Project 2:
Function Generator

CPE 329 Section 11
Spring 2017

Professor: Paul Hummel
Students: Derek Nola, Bevin Tang

Purpose:

The purpose of this project is to apply our knowledge of timers, interrupts, and a DAC to create a working function generator. This will be implemented with the MSP432R embedded system, the MCP4921 DAC, and a generic 4x3 button keypad.

System Requirements:

The function generator should support a square, sine, and sawtooth waveform. Every waveform -- except for the square wave -- need only to run a 50% duty cycle (the square wave will have to be able to run from 10% - 90%). All waveforms must be able to run at 100, 200, 300, 400, and 500Hz. Each Waveform will have a peak-to-peak output voltage of 5V with a DC offset of 2.5V.

The Keypad will control these features with the following buttons:

1) Set frequency to 100Hz
2) Set frequency to 200Hz
3) Set frequency to 300Hz
4) Set frequency to 400Hz
5) Set frequency to 500Hz
6) Additional Wave (Optional)
7) Square Wave
8) Sine Wave
9) Sawtooth Wave

*) Decrease Duty Cycle by 10%

0) Set Duty Cycle to 50%
#) Increase Duty Cycle by 10%

System Specification:

Table 1 shows the specifications of the project. This includes information on the Development board, the DAC, and the keypad.

MSP432R Development Board	
Size	2.25”x3.75”
Current Operating Clock Frequency	12Mhz
Processor	32Bit ARM Cortex -M4F
Input Voltage	5V
Input Current	100mA

MCP4921 DAC	
Resolution	12-bit with 50 steps
No. of Channels	1
Voltage Reference	External
Max Junction Temperature	150 °C
Communication Interface	SPI
Max V_{DD}	6.5V
Max Current at Input Pins	2mA
Max Current at Output Pins	25mA
Max Current at Supply Pins	50mA
Package	8-Pin PDP
KeyPad	
Buttons	12 Button (4 Rows by 3 Columns)
Connected Resistance	20 Ohms

Table 1: System Specifications

System Architecture:

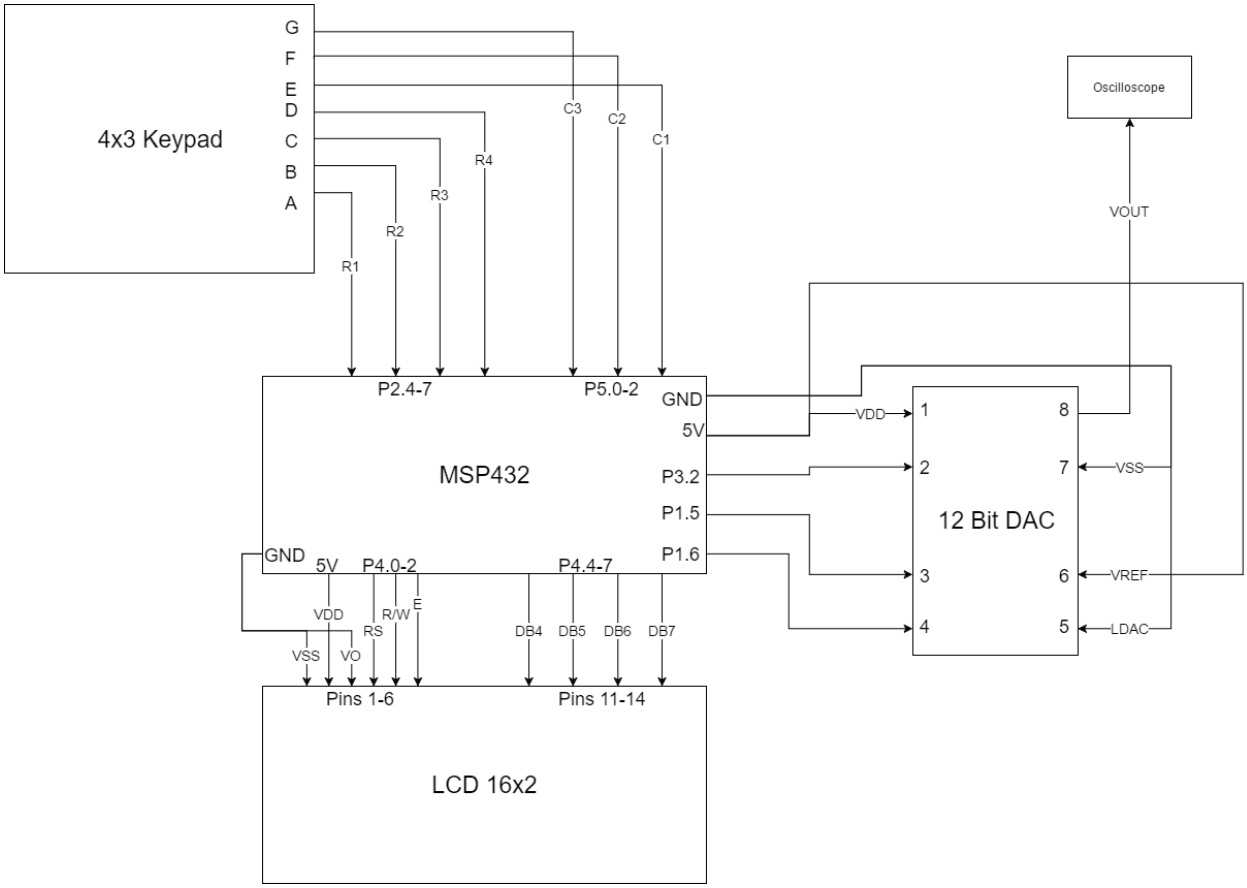

Figure 1: Circuit Schematic

Component Design:

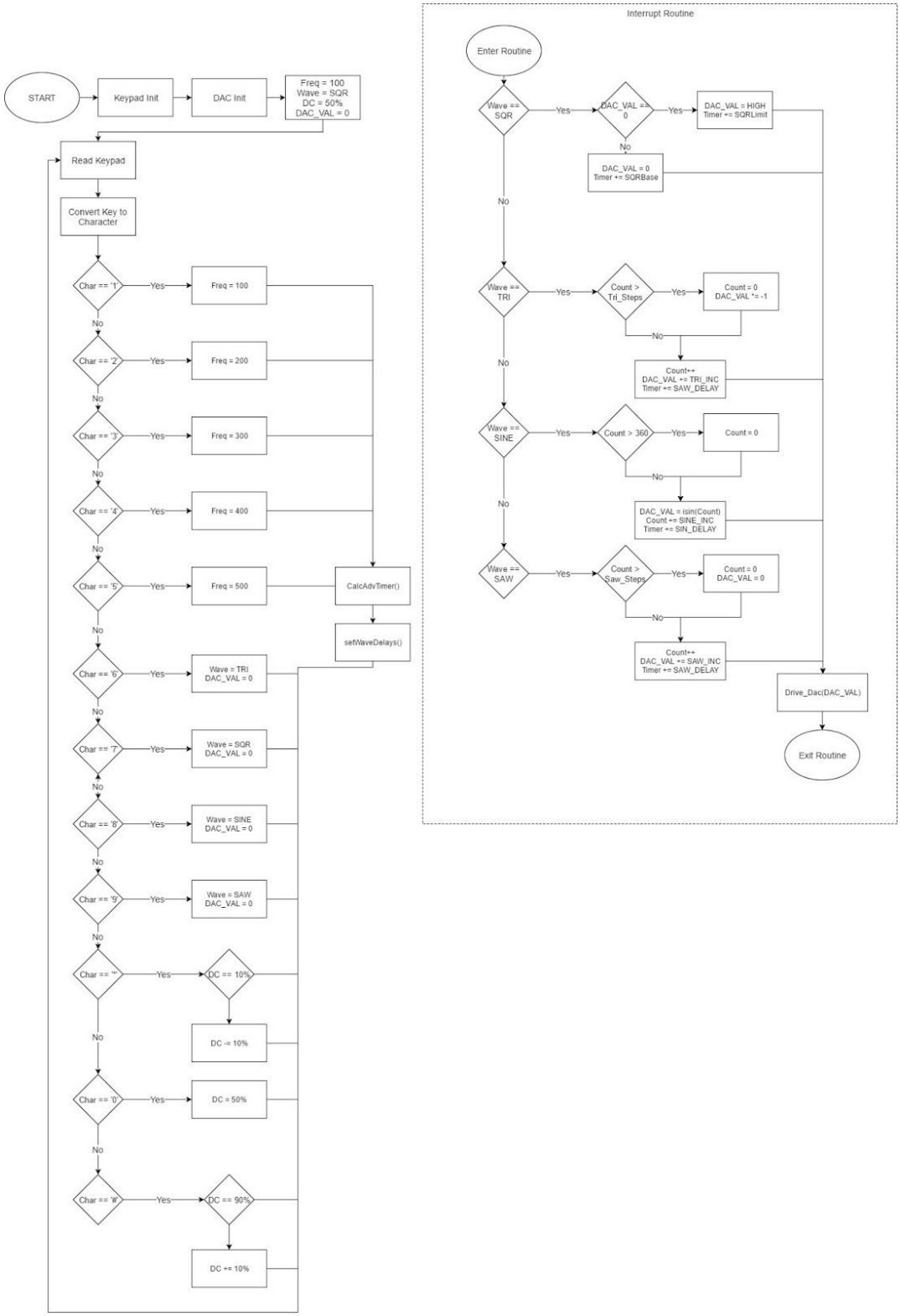

Figure 2: Program Flowchart

Bill of Materials:

Item #	Component Name	Supplier Name	Quantity	Price
1	MSP432P401 R Dev Board	Digi-Key	1	13.00
2	MCP4921 DAC	Digi-Key	1	2.00
3	MEMBRANE 3X4 MATRIX KEYPAD	Adafruit	1	4.00
4	BREAK-AWAY 0.1" 1X25-PIN STRIP MALE HEADER	Adafruit	1	0.75
5	Male/Male Jumper Wires - 20 x 6"	IEEE	1	1
6	Male/Female Jumper Wires - 20 x 6"	IEEE	1	1
			Total	21.75

Table 2: Bill of Materials

System Integration:

Waveforms:
First, the waveform code was generated; looking back at old assignments, we included an edited versions of our square triangle wave, and using the triangle wave code as a base, we made the sawtooth waveform. Next, we developed the sine code by approximating steps using Bhaskara's I approximation. The sine and triangle waveforms share the same implementation where the shape is achieved by incrementing and decrementing the DAC level with small steps. For example, in the sine waveform's case, we used 64 steps, each with a size according to Bhaskara's approximation.

To implement these forms with the keypad, we gave each type of waveform a key so that it can be reference with keypad input.

Duty Cycle:
Second, we implemented the duty cycle feature. This was achieved through the development of a function called *calcAdvTimer* which sets the *base* and *limit* counts that indicate how long a waveform should stay high or low. We then set keys to be used with keypad input that either decrement or increment the duty cycle by 10% per button input.

Frequency:
Next, we created 3 types of values that would help us alter the frequency and resolution of the waveforms: the number of steps, the step size, and the interrupt timer delay. To achieve a certain frequency, we had to tune these values and see what frequency they corresponded to on the oscilloscope.

This is where we ran into a little bit of trouble; Since we had 3 variables that can affect the frequency of the waveform, we had 3 sources of inaccuracy when searching for appropriate delay times. To make our lives easier, we decided to keep the number of steps and the step size constant, and just tampered with the delay time until we found the correct delay times that corresponded to 100, 200, 300, 400, and 500Hz. Possibly due to the variance in the step size and number of steps between waveforms, each waveform ultimately had a unique delay time.

