

ECE 5730 Lab 1 Report
Birdsong Synthesizer



Amber Li | Ivan Huang | Zarif Karim
09.10.2026

Introduction

Birds are awesome. They sing songs like Professor Batten singing Taylor Swift in Rhodes Hall.

In this lab, we implemented a birdcall synthesizer using the Raspberry Pico 2 (RP2350) that reproduces whistles of birds (almost) accurately enough to get recognized by a bird song recognition app (Merlin Bird ID). Sounds output from our system can be custom configured to replicate that of a birdcall. This project involved hardware/software where hardware was designed to output audio, while software was used to generate waveforms. The system generates audio via the Direct Digital Synthesis algorithm. We send frequency amplitude samples through SPI to a DAC, which is then wired to a 3.5mm audio jack. We control the bird song by way of a slider potentiometer being read by the ADC on board, and a keypad programmed to record sound bites which can be played back in a sequence, like a bird song.



Figure 1: Lab Overview

Mathematical Background

Direct Digital synthesis is a way to make analog waveforms, in our case a sine wave, via a digital system. This consists of a software reference clock, an accumulator, and a lookup table. Then, later in hardware, a DAC turns the digital numbers into an output voltage level for smooth audio output. An accumulator of 32 bits is constantly incrementing, and that accumulator is used as the index to a lookup table that consists of amplitude values. Those amplitude values are read at a certain frequency such that they are collected into what looks like a sine wave.

$$N = \text{increment amount} = \frac{F_{out}}{F_s} \cdot 2^{32}$$

Figure 2: DDS Formula

In our design, we decide the following values:

- $F\{s\}$, the sampling frequency, is 50k Hz
- $2^{32} = 4294967296$
- $F\{out\}$ = Value converted from the ADC input

Later in this lab, we discuss where this computation is performed.

Design and Testing Methods

Direct Digital Synthesis Implementation

We chose a sine table size of 256 to ensure a reasonable resolution of audio without using up too much RAM. Though we use a 32-bit accumulator, only the top 8 bits are used to index the sine table. Below is a code snippet of how we defined this in our code.

```
// DDS sine table
#define sine_table_size 256
...
... sin_table[phase_accum_main>>24] ...
```

Recall that we have a 12-bit DAC. It happens to be the case that $2^{12} = 4096$, so our DAC supports voltage levels between 0 and 4095. Our sine table is scaled to use this full range to provide the most resolution for our audio signal. If we were to increase this amplitude, we would see flattened-out peaks due to clipping. On the other hand, if we lowered the amplitude, the output waveform would still be valid, but the available voltage levels would decrease, reducing the resolution of the waveform.

Our DAC only supports 12 bits of resolution, that being 0 to 4095, which means it should be unsigned. Coming from our sine table, we have peak to peak values, so we must somehow convert this range from signed to unsigned. After scaling up values from our sine table by multiplying by 2047, we convert that range of $-2047 \rightarrow +2047$ over to $0 \rightarrow 4095$ by adding 2048.

```
DAC_data = (DAC_config_chan_B | (((uint16_t)(sin_table[phase_accum_main>>24]*vol_val) + 2048) & 0xffff)) ;
...
sin_table[ii] = (int)(2047*sin((float)ii*6.283/(float)sine_table_size));
...
```

Audio via DAC and SPI

The DAC channel configurations are displayed below. Given 16 bits, the bottom twelve are the DAC data bits, while the top 4 are for configuring the DAC. The MSB defines if output comes from channel A or channel B. Bit 14 is a don't care, bit 13 is a gain selection bit which we set to 1x, and bit 12 is an output shutdown control bit which we set to stay in active mode operation, so V_{OUT} remains available.

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

In order to implement the DDS algorithm and generate an audio waveform, we first set up the DAC and SPI. The DAC is used to construct an analog sound wave from the digital sine lookup table. We specify the sound wave parameters in our RP2350 code, which communicates digital data via SPI to the DAC, and the converted analog data is then streamed out over GPIO to the audio socket. Using the DAC datasheet ([MCP 48X2](#)) and the provided SPI connections, we chose suitable pins on the RP2350 to connect the SPI interface with the DAC module.

For the SPI pins on the Pico, our definitions are as follows:

PIN_CS	GPIO 13
PIN_SCK	GPIO 14
PIN_MOSI	GPIO 15
LDAC	GPIO 5
SPI_PORT	SPI1

Table 1: DAC GPIO Pin assignments

(Note that when assigning pin definitions in the code, the GPIO number should be used rather than the pin position.)

We selected these GPIO pins because they are multipurpose, can be configured for SPI communication, and are unused by any other peripherals. MOSI is sending data from the RP2350 (Main) to the DAC (Secondary), which corresponds to the TX direction for SPI. Since our DAC is a unidirectional device and does not receive data, we left PIN_MISO assigned to an unused pin and left it unwired. The $\overline{LDAC}$ signal is tied low, which transfers the contents of the DAC's input registers to the output registers so both V_{OUTA} and V_{OUTB} are updated at the same time. We also changed the SPI_PORT definition to use spi1.

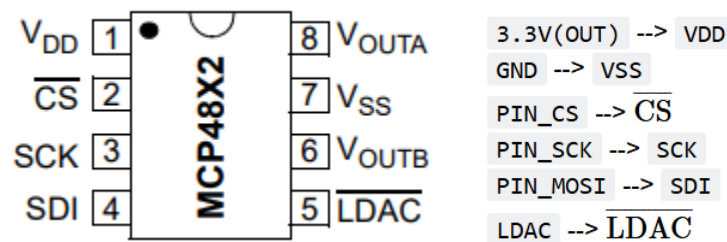


Figure 3: DAC Pinout and SPI Connections

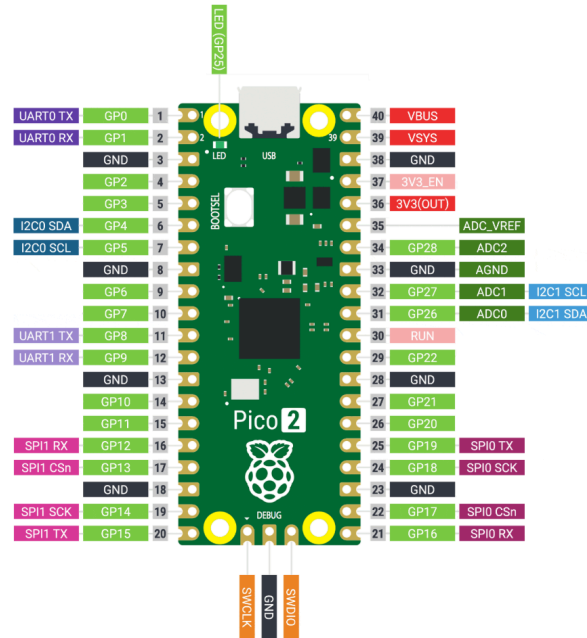


Figure 4: RP2350 Pinout

Our SPI initialization sets it to channel 1 (RP2350's second SPI peripheral instance) at a baud rate of 20MHz, and a message length of 16 bits. This means that 16-bit data is transmitted synchronously between two shift registers, one on the Pi and one on the DAC. The polarity and phase are both set to 0, so the clock is idle low, active high, and data is valid on rising edges.

