ECE 4760 Lab 1 Report: Bird Song Synthesizer

By Morgan Cupp (mmc274), Stefen Pegels (sgp62), and Maria Martucci (mlm423)

Introduction

In this lab, we used the PIC32 processor to synthesize the birdsong of a northern cardinal. Its call can be decomposed into swoops, chirps, and pauses, which we synthesized separately using direct digital synthesis. The phase of the desired output sine wave was calculated incrementally using a sine table lookup, and then sent to the digital to analog converter (DAC) over an SPI channel which output the analog sound. We also used a varying amplitude multiplier on the analog sine wave to perform amplitude modulation, ramping the swoop and chirp frequencies in an envelope that made the bird calls sound smooth to the human ear. The user interacted with the program through a python interface that also communicated with the program over serial and sent when, for example, a button was clicked. The interface also had a toggle for record mode or play mode, where in record mode the user could press a sequence of buttons and their actions would be saved and then played back to them in play mode. The series of swoop-chirp-pause, swoop-chirp-pause creates the northern cardinal's distinct birdcall.

Design and Testing

Concept

To accomplish the goal of synthesizing bird sounds, the DDS algorithm had to be implemented. The sounds all last 130 ms, but the DAC expects discrete samples that will approximate the sound waves at a rate of 44 kHz. Thus, a 130 ms audio signal will consist of 0.130 sec x 44000 samples/sec = 5720 samples. To generate these samples, functions were provided in the lab handout that plot each sound's frequency as a function of sample number over the range 0 to 5720. The objective of DDS is to turn this frequency information into an analog signal.

At a high level, the DDS algorithm works as follows. First, a phase accumulator variable is zeroed. The sound to be synthesized is then selected, and it has a corresponding frequency vs. sample number function. Next, for each sample, a phase increment value is computed based on the unique frequency vs. sample number function, and this increment value is added to the phase accumulator (*see Figure 1 - helper method that calculates the correct increment value*). Lastly, the current value of the phase accumulator is used as a lookup into a sine table, and the sine table outputs the value to be sent to the DAC (*see Figure 4*). As this process occurs 5720 times over the course of 130 ms, the DAC approximates the analog audio signal of a bird sound.

```
double dds_increment(){
    double frequency;
    switch (sound_type)
    {
        case 0://pause
            return 0.0;
        case 1: //swoop
            frequency = -260 * sin(sin_freq * isr_counter) + 1740;
            return frequency * two32divFs;
        case 2: //chirp
            frequency = (1.53e-4) * (isr_counter * isr_counter) + 2000;
            return frequency * two32divFs;
    }
}
```

Figure 1 - Function for computing the amount by which the accumulator will be incremented.

One option was to compute all of the DAC values at once, and then play the sound. However, this would require too much memory, especially when in record mode. To save memory, DAC values are computed in "real time" as they are needed so that they do not need to be stored. This also makes the system more responsive since it can immediately start playing the first DAC value rather than pre-computing every DAC value.

For this to work, a new DAC value must be ready every 1/44000 seconds. This suggested the use of an ISR triggered by a timer interrupt set to go off every 1/44000 seconds. By sending a new value to the DAC within every ISR, the audio samples would reliably be set at the correct rate. One more challenge requiring special attention was to make sure the computation of the next DAC value happened quickly. Between timer interrupts, the next DAC value had to be computed while also leaving enough time for the other threads to run. The most clear way to make the code fast enough was to avoid floating point arithmetic. However, floating point arithmetic ended up being fast enough, so it was used to keep the code simple and accurate.

To make the sounds sound better, an amplitude modulation envelope was used. The DDS algorithm generates a sine wave with a constant amplitude, but going from zero sound to a high amplitude creates a "pop" sound. Thus, the linear modulation envelope steadily transitioned the amplitude from zero to the maximum in the beginning and back to zero at the end. This got rid of the pops due to rapid amplitude variation.

The DDS computations could have been placed in either a thread or the ISR. We chose to place them in the ISR since the DDS computations must occur every 1/44000 seconds in order to generate a DAC value. Furthermore, these DDS computations must complete before the next timer interrupt. Since they are so important, doing them immediately in the ISR where thread preemption could not occur seemed reasonable.

Implementation

The core of the software implementation is centered around the global variable isr_counter. This variable acts as a counter of the number of samples sent to the DAC for one sound. To start the playing of a sound, isr_counter is set to 0 and the global variable sound_type is set to 0 for pause, 1 for swoop, or 2 for chirp. This means that on the next entry to the ISR, the isr_counter will pass a check that it is below 5720. 5720 corresponds to the length of the sound duration. Once past the check, isr_counter is incremented, and the DAC data is then calculated based on the sound type (see *Concept* section above) and sent over the SPI channel to the DAC, producing analog output.

Thus for the next 5719 entries into the ISR, the DAC is sent the incremental pieces of the swoop or chirp call, and isr_counter is incremented. Once isr_counter==5720, the sound has completed and been heard by the user, and the isr_counter remains constant until it is reset to zero by another event.

The implementation also relies on the button and toggle threads to act on the user input received from the python interface. The python interface creates the layout of buttons and then upon an event, writes a string via serial to the PIC32 that encodes for example, which button was pressed, the button id number, and the button value (1 for pressed and 0 for released). The serial thread reads the input string and decodes it, for example, into a button press, then signals the button thread to run by setting a variable new_button. The button thread waits for this new_button signal and yields at all other times.

The toggle thread similarly waits for a new event to come in from the python interface signaling that the user has switched to either record mode or play mode. The toggle thread sets the value of a global variable mode to be 1 for record mode and 0 for play mode. When there is a button press, the button thread checks the mode of the program. If the program is in play mode, it checks the button id of the pressed button, and sets the sound_type to match which button the user pressed and wants to hear. It sets the isr_counter to zero to start the sound being played immediately. If the program is in record mode, it fills an array of button presses (presses_array) with the current sound_type of the button pressed, and increments the presses_index that keeps track of how filled the presses_array is. In this way, the presses_array keeps track of the sequence of buttons the user pressed during record mode by saving the sound type the user will want to hear when switched to play mode.

