Function Generator

EE 329-01 Microprocessor-based Systems Design

Project #2, Spring 2021

Prepared by: Micah Jeffries

Prepared for: Professor Hummel

Date Submitted: May 12, 2021

I. What is the Function Generator?

The function generator utilizes SPI protocol to communicate with an external digital to analog converter (DAC) to generate various analog waveforms. The function generator can generate a saw tooth waveform, a triangle waveform, a square wave with a variable duty cycle, and a sinusoidal waveform. The frequency of the waveforms are also variable. The function generator also utilizes a keypad that can select the output waveform type, set the frequency of the waveform and set the duty cycle of the square wave.

II. System Specifications

Parameter	Value	Unit
Power Supply Voltage	5	V
Operating Voltage	3.3	V
Power Consumption	675	uW
Time Resolution	30,000	samples / sec
Bit Resolution	12	bits
Keypad Dimensions	69 x 76	mm
Clock Frequency	6	MHz
Operating Temperature	0 to 50	C
Communication Protocol	SPI	N/A
Waveform Frequencies	100 to 500	Hz
Square Wave Duty Cycle	10, 20, …, 90	N/A

Table 1: System Specifications

III. System Schematic

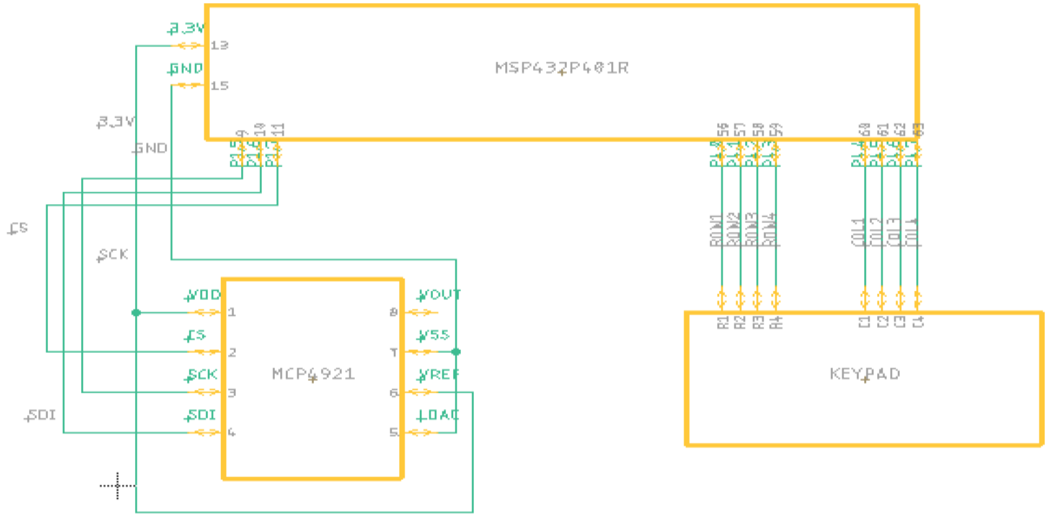
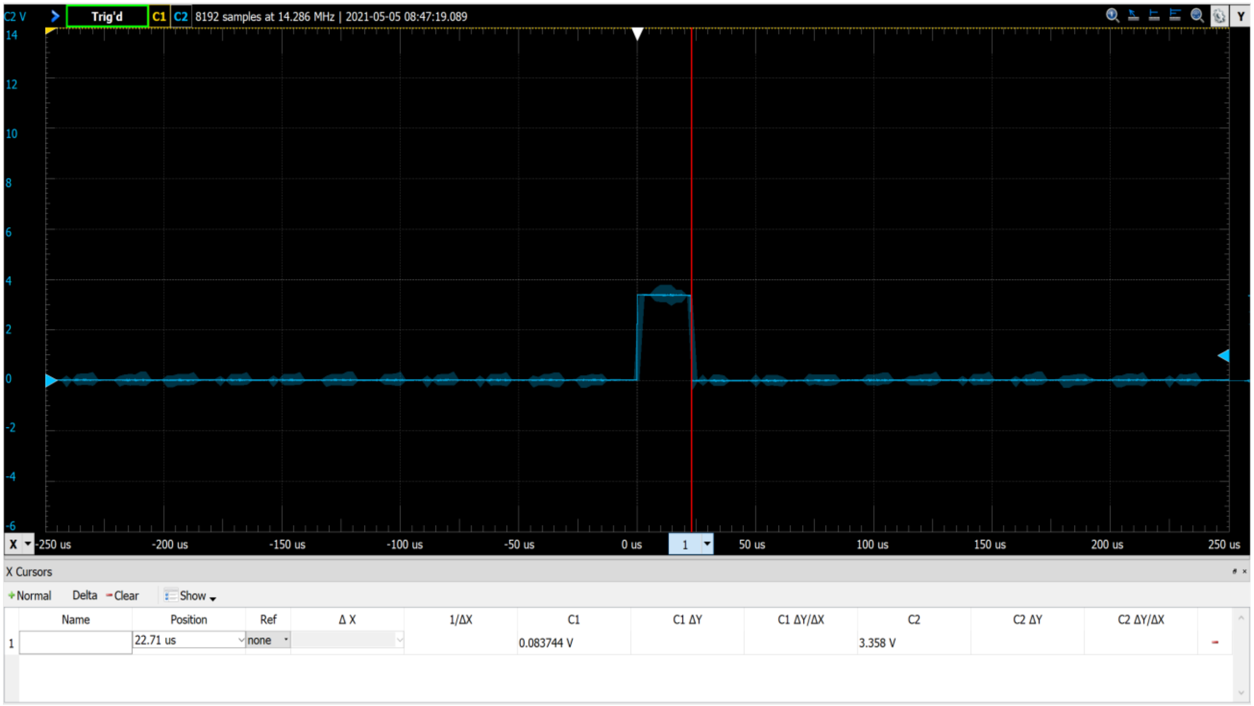


Figure 1: Schematic of Function Generator

IV. Software Architecture

A. Time Resolution

To ascertain the optimal time resolution for the function generator, the time required for sending one command from the microcontroller to the DAC needed to be determined. Timer A in the microcontroller is utilized to trigger interrupts in the microcontroller at precise timing intervals in order to send data to the DAC. Thus, the theoretical minimum amount of time for one command to the DAC can be determined by summing the DAC command and the ISR processing time. Figure 2 below shows an oscilloscope capture of the processing time for the DAC write command.



