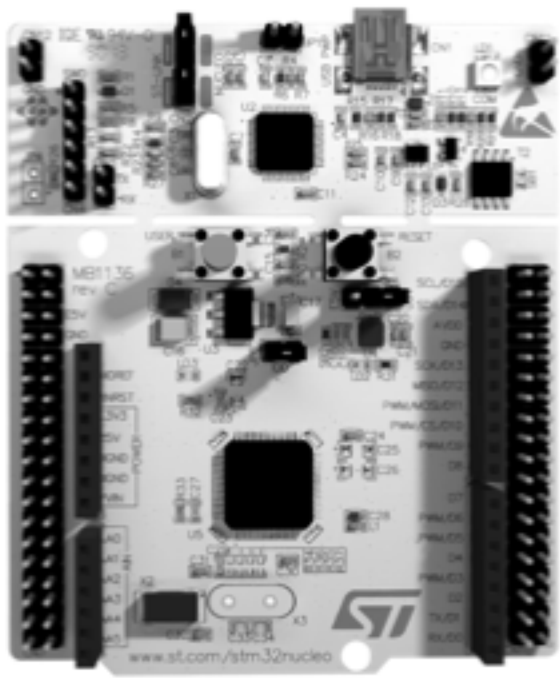
PROJECT #2 – FUNCTION GENERATOR

Cal Poly SLO | EE 329 - 01 | Professor Paul Hummel



Ethan Clark Najmy

Spring Quarter | May 9, 2022

[1] Photo taken from STM32 Lab Manual

1. Behavior Description

The function generator created in this project consists of a 4x4 keypad, a digital-to-analog converter (DAC) and an STM32L4A6ZGT6 board. The project functions as a variable function generator that can display 4 different types of waves at frequencies ranging between 100-500Hz. The 4 different types of waves consist of sine, sawtooth, triangle, and square, with square having an adjustable duty cycle ranging from 10% - 90%. All waves are $3V_{pp}$ with a 1.5V DC bias.

The function generator will power up into a square wave at 100Hz with a 50% duty cycle. While on the square wave setting, the user may press * to decrement the duty cycle, # to increment the duty cycle, and 0 to set it to 50%.

To switch between waves, the user may press 6 for a sine wave, 7 for a triangle wave, 8 for a sawtooth wave, and 9 for a square wave. The user may switch between frequencies on all waves by pressing 1 for 100Hz, 2 for 200Hz, 3 for 300Hz, 4 for 400Hz, and 5 for 500Hz.

2. System Specifications

Table 1. System Specifications

STM32L4A6ZGT6 Power Specifications	
Supply Voltage	-0.3V – 4V
Current Draw	150mA (max.)
Power Consumption	0.6W (max.)
Power Connection	Micro USB cable (not included)
Function Generator Specifications	
Wave Voltage (Peak-to-Peak)	3V
DC Bias	1.5V
Waveform Types	Sinusoid, Triangle, Sawtooth, Square
Frequency Range	100Hz – 500Hz (± 2.5Hz)
Keypad Specifications	
Total Keys	16
Usable Keys	12
Waveform Keys	• 6 – Sinusoid • 7 – Triangle • 8 – Sawtooth • 9 - Square
Frequency Keys	• 1 – 100Hz • 2 – 200Hz • 3- 300Hz • 4 – 400Hz • 5 – 500Hz
Duty Cycle Keys (Square Wave Only)	• * - decrement by 10% • # - increment by 10% • 0 – reset to 50%

3. System Schematic

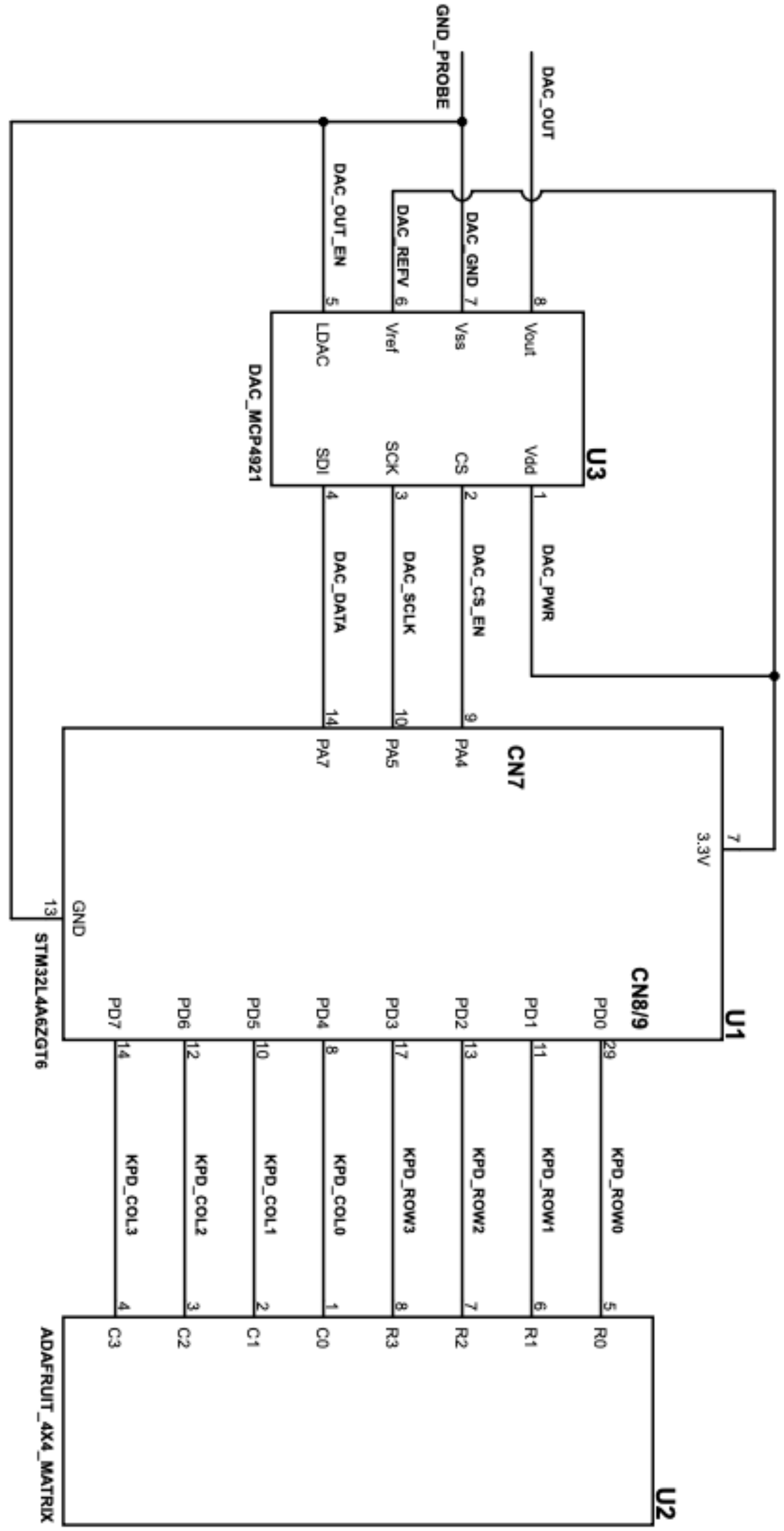


Figure 1. System Schematic

4. Software Architecture

4.1. Overview

The system functions through the use of a software-implemented Finite State Machine (FSM) and built-in timers and interrupts. When powered on, the system initializes the global variables and arrays, creates the FSM and states, creates lookup arrays for the waveforms, initializes the keypad and timers, then enters the FSM. The system will toggle between the FSM and interrupt handler while powered on and will switch between the 5 FSM states when commanded to. The key aspect of this system functioning is use of the built-in timers and interrupts and will be covered in a later section.

4.2. Initialization

The system initializes the global variables and the FSM when booted up. Global variables are used in this system to allow the interrupt handler function access to these necessary variables. The FSM is created through the use of the `typedef` keyword (source code can be found in *Appendix B*), creating a custom data type and creating the individual states. After initialization of the FSM, the system creates lookup arrays, or arrays with pre-defined voltage values for each wave. These lookup arrays are useful because instead of using time to generate each point while running, each point is generated once during startup. Next, the system initializes the keypad and DAC through the use of `keypad_init()` and `DAC_init()`. In each of these functions, the GPIO pins that each peripheral is connected to are initialized and configured as appropriate.

4.2.1. DAC Initialization

The DAC is initialized through the use of the `DAC_init()` function. In this initialization, the DAC is configured to interface with the STM board through the use of the Serial Peripheral Interface (SPI). The benefits of SPI are less wiring and low power usage. The DAC used for this system (MCP4921) is only usable with SPI.

Beginning the initialization, the proper GPIO clock and SPI clock are enabled and the GPIO pins are set to alternate function. Alternate function mode allows for SPI to be used on the STM board. The GPIO configuration is standard otherwise: push-pull output, low speed, no pull up or pull down resistors.

When configuring SPI for proper use, the system has specific requirements. These requirements consist of: SPI in simplex mode, MSB first, hardware chip select management, highest baud rate, low idle clock and polarity, MCU master, 16 bit communication, chip select pulse and enable modes.

All of these requirements are critical when initializing SPI. Fortunately, only a few of these need to be toggled, while a majority of them are already configured for our use.

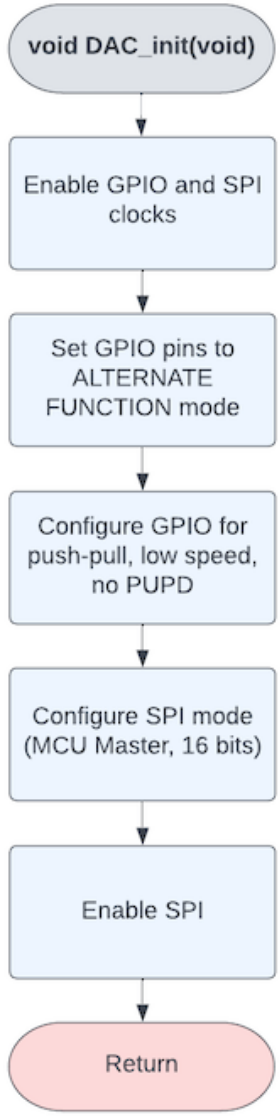


Figure 2. DAC Initialization Flowchart

4.2.2. Keypad Initialization

The keypad initialization is relatively simple when compared to the SPI. We begin by enabling the GPIO clock and configuring the columns as inputs and the rows as outputs. Each GPIO output will be configured as push-pull, low speed, and no pull up or pull down resistors. Lastly, the GPIO outputs are set high for keypad reading to occur.

** Keypad initialization flowchart omitted due to derivation by Professor Hummel*

4.3. Timers

This system uses timers and interrupts extensively and these are the backbone of proper operation and wave output.

