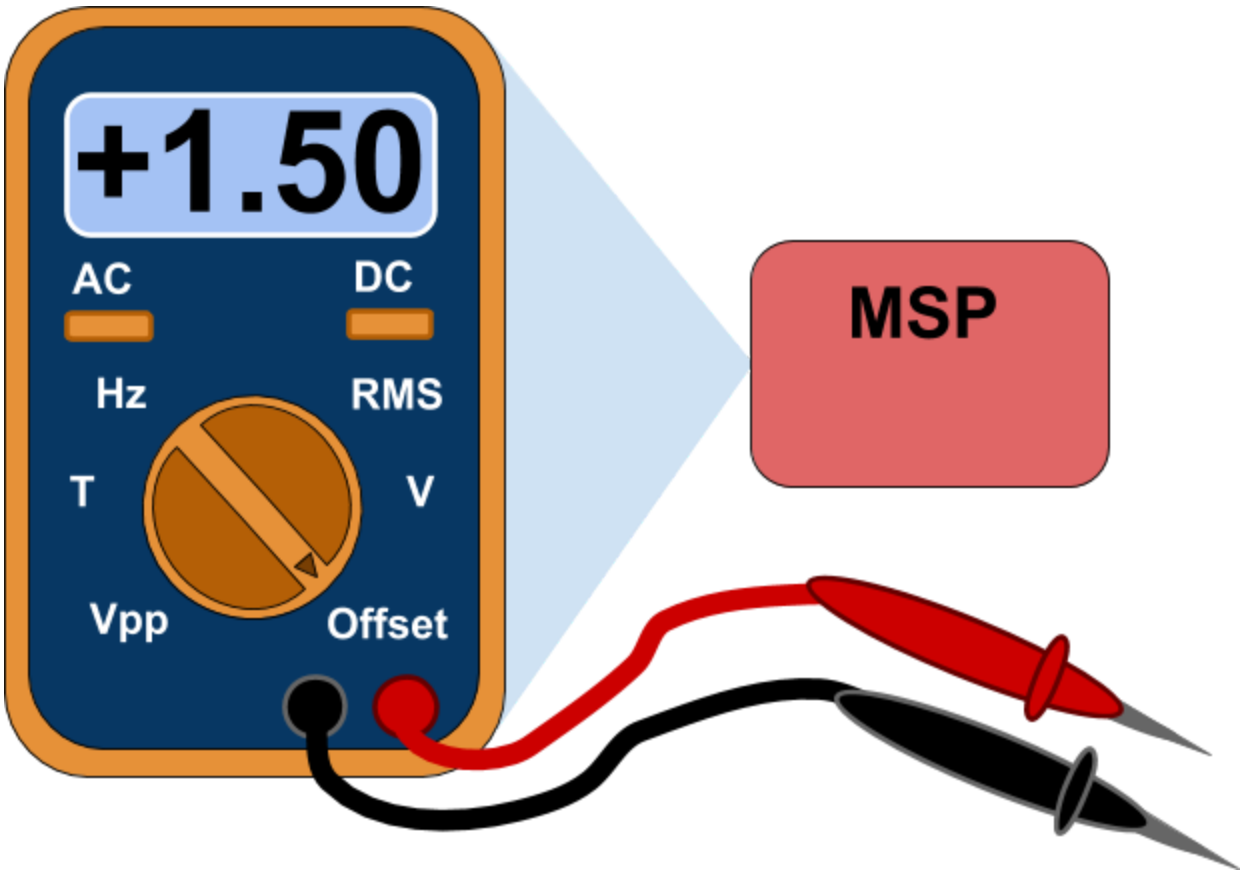
# Project 3
# Multimeter Development

Andrew McGuan, Denis Pyryev
CPE 329-01, Spring 2017
Prof. Gerfen
May 19, 2017



### *Demonstration:*

***Youtube link:*** Part 1: https://youtu.be/MbytUpmEI18
Part 2: https://youtu.be/lQxfaEbxyjw
Part 3: https://youtu.be/_gFYW-HRSHw

***Note:** there are three youtube videos because the phone being used kept running out of storage space, and would stop the video. We apologize for the inconvenience.