DAC Optimization:
Finally, we optimized our DAC's functionality by editing the provided DAC function code. By default, upon startup, the DAC would wait for about 200ms before inputting or outputting any signals. This was due to an artificially imposed delay that was implemented in the DAC code; this delay took on the form of two while loops and a for loop. To optimize the DAC, we simply removed these loops so that the DAC would respond immediately to input and output. The resulting performance is displayed in **Figure 3**.

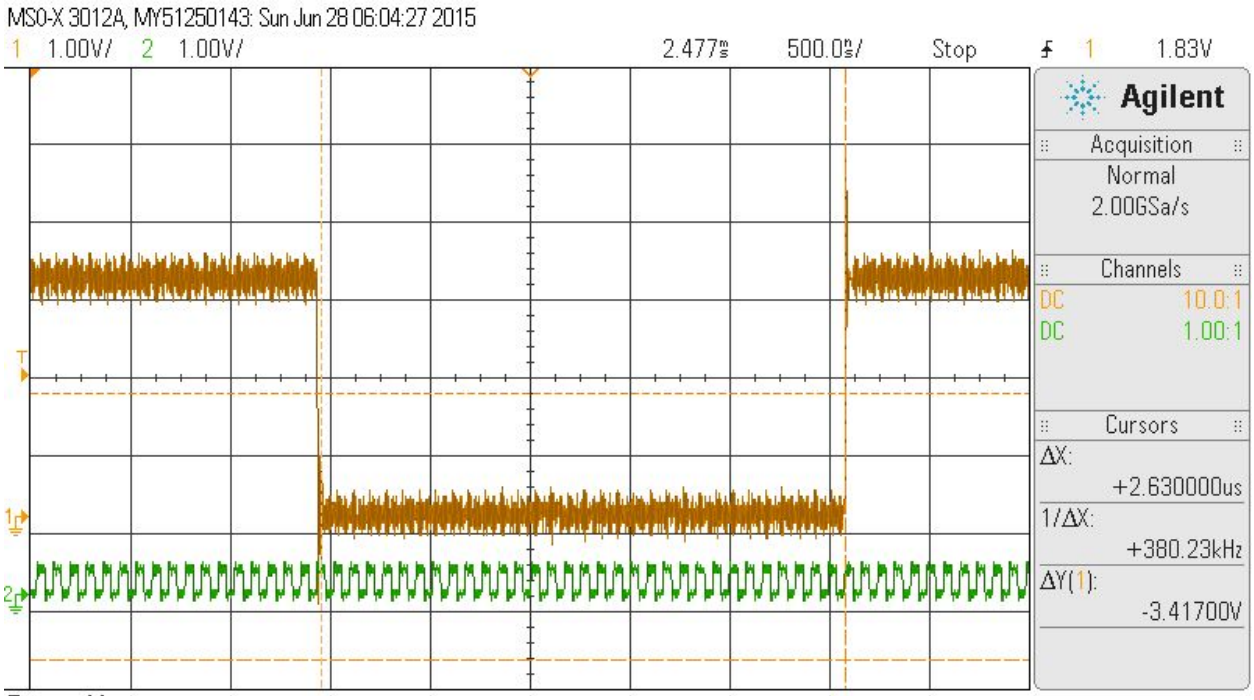

Figure 3: DAC Optimization -- The top waveform displays the time it takes for the DAC to complete a transmission of 16 bits. The system clock is shown below that as a reference. Without the optimization, we would expect a transmission delay in the order of milliseconds -- but depicted here, we have a transmission delay of only 2.63us!

YouTube Demonstration:
Here is a link to a live demonstration of Project 2: https://youtu.be/qVt0uopk_FA

Questions:

1) **For this lab, we used timers and interrupts to generate timing events. What are the benefits and drawbacks of using software delays instead of an interrupt-based system?**

   The benefits of using time-based systems is that we can accurately generate cyclic events (like the waveforms). Additionally, since the delays are software-based, the system is guaranteed to delay for that amount of in a sleep-like state before doing any processes. But the drawbacks come from this sleep-like state; in an interrupt-based system, the system could be doing meaningful work and processing in the time that the time-based system is being delayed.

