Project 2 - Function Generator

Tanvi Kharkar
EE 329-05
Spring Quarter 2021
Professor Hummel

Device Behavior Description

This project designs the MSP432P401R microcontroller in conjunction with a 12-bit voltage output digital-to-analog converter (DAC) and a 4x4 membrane keypad to act as a function generator. The function generator is capable of generating square, sine, triangle, and sawtooth waveforms of varying duty cycles[1] and frequencies. The waveform type, duty cycle, and frequency are all controlled by external user input through the keypad.

[1] *Varying duty cycles applies to the square waveforms only*

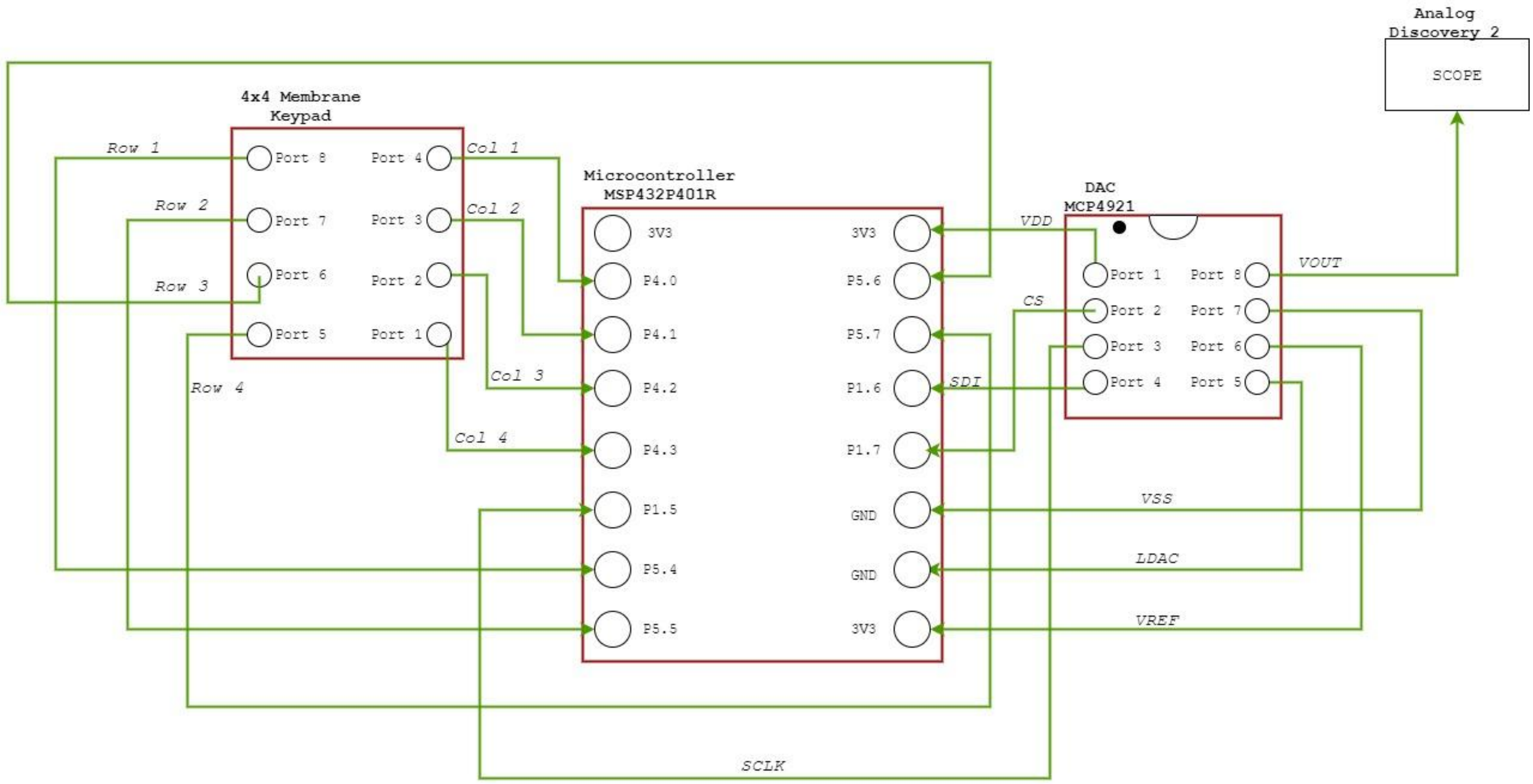
System Specification

Table 1: Function Generator Parameters and Respective Values

System Parameter	Value
Keypad Size	4 x 4
DAC Bit Resolution	12-Bit
Waveform Maximum Resolution	132427 samples/second
Waveform Output Voltage	$3V_{peak\text{-}to\text{-}peak}$