To test our synthesized audio, we first verified the waveform by hooking up our DAC output to an oscilloscope. The output waveform for the DDS demo is expected to be a smooth analog sine wave with a frequency of 800 Hz. Then we wired the 3.5mm audio socket, connecting the ground pin and one of the audio legs to our DAC output. Plugging in the speakers allowed us to play the generated tone.

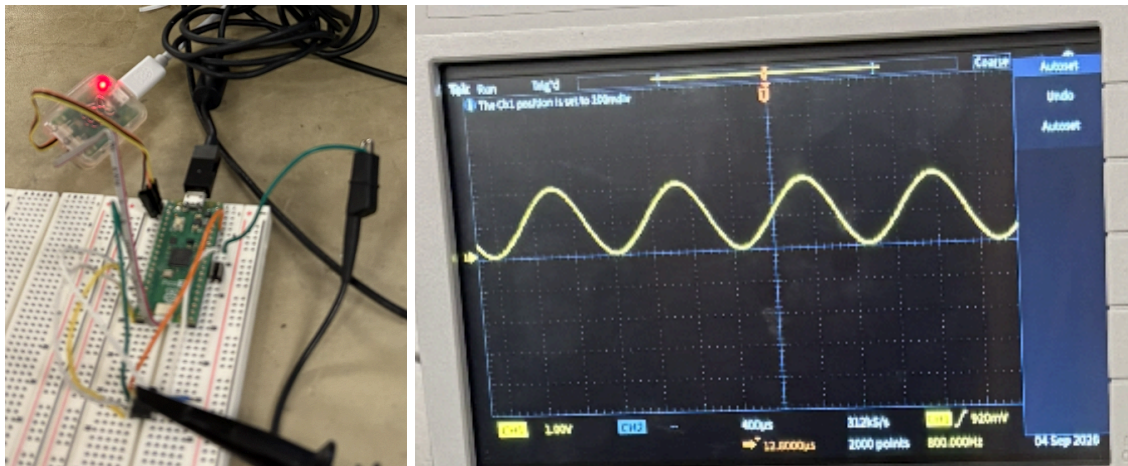


Figure 5:
Left: Initial DDS circuit
Right: 800 Hz sine wave

We did not ramp the sound amplitude in our tone generator. Without amplitude modulation, the sound at the beginning should have a click/pop due to the vertical square wave front containing a multitude of frequencies. However, we did not notice this artifact, likely due to the way we implement sound off. Rather than resetting the phase accumulator or directly outputting zero from the DAC, we set the phase accumulator increment to 0. This means that DDS continuously runs and the phase accumulator stays constant at the last value played, reducing large discontinuities compared to if you started from 0. If we did observe that this was impacting our sound, we would fix this by ramping the amplitude smoothly from 0 to full by multiplying by a linear ramp function at the beginning and end of the wave packet.

In addition, a bootselect button was added for easy programming. When the button is pressed, it shorts the Pi's RUN signal to ground. Pushing the button once resets the program, and pushing the button twice quickly puts the board into program mode. This avoids the need for unplugging the board for power cycling.

Slider Potentiometer

A 10k slide potentiometer was added to allow us to control the frequency of our DDS synthesizer. The ADC_MUX select was set to 0, and GPIO pin 26 was assigned to the potentiometer output and is read by the ADC on the Pi. After setting up the potentiometer, we connected the Pi to PuTTY's serial terminal to observe the printed outputs and confirmed that they ranged from roughly 0 - 4095 as the slider was moved.



Figure 6: 10k Potentiometer Pinout

In order to use the potentiometer to control DDS frequency, we mapped the ADC outputs over the range of 0-4096 to the range 0-10k by multiplying by 2.5 and modifying the ADC output value's type to be a float. This new freq_val is used to calculate the value of phase_incr_main, which is used in the ISR to form the audio data. The calculation occurs in the ADC thread so that the potentiometer output continuously updates, and the ISR will automatically have the remapped freq_val value in Hz. Finally, while integrating the ADC code with the DAC tone generator, we had to edit the CMakeLists.txt with the additional header files in order to provide proper linking for the different files.

We saw some noise in the raw ADC values. They were going wee-woo even when our slider was still! There's a ton of EM radiation in this lab from power supplies to PCs to other microcontrollers students

that may cause fluctuations in ADC output. A few other potential sources of noise include electrical noise from the connected supply and ground, long breadboard jumper wires acting as antennas, or small voltage fluctuations at the GPIO pins. We also noticed that physical imperfections in the pin contact to the breadboard were weak at times. The slide stick itself could have been loose or scratched from many slides, causing instability.

Our ADC worked just fine, so we did not implement any noise mitigation. We believe small fluctuations were rounded off with our integer-based calculations, or we did not end up noticing them in the end. Some ways we could mitigate noise for more robust ADC outputs may include performing some sampling averaging, doing some software filtering, or even adding a capacitor for electrical noise.

Keypad Scanning and Debouncing

The keypad is divided internally into rows and columns. First, one row is driven. All columns are then read. If a column is read low whilst a row is low, we know a key is pressed at the intersection. This produces an output pattern that we've encased in a table to identify which number (or character) was pressed. This routine is performed on all 4 rows. Here is what it looks like in software:

```
// Scan the keypad!
for (i=0; i<KEYROWS; i++) {
    gpio_put_masked((0xF << BASE_KEYPAD_PIN),
                    (scancodes[i] << BASE_KEYPAD_PIN)) ;
    sleep_us(1) ;
    keypad = ((gpio_get_all() >> BASE_KEYPAD_PIN) & 0x7F) ;
    if ((~keypad) & button) break ;
}
```

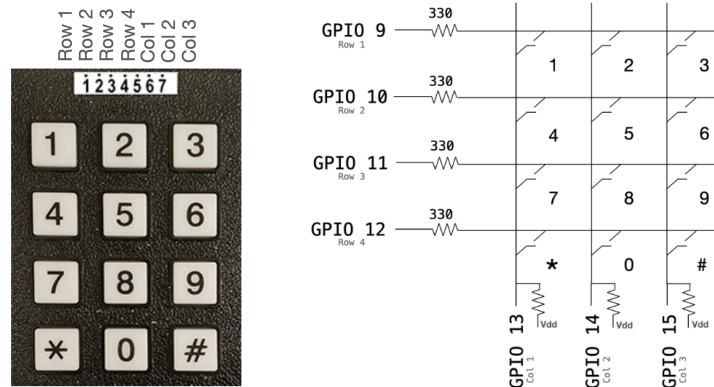


Figure 7: Keypad and Internal Connections

For the column input pins, we configured the GPIOs to enable their internal pull-up resistors. Without pull-up resistors, the columns will float when nothing is pressed. A floating input is a big no no, and would cause unpredictable behavior, since the microcontroller may detect it as either high or low. This resistor will guarantee that each column is read with the full digital voltage high level when not pressed, then can become low when pressed during the row scan that connects the column to the low row.

