ECE 4760 - Lab 1 Report

Angela Zou (az292), Kathleen Wang (kw456), Robby Huang (lh479)

Introduction

The purpose of this lab is to use direct digital synthesis to synthesize realistic birdsongs, specifically the song of the northern cardinal. We did this by decomposing a song into sound primitives (in our case, a low-frequency swoop, a chirp, and then a silence dividing each swoop/chirp combination) that we synthesized and applied an amplitude envelope separately to avoid non-natural clicks, before combining the sounds to recreate the song. The lab culminated in the user being able to use a keypad to control the audio output of a chirp, swoop, or period of silence in *play mode* and additionally record a sequence of chirps, swoops, and silences in *record mode* that will be played as soon as the user reenters the *play mode*.

Design and Testing Methods

General

To accomplish the goal of Lab 1, we started by looking at the deconstruction of the bird song (which is done in the lab page for us). In Week 1, we soldered the development board and tested its functionality on port expansion, digital to analog signal conversion, the TFT display, SPI communication, protothreading, and the interrupt service routine with various benchmarks provided to us. In Week 2, we started implementing the bird song synthesis (described further below) on the hardware and successfully integrated it with the keypad so certain keys made chirp and swoop sounds. Finally, in Week 3, we connected a switch to our circuit and used the ProtoThread library to add the functionality of being able to switch between *record* and *play* mode. We also thoroughly tested our implementation to make sure there were no software or hardware bugs. The details of the system are documented in the sections below.

Design - Hardware

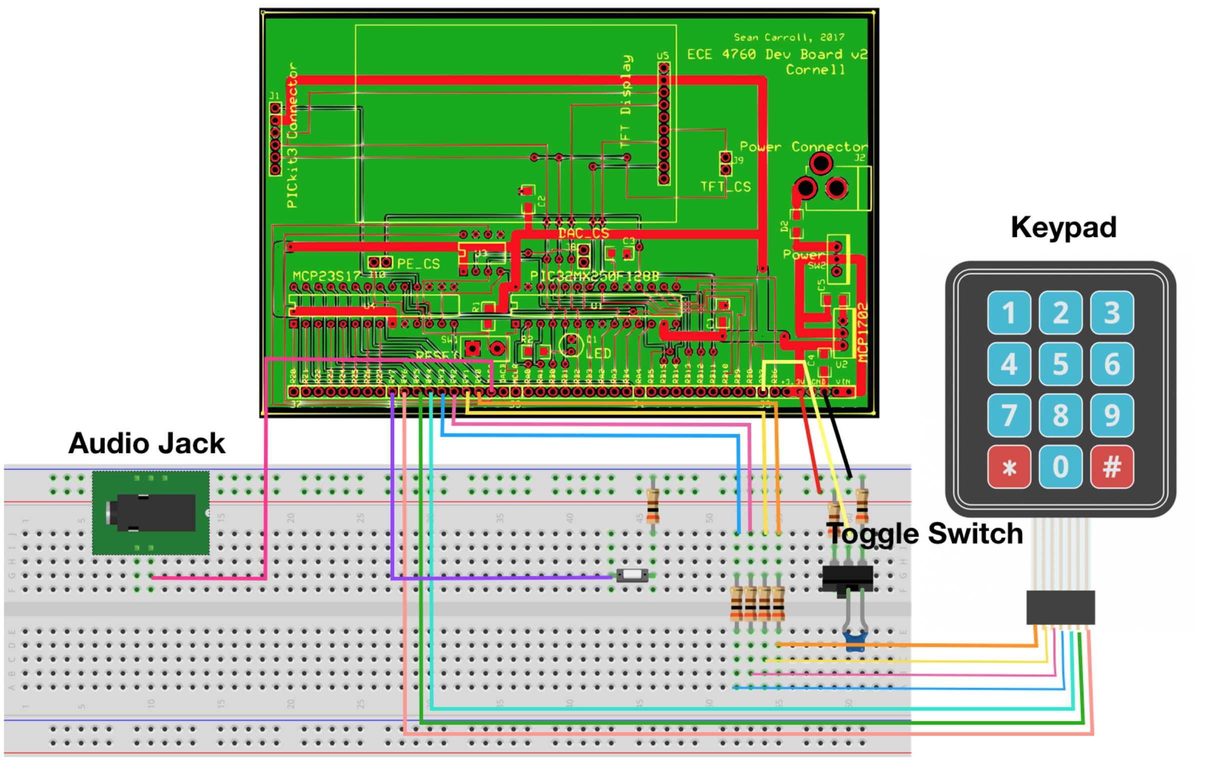

Figure 1: Hardware Setup

The hardware of the project includes the Course Development Board, its onboard components (PIC32 microcontroller, MCP4822 DAC, MCP23S17 I/O Port Expander, TFT Display, and LED), the PicKit3 Programmer, the audio socket, the keypad, and a switch for switching between play and record mode.