The playmode thread handles the playing of the sequence of recorded button presses. It yields until record mode is exited and the array that stores the sequence of button presses has at least one entry. When these conditions are met, it reads the first entry in the presses_array. It sets the sound_type to be the one stored in the array and sets the isr_counter to zero, so that when the ISR is entered it will start playing the sound. It waits for the sound to finish by stalling in the loop until isr_counter reaches 5720, meaning that the sound has fully played. It then processes remaining sounds in the presses_array in the same manner.

Below is a diagram of the main functionality:

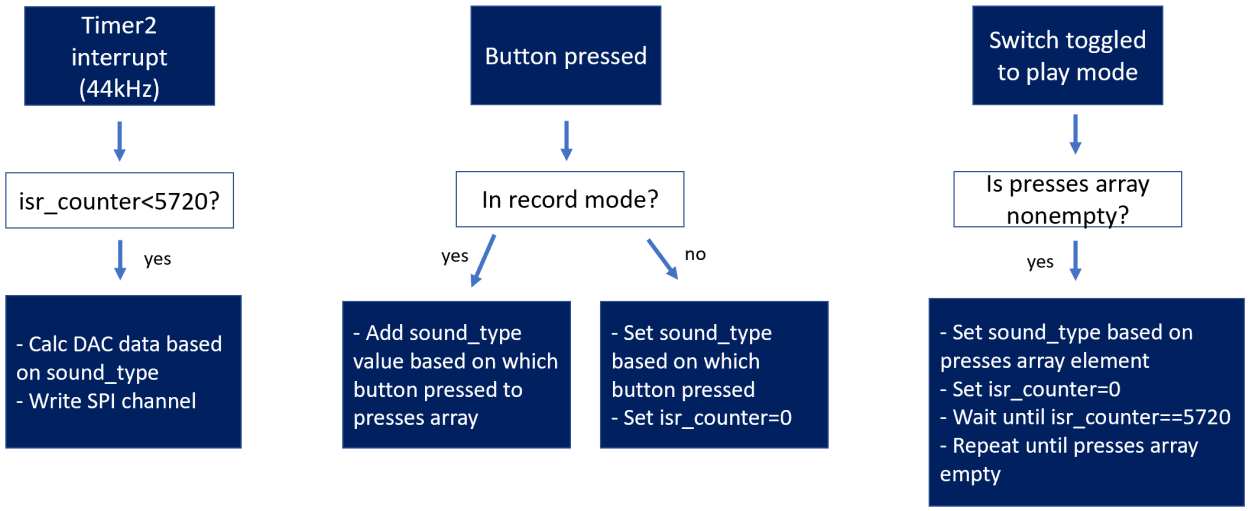

Figure 2 - Diagram illustrating the functionality created by the threads and ISR.

Testing

Because this lab centered around an audio sound being played, we were lucky to test most of the program by ear. With the remote desktop computer joined into our Zoom call, we could hear the audio signals we sent to the DAC be output. When we first implemented the swoop button, we were able to hear it on the first try. The audio also helped us determine if our amplitude modulation was working correctly. We first played the code without the amplitude changes and listened intently to the swoop, and then immediately played it again with the modulation commented in. This made the difference between the two clear, as there was an audible clicking noise at the beginning of the non-amplitude modulated swoop that confirmed it was correct.

The oscilloscope was also instrumental to our testing process. It made it easy to press the buttons on the interface and see if any audio output was coming out at all from the PIC32. It was particularly helpful in verifying our amplitude modulation, as we were able to zoom out on our audio signal and see the nice amplitude modulated envelope ramping up and down. It also helped confirm that the sounds were of the appropriate length (in time units). This will be discussed more in the results section.

The remote desktop through which we were programming the PIC32 had a camera pointed at the board such that we could see output changes. We tested the python interface connection using the demo Lab0 code functionality. For example, to test the creation of one of our buttons labeled “swoop”, we bound it to the LED turn on code that was provided to us. This helped us ensure that serial thread and button threads were working properly. Many times throughout the lab we then used the python interface’s input ability to type “h” into the serial bar and send it to the PIC32. The program had code to see the received “h” and print a help menu to

the interface, and this not working at some points helped us discover faulty physical hardware connections between the board.

Printing our own outputs to the python interface was also very helpful in debugging. For example, while implementing the switch from play mode to record mode and playing what was stored in the presses array, we heard through zoom that the sequence of swoop-chirp-pause seemed to have each sound played twice to the tune of swoop-swoop-chirp-chirp-pause-pause. By using printf() to print to the python interface the contents of the presses array, we saw that each time a button was pressed, its sound type was added twice to the array. This signaled to us that the button thread was adding the sound to the array for both the press and the release of the button.

We also tested the functionality of our project by playing different combinations of sounds, having short and long button presses/times between button presses, and repeatedly switching between modes. Doing so helped ensure that the software was robust and continued functioning correctly in all usage patterns.

Hardware Description

Below is a chart of main hardware components and peripherals used by our program.

PIC32MX250F128B Microcontroller	Microcontroller onto which our program is flashed. Also in charge of interfacing with our peripheral I/O devices (the DAC, the TFT display) with relevant communication protocols.
SECABB Big board	Big board breakout that houses the peripheral hardware interfaced with by the MCU: port expander for serial communication, DAC, header socket for the TFT, header for programmer, power supply.
Digital-To-Analog Converter	The DAC takes an entry from the sine table through SPI from the MCU and converts it into the analog output which is constructed with other SPI inputs into our birdcall output signal.
PicKit3 Programmer	Programmer that allows us to develop on the PIC32, interfaces with a 6-pin ICSP header on the BigBoard.
Python Interface UART	The UART connection from the MCU was used to send and receive messages from the python GUI, for debugging purposes and also receiving button commands.