8192 samples at 14.286 MHz | 2021-05-05 08:47:19.089

	Name	Position	Ref	Δ X	1/ΔX	C1	C1 ΔY	C1 ΔY/ΔX	C2	C2 ΔY	C2 ΔY/ΔX
1		22.71 us	none			0.083744 V			3.358 V		

Figure 2: Oscilloscope Capture of DAC Processing Time

The overhead time due to the ISR was determined in assignment 6 to be approximately 18 clock cycles. Refer to equation 1 to see how the theoretical minimum processing time was calculated.

(1) $T_{min} = T_{DAC} + T_{ISR} = 22.71$ us + 18 clock cycles / 6 MHz = 25.71 us

This minimum time corresponds to a maximum time resolution of approximately 40,000 samples / sec. The requirements for the function generator specify that the output rate should be at least 75% of the maximum value. Therefore, the time resolution for the function generator was designed to have a time resolution of 30,000 samples / sec.

B. System Operation

When the function generator is initially powered up, it will display a 100 Hz square wave with a 50% duty cycle. The three keys #, *, and 0 are uniquely available to the user when the function generator is outputting a square wave.

- (#) increase the current duty cycle in increments of 10% up to a maximum of 90%
- (*) decrease the current duty cycle in increments of 10% down to a minimum of 10%
- (0) reset the duty cycle back to 50%.

The five keys 1, 2, 3, 4, and 5 are for changing the frequency of the waveform.

- (1) sets the current waveform to 100 Hz
- (2) sets the current waveform to 200 Hz
- (3) sets the current waveform to 300 Hz
- (4) sets the current waveform to 400 Hz
- (5) sets the current waveform to 500 Hz

The four keys 6, 7, 8, and 9 are for changing the output waveform.

- (6) changes the output waveform to a sinusoid
- (7) changes the output waveform to a triangle
- (8) changes the output waveform to a sawtooth
- (9) changes the output waveform to a square

Refer to figure 3 for the state diagram representation of the main function of the function generator. Refer to the other figures to see the flowcharts for the main program and its subroutines.

