ELC 343

Lab 4 - Data Conversion: Analog-to-Digital and Digital-to-Analog

Brian Worts and Chris Jenson

11/8/19



**Table of Contents**

Introduction ………………………………………………………………………………. 1

Procedure and Design ………………………………………………………….………… 1

Part 1 …………………………………………………………………………...… 1

Part 2 ……………………………………………………………………………... 5

Discussion ………………………………………………………………………………... 16

Code ……………………………………………………………………………………… 16

**Introduction**:

The purpose of this lab is to become familiar with the digital-to-analog and analog-to-digital converters on the CY8CKIT-050 PSoC 5LP Development Kit Board. These converters can be used for many applications such as reading sensors, controlling actuators, gauges or lights, or enabling real time signal processing. To sample the bandwidth of a continuous time signal into a discrete set of values and reconstruct, use Nyquist theorem. The sampling frequency is at least twice the frequency to the highest component present in the component.

This lab gives a look at the effects of sampling and aliasing, and reconstruction filtering. The components used are an analog-to-digital converter(ADC) and digital-to-analog converter(DAC) which are built into the PSoC chip. This allows for the visualization of the sampling and reconstruction of low amplitude signals, and signals close to, and over, the Nyquist limit. Statistics will be gathered for a 1KHz periodic signal and use those to discern whether the input wave is a square, sine, or triangle wave and display the results.

**Procedure and Results**:

Part 1

The first step of this lab was to use paper and pencil analysis of the fourier series representation of a square and triangle waves at 1KHz. The corresponding Fourier series can be found in **Equation 1** and **Equation 2**, where L is the period. Sampling at 100KHz would be expected to preserve the signal integrity because of Nyquist’s theorem. The theorem says that one needs at least two times the frequency and that is 100 times the frequency. Given the peak amplitude A, the normalized rms for each signal type was theorized. The individual rms values are in **Equation 4, 5, 6**. This lab required a general rms formula in order for the system to automatically detect the input signal, the summation in **Equation 3** was used.

Square Fourier Series = **Equation1**

Triangle Fourier Series = **Equation 2**

Discrete RMS = **Equation 3**

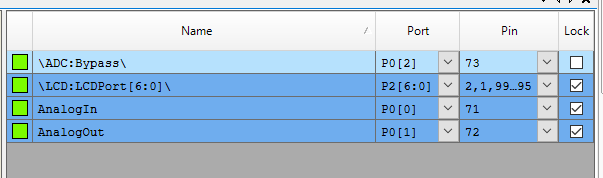
Sine RMS = **Equation 4**

Square RMS = **Equation 5**

Triangle RMS = **Equation 6**

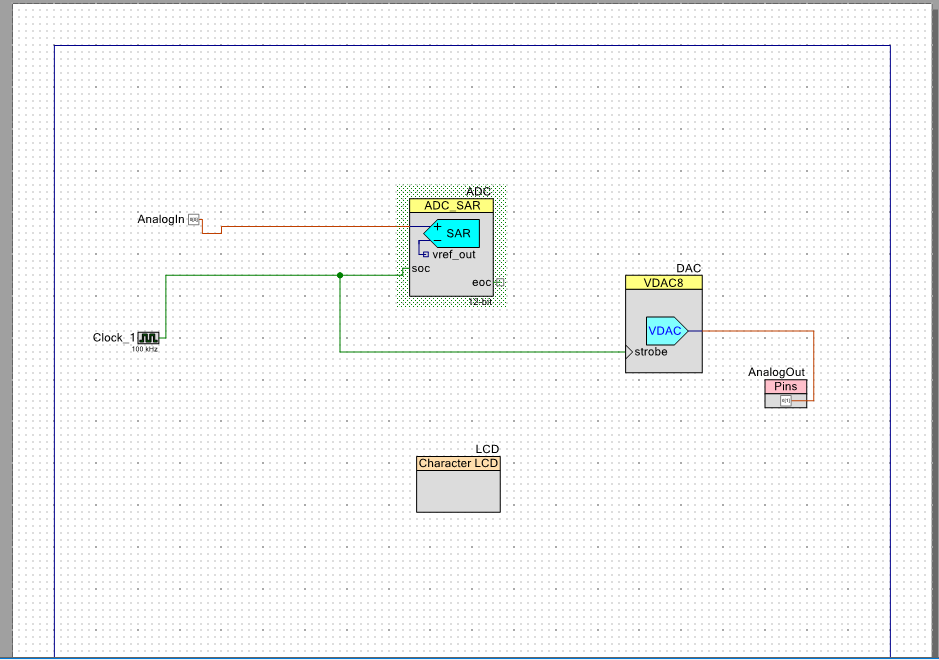
Part 2:

To begin part 2, the various components available were analyzed. An ADC\_SAR and a VDAC component were added to the project. An analog input pin was also added in order to feed the board signals from the virtual machine through P0[0]. Special care was taken to avoid using the pins P0[2] and P0[4] because those pins have 1μF capacitors on the pins. The pin mapping for the project is shown in *Figure 1*.



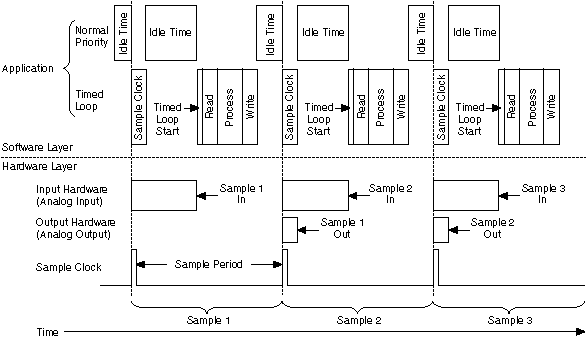
*Figure 1: Pin Mapping of the project*

For the ADC, the input voltage range was set to 0 through 2.048 volts and the internal reference set to bypassed. The ADC was configured for 12 bits and a conversion rate of 600Ksps. The sample mode was set to hardware trigger and the clock was provided to the soc pin. For the VDAC, the settings used were high-speed, datasource:CPU, and the clock connected to strobe external. The top design for the project is shown in *Figure 2.*