For the row output pins, external 330 ohm resistors were added to limit the current for rows. During a key press, a connection is made between the rows and the columns, with the row pins being configured as GPIO outputs and the column pins configured as GPIO inputs. During scanning, it's possible that this creates a path between GPIOs at different voltage levels, since the scan drives one row low at a time while the other rows are high. For example, if you press two keys in the same column at the same time, the row outputs may be directly connecting a GPIO low and high and cause large current to flow between the pins. The series resistors provide protection to prevent excess current from damaging the GPIOs, limiting the current to a few mA, which stays within the amount it can safely sink.

Scanning

In the scan code, a sleep is included at the end so there is a brief pause after every scan cycle. This is to let the row settle and prevent us from reading a previous row's value or some transient value. Remember, capacitance is everywhere! In class, we went over an RC delay that causes a mini lag. The voltage on a line does not change instantaneously, so adding a sleep ensures we read a stable correct value.

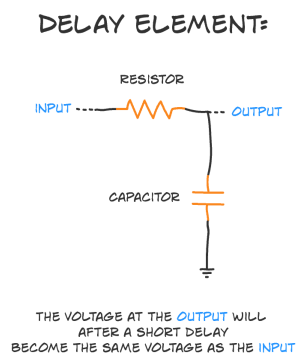


Figure 8: RC Delay

(Source: <https://www.build-electronic-circuits.com/rc-delay-element/>)

You can imagine the capacitance occurring between the intersection of wires, creating almost a parallel plate capacitor. But in schematic form, you should see a capacitor drawn at the output in a similar fashion to Figure 8, but for each row.

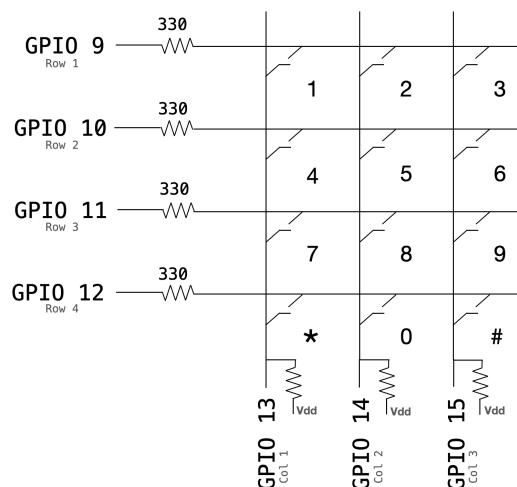


Figure 9: Schematic of keypad connections and corresponding keypad selections

Debouncing

The keypad utilizes a physical mechanism for signaling when a key is pressed by closing a mechanical switch. This leads to a bouncing effect where multiple button presses may be registered by the microcontroller each time a key is pressed/released. Thus, we can use some software logic to “debounce” the key presses by writing a state machine that checks whether the keypad reading is stable across multiple scans before declaring a press.

Debouncing was implemented using a debounce FSM, which scans for key presses and compares whether the current key press is the same as the previous key press. The FSM starts in the NOT_PRESSED state, then transitions to MAYBE_PRESSED if a key is detected. The key is stored in a *possible* variable and only transitions to the PRESSED state if that same key is detected again. This filters out bouncing on/off signals and returns a pressed signal only when a key is confirmed to be stabilized in the pressed state. When in PRESSED, the FSM stays until the key reading changes, when it moves to MAYBE_NOT_PRESSED. If a brief bounce occurred and the key is detected as pressed again, it would instead go back to the PRESSED state. Again, one more scan is required for a keycode != possible to transition to the NOT_PRESSED state.

