Project 1: Function Generator

Cole Costa

CPE 316-01

Spring Quarter 2022

May 3, 2023

Professor Hummel

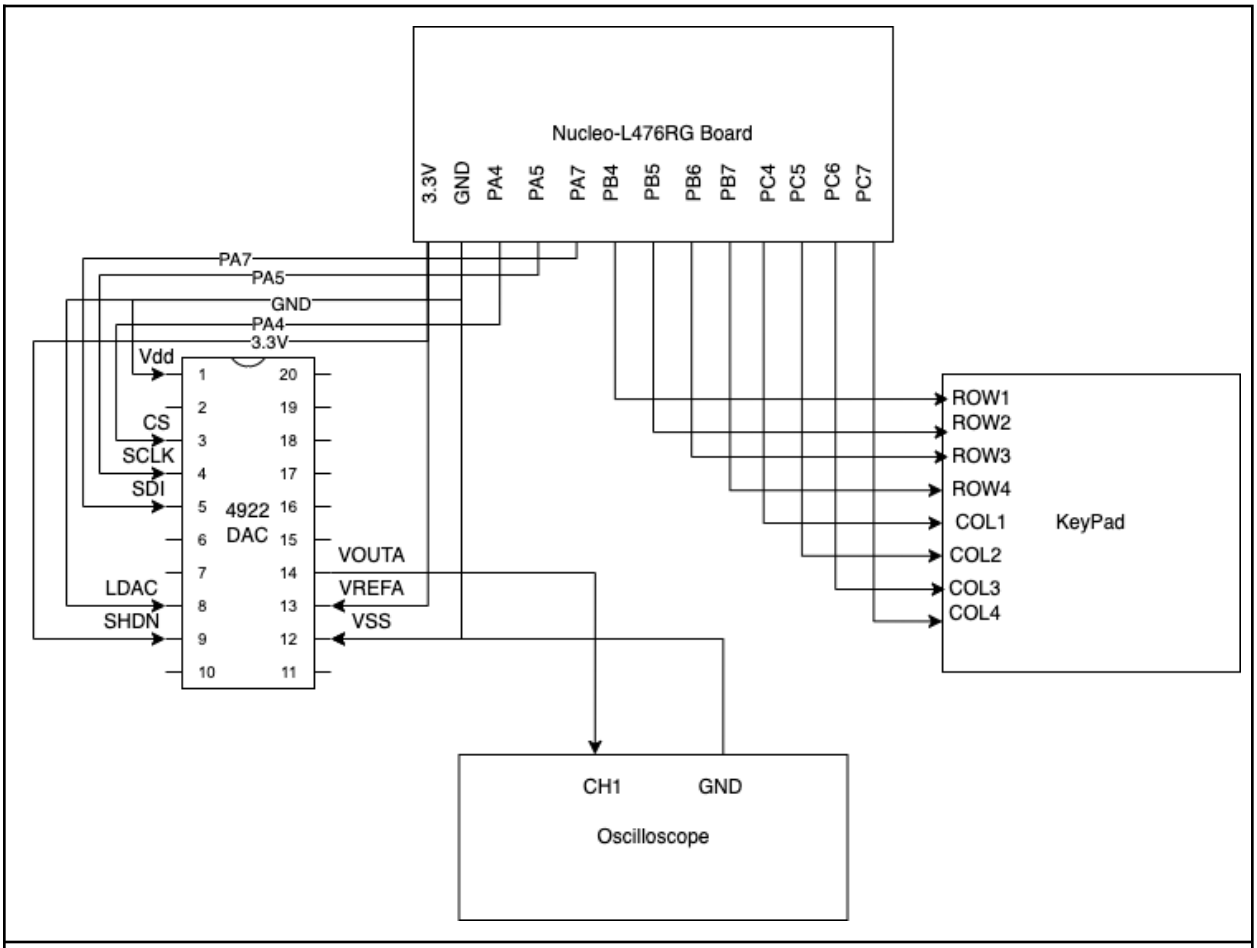
Behavior Description

The function generator designed in this experiment utilizes a 4922 DAC [1], a keypad, and a Nucleo-L476RG Board [2-3] in order to generate waveforms. On startup, the function generator displays a 100 Hz square wave with a 50% duty cycle. After this initialization, an FSM implemented in an infinite while loop inside of the main function allows the user to change the output waveform's type, frequency, and duty cycle via keypad input. The function generator allows for square, sawtooth, triangle, and sine waveforms that have adjustable frequencies from 100-500 Hz. The duty cycle of the square wave is also adjustable with a maximum of 90% and a minimum of 10%. All waveforms have a peak-to-peak voltage of 3.0 V and are biased at 1.5 V.

System Specifications

Clock Frequency	32 MHz [3]
Sample Rate / Maximum Resolution	42,000 samples/second
Minimum ISR Time	23 µs
$V_{OUT\ MAX}$	3.0 V
Frequency Range	100 - 500 Hz
Waveforms	Square, Sawtooth, Triangle, Sine
Keypad Type	Hard-Key
Keypad Size	4x4
Keypad Functionality	1-5 : Change frequency from 100 - 500 Hz 6: Sine Wave 7: Triangle Wave 8: Sawtooth Wave 9: Square Wave *: Decrease Square Wave Duty Cycle 10% 0: Reset Square Wave Duty Cycle to 50% #: Increase Square Wave Duty Cycle 10%
Initial Waveform	100 Hz Square Wave with 50% Duty Cycle