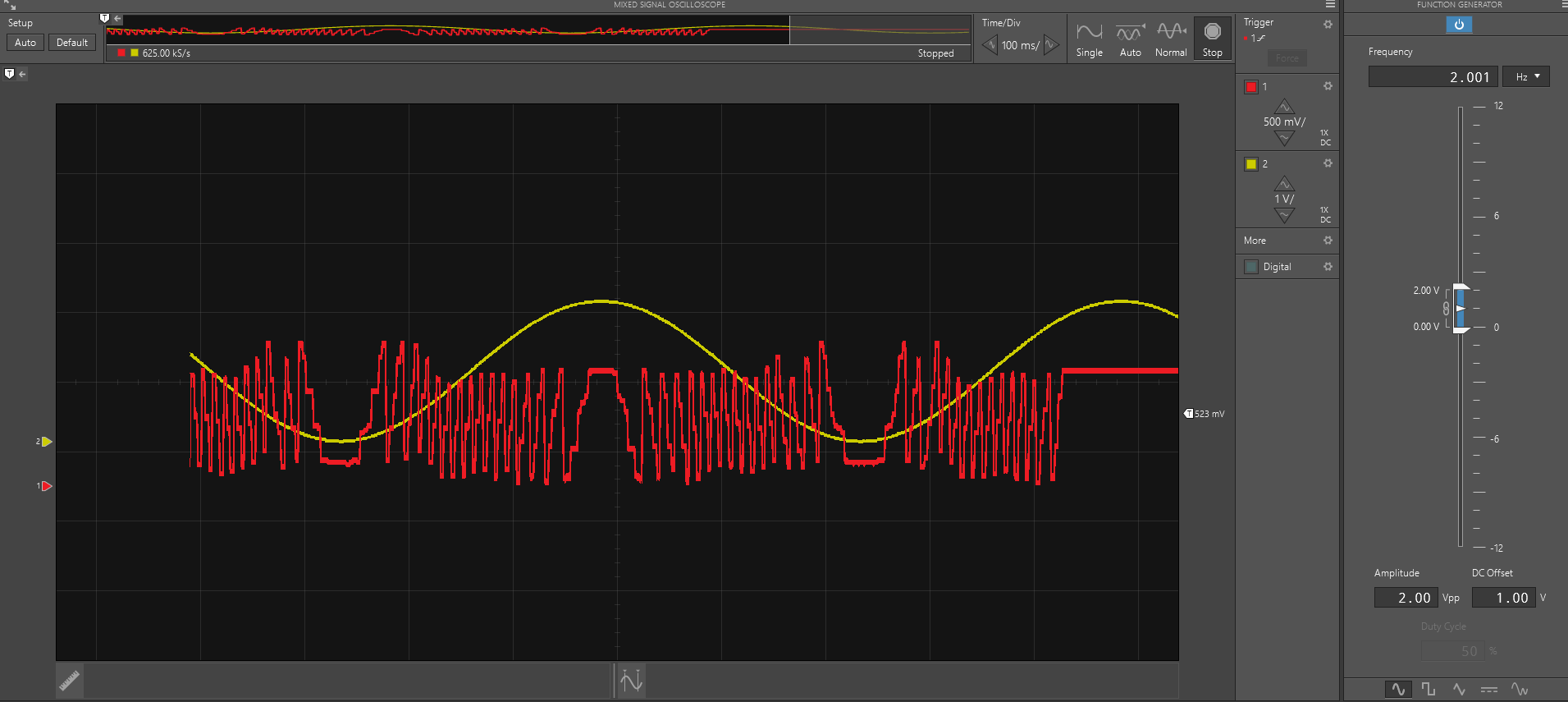
*Figure 2: Top Design of the project*

The program now had to be coded. The start of the code is a lot of variable declarations that will be needed for the various calculations used throughout. The for loop was set to run 1000 times for testing, but for actual runs should be set to 100,000 as per the lab instructions. Within the for loop, the code first did the ADC conversion. This used the **IsEndConversion()** call which could be blocking or unblocking. Blocking was used so that the value was not passed into the DAC before it could be converted. To get the converted value, **Get\_Result16()** was used which returns a signed 16-bit conversion result that was then converted to millivolts. The **SetValue** function input of the DAC should be represented as a value between 0-255 within the given range. After the conversion, the code would check for new min and max values, update the mean value, sum value, and the sum of squares value. Once out the loop, the mean squared is calculated, it will be used to determine the type of wave inputted. A timing diagram shows the clock, EOC signal, polling loops. ADC read and DAC write events, and DAC output. An example of such is shown below in *Figure 3.*

**

*Figure 3: Timing Diagram*

Using test runs, ranges of acceptable rms values were found for each input signal type. The values were found to be in a tight and consistent range which meant comparisons could be made to determine the type of input signal. The code used is shown in *Appendix A*. An int64\_t had to be used for sum of squares because of the potential of overflow associated with adding together so many samples. The waveforms were tested at 2Vpp, 1Vpp, and 0.2pp. All of the waveforms and their corresponding output on the LED screen are shown in *Figure 4.*