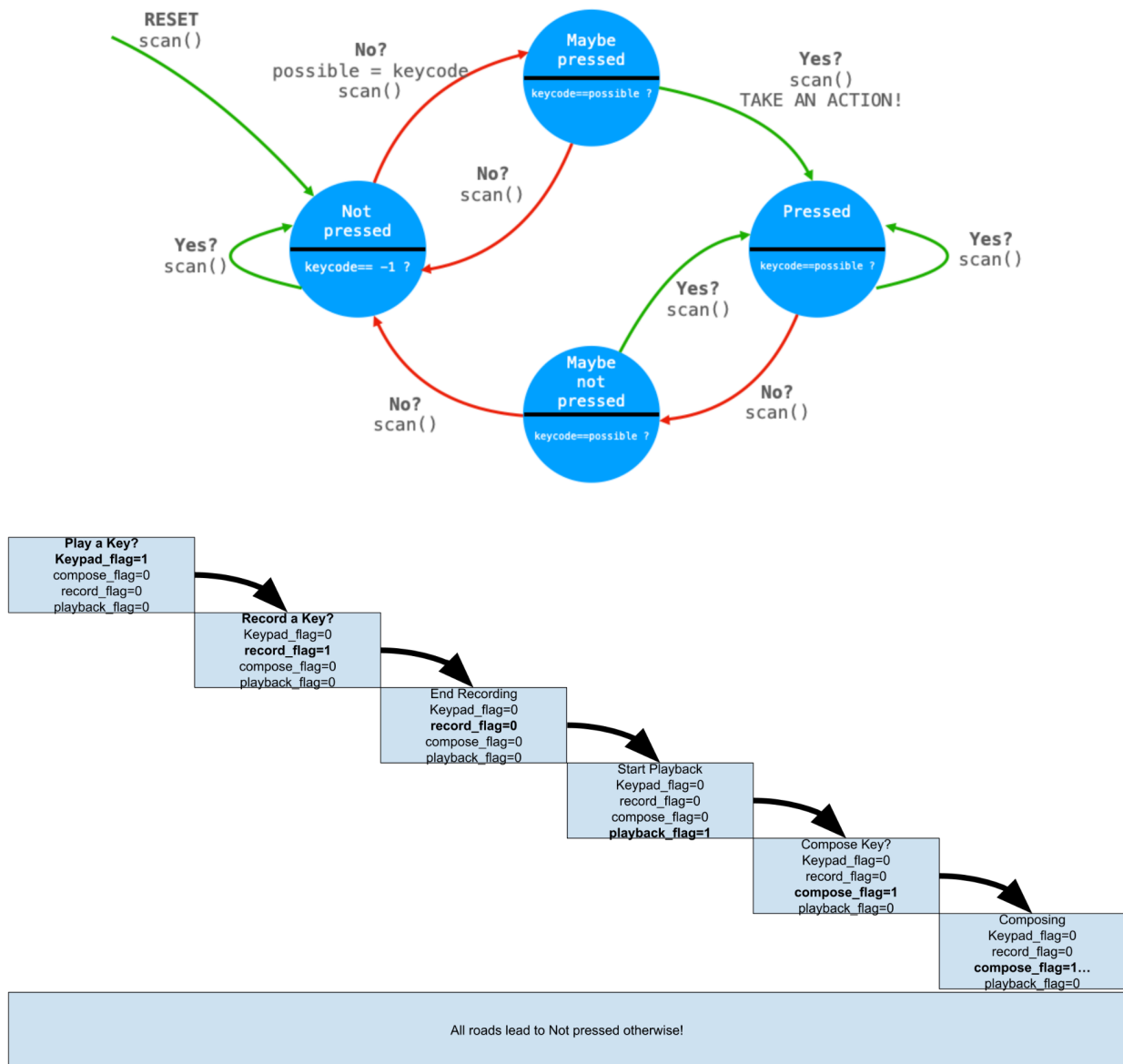


Figure 10: Modified Debouncing FSM with mode flags

Timer Interrupt

The timer ISR is running at 50kHz to match the frequency of the DAC in the DDS generator, so every time the DAC is ready to send an audio sample, the timer ISR has fired and retrieved the sine table value to send out through DAC_data. The ISR rate is set by the addition of the DELAY constant to the timerawl to rearm the alarm. DELAY was calculated to be $1/F_s$, or the inverse of the frequency of the DDS, giving a delay period of 20 microseconds.

```
timer_hw->alarm[ALARM_NUM] = timer_hw->timerawl + DELAY;
```

Audio output generation happens in the ISR. We increment the accumulator, then send the output to the DAC. We place this in the ISR so that audio generation can occur uninterrupted. ISR always takes precedence over protothreads. This is key for an audio based project because if another thread were to take over in the middle of audio playback, we would have instability in audio, and an unsatisfying bird chirper.

```
// Alarm ISR
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 ;

    // DDS phase and sine table lookup
    phase_accum_main += phase_incr_main ;
    DAC_data = (DAC_config_chan_B | (((uint16_t)(sin_table[phase_accum_main>>24]*vol_val) + 2048) & 0xffff)) ;

    // Perform an SPI transaction
    spi_write16_blocking(SPI_PORT, &DAC_data, 1) ;

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

}
```

To force all reads and writes to occur without assumptions of being the same from wherever it's spatially located in the code, making them stable and predictable for reading and writing between the ISR and our other protothreads. Take DAC_data, for example. It is defined outside of the ISR in regular code, however it's not actually used elsewhere. The compiler might assume it will just stay an empty variable, but that's not the case with the ISR constantly updating the variable, just at runtime. Volatile types help with this.

We used the oscilloscope to confirm the ISR is meeting timing. We have an extra GPIO set to high once the Pico enters the interrupt. We probed the pin to see that the execution was definitely less than the interrupt occurrence (every 20 microseconds).

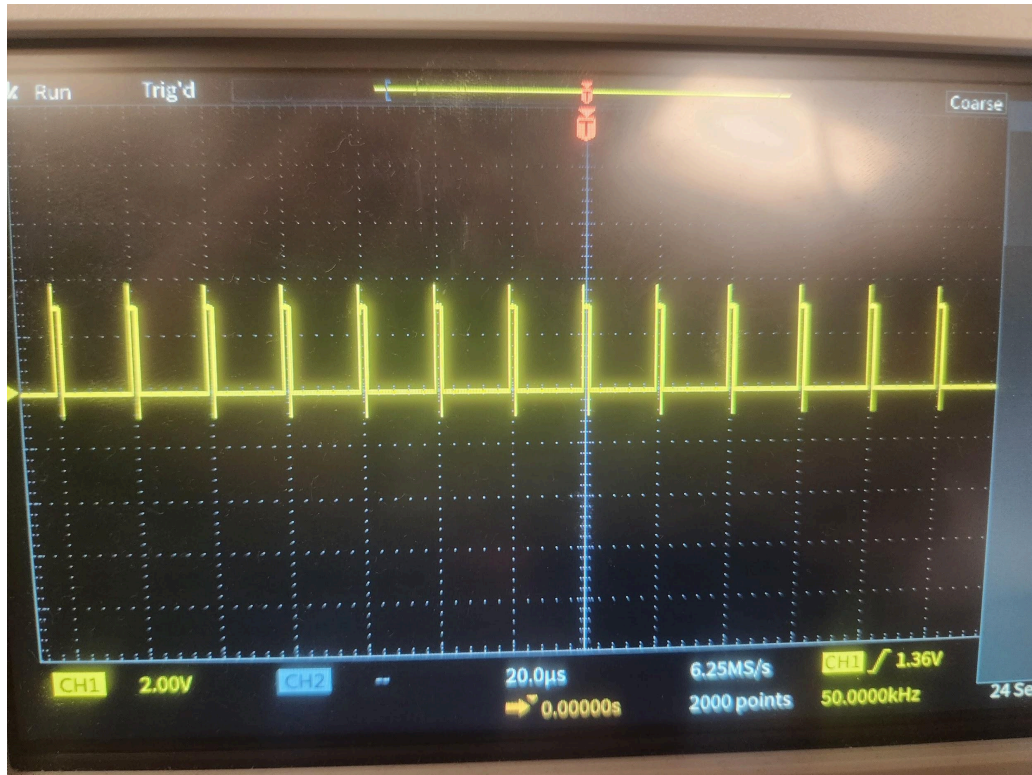


Figure 11: Timer ISR Scope Output (Period: 20us)

Volume Toggle Switch

As part of 5730, we added a Single Pole Double Throw (SPDT), or 2-state switch to choose between volume adjustment mode and frequency adjustment mode. This required adding a new volatile variable, as it is read by the ISR for final DAC output. Determining between volume and frequency required a simple if statement in our F{out} protothread:

```
adc_val = adc_read();
if(gpio_get(V_SWITCH_1) && !gpio_get(V_SWITCH_2)) {
    vol_val = adc_val / 4096; // normalize to range 0.0 - 1.0
    // printf("adc_out: %d, vol scale: %f\n", adc_val, vol_val);
}
else if (gpio_get(V_SWITCH_2) && !gpio_get(V_SWITCH_1)){
    freq_val = adc_val * 2.5; // normalize to range 0 - 10k
    // printf("adc_out: %d, freq scale: %f\n", adc_val, freq_val);
}
```

Each side of the switch was connected to a GPIO. If the first GPIO was on, then we are controlling volume. If the second GPIO was on, then we are controlling F{out}. When in volume mode, the potentiometer output is scaled by 4096 to normalize it to the range [0.0, 1.0], and this vol_val is multiplied in the DAC_data equation in the ISR. This scales the sine table output amplitude, which is how we control the volume.

```
DAC_data = (DAC_config_chan_B | (((uint16_t)(sin_table[phase_accum_main>>24]*vol_val) + 2048) & 0xffff)) ;
```

Application Level

Device Modes

Flags were used in the debounce FSM to transition between modes:

```
else if (possible == 10) {  
    compose_flag = 0;  
    keypad_flag = 0;  
    record_flag = !record_flag;  
    recording_key = -1;  
}
```

We wanted to travel between tone-generation, record, playback, and compose modes, so we created a flag just for each of them. Those flags are then read by other threads (say the record thread) for their condition of whether to perform the corresponding action. Flags were a simple low 0 or high 1.