4.3.1. Timing Overview

The CCR and ARR are used throughout this project to trigger interrupts and to call the interrupt handler function. When the CCR and ARR are triggered, the system will go into the IRQ function and check which waveform is currently active through the `waveform` variable. It will then go into the proper case statement that applies.

In the sine wave, triangle wave, and sawtooth wave case statements in the IRQ, it is a relatively similar process. In each case the system will clear the CCR interrupt flag and call `DAC_write(WAVE_array[count])` which will write to the DAC the voltage stored in the lookup table (mentioned in section 4.4) at index `count`.`WAVE` in this case is either `sine`, `tri`, or `saw`. It will then check to see if `count` >= the max array size and if so, it will reset `count` to zero, otherwise it will increment by `freq`. By incrementing by `freq`, this allows us to change our sampling rate without changing the CCR speed, allowing for easier and smoother operation of the STM board. The downside of this however is the reduced clarity and resolution on higher frequency waveforms.

NOTE: CCR timers are used only for sine, triangle, and sawtooth waves whereas CCR and ARR is used for the square waves. Doing this allows us to change duty cycle for the square wave and also keep the process of changing clocks smooth for the other waves.

4.3.2. Timing Calculations

Identifying when to trigger the CCR interrupts to plot the waves and the maximum resolution of the waves required some calculations.

Beginning our calculations, I measured how long at 40MHz my `DAC_write()` function would take to run by setting a GPIO pin high when entering the IRQ, calling `DAC_write()`, then setting the GPIO pin low and measuring the time between high and low. This time, we will call it Δt, came out to be 1.4μs. We can convert this to # of clock cycles by dividing it by 1/40MHz. This came out to equal 56 clock cycles.

I then added this number of clock cycles to the measured number of clock cycles from assignment 4 which was 40. This came out to be 96 total clock cycles. Converting these clock cycles back into time by multiplying by 0.25ns (1/40MHz) gave us a total ISR runtime of 2.4μs.

At 100Hz, we have a 10ms period. Taking this 10ms period and dividing it by the 2.4μs ISR runtime gave a total number of 4166 clock cycles that the ISR can run in 1 period. Because the DAC cannot output at this ideal rate, we will use a value around 60% of the ideal 4166 clock cycles that is divisible by 2, 3, 4, and 5 for ease of switching between frequencies. The final number of clock cycles in one period used in this project was 2640. This became the max array value used for storing points on a wave.

Our CCR value then used for the sine, triangle, and saw waves was equal to 60% of the # of clock cycles the ISR takes to run which came out to be around 152 clock cycles. This means that a point on a waveform will be printed every 152 clock cycles for a total of 2640 points per period.

Numerical Calculations

$$\text{DAC Write } \Delta\text{t} = 1.4\mu\text{s}$$

$$\text{DAC Write Clock Cycles} = \frac{1.4\mu\text{s}}{1/40\text{MHz}} = 56 \text{ clock cycles}$$

$$Total\ ISR\ Clock\ Cycles = 56 + 40 = 96\ clock\ cycles$$

$$Total\ ISR\ Runtime = 96 * \frac{1}{40MHz} = 2.4\mu s$$

$$Total\ \#\ of\ ISRs\ Per\ Period\ = \frac{10ms}{2.4\mu s} = 4166 * 60\% \approx\approx 2640\ clock\ cycles$$

$$Maximum\ Resolution = 96 + (96 * 60\%) \approx 152\ clock\ cycles$$

4.4. Arrays

This system relies on the use of lookup arrays to plot values for the sine, sawtooth, and triangle functions. The purpose of these arrays is to generate points for each graph during the initialization of the system in order to save time during plotting these points. Time is saved by reading from the arrays rather than using the equations for each waveform for each individual point.

The system works by creating an identical for loop for each function. Each for loop begins by creating a counting variable specific to the wave, initializing that variable at zero, and then counting up to the max array size calculated.

Inside each for loop the respective equation function (either `sineWave(count)`, `sawWave(count)`, `triWave(count)`) is called and the value returned is saved to the respective arrays, `sineArray[count]`,`sawArray[count]`,`triArray[count]`. When the system needs to output values for a wave, it will increment through these arrays and use these values.

`uint16_t sineWave(uint16_t sineInput)`

This function operates by taking an input, multiplying it by 2π and dividing it by the maximum array size. It will then take this value, input it into the C `sin` function, multiply it by 1475 and add 1475. The reason for multiplying by 1475 and adding 1475 is to provide a $3V_{pp}$ amplitude and 1.5V offset. The values are in the thousands due to how the `DAC_volt_conv()` function works. (Value was calibrated to be 1475 rather than 1500 exactly). The function is then converted to a voltage using the `DAC_volt_conv()` function and returned.

Full equation:

```
1475 * sin(((2 * PI * sinInput) / ARRAY_SIZE)) + 1475
```

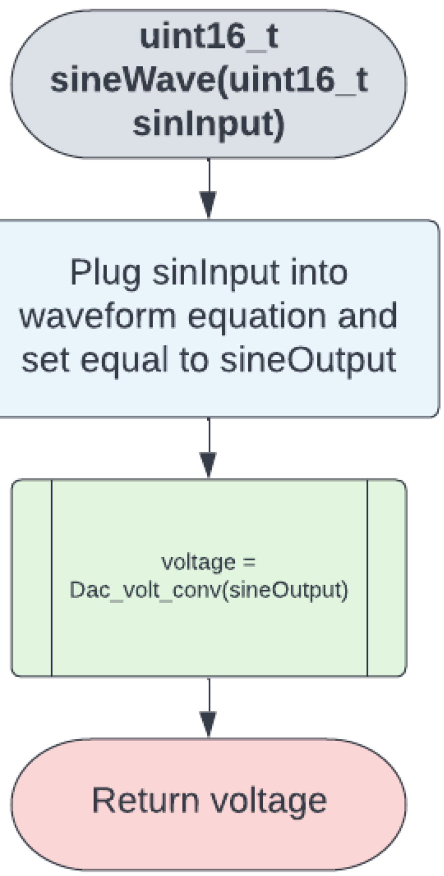


Figure 3. sineWave() Flowchart

`uint16_t sawWave(uint16_t sawInput)`

This function operates by taking an input, multiplying it by 1.137 (max array size ÷ 3000). It is divided by 3000 because the DAC reads 3000 as 3V. It will then input it to `DAC_volt_conv()` and return the value.

Full equation:

```
sawInput * 1.137666412;
```

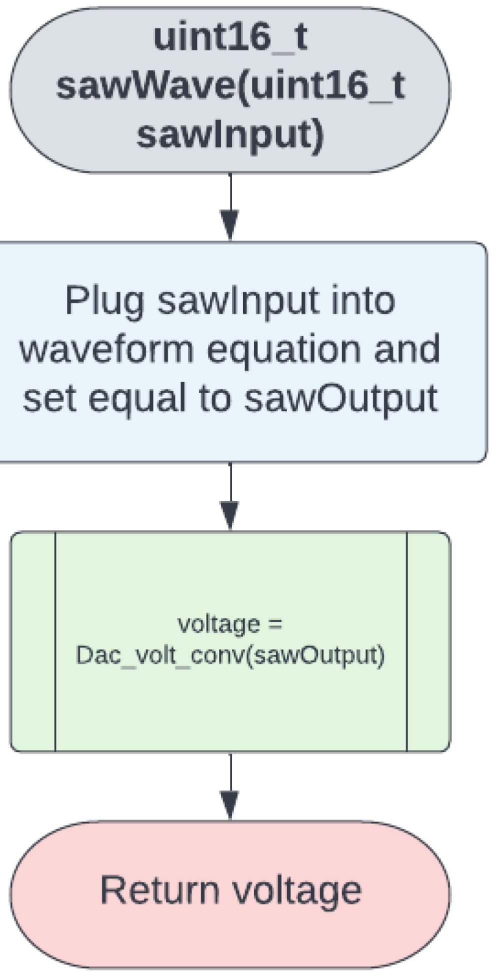


Figure 4. sawWave() Flowchart

`uint16_t triWave(uint16_t triInput)`

This function by taking an input, multiplying it by 1.137 (max array size ÷ 3000 - divided by 3000 because the DAC reads 3000 as 3V), and then multiplies it by 2. It will multiply by 2 because it is double the slope of a sawtooth function. If the value is greater than half the max array size, it will use a negative slope, and add 6000 (to offset the negative line) providing a triangle. It will then call `DAC_volt_conv()` and return this value.

Full equation:

```
    Input <= max array size: (triInput * 2 * 1.137666412);
 Input > max array size: (triInput * 2 * -1.137666412) + 6000;
```

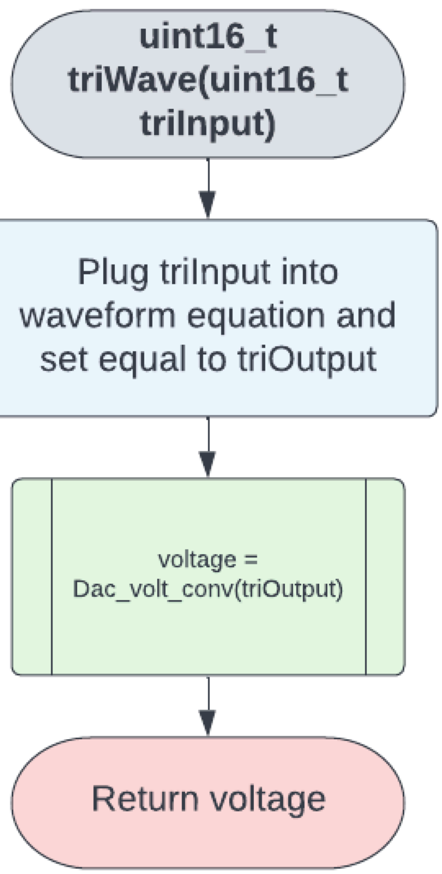


Figure 5. sawWave() Flowchart

4.5. FSM States

The FSM in this system has 5 states: ST_WAVEFORM, ST_SINE, ST_TRI, ST_SAW, ST_SQUARE. Each state represents an important part of the lockbox.