***Youtube link for Extra Implementation:*** https://youtu.be/DID5nXIyV-4

## *Project #3 : Function Generator:*

### *Purpose:*

The purpose of this project is to ensure that the students are familiar with the operation of an analog to digital converter, specifically the 14-bit ADC present inside the MSP432. Because exercises with a analog to digital converter have already been completed at this point, the basic operation of an ADC should be fairly intuitive. The method to test this knowledge was with the creation of a multimeter, using the MSP432. The ADC can take samples across the range of 0 to 3.3V, and convert them into a digital signal. It is up to the students doing this project to analyze the pattern of ADC values and identify various characteristics of the signal, such as DC offset, frequency, and V-rms.

Additionally, this multimeter must be able to communicate with a serial terminal on a computer to print the multimeter values in a legible display. This communication will take place via UART communication on the eUSCI port, which in this case is the micro-USB port used for debugging. The student should be familiar with sending and receiving information from a serial port terminal via UART due to past exercises. They must now be familiar with VT100 escape codes to be able to format the output and print to specific places on the terminal.

As an added feature, the RGB LED was utilized to output a certain color for a corresponding input waveform. For a rectangular input signal, the LED will output a green color, with a sinusoidal or triangular wave, a purple color, and for a DC voltage input, a blue color. In addition, the terminal will display the wave type out for the user in addition to all of the data that is already displayed.

*Final Product:*

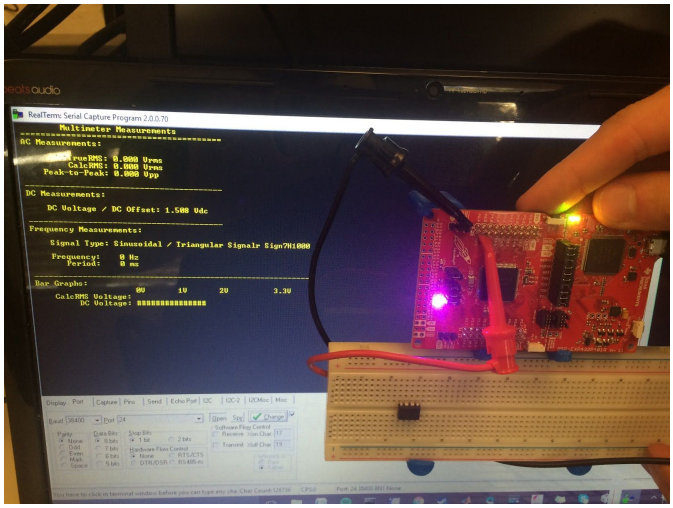



**Figure 1: Picture of the completed final product**

## *System Requirements:*

1. The multimeter shall measure voltage.
    a. The voltage measurements, both DC and AC, shall be within the range of 0 - 3.3 V, and be accurate for +/- 1 mV.
    b. The multimeter shall have a DC setting.
        i. DC measurements shall be recorded by the average value across a 1 ms or longer timer period.
        ii. The DC measurement of a sinusoid waveform shall be the equivalent to the DC offset of the sinusoid.
    c. The multimeter shall have an AC setting.
        i. AC measurements shall be given in true-RMS.
        ii. AC measurements shall be broken down into the following components:
            1. TrueRMS (DC offset included)
            2. CalcRms (The RMS with DC offset subtracted from each value)
            3. Vpp
        iii. AC shall be measureable for the following waveform types:
            1. Sinusoidal
            2. Triangular
            3. Square
            4. All other periodic waveforms
        iv. AC measurements shall be accurate for the following range of DC offset and amplitudes:
            1. The maximum measurable voltage shall be 3 V

2. The minimum measurable voltage shall be 0 V
3. The smallest Vpp measureable shall be 0.5 V
4. The largest DC offset values measurable shall be 2.75 V