System Schematic

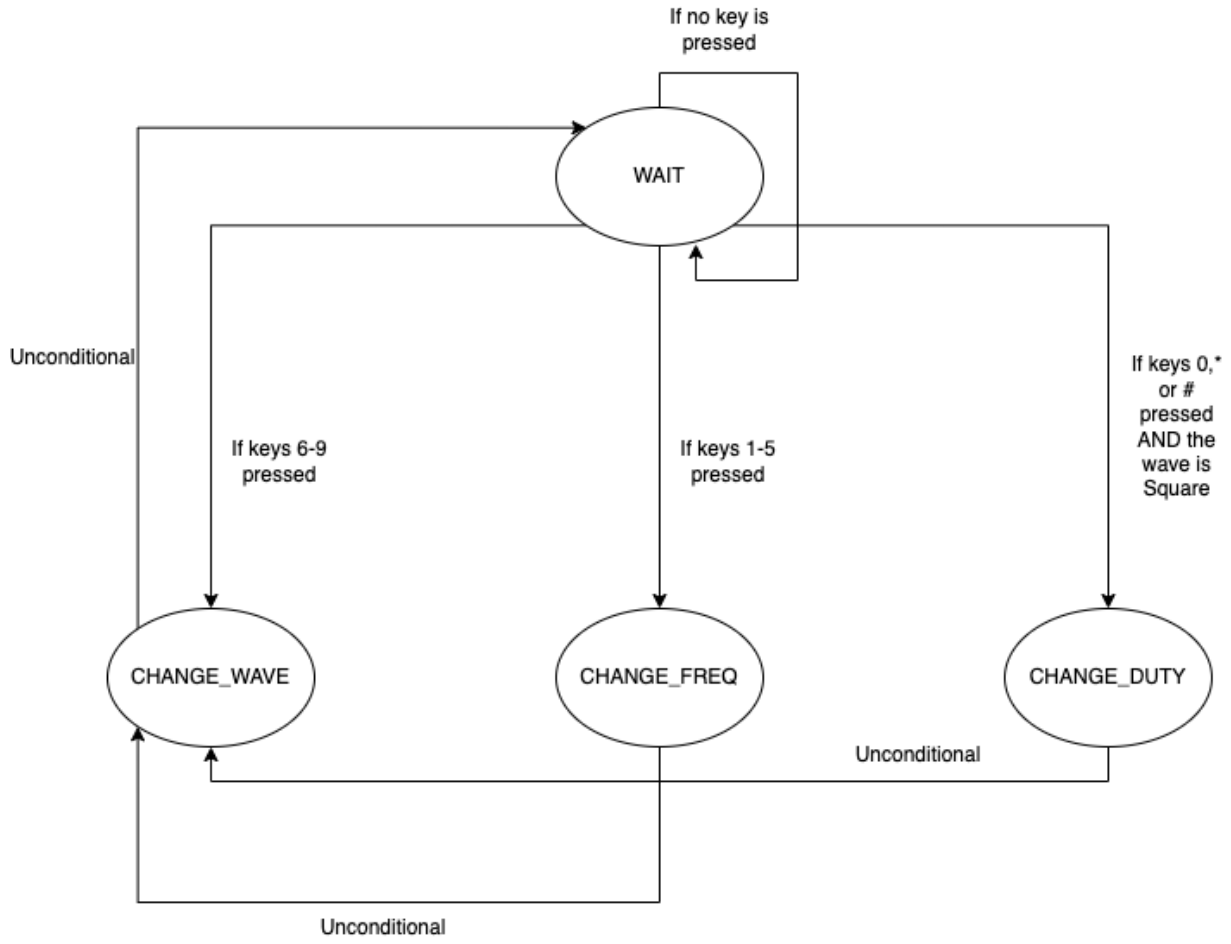


Function Generator Schematic
Includes 4922 DAC, Nucleo-L476RG Board, 4x4 Keypad, and Oscilloscope

Software Architecture

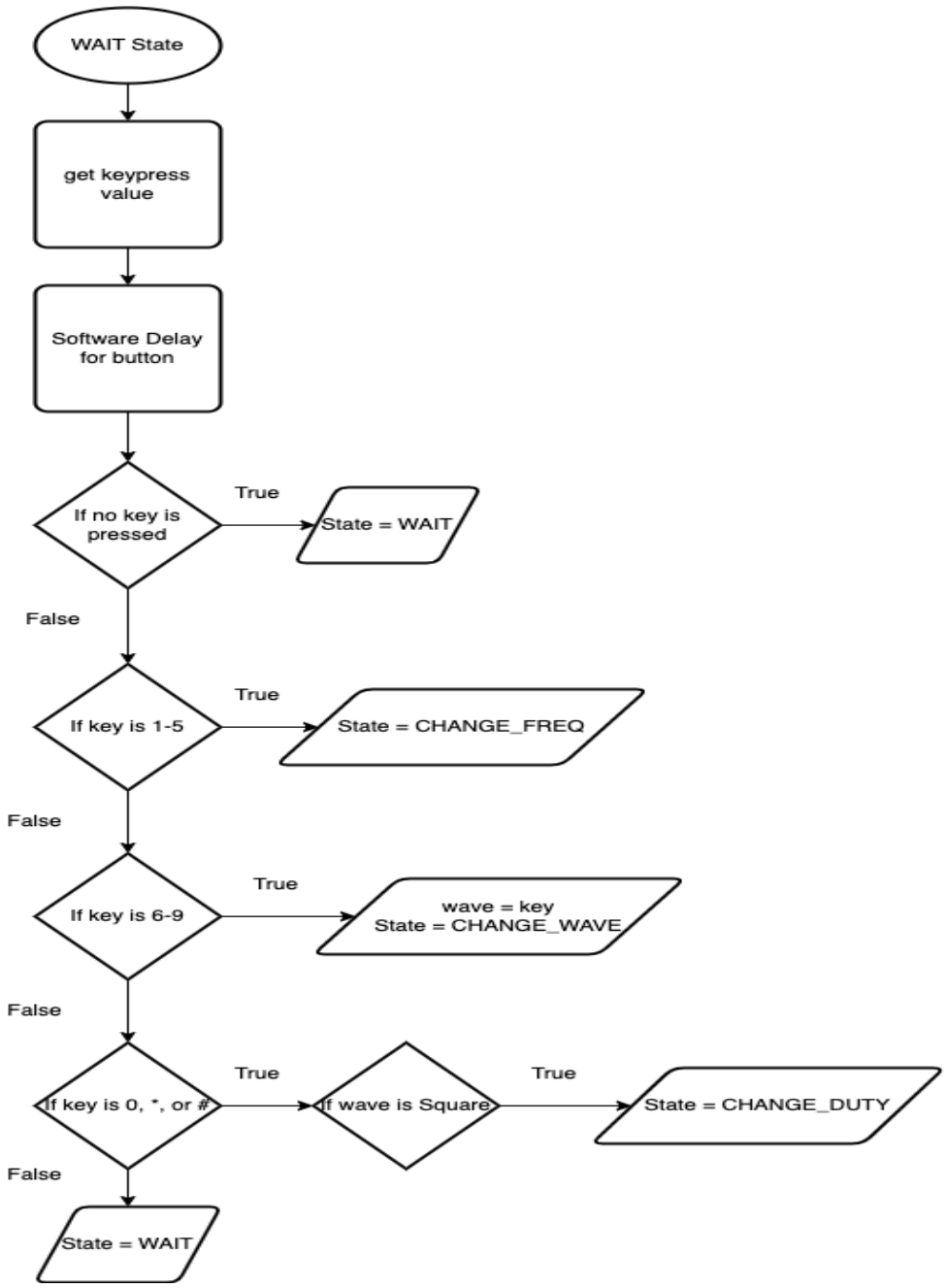
The main function initializes the peripherals, the timer, keypad, and the DAC. The program then sets the points array equal to a 100 Hz square wave with a 50% duty cycle. After these initialization steps, the FSM described next is used to guide any future desired changes to the waveform displayed.

