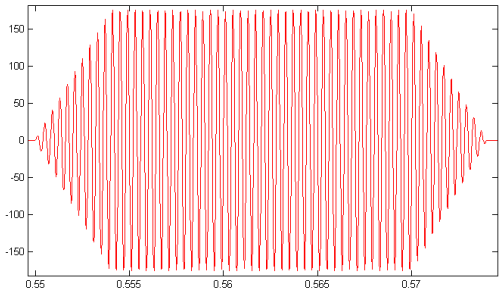
ECE 4760

Lab 2: Cricket-Call Generator

Author:	*NetID:*
Jonah Okike-Hephzibah	jo356
Sebastian Roubert	sr949

Section: Thursday 4:30pm
Performed on September 14th, 2017



Contents

List of Figures

1 Introduction

In this lab, we sought to produce a cricket-call generator. A cricket-call is actually composed of near perfect sine waves so it was necessary to generate analog sine waves where we could control the frequency. The generation process consisted of a user inputting 5 parameters (1. Chirp Repeat Interval, 2. Number of Syllables, 3. Syllable Duration, 4. Syllable Repeat Interval, 5. Burst Frequency). A call includes a syllable which is repeated for the number of times the user specifies. This block of syllables defines a chirp. After the specified number of syllables are sent out, a silent period will fill the remainder of the user specified chirp repeat interval. The range for chirp repeat interval, number of syllables, syllable duration, syllable repeat interval, and burst frequency were 10-1000, 1-100, 5-100, 10-100, and 1000-6000, respectively. The cricket-call would thus be made by indefinitely repeating this trill until the user decides to stop it. Users were able to input all of the necessary parameters using an external keypad and feedback from the TFT display. They then were able to decide to play a song, stop a song, or test the burst frequency. The frequency accuracy should be +/- 1%, time intervals and burst duration within +/- 1 msec, and harmonic distortion should be -20dB from fundamental tone. Nearly all specs were met by our project within an accuracy that sometimes our oscilloscope could not detect error, almost 0% error.

2 Design and Testing

2.1 Hardware

The hardware consisted of an external keypad as well as the PIC32 and the TFT display. The keypad that we were assigned consisted of 8 pin connections of which pins 1-4 (rows of the keypad) were connected using 300 ohm resistors to the A0-A3 pins and pins 5-7 (columns of keypad) were connected to B7-B9 on the pic. The keypad, as shown in Figure 1, works by connecting a row with a column via a user input on the keypad resulting in a complete circuit. The 300 ohm resistors were used to prevent the outputs from short circuiting if multiple buttons were pressed. The 10 KΩ resistors on the B7-B9 pins (in our system, these are replaced with internal pulldowns) provided an input source when a button was pressed. This, coupled with the debouncer outlined in the software section, allowed for the user to input the required parameters into the GUI.

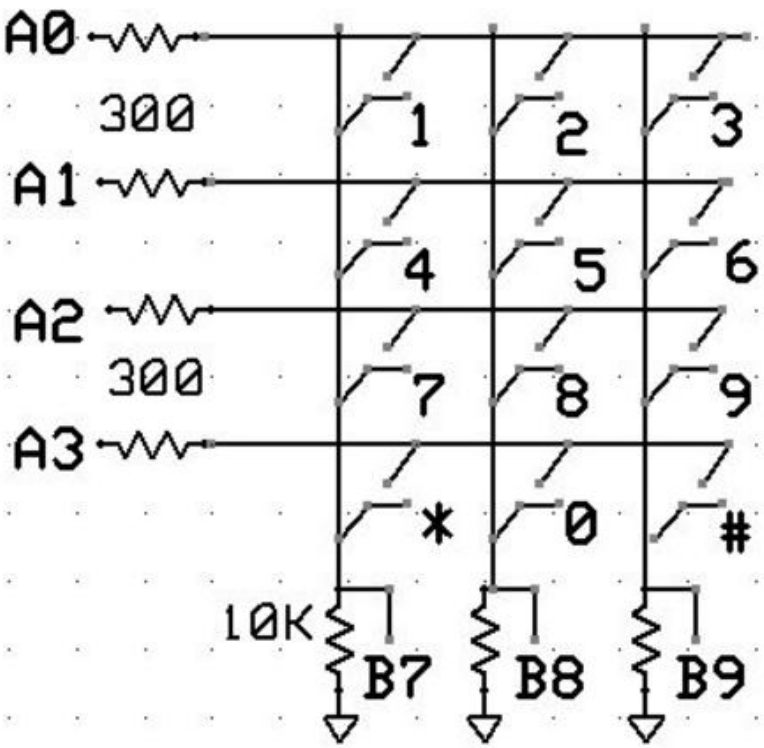


Figure 1: Keypad Circuit

The DAC (Digital to Analog Converter) was used to output signals sent from the SPI (Serial Peripheral Interface) channel 2. The SPI was setup so that it would output data on a transition from an active clock to idle clock in a 16 bit mode. The first 4 bits of this input was masked to contain the address of the DAC. In particular, we made use of DACA (DAC Channel A) whose output was sent through a lowpass filter.

The lowpass filter was used to compensate for our low sampling frequency choice. Since our sampling frequency of 100 KHz was not high enough to ensure that all spurious frequencies were 20dB below the main tone; we needed to filter these out. Using a resistance of 330 Ω and a capacitor of 20 nF, we were able to filter out a frequency of 24 KHz which is the average between our max (6 KHz) and first error frequency (42 KHz). We were able to determine this first error frequency by using the following relation:

$$\text{First\_error\_frequency} = \text{steps\_per\_cycle} * \text{frequency} - \text{frequency}$$

Choosing 8 steps per cycle and a frequency of 6 KHz (the max burst frequency), results in 42 KHz. However, since we cannot always guarantee that this first error frequency will happen at exactly 42 KHz, we took the average to ensure that we account for this fact.

2.2 Software

2.2.1 GUI

The GUI was run by updating the TFT via a protothread (dataAndTest) every 100 msec. Before the main portion of the protothread starts running, the TFT display is sent text that prompts the user to enter the required data. The prompts are as follows:

1. Chirp Interval:
2. Syllables:
3. Duration:
4. Syllable Interval:
5. Frequency:

Using the keypad, a user can enter the required parameters one at a time and advance to the next parameter by clicking the "#" key. If a user mistakenly clicks enters a value or wishes to retract their input, they can enter the "*" key which deletes the most recent entry each time it is pressed. This information, as seen in figure 2, is displayed to the user during this phase of the GUI. While the user is inputting their desired data the dataAndTest protothread is constantly updating the TFT with the most recent user entered value.

Figure 2: User Entry Display for the GUI

After the user has entered all of the required data, the place where the input information was displayed is replaced with the following text:

Press '1' to play.
Press '2' to pause.
Press '3' to test.
Press play when ready.

If the user decided to enter new data, they could enter the "*" key which would move back into the data entering portion of the GUI as seen in figure 3.

Figure 3: Play Options Display for the GUI

The end goal of the GUI was to make user input intuitive and simple while providing just enough information to clarify any of the less obvious features. We believe we were able to achieve this goal based on the ease of use we observed from the TAs' during the demo period.