The main portion in our code that controls the transitions between these modes occurs in the `MAYBE_NOT_PRESSED` state, since this is where key press/releases are detected. In this state, we check which key has been pressed/released and assign values to necessary flags. For example, if the key was 10 (or * on the keypad), the `record_flag` gets toggled to enter or exit record mode, while other flags get reset to 0 to ensure they are exited.

The ADC thread utilizes flags to determine what value `phase_incr_main` should be set to depending on which mode we are in. When in normal play mode or in record mode, the tone generator plays the usual frequency from the potentiometer. However, if the `playback_flag` is set, `phase_incr_main` reads from the `record_sound` list, which contains the frequencies that were stored on each key while in record mode. Similarly, `compose_playback_flag` sets `phase_incr_main` by iterating through the frequencies of each recorded key that was stored during composition.

We also created a separate record thread which is conditioned by the `record_flag` and a valid key press. This allows the audio to continue to be played while we record the frequency values on each key to store them for playback.

We debugged state transitions using `printf` statements! We added print statements at the conditional point (if statement) of the debounce FSM and throughout the FSM to check whether flags were being set correctly. However, they were quickly removed once verified, as sending heavy print statements over UART was slow enough to interfere with our threads.

System Design High-level Diagram

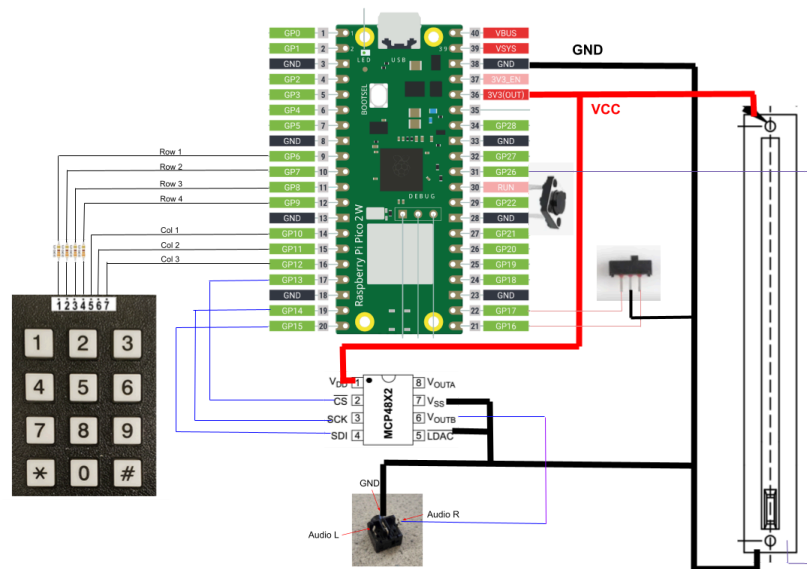


Figure 12: Wiring diagram

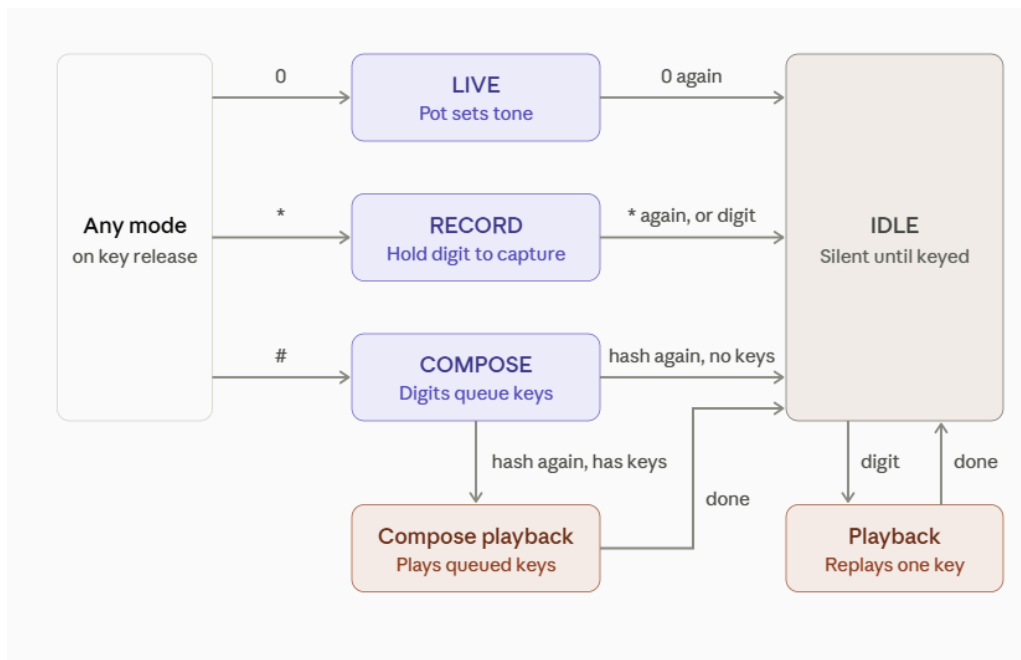


Figure 13: Main control FSM for Play, Record, and Compose

We took an incremental testing approach: Timer_Interrupt_DDS_Demo → ADC/Simple_Demo → Human_Input/Keypad. Each demo code was tested alone before we slowly added it back to the DDS demo, testing each combination one at a time, and figuring out how to mend the code accordingly.