PIC32 is a 32-bit peripheral interface controller. It has a high-performance RISCV core, 2GB of user space memory, up to 5 external interrupts, and support for multiple communication protocols including UART, SPI, and I2C. With the PicKit3 Programmer, we can connect the microcontroller to PC and load programs with MPLABX IDE and XC32 compiler.

MCP4822 is the dual channel 12-bit Digital-to-Analog Converter (DAC) we use to convert the digital bird song synthesized in the PIC32 to analog audio signals. The DAC receives digital value from the microcontroller through the SPI channel, converts it into an analog waveform, and outputs the signal onto DACA and DACB pins. We can visualize the output on an oscilloscope for debugging. In the context of this lab, we also use an audio socket to play the bird song on a speaker.

The MCP23S17 I/O Port Expander can support up to 16 bits of remote bidirectional I/O ports. It allows us to connect more devices/connect to devices with more ports. In the context of this lab, we use the port expander to connect to the keypad which needs 7 pins.

The keypad is a combination of row and column circuits (circuit connection in *Figure 2*). Each switch is connected to one row wire and one column wire. The row wires are tied to digital output pins with 300ohm protection resistors to prevent the circuits from being shorted when multiple switches are closed. The column circuits are tied to digital input pins with 10 internal pull-up resistors. To determine which key is pressed, we programmatically shift a low pulse to Y0-Y3 while scanning PortY(Y0-Y6) input. The value is compared to the prewritten key lookup table to determine which keys are pressed.

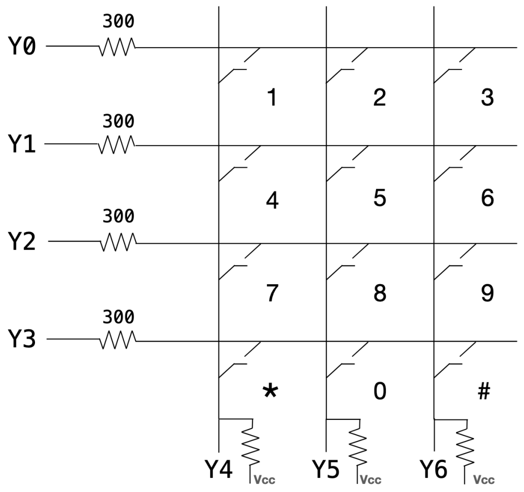

Figure 2: Keypad Internal Circuit Connection

The TFT display is a 2.2" 16-bit color TFT LCD that communicates with the PIC32 through SPI channel. We are provided with *printLine* functions to print messages onto the TFT at different positions. The LED soldered onto the development board is directly controlled by the PIC32 I/O. The TFT, along with the LED, are helpful tools for output and debugging.

To switch between the play mode and the record mode, we connected a toggle switch to a GPIO pin and used pullup and pulldown resistors for the two states. To debounce the toggle switch, we also added a ceramic capacitor between GND and the GPIO pin.

Design - Software

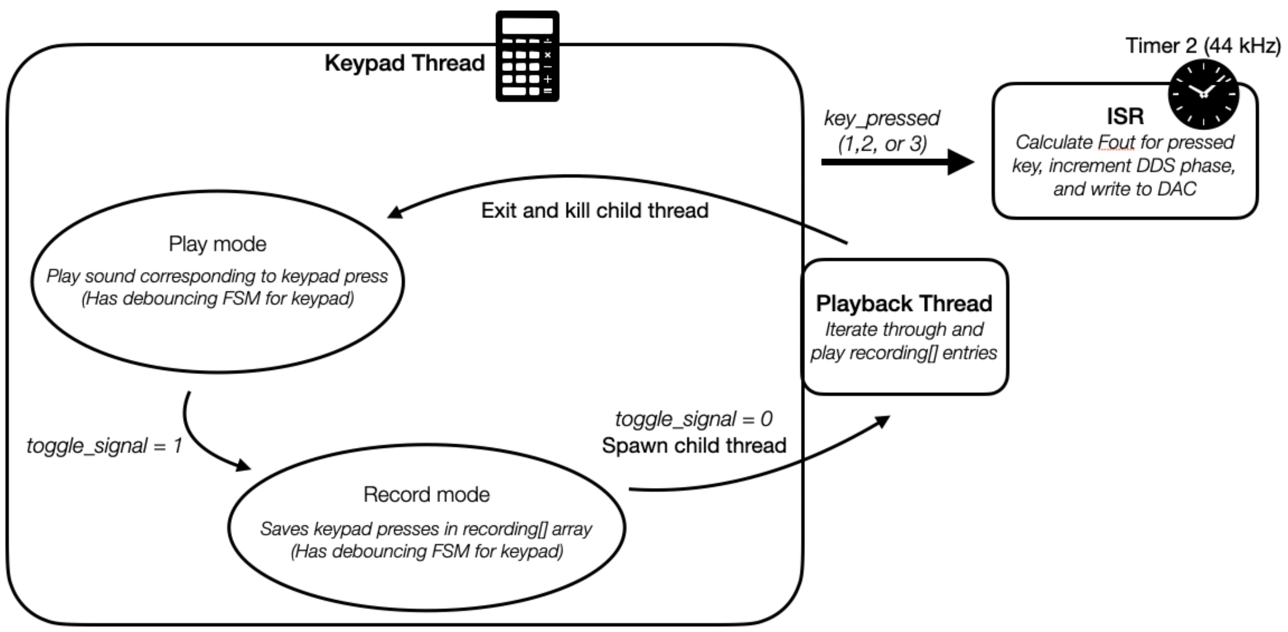
Keypad Thread
Timer 2 (44 kHz)
key_pressed (1,2, or 3)
ISR
Calculate Fout for pressed key, increment DDS phase, and write to DAC
Exit and kill child thread
Play mode
Play sound corresponding to keypad press (Has debouncing FSM for keypad)
Playback Thread
Iterate through and play recording[] entries
toggle_signal = 1
toggle_signal = 0
Spawn child thread
Record mode
Saves keypad presses in recording[] array (Has debouncing FSM for keypad)

