**Project 2:  
Function Generator**

**Cal Poly CPE 329-01**

**Spring 2017**

short line

Brendan Baronia

Melinda Ong

Professor Gerfen, Jeffrey B.

**May 3, 2017**

# Project Demonstration: https://youtu.be/JsQrYTTkXBg

# Purpose: The purpose of this system is to integrate an MSP432 board with an external DAC and keypad to create a function generator in order to:

1) Understand how to use timers and interrupt to generate timing events.

2) Understand how to use a serial-peripheral interface (SPI).

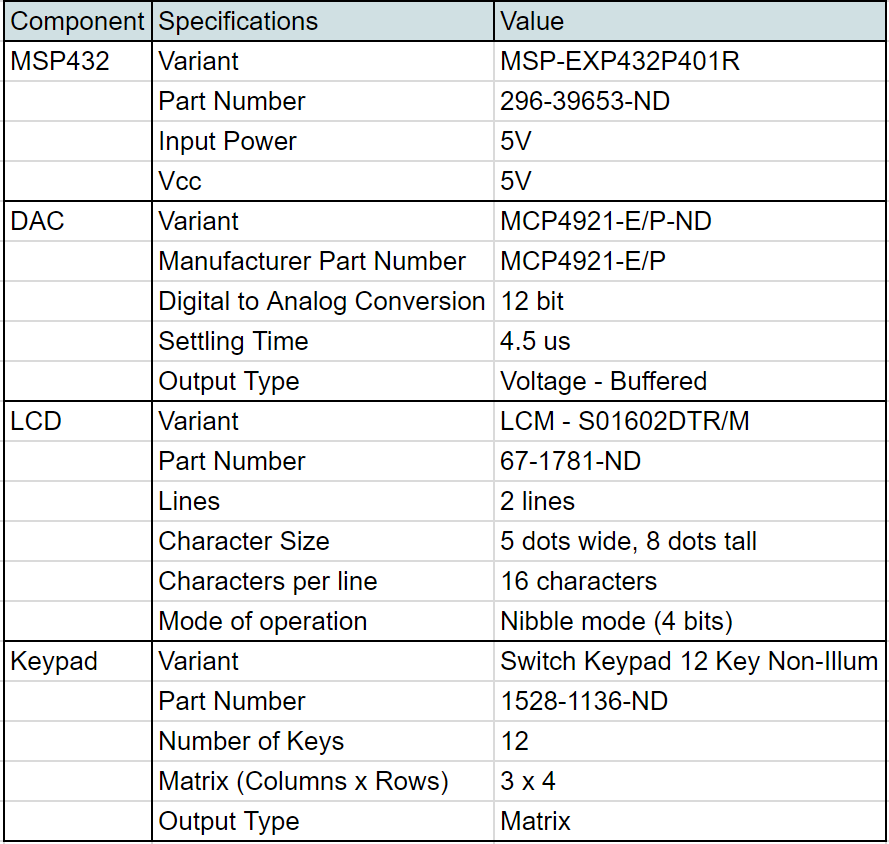
3) Understand how to use digital-to-analog (DAC) converters to generate analog signals from a microcontroller.

The system is able to generate a square wave with varying duty cycles, a sawtooth wave, and a sine wave at 100 Hz, 200 Hz, 300 Hz, 400 Hz, and 500 Hz.