2.2.2 Button Pushes

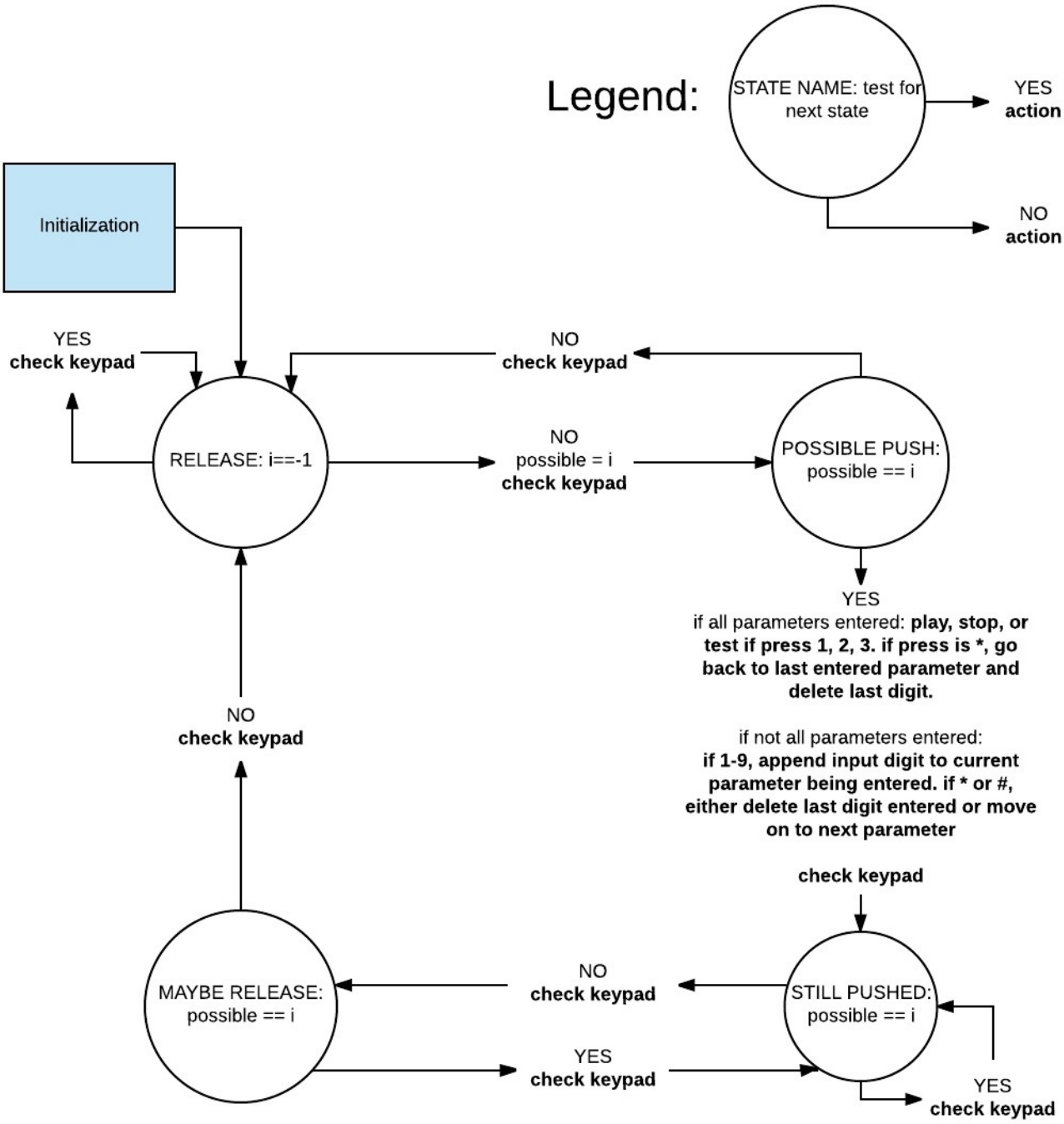


Figure 4: Finite State Machine of a button push

For the keypad code, the finite state machine seen in **Figure 4** was followed exactly. The implementation of a finite state machine was new to this lab. Because button press is an analog input, it comes with messy analog behavior. The button bounces. To account for this, the finite state machine exist in a protothread that was entered every 30msec. This was long enough to ensure that a button would stop bouncing by then unless a user was truly pressing yet short enough that user wouldn't stop pressing within that time and get a false read.

The check keypad functionality was done by checking if a button in each row was pushed and in each row checking the column. This was all done in a for loop where each iteration checked a row.

The rows in the keypad were set to PORTA of the PIC32. Bits 0-3 corresponded to the 1st-4th row of the keypad. The columns were set to PORTB. Bits 7-9 corresponded to the 1st-3rd row of the keypad. The check the row, you set the corresponding bit to high. This was done with the following:

```
mPORTAClearBits(BIT_0 | BIT_1 | BIT_2 | BIT_3);
pattern = 1; mPORTASetBits(pattern);
```

You then check if any of the bits in any of the columns is set to high:

```
keypad = mPORTBReadBits(BIT_7 | BIT_8 | BIT_9);
```

If they are high, you combine the keypad bits with the pattern bits and exit the for loop as seen here:

```
if (keypad !=0) {keypad |= pattern ; break;}
```

If not, high, you continue on to the next row. We created a 1-D array that corresponded to each possible button push. The index of the pattern in the array corresponded to the button. The table is as shown:

```
static int keytable[12]={0x108, 0x81, 0x101, 0x201, 0x82, 0x102, 0x202, 0x84, 0x104,
0x204, 0x88, 0x208};
```

We then iterated through the table in a for-loop where if the keypad corresponded to a value in the keytable, the for-loop ended and for-loop counter corresponded to the button push. If not, then it returned -1 which corresponded to no button pushed. See below:

```
        if (keypad > 0){ // then button is pushed

            for (i=0; i<12; i++){
                if (keytable[i]==keypad) break;
            }

        } else { i = -1;} // no button pushed
```

Once the keypad was read, the finite state machine was entered. The finite state machine was implemented using a switch statement. This allowed for ease between switching between states and allowed accurate expression of the state machine as seen in **Figure 4** . The switching between states was based on the variable i which was based on our keypad entry. An example can be see below which highlights the declaration/syntax of a switch and the decision of which state to move to next can be seen here:

```
switch( state ) {

    case 0: //release

        //if possible valid entry, store entry and go
        //to possible push state, else stay in release state
        if (i== -1) state =0;
        else { possible = i;
        state = 1;}
        break ;
```

Cases 0, 1, 2, and 3 corresponded to release, possible push, still pushed, and maybe release, respectively.

Cases 0, 2, and 3, are rather straightforward in their implementation. Case 1 was the most rigorous due to the fact that the transition between possible push and still pushed is what actually signified a push event. Upon this push event, a multitude of events could happen that all depended on the state of user entry. Refer to the GUI behavior section, **Figure 4**, and code appendix for the possible events upon a button press.