Figure 3: High Level Diagram of the Software

Project Setup

The basis of the software framework is the ProtoThread library. This is a lightweight threading library that allows us to schedule and executes multiple threads. Different functionalities are implemented within different threads to achieve the modularity and encapsulation of the design. At the same time, threads communicate with each other through global variables.

To support the microcontroller, the ProtoThread library, and the hardware features we needed, we included several header files to configure the clock (`config_1_3_2.h`), setup port expander (`port_expander_brl4.h`), initialize and schedule the thread (`pt_cornell_1_3_2.h`), and communicate with the TFT (`tft_gfx.h`, `tft_master.h`). The corresponding TFT and port expander functions are implemented in `tft_gfx.c`, `tft_master.c`, and `port_expander_brl4.c`. Additionally, `glcdfont.c` provides a lookup table for the Standard ASCII fonts and the main functioning threads are adapted from `DDS_Accum_Expander_BRL4.c`, which generates a constant frequency through Direct Digital Synthesis.

Direct Digital Synthesis

To successfully synthesize a bird song, the methodology is to convert the song into a frequency-time spectrogram and decompose it into digital synthesizable primitives. The northern cardinal song, which is “almost pure frequency-modulated tones”, can be broken down into three sound primitives: “a low-frequency swoop at the beginning of each call, a chirp after each swoop which moves rapidly from low frequency to high frequency, and silence which separates each swoop/chirp combination”. *Figure 4* shows the spectrogram of the song which we will compare our result with.

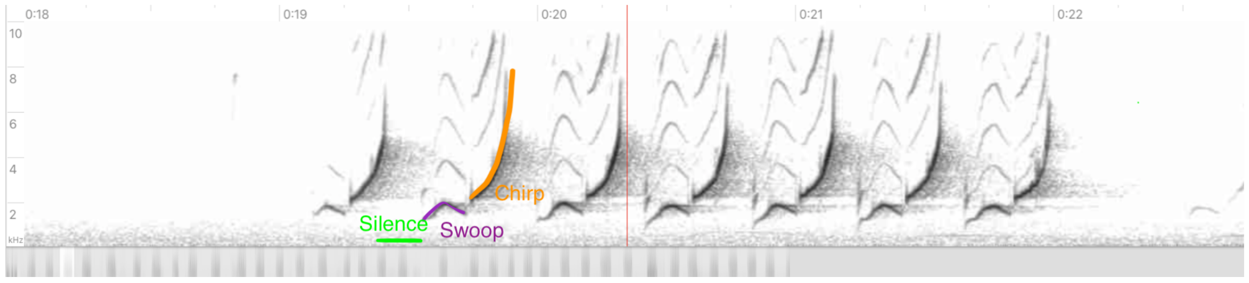

Figure 4. Spectrogram of the Bird Song

An analysis of the three song primitives indicates that each segment has a length of 130ms. As the DAC gathers audio samples at 44kHz, each segment will have 5720 sample points. To mathematically represent the relationship between frequency and time/sample for each segment, we approximated the swoop with a sine function and the chirp with a quadratic function:

$$f_{swoop} = -260 sin(-\frac{\pi}{5720}x) + 1740 \quad for\ x \in [0,\ 5720]$$

$$f_{chirp} = (1.53 \times 10^{-4})\, x^2 + 2000 \quad for\ x \in [0,\ 5720]$$

The most straightforward way for generating synthesized sound through digital devices is to convert the frequency function into an array where each entry represents the frequency value at each sampling point. The disadvantage of this method is that the size of the array grows linearly with the length of the sound and the resolution of the frequency. Thus, we implemented our bird song with a more storage-efficient algorithm: Direct Digital Synthesis (DDS).