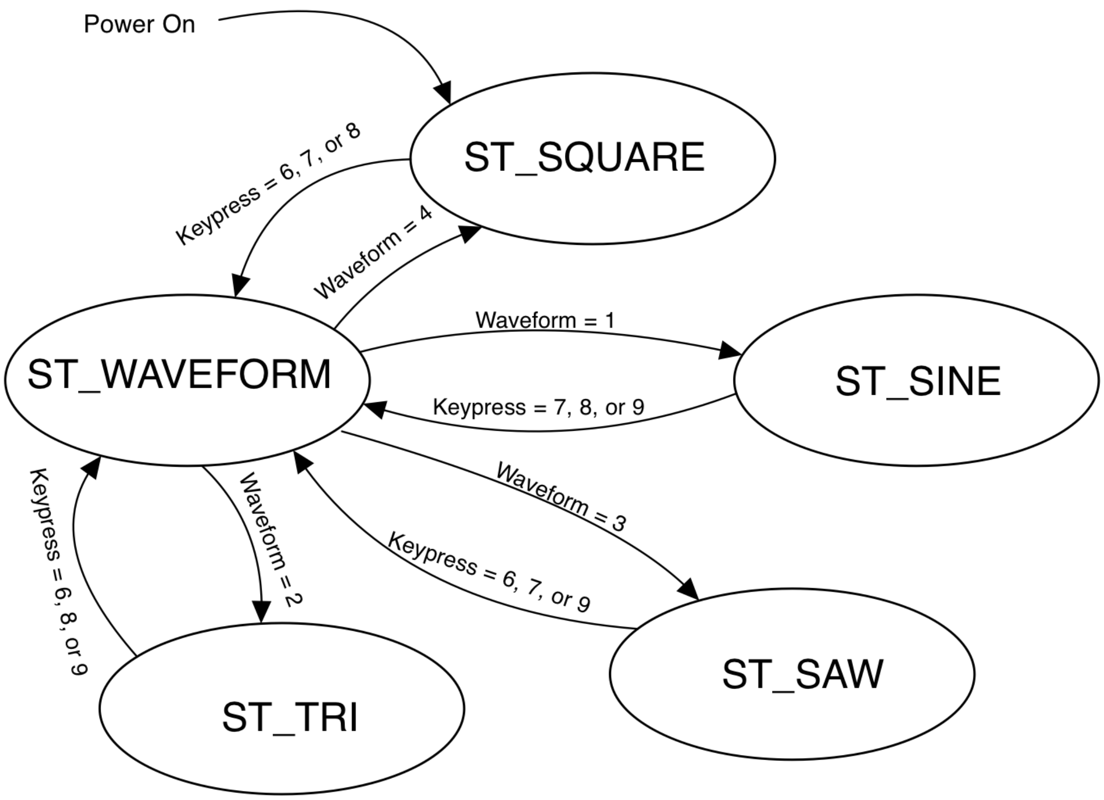


Figure 6. FSM State Diagram

4.5.1. ST_WAVEFORM

This state is used to select which state to move into depending on which key was pressed. At the beginning of the state, it clears and disables all flags and interrupts. Then it will take the keypress input and use a series of if statements to determine which state to move into. If a sine, triangle, or sawtooth waveform is selected, it will turn off the ARR and set CRR equal to the index value calculated previously. It will then only reenable CRR interrupts. If a square waveform is selected, it will set ARR and CRR to a time scaled by frequency and duty cycle. The equations are as follows:

```
TIM2->ARR = (CLK_40MHz / (100 * freq));
TIM2->CCR1 = ((TIM2->ARR * duty) / 10);
```

It will then reenable both global and CRR interrupts. It will also set a waveform variable that is used in the interrupt handler function.

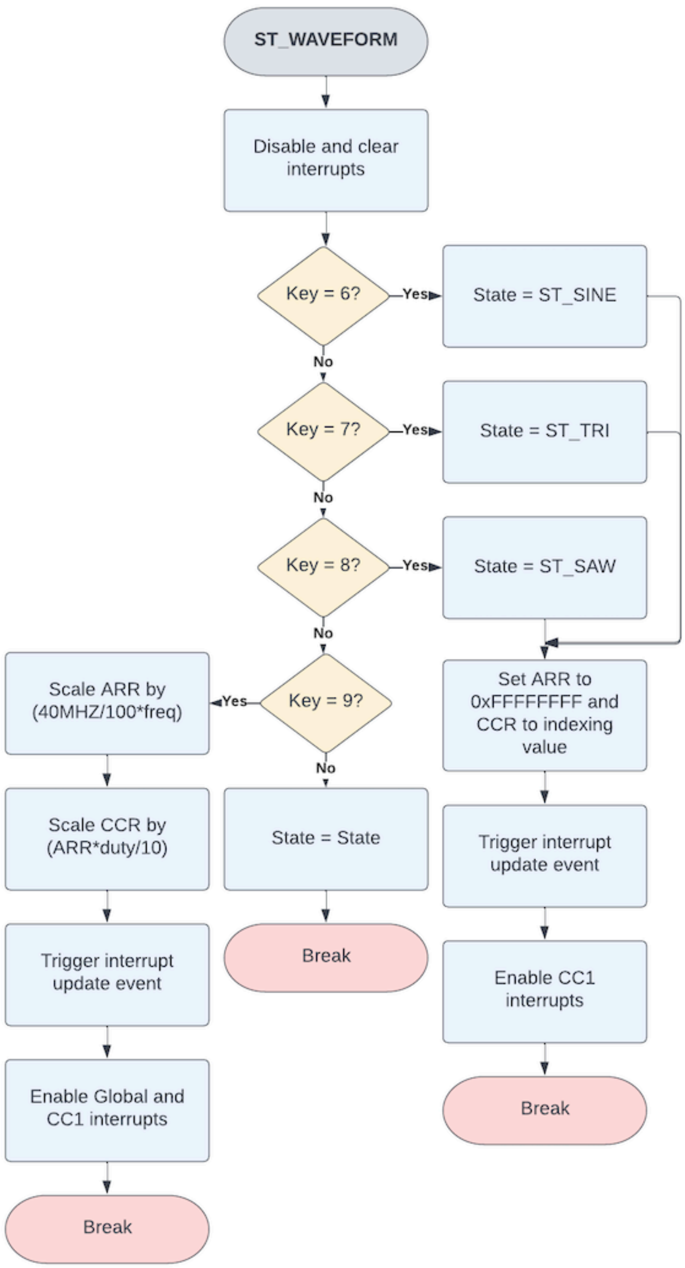


Figure 7. ST_WAVEFORM Flowchart

4.5.2. ST_SINE

In this state, the FSM will wait until a key is pressed and if the key is between 1 or 5, it will set the frequency to its respective frequency. If it is a number between 7-9 it will move to `ST_WAVEFORM`.

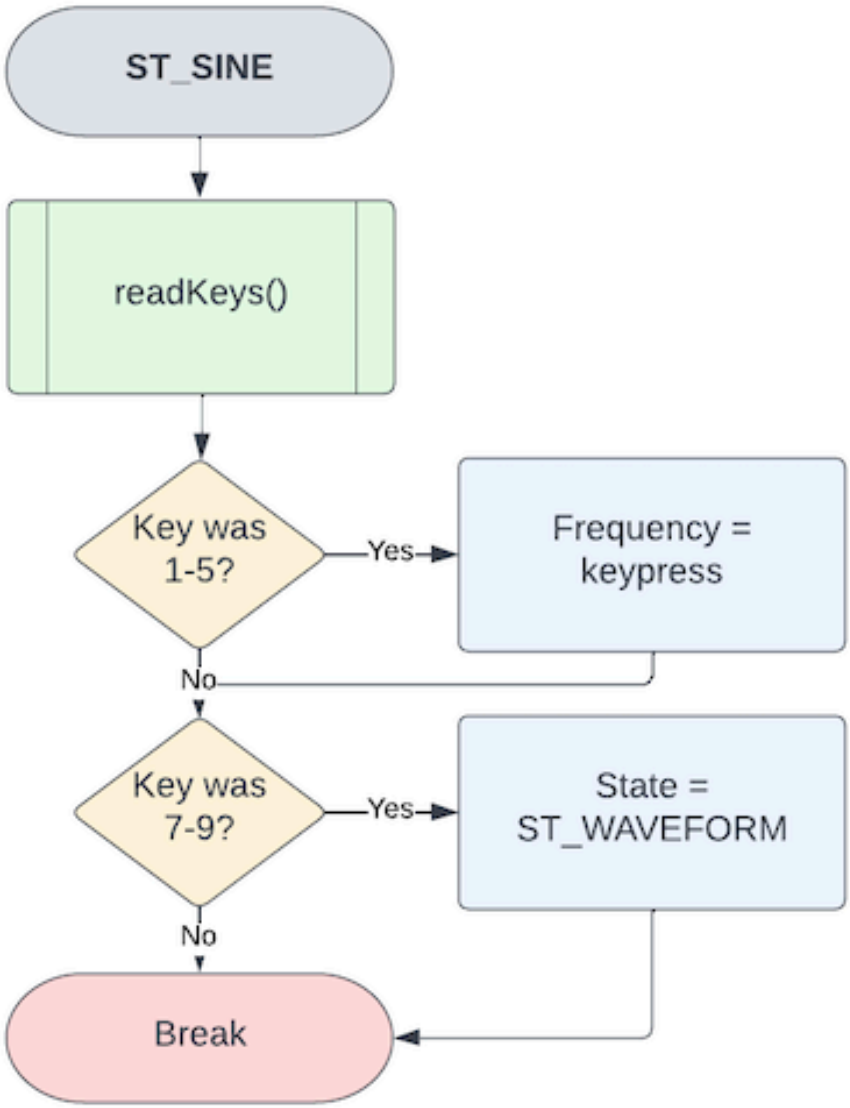


Figure 8. ST_SINE Flowchart

4.5.3. ST_TRI

In this state, the FSM will wait until a key is pressed and if the key is between 1 or 5, it will set the frequency to its respective frequency. If it is either 6, 8, or 9 it will move to `ST_WAVEFORM`.