2. The multimeter shall measure frequency of periodic waveforms.
    a. The range of frequency measurements shall be between 1 and 1000 Hz.
    b. Frequency measurements shall be accurate to within 1 Hz.
    c. The following waveform types shall have their frequency measured by this multimeter:
        i. Sinusoidal
        ii. Triangular
        iii. Square
        iv. Other periodic waveforms
3. The values measured by this multimeter shall be printed onto a computer terminal.
    a. The terminal shall operate at a frequency of 9600 baud or greater.
    b. The terminal shall utilize a VT100 protocol.
        i. The terminal shall display all multimeter values in singular locations, with no scrolling values.
    c. The terminal shall display the following AC and DC values:
        i. TrueRMS
        ii. CalcRMS
        iii. Vpp
        iv. DC offset
        v. Frequency
        vi. Period
    d. The terminal shall have borders or dividers to organize the data display.
    e. The terminal shall use bar graphs to display voltages.
        i. The following voltages shall have a bar graph:
            1. CalcRMS
            2. DC offset
        ii. The bar graphs shall have a scale that gives a reference to the current voltage.
        iii. The bar graph shall range from 0 - 3.3 V, and their length shall be proportional to the measured voltage.
        iv. The bar graph shall be a singular horizontal line of a specific character.
        v. The bar graphs shall be updated in real time to reflect the current voltage levels.
4. The source code for this project shall make use of the #define constants from the file *msp432p401r.h.*

5. As an added feature, the program shall be able to differentiate between when a wave is a sinusoid, triangle, square, or exclusively DC.
    a. If the signal is sinusoidal or triangular, the tri-color LCD shall turn purple.
    b. If the signal is rectangular, the tri-color LCD shall turn green.
    c. If the signal is DC, the tri-color LCD shall turn blue.
    d. The current waveform type shall be printed out to the terminal.
6. Additionally, the period of the input waveform shall be printed to the terminal.

## *System Specifications:*

| *Component:* | *Specification:* | *Value:* |
|---|---|---|
| **MSP432P401R** | Supply voltage | 3 V |
| | Supply current | 0.75 mA |
| | Supply power | 2.25 mW |
| | Clock frequency | 48 MHz |
| | Internal flash memory | 256 KB |
| | External pins in use | 3 |
| | Integrated RGB LED | Yes |
| | On-board ADC | Yes |
| | ADC Bit Count | 14 |
| | ADC Max Voltage | 3.3 V |
| | ADC Min Voltage | 0 V |
| | ADC Current Draw | 490 µA |
| *MSP432 specs taken from MSP datasheet | | |