# System Requirements:

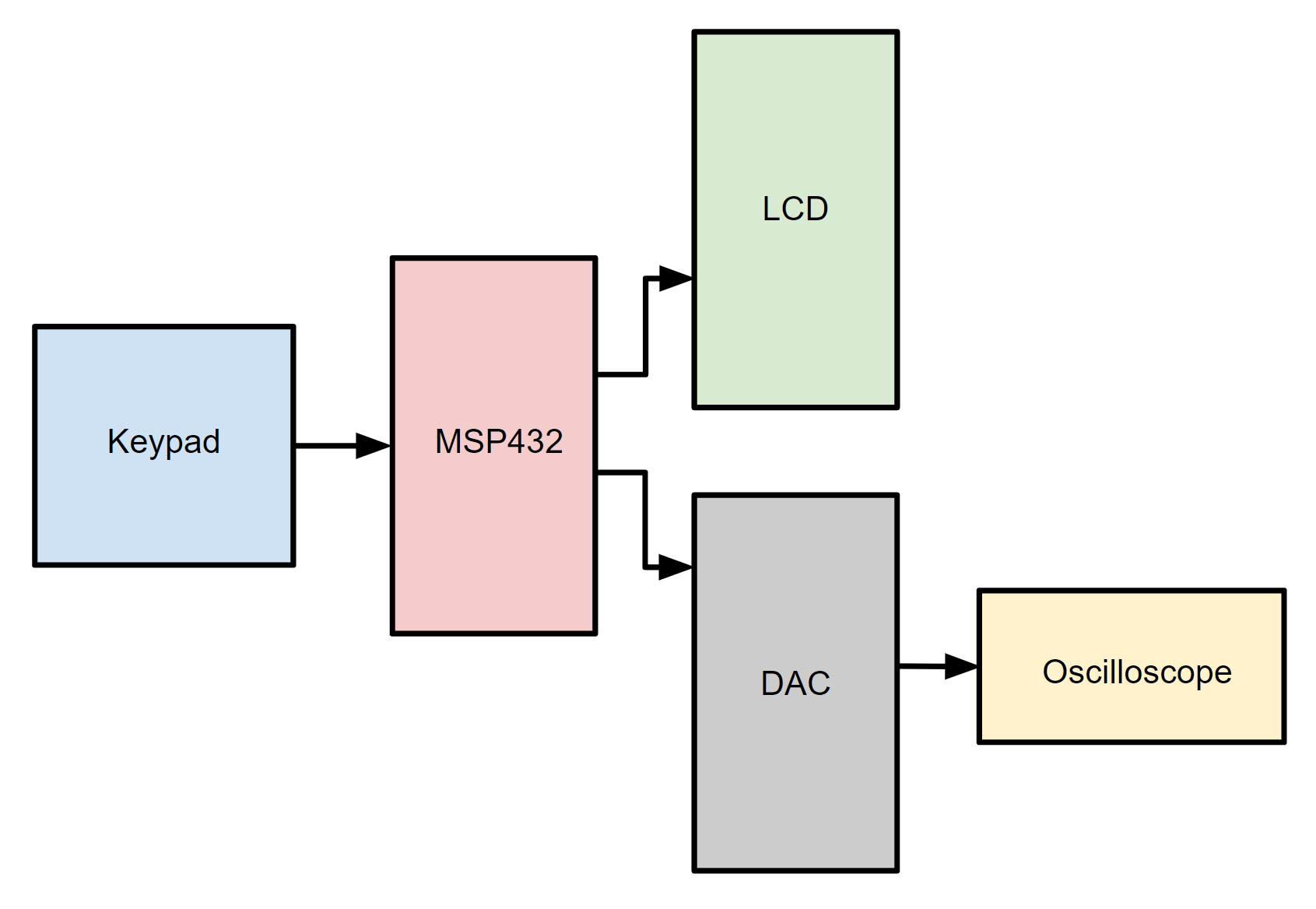
1. Shall optimize the SPI DAC drive to show:
   1. Total time required to complete the transmission of 16 bits and raise /CS at the end.
   2. Calculations which demonstrate the potential maximum update rate for the DAC and hence an upper bound on the resolution of your function generator when generating a 500 Hz sine wave.
2. Shall use a microcontroller and an external DAC.
   1. The external DAC shall have an SPI interface and a minimum of 8 bits resolution.
3. Shall be capable of producing:
   1. A square wave with variable duty cycle.
   2. A sinusoidal waveform.
   3. A sawtooth waveform.
4. Shall be DC-biased around Vdd/2.
5. Shall have adjustable frequencies:
   1. 100 Hz, 200 Hz, 300 Hz, 400 Hz, 500 Hz within 5% error.
6. Shall display a 100 Hz square wave with 50% duty cycle upon start up.
7. Shall utilize the keypad buttons 1-5 to set the waveform frequency in 100 Hz increments (1 for 100 Hz, 2 for 200 Hz, etc).
8. Shall utilize the keypad buttons 7, 8, and 9 to set the output waveform to Square, Sine, and Sawtooth.
9. Shall utilize the keypad buttons ✱, 0, and # to change the duty cycle of the square wave.
   1. The ✱ shall decrease the duty cycle by 10% down to a minimum of 10%.
   2. The # shall increase the duty cycle by 10% up to a maximum of 90%
   3. The 0 key shall reset the duty cycle to 50% d. The keys ✱, 0, and # shall have no affect on the sin or sawtooth waveforms.
10. Shall utilize the on-chip timer to generate interrupts that generate timing events for the function generator.