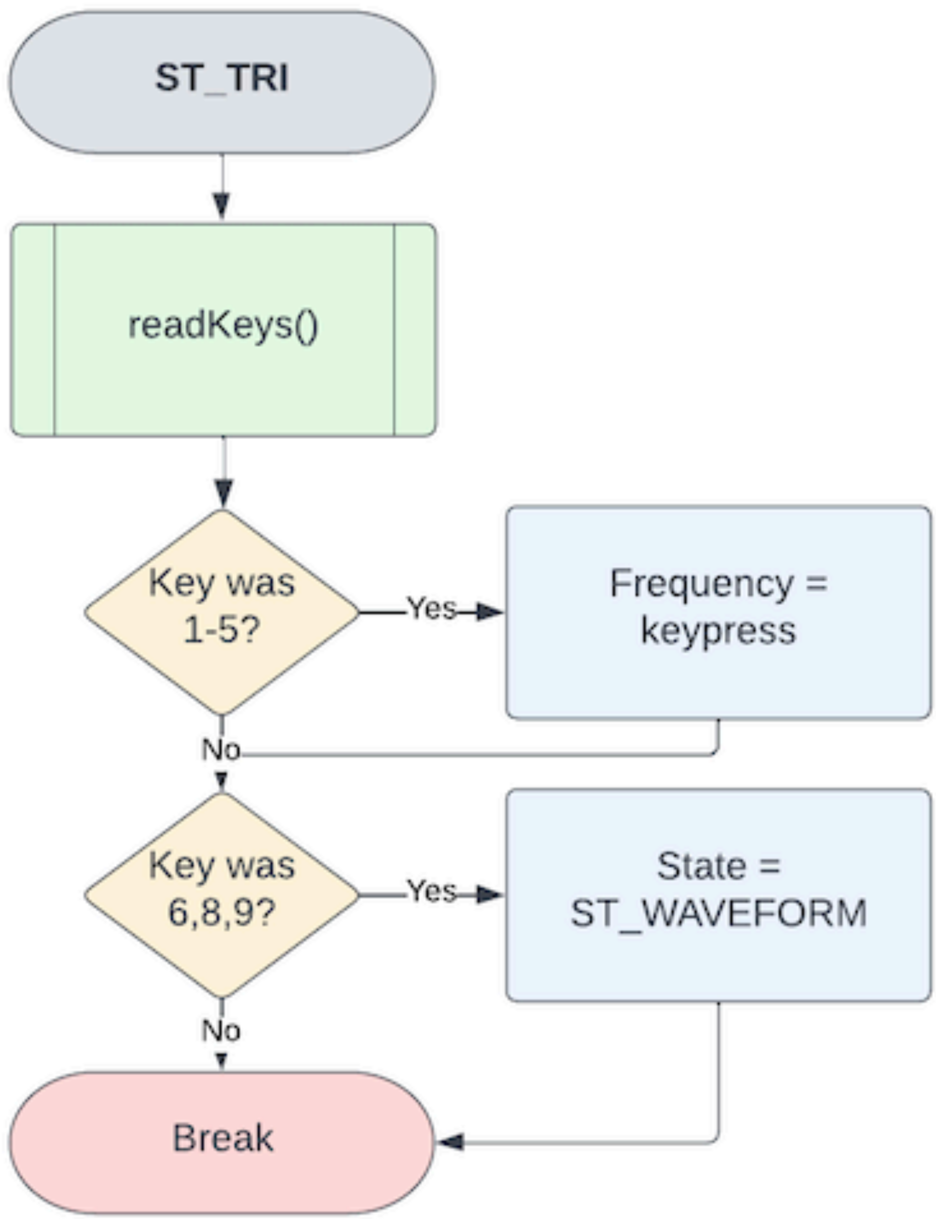


Figure 9. ST_TRI Flowchart

4.5.4. ST_SAW

In this state, the FSM will wait until a key is pressed and if the key is between 1 or 5, it will set the frequency to its respective frequency. If it is either 6, 7, or 9 it will move to `ST_WAVEFORM`.

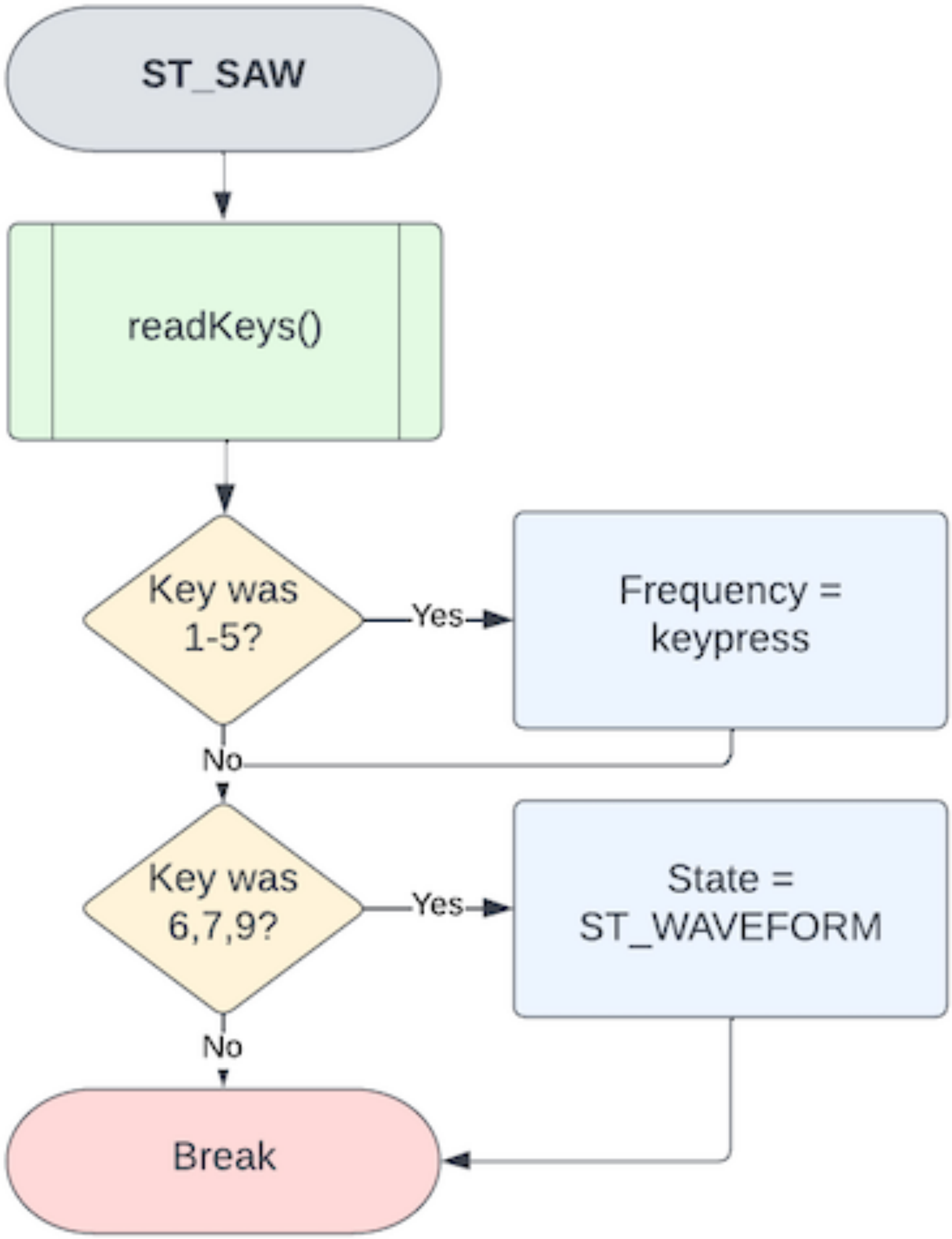


Figure 10. ST_SAW Flowchart

4.5.5. ST_SQUARE

In this state, the FSM will wait until a key is pressed and if the key is between 1 or 5, it will set the frequency to its respective frequency. If it is either 6, 7, or 9 it will move to `ST_WAVEFORM`. It will also check the duty cycle has been changed through the press of *, #, or 0. If * is pressed, it will decrement the duty cycle by 10% by adjusting CRR and ARR using the formula listed in `ST_WAVEFORM`. If # is pressed, it will increment the duty cycle by 10% and if 0 is pressed it will reset the duty cycle to 50%.

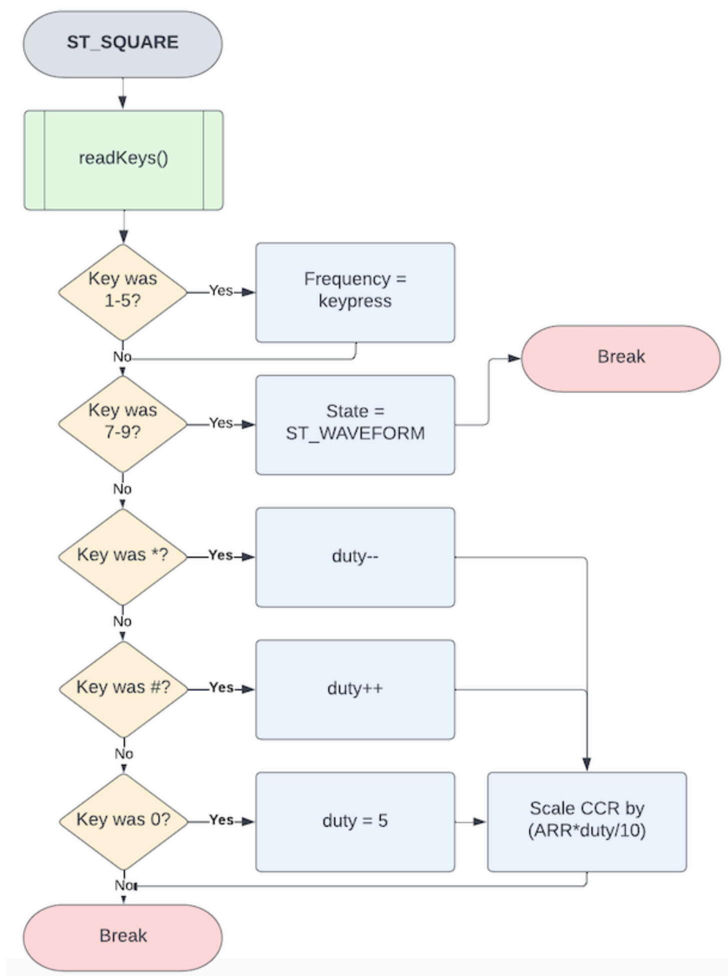


Figure 11. ST_SQUARE Flowchart

4.6. Subroutines and Functions

In each state, subroutines/functions are used to operate different aspects of the system. For example, reading the keypad requires the use of `keypad_read()`, and inside this is `keypoad_calc()`. If you want to get a ASCII value, you can use the subroutine `convertNum()`. In this section each function will be explained.

4.6.1. DAC Functions

`void DAC_write(uint16_t command)`

This function works by initializing variables `hiNibble` and `loNibble`. It will then set `hiNibble` equal to `0x3000 (0011 …)` This is done because the DAC requires the upper 4 bits of the data sent to be `0011`. It will then set `loNibble` equal to `command`. It then ORs together `hiNibble` and `loNibble` to get the overall command/data to output. Lastly, it will then output `command` to the SPI data register.