**Table 1: Specifications for the microprocessor**

## *System Architecture:*

### *Overall System:*

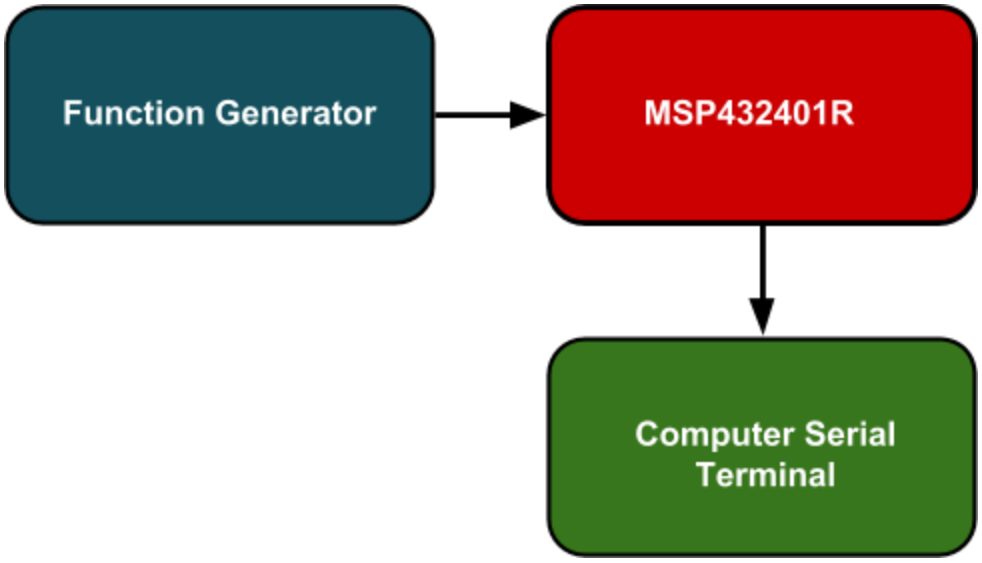



**Figure 2: Overall System Block Diagram**

***Software:***

*Main Flowchart:*