System Schematic

Figure 1: Schematic of function generator hardware configuration

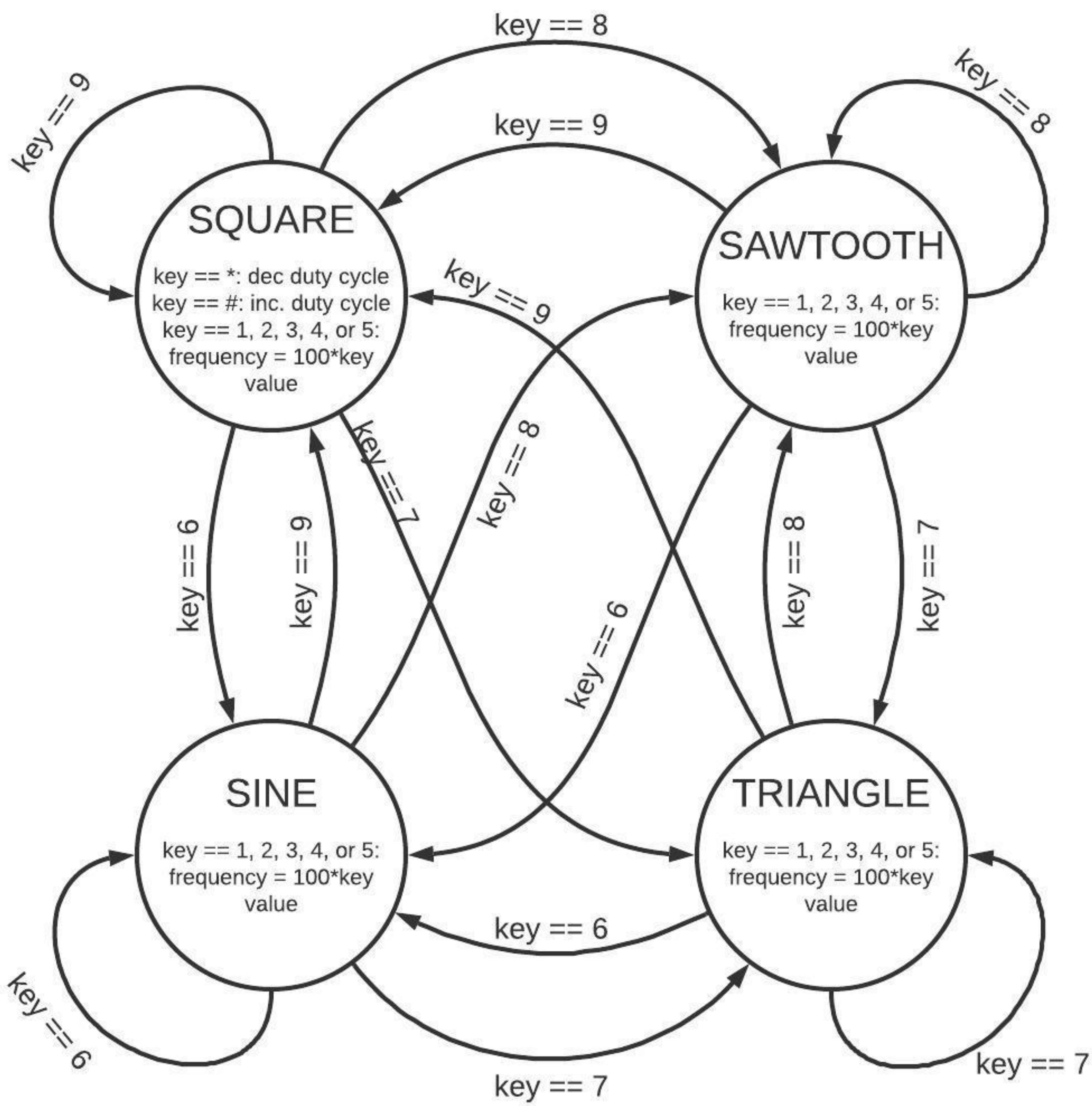


Software Architecture

The function generator is controlled by a finite state machine (FSM) with defined states for each waveform (square, sine, triangle, and sawtooth) and an interrupt service routine (ISR) for plotting each point on the waveform.