This finite state machine with conditional outputs based on user entry allowed us extreme flexibility in the making of the GUI and sound logic to the building of the rest of the project.

2.2.3 Sound Generation

The goal of this project in the end was to generate a sound from software to a speaker. Naturally, this exhibits Digital to Analog (DAC) behavior. We utilized DAC channel A as seen in hardware section. We then wrote to DAC using the Serial Peripheral Interface (SPI) functionality. In SPI, there's a slave to which the master writes and/or reads data. In our case, the master is the microcontroller; the DAC is the slave. Here, the We set the Bit 4 of Port B to be our chip select (CS) signal for the DAC. Basically, this signal is set an I/O pin on the board. When the pin is set to digital low, the SPI starts a transaction. When the pin is set to digital high, the SPI ends that transaction. See below:

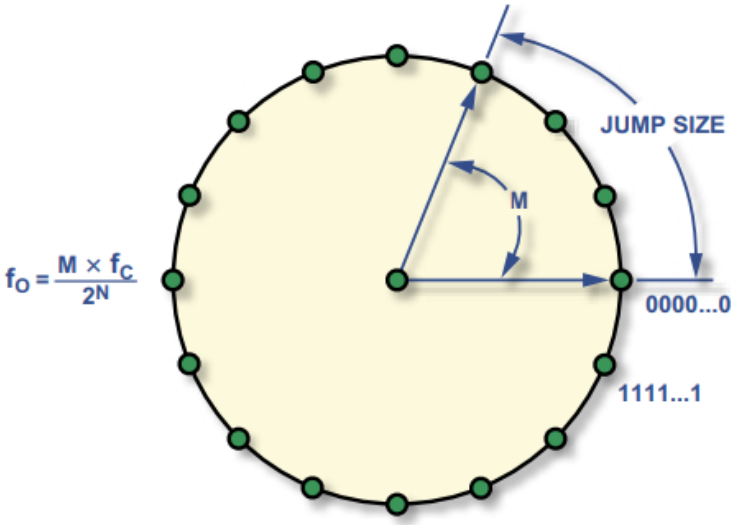
```
// === Channel A =============
    // CS low to start transaction
     mPORTBClearBits(BIT_4); // start transaction
```

```
// test for ready
// write to spi2
WriteSPI2( DAC_config_chan_A | (DAC_data + 2048));
while (SPI2STATbits.SPIBUSY); // wait for end of transaction
 // CS high
mPORTBSetBits(BIT_4); // end transaction
```

The reason we add 2048 to our DAC_data is because we did all our calculations based on real sine waves, which can have negative values. However, the DAC_data output can only be a real value between 0-4097.

The sound outputs of a cricket are basically sine waves with silent pauses parsed throughout. The first step of this is to actually generate a sine wave with any given frequency. We did this via direct digital synthesis (DDS). Direct digital synthesis produces an analog waveform via a digital signal followed digital to analog conversion.

The explaining of DDS can get a little abstract but is actually quite straightforward once you wrap your mind around it. Basically each value of a sine wave corresponds an angle on a circle. When it is time to output a value of the sine, you increment a variable called the phase accumulator. The phase accumulator is analogous to the angle on the circle. The phase accumulator has a maximum value and then resets, the same way angles can go up to 2π and then sine value resets. How fast you increment the phase accumulator, i.e. the sample frequency, multiplied by how much you increment the increment divided by the maximum value of your phase accumulator determines the output frequency of your sine wave. Refer to **Figure 5** for a pictorial representation.



n	NUMBER OF POINTS
8	256
12	4096
16	65535
20	1048576
24	16777216
28	268435456
32	4294967296
48	281474976710656

Figure 5: Pictorial representation of angle sweep for DDS output. M is the jump size, i.e. the phase increment. f_o is the output frequency, f_c is the sample frequency, and 2^n the maximum number of your phase accumulator. [1]

The main benefit of DDS is the robust frequency resolution with near exact approximations of values. It also allows for quick changes of the output frequency as you just need to change your phase increment.

Since crickets output near perfect sine waves, our design necessitated near perfection as well. For this, we chose our maximum number of our phase accumulator to be 232, 4294967296. This allows a high frequency resolution.

We chose our sample frequency to be 100kHz. By the Nyquist principle, we must choose our sample frequency based on at least twice the desired maximum output frequency, or 12 kHz. Because we must output a signal where the first spurious frequency must be lower than -20 db from main tone, we decided to have 8 samples per sine cycle. As such the sampling frequency should be at least 8*12kHz = 96kHz. We decided to round up to an even 100kHz. To achieve this sampling frequency we set an interrupt to go

off every 400 periods of the clock. The clock was set to 40MHz. Therefore the interrupt would go off at 40MHz/400 or 100kHz, our goal. We decided to use Timer3 and configured our interrupt service routine to that timer. See below:

```
OpenTimer3(T3_ON | T3_SOURCE_INT | T3_PS_1_1, 400);
```

To ensure the output frequency was within specification, we measured it using the oscilloscope.

The sine waves were generated via direct digital synthesis (DDS) of a premade sine table with 256 entries (8 bits).

The sine table was populated as follows:

```
for (i = 0; i < sine_table_size; i++){
          sin_table[i] = (int)(2047*sin((float)i*6.283/(float)sine_table_size));
    }
```

The reason why it was multiplied by 2047 is that the sine value is eventually output to the DAC which has digital inputs between 0 and 4097.

All the time based parameters input by the user were met by incrementing a counter variable every time the interrupt was fired.

We translated input parameters to the actual counter value of how many times the interrupt would have to be entered to reach that counter. For example, 1 ms would correspond to (100kHz * 1ms) = 100 counts. Therefore, 100 is the ratio between a 1ms and desired counts.

4ms Ramp Up/Down This was done by incrementing the running ISR counter until it reached a value of 4 ms * 100 counts/ms = 400 counts. We called this variable thresholdCount. While the counter was less than thresholdCount, the amplitude of the wave written to the DAC was modulated by the counter value divided by thresholdCounter. This resulted in a linear rise from 0 to 1 modulation over 400 counts. Once 400 counts had passed, the modulation was stopped until the user inputted syllable duration had been reached. At this point we began the ramp down which mirrors the steps taken in the ramp up.

Syllable Duration Using the calculated count proportion, we added 4ms to the inputted interval duration. This was done because we wanted the sine wave to be output for 4ms more than the input. This is due to the 2ms of ramp up and 2ms of ramp down that are not apart of the desired syllable yet still an output.

Syllable Repeat Interval Once the count has passed the syllable duration count, a zero signal was sent to the DAC until the count of the repeat interval had been reached. At this point, the counter was reset while the syllable number was incremented.

Chirp Repeat Interval To achieve the user inputted chirp repeat interval, we multiplied the interval duration by the counter proportion, 100. Every time the interrupt was fired and the counter was reset, we would check to see if the syllable number had surpassed the number of syllables specified by the user. Once this assessment was true, we sent a zero signal to the DAC until the count for the Chirp Repeat Interval had passed. At this point, we reset the count and number of syllables to zero which began the chirp process once again.