Results

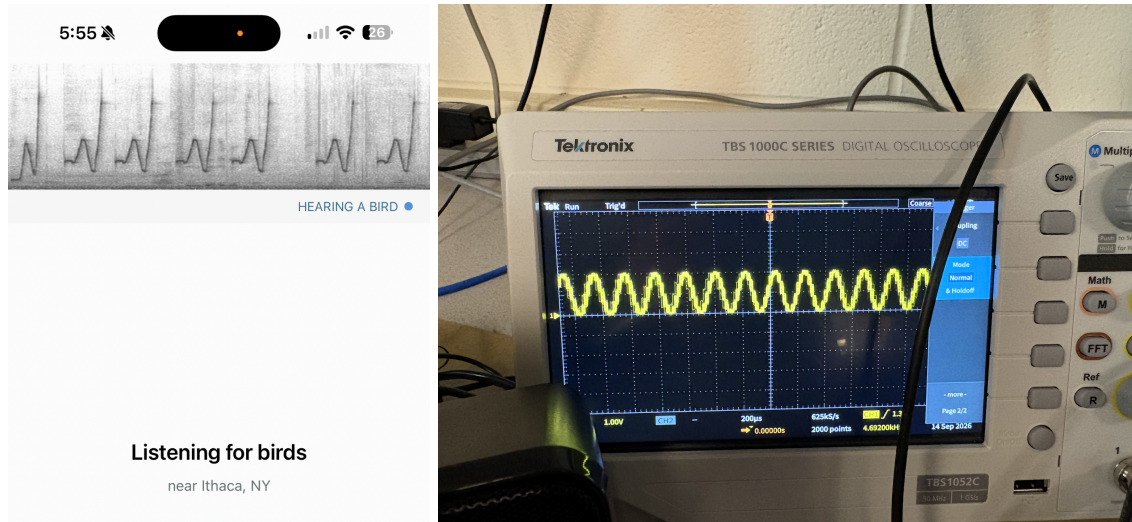


Figure 14:

Left:Merlin Bird

Right: Spectrogram of 4.7 kHz output from system

Conclusions

Though we failed at having Merlin detect us as a Northern Cardinal, we lowkey sound like Professor Batten singing Taylor Swift! We learned how to use protothreads to have multiple tasks running and to switch between tasks. We utilized mixed-signal hardware like an ADC and DAC, as well as the SPI communication protocol and an alarm ISR to generate sounds via DDS. Finally, we got to use some mechanical externals like a switch and potentiometer.

Appendix A: Heavily Commented Code

```

/**
 * V. Hunter Adams
 * DDS of sine wave on MCP4822 DAC w/ ISR
 *
 * Modified example code from Raspberry Pi
 * Copyright (c) 2020 Raspberry Pi (Trading) Ltd.
 *
 * SPDX-License-Identifier: BSD-3-Clause
 *
  GPIO 5 (pin 7) Chip select
  GPIO 6 (pin 9) SCK/spi0_sclk
  GPIO 7 (pin 10) MOSI/spi0_tx
  GPIO 2 (pin 4) GPIO output for timing ISR
  GPIO 3 (pin 5) !LDAC
  3.3v (pin 36) -> VCC on DAC
  GND (pin 3) -> GND on DAC
 */

// ===== include libraries =====
#include <stdio.h>                // used for printf
#include <math.h>                 // used for the sine func
#include <string.h>               // string stuff
#include "stdlib.h"              // always good to have
#include "pico/stdlib.h"         // Pico SDK added for GPIO and timing
#include "hardware/timer.h"      //ISR for DDS
#include "hardware/irq.h"        // IRQ
#include "hardware/spi.h"        // SPI stuff to send to DAC
#include "hardware/adc.h"        // ADC stuff to read from potentiometer
#include "hardware/gpio.h"       // GPIO stuff to read from keypad
#include "pt_cornell_rp2040_v1_4.h" // pt stands for "protothreads"
// =====end libraries =====

// Low-level alarm infrastructure we'll be using
// ISR = Interrupt Service Routine
#define ALARM_NUM 0
#define ALARM_IRQ timer_hardware_alarm_get_irq_num(timer_hw, ALARM_NUM)

//DDS parameters
#define two32 4294967296.0 // 2^32 as a constant for phase increment calculations
#define Fs 50000           // Sampling frequency (Hz)
#define DELAY 20 // 1/Fs (in microseconds)
// the DDS units:
volatile float adc_val;      // ADC value taken in from slide potentiometer
volatile float freq_val;     // desired frequency (Hz) from slide potentiometer
volatile float vol_val;      // volume scale factor from slide after toggling to volume control mode
// 32 bit accumulator (Note: overflows --> wrap back to 0)
volatile unsigned int phase_accum_main;
// freq*two32/Fs = phase increment assigned later
volatile unsigned int phase_incr_main;

// SPI data
uint16_t DAC_data ; // output value

//DAC parameters
// A-channel, 1x, active

```

```

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

//SPI configurations (GPIO pins)
#define PIN_MISO 4
#define LDAC 5
#define PIN_CS 13
#define PIN_SCK 14
#define PIN_MOSI 15
#define SPI_PORT spi1 // set as SPI channel 1
#define ADC_PIN 26
#define ADC_MUX 0

// Volumn toggle switch GPIO
#define V_SWITCH_1 16 // in this state: Volume
#define V_SWITCH_2 17 // in this state: Frequency

// Keypad config
#define BASE_KEYPAD_PIN 6 // base pin to add offset through GPIO 6, 7, 8, 9, 10, 11, 12
#define KEYROWS 4 // number of rows in the keypad
#define NUMKEYS 12 // number of keys in the keypad

// Record config
#define record_length 1000 // num samples stored for each recording
#define record_freq 10000 // recording freq (slow)
#define playback_freq 1000 // playback freq (fast)

// Compose config
#define compose_length 100 // num sound bites in one composition

// Keypad legend:
// 0x6E = 1 ; // 0x5E = 2 ; // 0x3E = 3
// 0x6D = 4 ; // 0x5D = 5 ; // 0x3D = 6
// 0x6B = 7 ; // 0x5B = 8 ; // 0x3B = 9
// 0x67 = * ; // 0x57 = 0 ; // 0x37 = #
unsigned int keycodes[NUMKEYS] = { 0x57, 0x6E, 0x5E, 0x3E, 0x6D,
                                   0x5D, 0x3D, 0x6B, 0x5B, 0x3B,
                                   0x67, 0x37} ;

unsigned int scancodes[KEYROWS] = { 0xE, 0xD, 0xB, 0x7} ;
unsigned int button = 0x70 ;

// what's this for? idk used to debug keypad probs
char keytext[40];
int prev_key = 0;

// FSM states for keypad debounce
typedef enum {
    NOT_PRESSED, // Idle... key is not pressed
    MAYBE_PRESSED, // Key is pressed, but we need to check if it's stable
    PRESSED, // Key is pressed and stable
    MAYBE_NOT_PRESSED // Key looked like it was released, but we need to check if it's stable
} debounce_state_t;

// Playback and recrding flags
static debounce_state_t key_state = NOT_PRESSED; // current state of FSM
static int possible = -1; // current keycode index
static int keypad_flag = 1; // flag to indicate if a key is pressed