Sine wave with 2Vp, 100ms/500mV



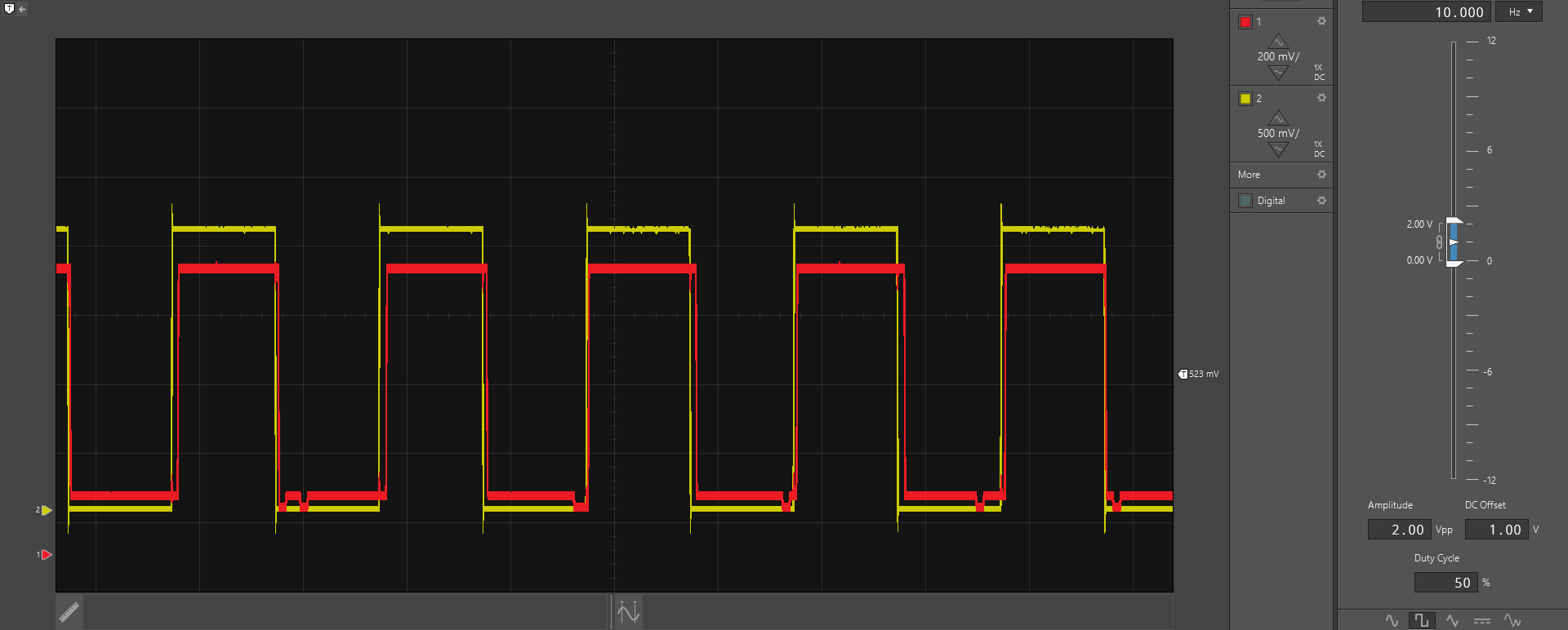


Sine wave with 1Vp, 100ms/500mV



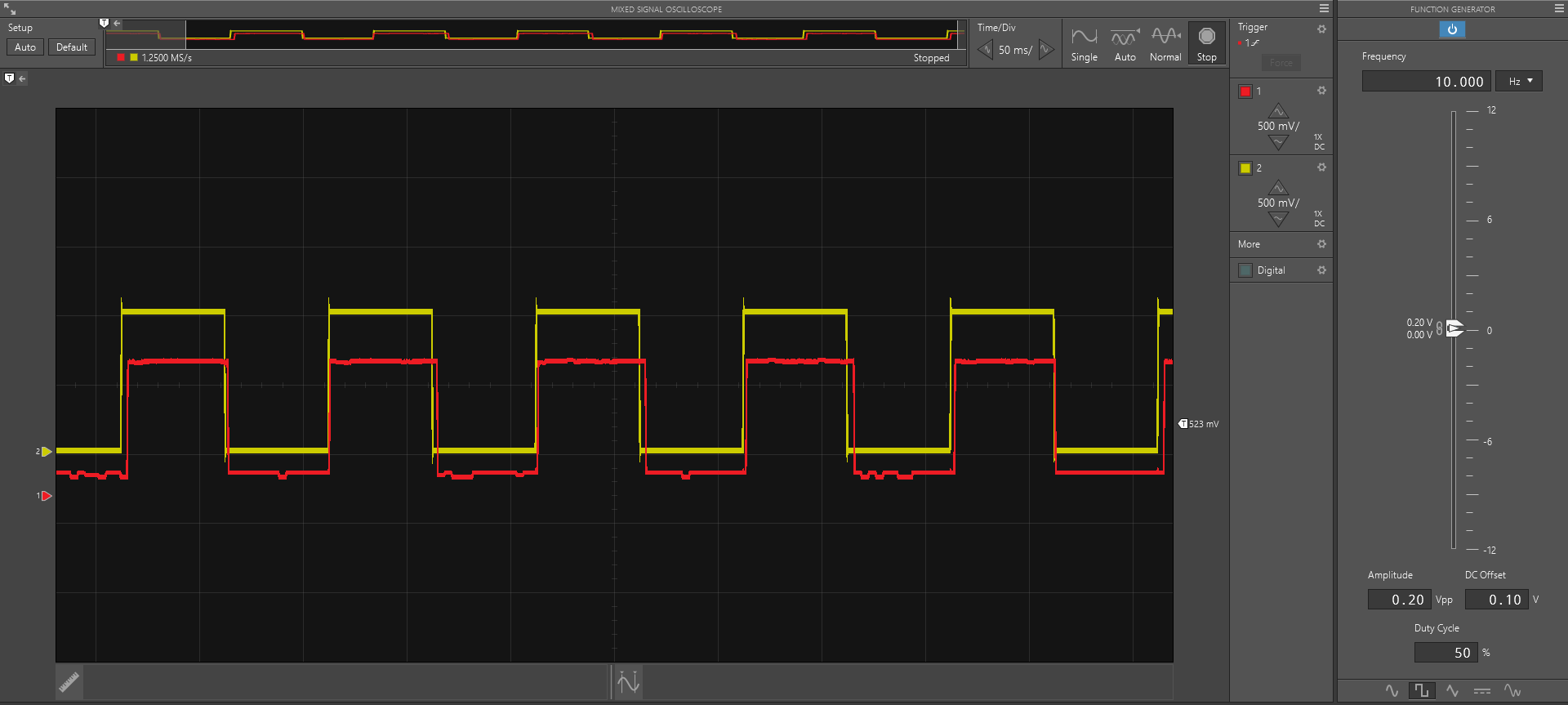


Sine wave with 0.2Vpp, 100ms/500mV (Unable to determine wave)





Square wave with 2Vpp, 50ms/200mV





Square wave 1Vpp, 50ms/500mV





Square wave with 0.2Vpp, 50ms/500mV (incorrect)