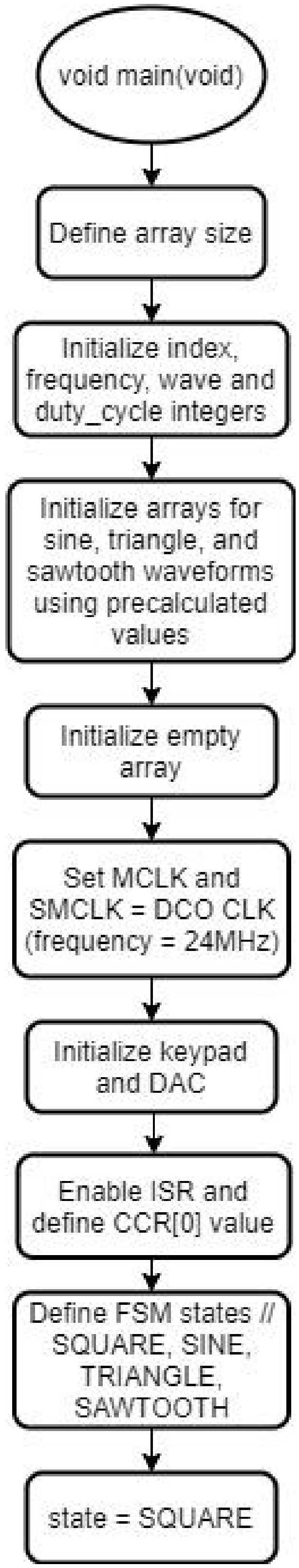
<u>State Diagram</u>

Figure 2: FSM Flowchart



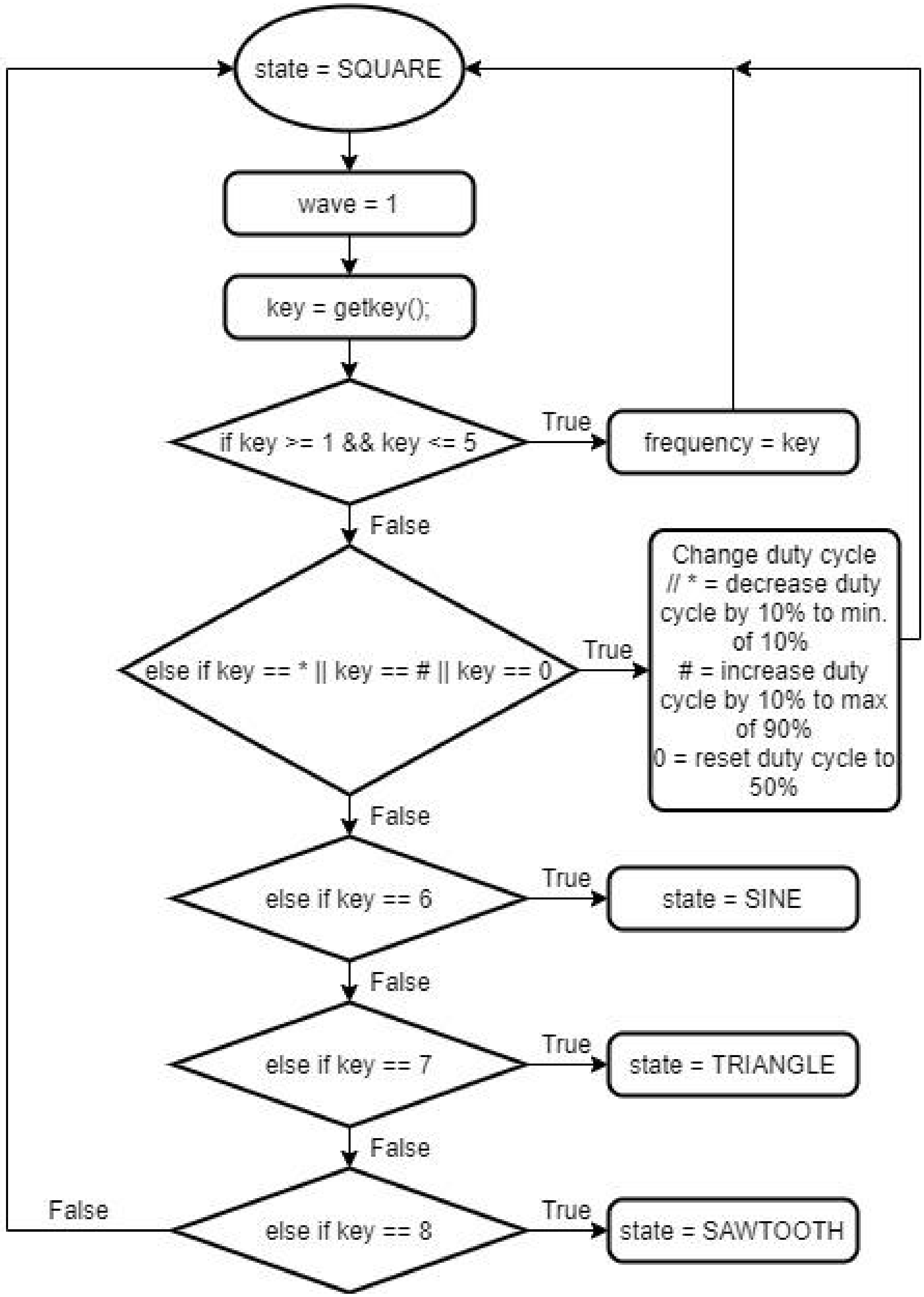
Main Diagram

Figure 3: Main flowchart



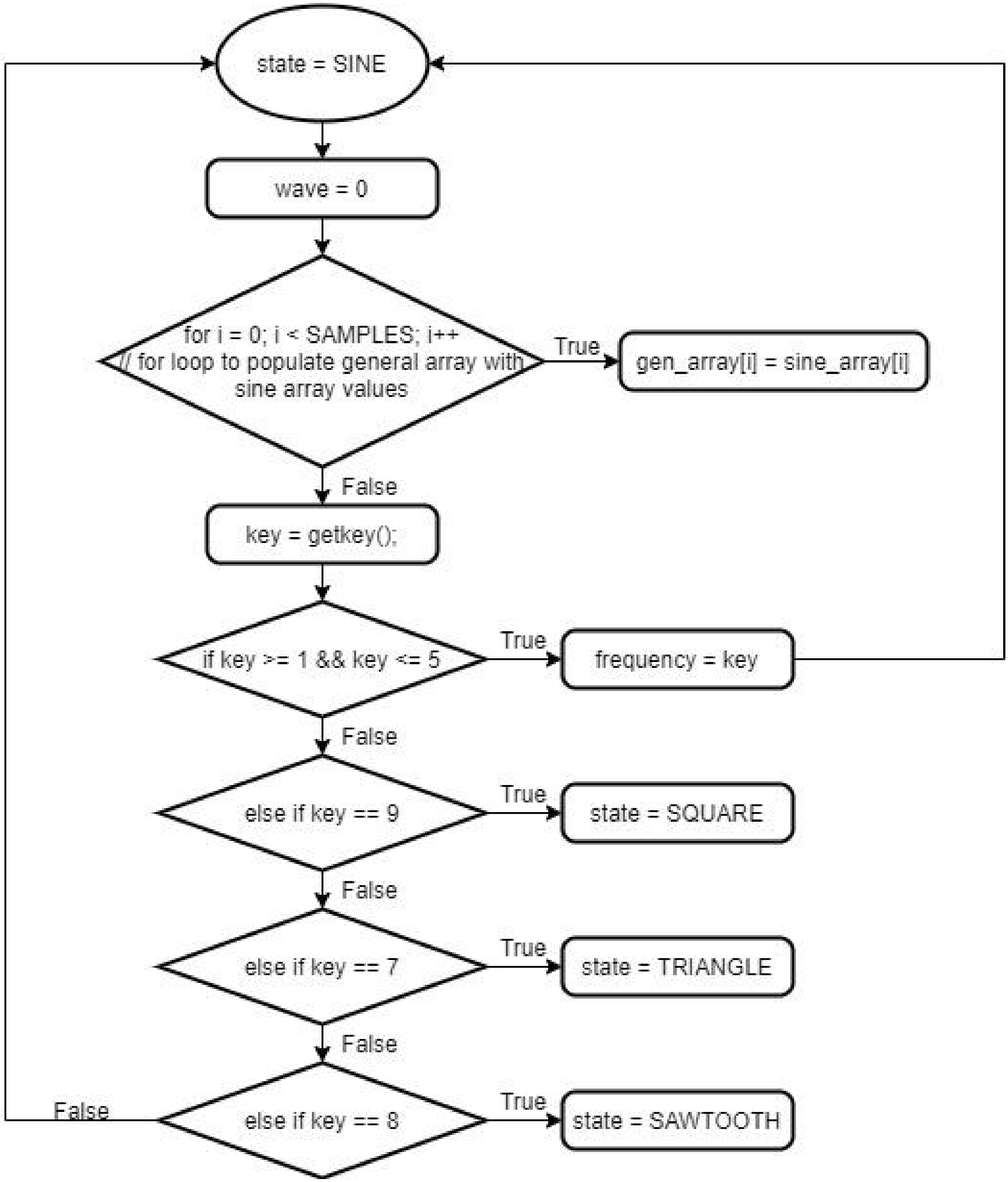
SQUARE State Diagram