# System Specification:

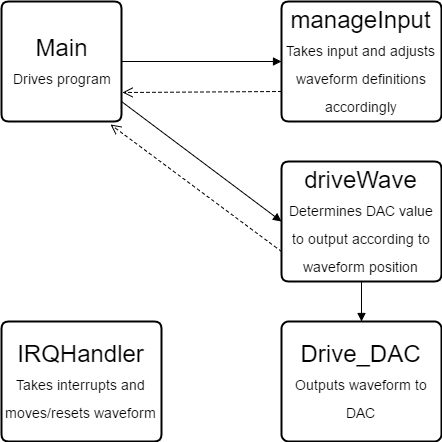


**Table 1: System Specifications**

System Architecture:

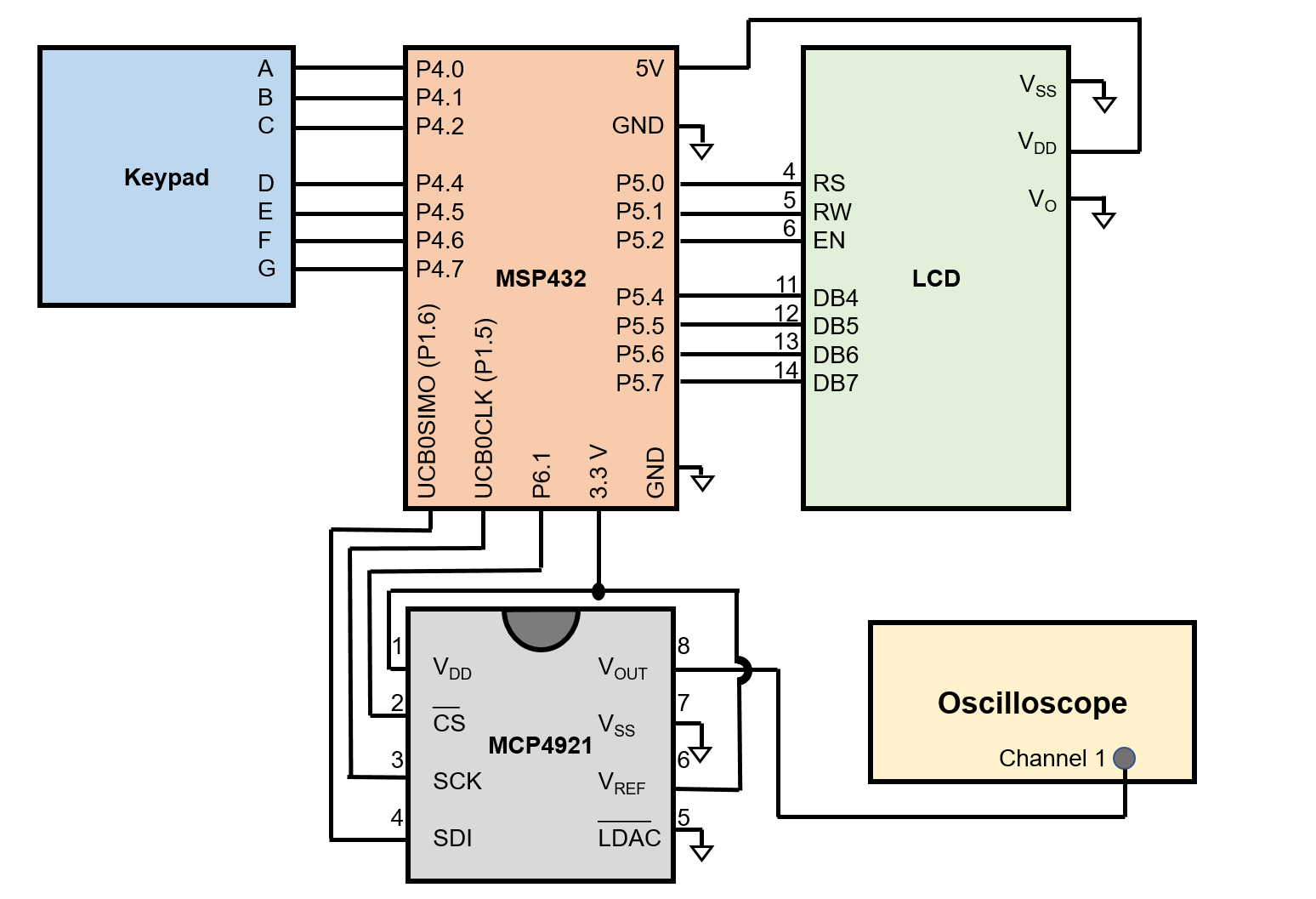


**Figure 1: Overall system block diagram**

****

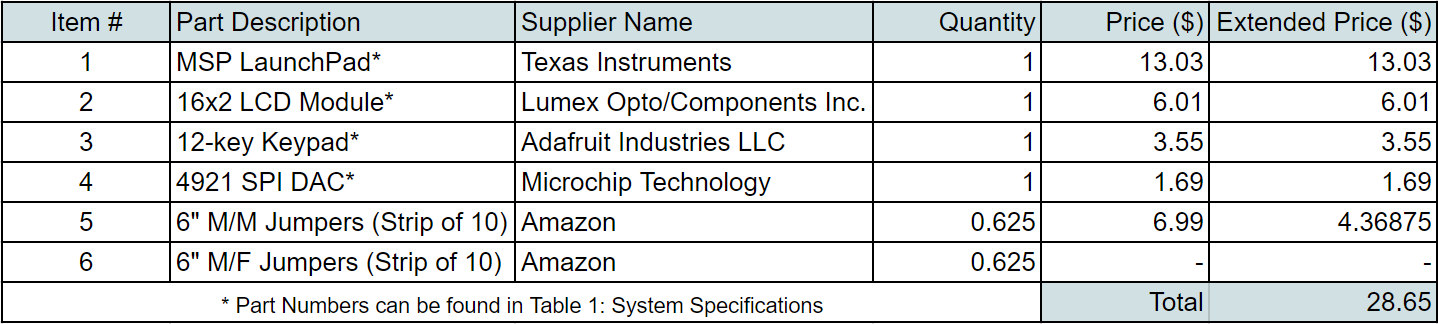
**Figure 2: Top-level Software flowchart**

# Component Design;



**Figure 3: Schematic Diagram of the MSP432, the LCD, the DAC, and the keypad**

# Bill of Materials:



**Table 2: Bill of Materials**

System Integration:

***a.) Development Process***

This system was developed from a simple program generating a 50 Hz square wave. The code controlling the LCD and keypad was developed independently from the code controlling the waveforms to help isolate bugs. This design choice was wise as problems arose with both the keypad and LCD. The keypad controlled variables within the program that defined wave type, frequency, and duty cycles, and once this operation was confirmed to be working, a simple waveform was integrated.

The basic 50 Hz square wave was migrated into the program and was confirmed to be working, then adjusted to follow the frequency set by the keypad. Once this frequency change was working, duty