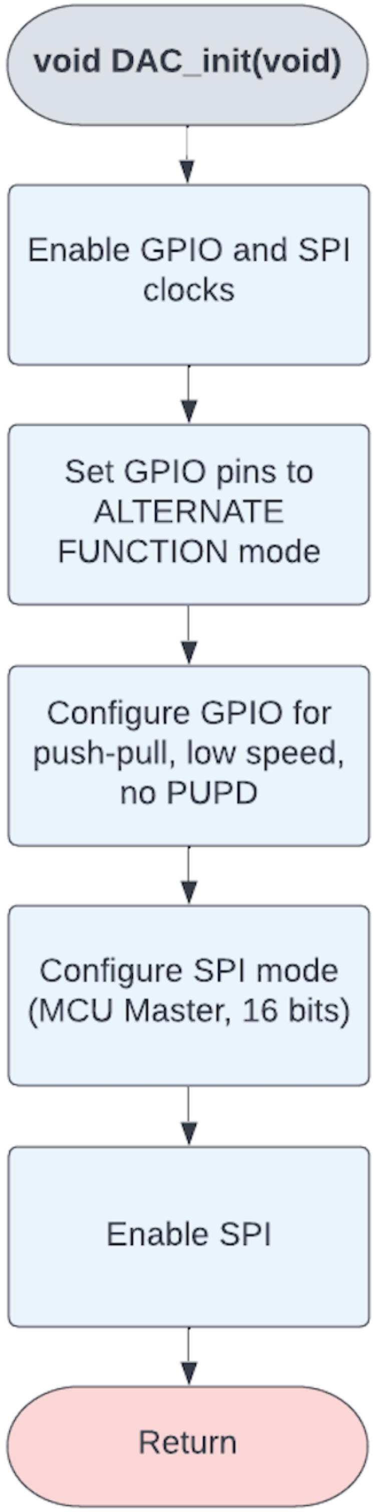


Figure 12. DAC_init() Flowchart

`uint16_t DAC_volt_conv(uint16_t command)`

This function works by taking in a desired voltage and converting it into a binary value that works with the DAC. The equation used is taken from the DAC reference manual and adapted/calibrated for proper voltage output.

$$V_{OUT} = \frac{(V_{REF} \times D_n)}{2^n} G$$

Where:

V_{REF}	=	External voltage reference
D_n	=	DAC input code
G	=	Gain Selection
	=	2 for <$\overline{GA}$> bit = 0
	=	1 for <$\overline{GA}$> bit = 1
n	=	DAC Resolution
	=	8 for MCP4901
	=	10 for MCP4911
	=	12 for MCP4912

The above equation came from the DAC reference manual and was used to create the basic formula.

```
voltage = ((voltage * (1.2412))
```

This was found by taking the max number of bits, $2^{12} = 4096$ and dividing it by $V_{REF} = 3300$. This gives a scaling factor of 1.2412.

Following the technical note in the lab manual, the DAC was calibrated by testing and comparing actual and desired values. An equation of best fit was found from this and it was implemented into the original equation.

Below is the final equation:

```
voltage = ((voltage * (1.2412)) / 1.0042) - 3.5727;
```

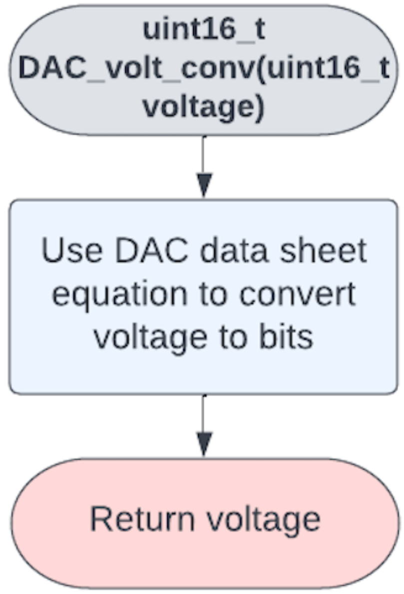


Figure 13. DAC_volt_conv() Flowchart

4.6.2. Keypad Functions

`uint8_t keypad_read(void)`

This function works by initializing variables `rows`, `cols`, `button`, and `row_output = 1`. These variables are used to keep track of what row and column is currently being incremented, what button is being pressed, and what value to output to the rows. After this initialization, the program reads the columns and checks if any of them are high. If one of these columns were to be high, that would mean that a button is being pressed because all rows were set high during the initialization. If no button was pressed, return an arbitrary very large value. However, if a button is pressed the program will then increment the rows one by one and will check if any columns are high as the rows are incremented. If a column is high, it will save the value of the rows and columns, then transfer those values into `uint8_t keypad_calc(uint8_t)`. After `uint8_t keypad_calc(uint8_t)` returns the value of the button press, it will save into the variable `button`. Lastly, the system will set all the rows high again and then return the value `button`.

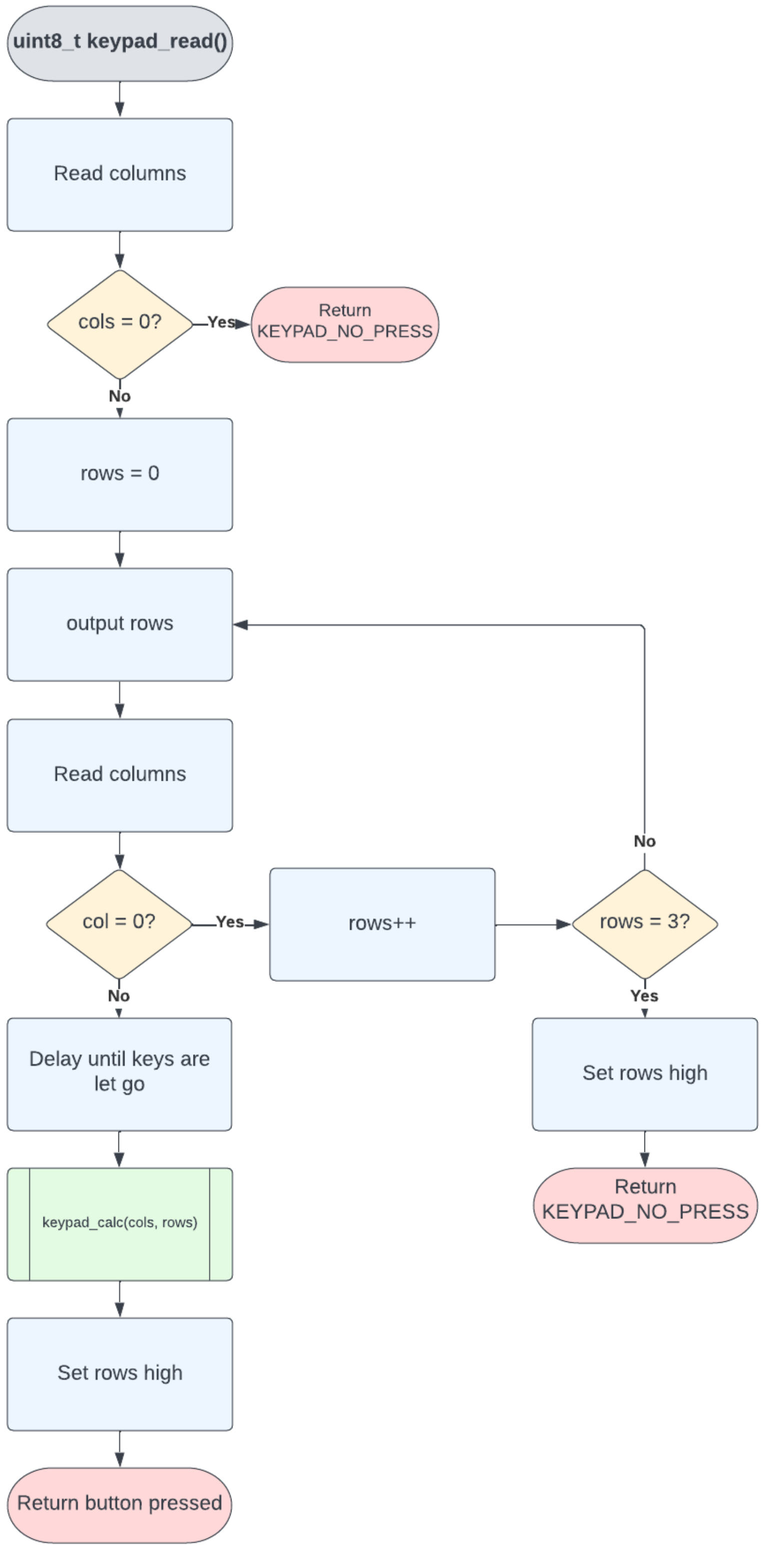


Figure 14. keypad_read() Flowchart

`uint8_t keypad_calc(uint8_t cols, uint8_t rows)`

This function works by taking the values `cols` and `rows` from the `keypad_read` function and using a math expression or if-statements to determine which value was pressed. If the rows are between 1-3 and columns are between 1-3, the system will use the following equation to determine what digit was pressed:

$$\text{key\_press} = \left((\text{rows} * 3) + (\text{cols} * 1) + 1\right)$$

(Eqn. 1)

If the column value is 4, then the key pressed must be a letter (because all letters are in column 4) and it will use the row value to determine the ASCII hex value corresponding to the letter with the following formula:

$$\text{key\_press} = \text{KEYPAD\_A} + \text{rows};$$

(Eqn. 2)

KEYPAD_A's hex value is equivalent to 0x41. If the row is 1, then the equation will add 1 to 0x41, obtaining 0x42 or the ASCII letter B. The same applies to the other characters.

Then, using if statements, the system will check if *, #, or 0 was pressed. If it cannot find any value it will return an arbitrary value identifying no key was pressed, otherwise it will return the value calculated.