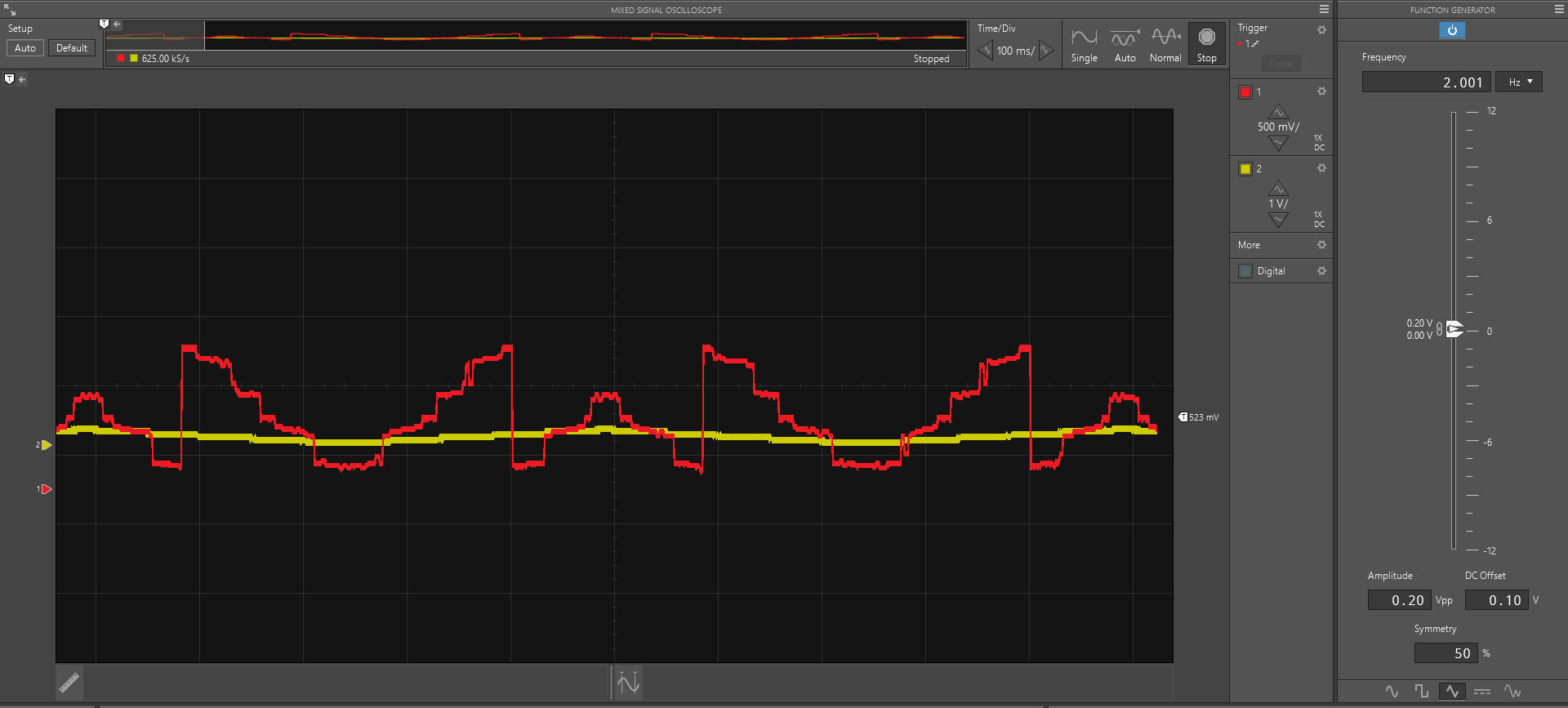


Triangle wave with 2Vpp, 100ms/500mV





Triangle wave with 1Vpp, 100ms/500mV





Triangle wave with 0.2Vpp, 100ms/500mV

*Figure 4: The various waveforms, yellow = fgen, red = dac*

**Discussion**:

PSoC offers a variety of useful components when working on the top down schematic. For this lab, where a signal has to be input into the system, a signal generator was attempted. The signal generator was able to create several different types of waveforms, but was unable to connect the ADC. Because of this, the virtual machine had to be used. With the virtual machine, the voltage range had to be set only on the positive side as the PSoC components are not equipped to handle negative values. Of the resulting waveforms, only the square wave had a clear output signal. This may be because of the sampling rate. The conversion was not fast enough for the other signals to reproduce an attractive output signal. Since a square wave is just highs and lows, the conversion is more accurate due to there being only slopes of zero.

**Code**:

#include "project.h"

#include "stdio.h"

#include "math.h"

int main(void)

{

CyGlobalIntEnable;

//The starting of the ADC, DAC, and Screen

ADC\_Start();

DAC\_Start();

LCD\_Start();

//Display a start message to the user

LCD\_Position(0,0);

LCD\_PrintString("START");

CyDelay(1000);

//initialize variables used for calculations

int mean = 0;

int sum = 0;

int max = 0;

int min = 5000;

int voltage;

int value;

int count = 0;

int sumOfSquare = 0;

//char arrays to hold hold results printed to screen

char str1[15];

char str2[15];

char str3[15];

char str4[15];

//64 bits because of how large the number will be

int64\_t mean\_sq;

//runs 1000 times, run 100,000 when not testing

while(count < 1000)