ISR time = **23 µs** Samples per second = 1 / ISR Time = **42,000 samples per second** Samples per period 100 Hz = Samples per second / 100 Hz = **420 Samples** Samples per period 200 Hz = Samples per second / 200 Hz = **210 Samples** Samples per period 300 Hz = Samples per second / 300 Hz = **140 Samples** Samples per period 400 Hz = Samples per second / 400 Hz = **105 Samples** Samples per period 500 Hz = Samples per second / 500 Hz = **84 Samples**
Maximum Resolution Calculations



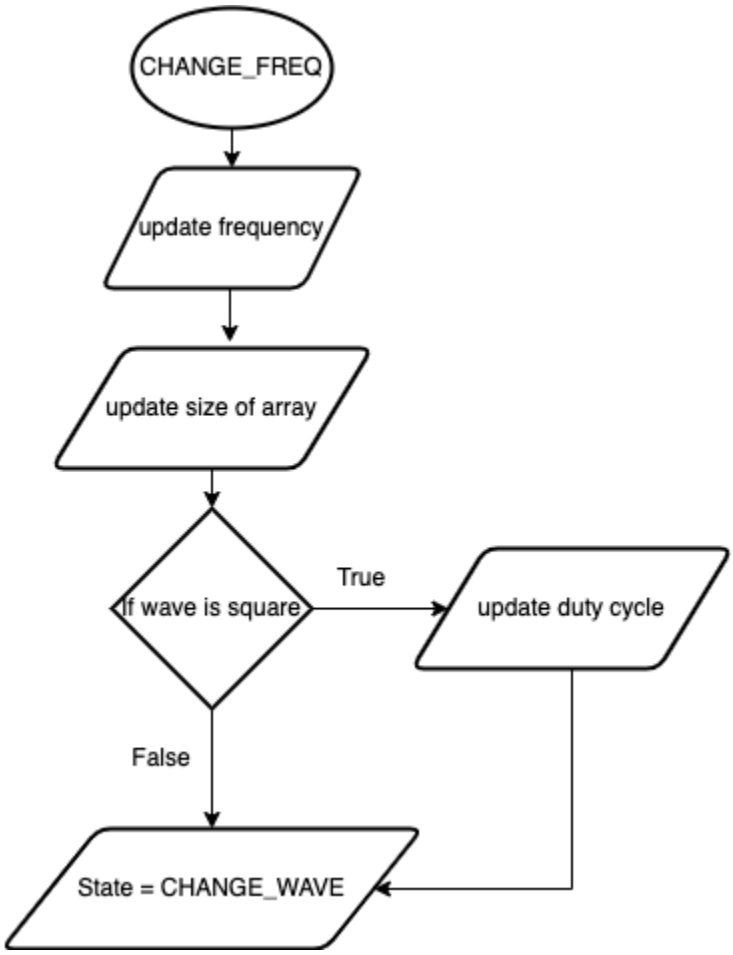
Flowchart of FSM in main.c

The bulk of this program is guided by the FSM, which consists of 4 different states: WAIT, CHANGE_WAVE, CHANGE_FREQ, and CHANGE_DUTY. The purpose of the WAIT state is to wait for a keypad input, then interpret that keypad press and set the next state accordingly. If no key is pressed, then the program remains in the WAIT state. The CHANGE_WAVE state is the state that updates any change made to the wave whether that is a new wave type, wave frequency, or square wave duty cycle. The next state is WAIT which happens unconditionally. The CHANGE_FREQ state updates the frequency and size of the points array, then unconditionally sets the next state to CHANGE_WAVE. The CHANGE_DUTY state only occurs if the wave is a square wave and updates the value for duty cycle, then unconditionally sets the next state to CHANGE_WAVE. The default state for the FSM is WAIT if anything goes wrong.



Flowchart of WAIT State

The WAIT state begins by calling keypad_getKey() in order to check if a key has been pressed. If no key has been pressed yet, the program stays in the WAIT, else the program checks to see which action the key corresponds to. For keys 1-5, the program continues to the CHANGE_FREQ state. For keys 6-9, the program updates the variable that stores the type of wave that is currently being displayed, then changes the state to CHANGE_WAVE. If keys 0, 14, or 15 were pressed, the program will change state to CHANGE_DUTY only if the current wave that is being displayed is a square wave. Finally, the default action is to remain in the WAIT state.

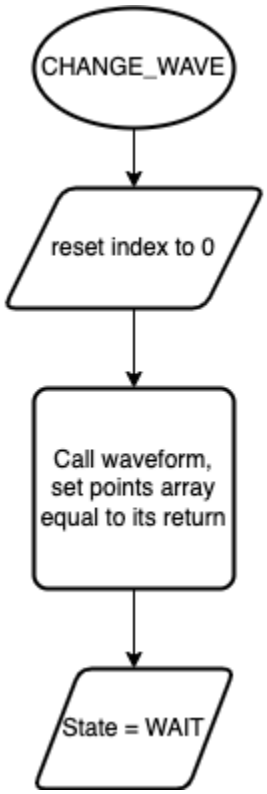


Flowchart of CHANGE_FREQ

The CHANGE_FREQ state begins by updating the variable in main that holds the frequency of the current displayed waveform and also updates the global variable that holds the size of the point array. If the wave is square, the variable in main that holds the duty cycle of the waveform is also updated to be half of the size of the points array. Finally, the state is then changed to CHANGE_WAVE.