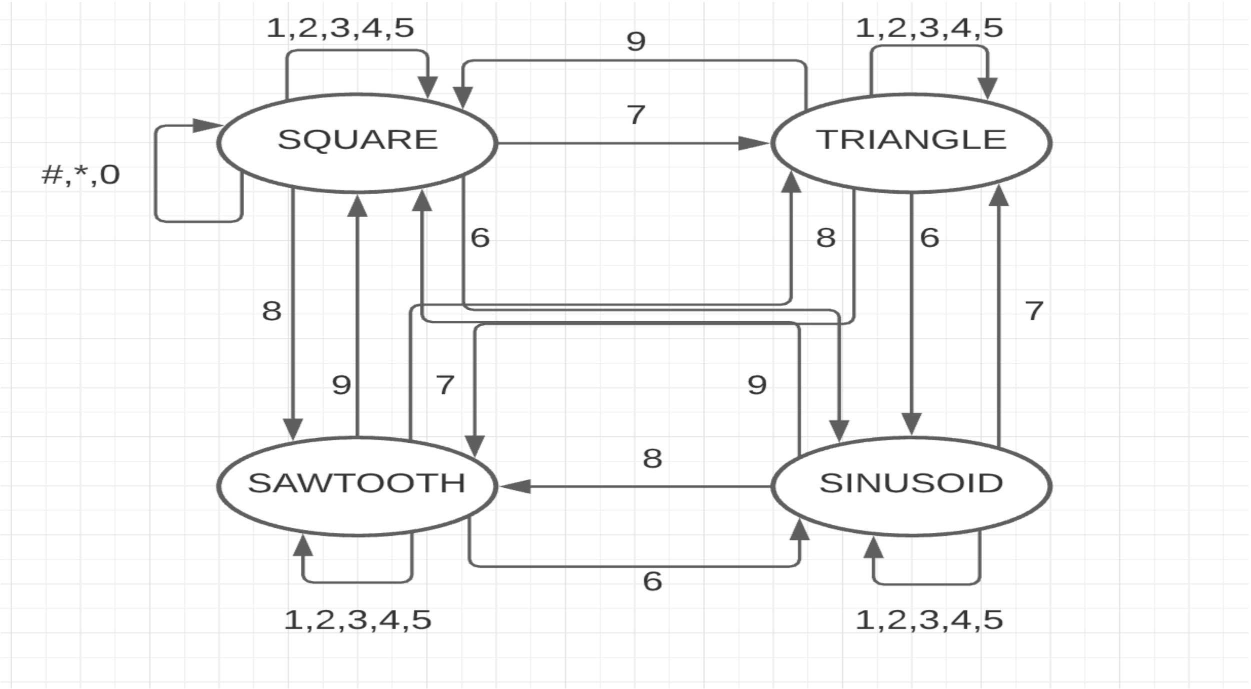


Figure 3: State Diagram of Main Function of Digital Lockbox

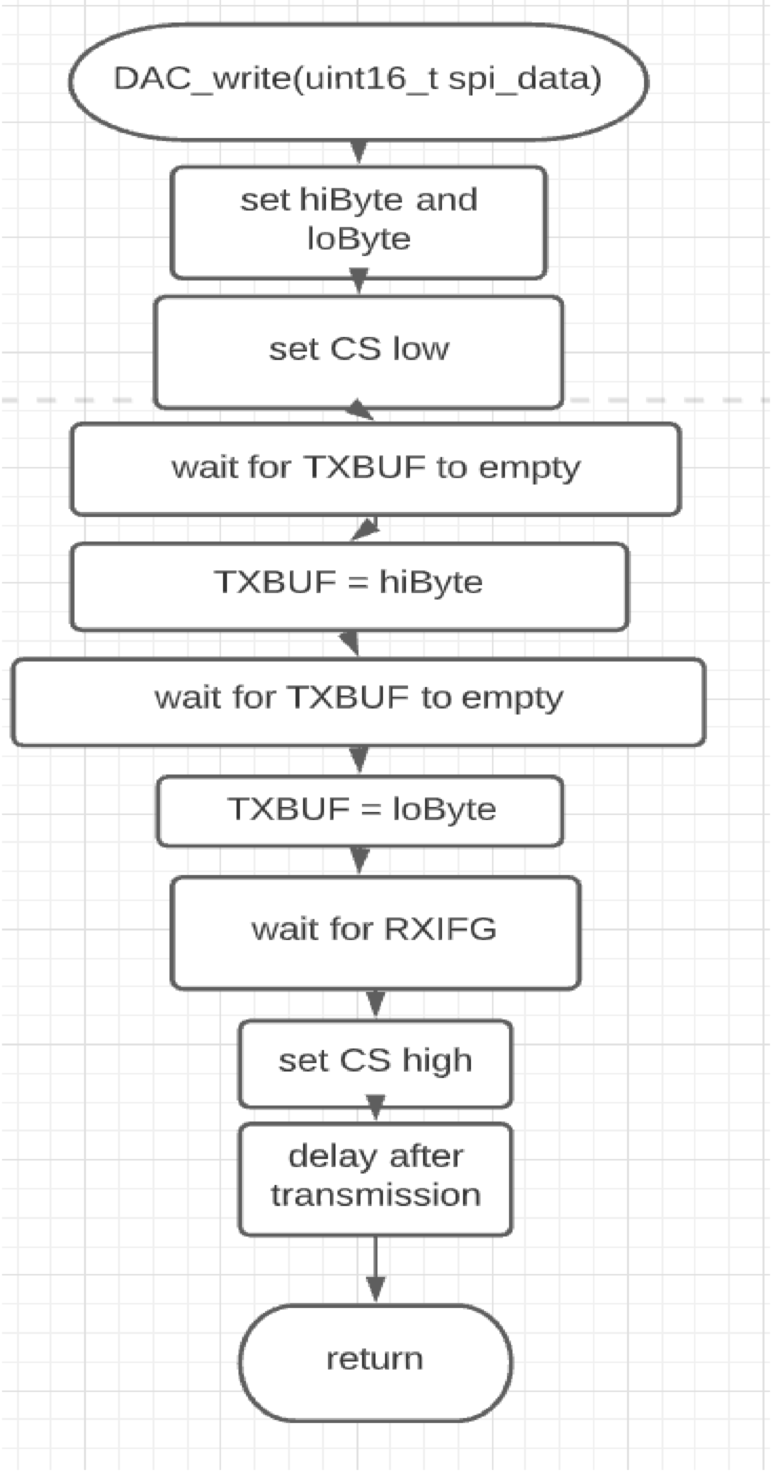


Figure 4: DAC Write Subroutine Flowchart

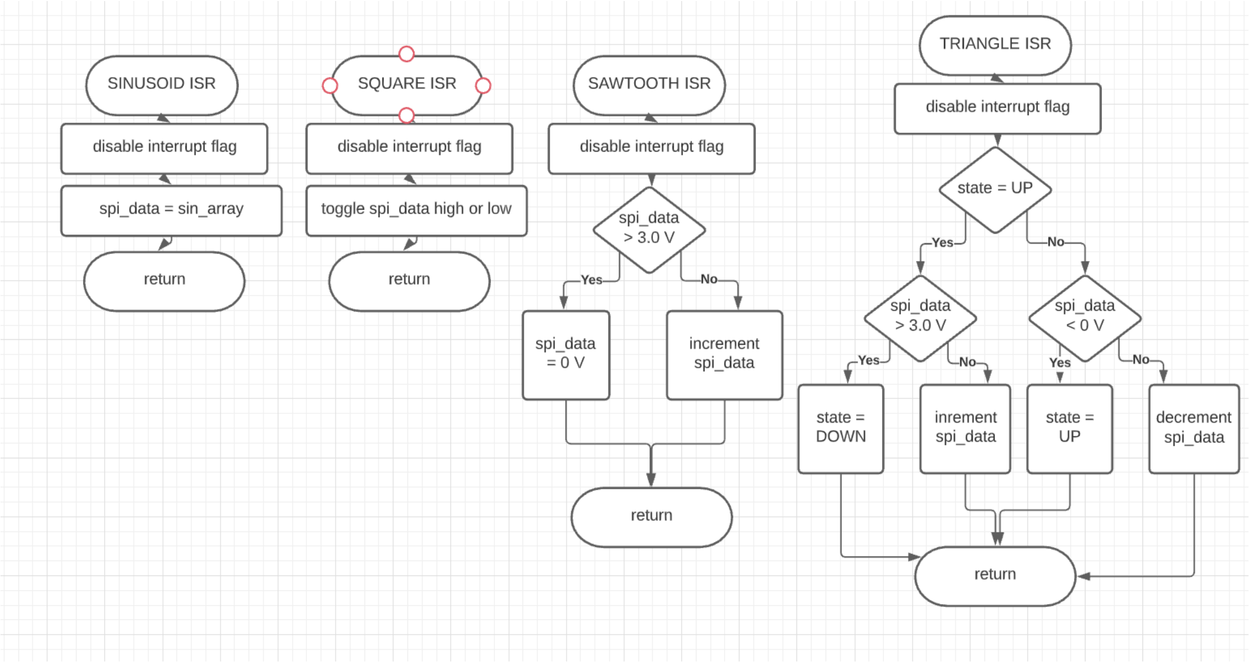


Figure 5: ISR subroutines of Function Generator

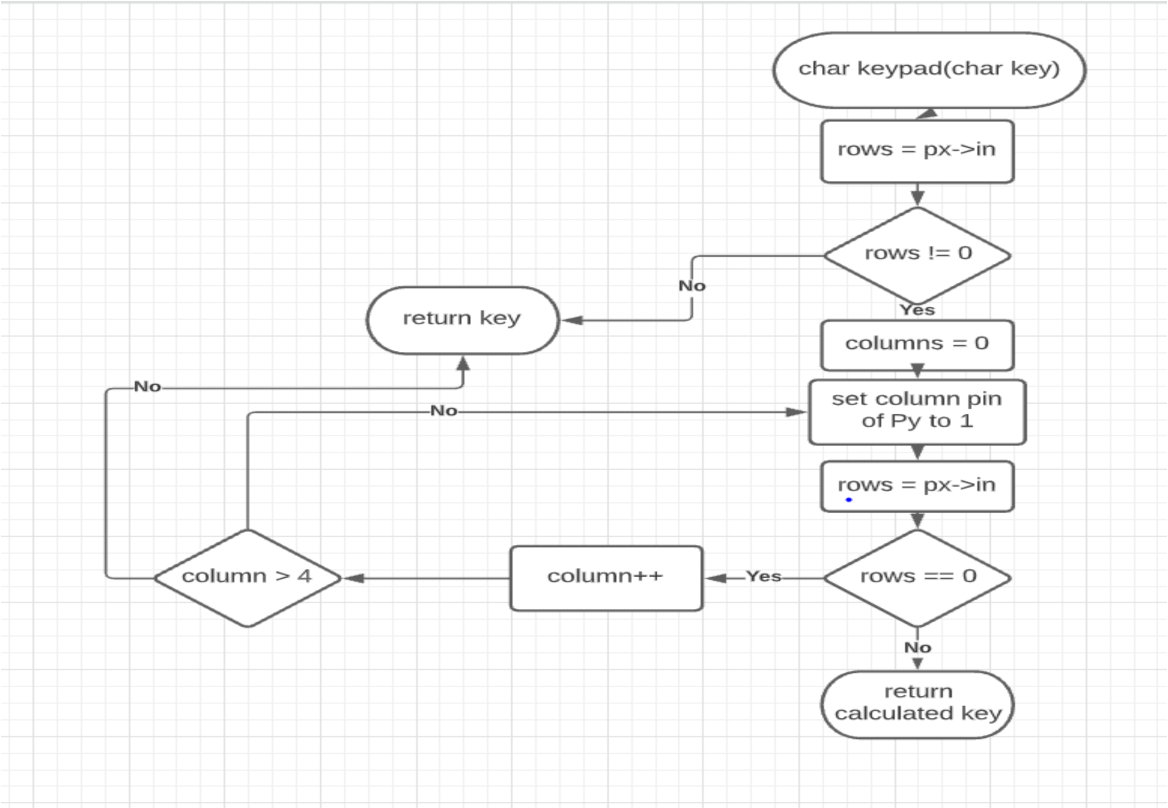


Figure 6: Keypad Subroutine Flowchart

V. Appendix

A. Code

main.c

```
#include "msp.h"
#include "DCO.h"
#include "keypad.h"
#include "DAC.h"

uint16_t spi_data = 0x6FFF, triangle_increment = T_100, sawtooth_increment = S_100,
sin_increment = 0;
uint16_t sin100[150] = {3800, 3960, 4120, 4279, 4438, 4595, 4751, 4905, 5058, 5208,
5355, 5500, 5642, 5780, 5915, 6047, 6174, 6297, 6415, 6529, 6638, 6742, 6841, 6934,
7022, 7104, 7180, 7251, 7315, 7372, 7424, 7469, 7507, 7539, 7564, 7583, 7595, 7600,
7598, 7590, 7574, 7553, 7524, 7489, 7447, 7399, 7344, 7283, 7216, 7143, 7064, 6979,
6888, 6792, 6691, 6584, 6473, 6357, 6236, 6111, 5981, 5848, 5712, 5571, 5428, 5282,
5133, 4982, 4829, 4673, 4517, 4359, 4200, 4040, 3880, 3720, 3560, 3400, 3241, 3083,
2927, 2771, 2618, 2467, 2318, 2172, 2029, 1888, 1752, 1619, 1489, 1364, 1243, 1127,
1016, 909, 808, 712, 621, 536, 457, 384, 317, 256, 201, 153, 111, 76, 47, 26, 10, 2,
0, 5, 17, 36, 61, 93, 131, 176, 228, 285, 349, 420, 496, 578, 666, 759, 858, 962,
1071, 1185, 1303, 1426, 1553, 1685, 1820, 1958, 2100, 2245, 2392, 2542, 2695, 2849,
3005, 3162, 3321, 3480, 3640, 3800};
uint16_t sin200[150] = {3800, 4120, 4438, 4751, 5058, 5355, 5642, 5915, 6174, 6415,
6638, 6841, 7022, 7180, 7315, 7424, 7507, 7564, 7595, 7598, 7574, 7524, 7447, 7344,
7216, 7064, 6888, 6691, 6473, 6236, 5981, 5712, 5428, 5133, 4829, 4517, 4200, 3880,
3560, 3241, 2927, 2618, 2318, 2029, 1752, 1489, 1243, 1016, 808, 621, 457, 317, 201,
111, 47, 10, 0, 17, 61, 131, 228, 349, 496, 666, 858, 1071, 1303, 1553, 1820, 2100,
2392, 2695, 3005, 3321, 3640, 3960, 4279, 4595, 4905, 5208, 5500, 5780, 6047, 6297,
6529, 6742, 6934, 7104, 7251, 7372, 7469, 7539, 7583, 7600, 7590, 7553, 7489, 7399,
7283, 7143, 6979, 6792, 6584, 6357, 6111, 5848, 5571, 5282, 4982, 4673, 4359, 4040,
3720, 3400, 3083, 2771, 2467, 2172, 1888, 1619, 1364, 1127, 909, 712, 536, 384, 256,
153, 76, 26, 2, 5, 36, 93, 176, 285, 420, 578, 759, 962, 1185, 1426, 1685, 1958,
2245, 2542, 2849, 3162, 3480, 3800};
uint16_t sin300[150] = {3800, 4279, 4751, 5208, 5642, 6047, 6415, 6742, 7022, 7251,
7424, 7539, 7595, 7590, 7524, 7399, 7216, 6979, 6691, 6357, 5981, 5571, 5133, 4673,
4200, 3720, 3241, 2771, 2318, 1888, 1489, 1127, 808, 536, 317, 153, 47, 2, 17, 93,
228, 420, 666, 962, 1303, 1685, 2100, 2542, 3005, 3480, 3960, 4438, 4905, 5355, 5780,
6174, 6529, 6841, 7104, 7315, 7469, 7564, 7600, 7574, 7489, 7344, 7143, 6888, 6584,
6236, 5848, 5428, 4982, 4517, 4040, 3560, 3083, 2618, 2172, 1752, 1364, 1016, 712,
457, 256, 111, 26, 0, 36, 131, 285, 496, 759, 1071, 1426, 1820, 2245, 2695, 3162,
3640, 4120, 4595, 5058, 5500, 5915, 6297, 6638, 6934, 7180, 7372, 7507, 7583, 7598,
7553, 7447, 7283, 7064, 6792, 6473, 6111, 5712, 5282, 4829, 4359, 3880, 3400, 2927,
2467, 2029, 1619, 1243, 909, 621, 384, 201, 76, 10, 5, 61, 176, 349, 578, 858, 1185,
1553, 1958, 2392, 2849, 3321, 3800};
uint16_t sin400[150] = {3800, 4438, 5058, 5642, 6174, 6638, 7022, 7315, 7507, 7595,
7574, 7447, 7216, 6888, 6473, 5981, 5428, 4829, 4200, 3560, 2927, 2318, 1752, 1243,
808, 457, 201, 47, 0, 61, 228, 496, 858, 1303, 1820, 2392, 3005, 3640, 4279, 4905,
5500, 6047, 6529, 6934, 7251, 7469, 7583, 7590, 7489, 7283, 6979, 6584, 6111, 5571,
4982, 4359, 3720, 3083, 2467, 1888, 1364, 909, 536, 256, 76, 2, 36, 176, 420, 759,
1185, 1685, 2245, 2849, 3480, 4120, 4751, 5355, 5915, 6415, 6841, 7180, 7424, 7564,
7598, 7524, 7344, 7064, 6691, 6236, 5712, 5133, 4517, 3880, 3241, 2618, 2029, 1489,
1016, 621, 317, 111, 10, 17, 131, 349, 666, 1071, 1553, 2100, 2695, 3321, 3960, 4595,
5208, 5780, 6297, 6742, 7104, 7372, 7539, 7600, 7553, 7399, 7143, 6792, 6357, 5848,
5282, 4673, 4040, 3400, 2771, 2172, 1619, 1127, 712, 384, 153, 26, 5, 93, 285, 578,
962, 1426, 1958, 2542, 3162, 3800};
uint16_t sin500[150] = {3800, 4595, 5355, 6047, 6638, 7104, 7424, 7583, 7574, 7399,
7064, 6584, 5981, 5282, 4517, 3720, 2927, 2172, 1489, 909, 457, 153, 10, 36, 228,
578, 1071, 1685, 2392, 3162, 3960, 4751, 5500, 6174, 6742, 7180, 7469, 7595, 7553,
7344, 6979, 6473, 5848, 5133, 4359, 3560, 2771, 2029, 1364, 808, 384, 111, 2, 61,
285, 666, 1185, 1820, 2542, 3321, 4120, 4905, 5642, 6297, 6841, 7251, 7507, 7600,
7524, 7283, 6888, 6357, 5712, 4982, 4200, 3400, 2618, 1888, 1243, 712, 317, 76, 0,
93, 349, 759, 1303, 1958, 2695, 3480, 4279, 5058, 5780, 6415, 6934, 7315, 7539, 7598,
7489, 7216, 6792, 6236, 5571, 4829, 4040, 3241, 2467, 1752, 1127, 621, 256, 47, 5,