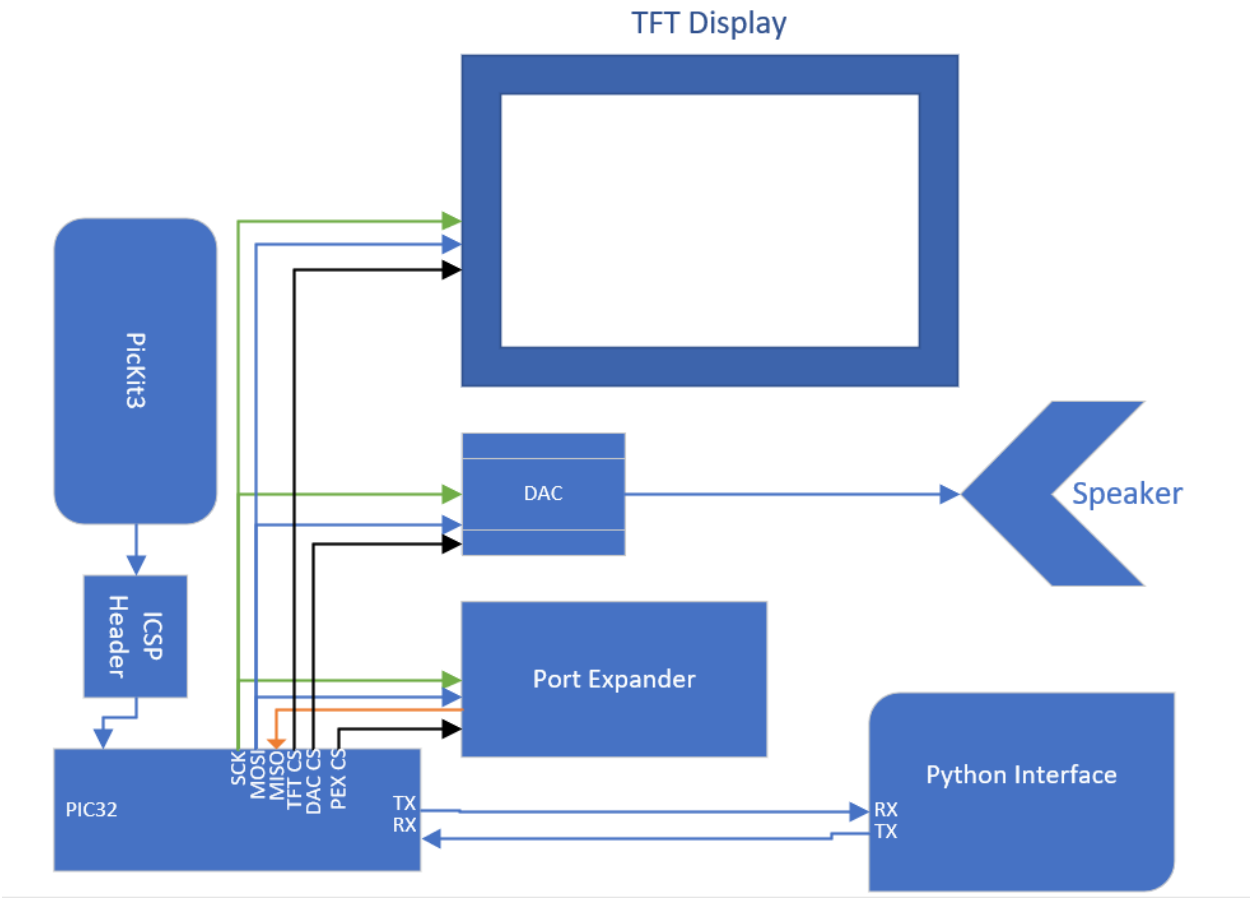

Figure 3: Circuit Layout of Prominent Hardware Components Within the Big Board

Software Description

Below is a chart of the main functions, threads, variables, and ISR in our program.

unsigned int isr_counter	This counter is incremented every time the ISR is entered at a rate of 44kHz. It is reset to zero when a button is pressed or when playing back the sequence of button presses after record mode. Resetting to zero starts the 5720 samples being sent to the DAC to play a swoop, chirp, or pause.
unsigned int sound_type	This variable corresponds to what sound the user wants to play. It is set to 0 for a pause, 1 for a swoop, and 2 for a chirp.
dds_increment()	Function that returned the correct phase accumulator using floating point calculations based on the sound_type global variable (*see Concept section, Figure 1*). Called from ISR.
Timer2 ISR	Triggered at a rate of 44kHz, checks if isr_counter is between 0 and 5720 meaning that it is in the middle of playing a sound

	being played, then calls the dds_increment() function to get the proper phase accumulator. Looks up in the sin table and multiplies by the global variable amp, which is the amplitude ramping from 1 to 1000 then back down 1000 over the course of the 5720 samples of the sound. Clears port B to start the SPI transaction with the DAC, and writes the DAC data input. Waits for the SPI transaction to complete.
protothread_python_string	This thread was inherited from the Lab0 demo code. It waits for a new python input string is received from the user interface, and reads to see if the input is for example "h" meaning the program should print the help menu to the python interface. We kept this thread for testing purposes to ensure our python interface and serial channel were working.
protothread_buttons	This thread yields until there is a new button event from the interface meaning a button is pressed or released. If the program is in play mode, it checks the button id of the pressed button, and sets the sound_type to match which button the user pressed and wants to hear. It sets the isr_counter to zero to start the sound being played immediately. If the program is in play mode, it fills the array of button presses (presses_array) with the current sound_type of the button pressed, and increments the presses_index that keeps track of how filled the presses_array is.
protothread_toggles	This thread yields until there is a new toggle event. If the toggle value is 1 meaning the record mode toggle is checked, then it sets the global mode variable to be 1. If the toggle value is zero, then it sets the mode to be 0 meaning that the user unchecked and is ready for play mode. This changing of mode may trigger the protothread_playmode to run if the presses_array has been filled with at least one user button press.
protothread_playmode	This thread yields until record mode is exited and the array that stores the sequence of button presses has at least one entry. When these conditions are met, it reads the array that stores which sound type the user wanted (because this corresponds to which button they pressed on the interface) and sets the isr_counter to zero, so that when the timer ISR is entered it will start playing the sound. It waits for the sound to finish after 5720 samples before processing the next sound in the array. *See Figure 3 - commented code snippet below*
main	Sets up the timeout for the timer to be 44kHz and configures the interrupt to have priority two. Clears the interrupt flag. Performs setup for the DAC by setting the port B pin 4 as digital output. Opens the SPI channel 2 in 16 bit mode. Builds the sin