3 Documentation

static PT_THREAD (protothread_param(struct pt *pt))

Sole purpose of this thread is to check and update the output burst frequency. If the user input a burst frequency within the range, it updated. Once updated, the phase increment, i.e. the frequency about which sine wave is output, was updated. This phase increment was used in the interrupt for the direct digital synthesis.

static PT_THREAD (protothread_dataAndTest(struct pt *pt))

This protothread dealt with updating the GUI depending on what the user input. It also populates variables that check if the user inputs are valid. These variables are used for logic in the interrupt. It originally prompts the user to input data and outputs the instructions ('#' to finish entering a parameter, '*' to backspace). It then continuously updates the parameters displayed on the screen based on user input. If the user has finished entering all parameters, it changes the screen from instructions dealing with input to instructions dealing with playing the song, stopping the song, and testing the burst frequency.

static PT_THREAD (protothread_key(struct pt *pt))

This protothread dealt with the keypad. It implemented the state machine logic to read and store user button inputs. It checked the state of the keypad every 30ms since a bounce lasts far less than that time and a human is unlikely to stop pushing a button in that amount of time if they are pushing with purpose. It also counts the amount of parameters entered. This is especially useful in GUI updating in the dataAndTest protothread. It also checks, based on if all parameters are entered, if the user is commanding to play song, stop song, or test the burst frequency.

4 Results and Discussion

As shown in Figure 6, the low pass filter was successful in filtering out the previously mentioned spurious frequencies. The difference between the first harmonic and first error frequency is 57.2 dB which is almost 3 times the required for this assignment (20 dB).

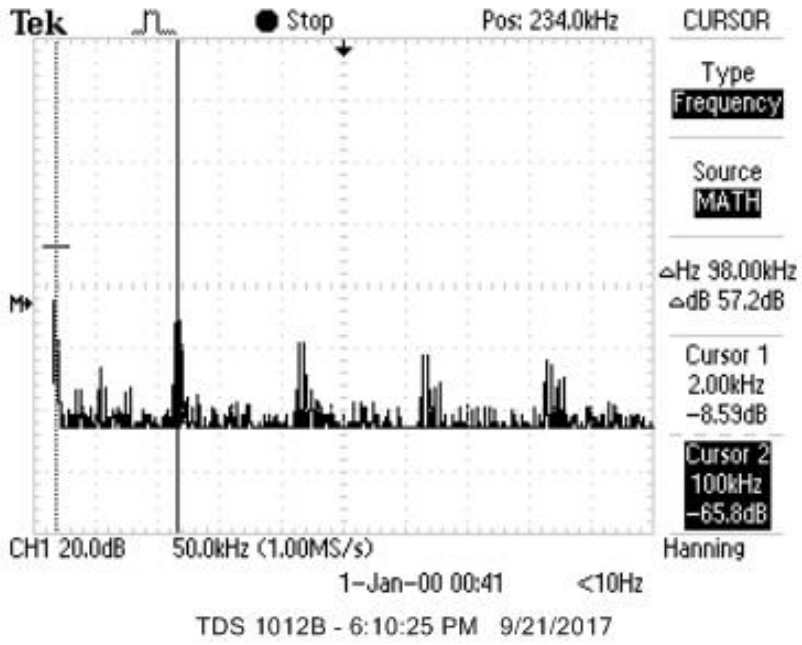


Figure 6: FFt Plot for 2500 Burst Frequency

In regards to the performance of the cricket-call generator, the system was able to meet all of the user specified inputs with little to no error. When given the following inputs:

Call 3:
Chirp Repeat Interval: 1000 ms
Number Of Syllables: 5
Syllable Duration: 30 ms
Syllable Repeat Interval: 50 ms
Burst Frequency: 3000 Hz.

The system was able to output a cricket call while maintaining all of these features as shown in figures 7-11.

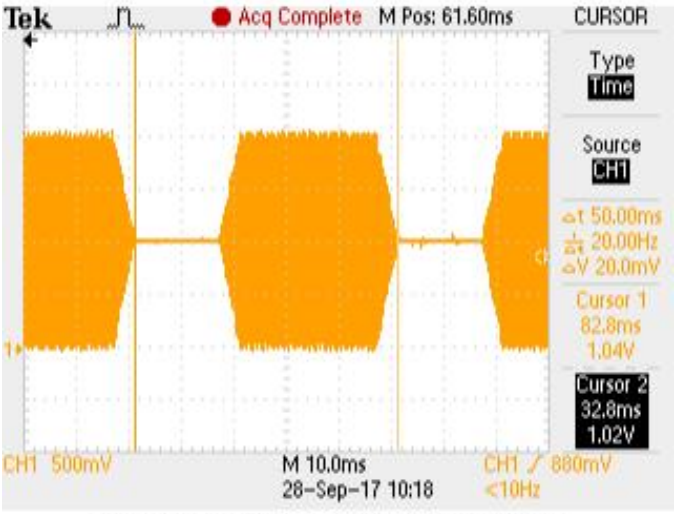


Figure 7: Syllable Repeat Interval For Call 3

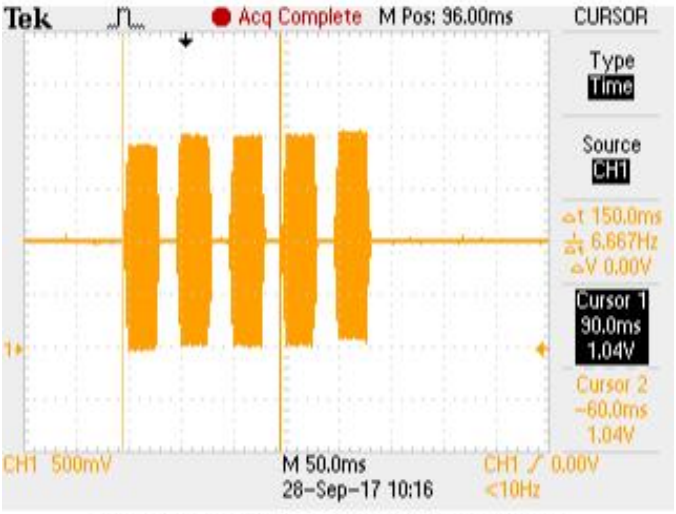


Figure 8: Syllables Output for Call 3

As seen in figure 8, the DAC outputted chirp contained 5 syllables as required by the input. On top of this, the syllable repeat interval exactly matches the specification of call 3 per figure 7.