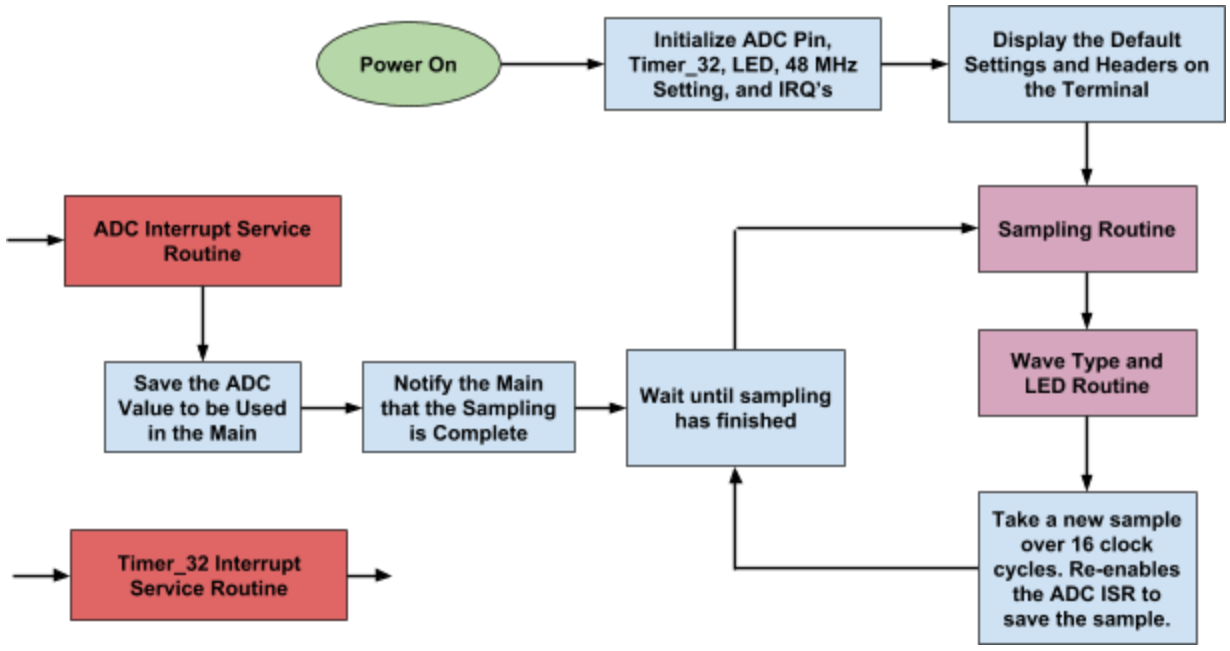



**Figure 3: Overall software flowchart showing the main code blocks and decisions.**

*Sampling Routine Flowchart:*

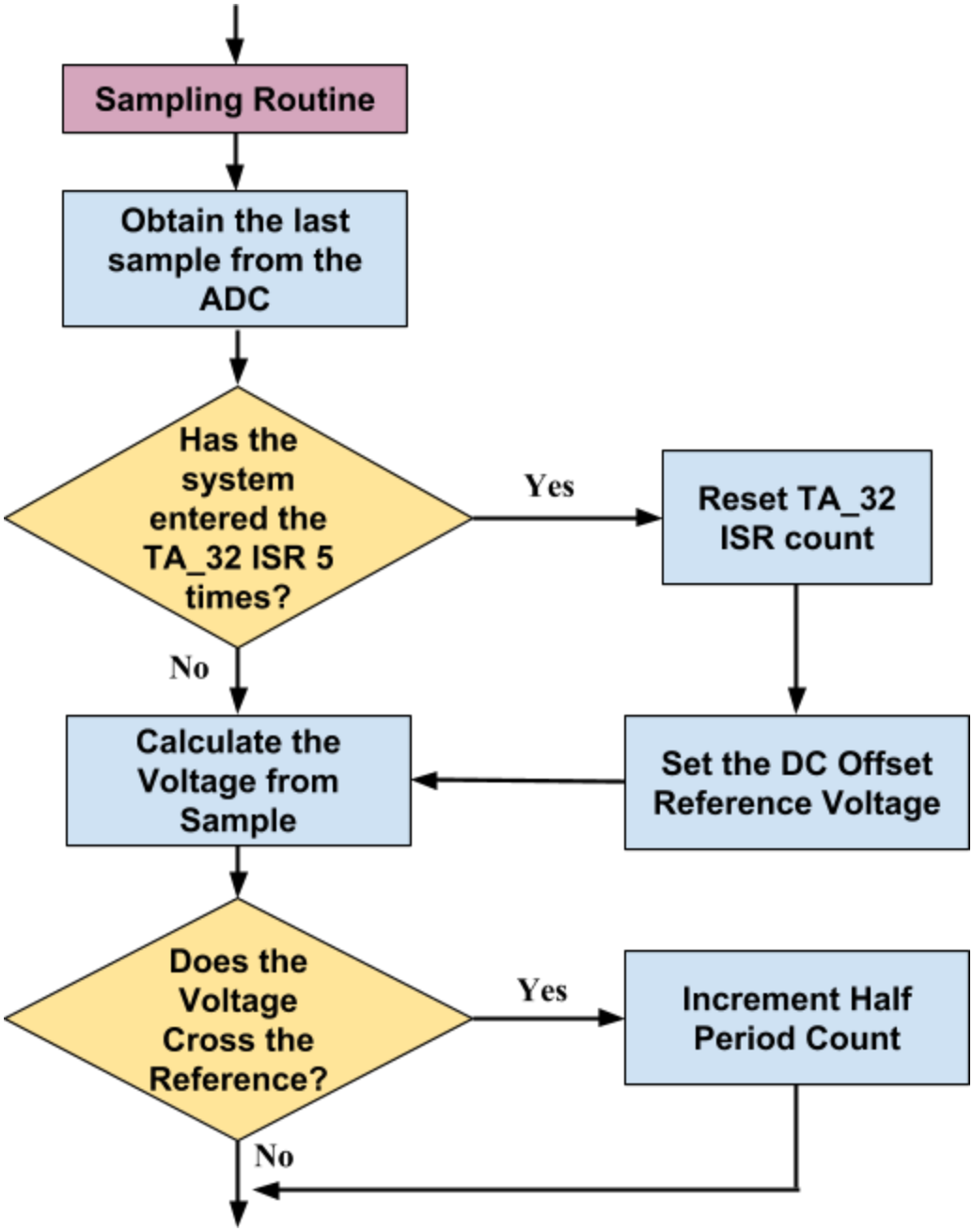



**Figure 4: Flowchart showing the steps to process a sampling