2) **What are the different sources of error in your function generator? How do these sources of error impact the accuracy of your output waveforms? Try to express the magnitude of this error as specifically as possible.**

Our main sources of error were the magical numbers we found to set our waveform frequency and output voltage. These numbers emerged from trial and error and were not linearly proportional. Since we could only assign these numbers through our observations, we could just barely make our pk-pk voltage ratings within 5% error. Making things worse, the delay values (to set waveform frequency) were not compatible between waveforms. For example, we found 0x670 to be the 100Hz setting for the sine, and the sawtooth wave has a delay setting of 0x770. Despite these flaws in the frequency number assignments, our output frequency was our most accurate output property, being within 1% error.

Conclusion:

This project involved a particular balance of waveform resolution and output accuracy. Our project in particular implemented 3 things that directly affect waveform frequency, output voltage, and resolution: DAC step size, number of steps, and timer delay; the DAC step size determines the magnitude that the DAC level differs between steps and the timer delay determines the time in between steps. Initially, we tampered with all 3 of these steps at one time, which threw a lot of variables into determining appropriate settings for the waveforms. But once we decided to keep two of these aspects constant (the step size and number of steps), the calibration of these settings became easy to control and determine.

For future projects, keep as many variables constant as possible to isolate the variance in expected output values.

Appendices:

Code:

Main.c

```
/*
* Project 2
* Derek Nola and Bevin Tang
*/


#include "timers.h"
#include "keypad.h"
#include "msp.h"

#include <stdlib.h>
#include <stdio.h>

#define SQR 0x0D
#define SINE 0x0E
```

```
#define SAW 0x0F
#define TRI 0xA0

#define SINE_STEPS 64
#define SAW_STEPS 62
#define TRI_STEPS 32
#define VOLT_SCALE 4560


//GLOBALS
char GKEY = 'E';
float duty_cycle = 0.5f;
uint32_t freq = 100;
int16_t wave_type = SQR;
volatile unsigned int TempDAC_Value = 0;

//Freq delays arrays for all the wave types
uint16_t SINE_DELAYS[5] = {0x0670, 0x0338, 0x0225, 0x019C, 0x0149};
uint16_t TRI_DELAYS[5] =  {0x0770, 0x03B8, 0x027A, 0x01DA, 0x017A};
uint16_t SAW_DELAYS[5] =  {0x0780, 0x03B8, 0x028A, 0x01EC, 0x018C};

//Amount to increase dac output by each cycle
int16_t DAC_COUNT = 0;
int16_t SINE_INC = 32;
int16_t SAW_INC = 75;
int16_t TRI_INC = 145;

int16_t SINE_DELAY, SAW_DELAY, TRI_DELAY;

//Square wave delays for varying DC
uint16_t SQRBase, SQRLimit;


int32_t isin(int32_t deg, int32_t scale);
void setWaveDelays();
void Drive_DAC(unsigned int level);

int main(int argc, char const *argv[])
{

   //INIT Keypad
   keypad_init();

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


   set_DC0(FREQ_12_MHz);

   // Configure port bits for SPI
   P3->DIR |= BIT2;                     // Will use BIT4 to activate /CE on the DAC
```

```
P1SEL0 |= BIT6 + BIT5;                // Configure P1.6 and P1.5 for UCB0SIMO and UCB0CLK
P1SEL1 &= ~(BIT6 + BIT5);             //

//Configure SMCLK to output P4.3
P4->DIR |= BIT3;
P4->SEL0 |= BIT3;

//generate default 100 Hz square wave with 50% duty cycle
calcAdvTimer(&SQRLimit, &SQRBase, freq, duty_cycle);
setWaveDelays(freq/100 - 1);

// SPI Setup
EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_SWRST;   // Put eUSCI state machine in reset

EUSCI_B0->CTLW0 = EUSCI_B_CTLW0_SWRST |   // Remain eUSCI state machine in reset
                  EUSCI_B_CTLW0_MST   |     // Set as SPI master
                  EUSCI_B_CTLW0_SYNC  |     // Set as synchronous mode
                  EUSCI_B_CTLW0_CKPL  |     // Set clock polarity high
                  EUSCI_B_CTLW0_MSB;        // MSB first

EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_SSEL__SMCLK; // SMCLK
EUSCI_B0->BRW = 0x01;                         // divide by 16, clock = fBRCLK/(UCBRx)
EUSCI_B0->CTLW0 &= ~EUSCI_B_CTLW0_SWRST;      // Initialize USCI state machine, SPI
                                              // now waiting for something to
                                              // be placed in TXBUF

EUSCI_B0->IFG |= EUSCI_B_IFG_TXIFG;           // Clear TXIFG flag


//Setup interrupt and timer
TIMER_A0->CCTL[0] = TIMER_A_CCTLN_CCIE;       // TACCR0 interrupt enabled
TIMER_A0->CCR[0] = SQRBase;

TIMER_A0->CTL = TIMER_A_CTL_SSEL__SMCLK |     // SMCLK, continuous mode
        TIMER_A_CTL_MC__CONTINUOUS;

    // Enable global interrupt
__enable_irq();
NVIC->ISER[0] = 1 << ((TA0_0_IRQn) & 31);



while(1){
   GKEY = num_to_char(keypad_getkey());
   //main switch statement changes
   //global values based off keypress
   switch(GKEY){
      case '1':
          freq = 100;
          break;
```

```
        case '2':
            freq = 200;
            break;
        case '3':
            freq = 300;
            break;
        case '4':
            freq = 400;
            break;
        case '5':
           freq = 500;
           break;
        case '6':
            TempDAC_Value = 0;
            DAC_COUNT = 0;
            wave_type = TRI;
            break;
        case '7':
           TempDAC_Value = 0;
           wave_type = SQR;
           break;
        case '8':
           TempDAC_Value = 0;
           wave_name = "SINE";
           wave_type = SINE;
           break;
        case '9':
           TempDAC_Value = 0;
           DAC_COUNT = 0;
           wave_name = "SAW";
           wave_type = SAW;
        case '*':
           if(wave_type == SQR && duty_cycle - 0.1f > 0.1f)
               duty_cycle -= 0.1f;
           break;
        case '0':
           if(wave_type == SQR)
              duty_cycle = 0.5f;
           break;
        case '#':
            if(wave_type == SQR && duty_cycle + 0.1f < 0.91f)
                duty_cycle += 0.1f;
            break;
    }

    calcAdvTimer(&SQRLimit, &SQRBase, freq, duty_cycle);
    delay_ms(100);
}

return 0;
```

```
}


void TA0_0_IRQHandler(void){