131, 420, 858, 1426, 2100, 2849, 3640, 4438, 5208, 5915, 6529, 7022, 7372, 7564,
7590, 7447, 7143, 6691, 6111, 5428, 4673, 3880, 3083, 2318, 1619, 1016, 536, 201, 26,
17, 176, 496, 962, 1553, 2245, 3005, 3800};

uint16_t *sin_array = sin100;

typedef enum {                                  // define states of the triangle ISR
    UP,
    DOWN
} TRAINGLE_STATE;

TRAINGLE_STATE t_state = UP;

void main(void)
{
    double duty_cycle = 0.5;
    uint8_t key = 0;

    typedef enum {                                  // define states of the main
function
        SQUARE,
        SINUSOID,
        SAWTOOTH,
        TRIANGLE
    } STATE_TYPE;

    STATE_TYPE state = SQUARE;                      // initial state is the square
waveform

      WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;          // stop watchdog timer

      set_DCO(FREQ_6_MHZ);                             // set the MCLK to 6 MHz

      DAC_init();                                      // initialize the eUSCI
peripheral to communicate with the DAC
      keypad_init();                                   // initialize the keypad

      __enable_irq();                                  // global interrupt enable

      NVIC->ISER[0] = (1 << TA0_0_IRQn);              // enable CCTL0.CCIFG

      NVIC->ISER[0] = (1 << TA1_0_IRQn);              // enable square ISR
      NVIC->ISER[0] = (1 << TA1_N_IRQn);

      NVIC->ISER[0] = (1 << TA2_0_IRQn);              // enable triangle ISR
```

```
       NVIC->ISER[0] = (1 << TA2_N_IRQn);              // enable sawtooth ISR

       NVIC->ISER[0] = (1 << TA3_0_IRQn);              // enable sinusoid ISR

       TA0CCTL0 |= (TIMER_A_CCTLN_CCIE);               // enable CCIFG interrupt
       TA0CCTL0 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag

       TA1CCTL0 |= (TIMER_A_CCTLN_CCIE);               // enable CCIFG interrupt
       TA1CCTL0 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag

       TA1CCTL1 |= (TIMER_A_CCTLN_CCIE);               // enable CCIFG interrupt
       TA1CCTL1 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag

       TA2CCTL0 |= (TIMER_A_CCTLN_CCIE);               // enable CCIFG interrupt
       TA2CCTL0 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag

       TA2CCTL1 |= (TIMER_A_CCTLN_CCIE);               // enable CCIFG interrupt
       TA2CCTL1 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag

       TA3CCTL0 |= (TIMER_A_CCTLN_CCIE);               // enable CCIFG interrupt
       TA3CCTL0 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag

    TA0CCR0 = PERIOD;                               // set CCR0

    TA1CCR0 = PERIOD_100;                           // set square ISR registers
    TA1CCR1 = duty_cycle*PERIOD_100;

    TA2CCR0 = PERIOD * 2;                           // set triangle ISR register
    TA2CCR1 = PERIOD * 2;                           // set sawtooth ISR register
    TA3CCR0 = PERIOD * 2;                           // set sinusoid ISR register

       TIMER_A0->CTL = (TIMER_A_CTL_SSEL__SMCLK        // set timerA0 clock source to
SMCLK
                       | TIMER_A_CTL_MC__UP);          // set timerA0 counting mode
to up

       TIMER_A1->CTL = (TIMER_A_CTL_SSEL__SMCLK        // set timerA1 clock source to
SMCLK
                       | TIMER_A_CTL_MC__UP);          // set timerA1 counting mode
to up

       TIMER_A2->CTL = (TIMER_A_CTL_SSEL__SMCLK        // set timerA2 clock source to
SMCLK
                       | TIMER_A_CTL_MC__UP);          // set timerA2 counting mode
to up

       TIMER_A3->CTL = (TIMER_A_CTL_SSEL__SMCLK        // set timerA3 clock source to
SMCLK
                       | TIMER_A_CTL_MC__UP);          // set timerA3 counting mode
to up

       while (1) {
        switch(state) {
            case SQUARE:                            // square waveform
```

```
                TA1CCTL0 |= (TIMER_A_CCTLN_CCIE);   // enable and disable interrupt
flags
                TA1CCTL1 |= (TIMER_A_CCTLN_CCIE);
                TA2CCTL0 &= ~(TIMER_A_CCTLN_CCIE);
                TA2CCTL1 &= ~(TIMER_A_CCTLN_CCIE);
                TA3CCTL0 &= ~(TIMER_A_CCTLN_CCIE);

                key = keypad();                     // get user input
                __delay_us(200000);                 // software debounce

                switch(key) {
                    case 1:                         // set square wave to 100 Hz
                        TA1CCR0 = PERIOD_100;
                        TA1CCR1 = duty_cycle * PERIOD_100;
                        break;
                    case 2:                         // set square wave to 200 Hz
                        TA1CCR0 = PERIOD_200;
                        TA1CCR1 = duty_cycle * PERIOD_200;
                        break;