The basis of the DDS algorithm is the phasor. A rotation of a phasor is isomorphic to the overflowing of a variable. Changing the angle of the phasor (the accumulator) changes the phasor value. For an accumulator with 32-bit resolution, assuming we want to change N units in the accumulator for 1 audio sample, the output frequency can be expressed in the following formula:

$$F_{out} = \frac{1\ overflow(sine\ period)}{2^{32}\ accumulator\ units} \frac{N\ accumulator\ units}{1\ audio\ sample} \frac{F_s\ audio\ samples}{1\ sec} = (\frac{F_s}{2^{32}}N)Hz$$

As we already know the output frequency we want, we can easily express N as:

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

This allows us to only create a sine table and index into it by increasing N in the index for each audio sample. While our sine lookup table can, in principle, take on 2^{32} different states, we are able to get away with just 256 entries. While fewer entries in the sine table cause more harmonic distortion, 256 entries seem to be where the first error harmonic is not too noticeable in our DAC output. This is also the reason why we left shift our 32-bit accumulator by 24 bits since we only need the most significant 8 bits of our accumulator to index into our sine table.

Synthesis ISR Thread

To guarantee a steady output of the sound wave, we have to iterate through the audio sample at a constant speed. This is done in the Interrupt Service Routine (ISR) with a timer trigger.

The ISR we use for DDS is triggered by Timer 2 at a 44kHz synthesis sample rate. When we enter the interrupt routine, we first clear the interrupt flag. This is to prevent the ISR from interrupting on the old flag when it exits the previous interrupt routine. Then it reads the global variable `key_pressed` which is set by the *keypad thread* whenever a key is pressed. The `Fout` variable, which denotes the output frequency of the sound wave, is set to swoop for *key1*, chirp for *key2*, and silence for *key3*.

Knowing the output frequency, we call on the DDS algorithm to index into the pre-generated sine table. The increment N is calculated using the formula $F_{out} \frac{2^{32}}{F_s} = F_{out}(2^{32}/44000)$ and added to the previous phase value. Since the sine table only has $2^8 = 256$ entries, we left shift the 32-bit `DDS_phase` value and only use the upper 8 bits for indexing. This sine table entry value is then written into the Digital Analog Converter (DAC) output. Since frequency has to be positive to produce sound, we leveled the DDS output to make sure it stays at the upper half of the DAC range (0-4096).

To avoid non-natural clicks between sound primitives and make sure we only generate the audio when keys are pressed, we implemented a sound envelope in the ISR. A linear ramp function modulates the 5720 audio samples for each of the sound segments: `note_time` starts at 0, incrementing 1 sample at a time to point at the current audio sample point we need to process until it reaches 5720. `current_amplitude` ramps up to the maximum amplitude in the first 1000 samples, sustains the amplitude for 3720 samples and then ramps down in the last 1000 samples. The amplitude is set to 0 (meaning no sound is outputted) for the remaining time when `note_time` is greater than 5720. The actual audio generated is output to DAC A while DAC B only has the amplitude. By connecting DAC B to the oscilloscope we can use the amplitude envelope for debugging purposes.

Keypad Thread

The main goal of the keypad thread is to set the global variable `key_pressed` to tell the ISR which frequency it needs to output. It is responsible for reading the keypad input, debouncing the press to identify the keypress, and determining the *play/record mode* selection.

To detect the keypress, we implemented a keytable which is a 24 elements array that has already hardcoded all the possible input from the keypad including the 12 buttons and shift + the 12 buttons. In a for loop, the program shifts a low pulse through Y0-Y3 and reads the output value on the Y port. During the scanning of the keypad SPI transmission is blocked in case it might interrupt the keypad reading. If a key press is detected, the program searches for the keycode that matches the entries in the keytable array. In the birdsong synthesis scenario, we only recognize keypresses of 1, 2, and 3 and treat the others as invalid.

As the keys serve as digital inputs to the microcontroller, we need to denounce the detected keypresses so that glitches in the signal will not be double counted. The FSM below details the debouncing logic. `NOT_PRESSED` serves as the idle state, meaning none of the assigned keys are pressed. Any time the keypad signal is not 1, 2, or 3, we transition back to the `NOT_PRESSED` state. `MAYBE_PRESSED` indicates the first press of a key in the assigned key group. It denounces the 0-1-0 glitch in the keypad signal, preventing it from being recognized as a valid press. Anytime we detect a new keypress of 1, 2, or 3, we transition to this state. The `PRESS` state is the key registering state. The only way to enter this state is to go through the `MAYBE_PRESSED` buffering state to enforce the debouncing. Any press of the same key afterward will advance the PRESS state into `PRESSED` state. The only way to go back to PRESS state is to go through `NOT_PRESSED` or `MAYBE_PRESSED`, which performs the debouncing of 1-0-1 glitches.