value from the ADC**

*Wave Types and LED Routine:*

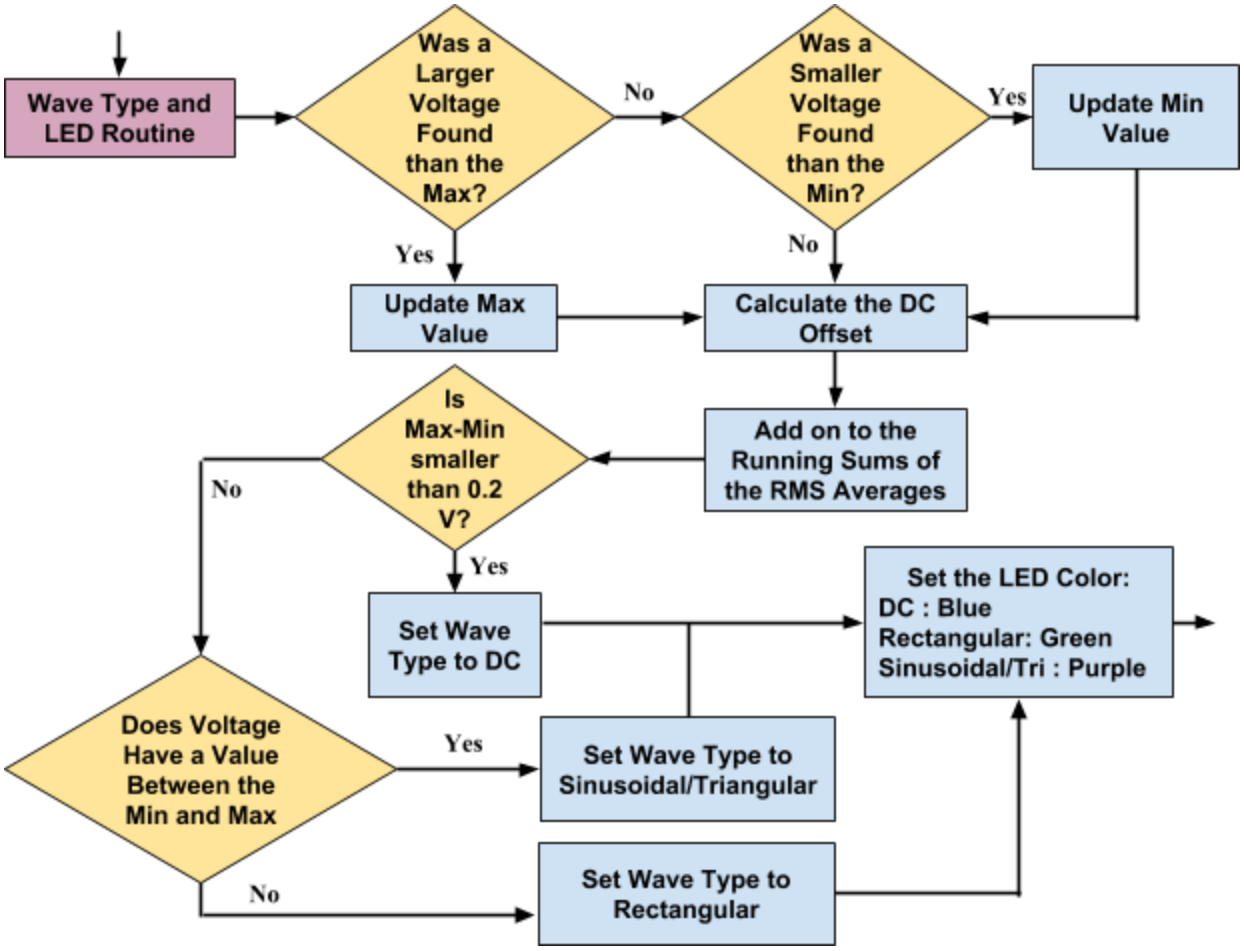



**Figure 5: Flowchart continuing to process the ADC Value as well as determining what kind of signal is being input and configuring the LED**