                    case 3:                         // set square wave to 300 Hz
                        TA1CCR0 = PERIOD_300;
                        TA1CCR1 = duty_cycle * PERIOD_300;
                        break;
                    case 4:                         // set square wave to 400 Hz
                        TA1CCR0 = PERIOD_400;
                        TA1CCR1 = duty_cycle * PERIOD_400;
                        break;
                    case 5:                         // set square wave to 500 Hz
                        TA1CCR0 = PERIOD_500;
                        TA1CCR1 = duty_cycle * PERIOD_500;
                        break;
                    case 6:
                        state = SINUSOID;
                        break;
                    case 7:
                        state = TRIANGLE;
                        break;
                    case 8:
                        state = SAWTOOTH;
                        break;
                    case 9:
                        state = SQUARE;
                        break;
                    case 0:                         // change duty cycle to 50%
                        duty_cycle = 0.5;
                        TA1CCR1 = duty_cycle * TA1CCR0;
                        break;
                    case STAR:                      // decrement duty cycle
                        if (duty_cycle > 0.15)
                            duty_cycle -= 0.1;
                        TA1CCR1 = duty_cycle * TA1CCR0;
                        break;
                    case POUND:                     // increment duty cycle
                        if (duty_cycle < 0.85)
                            duty_cycle += 0.1;
                        TA1CCR1 = duty_cycle * TA1CCR0;
                        break;
                    default:
                        state = SQUARE;
                }

                break;

            case SINUSOID:                          // sinusoidal waveform

                TA1CCTL0 &= ~(TIMER_A_CCTLN_CCIE);  // enable and disable interrupt
flags
                TA1CCTL1 &= ~(TIMER_A_CCTLN_CCIE);
                TA2CCTL0 &= ~(TIMER_A_CCTLN_CCIE);
                TA2CCTL1 &= ~(TIMER_A_CCTLN_CCIE);
                TA3CCTL0 |= (TIMER_A_CCTLN_CCIE);

                sin_increment = 0;

                key = keypad();                     // get user input
                __delay_us(200000);                 // software debounce

                switch(key) {
                    case 1:                         // set sine wave to 100 Hz
                        sin_array = sin100;
                        break;
                    case 2:                         // set sine wave to 200 Hz
                        sin_array = sin200;
                        break;
                    case 3:                         // set sine wave to 300 Hz
                        sin_array = sin300;
                        break;
                    case 4:                         // set sine wave to 400 Hz
                        sin_array = sin400;
                        break;
                    case 5:                         // set sine wave to 500 Hz
                        sin_array = sin500;
                        break;
                    case 6:
                        state = SINUSOID;
                        break;
                    case 7:
                        state = TRIANGLE;
                        break;
                    case 8:
                        state = SAWTOOTH;
                        break;
                    case 9:
                        state = SQUARE;
                        break;
                    default:
                        state = SAWTOOTH;
                }

                break;
            case SAWTOOTH:                          // sawtooth waveform

                TA1CCTL0 &= ~(TIMER_A_CCTLN_CCIE);  // enable and disable interrupt
flags

                TA1CCTL1 &= ~(TIMER_A_CCTLN_CCIE);
                TA2CCTL0 &= ~(TIMER_A_CCTLN_CCIE);
                TA2CCTL1 |= (TIMER_A_CCTLN_CCIE);
                TA3CCTL0 &= ~(TIMER_A_CCTLN_CCIE);

                spi_data = 0x6000;

                key = keypad();                     // get user input
                __delay_us(200000);                 // software debounce

                switch(key) {
                    case 1:                         // set sawtooth wave to 100 Hz
                        sawtooth_increment = S_100;
                        break;
                    case 2:                         // set sawtooth wave to 200 Hz
                        sawtooth_increment = S_200;
                        break;
                    case 3:                         // set sawtooth wave to 300 Hz
                        sawtooth_increment = S_300;