Figure 4: Square waveform flowchart



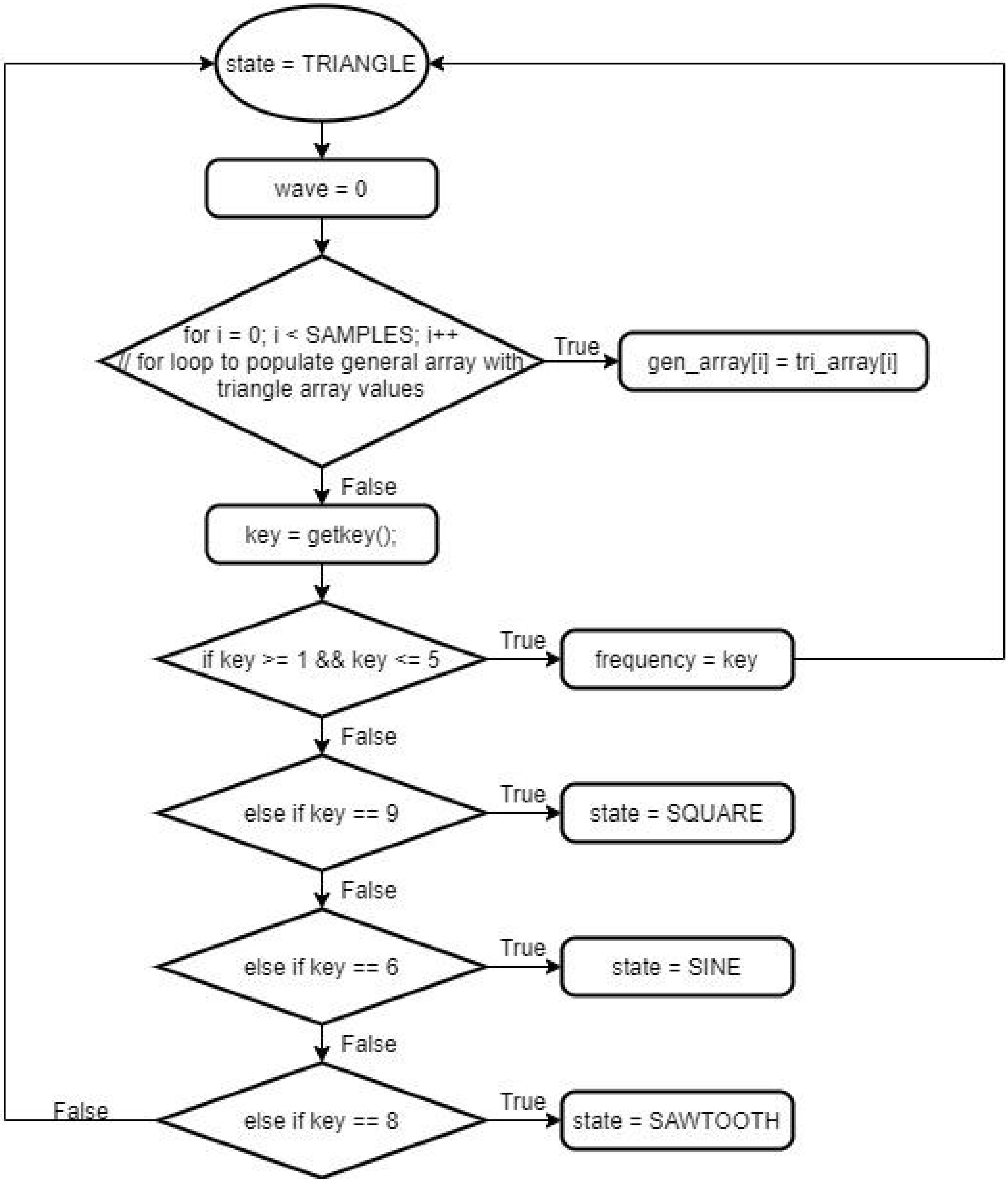
SINE State Diagram

Figure 5: Sine waveform flowchart



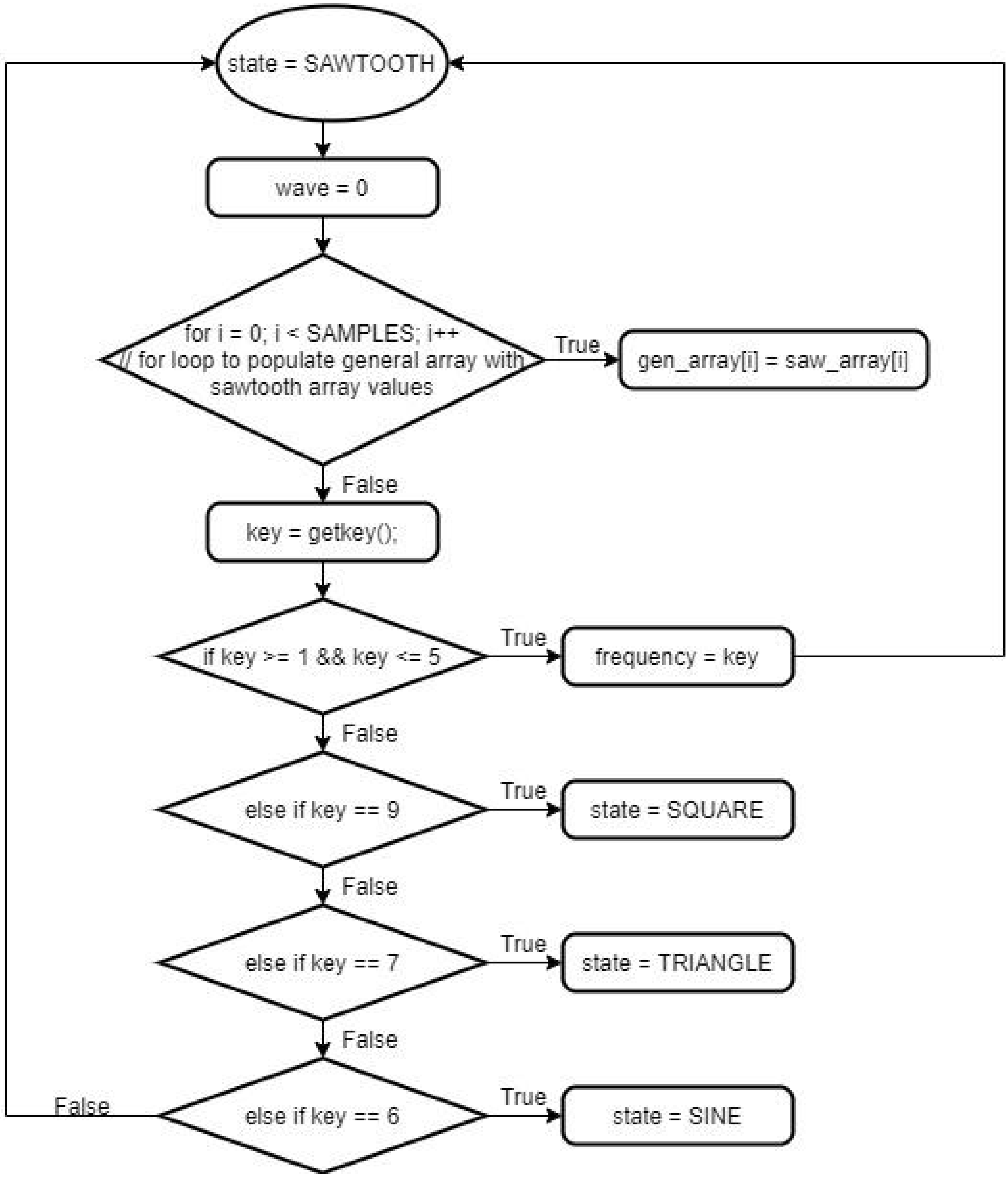
TRIANGLE State Diagram

Figure 6: Triangle waveform flowchart