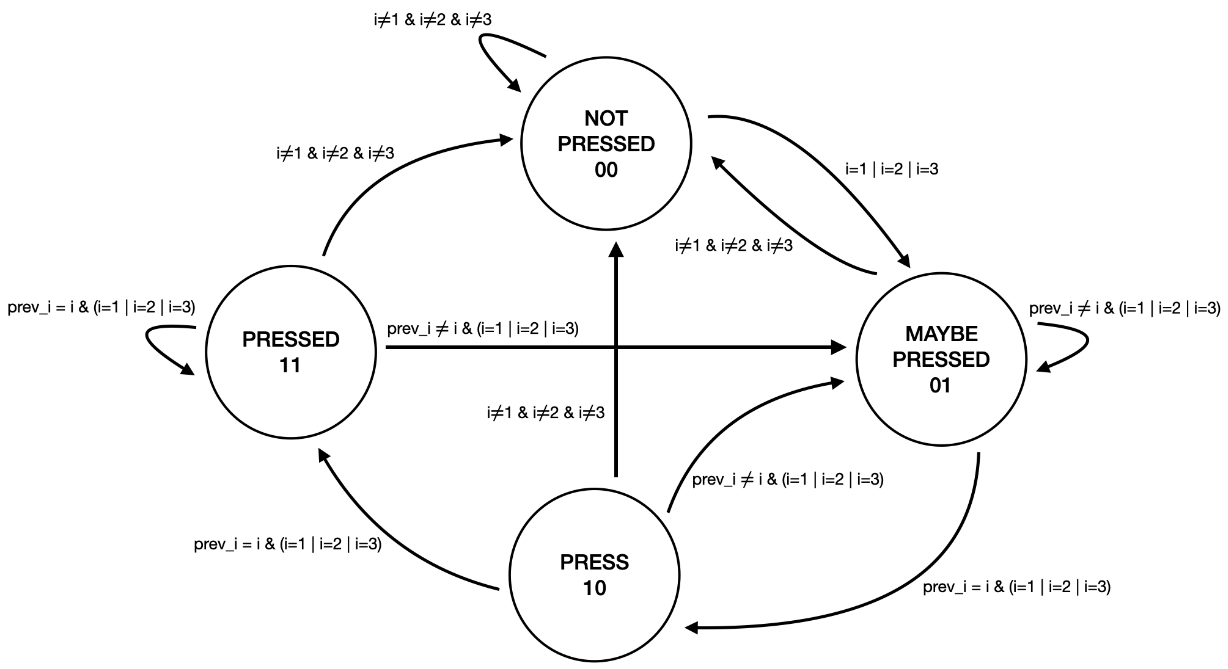

Figure 5. Keypad debouncing FSM

While the keypress is registered in the `PRESS` state, we handle the logic differently based on which mode we are in. In the *play* mode, we assign the pressed key value of the `key_pressed` variable and set both `note_time` and `current_amplitude` to 0. `key_pressed` value indicates the ISR which frequency to output and zeroing `note_time` start the amplitude envelope iteration.

If the `toggle_signal` is high, we know we entered the *record* mode. After performing the debouncing FSM, we store the pressed key in the `recording` array. To prevent the index out of bounds error, we mod the `recording_index` with the preallocated array size of 30.

While we transition from record mode to play mode, we play the recorded sound sequence automatically by spawning a child *playback* thread.

Playback Child Thread

Once the user records a song in record mode and switches back to play mode, we want the board to then automatically playback the song that had just been recorded. During this playback, we don't want the user to be able to engage in typical play mode (generating swoops or chirps when they press keys), meaning we had to block typical play mode while the song was being played back.

Because of this, we chose to spawn a child thread within the keypad when `toggle_signal` is switched from 1 to 0, meaning we are switching from *record* to *play* mode. This way, the child playback thread blocks its parent thread (the main *keypad* thread) until it exits. Besides, compared to a normally scheduled thread that yields every time when it should not be played, a child thread avoids the extra CPU cycle on context switching.

In the thread, we iterate through the `recording` array, set the global variable `key_pressed` to the entry value, and signal the ISR to generate the sound (by setting `note_time` and `current_amplitude` to 0). While the Timer 2 ISR picks and plays the sound primitive based on `key_pressed,` we spin in a while loop until `note_time` reaches 5720, meaning the ISR finishes playing the sound. When the `play_index` catches up with the `recording_index` (last item in the array), we know we finished processing all primitives being recorded. Then we reset the helper array index variables, exit the *child* thread, and unblock the *keypad* thread. The *keypad* thread can then proceed normally into the *play* mode.

Main Function

The first part of the `Main` function is initialization since it is the first code segment being executed. We firstly set up the analog selection pins and the SPI communication to the DAC, including MOSI, chip select, and clock divider.

As Timer 1 is being used by the Protothreads library provided, we set up the next available timer (Timer2) with a priority of 2 in order to trigger an interrupt at an exact 44kHz synthesis sample rate. We also clear the interrupt flag for Timer 2 and enable system-wide interrupts.