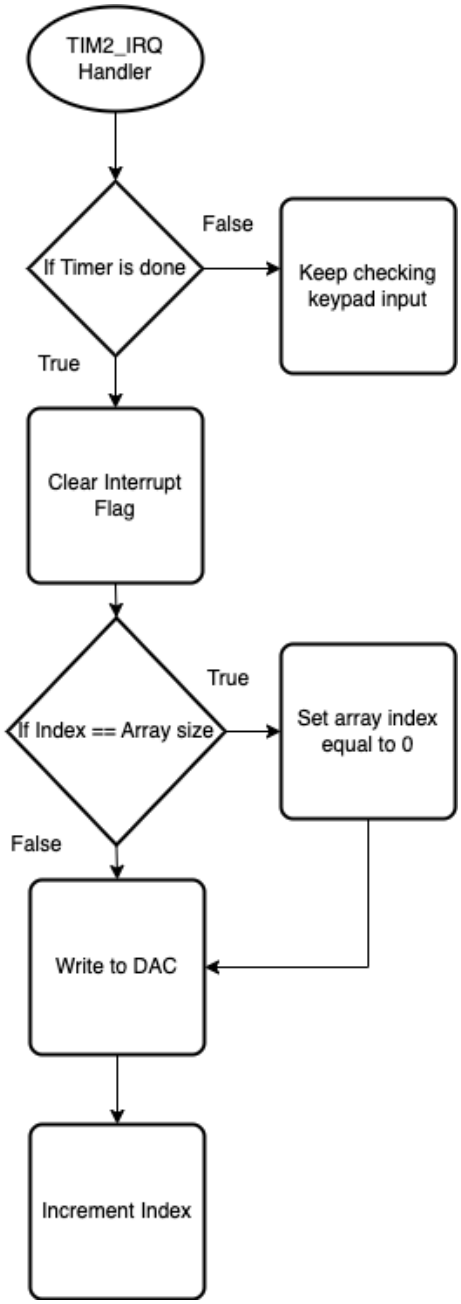
Flowchart of CHANGE_DUTY

The CHANGE_DUTY state increments or decrements the duty cycle of the square wave depending on the button pressed. The program checks if the key is 0, if so, then the duty is set to half the size of the points array, or a 50% duty cycle. The state after this action is CHANGE_WAVE. If the key pressed was an *, then the program checks if the duty cycle is greater than 90% of the size of the points array. If so, the next state is WAIT, otherwise, the program adds 10% of the value of the size array to the current value of duty, then changes state to CHANGE_WAVE. If the key that was pressed was a #, then the program will check if the duty cycle is less than 10% of the size of the points array. If so, the next state is WAIT, otherwise, the program subtracts 10% of the size array from the current value of duty, then changes the state to CHANGE_WAVE. Finally, if all else fails, the next state is WAIT.



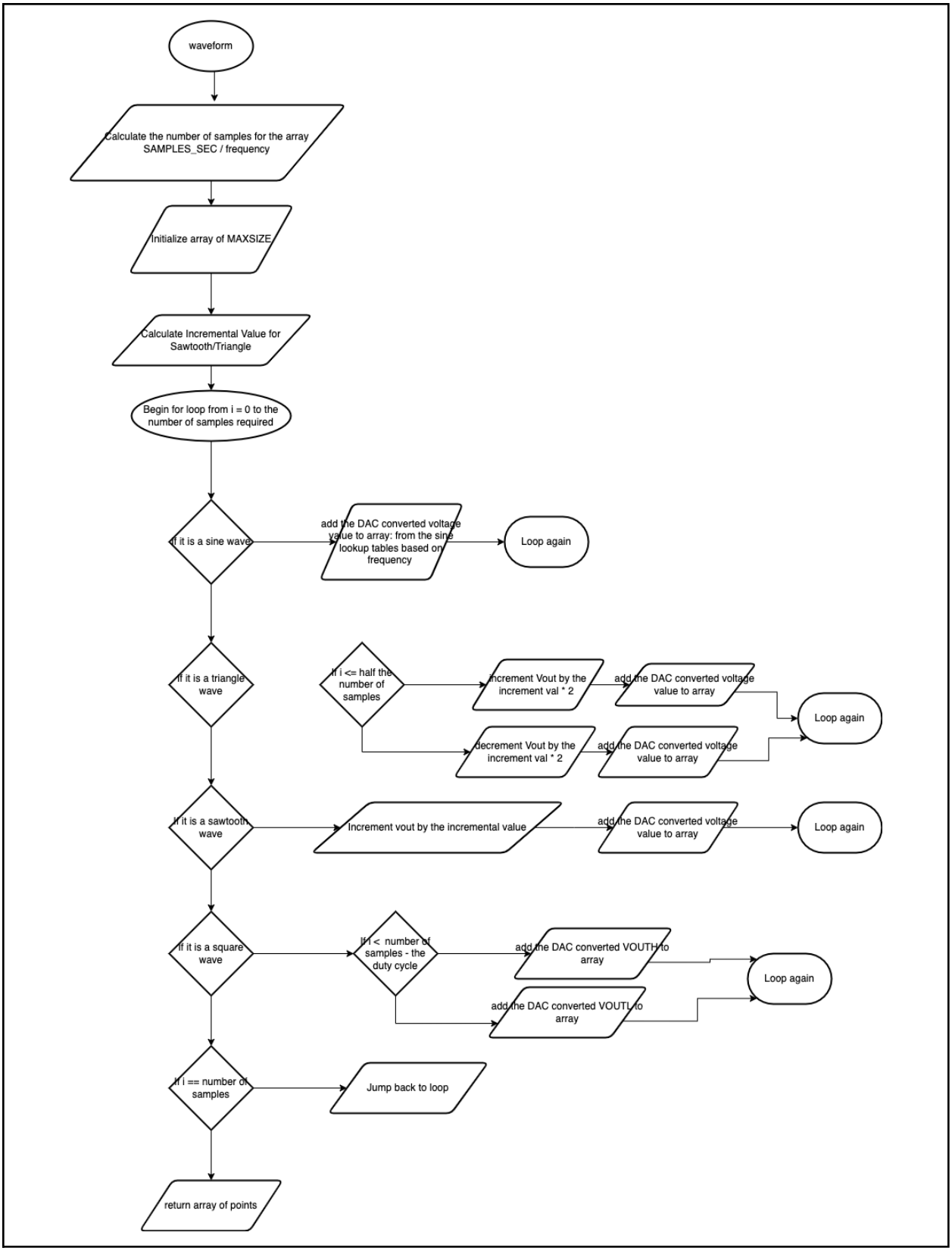
Flowchart of CHANGE_WAVE

The CHANGE_WAVE state is where the current displayed waveform is updated according to the keypress found in the previous states. This state starts off by resetting the index into the global array of points to 0 so the next waveform starts at the first index in the points array. Then, the points array is updated by calling waveform() with the wave type, frequency, and duty cycle as inputs. This function returns a pointer to the array, which will be used to access the points array. After these actions, the next state is WAIT where the program will await another keypress.