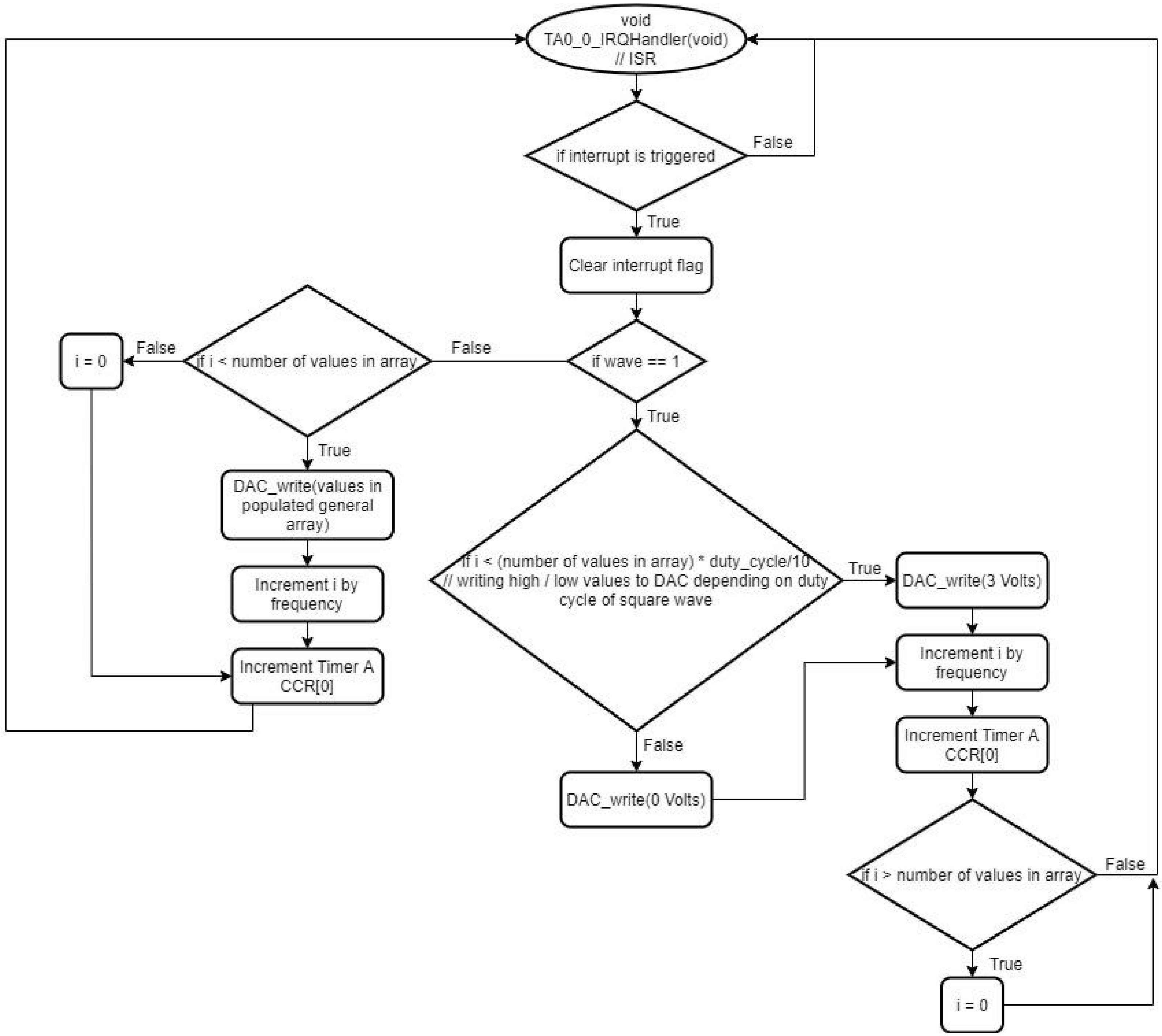
SAWTOOTH State Diagram

Figure 7: Sawtooth waveform flowchart



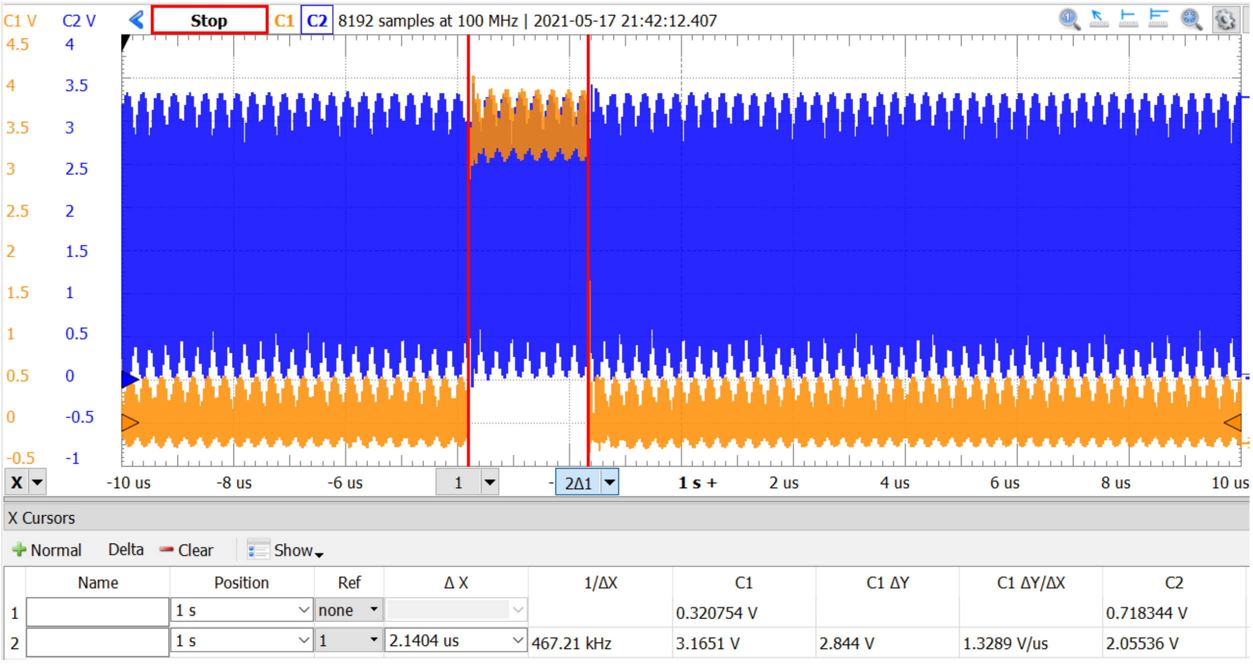
ISR Diagram

Figure 8: ISR flowchart



Maximum Resolution Calculation:

Figure 9: ISR Execution Timing (Yellow/Orange) Output Wave and MCLK (Blue) Output Wave



$$DCO\ Clock\ Frequency\ =\ 24MHz$$

$$Time\ taken\ to\ execute\ ISR\ \approx\ 2.1404\mu s\ \Rightarrow\ 4.672kHz$$

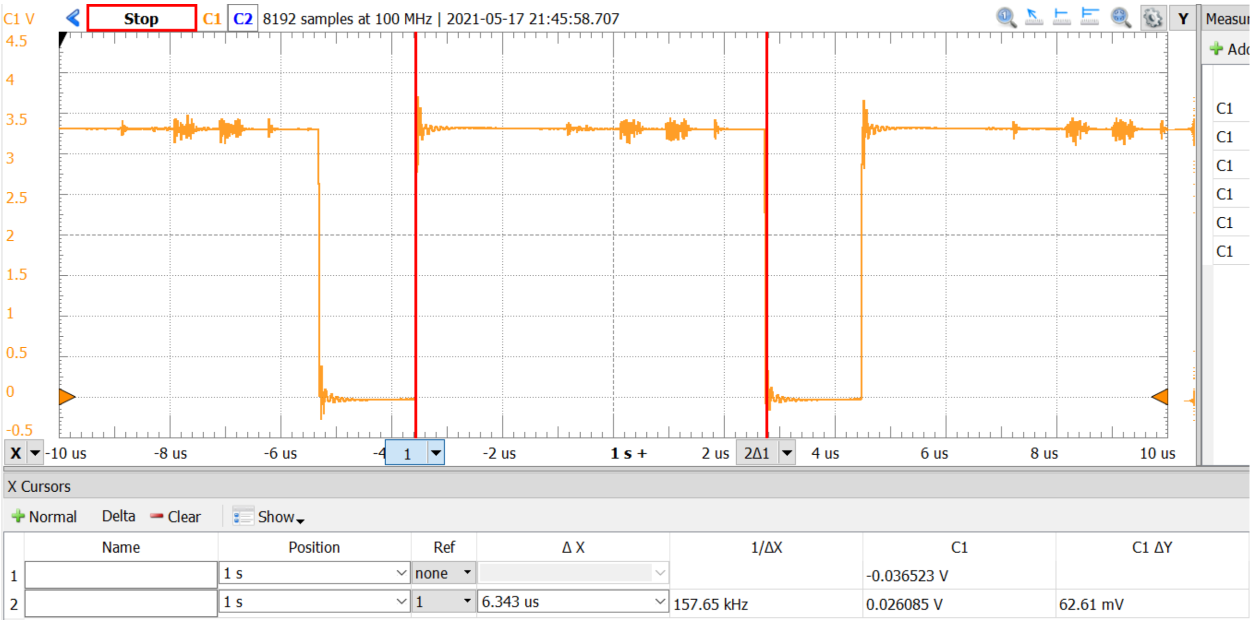
$$MCLK\ pulses\ needed\ to\ execute\ ISR\ =\ \frac{24MHz}{4.672kHz}\ \approx\ 51\ cycles$$

Figure 10: ISR Execution Timing (Yellow/Orange) vs. Count Pulses (Blue)

$$Lowest\ CCR[0]\ value\ before\ the\ ISR\ execution\ timing\ waveform\ "breaks"\ =\ 80\ cycles$$

$$80\ cycles\ -\ (MCLK\ pusles\ =\ 51\ cycles)\ =\ 29\ cycles\ \Rightarrow\ \frac{33\ cycles}{24MHz}\ =\ 1.208\mu s$$

Figure 11: DAC_write() Execution Timing



$$Time\ taken\ to\ execute\ DAC\ write() \ = 6.343\mu s$$

$$Minimum\ Time\ to\ Execute\ ISR\ and\ DAC\ write() \ = \ 1.208\mu s \ + \ 6.343\mu s \ = \ 7.551\mu s$$

$$Maximum\ Resolution \ = \ \frac{1s}{minimum\ time} \ = 132427 \ \frac{samples}{second}$$

$$Chosen\ Number\ of\ Samples \ = 1300$$

Appendices

main.c

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

#define NONE 0xFF   // No key pressed
#define SAMPLES 1300    // Number of samples in arrays, 1300
#define TIMER_A0_UP 350
#define USER_DELAY 10000000

int duty_cycle = 5; // Duty cycle = 50%
int frequency = 1;  // Frequency = 100Hz
int i = 0;  // Variable for indexing through arrays
uint8_t wave = 0;   // Wave = square wave
//int duty_cycle_samples = SAMPLES * duty_cycle;

int gen_array[];    // General array
int square_array_high[]; // Initialize arrays
int square_array_low[];
int tri_array[SAMPLES] = {
// precalculated values
};


int saw_array[SAMPLES] = {
// precalculated values
};


int sine_array[SAMPLES] = {
// precalculated values
};

void TA0_0_IRQHandler(void);

void main(void)
{
    WDT_A->CTL = WDT_A_CTL_PW | WDT_A_CTL_HOLD;     // stop watchdog timer

    set_DCO(FREQ_24_MHz);   // Set f = 24MHz
    keypad_init();  // Initialize keypad
    DAC_init();     // Initialize DAC


    char key = NONE;        // Start w/ assumption that no key is pressed

    TIMER_A0->CTL |= (TIMER_A_CTL_SSEL__SMCLK | TIMER_A_CTL_MC__CONTINUOUS);    // Select SMCLK for Timer A, cont. mode
    TIMER_A0->CCTL[0] |= TIMER_A_CCTLN_CCIE;     // Enable interrupts
    TIMER_A0->CCR[0] += TIMER_A0_UP;
    NVIC->ISER[0] = 1 << (TA0_0_IRQn);      // Enable flag for Timer A
    __enable_irq();     // Enable global interrupts


    typedef enum // Define FSM States
    {
        SQUARE,
```

```
    SINE,
    TRIANGLE,
    SAWTOOTH
} STATE_TYPE;

STATE_TYPE state = SQUARE; // Show SQUARE wave upon power up

while(1)
{
    switch(state)   // Switch statement to cycle through FSM cases
    {
        case SQUARE:
            wave = 1;   // For ISR to know when to write HIGH / LOW for square wave
            key = getkey(); // Detect keypad presses
            if (key >= 1 && key <= 5)   // Keys 1-5: frequency = 100 * key value
            {
                frequency = key;
                state = SQUARE;
                break;
            }

            else if (key == CHG_DUTY_CYCLE) // If any of the duty cycle keys pressed
            {
                if (key == RST_DUTY_CYCLE)  // If 50% duty cycle key pressed
                {
                    duty_cycle = 5;         // Reset duty cycle to 50%
                    state = SQUARE;         // Stay in SQUARE state
                    __delay_cycles(USER_DELAY); // Delay to debounce button presses
                    break;
                }

                else if (key == INC_DUTY_CYCLE) // If increase duty cycle key pressed
                {
                    if (duty_cycle < 9) // Do not increment duty cycle greater than 90%
                        duty_cycle++;   // Increment duty cycle
                    state = SQUARE;     // Stay in SQUARE state
                    __delay_cycles(USER_DELAY); // Button debouncer
                    break;
                }

                else if (key == DEC_DUTY_CYCLE) // If decrease duty cycle key pressed
                {
                    if (duty_cycle > 1) // Do no decrement duty cycle less than 10%
                    {
                        duty_cycle--;   // Decrement duty cycle
                    }
                    state = SQUARE;     // Stay in SQUARE state
                    __delay_cycles(USER_DELAY); // Button debouncer
                    break;
                }

                else if (key == SINE_KEY)   // If sine key pressed
                {
                    state = SINE;           // state = SINE
                    break;
                }

                else if (key == TRI_KEY)    // If triangle key pressed
                {
                    state = TRIANGLE;       // state = TRIANGLE
                    break;
                }
```