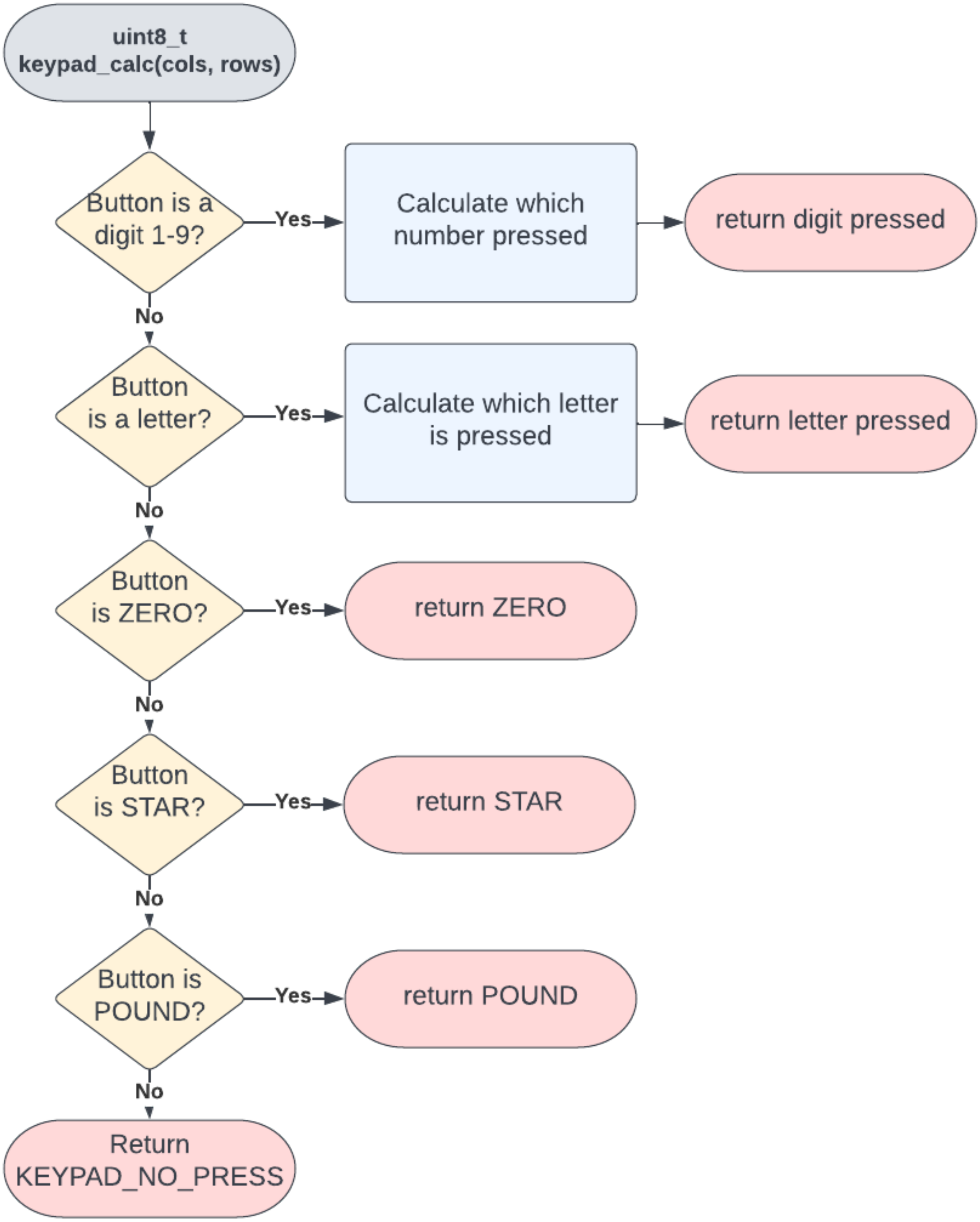


Figure 15. keypad_calc() Flowchart

`uint8_t convertNum(uint8_t num)`

This function simply adds hex 0x30 to the digit returned by `keypad_read`. The reason being is because zero is equivalent to 0x30, 1 is equivalent to 0x31, 2 is equivalent to 0x32, and so on. So to convert from a digit to an ASCII char, this is needed. The reason for this conversion is to save the digit in integer/decimal form to `pin` and then print the ASCII char.

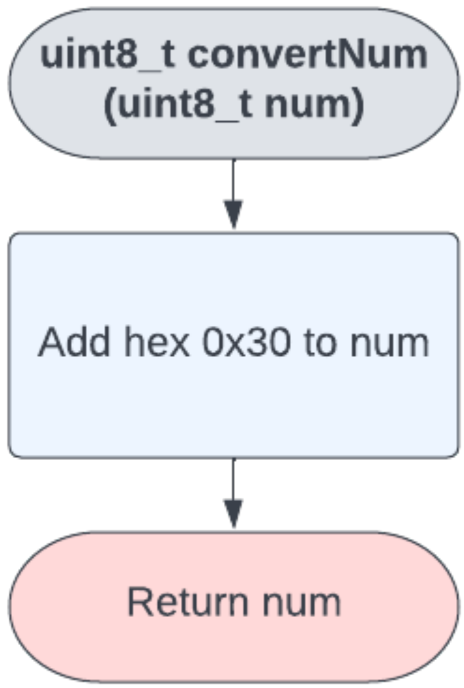


Figure 16. convertNum() Flowchart

`uint8_t readKeys(void)`

This function simply reads the keypads by calling `keypad_read(void)` and checks if any button was pressed. If no button was pressed, it will keep reading the keypads until a button is pressed. When a button is finally pressed, it will return the value of that button.

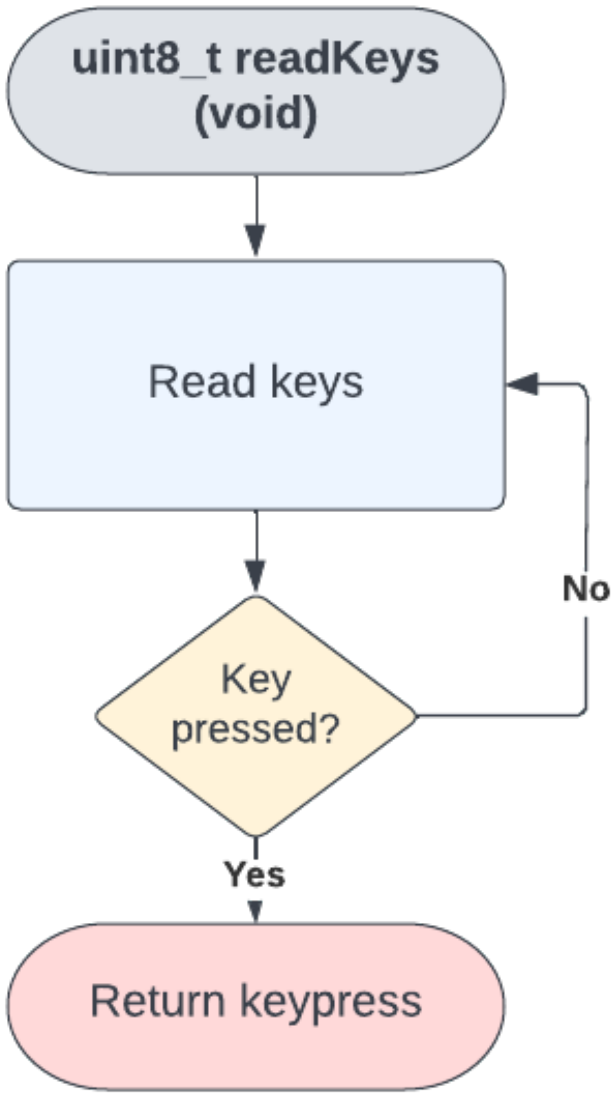


Figure 17. readKeys() Flowchart

5. Appendices

Appendix A – References

[1] P. Hummel and J. Green, “STM32 Lab Manual,” *Google Docs*. [Online]. Available: https://docs.google.com/document/d/1NdE5188B2JWkEPdFAOdvxrEYo0R90Cg7wsG6oqHYOwk/edit#. [Accessed: 20-Apr-2022].

[2] “ST32L476 Data Sheet,” *ST.com*. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32l476je.pdf. [Accessed: 20-Apr-2022].

[3] “ST32L476 Reference Manual,” *ST.com*. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0351-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. [Accessed: 20-Apr-2022].

[4] M. Tsoi, “Incomplete Guide to C (With A Focus on Embedded Systems Development),” 29-Mar-2021. .

[5] “MCP4901/4911/4921 Datasheet.” Microship Technology Inc., 2010.

Appendix B – Source Code

Source code is attached on the following pages:

```c
// EE 329 - 01, Ethan Najmy
// Project #2 - Function Generator

// -------- main.c --------//
#include "main.h"
#include "keypad.h"
#include "DAC.h"
#include <math.h>

#define TIM2_IRQn 28
#define DAC_3Volts 3540
#define DAC_0Volts 0000

#define CLK_40MHz 40000000

// Defined for a 50% duty cycle square wave
#define ARR_MAX 400000
#define CCR1_MAX 200000

// Will turn off ARR
#define ARR_ON 0xFFFFFFFF

// Calculated values
#define ARRAY_SIZE 2640
#define INDEX 152           // Max Resolution

#define PI 3.14159265358979323846

// Declare functions
void SystemClock_Config(void);
uint16_t sineWave(uint16_t count);
uint16_t sawWave(uint16_t count);
uint16_t triWave(uint16_t count);

// Initialize variables
uint16_t sine_array[ARRAY_SIZE], tri_array[ARRAY_SIZE], saw_array[ARRAY_SIZE];

// Waveform = 4 for initial startup into square with 100Hz freq and 50% duty
uint8_t keypress, waveform = 4, freq = 1, duty = 5;
uint16_t count = 0;



int main(void){
    // Create FSM
    typedef enum {ST_WAVEFORM, ST_SINE, ST_TRI, ST_SAW, ST_SQUARE} State_Type;
    State_Type state = ST_SQUARE;

    // Create Lookup tables for each wave
    for (uint16_t sineCount = 0; sineCount < ARRAY_SIZE; sineCount++){
        sine_array[sineCount] = sineWave(sineCount);
    }
    for (uint16_t triCount = 0; triCount < ARRAY_SIZE; triCount++){
        tri_array[triCount] = triWave(triCount);
    }

    for (uint16_t sawCount = 0; sawCount < ARRAY_SIZE; sawCount++){
        saw_array[sawCount] = sawWave(sawCount);
    }

    // Initialize
    HAL_Init();
    SystemClock_Config();

    DAC_init();
    keypad_init();

    // Turn on Timer
```

```c
    RCC->APB1ENR1 |= RCC_APB1ENR1_TIM2EN;
    // Enable Auto-reload preload
    TIM2->CR1 = (TIM_CR1_ARPE);

    // Set ARR and CCR1 for 100Hz, 50% duty cycle wave
    TIM2->ARR = (CLK_40MHz / (100 * freq));
    TIM2->CCR1 = ((TIM2->ARR * duty) / 10);

    // Enable interrupt flags
    TIM2->EGR |= (TIM_EGR_UG);
    TIM2->DIER |= (TIM_DIER_UIE);
    TIM2->DIER |= (TIM_DIER_CC1IE);


    // Enable global interrupts
    NVIC->ISER[0] = (1 << (TIM2_IRQn & 0x1F));
    __enable_irq();

    // Start timer
    TIM2->CR1 |= TIM_CR1_CEN;

    while (1) {

        // FSM!
        switch(state){

        // Used to switch between waveforms
        case ST_WAVEFORM:{

            // Disable all interrupts and clear flags
            TIM2->DIER &= ~(TIM_DIER_UIE | TIM_DIER_CC1IE);
            TIM2->DIER &= ~(TIM_SR_UIF | TIM_SR_CC1IF);

            // If sine wave selected:
            if (keypress == 6){

                // Used to identify waveform in IRQ
                waveform = 1;

                // Turn off ARR and set CCR1 to max res.
                TIM2->ARR = ARR_ON;
                TIM2->CCR1 = INDEX;

                // Update interrupt event!!!!
                TIM2->EGR |= TIM_EGR_UG;
                count = 0;

                // Re-enable CCR1 ONLY
                TIM2->DIER |= (TIM_DIER_CC1IE);

                // Move to sine wave
                state = ST_SINE;
                break;

                // If triangle wave selected:
            } else if (keypress == 7){

                // All same comments as above
                waveform = 2;

                TIM2->ARR = ARR_ON;
                TIM2->CCR1 = INDEX;
                TIM2->EGR |= TIM_EGR_UG;
                count = 0;

                TIM2->DIER |= (TIM_DIER_CC1IE);

                state = ST_TRI;
```

```
                break;

                // If saw wave selected:
            } else if (keypress == 8){

                // All same comments as above
                waveform = 3;

                TIM2->ARR = ARR_ON;
                TIM2->CCR1 = INDEX;
                TIM2->EGR |= TIM_EGR_UG;
                count = 0;

                TIM2->DIER |= (TIM_DIER_CC1IE);

                state = ST_SAW;
                break;

                // If square wave selected
            } else if (keypress == 9){
                waveform = 4;

                // Set ARR and CCR using formulas according to
                // frequency and duty cycle selected.
                TIM2->ARR = (CLK_40MHz / (100 * freq));
                TIM2->CCR1 = ((TIM2->ARR * duty) / 10);
                TIM2->EGR |= TIM_EGR_UG;
                count = 0;

                // Enable both ARR and CCR
                TIM2->DIER |= (TIM_DIER_UIE | TIM_DIER_CC1IE);

                state = ST_SQUARE;
                break;

                // If for some reason no key was pressed
                // stay in the same state
            } else {
                state = state;
                break;
            }
            break;
        }

        // Sawtooth Wave
        case ST_SAW:{
            // Wait for keypress
            keypress = readKeys();

            // If a number between 1-5, change freq
            if (keypress > 0 && keypress < 6)
                freq = keypress;

            // If a number to change waves, go to ST_WAVEFORM
            if ((keypress >= 6 && keypress <= 7) || keypress == 9){
                state = ST_WAVEFORM;
                break;
            }
            break;
        }

        case ST_SINE:{

            // Same comments as above
            keypress = readKeys();

            if (keypress > 0 && keypress < 6)
                freq = keypress;
```

```c

            if (keypress >= 7 && keypress <= 9){
                state = ST_WAVEFORM;
                break;
            }
            break;
        }

        case ST_TRI:{

            // Same comments as above
            keypress = readKeys();

            if (keypress > 0 && keypress < 6)
                freq = keypress;

            if (keypress == 6 || (keypress >= 8 && keypress <= 9)){
                state = ST_WAVEFORM;
                break;
            }
            break;
        }

        case ST_SQUARE:{
            keypress = readKeys();

            // If changing frequencies, update ARR and CCR
            if (keypress > 0 && keypress < 6){
                freq = keypress;
                TIM2->ARR = (CLK_40MHz / (100 * freq));
                TIM2->CCR1 = ((TIM2->ARR * duty) / 10);
            }

            // If wanting to decrement duty cycle, only adjust CCR
            if (keypress == KEYPAD_STAR){
                // decrement duty by 10%
                if (duty > 1){
                    duty--;
                    TIM2->CCR1 = ((TIM2->ARR * duty) / 10);
                } else {
                    duty = 1;
                }
            }

            // If wanting to increment duty cycle, only adjust CCR
            if (keypress == KEYPAD_POUND){
                // increment duty by 10%
                if (duty < 9){
                    duty++;
                    TIM2->CCR1 = ((TIM2->ARR * duty) / 10);
                } else {
                    duty = 9;
                }
            }

            // Will reset duty cycle
            if (keypress == KEYPAD_0){
                // reset duty to 50%
                duty = 5;
                TIM2->CCR1 = ((TIM2->ARR * duty) / 10);
            }

            // Change states if wave selected.
            if (keypress >= 6 && keypress <= 8){
                state = ST_WAVEFORM;
                break;
            }

