplitude
        if (count_0 < ATTACK_TIME) {
            current_amplitude_0 = (current_amplitude_0 + attack_inc) ;
        }
        // Ramp down amplitude
        else if (count_0 > BEEP_DURATION - DECAY_TIME) {
            current_amplitude_0 = (current_amplitude_0 - decay_inc) ;
        }

        // Mask with DAC control bits
        DAC_data_0 = (DAC_config_chan_B | (DAC_output_0 & 0xffff))  ;

        // SPI write (no spinlock b/c of SPI buffer)
        spi_write16_blocking(SPI_PORT, &DAC_data_0, 1) ;

        // Increment the counter
        count_0 += 1 ;

        // Update flags and counters
        if (count_0 == BEEP_DURATION) {
            count_0 = 0 ;
            trigger = 0;
            soundnumber = 0;
            current_amplitude_0 = 0 ;
            // now can increment the counter in playback array only if we
are in playback mode and iterator < max size
        }
    }

    // De-assert the GPIO when we leave the interrupt
    gpio_put(ISR_GPIO, 0) ;
```

```
}



int main() {

    // Initialize stdio
    stdio_init_all();

    // Initialize SPI channel (channel, baud rate set to 20MHz)
    spi_init(SPI_PORT, 20000000) ;
    // Format (channel, data bits per transfer, polarity, phase, order)
    spi_set_format(SPI_PORT, 16, 0, 0, 0);

    // Map SPI signals to GPIO ports
    gpio_set_function(PIN_MISO, GPIO_FUNC_SPI);
    gpio_set_function(PIN_SCK, GPIO_FUNC_SPI);
    gpio_set_function(PIN_MOSI, GPIO_FUNC_SPI);
    gpio_set_function(PIN_CS, GPIO_FUNC_SPI) ;

    // Map LDAC pin to GPIO port, hold it low (could alternatively tie to
GND)
    gpio_init(LDAC) ;
    gpio_set_dir(LDAC, GPIO_OUT) ;
    gpio_put(LDAC, 0) ;

    // Setup the ISR-timing GPIO
    gpio_init(ISR_GPIO) ;
    gpio_set_dir(ISR_GPIO, GPIO_OUT);
    gpio_put(ISR_GPIO, 0) ;

    // set up increments for calculating bow envelope
    attack_inc = divfix(max_amplitude, int2fix15(ATTACK_TIME)) ;
    decay_inc =  divfix(max_amplitude, int2fix15(DECAY_TIME)) ;
    // Build the sine lookup table
    // scaled to produce values between 0 and 4096 (for 12-bit DAC)
    int ii;
    for (ii = 0; ii < sine_table_size; ii++){
         sin_table[ii] =
float2fix15(2047*sin((float)ii*6.283/(float)sine_table_size));
    }
```

```
    // table for lookup of the sine value for swoop
    int jj;
    for (jj = 0; jj < sine_table_size_swoop; jj++){
         sin_table_swoop[jj] =
float2fix15(260*sin((float)jj*6.283/(float)sine_table_size_swoop));
    }

    // Enable the interrupt for the alarm (we're using Alarm 0)
    hw_set_bits(&timer_hw->inte, 1u << ALARM_NUM) ;
    // Associate an interrupt handler with the ALARM_IRQ
    irq_set_exclusive_handler(ALARM_IRQ, alarm_irq) ;
    // Enable the alarm interrupt
    irq_set_enabled(ALARM_IRQ, true) ;
    // Write the lower 32 bits of the target time to the alarm register,
arming it.
    timer_hw->alarm[ALARM_NUM] = timer_hw->timerawl + DELAY ;

    // Initialize the VGA screen
    initVGA() ;

    // Draw some filled rectangles
    fillRect(64, 0, 176, 50, BLUE); // blue box
    fillRect(250, 0, 176, 50, RED); // red box
    fillRect(435, 0, 176, 50, GREEN); // green box

    // Write some text
    setTextColor(WHITE) ;
    setCursor(65, 0) ;
    setTextSize(1) ;
    writeString("Raspberry Pi Pico") ;
    setCursor(65, 10) ;
    writeString("Keypad demo") ;
    setCursor(65, 20) ;
    writeString("Hunter Adams") ;
    setCursor(65, 30) ;
    writeString("vha3@cornell.edu") ;
    setCursor(250, 0) ;
    setTextSize(2) ;
    writeString("Key Pressed:") ;
```

```
    // Map LED to GPIO port, make it low
    gpio_init(LED) ;
    gpio_set_dir(LED, GPIO_OUT) ;
    gpio_put(LED, 0) ;

    ////////////////// KEYPAD INITS //////////////////////
    // Initialize the keypad GPIO's
    gpio_init_mask((0x7F << BASE_KEYPAD_PIN)) ;
    // Set row-pins to output
    gpio_set_dir_out_masked((0xF << BASE_KEYPAD_PIN)) ;
    // Set all output pins to low
    gpio_put_masked((0xF << BASE_KEYPAD_PIN), (0x0 << BASE_KEYPAD_PIN)) ;
    // Turn on pulldown resistors for column pins (on by default)
    gpio_pull_down((BASE_KEYPAD_PIN + 4)) ;
    gpio_pull_down((BASE_KEYPAD_PIN + 5)) ;
    gpio_pull_down((BASE_KEYPAD_PIN + 6)) ;

    // Add core 0 and playback threads
    pt_add_thread(protothread_core_0) ;
    pt_add_thread(protothread_playback);

    // Start scheduling core 0 threads
    pt_schedule_start ;

}
```