{

//pause here until the conversion finishes

ADC\_IsEndConversion(ADC\_WAIT\_FOR\_RESULT);

//get the result as signed 16-bit, convert to mV, pass to DAC

voltage = ADC\_GetResult16();

value = ADC\_CountsTo\_mVolts(voltage);

DAC\_SetValue(voltage);

//comparisons to keep min and max updated

if (value > max)

{

max = value;

}

if (value < min)

{

min = value;

}

//update sum and mean every iteration

sum = sum + value;

mean = sum / count;

sumOfSquare = sumOfSquare + ((value - mean)\*(value - mean));

//1 iteration every ms and increment count

CyDelay(1);

count = count + 1;

}

//calculate the mean square with the final values

mean\_sq = sumOfSquare;

mean\_sq = mean\_sq/(((max - min)/2)\*((max - min)/2));

LCD\_ClearDisplay();

//display the max as “M”

LCD\_Position(0,0);

sprintf(str2, "M:%d", max);

LCD\_PrintString(str2);

//Display the min as “m”

LCD\_Position(0,7);

sprintf(str3, "m:%d", min);

LCD\_PrintString(str3);

//display the mean as “v”

LCD\_Position(1,0);

sprintf(str4, "v:%d", mean);

LCD\_PrintString(str4);

//display the mean square as “q”

LCD\_Position(1,7);

sprintf(str1, "q:%d", (int) mean\_sq);

LCD\_PrintString(str1);

//check for sine section

//test range values: 506, 507, 526

if ((mean\_sq <= 550) && (mean\_sq >= 450))

{

LCD\_Position(1,15);

LCD\_PrintString("S");

}

//check for Square section

//test range values: 991,994,982

else if ((mean\_sq <= 1100) && (mean\_sq >= 900))

{

LCD\_Position(1,15);

LCD\_PrintString("Q");

}

//check for Triangle section

//test range values: 334, 345, 345

else if ((mean\_sq <= 400) && (mean\_sq >= 300))

{

LCD\_Position(1,15);

LCD\_PrintString("T");

}

else //if the value meets none of the criteria then display a ? mark

{

LCD\_Position(1,15);

LCD\_PrintString("?");

}

}//end main