   uint16_t SINE_INC = 360 / SINE_STEPS;

   static DBLE_COUNT = 0;
   if(wave_type == SQR){
       TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG;

       //Special Case for 100Hz to prevent overflow of timer
       if(freq == 100){
           if(TempDAC_Value == 0 && DBLE_COUNT == 0){
               TIMER_A0->CCR[0] += SQRLimit;
               TempDAC_Value = VOLT_SCALE;
               DBLE_COUNT = 0;
            }
           else if(TempDAC_Value != 0 && DBLE_COUNT == 0){
               DBLE_COUNT++;
               TIMER_A0->CCR[0] += SQRLimit;
           }
           else if(DBLE_COUNT == 1){
              DBLE_COUNT++;
              TempDAC_Value = 0;
              TIMER_A0->CCR[0] += SQRBase;
           }
           else if(DBLE_COUNT == 2) {
               TIMER_A0->CCR[0] += SQRBase;
               DBLE_COUNT = 0;
            }
      }
       else{
           //all other frequencies SQR wave
          if(!TempDAC_Value){
             TIMER_A0->CCR[0] += SQRLimit;
             TempDAC_Value = VOLT_SCALE;
          }
          else{
             TIMER_A0->CCR[0] += SQRBase;
             TempDAC_Value = 0;
          }
       }
      Drive_DAC(TempDAC_Value);
   }
   else if(wave_type == SINE){
       //SINE Wave delay
      TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG;

      //Can only do 360 due to the isin function
```

```
      //operating on a degree scale
      if(DAC_COUNT > 360)
         DAC_COUNT = 0;

      //calc sin value with a half scale to oscillate +-2.5V
      TempDAC_Value = isin(DAC_COUNT, VOLT_SCALE/2);
      Drive_DAC(TempDAC_Value);
      DAC_COUNT+= SINE_INC; //change voltage
      TIMER_A0->CCR[0] += SINE_DELAY; //reset timer

   }
   else if(wave_type == SAW){
       //Saw wave delay
      TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG;

      if (DAC_COUNT >= SAW_STEPS) {
         DAC_COUNT = 0;
         TempDAC_Value = 0;          //reset voltage
      }
      Drive_DAC(TempDAC_Value);
      TempDAC_Value += SAW_INC;      //increase voltage
      TIMER_A0->CCR[0] += SAW_DELAY; //reset timer
      DAC_COUNT++;
   }
   else if (wave_type == TRI){
       //Tri wave delay
        TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG;

        if (DAC_COUNT >= TRI_STEPS) {
           TRI_INC *= -1; //invert shift once count hits 5V limit
           DAC_COUNT = 0;
        }
        Drive_DAC(TempDAC_Value);
        TempDAC_Value += TRI_INC;      //change voltage
        TIMER_A0->CCR[0] += TRI_DELAY; //reset timer
        DAC_COUNT++;
   }
}

// A sine approximation via Bhaskara I's approx.
// @param x    Angle (0 to 360 allowed)
// @return     sine value*scale
int32_t isin(int32_t deg, int32_t scale){
   int8_t shift = 1;
   if(deg > 180){
      deg -= 180;
      shift = -1;
   }
   int32_t num = (deg << 2)*(180 - deg);
   float dem = 40500 - (deg*(180-deg));
```

```
   return scale * (num / dem * shift + 1);
}

//sets the appropriate delays for each wave type
void setWaveDelays(uint32_t freq){
    SINE_DELAY = SINE_DELAYS[freq];
    SAW_DELAY = SAW_DELAYS[freq];
    TRI_DELAY = TRI_DELAYS[freq];
}

//Drives the DAC Output
void Drive_DAC(unsigned int level){
  unsigned int DAC_Word = 0;
  DAC_Word = (0x1000) | (2*level/5 & 0x0FFF);   // 0x1000 sets DAC for Write
                                                // to DAC, Gain = 2, /SHDN = 1
                                                // and put 12-bit level value
                                                // in low 12 bits.

  P3->OUT &= ~BIT2;                // Clear P3.2 (drive /CS low on DAC)
                                   // Using a port output to do this for now

  EUSCI_B0->TXBUF = (unsigned char) (DAC_Word >> 8);  // Shift upper byte of DAC_Word
                                                      // 8-bits to right

  EUSCI_B0->TXBUF = (unsigned char) (DAC_Word & 0x00FF);  // Transmit lower byte to DAC
  P3->OUT |= BIT2;                                       // Set P3.2   (drive /CS high on DAC)

  return;
}
```

References:

Texas Instruments, MPS432_Technical_Reference_slau356f. March 2017. 2 May 2017
Texas Instruments, MSP432_Datasheet_slas826f. March 2017. 2 May 2017
Paul Hummel, MCP4921 DAC Datasheet. 2010. 2 May 2017
Paul Hummel, KeypadPinout. March 2017. 4 April 2017
Wikipedia, “Bhaskara I's sine approximation formula”. Feb 2017. 2 May 2017