Additionally, we do some of the setup for the DDS algorithm. In particular, we first build the sine lookup table, scaling it between 0 and 4096 to match the output range of the DAC. The sine table has $2^8 = 256$ entries, corresponding to the upper 8 bits of the `DDS_phase` being used to index into the table. We also zero the array which keeps track of which keys have been pressed in *record* mode. This is also where we build the amplitude envelope parameters and check that they are in range (not less than 1), as well as set up increments for calculating the bow envelope. To make sure the TFT screen can be used for debugging purposes, we next initialize the TFT display attached to our main board to display a black screen and set the orientation to vertical.

Lastly, we initialize and schedule our *keypad* and *timer* threads. We use a round robin scheduler, so each thread gets its time to run in a cyclic way. In our case, we are looping through two threads: *protothread_key* (which is relevant to this lab and used for running all our keypad and play/record mode logic) and *timer* thread which is simply used by us to help with debugging.

Debugging, Testing, and Optimizations

Our main methods for debugging consisted of printing on the TFT display, using an oscilloscope to probe the DAC output, using a speaker to play the DAC audio output, using a helper timer thread, and blinking the LED. In particular for the TFT display, we used a helper function called `printLine2` from the example code to print out important variable values for debugging purposes.

We tested the functionality of our code incrementally throughout the lab, starting off by confirming that the TFT and the basic keypad interface was functional by printing the pressed key value to TFT, meaning that we were able to read in the correct keys.

Our next step was to map sounds to keys of the keypad. We first confirmed we could map a singular tone to each of our three keys before we focused on generating our swoop and chirp functions correctly in our DDS ISR, which is also where we made our optimizations to cut down on our computation time. We examined these waveforms on the oscilloscope as well as through the audio output after connecting a speaker. Since this is the *play* mode implementation, both hearing one sound primitive and observing 1 waveform per keypad press prove that our debouncing FSM is functioning properly.

As the CPU is running at 40 MHz and we need a standard audio rate of 44 KHz, we have a max of 909 cycles to spend in each ISR cycle. This required us to optimize our code and minimize our computation time in the DDS ISR in order to meet the timing deadline for the interrupt. Before making the following optimizations, the board was constantly resetting on the swoop swoop (which we could deduce due to the inactive TFT display).

As this issue only affects our swoop sound but not our chirp, we suspected that the computation of the swoop frequency takes longer than the chirp. Since the chirp was modeled by a quadratic function, which is faster than the sine function the swoop was originally approximated by, we try to mitigate the issue by approximating the swoop as a quadratic function as shown in *Figure 6.* Additionally, we define the DDS increment constant $\frac{2^{32}}{F_s}$, thinking this can precompute the value.

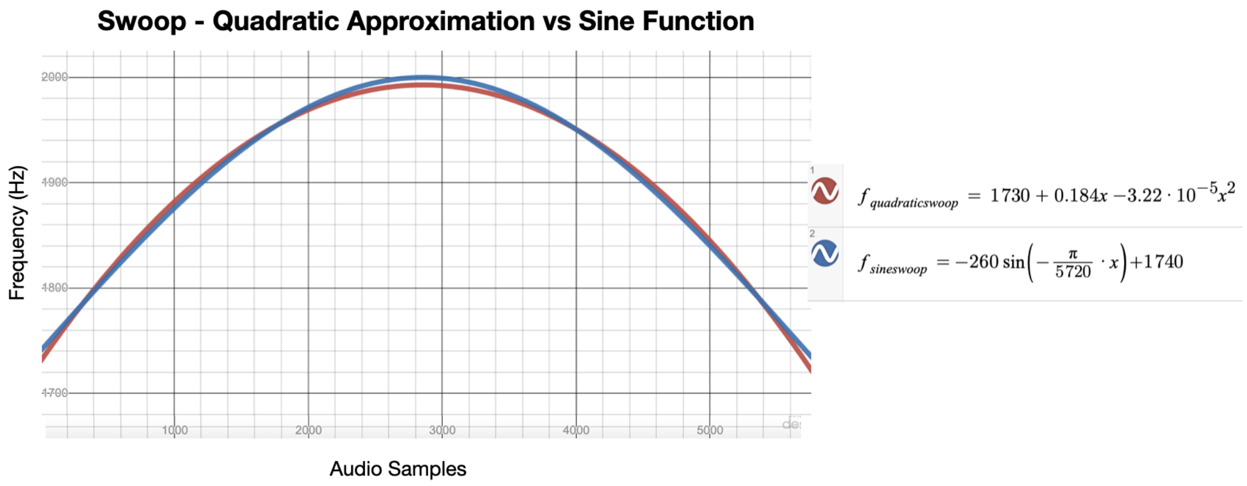

Figure 6. Swoop Function Approximations

After we made those modifications, the TFT stopped freezing, yet still refreshed at an extremely slow rate. We later realized that "`#define DDS_constant 4294967296.0/44000.0`" only acts as a text replacement instead of a precalculation, meaning we were still performing the floating-point computation every time. By assigning `DDS_constant` to a global float variable, our program ran a lot faster.