                        break;
                    case 4:                         // set sawtooth wave to 400 Hz
                        sawtooth_increment = S_400;
                        break;
                    case 5:                         // set sawtooth wave to 500 Hz
                        sawtooth_increment = S_500;
                        break;
                    case 6:
                        state = SINUSOID;
                        break;
                    case 7:
                        state = TRIANGLE;
                        break;
                    case 8:
                        state = SAWTOOTH;
                        break;
                    case 9:
                        state = SQUARE;
                        break;
                    default:
                        state = SAWTOOTH;
                }

                break;

            case TRIANGLE:                          // triangle waveform

                TA1CCTL0 &= ~(TIMER_A_CCTLN_CCIE);  // enable and disable interrupt
flags

                TA1CCTL1 &= ~(TIMER_A_CCTLN_CCIE);
                TA2CCTL0 |= (TIMER_A_CCTLN_CCIE);
                TA2CCTL1 &= ~(TIMER_A_CCTLN_CCIE);
                TA3CCTL0 &= ~(TIMER_A_CCTLN_CCIE);

                spi_data = 0x6000;

                key = keypad();                     // get user input
                __delay_us(200000);                 // software debounce

                switch(key) {
                    case 1:                         // set triangle wave to 100 Hz
                        triangle_increment = T_100;
                        break;
                    case 2:                         // set triangle wave to 200 Hz
                        triangle_increment = T_200;
                        break;
                    case 3:                         // set triangle wave to 300 Hz
                        triangle_increment = T_300;
                        break;
                    case 4:                         // set triangle wave to 400 Hz
                        triangle_increment = T_400;
                        break;
                    case 5:                         // set triangle wave to 500 Hz
                        triangle_increment = T_500;
                        break;
                    case 6:
                        state = SINUSOID;
                        break;
                    case 7:
                        state = TRIANGLE;
                        break;
                    case 8:
                        state = SAWTOOTH;
                        break;
                    case 9:
                        state = SQUARE;
                        break;
                    default:
                        state = TRIANGLE;
                }

                break;

            default:
                state = SQUARE;
        }
    }
}

void TA0_0_IRQHandler (void) {

    TA0CCTL0 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag
    DAC_write(spi_data);                            // write to the DAC

}
```

```
void TA1_0_IRQHandler(void) {                       // square wave ISR

    spi_data = 0x7DC0;                              // set spi_data to 3.0 V
    TA1CCTL0 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag

}

void TA1_N_IRQHandler(void) {                       // square wave ISR

    spi_data = 0x6000;                              // set spi_data to 0 V
    TA1CCTL1 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag

}

void TA2_0_IRQHandler (void) {                      // triangle wave ISR

    TA2CCTL0 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag

    switch(t_state){
        case UP:                                    // increment spi_data
            if (spi_data > 0x7DC0)
                t_state = DOWN;
            else
                spi_data += triangle_increment;
            break;
        case DOWN:                                  // decrement spi_data
            if (spi_data < 0x6250)
                t_state = UP;
            else
                spi_data -= triangle_increment;
            break;
        default:
            spi_data += 0;
    }

}

void TA2_N_IRQHandler (void) {                      // sawtooth wave ISR
    TA2CCTL1 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag
    if (spi_data > 0x7DC0)                          // if spi_data > 3.0 V, then
reset to 0V
        spi_data = 0x6000;
    else
        spi_data += sawtooth_increment;
}

void TA3_0_IRQHandler (void) {                      // sinusoid ISR
    TA3CCTL0 &= ~(TIMER_A_CTL_IFG);                 // clear interrupt flag
    if (sin_increment == 150)
        sin_increment = 0;
    spi_data = 0x6000 + sin_array[sin_increment++];

}
```