*Timer_32 Interrupt Service Routine:*

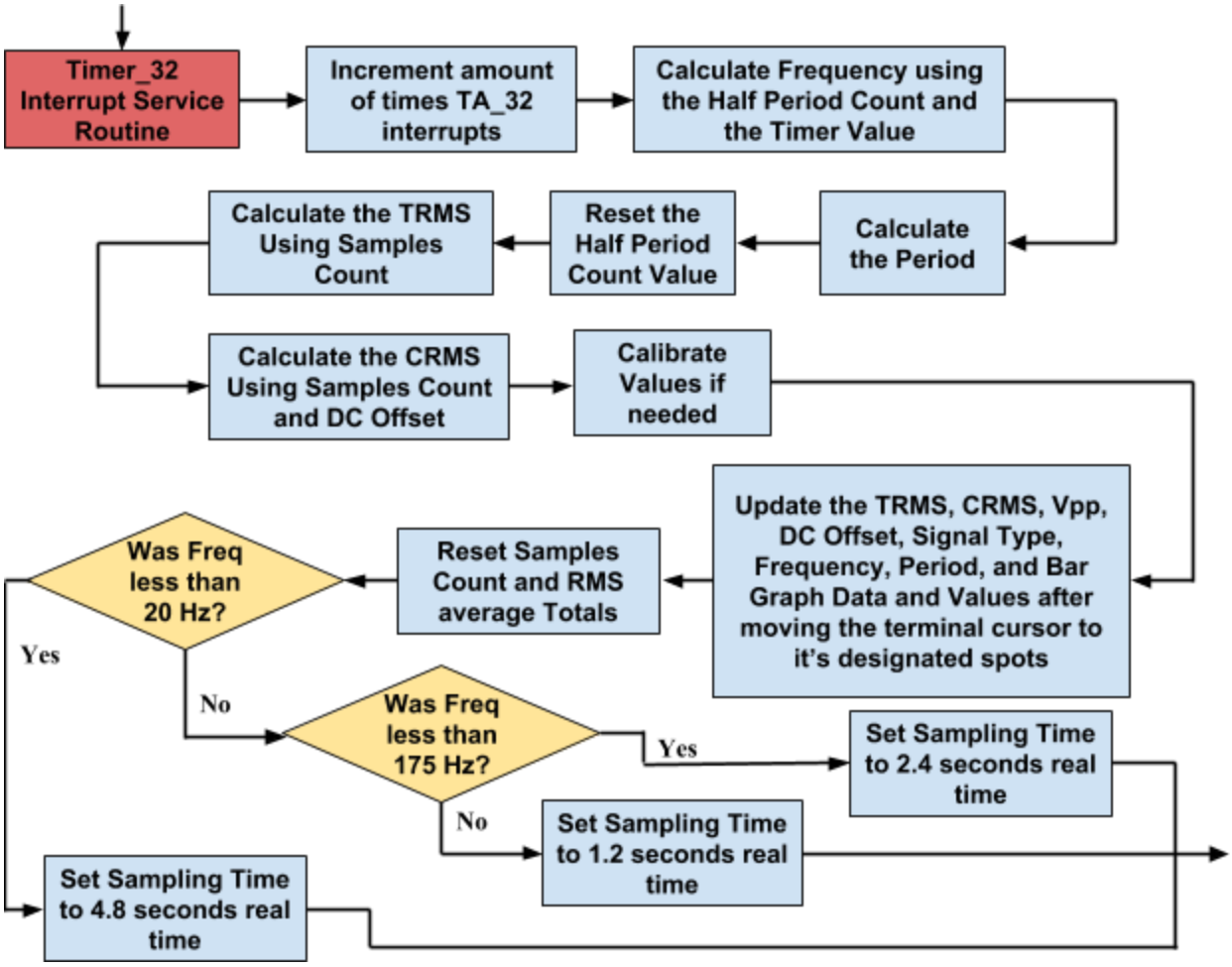



**Figure 6: The Interrupt Service Routine of the Timer that handles the display updates for the AC/DC values**

## *Component Design:*

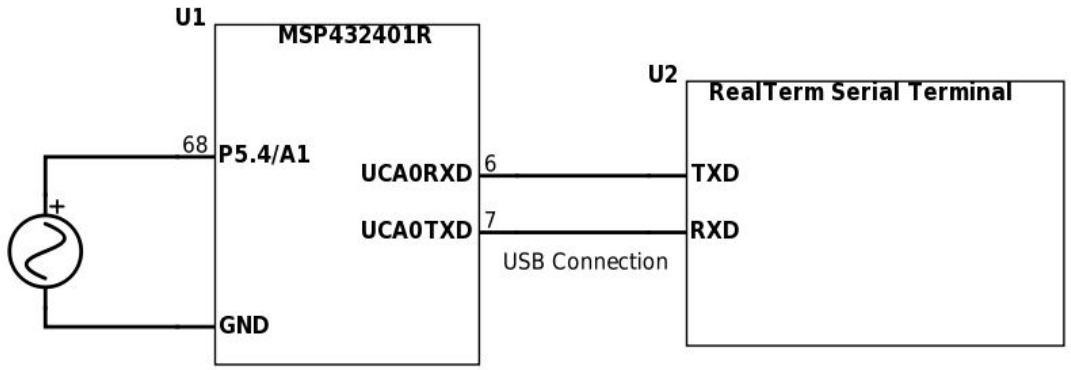



**Figure 7: Schematic showing the final circuit configuration**

The key to getting accurate measurements in this project lies in the timing. It is advantageous to lump all the time intensive calculations into one block of code. When these calculations are spread out among the rest of the code, it brings a level of variability into the code execution time. In this project, a timer 32 is used to generate an interrupt every 1.2 seconds. During this period of time, the ADC is continuously gathering new samples. Quick calculations can be performed after every sample, such as comparing the new value to the max and minimum, or squaring the input value and adding it to a running sum of squares, that will be later turned into