For Week 3, we first confirmed the functionality of the switch we were attaching to switch between *play/record* mode before implementing our child playbreak thread. This is done by turning on the LED for *record* mode and off food *play* mode. One major bug we faced was with our child playback thread. When transitioning from the *record* mode back to the *play* mode, the recorded sequence was not played. We added a TFT print function in the playback child thread and realized that we never got a chance to enter the child thread. The issue ended up being that we were using Bit 7 of Port B to read in the switch state, meaning the read value should be 128($7'b10000000$) instead of 1 when it is HIGH. After resolving this issue, our code was able to successfully enter the child thread.

Besides, as we need to use the keypad to control the audio output by registering glitchless keypresses, We also tested the implementation of our debouncing FSM through the TFT by displaying the tail index in the `recording` array. Each increment indicates that 1 keypress is registered. As long as the displayed number matches with our keypresses, we can confirm that our debouncing was successful.

To confirm the functionality of our birdsong synthesis, we played random sequences of birdsong primitives and compared the output audio with the actual birdsong. We also unit tested our *play*/*record* mode transitions and the playback of our recorded array. We also checked the DAC output on the oscilloscope and the spectrogram of our output.

Results

The objective of the lab was to be able to generate accurate DAC outputs for swoop and chirp bird sounds that correspond to specific keys on the keypad, as well as implement a *play* mode and a *record mode*.

During the checkout, our system was tested for reliability by the TA. In *play* mode, when pressing keys 1, 2, and 3 multiple times in different orders, we hear no errors in the audio output as well as the waveform on the scope. Additionally, all other keys are invalidated in the software and no functions were affected. For *record* mode, we recorded sequences that were different in length and order. Our system replayed all the test cases without any errors, up to an input buffer length of 30 sounds (which is our array size for holding the recorded entries).

To test the accuracy of the output, we used the soundcard of the lab computer and the Waveform software to obtain the scope display of our audio output (See *Figure 7*). We could confirm from the scope that each sound played for 130 ms, exactly once, independent of the duration of the button push, indicating our debouncing FSM was working as intended. Additionally, we checked the amplitude envelope and confirmed through the audio output of a speaker that there were no clicks, pops, or other audio artifacts.

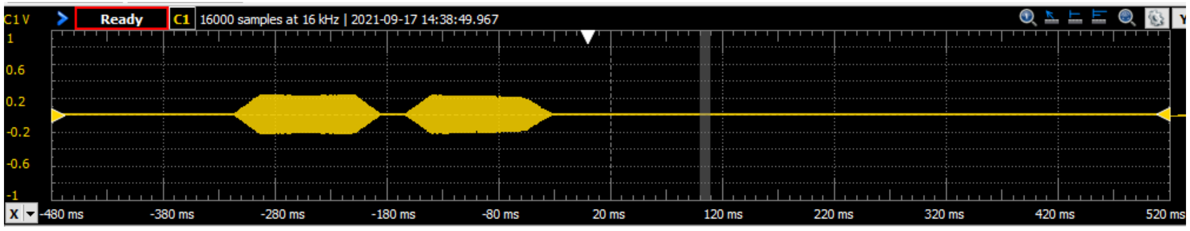

Figure 7. Scope Display of One Swoop and One Chirp

The spectrogram (*Figure 8*) visualized the frequency of our audio over time. With our spectrogram, we can directly compare our output with the spectrogram of the northern cardinal's recording in *Figure 4*. As shown, our synthesized output is very similar.

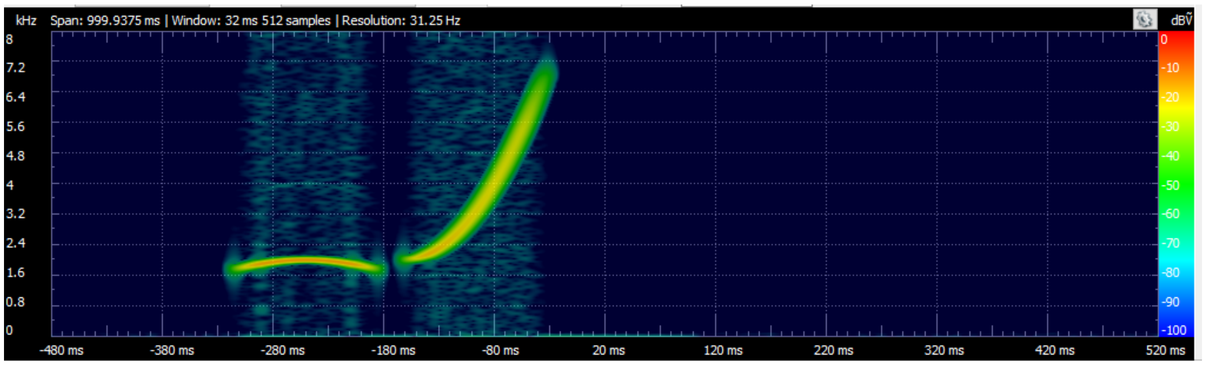

Figure 8. Spectrogram of One Swoop and One Chirp (0 to -100 dBv shown)

A potential area for improvement was with the sound quality of the birdsong we generated. While our generated output was confirmed to be correct, we should note that the swoop and chirp we generated seem to demonstrate some noise that slightly affects the audio of our DAC output. In the 0 to -100 dBv spectrogram, we see broadband noise frequency, which we suspect comes from interference of other electronic devices in the lab. By changing the signal strength sensitivity to a range of 0 to -40 dBv (shown in *Figure 9*), our output audio is clean without the distortion. Though we were assured by Professor Adams and Bruce that this minor interference was acceptable and has negligible influence on our sound quality, it is still an area that could be investigated.