cycles were added as well. With this system working, the sawtooth wave was the next step. The frequency adjustments from the square wave were recycled and simple mathematic operations were done to generate a sawtooth waveform. Finally, the sine wave was integrated, using a lookup table to generate values rather than calculating the sinusoid during runtime.

***b.) Screenshots***

|  |  |
| --- | --- |
| **Square Wave (Duty Cycle Variable)\*** | |
| **Figure 3.1: 10% Duty Cycle ~ 300 Hz** | **Figure 3.6: 60% Duty Cycle ~ 300 Hz** |
| **Figure 3.2: 20% Duty Cycle ~ 300 Hz** | **Figure 3.7: 70% Duty Cycle ~ 300 Hz** |
| **Figure 3.3: 30% Duty Cycle ~ 300 Hz** | **Figure 3.8: 80% Duty Cycle ~ 300 Hz** |
| **Figure 3.4: 40% Duty Cycle ~ 300 Hz** | **Figure 3.9: 90% Duty Cycle ~ 300 Hz** |
| **Figure 3.5: 50% Duty Cycle ~ 300 Hz** | **\*** The square wave screenshots above represent the successful implementation of the duty cycle variation feature.  Screenshots representing the successful implementation of the frequency variation feature are represented below in **Table 4**. A complete demonstration of the function generator is above under Project Demonstration. |