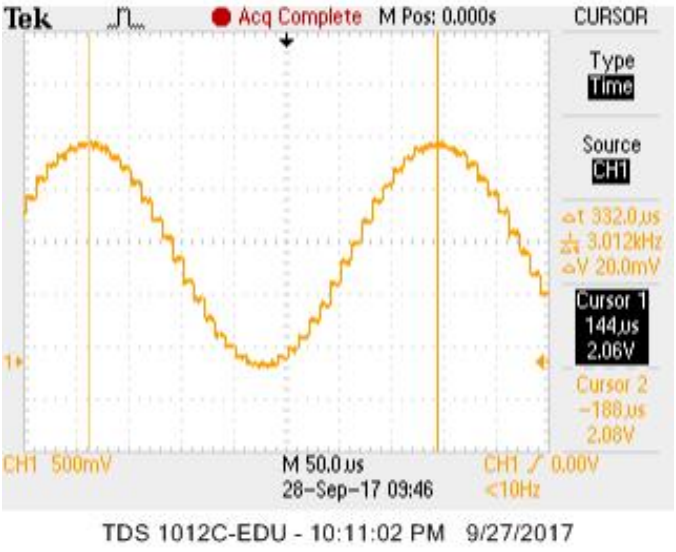


Figure 9: Sine Wave Frequency For Call 3

From the specification of call 3, the burst frequency had a required value of 3 KHz. As seen in figure 9, we were able to almost perfectly match this spec. However, this results in a 0.4% error which is within the maximum error of 1%.

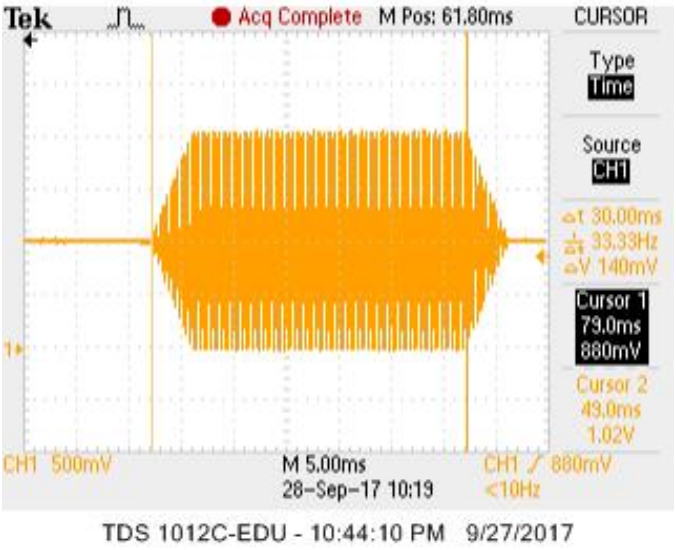


Figure 10: Duration For Call 3

In measuring the duration, it is important to remember that the duration is taken to be the width of the burst at half amplitude. Since the ramp up and down takes 4ms and is linear, we can assume that the burst half amplitude happens in 2ms. This adds up to a total of 4ms or rather, a full amplitude burst. This is why we were able to measure the syllable duration from the beginning of the ramp up to the beginning of the ramp down. Moving on to the duration, it is clear to see that we were able to match the 30 ms user specification perfectly.

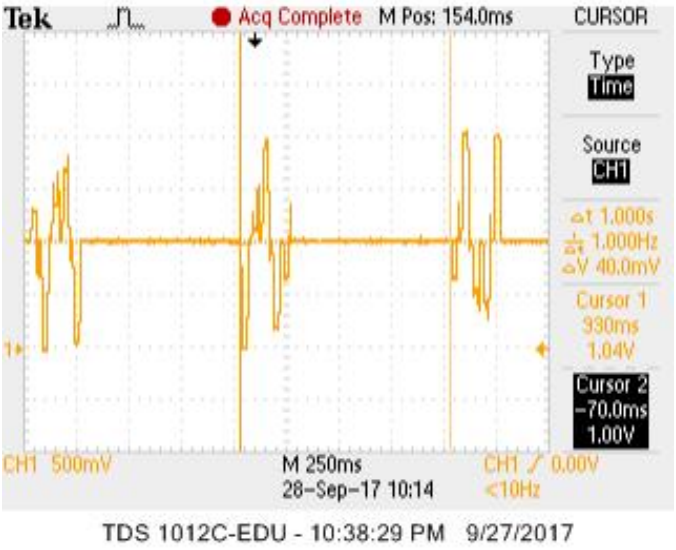


Figure 11: Chirp Repeat Interval For Call 3

As seen in figure 11, the chirp interval also perfectly aligned with the requirements outlined in the specification for call 3.

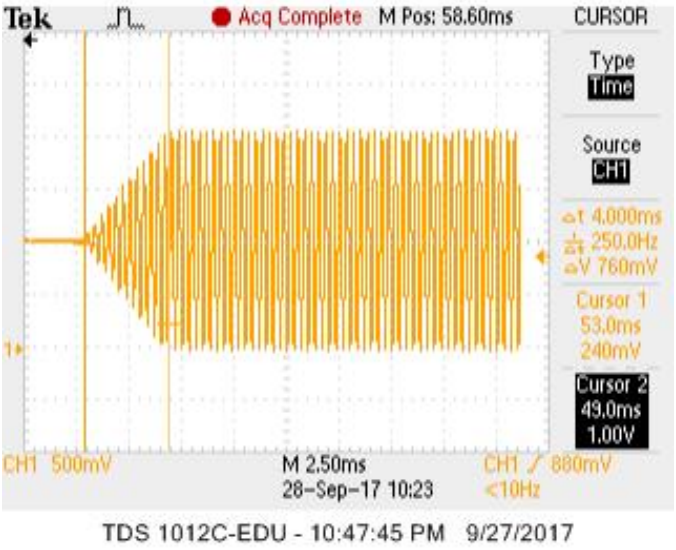


Figure 12: Ramp Up for Call 3

Finally, our analysis of the output for call 3 shows that we were also able to perfectly attain a ramp up in 4ms (refer to figure 12). These results solidify our proportion calculations for the counters we made use of in the interrupts. The system was almost perfect in every category which goes to show that the timers on the PIC32 actually performed as we expected (firing every 400 counts).

5 Conclusion

In this lab, we learned new skills associated with timing, master-slave protocols, abstraction with DDS and state machines, cleaning up analog inputs/outputs, and GUI design. DDS is approaching wide usage for its high frequency resolution, relative ease of use, and ability to change output frequency reliably and on the fly. We learned the SPI protocol which allowed us to write data to the DAC and eventually output our sine wave. We dealt with the messiness that is an analog input, the dreaded button press. The button press bounces and so we had to debounce. Then we dealt with a potentially messy analog outputs via a low-pass filter. We exercised robust state machine logic to read a true button push. It was a solution to a real world problem; people seldom think of the problems associated with a simple button push. We also had to think carefully about timing to accurately utilize our interrupt to give the desired output. The logic in our if-cascade had to be spot on which really exercised our computer engineering skills. This lab felt hefty but extremely rewarding. We ended up hearing crickets whenever we heard music for a couple

days.

There are couple of ways to make the lab more interesting. For instance, we could have multiple crickets singing. We could utilize DAC channel B We could use the same interrupt just change the output frequency for one of the crickets. We could also play around with our if-cascade logic for the multiple crickets so as to stagger their songs. Also the output does not have to be a crickets. It could very well be a crowd clapping to a beat. Some beats easily follow the same hierarchal structure that a cricket call does. The future ideas here are pretty large considering we're working with sound, a topic that fills many with passion.