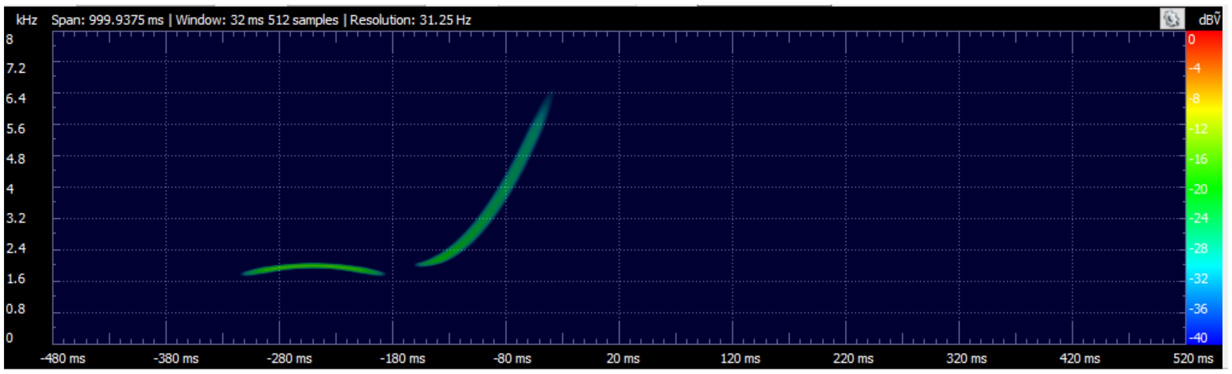

Figure 9. Spectrogram of One Swoop and One Chirp (0 to -40 dBv shown)

Besides accuracy, the performance of our system can be also measured by the time spent in the ISR. We know if we spend more than ~850 cycles in the ISR, the TFT will start to refresh at a noticeably slower rate and the audio output may become distorted. At first, our swoop function was over the max limit of ISR cycle length, but with our previously mentioned optimization in our computation of `DDS_constant`, we reduced this to around 730 cycles. We later further optimized it to around 680 cycles by using a quadratic equation to approximate the sine function of the swoop.

In terms of unique features in our code, our approach slightly differs from that of the suggested program organization as follows. Instead of scheduling both the keypad and playback threads in the main function and using a semaphore from the keypad thread to signal the playback thread to start, we instead opted to spawn the playback thread as a child thread of the keypad thread. Instead of a scheduled thread (which acts as an independent flow of control), a spawn thread blocks only the parent thread until it exits, which in our opinion simplified switching between threads.

Conclusions

Overall, we consider our system to be successful in fulfilling the requirements of the lab handout. The interface with a toggle switch, a keypad, an audio jack and an indicator LED is relatively user-friendly. Additionally, we paid attention to scalability while programming. For example, the state machine we designed for debouncing the three buttons on the keypad can be easily scaled up to be used to debounce more buttons. It is also quite easy to assign a key to output a different sound, which we tried when debugging.

In terms of what we learned, direct digital synthesis was an interesting concept and has made us more aware of how digital sounds are generated and able to sound realistic. In terms of memory space optimization, DDS serves as an example of using phasor indexing and lookup tables to generate output value. We also learned some important lessons on the underlying logic behind C syntax (especially the difference between how the compiler handles `#define` and a constant variable declaration). Additionally, since we had to meet the timing deadline for our ISR, we considered optimization in code and ways to make computations more efficient.

Issues we faced during the lab include not being able to hear the audio output due to over spending time in ISR, debouncing for both play mode and record mode, and incorrectly processing digital input `HIGH` signal. We discussed how we identified the bugs and what our solution was in the optimization and debugging section.

In terms of further improvements we could have made in our implementation, we tried balancing complexity and code efficiency during our implementation but we did not do all the possible optimizations. For example, we could have used a look-up table to replace the swoop equation to further reduce the time in ISR. We could also have used a Mealy machine instead of a Moore Machine for debouncing, which would have enabled us to reduce the number of states in the state machine.

If we were to keep improving on this project, here are some potential interesting improvements we can make: 1. Use python to make a GUI with different slider bars to control the parameters such amplitude envelope shape, bird song duration, and frequency shifting. 2. Add audio analysis feature to allow users to have actual harmonic modulatable sounds as input and the system to output digital synthesized audios. 3.

Add spatial audio into the design so that the users will feel like the ‘birds’ are moving around them.