Flowchart of TIM2 ISR

The TIM2 ISR is where the program writes values from the points array to the DAC. The timer is set to interrupt about every 23 µs in order to write another voltage to the DAC. The interrupt handler starts by checking if the interrupt flag is high, if not, the handler is not run. If so, the interrupt flag is cleared. The handler then checks to see if the global index value is equal to the global size of the points array. If so, then it resets the global index to 0 so the next period of the waveform can be displayed. The handler then writes the corresponding point from the array to the DAC and increments the index value so the next point will be plotted the next time the timer interrupts.


waveform
Calculate the number of samples for the array SAMPLES_SEC / frequency
Initialize array of MAXSIZE
Calculate Incremental Value for Sawtooth/Triangle
Begin for loop from i = 0 to the number of samples required
If it is a sine wave
add the DAC converted voltage value to array: from the sine lookup tables based on frequency
Loop again
If it is a triangle wave
If i <= half the number of samples
increment Vout by the increment val * 2
add the DAC converted voltage value to array
decrement Vout by the increment val * 2
add the DAC converted voltage value to array
Loop again
If it is a sawtooth wave
Increment vout by the incremental value
add the DAC converted voltage value to array
Loop again
If it is a square wave
If i < number of samples - the duty cycle
add the DAC converted VOUTH to array
add the DAC converted VOUTL to array
Loop again
If i == number of samples
Jump back to loop
return array of points

Flowchart of waveform()

The waveform function is where the global array of points for a period of the waveform is calculated. It takes the desired waveforms type, frequency, and duty cycle as inputs. First, the function calculated the size of the array, which is the samples per second divided by the desired frequency. Then, an array of MAXSIZE = 420 is initialized to be all zeroes. The function then calculates the incremental value necessary for the triangle and sawtooth waveforms, which is VOUTH (3.0 V) / Size of the points array. The function then loops from 0 to the number of points in the array, calculates the point for that index in the array, then converts it to a DAC-readable value and adds it to the array. For the sine wave, a lookup table in sin.h is used to determine the point for a specific index. For the Triangle wave, a variable called Vout is used to store the current voltage value for that index. If the current i value in the for loop is less than half the number of samples required, Vout is incremented by the previously found increment value * 2. Else, Vout is decremented by the previously found increment value * 2. For the sawtooth waveform, Vout is used again, and is incremented by the previously found increment value for each value in the loop. Lastly, the square wave points are equal to VOUTH if the current i value in the for loop is less then the number of samples - the desired duty cycle. Otherwise, VOUTL is added to the points array. When the for loop is complete, the function returns a pointer to the array.

References

1. **4922 Microchip DAC Datasheet [Accessed April 30, 2023]**
2. **STM32-L47xxx Reference Manual [Accessed April 30, 2023]**
3. **STM32-L47xxx Datasheet [Accessed April 30, 2023]**

Appendix A: Main Source File

Main.c

```
/* USER CODE BEGIN Header */
/**
  ******************************************************************************
  * @file           : main.c
  * @brief          : Function Generator
  ******************************************************************************
  *Program interfaces DAC, Board, and keypad to display waveforms
  ******************************************************************************
  */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "KEYPAD_lib/keypad_lib.h"
#include "DAC_lib/DAC_lib.h"
#include "waveforms.h"
void SystemClock_Config(void);
//points holds the points to be plotted on waveform
static int32_t* points;
//current index into array
int16_t idx;
//size of the array
int16_t size;
int main(void)
{
  HAL_Init();
  SystemClock_Config();
  //configure keypad
  keypad_config();
  //initialize DAC
  DAC_init();
  //initial wave to display
  int16_t freq = 100;
  int8_t wave = SQUARE;
  int16_t duty;
  //index into points array
  idx = 0;
  //size of the points array (per period)
  size = SAMPLES_SEC / freq;
  //set to 50% duty cycle
  duty = size / 2;
  //set array to hold values of 100 HZ square wave
  points = waveform(freq, wave, duty);
  //initialize TIM2
  RCC->APB1ENR1 |= (RCC_APB1ENR1_TIM2EN);      //initialize RCC
  TIM2->DIER |= (TIM_DIER_UIE);                // enable interrupt on update event
  TIM2->SR &= ~(TIM_SR_UIF);                   // clear update interrupt flag
  TIM2->ARR = CLOCKFREQ/SAMPLES_SEC + 5;       //set ARR to Output rate
```

```
TIM2->CR1 |= (TIM_CR1_CEN);                    //start the timer
// enable interrupts in NVIC
NVIC->ISER[0] = (1 << (TIM2_IRQn & 0x1F));
__enable_irq();   // enable interrupts globally
//create state type variable
typedef enum{
       WAIT,
       CHANGE_DUTY,
       CHANGE_FREQ,
       CHANGE_WAVE
}state_type;
//initial state
state_type state = WAIT;
//holds current key press
int8_t key;
while (1)
{
     //FSM
  switch(state){
     //state to wait for key press
     case WAIT:
            //get a key
            key = keypad_getKey();
            //software delay for keypad
            for(int32_t i = 0; i<200000; i++);
            //if a key was pressed
            if (key != -1){
                  //if the key was 1,2,3,4,5 change frequency
                  if(key > 0 && key <= 5){
                        state = CHANGE_FREQ;
                  }
                  //change state to wave
                  else if(key > 5 && key <= 9){
                        wave = key;
                        state = CHANGE_WAVE;
                  }
                  //change state to duty
                  else if((key == 0) || (key == 14) || (key == 15)){
                        //only change duty if it is a square wave
                        if(wave == SQUARE)
                              state = CHANGE_DUTY;
                  }
            }
            //else stay in wait state
            else {
                  state = WAIT;
            }
            break;
     //state to change duty cycle of square wave
     case CHANGE_DUTY:
            //key 0  resets 50% duty cycle
            if(key == 0){
                  duty = size / 2;
                  state = CHANGE_WAVE;
```

```
            }
            //key * decreases duty cycle by 10%
            else if(key == 14){
                  //do not increase if it is at minimum of 10%
                  int16_t decr = size / 10;
                  if((duty + decr) <= (size - decr)){
                        //decrease duty cycle by 10%
                        duty += size / 10;
                        state = CHANGE_WAVE;
                  }
                  //if at minimum go back to wait
                  else
                        state = WAIT;
            }
            //key # increases duty cycle by 10%
            else if(key == 15){
                  //do not increase if it is at maximum of 90%
                  int16_t incr = size / 10;
                  if((duty - incr) >= (size - (size * 9 / 10))){
                        //increase duty cycle by 10%
                        duty -= size / 10;
                        state = CHANGE_WAVE;
                  }
                  //if at maximum go back to wait
                  else
                        state = WAIT;
            }
            //default
            else
                  state = WAIT;
            break;
      //state to change the frequency
      case CHANGE_FREQ:
            //keys are 1,2,3,4,5 representing each frequency
            freq = 100 * key;
            //update the size of the table
            size = SAMPLES_SEC / freq;
            //update the new 50% duty cycle if it is a square
            if(wave == SQUARE)
                  duty = size / 2;
            state = CHANGE_WAVE;
            break;
      case CHANGE_WAVE:
            //reset the index
            idx = 0;
            //generate a new table of points for every wave change
            points = waveform(freq, wave, duty);
            state = WAIT;
            break;
      default:
            state = WAIT;
   }//end switch
  }//end while
}//end main
```

```
//Timer 2 ISR: plot a new point every time the timer goes off
void TIM2_IRQHandler(void){
	//check status register for universal event flag
	if(TIM2->SR & TIM_SR_UIF){		//if the interrupt occurs
		TIM2->SR &= ~(TIM_SR_UIF);	//clear interrupt flag
		if(idx == size){		//if the end of table is reached
			idx = 0;		//reset the index
		}
		DAC_write(*(points + idx));	//write a point to the DAC
		idx++;				//increment index into lookup table
	}
}//end TIM2ISR
```