The difficulties faced were numerous. The amount of obscure behavior this lab produced was incredible. For example, if you used any kind of division or worked with floats at all in your interrupt, the frequency would change subtly depending on which stage of your song you were in. Because it happened at different stages, it led to long sessions of debugging. Once, we divided by integers in our ramp up and down. Big mistake. Led to zero output. That took us a while to figure out. Your finite state machine logic had to be airtight. If not, obscure behavior would arise again. One time, our output to the screen was not the number we pushed but rather the row we were in.

Work Distribution:

Sebastian Primary on sound output generation (DDS, intervals, output frequency). Primary button push debounce finite state machine logic. Equal on debounce coding. Secondary on GUI.

Jonah Primary on GUI. Equal on debounce coding. Secondary on sound output generation.

6 References

[1] "All About Direct Digital Synthesis," Analog, Aug-2004. [Online]. Available: http://www.analog.com/media/e
dialogue/volume-38/number-3/articles/all-about-direct-digital-synthesis.pdf. [Accessed: 26-Sep-2017].

7 Appendix

7.1 C Code

```
/*
 * File:            lab2.c
 * Author:          Jonah Okike-Hepzibah & Sebastian Roubert
 * Adapted from:
```

```
 *                TFT_keypadBRL4.c & DDS demo code
                                  by
 * Author:        Bruce Land
 * Target PIC:    PIC32MX250F128B
 */

// graphics libraries
#include "config.h"
#include "tft_master.h"
#include "tft_gfx.h"
#include <stdlib.h>

// threading library
#include <plib.h>
// config.h sets 40 MHz
#define SYS_FREQ 40000000
#include "pt_cornell_1_2_1.h"
// for sine
#include <math.h>

 /*****************************************************************
The following code was developed by us to fulfill the requirements
set forth by Lab 2 of ECE 4760: Digital Systems Design using
Microcontrollers.

The goal was to generate cricket calls in speakers. The sounds generated
by a cricket are sine waves. Here we used linearly ramped sine waves
to make sure the energy output was smooth.

The user could input multiple parameters to characterize the cricket
calls:
1) chirp repeat interval (total time of chirp),
2) number of syllables (times where sound is continuously generated followed
by little moment of silence),
3) syllable duration (how long the sound generated portion of the syllable
lasts, not including the silence),
4) syllable repeat interval (how long the syllable totally lasts, including
the silence), and
```

```
5) burst frequency (the frequency of the sine wave generated)

The user inputs the parameters on a keypad with a debouncer that detects
legitimate button presses. The parameters are displayed on a TFT LCD. Once
all the parameters are entered, the '1' and '2' buttons play and stop
song production, respectively. The '3' button continuously plays the burst
frequency while held down. Pressing the '*' allows the user to change the
parameters entered.

The sine waves were generated via direct digital synthesis (DDS) of a premade
sine table with 256 entries (8 bits). The direct digital synthesis was
interrupt driven. The frequency generated was configured by changing the
phase increment of the DDS. The interrupt was configured to fire at 100kHz.
The execution of the given parameters, other than burst frequency, was
controlled via if-cascade in the interrupt, where a counter variable was
constantly incremented everytime the interrupt was entered.

The button debouncer was made via a finite state machine in a protothread
which entered the state machine every 30 msecs. Upon a legitimate
button press, the keypad updated. The '#' button was used to signal
the full entering of a parameter, the '*' button was used to erase
the last entered number.



 ************************************************************/

volatile SpiChannel spiChn = SPI_CHANNEL2 ;      // the SPI channel to use

// for 60 MHz PB clock use divide-by-3
volatile int spiClkDiv = 2 ; // 20 MHz DAC clock

// A-channel, 1x, active
#define DAC_config_chan_A 0b0011000000000000
#define DAC_config_chan_B 0b1011000000000000
#define Fs 100000.0         //sample frequency of our interrupt
#define two32 4294967296.0 // 2^32
```

```
// PORT B
#define EnablePullDownB(bits) CNPUBCLR=bits; CNPDBSET=bits;
#define DisablePullDownB(bits) CNPDBCLR=bits;
#define EnablePullUpB(bits) CNPDBCLR=bits; CNPUBSET=bits;
#define DisablePullUpB(bits) CNPUBCLR=bits;
//PORT A
#define EnablePullDownA(bits) CNPUACLR=bits; CNPDASET=bits;
#define DisablePullDownA(bits) CNPDACLR=bits;
#define EnablePullUpA(bits) CNPDACLR=bits; CNPUASET=bits;
#define DisablePullUpA(bits) CNPUACLR=bits;

// string buffer
char buffer[60];

//global variable used to store previous parameter input
static int prevValue = 0 ;

// === thread structures ============================================
// thread control structss
// one for the keypad state machine, one for updating the GUI, and one
//for updating the parameters in the chirp
static struct pt pt_key, pt_dataAndTest, pt_param;

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


static int storedData[5] ; //instantiation empty array for entered data
int pushState = 0 ;        //instantiate variable for state machine check
volatile int count=0;   //counter to use which parameter is currently being enter
volatile float amp = 0 ; //variable

//== Timer 2 interrupt handler ====================================
// actual scaled DAC
volatile int DAC_data;
// the DDS units:
volatile unsigned int phase_accum_main, phase_incr_main;//
```

```
// profiling of ISR
volatile int isr_time, tim3;

volatile int counter =0; //counter value incremented upon entering of
                         //interrupt. Used for most if-cascade logic.

volatile int maxCount = 0 ; //total count value of over which a syllable
                            //plus ramp up and down lasts

static int thresholdCount = 400; //count value where sine wave is ramping
                                 //to ensure 4ms

volatile int maxRepeatSylCount = 0 ; //used to store count value of
                                     //syllable repeat interval

volatile int maxChirpRepIntervalCount = 0 ;//used to store count value
                                           //of chirp repeat interval
static int totalSyl = 0;//used to store total # of syllables

static int sylCounted = 0;//used to store # of syllables output

volatile int play = 0; //boolean variable used to signify
unsigned int status ;
static int test = 0 ;

//================================
void __ISR(_TIMER_3_VECTOR, ipl2) Timer3Handler(void)
{
    // 74 cycles to get to this point from timer event
    mT3ClearIntFlag();

    // main DDS phase
    phase_accum_main += phase_incr_main  ;

        //updating variable values based on user keypad inputs
        //some are multiple by a constant to convert user input
        //time units to counts of the interrupt
    maxCount = (storedData[2]+4) * 100 ;
```

```
maxRepeatSylCount = (storedData[3]) * 100 ;
maxChirpRepIntervalCount = storedData[0] * 100;