```

```

static unsigned int record_flag = 0;           // flag to indicate if we are recording
// static int record_key[10] = {-1, -1, -1, -1, -1, -1, -1, -1, -1, -1};
static int record_idx = 0;                     // recording key/sound index
static int recording_key = -1;                 // flag to indicate which key is being recorded
static float record_sound[10][record_length] = {0.0}; // 100 freq for 10 sec recording
static int record_sound_idx[10] = {0};        // record end idx for each recording

static int playback_flag = 0; // flag to indicate if we are playing back a recorded sound
static int playback_key = -1; // flag to indicate which key is being played back

static unsigned int compose_flag = 0;
static int compose_key_seq[compose_length] = {0};
static int compose_key_seq_idx = 0;
static int compose_playback_flag = 0;

static float volume_scale = 1.0;

//GPIO for timing the ISR
#define ISR_GPIO 2

// DDS sine table
#define sine_table_size 256
volatile int sin_table[sine_table_size] ;

// Alarm ISR
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 ;

    // DDS phase and sine table lookup
    phase_accum_main += phase_incr_main ;
    DAC_data = (DAC_config_chan_B | (((uint16_t)(sin_table[phase_accum_main>>24]*vol_val) + 2048) & 0xffff)) ;

    // Perform an SPI transaction
    spi_write16_blocking(SPI_PORT, &DAC_data, 1) ;

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

}

// ADC Potentiometer thread
static PT_THREAD (protothread_FoutInput(struct pt *pt))
{
    PT_BEGIN(pt);

    static unsigned int i;
    static unsigned int k;

    while(1) {

        adc_val = adc_read();

```

```

if(gpio_get(V_SWITCH_1) && !gpio_get(V_SWITCH_2)) {
    vol_val = adc_val / 4096; // normalize to range 0.0 - 1.0
    // printf("adc_out: %d, vol scale: %f\n", adc_val, vol_val);
}
else if (gpio_get(V_SWITCH_2) && !gpio_get(V_SWITCH_1)){
    freq_val = adc_val * 2.5; // normalize to range 0 - 10k
    // printf("adc_out: %d, freq scale: %f\n", adc_val, freq_val);
}

// Print the value
if (keypad_flag || record_flag) { // key 0 pressed to play current sound controlled by potentiometer
    phase_incr_main = ((int)freq_val*two32)/Fs ; // update the phase increment
}
else if (!compose_playback_flag && playback_flag && playback_key != -1) { // play recorded sound with corresponding key
    for (i = 0; i < record_sound_idx[playback_key]; i++) {
        phase_incr_main = ((int)record_sound[playback_key][i]*two32)/Fs;
        // printf("Playing back on key: %d, i: %d, freq: %f, len:%d\n", playback_key, i, record_sound[playback_key][i],
record_sound_idx[playback_key]);
        PT_YIELD_usec(playback_freq);
    }
    if (i >= record_sound_idx[playback_key]) {
        // printf("Playback finished key %d, i=%d, len=%d\n", playback_key, i, record_sound_idx[playback_key]);
        playback_flag = 0;
        playback_key = -1;
    }
}
else if (compose_playback_flag) { // playback for song compose using record keys
    // printf("Compose playback start\n");
    for(k = 0; k < compose_key_seq_idx; k++) {
        // printf("Compose playing back on key: %d, len: %d\n", compose_key_seq[k], record_sound_idx[compose_key_seq[k]]);
        for (i = 0; i < record_sound_idx[compose_key_seq[k]]; i++) {
            phase_incr_main = ((int)record_sound[compose_key_seq[k]][i]*two32)/Fs;
            PT_YIELD_usec(playback_freq);
        }
    }
    compose_playback_flag = 0;
    // printf("Compose playing finished\n");
}
else {
    phase_incr_main = 0;
}

// Yield
PT_YIELD_usec(playback_freq) ;
} // END WHILE(1)
PT_END(pt);
}

void debounce_fsm_tick(int keycode) {
    switch (key_state) {
        case NOT_PRESSED:
            if (keycode != -1) { // button detected first time
                possible = keycode;
                key_state = MAYBE_PRESSED;
            }
            break;

        case MAYBE_PRESSED:
            if (keycode == possible) { // key press stable by current scan matched with first scan

```

```

        key_state = PRESSED;
        // printf("\nKey pressed: %d", possible);
    } else {
        key_state = NOT_PRESSED;
    }
    break;

case PRESSED:
    if (keycode != possible) { // key detected changed
        key_state = MAYBE_NOT_PRESSED;
    }
    if (possible != 10 && possible != 11 && possible != 0) {
        record_idx = possible;
    }
    break;

case MAYBE_NOT_PRESSED:
    if (keycode == possible) {
        key_state = PRESSED;
    } else { // key release detected
        // printf("\nKey released: %d\n", possible);
        if (possible == 0) {
            compose_flag = 0; // reset compose mode
            record_flag = 0; // reset record mode
            keypad_flag = !keypad_flag; // toggle play mode
            // printf("play mode toggled, keypad_flag: %d, record_flag: %d, compose_flag: %d, record_clear: %d\n", keypad_flag,
record_flag, compose_flag, recording_key);
        }
        else if (possible == 10) {
            compose_flag = 0; // reset compose mode
            keypad_flag = 0; // reset play mode
            record_flag = !record_flag; // toggle record mode
            recording_key = -1; // reset recording key
            // printf("record mode toggled, keypad_flag: %d, record_flag: %d, compose_flag: %d, record_clear: %d\n",
keypad_flag, record_flag, compose_flag, recording_key);
        }
        else if (record_flag && possible != 0 && possible != 11) {
            record_flag = 0; // end recording
            recording_key = -1; // reset recording key
            // printf("Record end, keypad_flag: %d, record_flag: %d, compose_flag: %d, record_clear: %d\n", keypad_flag,
record_flag, compose_flag, recording_key);
        }
        else if (!compose_flag && !record_flag && possible != 10 && possible != 0 && possible != 11 && possible != -1) {
            playback_flag = 1; // start playback of recordd sound
            playback_key = possible; // set playback key to the key presed
            // printf("Playback pressed, key: %d\n", possible);
        }
        else if (possible == 11) {
            record_flag = 0;
            keypad_flag = 0;
            compose_flag = !compose_flag;
            // printf("Compose mode toggled, compose_flag: %d\n", compose_flag);
            if (!compose_playback_flag && compose_key_seq_idx != 0) {
                compose_playback_flag = 1;
                // printf("Compose finished, sequence (%d keys): ", compose_key_seq_idx);
                // for (int j = 0; j < compose_key_seq_idx; j++) {
                //     printf("%d ", compose_key_seq[j]);
                // }
                // printf("\nPlayback start, keypad_flag: %d, record_flag: %d, compose_flag: %d, record_clear: %d\n",