Appendix B: Waveforms Header and Source Files

waveforms.h

```
/*
 * waveforms.h
 *
 *  Created on: May 2, 2023
 *      Author: colecosta7
 */
#ifndef SRC_WAVEFORMS_H_
#define SRC_WAVEFORMS_H_
#include "main.h"
#include "math.h"
#include "sin.h"
#include <stdlib.h>
#define CLOCKFREQ 32000000
#define SAMPLES_SEC 42000
#define SINE 6
#define TRIANGLE 7
#define SAWTOOTH 8
#define SQUARE 9
#define VOUTH 3000
#define VOUTL 0
#define MAXSIZE 420
int32_t* waveform(int16_t freq, int8_t wavetype, int16_t duty);
#endif /* SRC_WAVEFORMS_H_ */
```

waveforms.c

```
/*
 * waveforms.c
 *
 *  Created on: May 2, 2023
 *      Author: colecosta7
 */
#include "waveforms.h"
int32_t* waveform(int16_t freq, int8_t wavetype, int16_t duty){
	//compute the number of samples for the given frequency
	int32_t numsamples = (SAMPLES_SEC) / (freq);
	//create an array of MAXSIZE in order to hold points
	static int32_t points[MAXSIZE] = {0};
	//the increment size for triangle and sawtooth
	int8_t inc = VOUTH/numsamples;
	//starting voltage
	int16_t vout = 0;
	//calculate a point for every sample, then convert it to a DAC value
	for(int16_t i = 0; i < numsamples; i++){
		//make the right calculation according to wavetype
		switch(wavetype){
			//uses a lookup table in order to plot sine points
			case SINE:
				if(freq == 100){
					points[i] = DAC_volt_conv(sineTable100[i]);
```

```
                        }
                        else if(freq == 200){
                                points[i] = DAC_volt_conv(sineTable200[i]);
                        }
                        else if(freq == 300){
                                points[i] = DAC_volt_conv(sineTable300[i]);
                        }
                        else if(freq == 400){
                                points[i] = DAC_volt_conv(sineTable400[i]);
                        }
                        else{
                                points[i] = DAC_volt_conv(sineTable500[i]);
                        }
                        break;
                //increases for half of the samples, decreases other half
                case TRIANGLE:
                        if(i <= numsamples/2){
                                vout += inc * 2;
                                points[i] = DAC_volt_conv(vout);
                        }
                        else{
                                vout -= inc * 2;
                                points[i] = DAC_volt_conv(vout);
                        }
                        break;
                //increases to 3.0V per period, then back to 0V
                case SAWTOOTH:
                        vout += inc;
                        points[i] = DAC_volt_conv(vout);
                        break;
                //high for the number of samples - the duty cycle, else low
                case SQUARE:
                        if(i < (numsamples - duty)) {
                                points[i] = DAC_volt_conv(VOUTH);
                        }
                        else {
                                points[i] = DAC_volt_conv(VOUTL);
                        }
                        break;
        }
    }
    //return pointer to array
    return points;
}
```

Appendix C: Keypad Header and Source Files

keypad_lib.h

```
/*
 * keypad_lib.h
 *
 *  Created on: Apr 12, 2023
 *      Author: colecosta7
 */
#ifndef SRC_KEYPAD_LIB_H_
#define SRC_KEYPAD_LIB_H_
#include "main.h"
#define ROW_PORT GPIOC
#define COL_PORT GPIOB
#define ROW_MASK (GPIO_IDR_ID4 | GPIO_IDR_ID5 | GPIO_IDR_ID6 | GPIO_IDR_ID7)
void keypad_config(void);
int8_t keypad_getKey(void);
#endif /* SRC_KEYPAD_LIB_H_ */
```

keypad_lib.c

```
/*
 * keypad_lib.c
 *
 *  Created on: Apr 24, 2023
 *      Author: colecosta7
 */
#include "keypad_lib.h"
//2d constant lookup table to find each number
const int8_t keypad_matrix[4][4] = {{1,2,3,10}, {4,5,6,11}, {7,8,9,12}, {14, 0, 15, 13}};
void keypad_config(void){
	//Enable Clock
	RCC -> AHB2ENR |= (RCC_AHB2ENR_GPIOBEN | RCC_AHB2ENR_GPIOCEN);
	//Enable Rows and Cols MODE Register
	//Sets ROW as INPUT type
	ROW_PORT -> MODER &= ~(GPIO_MODER_MODE4 | GPIO_MODER_MODE5 | GPIO_MODER_MODE6 |
			GPIO_MODER_MODE7);
	//Sets COL as OUTPUT type
	COL_PORT -> MODER &= ~(GPIO_MODER_MODE4 | GPIO_MODER_MODE5 | GPIO_MODER_MODE6 |
				GPIO_MODER_MODE7);
	COL_PORT -> MODER |= (GPIO_MODER_MODE4_0 | GPIO_MODER_MODE5_0 | GPIO_MODER_MODE6_0 |
			GPIO_MODER_MODE7_0);
	//Enable PUPDR as PULL DOWN for ROWS
	ROW_PORT -> PUPDR &= ~(GPIO_PUPDR_PUPD4 | GPIO_PUPDR_PUPD5 | GPIO_PUPDR_PUPD6 |
			GPIO_PUPDR_PUPD7);
	ROW_PORT -> PUPDR |= (GPIO_PUPDR_PUPD4_1 | GPIO_PUPDR_PUPD5_1 | GPIO_PUPDR_PUPD6_1 |
			GPIO_PUPDR_PUPD7_1);
	//Enable PUPDR as OFF for COLS
	COL_PORT -> PUPDR &= ~(GPIO_PUPDR_PUPD4 | GPIO_PUPDR_PUPD5 | GPIO_PUPDR_PUPD6 |
			GPIO_PUPDR_PUPD7);
	//Enable LOW Output Speed for COLS
	COL_PORT -> OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED4 | GPIO_OSPEEDR_OSPEED5 |
GPIO_OSPEEDR_OSPEED6 |
			GPIO_OSPEEDR_OSPEED7);
```

```
	//Enable OTYPE to Push Pull for COLS
	COL_PORT -> OTYPER &= ~(GPIO_OTYPER_OT4 | GPIO_OTYPER_OT5 | GPIO_OTYPER_OT6 |
GPIO_OTYPER_OT7);
	//Turn on all COLS
	COL_PORT -> BSRR = (GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7);
}
int8_t keypad_getKey(void){
	//if a row is detected high
	if(ROW_PORT -> IDR & ROW_MASK){
		int8_t row_num = 0;
		int8_t col_num = 0;
		//turn off all columns
		COL_PORT -> BRR = (GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7);
		//iterate through each column
		for(int col = GPIO_PIN_4; col <= GPIO_PIN_7; col <<= 1){
			//turn on individual column
			COL_PORT -> BSRR = col;
			//if a Row is high on this column
			if(ROW_PORT -> IDR & ROW_MASK){
				//iterate through each row
				for(int row = GPIO_PIN_4; row <= GPIO_PIN_7; row <<= 1){
					//if the row is high
					if(ROW_PORT -> IDR & row){
					//Turn on all the columns to prepare for the next key
press
						COL_PORT -> BSRR = (GPIO_PIN_4 | GPIO_PIN_5 |
GPIO_PIN_6 | GPIO_PIN_7);
						//return the right number from 2d lookup table
						return keypad_matrix[row_num][col_num];
					}
					//else increment
					row_num++;
				}
			}
			//else increment column # and turn off the column
			col_num++;
			COL_PORT -> BRR = (GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7);
		}
	}
	//if no key press, return -1
	COL_PORT -> BSRR = (GPIO_PIN_4 | GPIO_PIN_5 | GPIO_PIN_6 | GPIO_PIN_7);
	return -1;
}
```