```
        else if (key == SAW_KEY)     // If sawtooth key pressed
        {
            state = SAWTOOTH;        // state = SAWTOOTH
            break;
        }

        else
        {
            state = SQUARE;          // default for SQUARE state: stay in SQUARE state
        }

    }
    break;


case SINE:
    wave = 0;        // For ISR to know not to create a square wave
    key = getkey(); // Detect keypad presses
    for (i = 0; i < SAMPLES; i++)
    {
        gen_array[i] = sine_array[i];   // Populate general array for DAC_write()
    }

    if (key >= 1 && key <= 5)
    {
        frequency = key;    // Set frequency equal to key press
        state = SINE;
        break;
    }

    else if (key == SQUARE_KEY) // If square key pressed
    {
        state = SQUARE;         // state = SQUARE
        break;
    }

    else if (key == TRI_KEY)    // If triangle key pressed
    {
        state = TRIANGLE;       // state = TRIANGLE
        break;
    }

    else if (key == SAW_KEY)    // If sawtooth key pressed
    {
        state = SAWTOOTH;       // state = SAWTOOTH
        break;
    }

    else
    {
        state = SINE;           // default for SINE state: stay in SINE state
    }
    break;


case TRIANGLE:
    wave = 0;        // For ISR to know not to create a square wave
    key = getkey(); // Detect keypad presses
    for (i = 0; i < SAMPLES; i++)
    {
        gen_array[i] = tri_array[i];   // Populate general array for DAC_write()
```

```
    }

    if (key >= 1 && key <= 5)
    {
        frequency = key;    // Set frequency equal to key press
        state = TRIANGLE;
        break;
    }

    else if (key == SQUARE_KEY)     // If square key pressed
    {
        state = SQUARE;             // state = SQUARE
        break;
    }

    else if (key == SINE_KEY)       // If sine key pressed
    {
        state = SINE;               // state = SINE
        break;
    }

    else if (key == SAW_KEY)        // If sawtooth key pressed
    {
        state = SAWTOOTH;           // state = SAWTOOTH
        break;
    }

    else
    {
        state = TRIANGLE;           // default for TRIANGLE state: stay in TRIANGLE state
    }
    break;


case SAWTOOTH:
    wave = 0;       // For ISR to know not to create a square wave
    key = getkey(); // Detect keypad presses
    for (i = 0; i < SAMPLES; i++)
    {
        gen_array[i] = saw_array[i];   // Populate general array for DAC_write()
    }

    if (key >= 1 && key <= 5)
    {
        frequency = key;    // Set frequency equal to key press
        state = SAWTOOTH;
        break;
    }

    else if (key == SQUARE_KEY)     // If square key pressed
    {
        state = SQUARE;             // state = SQUARE
        break;
    }

    else if (key == TRI_KEY)        // If triangle key pressed
    {
        state = TRIANGLE;           // state = TRIANGLE
        break;
    }

    else if (key == SINE_KEY)       // If sine key pressed
```

```c
                {
                    state = SINE;               // state = SINE
                    break;
                }

                else
                {
                    state = SAWTOOTH;           // default for SAWTOOTH state: stay in SAWTOOTH state
                }
                break;
        }
    }
}


void TA0_0_IRQHandler(void)
{
    if (TIMER_A0->CCTL[0] & TIMER_A_CCTLN_CCIFG)    // Check if CCIFG(CCTL[0]) is high (interrupted)
    {
        TIMER_A0->CCTL[0] &= ~TIMER_A_CCTLN_CCIFG;  // Clear interrupt flag

        if (wave == 1)  // For square wave
        {
            if (i < (SAMPLES * duty_cycle / 10))    // Write HIGH according to current duty cycle
            {
                DAC_write(HIGH);
                i += frequency;     // Increment index by frequency
                TIMER_A0->CCR[0] += TIMER_A0_UP;    // Increment timer
            }

            else
            {
                DAC_write(LOW);     // Write LOW according to current duty cycle
                i += frequency;     // Increment index by frequency
                TIMER_A0->CCR[0] += TIMER_A0_UP;    // Increment timer
                if (i > SAMPLES)    // Check if index > SAMPLES
                    i = 0;  // Reinitialize i = 0
            }
        }

        else    // For sine, triangle, and sawtooth waveforms
        {
            if (i < SAMPLES)    // Check if index < samples
            {
                DAC_write(gen_array[i]);    // Write values from populated general array
                i += frequency;     // Increment index by frequency
                TIMER_A0->CCR[0] += TIMER_A0_UP;    // Increment timer
            }

            else    // index > SAMPLES
            {
                i = 0;  // Reinitialize index to zero
                TIMER_A0->CCR[0] += TIMER_A0_UP;    // Increment timer
            }
        }

    }
}
```

keypad.h

```c
// Keypad Header

#ifndef KEYPAD_H_
#define KEYPAD_H_

#define COLUMNS (BIT0 | BIT1 | BIT2 | BIT3)     // Set column bits
#define ROWS (BIT4 | BIT5 | BIT6 | BIT7)        // Set row bits
#define DEC_DUTY_CYCLE 0x0E // Key * = decreases duty cycle by 10%
#define RST_DUTY_CYCLE 0x00 // Key 0 = resets duty cycle to 50%
#define INC_DUTY_CYCLE 0x0F // Key # = increases duty cycle by 10%
#define CHG_DUTY_CYCLE 0x0E | 0x00 | 0x0F   // Use to check if any duty cycle keys are pressed
#define SINE_KEY 0x06       // Key 6 = SINE wave
#define TRI_KEY 0x07        // Key 7 = TRIANGLE wave
#define SAW_KEY 0x08        // Key 8 = SAWTOOTH wave
#define SQUARE_KEY 0x09     // Key 9 = SQUARE wave

// Define Functions
void keypad_init(void);
char getkey(void);