totalSyl = storedData[1];

//checking to see if user said it's time to play and not test
//and that frequency input is valid
if ( play == 1 && test == 0 && storedData[4] != 0) {

    //ramp up stage
    if (counter < thresholdCount ) {
        amp = counter * 0.0025 * (sin_table[phase_accum_main>>24]) ;
        DAC_data = (int) amp ;
        counter++ ;

    //steady state output stage
    } else if (counter < (maxCount - thresholdCount)) {
        DAC_data = (sin_table[phase_accum_main>>24]) ;
        counter++ ;

    //ramp down
    } else if (counter <= maxCount ) {
        amp = (maxCount-counter)*0.0025*(sin_table[phase_accum_main>>24]) ;
        DAC_data = (int) amp ;
        counter++ ;

    //silent part of syllable interval
    } else if (counter < maxRepeatSylCount ) {
        DAC_data = 0 ;
        counter++;

    //interating syllable counter and resetting counter value to
    //start playing new syllable. If reached total syllables,
    //stop playing for this chirp
    } else {
        sylCounted++;
        counter = 0 ;
        if (sylCounted == totalSyl) play = 0;
```

```
        }

    //if user input testing burst freqency, output just the burst
    } else if (test == 1 && play == 0) {
        DAC_data = sin_table[phase_accum_main>>24] ;
    }

        //if not testing and valid frequency
    if (test == 0 && storedData[4] != 0) {

        //if still in silent part of chirp repeat interval, no output
        //but still increment
        if (play == 0 && sylCounted == totalSyl &&
            counter < (maxChirpRepIntervalCount- maxRepeatSylCount*totalSyl)){
            counter++;
            DAC_data = 0;

        //however if finished silent part of chirp interval, reset sylables, and
        //time to play more syllables
        } else if (sylCounted == totalSyl){
            sylCounted =0;
            counter = 0;
            play = 1;
        }
    }

    // === Channel A =============
    // CS low to start transaction
     mPORTBClearBits(BIT_4); // start transaction
    // test for ready
    // write to spi2
    WriteSPI2( DAC_config_chan_A | (DAC_data + 2048));
    while (SPI2STATbits.SPIBUSY); // wait for end of transaction
     // CS high
    mPORTBSetBits(BIT_4); // end transaction
} // end ISR TIMER3

// === set output frequency ===================================================
```

```
// thread configures the output frequency based on user input

static float Fout = 0; //output frequency
static PT_THREAD (protothread_param(struct pt *pt))
{
    PT_BEGIN(pt);
      while(1) {
            // yield 100msec
            PT_YIELD_TIME_msec(100) ;

            //check that input is within valid range
            if (storedData[4] > 1000 & storedData[4] < 6000){
                Fout = storedData[4];
            } else Fout = 0;

            //update the phase increment based on output frequency
            phase_incr_main = (int)(Fout*(float)two32/Fs);


            // NEVER exit while
      } // END WHILE(1)
  PT_END(pt);
} // thread 4

// === Data entry thread ======================================
// following thread updates the GUI based on user input and checks
//if entries are within valid ranges

static int sylEntryOk = 0, chirpEntryOk = 0, freqOk = 0;

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

    storedData[0] = 0 ; //chirp interval (ms)


    storedData[1] = 0 ; //syllables (# of syl)
```

```
storedData[2] = 0 ; //duration of syllable (ms)
storedData[3] = 0 ; //syllable interval (ms)
storedData[4] = 0 ; //frequency (Hz)

static int prevData1[5] ;

char buffer[128] ;

//GUI for entering data
tft_setCursor(0, 0);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(2);
tft_fillRoundRect(0,0, 200, 30, 2, ILI9340_BLACK);// x,y,w,h,radius,color
tft_writeString("Enter data for ") ;

tft_setCursor(0, 30);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_writeString("1. Chirp Interval: ");
tft_setCursor(0, 60);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_writeString("2. Syllables: ");
tft_setCursor(0, 90);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_writeString("3. Duration: ");
tft_setCursor(0, 120);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_writeString("4. Syllable Interval: ");
tft_setCursor(0, 150);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_writeString("5. Frequency: ");

tft_setCursor(0, 180);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_writeString("Press '*' to backspace. ");
tft_setCursor(0, 200);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_writeString("Press '#' to Enter. ");

while(1) {
```

```
//checks that entered parameters are valid to play
sylEntryOk = (int) (storedData[1]*storedData[3] < storedData[0] );
chirpEntryOk = (int) (storedData[2] < storedData[3] ) ;
freqOk = (int) ( storedData[4] >= 1000 && storedData[4] <= 6000 ) ;

// yield time
PT_YIELD_TIME_msec(100);

//updating TFT based on user input
tft_setCursor(0, 40);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_fillRoundRect(0,40, 100, 10, 1, ILI9340_BLACK);// x,y,w,h,radius,color
sprintf(buffer,"%d", storedData[0]);
tft_writeString(buffer) ;

tft_setCursor(0, 70);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_fillRoundRect(0,70, 100, 10, 1, ILI9340_BLACK);// x,y,w,h,radius,color
sprintf(buffer,"%d", storedData[1]);
tft_writeString(buffer) ;

tft_setCursor(0, 100);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_fillRoundRect(0,100, 100, 10, 1, ILI9340_BLACK);
sprintf(buffer,"%d", storedData[2]);
tft_writeString(buffer) ;

tft_setCursor(0, 130);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_fillRoundRect(0,130, 100, 10, 1, ILI9340_BLACK);
sprintf(buffer,"%d", storedData[3]);
tft_writeString(buffer) ;

tft_setCursor(0, 160);
tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
tft_fillRoundRect(0,160, 100, 10, 1, ILI9340_BLACK);
sprintf(buffer,"%d Hz", storedData[4]);
```

```
tft_writeString(buffer) ;

//updating GUI if user entered all 5 parameters
if (count >= 5) {

    tft_setCursor(0, 180);
    tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
    tft_fillRoundRect(0,180, 150, 10, 1, ILI9340_BLACK);
    tft_writeString("Press '1' to play.");
    tft_setCursor(0, 200);
    tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
    tft_fillRoundRect(0,200, 150, 10, 1, ILI9340_BLACK);
    tft_writeString("Press '2' to pause.");
    tft_setCursor(0, 220);
    tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
    tft_fillRoundRect(0,220, 150, 10, 1, ILI9340_BLACK);
    tft_writeString("Press '3' to test.");

    tft_setCursor(0, 0);
    tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
    tft_fillRoundRect(0,0, 200, 30, 2, ILI9340_BLACK);
    tft_writeString("Press play when ready.") ;
} else {
        //updating GUI if still parameters to enter
    if (prevValue != count+1) {
        tft_setCursor(0, 0);
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(2);
        tft_fillRoundRect(0,0, 200, 30, 2, ILI9340_BLACK)
        sprintf(buffer," %d.", count+1);
        tft_writeString(buffer) ;

        tft_fillRoundRect(0,200, 200, 30, 2, ILI9340_BLACK);