**Table 3: Square Wave (Duty Cycle Variable)**

|  |  |
| --- | --- |
| **Sawtooth/Sine Wave (Frequency Variable)\*** | |
| **Figure 4.1: Sawtooth Wave ~ 100 Hz** | **Figure 4.6: Sine Wave ~ 100 Hz** |
| **Figure 4.2: Sawtooth Wave ~ 200 Hz** | **Figure 4.7: Sine Wave ~ 200 Hz** |
| **Figure 4.3: Sawtooth Wave ~ 300 Hz** | **Figure 4.8: Sine Wave ~ 300 Hz** |
| **Figure 4.4: Sawtooth Wave ~ 400 Hz** | **Figure 4.9: Sine Wave ~ 400 Hz** |
| **Figure 4.5: Sawtooth Wave ~ 500 Hz** | **Figure 4.10: Sine Wave ~ 500 Hz** |

**Table 4: Sawtooth/Sine Wave (Frequency Variable)**

***c.) Bugs and Resolutions***

The first bug encountered was with the keypad. All the code was recycled and expected to be functional, but the bottom row didn’t work. After troubleshooting wiring and finding no errors, the pin controlling the fourth row was switched on the MSP432 and confirmed to be functional.

Along with the keypad bug, there were errors integrating the LCD. Once again, all code was recycled from a functioning project. Unfortunately, the error was not found, but since previous code also doesn’t work, the LCD may be nonfunctional.

The final major error encountered was a struggle to generate the sine wave. Initially, 2000 sample values were included to generate the sine wave, but they were generated using a standard sinusoid based around 0. Unfortunately, the DAC could only be driven with positive values, so upon inputting negative values, the result was shifted up. This resulted in an oddly shaped waveform with issues in continuity, and the bug was difficult to track down. Once the lookup table was rewritten so that all sample values were positive, the system worked as expected.

# Answers to 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?***

Software delays are easier to trace since they appear in code, and are therefore easier to troubleshoot. A delay can be implemented for a specific amount of time via a function, and delays can be adjusted in code. However, these delays are difficult to time exactly as code takes multiple instruction cycles to complete.

Software delays are, however, very energy intensive and waste processing power. Since software must essentially waste time on the processor to delay the system, no other tasks can be accomplished during the delay. An interrupt based system allows code to run as expected with interrupts generated after a number of clock cycles, so they do not waste power or processing time. However, unlike software delays, they are more difficult to track and adjust in code because of their hardware base.

# ***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.***

The sources of error in our function generator originate from a variety of sources, including variation in clock speeds of the MSP432, signal propagation across wiring, software execution time, and a lack of adjustments to be made in timing. The sinusoidal and sawtooth waveforms also had errors in the sample size, with only 2000 samples to create the waveform. Overall, these errors had a minute effect on the waveforms. 2000 samples is still plenty to create a fairly continuous waveform, with the DAC creating almost perfect waveforms from the digital inputs. However, there was a range up to 5% error in timing due to the constraints listed, causing a minor difference in frequency. This error is largely irrelevant in most applications, but could be significant in very precise systems.

# Conclusion:

The purpose of this project was to integrate the MSP432 board with an external DAC and keypad to create a function generator. Through input from the keypad, a square wave, sawtooth wave, and sine wave can be selected. Duty cycles can be adjusted for the square wave, and all waves can be displayed in 100 Hz, 200 Hz, 300 Hz, 400 Hz, and 500 Hz frequencies. The system was required to be accurate within 5% for each frequency, and was also restricted to the timing constraints of the MCP4921 DAC when producing the signal.

This system could be improved with an external display with waveform information. During development, an external display was considered, but due to hardware issues the display was never properly integrated. The code does exist to integrate a display, however, and on a future implementation that system could be added.

Other improvements could be to waveform resolution and frequency accuracy. Due to error in system timing, the actual waveforms generated were not produced at exactly the frequency selected, but were within 5% of the desired value. With further optimization and adjustment, closer frequencies could be achieved. Other frequencies could also be added to the system, expanding its functionality. The system also generates a maximum of 2000 digital points to reproduce waveforms in an analog form, but further points could be added to improve waveform resolution.