#endif /* KEYPAD_H_ */
```

keypad.c

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

void keypad_init(void){

    P5->SEL0 &= ~ROWS;      // Set rows and columns to GPIO
    P5->SEL1 &= ~ROWS;
    P4->SEL0 &= ~COLUMNS;
    P4->SEL1 &= ~COLUMNS;

    P4->DIR &= ~COLUMNS;    // Set P4.0 - 4.3 as inputs (columns)

    P5->DIR |= ROWS;        // Set P5.4 - 5.7 as outputs (rows)

    P4->REN |= COLUMNS;     // Enable pull down resistor on inputs (columns)
    P4->OUT &= ~COLUMNS;

    P5->REN |= ROWS;        // Set all rows = 1
    P5->OUT |= ROWS;

    return;
}


char getkey(void){
    char columns, columns_read, rows; // Initialize variables

    P5->OUT |= ROWS;
    columns = P4->IN & COLUMNS;          // Read columns to detect button press

    if (columns == 0x00){                  // No button press detected
        return 0xFF;
    }

    rows = 0;
    P5->OUT &= ~ROWS;                        // Set all row outputs to 0

  while(rows < 4){
    P5->OUT = 0x10 << rows;                  // 0001_0000 << 1
    __delay_cycles(25);                      // Delay
    columns_read = P4->IN & COLUMNS;         // Reading columns

    if (columns_read != 0){                  // Button found
        break;
    }

    rows++;                                  // Increment rows by 1 if button not found
  }

  if (rows == 0){
      if (columns == 0x01){
          return 0x01;
      }
      if (columns == 0x02){
          return 0x02;
      }
      if (columns == 0x04){
          return 0x03;
      }
```

```
        if (columns == 0x08){
            return 0x0A;
        }
    }

    if (rows == 1){
        if (columns == 0x01){
            return 0x04;
        }
        if (columns == 0x02){
            return 0x05;
        }
        if (columns == 0x04){
            return 0x06;
        }
        if (columns == 0x08){
            return 0x0B;
        }
    }

    if (rows == 2){
        if (columns == 0x01){
            return 0x07;
        }
        if (columns == 0x02){
            return 0x08;
        }
        if (columns == 0x04){
            return 0x09;
        }
        if (columns == 0x08){
            return 0x0C;
        }
    }

    if (rows == 3){
        if (columns == 0x01){
            return 0x0E;
        }
        if (columns == 0x02){
            return 0x00;
        }
        if (columns == 0x04){
            return 0x0F;
        }
        if (columns == 0x08){
            return 0x0D;
        }
    }

    if (rows > 4){
        return 0xFF;
    }

    return 0xFF;
}
```

DCO.h

```c
// DCO Code

#ifndef DCO_H_
#define DCO_H_

#define CS_KEY_VAL 0x695A

#define CYCLES 3000000
#define FREQ_1_5_MHz CS_CTL0_DCORSEL_0
#define FREQ_3_MHz CS_CTL0_DCORSEL_1
#define FREQ_6_MHz CS_CTL0_DCORSEL_2
#define FREQ_12_MHz CS_CTL0_DCORSEL_3
#define FREQ_24_MHz CS_CTL0_DCORSEL_4

// function prototypes
void set_DCO(int);


#endif /* DCO_H_ */
```

DCO.c

```c
// DCO Source Code

#include "msp.h"
#include "DCO.h"

void set_DCO(int freq)
{
    CS->KEY = CS_KEY_VAL;   // Unlock CS Registers
    CS->CTL0 = 0;
    CS->CTL0 |= freq;           // Reset Control 0 Register
    CS->CTL1 = 0;
    CS->CTL1 = (CS_CTL1_DIVM__1 | CS_CTL1_SELM__DCOCLK
            | CS_CTL1_SELS__DCOCLK | CS_CTL1_DIVS__1);

    CS->KEY = 0;    // Lock CS Registers
}
```

DAC.h

```c
// DAC Header

/**
 * P1.7   Port 1.7   CS
 * P1.5   UCB0CLK    SCLK
 * P1.6   UCB0SIMO   COPI
 */

#define SPI_PORT P1
#define SPI_CS   BIT7
#define SPI_SCLK BIT5
#define SPI_COPI BIT6
#define DAC_BITS 4096   // 12-bit DAC, 2^12 = 4096
#define MAX_VOLTS 3300  // 3300mV = 3.3V
#define FLAT_CALIBRATION 42     // Measured voltage ~42mV from desired voltage, 94?
#define THREE_PT_THREE 0xFFF
#define HIGH 3070
#define LOW 0

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

DAC.c

```c
// DAC Source Code

#include "msp.h"
#include "DAC.h"

void DAC_init(void)
{
    EUSCI_B0->CTLW0 |= EUSCI_B_CTLW0_SWRST;      // Put BUS into software reset
    EUSCI_B0->CTLW0 = (EUSCI_B_CTLW0_MSB         // MSB first
                     | EUSCI_B_CTLW0_MST         // Controller mode
                     | EUSCI_B_CTLW0_MODE0       // 3-pin SPI
                     | EUSCI_B_CTLW0_SYNC        // Synchronous mode
                     | EUSCI_B_CTLW0_SSEL__SMCLK // Select SMCLK as source clock
                     | EUSCI_B_CTLW0_SWRST
                     | EUSCI_B_CTLW0_CKPL
                     );     // Keep software reset on

    EUSCI_B0->BRW = 0x01;   // Divide clock by 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);      // Initialize high (active low)

    EUSCI_B0->CTLW0 &= ~(EUSCI_B_CTLW0_SWRST);
}


void DAC_write(uint16_t mVolts)
{
    uint8_t loByte;     // Split voltage value into upper and lower bytes
    uint8_t hiByte;
    uint16_t data;

    data = DAC_volt_conv(mVolts);

    loByte = data & 0xFF;   // Clear upper byte of data
    hiByte = (data >> 8) & 0xFF;   // Shift upper byte, clear upper byte, then configure bits
    hiByte |= 0x30;

    SPI_PORT->OUT &= ~(SPI_CS);     // Set chip select low before transmission

    while(!(EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG));    // Wait for TXBUF to be empty
    EUSCI_B0->TXBUF = hiByte;   // Put upper byte into buffer

    while(!(EUSCI_B0->IFG & EUSCI_B_IFG_TXIFG));    // Wait for TXBUF to be set (empty)
    EUSCI_B0->TXBUF = loByte;   // Put lower byte into buffer

    while(!(EUSCI_B0->IFG & EUSCI_B_IFG_RXIFG));    // Wait for RXBUF to be set (empty)
    SPI_PORT->OUT |= SPI_CS;    // Set chip select high after transmission

    //__delay_cycles(50);
}

uint16_t DAC_volt_conv(uint16_t mVolts)
{
```

```
    uint16_t data = ((mVolts + FLAT_CALIBRATION) * DAC_BITS / MAX_VOLTS);   // Data conversion from keypad reading to
voltage output
    return data;
}
```

External References

[1] *MSP432P4xx SimpleLink™ Microcontrollers Technical Reference Manual*, Texas Instruments, Mar. 2015 [Revised June 2019].

[2] Texas Instruments, "MSP432P401R, MSP432P401M SimpleLink™ Mixed-Signal Microcontrollers," MSP432P401R datasheet, Mar. 2015 [Revised June 2019].

[3] Parallax, "4x4 Matrix Membrane Keypad (#27899)," Keypad datasheet, Dec. 2011.

[4] Microchip Technology Inc., "8/10/12-Bit Voltage Output Digital-to-Analog Converter with SPI Interface," DAC datasheet, Apr. 2010.