        tft_fillRoundRect(0,220, 200, 30, 2, ILI9340_BLACK);
        tft_fillRoundRect(0,180, 200, 30, 2, ILI9340_BLACK);

        tft_setCursor(0, 180);
        tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
        tft_writeString("Press '*' to backspace. ");
```

```
                tft_setCursor(0, 200);
                tft_setTextColor(ILI9340_YELLOW); tft_setTextSize(1);
                tft_writeString("Press '#' to Enter. ");

                prevValue = count+1 ;
            }
        }
    }
    PT_END(pt);
}

// === Keypad Thread =========================================================
// connections:
// A0 -- row 1 -- thru 300 ohm resistor
                -- avoid short when two buttons pushed
// A1 -- row 2 -- thru 300 ohm resistor
// A2 -- row 3 -- thru 300 ohm resistor
// A3 -- row 4 -- thru 300 ohm resistor
// B7 -- col 1 -- 10k pulldown resistor
                -- avoid open circuit input when no button pushed
// B8 -- col 2 -- 10k pulldown resistor
// B9 -- col 3 -- 10k pulldown resistor

//This thread updates the TFT LCD based on keypad button entries
//that are debounced via a finite state machine.

static PT_THREAD (protothread_key(struct pt *pt))
{
    PT_BEGIN(pt);
    static int keypad, i, pattern, press,j, keycode, possible;
    static int state = 0 ;
    count = 0 ;

    // order is 0 thru 9 then * ==10 and # ==11
    // no press = -1
    // table is decoded to natural digit order (except for * and #)
    // 0x80 for col 1 ; 0x100 for col 2 ; 0x200 for col 3
```

```
// 0x01 for row 1 ; 0x02 for row 2; etc
static int keytable[12]={0x108, 0x81, 0x101, 0x201, 0x82, 0x102, 0x202,
                                0x84, 0x104, 0x204, 0x88, 0x208};
// init the keypad pins A0-A3 and B7-B9
// PortA ports as digital outputs
mPORTASetPinsDigitalOut(BIT_0 | BIT_1 | BIT_2 | BIT_3);//Set port as output
// PortB as inputs
mPORTBSetPinsDigitalIn(BIT_7 | BIT_8 | BIT_9);//Set port as input

// No Push = 0
// Maybe Push = 1
// Pushed = 2
// Maybe No Push = 3

  while(1) {

    // read each row sequentially
    mPORTAClearBits(BIT_0 | BIT_1 | BIT_2 | BIT_3);
    pattern = 1; mPORTASetBits(pattern);

    // yield time
    PT_YIELD_TIME_msec(30);

    //checking through keypad to see which button is sensing push,
    //if it is
    for (i=0; i<4; i++){
        keypad = mPORTBReadBits(BIT_7 | BIT_8 | BIT_9);
        if (keypad !=0) {keypad |= pattern ; break;}
        mPORTAClearBits(pattern);
        pattern <<=1;
        mPORTASetBits(pattern);
    }

    if (keypad > 0){ // then button is pushed

        for (i=0; i<12; i++){
            if (keytable[i]==keypad) break;
        }
```

```
} else { i = -1;} // no button pushed

if (i==12) i = -1 ;//setting any other values to invalid

switch( state ) {

    case 0: //release

        //if possible valid entry, store entry and go
        //to possible push state
        if (i== -1) state =0;
        else { possible = i;
        state = 1;}
        break ;

    case 1: //possible

        //if possible valid entry is current keypad entry,
        //go to push state
        if (possible == i) {
            state = 2;

            //if not all parameters are entered
            if (count < 5) {

                //if user is inputting * at beginning of parameter entry
                //go to previous parameter entry line
                if ( storedData[count] == 0 && count != 0 && i == 10){
                    count -= 1 ;
                }
                //if user is inputting * in still valid parameter
                //entry, take off last digit
                else if (i == 10)
                { storedData[count] /= 10; }

                //add current entry to parameter
                else if (i != (10|11)){
```

```
            storedData[count] = storedData[count]*10 + i ;}

            //alter next parameter
            else if (i == 11) {
            count += 1 ;
            }
        } else {

            //This is the state when all parameters are entered.
            //if okay then play.
            if (i==1 && (sylEntryOk & chirpEntryOk & freqOk)) {
                play = 1 ;

            //pause
            } else if (i==2) {
                play = 0 ;

            //test
            } else if (i==3) {
                test = 1 ;
                play = 0 ;

            //go back to parameter entry
            } else if (i == 10) {
                count -= 1 ;
            }
        }

     //go back to release
    }else state=0;
    break;

case 2: // still pushed
    if (possible == i){
        state = 2;


       //go to maybe release
```

```
                } else state = 3;
                break;

            case 3: //maybe release
                if (possible ==i) state =2;
                else {
                    state = 0;
                    test = 0 ;
                }
        }
    }
        // END WHILE(1)
  PT_END(pt);
} // keypad thread


// === Main  ======================================================
void main(void) {
 SYSTEMConfigPerformance(PBCLK);

    /// timer interrupt //////////////////////////
    // Set up timer2 on,  interrupts, internal clock, prescalar 1, toggle rate
    // 400 is 100 ksamples/sec at 40 MHz clock
    // 200 is 200 ksamples/sec

    OpenTimer3(T3_ON | T3_SOURCE_INT | T3_PS_1_1, 400);

    // set up the timer interrupt with a priority of 2
    ConfigIntTimer3(T3_INT_ON | T3_INT_PRIOR_2);
    mT2ClearIntFlag(); // and clear the interrupt flag

    /// SPI setup /////////////////////////////////////////
    // SCK2 is pin 26
    // SDO2 is in PPS output group 2, could be connected to RB5 which is pin 14
    PPSOutput(2, RPB5, SDO2);
    // control CS for DAC
    mPORTBSetPinsDigitalOut(BIT_4);
    mPORTBSetBits(BIT_4);
```

```
  // divide Fpb by 2, configure the I/O ports. Not using SS in this example
  // 16 bit transfer CKP=1 CKE=1
  // possibles SPI_OPEN_CKP_HIGH;   SPI_OPEN_SMP_END;  SPI_OPEN_CKE_REV
  // For any given peripherial, you will need to match these
  SpiChnOpen(spiChn, SPI_OPEN_ON | SPI_OPEN_MODE16 | SPI_OPEN_MSTEN | SPI_OPEN_CK

  // the LED
  mPORTASetPinsDigitalOut(BIT_0);
  mPORTASetBits(BIT_0);

// === config threads ==========
// turns OFF UART support and debugger pin
PT_setup();

// === setup system wide interrupts  ========
INTEnableSystemMultiVectoredInt();

// init the threads
PT_INIT(&pt_key);
PT_INIT(&pt_dataAndTest);
PT_INIT(&pt_param);

// init the display
tft_init_hw();
tft_begin();
tft_fillScreen(ILI9340_BLACK);
//240x320 vertical display
tft_setRotation(0); // Use tft_setRotation(1) for 320x240

EnablePullDownB( BIT_7 | BIT_8 | BIT_9);

  // build the sine lookup table
 // scaled to produce values between 0 and 4096
 int i;
 for (i = 0; i < sine_table_size; i++){
       sin_table[i] = (int)(2047*sin((float)i*6.283/(float)sine_table_size));
  }
// schedule as fast as possible
```

```
  while (1){

      PT_SCHEDULE(protothread_param(&pt_param));
      PT_SCHEDULE(protothread_key(&pt_key));
      PT_SCHEDULE(protothread_dataAndTest(&pt_dataAndTest));
      }
  } // main

// === end  ============================================================
```