DAC.h

```
#ifndef DAC_H_
#define DAC_H_

#define CPU_FREQ 6000000
#define __delay_us(t_us) (__delay_cycles((((uint64_t)t_us)*CPU_FREQ) / 1000000))

#define MAX_VOLTAGE 330

#define SPI_CS BIT7
#define SPI_SCLK BIT5
#define SPI_COPI BIT6
#define SPI_PORT P1

void DAC_init(void);
void DAC_write(uint16_t spi_data);

#endif /* DAC_H_ */
```

DAC.c

```
#include "msp.h"
#include "DAC.h"
#include <math.h>

void DAC_init(void) {
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_SWRST;         // put the eUSCI into sowftware
reset
    EUSCI_B0->CTLW0 = (EUSCI_B_CTLW0_MSB            // configure SPI
                     | EUSCI_B_CTLW0_MST
                     | EUSCI_B_CTLW0_MODE_0
                     | EUSCI_B_CTLW0_SYNC
                     | EUSCI_B_CTLW0_SSEL__SMCLK
                     | EUSCI_B_CTLW0_SWRST);

    EUSCI_B0->BRW = 0x01;                           // clock divider at 1

    SPI_PORT->SEL0 |= (SPI_SCLK | SPI_COPI);        // configure SPI pins
    SPI_PORT->SEL1 &= ~(SPI_SCLK | SPI_COPI);

    SPI_PORT->SEL0 &= ~(SPI_CS);                    // configure CS as GPIO
    SPI_PORT->SEL1 &= ~(SPI_CS);
    SPI_PORT->DIR |= (SPI_CS);
    SPI_PORT->OUT |= (SPI_CS);                      // active low, so initialize high

    EUSCI_B0->CTLW0 &= ~(EUSCI_B_CTLW0_SWRST);      // clear software reset
}

void DAC_write(uint16_t spi_data) {

    uint8_t loByte, hiByte;
    loByte = spi_data & 0xFF;                   // mask lower byte
    hiByte = (spi_data >> 8) & 0xFF;            // mask upper byte

    SPI_PORT->OUT &= ~(SPI_CS);                 // set CS low

    while(!(EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG));    // wait for TXIFG to be set
(TXBUF is empty)
    EUSCI_B0->TXBUF = hiByte;

    while(!(EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG));    // wait for TXIFG to be set
(TXBUF is empty)
    EUSCI_B0->TXBUF = loByte;

    while(!(EUSCI_B0->IFG & EUSCI_B_IFG_RXIFG));    // wait for RXIFG at end of
transmission

    SPI_PORT->OUT |= (SPI_CS);                  // set CS high after transmission

    __delay_cycles(20);                         // delay after transmission
}
```

DCO.h

```
#ifndef DCO_H_
#define DCO_H_

#define FREQ_15_MHZ 1500000
#define FREQ_3_MHZ 3000000
#define FREQ_6_MHZ 6000000
#define FREQ_12_MHZ 12000000
#define FREQ_24_MHZ 24000000
#define PERIOD_100 60000
#define PERIOD_200 30000
#define PERIOD_300 20000
#define PERIOD_400 15000
#define PERIOD_500 12000
#define PERIOD 200
#define T_100 95
#define T_200 200
#define T_300 315
#define T_400 440
#define T_500 560
#define S_100 52
#define S_200 104
#define S_300 160
#define S_400 215
#define S_500 275

void set_DCO(int frequency);

#endif /* DCO_H_ */
```

DCO.c

```
#include "msp.h"
#include "DCO.h"
#include <math.h>

void set_DCO(int frequency){

    CS->KEY = CS_KEY_VAL; // unlock CS registers

    switch (frequency) {
        case FREQ_15_MHZ:
            CS->CTL0 = (CS_CTL0_DCORSEL_0); // set DCO to 1.5 MHz
            break;

        case FREQ_3_MHZ:
            CS->CTL0 = (CS_CTL0_DCORSEL_1);   // set DCO to 3 MHz
            break;

        case FREQ_6_MHZ:
            CS->CTL0 = (CS_CTL0_DCORSEL_2);   // set DCO to 6 MHz
            break;

        case FREQ_12_MHZ:
            CS->CTL0 = (CS_CTL0_DCORSEL_3);  // set DCO to 12 MHz
            break;

        case FREQ_24_MHZ:
            CS->CTL0 = (CS_CTL0_DCORSEL_4);  // set DCO to 24 MHz
            break;

        default: break;
    }

    CS->CTL1 = (CS_CTL1_DIVM__1 |      // MCLK / 1
                CS_CTL1_SELS__DCOCLK | // SMCLK / HSMCLK using DCO
                CS_CTL1_SELM__DCOCLK); // MCLK using DCO

    CS->KEY = 0; // lock CS registers

}
```