```

```
            break;
        }
        }
    }
}

// INTERRUPT HANDLER
void TIM2_IRQHandler(void){

    // Switch statement to choose how to handle the
    // interrupt depending on waveform variable
    switch(waveform){

    // Waveform is SINE
    case 1:{
        // If CCR triggered, reset flag
        if (TIM2->SR & TIM_SR_CC1IF){
            TIM2->SR = ~(TIM_SR_CC1IF);

            // Write to the DAC using the array lookup
            DAC_write(sine_array[count]);

            // If count exceeded the array (minus freq
            // to obtain no jumping b/w points), reset
            // count to zero.
            if (count >= (ARRAY_SIZE - freq))
                count = 0;
            // Otherwise increment count by freq
            // (Increment by freq to change sampling rate
            // which allows for frequency changes without
            // changing CCR).
            else
                count += freq;

            // Keep incrementing CCR so it doesn't stop
            TIM2->CCR1 += INDEX;
        }
        break;
    }

    // Triangle Wave
    case 2:{

        // Same exact comments just with tri_array
        if (TIM2->SR & TIM_SR_CC1IF){
            TIM2->SR = ~(TIM_SR_CC1IF);
            DAC_write(tri_array[count]);
            if (count >= (ARRAY_SIZE))
                count = 0;
            else
                count += freq;
            TIM2->CCR1 += INDEX;
        }
        break;
    }

    // Sawtooth Wave
    case 3:{

        // Same exact comments just with saw_array
        if (TIM2->SR & TIM_SR_CC1IF){
            TIM2->SR = ~(TIM_SR_CC1IF);
            DAC_write(saw_array[count]);
            if (count >= (ARRAY_SIZE))
                count = 0;
            else
                count += freq;
            TIM2->CCR1 += INDEX;
```

```
        }
        break;
    }

    // Square Wave
    case 4:{

        // If ARR triggered, clear flag and write
        // 3V to the DAC
        if (TIM2->SR & TIM_SR_UIF){
            TIM2->SR &= ~(TIM_SR_UIF);
            DAC_write(DAC_3Volts);
        }
        // IF CCR triggered, clear the flag and
        // write 0V to the DAC
        if (TIM2->SR & TIM_SR_CC1IF){
            TIM2->SR &= ~(TIM_SR_CC1IF);
            DAC_write(DAC_0Volts);
        }
        break;
    }
    }
}

// Array lookup functions

uint16_t sineWave(uint16_t sinInput){
    uint16_t sine;

    // Use double to be as accurate as possible
    double DAC_in;

    // Formula for a sine wave adjusted for 3V exact output
    // Multiply and add by 1475 for 3Vpp, 1.5V offset
    // In the thousands because thats how the DAC interprets
    DAC_in = (1475 * sin(((2 * PI * sinInput) / ARRAY_SIZE)) + 1475);

    // Convert back to int
    DAC_in = (uint16_t)DAC_in;

    // Convert to voltage that can be written to DAC and return
    sine = DAC_volt_conv(DAC_in);
    return sine;
}

uint16_t sawWave(uint16_t sawInput){
    uint16_t saw, DAC_in;

    // Multiply count input by (3000/2640)
    // (3V out / array size) (rise / run)
    // Give scaling factor to ensure points
    // Reach 3V in size of array
    DAC_in = sawInput * 1.137666412;

    // Convert to DAC voltage
    saw = DAC_volt_conv(DAC_in);
    return saw;
}

uint16_t triWave(uint16_t triInput){
    uint16_t tri, DAC_in;

    // Same concept as above with the (3000/2640)
    // However, if we have reached over half the
    // array, begin inverting the values with a
    // negative slope and a 6000 offset so the
    // values begin at 3V rather than -3V
    if (triInput <= (ARRAY_SIZE/2)){
```

```c
        DAC_in = (triInput * 2 * 1.137666412);
        tri = DAC_volt_conv(DAC_in);
        return tri;
    } else {
        DAC_in = (triInput * 2 * -1.137666412) + 6000;
        tri = DAC_volt_conv(DAC_in);
        return tri;
    }
}

// -----------------------------------------------------------
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /** Configure the main internal regulator output voltage
     */
    if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
    {
        Error_Handler();
    }

    /** Initializes the RCC Oscillators according to the specified parameters
     * in the RCC_OscInitTypeDef structure.
     */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_MSI;
    RCC_OscInitStruct.MSIState = RCC_MSI_ON;
    RCC_OscInitStruct.MSICalibrationValue = 0;
    RCC_OscInitStruct.MSIClockRange = RCC_MSIRANGE_6;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_MSI;
    RCC_OscInitStruct.PLL.PLLM = 1;
    RCC_OscInitStruct.PLL.PLLN = 20;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
    RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
    RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        Error_Handler();
    }

    /** Initializes the CPU, AHB and APB buses clocks
     */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
            |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2) != HAL_OK)
    {
        Error_Handler();
    }
}

void Error_Handler(void)
{
    /* USER CODE BEGIN Error_Handler_Debug */
    /* User can add his own implementation to report the HAL error return state */
    __disable_irq();
    while (1)
    {
    }
    /* USER CODE END Error_Handler_Debug */
}

```

```
#ifdef  USE_FULL_ASSERT
void assert_failed(uint8_t *file, uint32_t line)
{
    /* USER CODE BEGIN 6 */
    /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
    /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

```

```c
/* ----------- keypad.h ------------ */
#ifndef SRC_KEYPAD_H_
#define SRC_KEYPAD_H_

// Include #defines
#define ROW0 0x01                                    // Define ROW0 as 'pin 1'
#define ROW1 0x02                                    // Define ROW1 as 'pin 2'
#define ROW2 0x04                                    // Define ROW2 as 'pin 3'
#define ROW3 0x08                                    // Define ROW3 as 'pin 4'
#define COL0 0x10                                    // Define COL0 as 'pin 5'
#define COL1 0x20                                    // Define COL1 as 'pin 6'
#define COL2 0x40                                    // Define COL2 as 'pin 7'
#define COL3 0x80                                    // Define COL3 as 'pin 8'
#define KEYPAD_ROW_MASK (ROW0 | ROW1 | ROW2 | ROW3) // KEYPAD_ROW_MASK = 0x0F
                                                     // - when ANDed, only gives ROW values
#define KEYPAD_COL_MASK (COL0 | COL1 | COL2 | COL3) // KEYPAD_COL_MASK = 0xF0
                                                     // - when ANDed, only gives COL values
#define KEYPAD_NO_PRESS 0xF3                         // Return 0b11110011 ('?') when no key is pressed
#define KEYPAD GPIOD                                 // Define keypad as GPIOD

#define KEYPAD_0 0x00
#define KEYPAD_A 0x41
#define KEYPAD_B 0x42
#define KEYPAD_C 0x43
#define KEYPAD_D 0x44
#define KEYPAD_STAR 0x2A
#define KEYPAD_POUND 0x23

// Include function declarations / prototypes
void keypad_init(void);
uint8_t keypad_read(void);
uint8_t keypad_calc(uint8_t, uint8_t);
uint8_t convertNum(uint8_t);
uint8_t readKeys(void);

#endif /* SRC_KEYPAD_H_ */