	table. Initializes threads and round robin scheduling.

```
static PT_THREAD(protothread_playmode(struct pt *pt)) {
    PT_BEGIN(pt);
    while (1) {
        //yield from the toggle thread sets mode to play mode,
        //meaning the user is finished recording,
        //and the press_index is nonnegative, meaning that
        //it was incremented by button presses and the presses_array
        //is filled with at least one button press sound type
        PT_YIELD_UNTIL(pt, (mode == 0 && press_index != -1));

        int i;
        //the press array is populated with 0,1, or 2 corresponding
        //to pause, swoop, and chirp sound type
        for (i = 0; i <= press_index; i++) {
            amp = 0;  //reset the amplitude modulation
            //set the global sound_type so that the ISR will produce
            //sound that was stored in array
            sound_type = press_array[i];
            //resetting the isr_counter starts the playing of the sound
            isr_counter = 0;
            //must wait for 5720 samples (5720 entries into the ISR)
            //for the sound to complete before processing next array entry
            while (isr_counter < 5720); //
        }
        press_index = -1; //reset the array index so it can be increased again
        //the next time user is in record mode
    }
    PT_END(pt);
}
```

Figure 3 - Playmode thread for generating the sequences stored during record mode.

```
if(isr_counter < 5720){ // if true, then play a sound
    isr_counter++;
    //  DDS phase and sine table lookup
    phase_accum_main += (int) dds_increment();

    if (isr_counter < 1000) { //linear ramp up
        amp += .001;
    }
    else if (isr_counter > 4720) { //linear ramp down
        amp -= 0.001;
    }
    // multiply by amp to create envelope
    DAC_data = amp*sin_table[phase_accum_main>>24][wave_type]  ;

    // === DAC Channel A =============
    // wait for possible port expander transactions to complete
    // CS low to start transaction
     mPORTBClearBits(BIT_4); // start transaction
    // write to spi2
    if (dds_state==1)
        WriteSPI2( DAC_config_chan_A | ((DAC_data + 2048) & 0xfff));
    else
        WriteSPI2( DAC_config_chan_A | ((int)(V_data) & 0xfff));
    while (SPI2STATbits.SPIBUSY) WAIT; // wait for end of transaction
     // CS high
    mPORTBSetBits(BIT_4) ;
```

Figure 4 - ISR code to send values to the DAC.

Results

This project performs correctly and as expected. Pressing buttons in play mode immediately plays the appropriate sounds, and record mode successfully stores button press sequences that get played back in play mode. The sounds all sound correct, and the user interface behaves how it is supposed to.

Output Analysis

This program's outputs are all in the form of audio, and verifying their correctness is largely a qualitative task. When listening to the real swoop-chirp-pause sequence and comparing it with the swoop-chirp-pause sequence generated by the PIC32, they sound quite similar. Because what ultimately matters for this project is what the user hears, this is an important way to judge the correctness of the audio output. While the audio lacks some fine details of the real bird sounds being synthesized, the primary frequencies are clear and accurate as desired.

The correctness of the audio outputs can be verified quantitatively as well. For example, here is a zoomed out oscilloscope reading for a swoop:

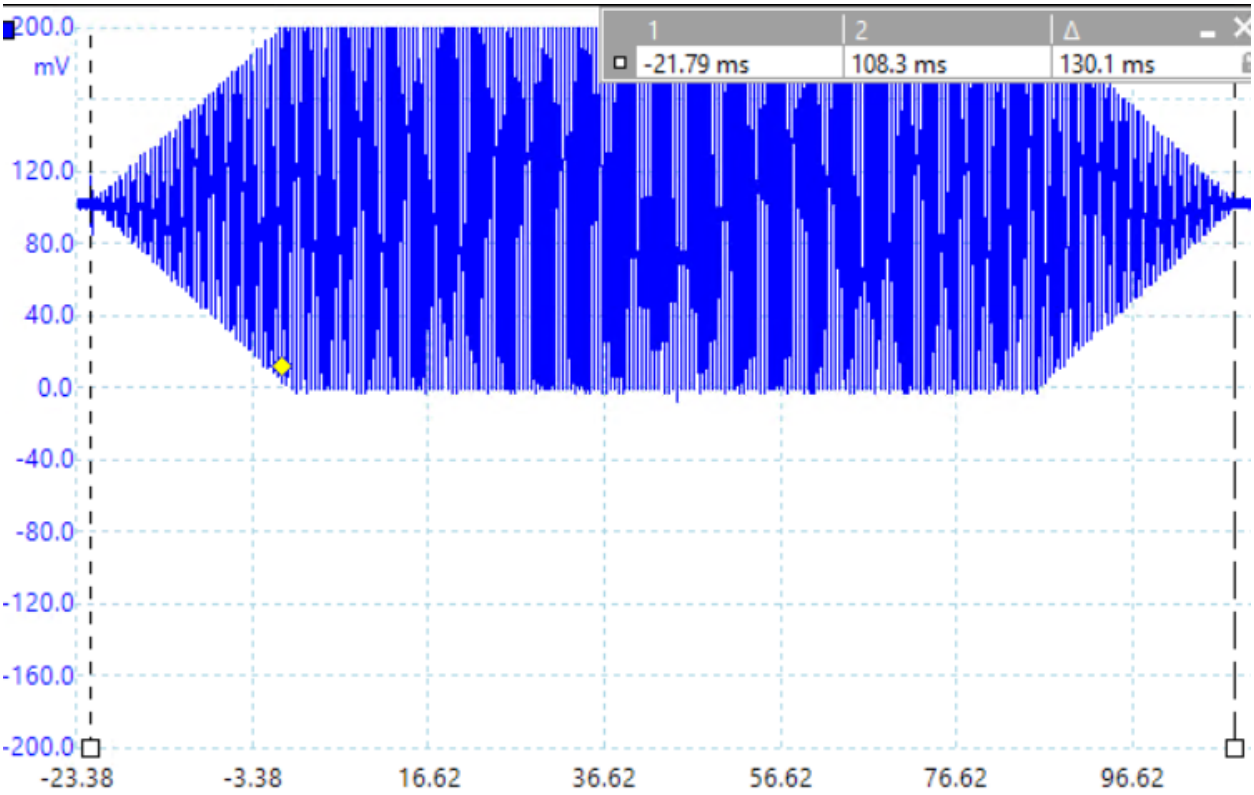

Figure 5- Oscilloscope measurement of a swoop.