Appendix D: DAC Header and Source Files

DAC_lib.h

```
/*
 * DAC_lib.h
 *
 *  Created on: Apr 23, 2023
 *      Author: colecosta7
 */
#ifndef SRC_DAC_LIB_H_
#define SRC_DAC_LIB_H_
#include "main.h"
//useful defines
#define AF5 5
//function declarations
void DAC_init(void);
void DAC_write(uint16_t val);
uint16_t DAC_volt_conv(uint16_t voltage);
#endif /* SRC_DAC_LIB_H_ */
```

DAC_lib.c

```
/*
 * DAC_lib.c
 *
 *  Created on: Apr 24, 2023
 *      Author: colecosta7
 */
#include "../DAC_lib/DAC_lib.h"
void DAC_init(void){
	//enable SPI clock, GPIO clock
	RCC -> APB2ENR |= (RCC_APB2ENR_SPI1EN);
	RCC -> AHB2ENR |= (RCC_AHB2ENR_GPIOAEN);
	//configure GPIO for MOSI PA7, SCLK PA5, NSS PA4
	GPIOA -> MODER &= ~(GPIO_MODER_MODE5 |
                                        GPIO_MODER_MODE7 |
                                        GPIO_MODER_MODE4);
	GPIOA -> MODER |= (GPIO_MODER_MODE5_1 |
                                        GPIO_MODER_MODE7_1 |
                                        GPIO_MODER_MODE4_1);  //AF for PA5, PA7, PA4
	GPIOA -> OTYPER &= ~(GPIO_OTYPER_OT5 |
                                        GPIO_OTYPER_OT7 |
                                        GPIO_OTYPER_OT4);            //push-pull for all
	GPIOA -> PUPDR &= ~(GPIO_PUPDR_PUPD5 |
                                        GPIO_PUPDR_PUPD7 |
                                        GPIO_PUPDR_PUPD4);           //none for all
	GPIOA -> OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED5 |
                                        GPIO_OSPEEDR_OSPEED7 |
                                        GPIO_OSPEEDR_OSPEED4);       //low speed for all
possibly change
	GPIOA -> AFR[0] |= (AF5 << GPIO_AFRL_AFSEL5_Pos |
                                    AF5 << GPIO_AFRL_AFSEL7_Pos |
                                    AF5 << GPIO_AFRL_AFSEL4_Pos);//AF5 Alternate Function
for SPI1
```

```
        //configure the SPI CR1 register
        SPI1 -> CR1 &= ~(SPI_CR1_BR);
        //SPI1 -> CR1 |= (SPI_CR1_BR_1);     //set baud rate to clk/2
        SPI1 -> CR1 &= ~(SPI_CR1_CPHA | SPI_CR1_CPOL);     //set MODE 0'0
        SPI1 -> CR1 &= ~(SPI_CR1_RXONLY);                  //set transmit only mode
        SPI1 -> CR1 &= ~(SPI_CR1_LSBFIRST);                //MSB transferred first
        SPI1 -> CR1 &= ~(SPI_CR1_SSM);                             //hardware controlled CS
        SPI1 -> CR1 |= (SPI_CR1_MSTR);                             //set master mode for
peripheral
        SPI1 -> CR2 |= (SPI_CR2_DS_0 | SPI_CR2_DS_1 |
                                    SPI_CR2_DS_3);                 //set data frame 12 bits
        SPI1 -> CR2 |= (SPI_CR2_SSOE);                                    //set SSOE
        SPI1 -> CR2 |= (SPI_CR2_NSSP);                                    //pulse mode
        SPI1 -> CR1 |= (SPI_CR1_SPE);                              //enable SPI1
}
void DAC_write(uint16_t voltage){
        //or the 12 bit voltage value with configuration byte
        uint16_t data = voltage | 0x3000;
        //wait for TX buffer to be empty
        while (!(SPI1->SR & SPI_SR_TXE)){};
        //set the data register
        SPI1 -> DR = data;
}
uint16_t DAC_volt_conv(uint16_t voltage){
        uint16_t value = (uint16_t)(4095 * voltage / 3300);
        return value;
}
```