```

```

keypad_flag, record_flag, compose_flag, recording_key);
    }
    else {
        compose_key_seq_idx = 0;
        // printf("Compose start");
    }
} else if(compose_flag && possible != 11 && possible != 0 && possible != 10) {
    if (compose_key_seq_idx >= compose_length) {
        compose_flag = !compose_flag;
        // printf("Compose sequence max length reached: %d\n", compose_length);
    }
    else {
        compose_key_seq[compose_key_seq_idx] = possible;
        compose_key_seq_idx++;
        // printf("Compose key sequence added: %d\n", possible);
    }
}

key_state = NOT_PRESSED;
}
break;
}
}

static PT_THREAD (protothread_key_debounce(struct pt *pt))
{
    PT_BEGIN(pt) ;

    static int i ;
    static uint32_t keypad ;

    while(1) {

        // Scan the keypad!
        for (i=0; i<KEYROWS; i++) {
            gpio_put_masked((0xF << BASE_KEYPAD_PIN),
                           (scancodes[i] << BASE_KEYPAD_PIN)) ;

            sleep_us(1) ;
            keypad = ((gpio_get_all() >> BASE_KEYPAD_PIN) & 0x7F) ;
            if ((~keypad) & button) break ;
        }
        if ((~keypad) & button) {
            for (i=0; i<NUMKEYS; i++) {
                if (keypad == keycodes[i]) break ;
            }
            if (i==NUMKEYS) (i = -1) ;
        }
        else (i=-1) ;

        //FSM update here
        debounce_fsm_tick(i);

        PT_YIELD_usec(30000) ;
    }
    PT_END(pt) ;
}

// Record thread
static PT_THREAD (protothread_record(struct pt *pt)){

```

```

PT_BEGIN(pt);

static int k;

while(1) { // record sound for the key pressed
    if (record_flag && key_state == PRESSED && possible != 10 && possible != 0 && possible != 11 && possible != -1) {
        k = possible;

        if (recording_key != k) {
            recording_key = k;          // new key to record on
            record_sound_idx[k] = 0;    // go back to first index of this keys recording
        }
        if (record_sound_idx[record_idx] >= record_length) { // record max length reached
            // printf("Record key: %d Max length recorded, end record\n", record_idx);
            record_flag = 0; //end reocridng
        }
        else {
            record_sound[record_idx][record_sound_idx[record_idx]] = freq_val; // record freq continuously
            record_sound_idx[record_idx]++; // move on to next recording index
            // printf("Record key: %d; current sound freq: %f\n", record_idx, freq_val);
        }
    }
    PT_YIELD_usec(record_freq);
}
PT_END(pt);
}

```

```

int main() {
    // Initialize stdio
    stdio_init_all();
    printf("Hello, DAC!\n");

    // 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);

    // Setup the ADC
    adc_init();
    adc_gpio_init(ADC_PIN);
    adc_select_input(ADC_MUX);

    // Setup volumn switch GPIO
    gpio_init(V_SWITCH_1);
    gpio_init(V_SWITCH_2);
    gpio_set_dir(V_SWITCH_1, GPIO_IN);
    gpio_set_dir(V_SWITCH_2, GPIO_IN);
    gpio_pull_up(V_SWITCH_1);
    gpio_pull_up(V_SWITCH_2);

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

    // MAP LDAC to GPIO
    gpio_init(LDAC);
    gpio_set_dir(LDAC, GPIO_OUT);
    gpio_put(LDAC, 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) ;

// Keypad setup
gpio_init_mask((0x7F << BASE_KEYPAD_PIN)) ;
gpio_set_dir((BASE_KEYPAD_PIN+4), GPIO_IN);
gpio_set_dir((BASE_KEYPAD_PIN+5), GPIO_IN);
gpio_set_dir((BASE_KEYPAD_PIN+6), GPIO_IN);
gpio_set_dir_out_masked((0xF << BASE_KEYPAD_PIN)) ;
gpio_put_masked((0xF << BASE_KEYPAD_PIN), (0xF << BASE_KEYPAD_PIN)) ;
gpio_pull_up((BASE_KEYPAD_PIN+4)) ;
gpio_pull_up((BASE_KEYPAD_PIN+5)) ;
gpio_pull_up((BASE_KEYPAD_PIN+6)) ;

// === build the sine lookup table =====
// scaled to produce values between 0 and 4096
int ii;
for (ii = 0; ii < sine_table_size; ii++){

    sin_table[ii] = (int)(2047*sin((float)ii*6.283/(float)sine_table_size));
}

// 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 ;

// === config threads =====
pt_add_thread(protothread_FoutInput); // ADC/freq/sound out thread
pt_add_thread(protothread_key_debounce); // keypad debounce FSM thread
pt_add_thread(protothread_record); // recording thread

// === initialize the scheduler =====
pt_schedule_start ;

// Nothing happening here
// unreachable
while(1){
}
return 0;
}

```

Appendix B: AI Chat Log

Though not ended up exactly in our final code, we prompted Claude Sonnet 5 (Medium Effort) to aid in our debouncing FSM. We then hand wrote the code based on the ideas provided by the chatbot:

"I'd like to implement [a debouncing state machine] into this code" -- referencing a 4-state FSM (Not pressed / Maybe pressed / Pressed / Maybe not pressed) described in the lab handout (Fig. 3), where a "possible" keycode is stored and confirmed across successive scans before a press event fires.

Response:

```
/* =====
 * BEGIN CLAUDE-GENERATED CODE
 *
 * Originating user prompt (paraphrased from chat history):
 * "I'd like to implement [a debouncing state machine] into
 * this code" -- referencing a 4-state FSM (Not pressed / Maybe
 * pressed / Pressed / Maybe not pressed) described in the lab
 * handout (Fig. 3), where a "possible" keycode is stored and
 * confirmed across successive scans before a press event fires.
 *
 * The states, transitions, and tick function below are new code
 * written by Claude (Anthropic) implementing that FSM in C.
 * ===== */
typedef enum {
    NOT_PRESSED,
    MAYBE_PRESSED,
    PRESSED,
    MAYBE_NOT_PRESSED
} debounce_state_t;

static debounce_state_t key_state = NOT_PRESSED;
static int possible = -1; // candidate keycode being confirmed

// Called once per scan with the raw keycode from this pass (-1 = nothing valid)
void debounce_fsm_tick(int keycode) {
    switch (key_state) {

        case NOT_PRESSED:
            if (keycode != -1) {
                possible = keycode;
                key_state = MAYBE_PRESSED;
            }
    }
```

```

        break;

    case MAYBE_PRESSED:
        if (keycode == possible) {
            key_state = PRESSED;
            // ---- single "key pressed" event fires here ----
            printf("\nKey pressed: %d", possible);
            // e.g. trigger a note: set phase_incr_main from possible, etc.
        } else {
            key_state = NOT_PRESSED;
        }
        break;

    case PRESSED:
        if (keycode != possible) {
            key_state = MAYBE_NOT_PRESSED;
        }
        break;

    case MAYBE_NOT_PRESSED:
        if (keycode == possible) {
            key_state = PRESSED;    // was just a blip, still held
        } else {
            key_state = NOT_PRESSED;
            // ---- optional "key released" event ----
            printf("\nKey released: %d", possible);
        }
        break;
    }
}
/* =====
 * END CLAUDE-GENERATED CODE
 * ===== */

```

Another Prompt: "please make comments in the code accordingly for where you generated code" (follow-up asking to wire the FSM into this loop). Replaces the original `printf("\n%d", i);` with a call into the debounce FSM, so a press registers as one clean event instead of printing every raw scan result.