There are several key things to note from *Figure 5*. First, the time measurement verifies that the swoop is 130 ms long. It likely is exactly 130 ms long, but knowing exactly where to place the measurement range markers is challenging. Furthermore, the amplitude modulation envelope is clearly shown. As specified in the lab handout, the envelope is linear with a rise, sustain, and fall. Measuring the length of the rise and fall reveals that they each last roughly 22.7 ms. Based on the 44k kHz sample rate, this means the rise and fall each last (22.7 ms)*(1 sec/1000 ms)*(44000 samples/sec) = 998.8 samples. Again, slightly inaccuracy with the measurement indicates that the rise and fall are likely exactly 1000 samples long as specified in the lab handout. This should be the case based on the amplitude modulation code.

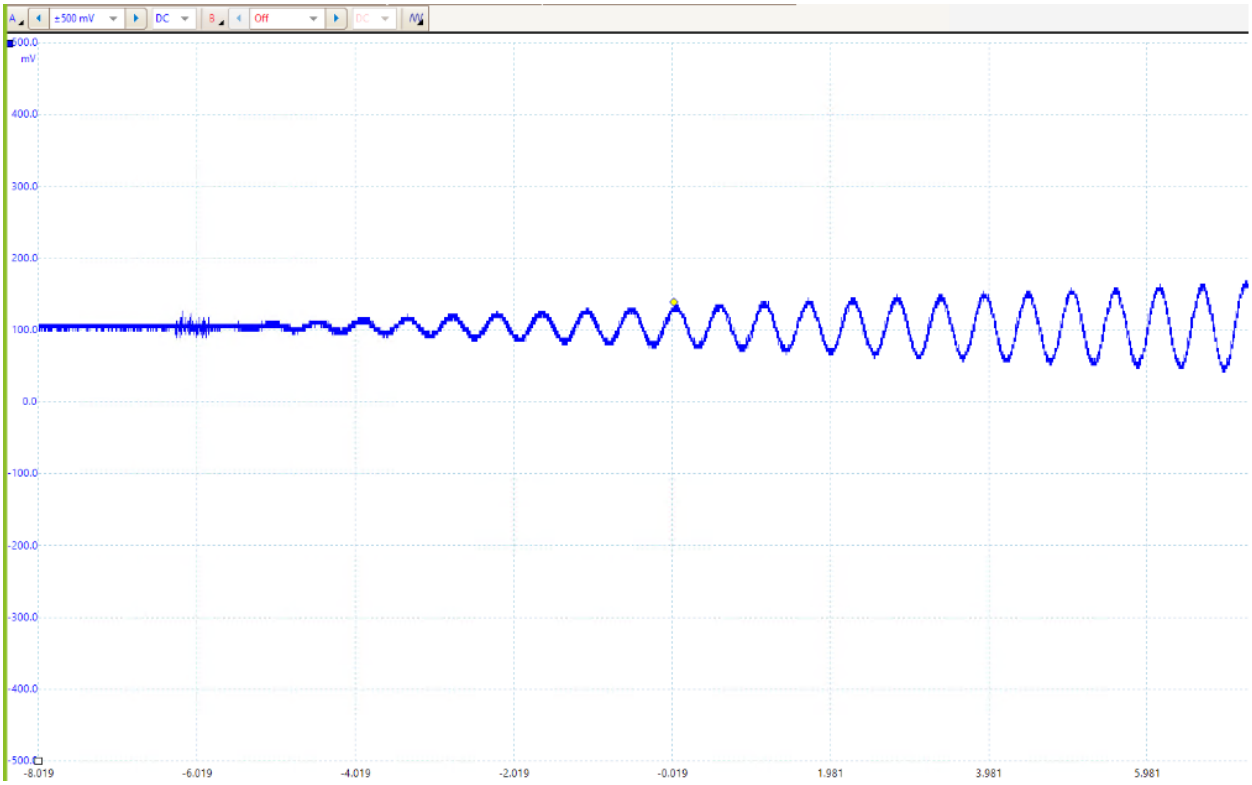

Figure 6- Oscilloscope measurement of the beginning of a chirp.

Figure 6 shows the beginning of the chirp amplitude envelope in more detail. It is clearly linear and does not interfere with the frequency content that gives the chirp (and swoop) their unique sounds.