Finally, packaging could be improved to reduce the amount of hardware and wiring needed. The MSP432 does include an internal DAC, so the external DAC used could be removed. The keypad used to control the system is also a basic 12 key system, with buttons labeled generically. These buttons could be manufactured and labelled according to function to simplify product use.

# Appendices:

***a.) C Code***  
#include "msp.h"  
#include "keypad.h"  
#include "sin.h"

// Definitions for keypad inputs  
#define SQUARE 7  
#define SINE 8  
#define SAWTOOTH 9  
#define DUTYMINUS 10  
#define DUTYMID 11  
#define DUTYPLUS 12

// Cycles between interrupts for timing waveforms  
#define CYCLETIMER 15

// Maximum value to output to DAC  
#define MAXDAC 3700

// Function definitions  
void driveWave();

void manageInput();  
void Drive\_DAC(unsigned int level);

// Variable storage  
volatile unsigned int outHigh;  
int wave;  
int duty;  
int frequency;  
int squareTimeHigh;  
int squareTimeLow;  
int numInterrupts;  
int sawtoothMax;  
  
int main() {  
 WDT\_A->CTL = WDT\_A\_CTL\_PW | WDT\_A\_CTL\_HOLD; // Stop watchdog timer  
  
 // DCO = 24 MHz, SMCLK and MCLK = DCO  
 CS->KEY = CS\_KEY\_VAL;  
 CS->CTL0 = 0;  
 CS->CTL0 = CS\_CTL0\_DCORSEL\_4; // DCO = 24 MHz  
 CS->CTL1 = CS\_CTL1\_SELA\_2 | CS\_CTL1\_SELS\_3 | CS\_CTL1\_SELM\_3;  
 CS->KEY = 0;  
  
 // Configure port bits for SPI  
 P6->DIR |= BIT1; // Will use Port6 to activate /CE on the DAC  
 P1SEL0 |= BIT6 + BIT5; // Configure P1.6 and P1.5 for UCB0SIMO and UCB0CLK  
 P1SEL1 &= ~(BIT6 + BIT5);  
  
 // 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  
  
 TIMER\_A0->CCTL[0] = TIMER\_A\_CCTLN\_CCIE; // TACCR0 interrupt enabled  
 TIMER\_A0->EX0 = 7;  
 TIMER\_A0->CCR[0] = CYCLETIMER; //Set CCR0 for on cycle  
  
 TIMER\_A0->CTL = TIMER\_A\_CTL\_SSEL\_\_SMCLK | 0x10; // SMCLK, continuous mode  
  
  
 // Enable global interrupt  
 \_\_enable\_irq();  
  
 NVIC->ISER[0] = 1 << ((TA0\_0\_IRQn) & 31);  
  
  
  
  
  
 // Initialize keypad  
 keypad\_init();  
  
 // Set default wave type and parameters  
 wave = SQUARE;  
 frequency = 100;  
 duty = 50;  
 squareTimeHigh = duty \* 2000 / frequency;  
 squareTimeLow = duty \* 2000 / frequency;  
 sawtoothMax = 200000 / frequency;  
 numInterrupts = 0;  
  
 displayWaveform();  
  
 while (1) {  
 // Manage inputs and drive waveform infinitely  
 manageInput();  
 driveWave();  
 }

return 0;  
}  
  
/\*  
 \* Manages input from the keypad  
 \*/  
void manageInput() {  
 // Get input from keypad  
 char input = 0;  
 input = keypad\_getkey();  
  
 if (input != 0)  
 {  
 // debounce  
 delayMs(100);

// Check for 1-5 frequence changes  
 if (input >=1 && input <= 5) {  
 frequency = input \* 100;  
 squareTimeHigh = duty \* 2000 / frequency;  
 squareTimeLow = (100 - duty) \* 2000 / frequency;  
 numInterrupts = 0;  
 sawtoothMax = 200000 / frequency;  
 }  
  