```

```c
/* ---------- keypad.c ----------- */
#include "main.h"
#include "keypad.h"

void keypad_init(void) {
  /* -- Configure GPIO D -- */
  RCC->AHB2ENR |= (RCC_AHB2ENR_GPIODEN);    //Enable GPIO D Clock

  KEYPAD->MODER &= ~(GPIO_MODER_MODE0       // Set MODE[0:7][1:0] to 0
          | GPIO_MODER_MODE1                // Will keep pins 4-7 as 0 for input mode
          | GPIO_MODER_MODE2
          | GPIO_MODER_MODE3
          | GPIO_MODER_MODE4
          | GPIO_MODER_MODE5
          | GPIO_MODER_MODE6
          | GPIO_MODER_MODE7);

  KEYPAD->MODER |= (GPIO_MODER_MODE0_0      // Set MODE[0:3][0] to 1
          | GPIO_MODER_MODE1_0              // Set as output pins
          | GPIO_MODER_MODE2_0
          | GPIO_MODER_MODE3_0);

  KEYPAD->OTYPER &= ~(GPIO_OTYPER_OT0       // Set OTYPE[0:3] to 0
          | GPIO_OTYPER_OT1                 // Set as push-pull
          | GPIO_OTYPER_OT2
          | GPIO_OTYPER_OT3);

  KEYPAD->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED0 // Set OSPEED[0:3] to 0
          | GPIO_OSPEEDR_OSPEED1            // Set output speed as low
          | GPIO_OSPEEDR_OSPEED2
          | GPIO_OSPEEDR_OSPEED3);

  KEYPAD->PUPDR &= ~(GPIO_PUPDR_PUPD0       // Set PUPD[0:7][1:0] to 0
          | GPIO_PUPDR_PUPD1                // Will keep pins 0-3 at 0
          | GPIO_PUPDR_PUPD2
          | GPIO_PUPDR_PUPD3
          | GPIO_PUPDR_PUPD4
          | GPIO_PUPDR_PUPD5
          | GPIO_PUPDR_PUPD6
          | GPIO_PUPDR_PUPD7);

  KEYPAD->PUPDR |= (GPIO_PUPDR_PUPD4_1      // Set PUPD[4:7][1] to 1
          | GPIO_PUPDR_PUPD5_1              // Enables pull-down resistors
          | GPIO_PUPDR_PUPD6_1
          | GPIO_PUPDR_PUPD7_1);

  KEYPAD->ODR &= ~(KEYPAD_ROW_MASK);        // Set rows to high
  KEYPAD->ODR |= (KEYPAD_ROW_MASK);
}

// READ KEYPAD FUNCTION
uint8_t keypad_read(void) {

    /* -- Initialize variables -- */

    // rows, cols - keep track of row # and column #
    // button - saves key pressed on keypad
    // row_output - enables rows to be turned high one at a time
    uint8_t rows, cols, button, row_output = 1;

    cols = KEYPAD->IDR & KEYPAD_COL_MASK;   // Read input register and only save columns
    if (cols == 0)                          // If no keys pressed (cols = 0), return KEYPAD_NO_PRESS
        return KEYPAD_NO_PRESS;

    /* -- Row Incrementer -- */
    for(rows = 0; rows < 4; rows++) {
        KEYPAD->ODR &= ~(KEYPAD_ROW_MASK);
        KEYPAD->ODR |= row_output;                      // Set ROW0 high
```

```c
        row_output *= 2;                                // Multiply row by 2, output to ROW1,2,3
        cols = (KEYPAD->IDR & KEYPAD_COL_MASK);         // Read columns
        if (cols != 0) {                                // If no column high, skip loop

            while (KEYPAD->IDR & KEYPAD_COL_MASK){
                //  Delay while buttons are pressed
            }

            button = keypad_calc(cols,rows);            // Calc key press from row and col value
                                                        // and save to button
            KEYPAD->ODR &= ~(KEYPAD_ROW_MASK);          // Set rows high
            KEYPAD->ODR |= (KEYPAD_ROW_MASK);
            return button;                              // Return button


        }

    }
    KEYPAD->ODR &= ~(KEYPAD_ROW_MASK);              // Set rows high
    KEYPAD->ODR |= (KEYPAD_ROW_MASK);
    return KEYPAD_NO_PRESS;                         // Return 0xFF
}

// KEYPAD CALCULATE WHAT BUTTON WAS PRESSED
uint8_t keypad_calc(uint8_t cols, uint8_t rows) {
    uint8_t key_press;                              // Initialize variable to save key value

    cols = (cols >> 5);                             // Shift column value 5 bits, allows columns to
                                                    // be numbered 0,1,2,4

    if ((cols < 4) && (rows < 3)) {                 // 4x4 KEYPAD, check IF value is number 1-9
        key_press = ((rows*3)+(cols*1)+ 1);         // Use eqn to calculate number key pressed and
                                                    // add 0x30 to convert to ASCII value

    } else if (cols == 4) {                         // If a value in column 4 was pressed
        key_press = KEYPAD_A + rows;                // Add rows to KEYPAD_A (10),
                                                    // if row = 0, key_press = 10 (A),
                                                    // if row = 1, key_press = 11 (B), etc.

    } else if (rows == 3 && cols == 0) {            // If bottom left button, key pressed = *
        key_press = KEYPAD_STAR;

    } else if (rows == 3 && cols == 1) {            // If bottom middle button, key pressed = 0
        key_press = KEYPAD_0;

    } else if (rows == 3 && cols == 2) {            // If bottom right button, key pressed = #
        key_press = KEYPAD_POUND;

    } else
        key_press = KEYPAD_NO_PRESS;                // If nothing pressed, return 0xFF

    return key_press;                               // Return key_press value
}

uint8_t readKeys(void){
    uint8_t keypress;                           // Initialize Variable
    keypress = keypad_read();                   // Read keypad
    while (keypress == KEYPAD_NO_PRESS)         // Wait until button has been pressed
        keypress = keypad_read();
    return keypress;                        // Return button pressed
}

// USED TO CONVERT DIGIT 0-9 TO ASCII CHAR VALUE
uint8_t convertNum (uint8_t num) {
    num += 0x30;    // Add hex 0x30 and save as num.
    return num;         // Return num
}
```

```
/* ----------- DAC.h ------------ */
#ifndef SRC_DAC_H_
#define SRC_DAC_H_

// Include #defines
#define DACC GPIOA

// Include function definitions / prototypes

void DAC_init(void);
void DAC_write(uint16_t);
uint16_t DAC_volt_conv(uint16_t);

#endif /* SRC_DAC1_H_ */

```

```c
/* ----------- DAC.c ------------ */
#include "main.h"
#include "DAC.h"

// Function to Initialize the DAC
void DAC_init(void){
    /* -- Configure GPIO PA4, PA5, PA7 for SPI Control -- */

    //Enable GPIO A Clock
    RCC->AHB2ENR |= (RCC_AHB2ENR_GPIOAEN);
    RCC->APB2ENR |= (RCC_APB2ENR_SPI1EN);

    // Set MODER to alternate function mode
    DACC->MODER &= ~(GPIO_MODER_MODE4
            | GPIO_MODER_MODE5
            | GPIO_MODER_MODE7);
    DACC->MODER |= (GPIO_MODER_MODE4_1
            | GPIO_MODER_MODE5_1
            | GPIO_MODER_MODE7_1);

    // Set AFRL to AF5 for SPI1 controller
    DACC->AFR[0] &= ~(GPIO_AFRL_AFSEL4
            | GPIO_AFRL_AFSEL5
            | GPIO_AFRL_AFSEL7);
    DACC->AFR[0] |= (GPIO_AFRL_AFSEL4_2
            | GPIO_AFRL_AFSEL4_0
            | GPIO_AFRL_AFSEL5_2
            | GPIO_AFRL_AFSEL5_0
            | GPIO_AFRL_AFSEL7_2
            | GPIO_AFRL_AFSEL7_0);

    // Set OTYPER to output push pull
    DACC->OTYPER &= ~(GPIO_OTYPER_OT4
            | GPIO_OTYPER_OT5
            | GPIO_OTYPER_OT7);

    // Set OSPEED to low speed
    DACC->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED4
            | GPIO_OSPEEDR_OSPEED5
            | GPIO_OSPEEDR_OSPEED7);

    // Set PUPDR to no pull up pull down
    DACC->PUPDR &= ~(GPIO_PUPDR_PUPD4
            | GPIO_PUPDR_PUPD5
            | GPIO_PUPDR_PUPD7);


    /* -- Enable SPI Mode -- */

    /* CR1 code is as follows:
     * -- RXONLY = 0 (Enables Simplex Mode)
     * -- SSM = 0 (Hardware Chip Select Management)
     * -- LSBFIRST = 0 (Sending MSB first to DAC)
     * -- BR = 0 (Baud rate = f/2 (highest))
     * -- CPOL & CPHA = 0 (Sets clock phase and polarity
     *      to have a low idle and send data on rising edge)
     * -- MSTR = 1 (Set MCU to controller configuration) */
    SPI1->CR1 = (SPI_CR1_MSTR);

    /* CR2 code is as follows:
     * -- DS = 0xF (16 bit communications)
     * -- NSSP = 1 (chip select pulse mode)
     * -- SSOE = 1 (chip select enable) */
    SPI1->CR2 = (SPI_CR2_DS
            | SPI_CR2_NSSP
            | SPI_CR2_SSOE);

    /* -- Lastly, enable SPI -- */
```

```c
    SPI1->CR1 |= (SPI_CR1_SPE);
}

void DAC_write(uint16_t command){
    /* Set hiNibble to 0x3000, this is to ensure bits 15-12
     * sent to the DAC are 0011 as required by the DAC */
    uint16_t hiNibble = 0x3000;

    // Remove upper 4 bits of command
    uint16_t loNibble = (command & 0x0FFF);

    /* Set the upper 4 bits of command = hiNibble and
     * the lower 12 bits of command to loNibble, which
     * in our case is the command (or voltage level) */
    command = hiNibble | loNibble;

    // Output to SPI data register
    SPI1->DR = command;
}

uint16_t DAC_volt_conv(uint16_t voltage){
    /* The following formula has been adapted from
     * the DAC data sheet.
     * The 1.2412 is used to scale the voltage entered
     * into a range from 0-4096 (12 bits).
     * 1.2412 = 4096/3300 (total bits/max voltage)
     * The 1.0042 and -3.5727 are calibrations
     * used from recording desired data vs actual output
     * and came from an excel trendline as outlined
     * in technical note 5 of lab manual.
     */
    voltage = ((voltage * (1.2412)) / 1.0042) - 3.5727;
    return voltage;
}
```