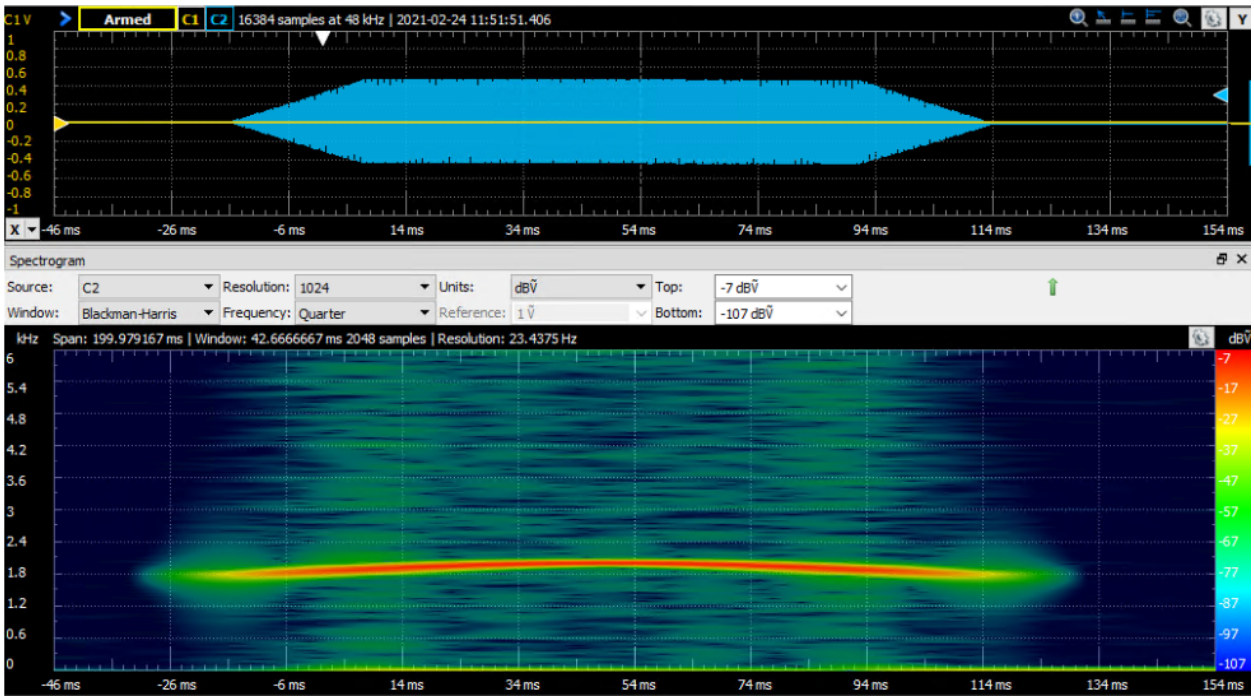

Figure 7- Oscilloscope measurement (top) and spectrogram (bottom) of a swoop.

The spectrogram in *Figure 7* shows how the swoop's frequency as a function of time approximates the first half of a sine wave. It starts and ends around 1.8 kHz and peaks at 2 kHz. Again, the graph's resolution makes exact measurement hard, but it is reasonable to assume the swoop had minimum and maximum values of 1.74 kHz and 2 kHz as specified in the handout.

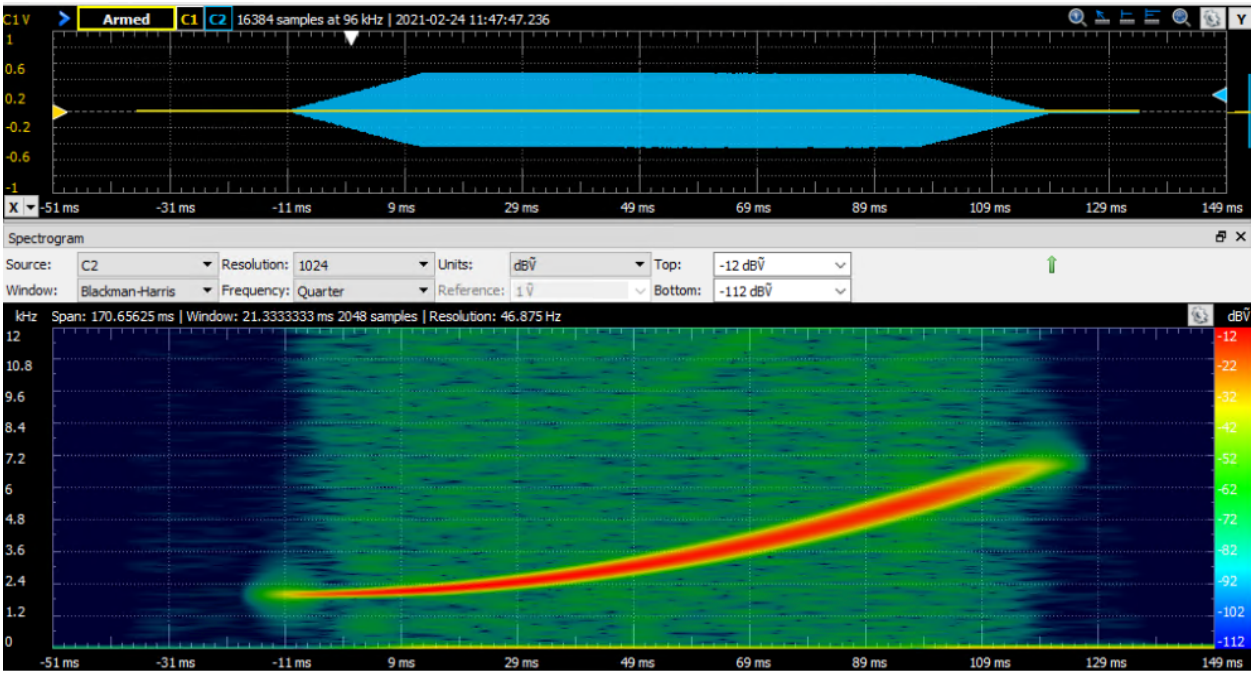

Figure 8- Oscilloscope measurement (top) and spectrogram (bottom) of a chirp.

The spectrogram of the chirp is shown in *Figure 8*. Its exponential shape also matches the desired output. Furthermore, the approximate starting and ending frequencies appear to be right at the desired values of 2 kHz and 7 kHz.