 // handle wave/duty cases  
 switch (input) {  
 case SQUARE:  
 wave = SQUARE;  
 numInterrupts = 0;  
 break;  
 case SINE:  
 wave = SINE;  
 numInterrupts = 0;  
 break;  
 case SAWTOOTH:  
 wave = SAWTOOTH;  
 numInterrupts = 0;  
 break;  
 case DUTYMINUS:  
 if (duty > 10 && wave == SQUARE)  
 duty -= 10;  
 squareTimeHigh = duty \* 2000 / frequency;  
 squareTimeLow = (100 - duty) \* 2000 / frequency;  
 numInterrupts = 0;  
 break;  
 case DUTYMID:  
 duty = 50;  
 squareTimeHigh = duty \* 2000 / frequency;  
 squareTimeLow = duty \* 2000 / frequency;  
 numInterrupts = 0;  
 break;  
 case DUTYPLUS:  
 if (duty < 90 && wave == SQUARE)  
 duty += 10;  
 squareTimeHigh = duty \* 2000 / frequency;  
 squareTimeLow = (100 - duty) \* 2000 / frequency;  
 numInterrupts = 0;  
 break;  
 }  
 }  
}  
  
void driveWave() {

// Outputs square wave high or low  
 if (wave == SQUARE) {  
 if (outHigh)  
 Drive\_DAC(MAXDAC);  
 else  
 Drive\_DAC(0);  
 }

// Increases sawtooth according to number of interrupts since last reset  
 else if (wave == SAWTOOTH) {  
 int driver = MAXDAC \* numInterrupts;  
 driver = driver / sawtoothMax;  
 Drive\_DAC(driver);  
 }

// Output sine wave according to value in lookup table sinVal[]  
 else if (wave == SINE) {  
 int sinLoc = numInterrupts \* frequency / 100;  
 int driver = sinVal[sinLoc];  
 Drive\_DAC(driver);  
 }  
  
}  
  
// Timer A0 interrupt service routine  
void TA0\_0\_IRQHandler() {  
 TIMER\_A0->CCTL[0] &= ~TIMER\_A\_CCTLN\_CCIFG;  
 numInterrupts++; // Increase interrupt counter

// Handle wave resets for different cases  
 if (wave == SQUARE && outHigh && numInterrupts == squareTimeHigh) {  
 outHigh = ~outHigh;  
 numInterrupts = 0;  
 }  
 if (wave == SQUARE && !outHigh && numInterrupts == squareTimeLow) {  
 outHigh = ~outHigh;  
 numInterrupts = 0;  
 }  
 if (wave == SAWTOOTH && numInterrupts == sawtoothMax) {  
 numInterrupts = 0;  
 }  
 if (wave == SINE && numInterrupts == sawtoothMax) {  
 numInterrupts = 0;  
 }  
}  
  
void Drive\_DAC(unsigned int level){  
 unsigned int DAC\_Word = 0;  
 int i;  
  
 DAC\_Word = (0x3000) | (level & 0x0FFF); // 0x1000 sets DAC for Write  
 // to DAC, Gain = 1, /SHDN = 1  
 // and put 12-bit level value  
 // in low 12 bits.  
  
 P6->OUT &= ~BIT1; // Clear P4.1 (drive /CS low on DAC)  
  
 EUSCI\_B0->TXBUF = (unsigned char) (DAC\_Word >> 8); // Shift upper byte of

// DAC\_Word 8-bits to right  
  
 while (!(EUSCI\_B0->IFG & EUSCI\_B\_IFG\_TXIFG)); // USCI\_A0 TX buffer ready?  
  
 EUSCI\_B0->TXBUF = (unsigned char) (DAC\_Word & 0x00FF); // Transmit lower

// byte to DAC  
  
 while (!(EUSCI\_B0->IFG & EUSCI\_B\_IFG\_TXIFG)); // Poll the TX flag to wait

// for completion  
  
 for (i = 200; i > 0; i--); // Delay 200 16 MHz SMCLK periods to ensure TX is

// complete by SIMO  
  
 P6->OUT |= BIT1; // Set P4.1 (drive /CS high on DAC)  
}

***b.) References***

KeypadPinout: Provided by Professor Gerfen via PolyLearn

Bill of Materials: Provided by Professor Gerfen via PolyLearn shared through Cal Poly IEEE

DAC Datasheet: *Microchip MCP4901/4911/4921*; Microchip; January, 5, 2010.