Appendix E: Sine Wave Header File and Python Script

sin.h

```
/*
 * sin.h
 *
 *  Created on: May 3, 2023
 *      Author: colecosta7
 */
#ifndef SRC_SIN_H_
#define SRC_SIN_H_
static const uint16_t sineTable100[] = {
              1500,
              1522,1544,1567,1589,1612,1634,1656,1679,1701,1723,
              1745,1767,1789,1811,1833,1855,1877,1899,1920,1942,
              1963,1984,2005,2027,2048,2068,2089,2110,2130,2150,
              2170,2190,2210,2230,2250,2269,2288,2307,2326,2344,
              2363,2381,2399,2417,2435,2452,2469,2486,2503,2520,
              2536,2552,2568,2584,2599,2614,2629,2644,2658,2672,
              2686,2700,2713,2726,2739,2751,2764,2776,2787,2799,
              2810,2820,2831,2841,2851,2861,2870,2879,2887,2896,
              2904,2912,2919,2926,2933,2939,2945,2951,2957,2962,
              2967,2971,2975,2979,2983,2986,2989,2991,2993,2995,
              2997,2998,2999,2999,2999,2999,2999,2998,2997,2995,
              2993,2991,2989,2986,2983,2979,2975,2971,2967,2962,
              2957,2951,2945,2939,2933,2926,2919,2912,2904,2896,
              2887,2879,2870,2861,2851,2841,2831,2820,2810,2799,
              2787,2776,2764,2751,2739,2726,2713,2700,2686,2672,
              2658,2644,2629,2614,2599,2584,2568,2552,2536,2520,
              2503,2486,2469,2452,2435,2417,2399,2381,2363,2344,
              2326,2307,2288,2269,2249,2230,2210,2190,2170,2150,
              2130,2110,2089,2068,2048,2027,2005,1984,1963,1942,
              1920,1899,1877,1855,1833,1811,1789,1767,1745,1723,
              1701,1679,1656,1634,1612,1589,1567,1544,1522,1500,
              1478,1456,1433,1411,1388,1366,1344,1321,1299,1277,
              1255,1233,1211,1189,1167,1145,1123,1101,1080,1058,
              1037,1016,994,973,952,932,911,890,870,850,
              830,810,790,770,750,731,712,693,674,656,
              637,619,601,583,565,548,531,514,497,480,
              464,448,432,416,401,386,371,356,342,328,
              314,300,287,274,261,249,236,224,213,201,
              190,180,169,159,149,139,130,121,113,104,
              96,88,81,74,67,61,55,49,43,38,
              33,29,25,21,17,14,11,9,7,5,
              3,2,1,1,1,1,1,2,3,5,
              7,9,11,14,17,21,25,29,33,38,
              43,49,55,61,67,74,81,88,96,104,
              113,121,130,139,149,159,169,180,190,201,
```

```
            213,224,236,249,261,274,287,300,314,328,
            342,356,371,386,401,416,432,448,464,480,
            497,514,531,548,565,583,601,619,637,656,
            674,693,712,731,751,770,790,810,830,850,
            870,890,911,932,953,973,995,1016,1037,1058,
            1080,1101,1123,1145,1167,1189,1211,1233,1255,1277,
            1299,1321,1344,1366,1388,1411,1433,1456,1478
};
static const int16_t sineTable200[] = {
            1500,
            1544,1589,1634,1679,1723,1767,1811,1855,1899,1942,
            1984,2027,2068,2110,2150,2190,2230,2269,2307,2344,
            2381,2417,2452,2486,2520,2552,2584,2614,2644,2672,
            2700,2726,2751,2776,2799,2820,2841,2861,2879,2896,
            2912,2926,2939,2951,2962,2971,2979,2986,2991,2995,
            2998,2999,2999,2998,2995,2991,2986,2979,2971,2962,
            2951,2939,2926,2912,2896,2879,2861,2841,2820,2799,
            2776,2751,2726,2700,2672,2644,2614,2584,2552,2520,
            2486,2452,2417,2381,2344,2307,2269,2230,2190,2150,
            2110,2068,2027,1984,1942,1899,1855,1811,1767,1723,
            1679,1634,1589,1544,1500,1456,1411,1366,1321,1277,
            1233,1189,1145,1101,1058,1016,973,932,890,850,
            810,770,731,693,656,619,583,548,514,480,
            448,416,386,356,328,300,274,249,224,201,
            180,159,139,121,104,88,74,61,49,38,
            29,21,14,9,5,2,1,1,2,5,
            9,14,21,29,38,49,61,74,88,104,
            121,139,159,180,201,224,249,274,300,328,
            356,386,416,448,480,514,548,583,619,656,
            693,731,770,810,850,890,932,973,1016,1058,
            1101,1145,1189,1233,1277,1321,1366,1411,1456
};
static const int16_t sineTable300[] = {
            1500,
            1567,1634,1701,1767,1833,1899,1963,2027,2089,2150,
            2210,2269,2326,2381,2435,2486,2536,2584,2629,2672,
            2713,2751,2787,2820,2851,2879,2904,2926,2945,2962,
            2975,2986,2993,2998,2999,2998,2993,2986,2975,2962,
            2945,2926,2904,2879,2851,2820,2787,2751,2713,2672,
            2629,2584,2536,2486,2435,2381,2326,2269,2210,2150,
            2089,2027,1963,1899,1833,1767,1701,1634,1567,1500,
            1433,1366,1299,1233,1167,1101,1037,973,911,850,
            790,731,674,619,565,514,464,416,371,328,
            287,249,213,180,149,121,96,74,55,38,
            25,14,7,2,1,2,7,14,25,38,
            55,74,96,121,149,180,213,249,287,328,
            371,416,464,514,565,619,674,731,790,850,
            911,973,1037,1101,1167,1233,1299,1366,1433
};
static const int16_t sineTable400[] = {
            1500,
            1589,1679,1767,1855,1942,2027,2110,2190,2269,2344,
            2417,2486,2552,2614,2672,2726,2776,2820,2861,2896,
            2926,2951,2971,2986,2995,2999,2998,2991,2979,2962,
```

```
            2939,2912,2879,2841,2799,2751,2700,2644,2584,2520,
            2452,2381,2307,2230,2150,2068,1984,1899,1811,1723,
            1634,1544,1456,1366,1277,1189,1101,1016,932,850,
            770,693,619,548,480,416,356,300,249,201,
            159,121,88,61,38,21,9,2,1,5,
            14,29,49,74,104,139,180,224,274,328,
            386,448,514,583,656,731,810,890,973,1058,
            1145,1233,1321,1411
};
static const int16_t sineTable500[] = {
            1500,
            1612,1723,1833,1942,2048,2150,2250,2344,2435,2520,
            2599,2672,2739,2799,2851,2896,2933,2962,2983,2995,
            2999,2995,2983,2962,2933,2896,2851,2799,2739,2672,
            2599,2520,2435,2344,2249,2150,2048,1942,1833,1723,
            1612,1500,1388,1277,1167,1058,952,850,750,656,
            565,480,401,328,261,201,149,104,67,38,
            17,5,1,5,17,38,67,104,149,201,
            261,328,401,480,565,656,751,850,953,1058,
            1167,1277,1388
};
#endif /* SRC_SIN_H_ */
```

Sine Python Script

```
# Sine wave points generation

import math

def sin_points(numsamples):
   # Use a breakpoint in the code line below to debug your script.
   for i in range():
       angle = i / numsamples * 360            #find the angle in degrees
       sval = math.sin((angle * 3.1416) / 180) #find sin value in radians
       sval = sval * 1500                      #multiply the value by the peak
       sval = int(sval) + 1500                 #offset the value
       print(sval, end=",")                    #print each value and format for list
       if(i % 10 == 0):
           print("\n")


# Press the green button in the gutter to run the script.
if __name__ == '__main__':
   sin_points(420)
```