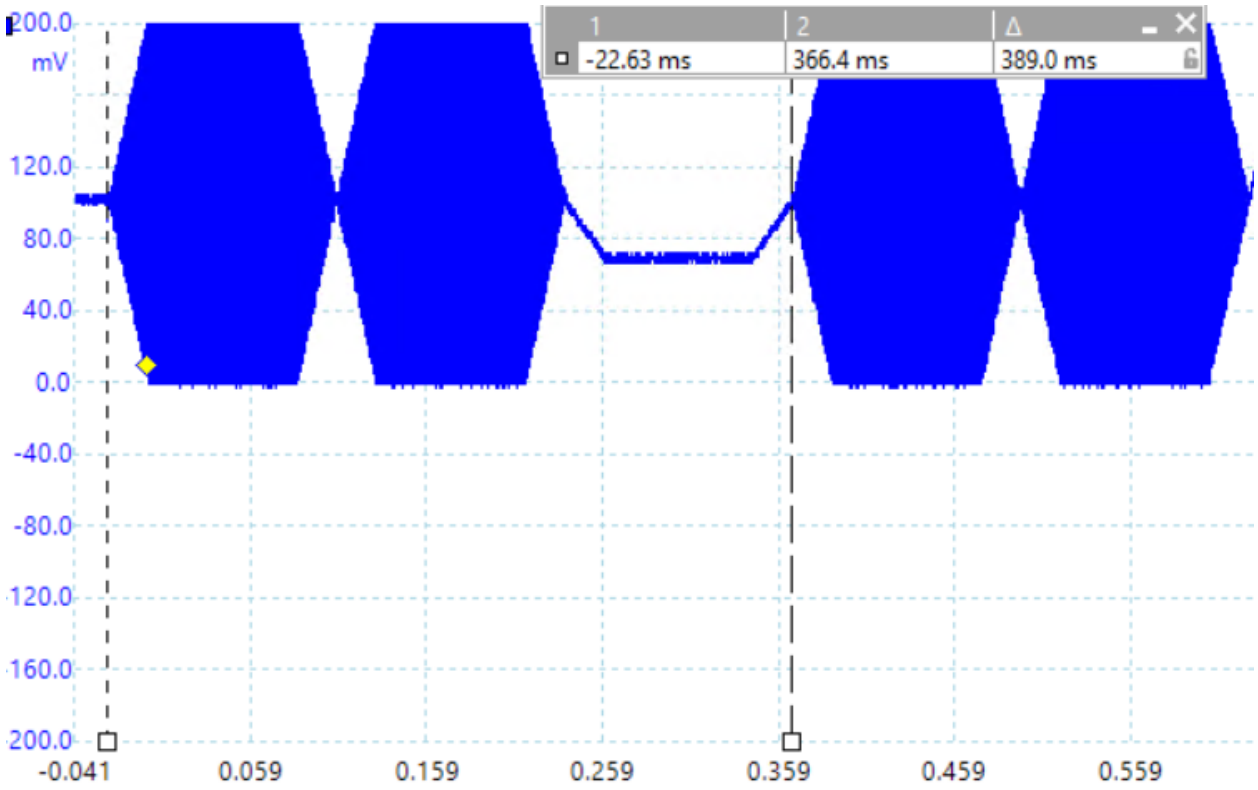

Figure 9- Oscilloscope measurement of a swoop-chirp-pause-swoop-chirp sequence.

Figure 9 displays a swoop-chirp-pause-swoop-chirp sequence that was created in record mode and played in play mode. The first swoop-chirp-pause lasts 389-390 ms, again demonstrating that each sound lasts 390/3 = 130 ms. The sounds play distinct from each other and do not overlap at all; this makes the audio clear and accurate. Furthermore, the amplitude of both the swoop and chirp is always 100 mV. This is desirable because the sounds should have the same volume, and one should not dominate the other.

One important note is that the pause, which should be silent, reads on the oscilloscope as being 100 mV - 60 mV = 40 mV from the equilibrium point. This occurs because the DDS algorithm outputs a small, constant phase value. Thus, a sound technically plays during the pause and may initially seem like an error. However, because the amplitude is so small, no sound is actually audible to the user during the pause. Therefore, modifying the code and adding complexity to it was deemed unnecessary, and this is not actually an error.