keypad.h

```
#ifndef KEYPAD_H_
#define KEYPAD_H_

#define ROW_PORT P4
#define COL_PORT P4
#define ROWS 0x0F
#define COLS 0xF0

#define ROW_1 0x01
#define ROW_2 0x02
```

```
#define ROW_3 0x04
#define ROW_4 0x08

#define COL_1 0x10
#define COL_2 0x20
#define COL_3 0x40
#define COL_4 0x80

#define KEY_LENGTH 3
#define STAR 10
#define POUND 11
#define NO_KEY 0xFF

void keypad_init(void);
uint8_t keypad(void);

#endif /* KEYPAD_H_ */
```

keypad.c

```
#include "msp.h"
#include "keypad.h"
#include "DAC.h"

void keypad_init(void)
{
    ROW_PORT->SEL0 &= ~(ROWS);      // Set row pins to GPIO
    ROW_PORT->SEL1 &= ~(ROWS);

    COL_PORT->SEL0 &= ~(COLS);      // Set col pins to GPIO
    COL_PORT->SEL1 &= ~(COLS);

    ROW_PORT->DIR &= ~(ROWS);       // Set row pins to Input
    COL_PORT->DIR |= (COLS);        // Set col pins to Output

    ROW_PORT->REN |= (ROWS);        // Enable row pin resistors
    ROW_PORT->OUT &= ~(ROWS);       // Set row pin pull down resistors
}

uint8_t keypad(void)
{
    int row = 0;

    COL_PORT->OUT |= (COLS);                // Set cols to high

    while (row == 0) {                      // Wait for button press
        row = (ROWS & ROW_PORT->IN);        // Get the row
    }

    switch (row) {
        case ROW_1:
            COL_PORT->OUT &= ~(COLS);       // Set Col 1 high
            COL_PORT->OUT |= (COL_1);
            if ((ROWS & ROW_PORT->IN) == ROW_1)
                return 1;                   // 1 was pressed

            COL_PORT->OUT &= ~(COLS);       // Set Col 2 high
            COL_PORT->OUT |= (COL_2);
            if ((ROWS & ROW_PORT->IN) == ROW_1)
                return 2;                   // 2 was pressed

            COL_PORT->OUT &= ~(COLS);       // Set Col 3 high
            COL_PORT->OUT |= (COL_3);
            if ((ROWS & ROW_PORT->IN) == ROW_1)
                return 3;                   // 3 was pressed

            COL_PORT->OUT &= ~(COLS);       // Set Col 4 high
            COL_PORT->OUT |= (COL_4);
            if ((ROWS & ROW_PORT->IN) == ROW_1)
                return 65;                  // A was pressed

        case ROW_2:
            COL_PORT->OUT &= ~(COLS);       // Set Col 1 high
            COL_PORT->OUT |= (COL_1);
            if ((ROWS & ROW_PORT->IN) == ROW_2)
                return 4;                   // 4 was pressed

            COL_PORT->OUT &= ~(COLS);       // Set Col 2 high
            COL_PORT->OUT |= (COL_2);
            if ((ROWS & ROW_PORT->IN) == ROW_2)
                return 5;                   // 5 was pressed

            COL_PORT->OUT &= ~(COLS);       // Set Col 3 high
            COL_PORT->OUT |= (COL_3);
            if ((ROWS & ROW_PORT->IN) == ROW_2)
                return 6;                   // 6 was pressed

            COL_PORT->OUT &= ~(COLS);       // Set Col 4 high
            COL_PORT->OUT |= (COL_4);
            if ((ROWS & ROW_PORT->IN) == ROW_2)
                return 66;                  // B was pressed

        case ROW_3:
            COL_PORT->OUT &= ~(COLS);       // Set Col 1 high
            COL_PORT->OUT |= (COL_1);
            if ((ROWS & ROW_PORT->IN) == ROW_3)
                return 7;                   // 7 was pressed

            COL_PORT->OUT &= ~(COLS);       // Set Col 2 high
            COL_PORT->OUT |= (COL_2);
            if ((ROWS & ROW_PORT->IN) == ROW_3)
                return 8;                   // 8 was pressed

            COL_PORT->OUT &= ~(COLS);       // Set Col 3 high
            COL_PORT->OUT |= (COL_3);
            if ((ROWS & ROW_PORT->IN) == ROW_3)
                return 9;                   // 9 was pressed
```

```
            COL_PORT->OUT &= ~(COLS);       // Set Col 4 high
            COL_PORT->OUT |= (COL_4);
            if ((ROWS & ROW_PORT->IN) == ROW_3)
                return 67;                  // C was pressed

        case ROW_4:
            COL_PORT->OUT &= ~(COLS);       // Set Col 1 high
            COL_PORT->OUT |= (COL_1);
            if ((ROWS & ROW_PORT->IN) == ROW_4)
                return STAR;                // * was pressed

            COL_PORT->OUT &= ~(COLS);       // Set Col 2 high
            COL_PORT->OUT |= (COL_2);
            if ((ROWS & ROW_PORT->IN) == ROW_4)
                return 0;                   // 0 was pressed

            COL_PORT->OUT &= ~(COLS);       // Set Col 3 high
            COL_PORT->OUT |= (COL_3);
            if ((ROWS & ROW_PORT->IN) == ROW_4)
                return POUND;               // # was pressed

            COL_PORT->OUT &= ~(COLS);       // Set Col 4 high
            COL_PORT->OUT |= (COL_4);
            if ((ROWS & ROW_PORT->IN) == ROW_4)
                return 68;                  // D was pressed

        default: return NO_KEY;

    }
}
```

B. References

[1] Microchip Technology, Chandler AZ, United States, *8/10/12-Bit Voltage Output Digital-to-Analog Converter with SPI Interface*, Accessed: May 5, 2021. [Online]. Available: https://ww1.microchip.com/downloads/en/DeviceDoc/22248a.pdf

[2] Parralax Inc., Rocklin CA, United States, *4x4 Matrix Membrane Keypad (#27899),* Accessed: May 5, 2021. [Online]. Available: https://ieee.ee.ucr.edu/sites/g/files/rcwecm1621/files/2018-10/2161473.pdf

[3] Texas Instruments, Dallas TX, United States, *MSP432P4xx SimpleLink™ Microcontrollers Technical Reference Manual*, Accessed: May 5, 2021. [Online]. Available: https://www.ti.com/lit/ug/slau356i/slau356i.pdf