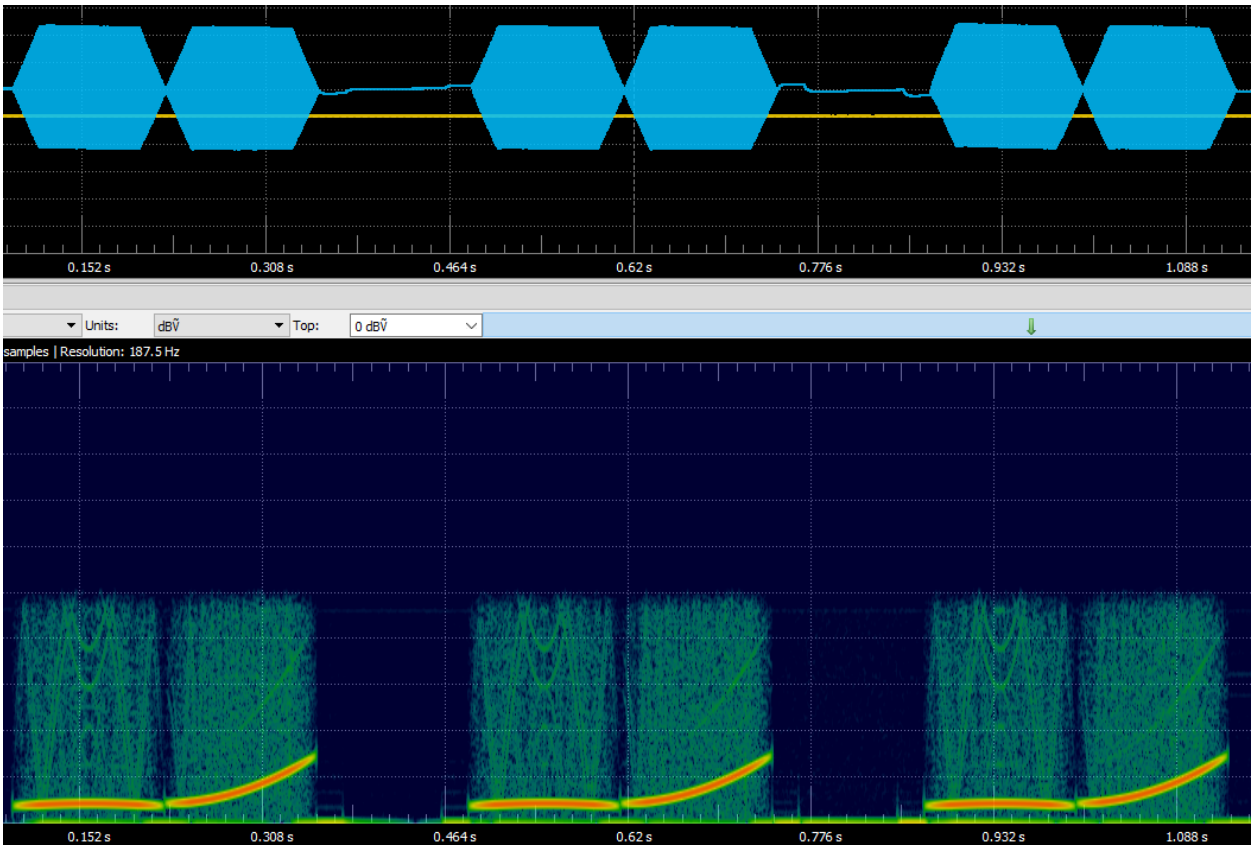

Figure 10- Oscilloscope measurement (top) and spectrogram (bottom) of three swoop-chirp-pause sequences.

The corresponding spectrogram for the swoop-chirp-pause in *Figure 10* matches the spectrogram provided to us in the lab handout. The sinusoidal swoop is followed by an exponential chirp and finally nothing during the pause. Note that the swoop is sinusoidal but appears a bit flat due to vertical scaling.

Code Speed Analysis

For our implementation to function correctly, it was crucial that all necessary computations had time to finish between timer interrupts. The timer interrupt occurs at 44 kHz, or every 1/44000 seconds. The CPU clock runs at 44 MHz. Thus, between each timer interrupt there are (1/44000 seconds)*(40000000 cycles/second) = 909 cycles. This means the ISR and threads have 909 cycles to prepare the next correct output value to be sent to the DAC.

In our implementation, all of the threads were not computation heavy. Most of the computation is due to performing the DDS algorithm. We did this in the ISR. As a result, the code functions correctly as long as the ISR requires well under 909 cycles.

In the worst case scenario, the ISR contains 4 floating point additions and 4 floating point multiplications. These floating point operations require the most cycles. According to the “Fixed Point Arithmetic” page on the ECE 4760 course website, these operations will require (4*60) +

(4*55) = 460 cycles. This leaves 909 - 460 = 449 cycles for the other ISR operations and threads to do their computations before the next timer interrupt. Qualitatively speaking, this is a substantial amount of cycles since the rest of the code does not do fixed point arithmetic. The analysis of the outputs above verifies that the code does indeed have time to finish; if it did not, then the output signals would be incorrect.

What Performed Well?

This project was fairly straightforward and did not leave much room to improve performance. With that said, our project performs well in the sense that everything works exactly as intended. The user experience is exactly as described in the lab handout; this includes the behavior of different modes, button functionality, etc. The sounds sound as desired and seem to exactly match the duration, frequencies, and amplitudes specified by the lab handout.

Another implementation decision that worked well was using floating point arithmetic. The concern with this was that it would not be fast enough. Alternatives such as using fixed point arithmetic or an Euler approximation would have been faster but less accurate. Fixed point, on the other hand, is very accurate. Furthermore, the code is simpler using floating point arithmetic since the provided algorithm could be directly translated into C code.

What Performed Poorly?

While the floating point arithmetic improved simplicity and accuracy, it did come at the cost of performance. Because all of the computations still met their deadlines, this did not matter. However, suppose new features were to be added to the project that require additional CPU cycles between interrupts. In this case, the floating point operations could be problematic and may need to be replaced with the faster, less accurate solutions.

What is Unique About our Implementation?

Our implementation is unique for two main reasons. First, we used floating point arithmetic. Again, this improved the simplicity of the code and accuracy of the audio. The drawback of additional CPU use was worth it to us since we did not plan on adding new features and thus did not need to preserve unused cycles.

Second, the playmode thread is unique to our implementation. Having a separate thread to play stored sound sequences kept the code organized and easy to understand, because the playmode thread's functionality is not very related to that of the other threads. Moreover, the playmode thread's functionality is only needed when there is a stored sequence of button presses after record mode. When this functionality is not needed, the thread can easily yield and not waste CPU cycles.

Conclusions

This lab was immensely useful in familiarizing ourselves with the lab equipment - which will stay the same for future labs - and also exploring the concepts of communication protocols and interrupt service routines. This lab taught us about machine timing and how we as programmers are confined within specific time intervals for operations and computations. Exploring ways to avoid lengthening the ISR exposed us to an aspect of low level programming we were not exposed to in more abstracted programming courses. We had to compartmentalize our floating point operations into specific threads while keeping track of the linear sequence of operations that needed to happen between interrupts in order for the frequency spectrum for the bird signal to be created in time for the next one.

This lab also taught us how to organize a project with multiple threads, and understanding the impact of context switching and dividing both responsibility and execution time among threads. Figuring out when to properly yield threads to let others continue running/incrementing and understanding the role of the interrupt service routine in the fabric of other threads was something new and exciting to grasp.

With this discussion of thread organization and interrupt timing, we learned about how our programs use communication protocols like UART and SPI to interface with peripherals. SPI was the method in which we communicated with the DAC, and this lab showed us how to set up an SPI data transaction coming from our PIC32 master with the DAC slave receiving the output. The UART lines were useful for debugging with the serial output on the python interface, which helped with our error in the duplication of stored signals in record mode (see **Testing** section).

Another issue we faced was that we originally received no sound output from the speaker. The culprit here was that we had not cast our sine table index to an integer from a float, so we were unable to select proper data to send to the DAC.

For future improvements on our design, I think we would incorporate fixed arithmetic with the _Accum data type that we learned about in lab 2, to possibly optimize our design in the event we want to add other complexity to the ISR and need leftover cycles to enable this implementation. For the lab itself, I think investigating bird calls of multiple species, or a conversation between two birds calling back and forth would be a unique extension of the potent framework we have.