the RMS values. The amount of times the input waveform crosses over the DC offset is also counted, and will be used to count the frequency. The algorithm for identifying a crossover is described later on. Once 1.2 seconds has passed, timer 32's ISR takes care of the longer calculations. Floating point operations, such as the calculation of frequency and period, take place here. The values to be displayed are loaded into an array, and the array is sent to the function that updates the values on the terminal. This operation is also time intensive, which is why it is done only once every 1.2 seconds. Shown below in Figure 8 is the scope trace of the sampling period. Each jitter is an individual sample being taken, and the periods with no samples is the calculation and UART transmission time. A more detailed picture showing the time between each ADC sample is also shown below in Figure 9.

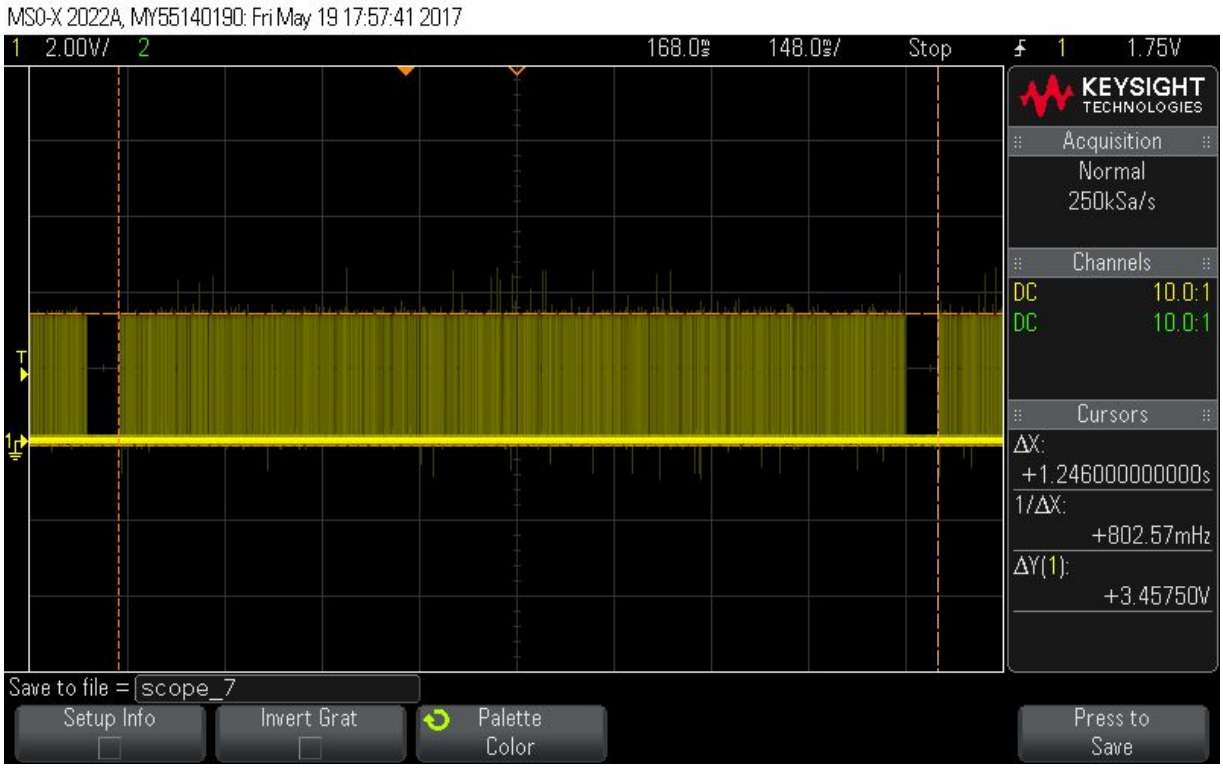



**Figure 8: Period of UART Values Update**

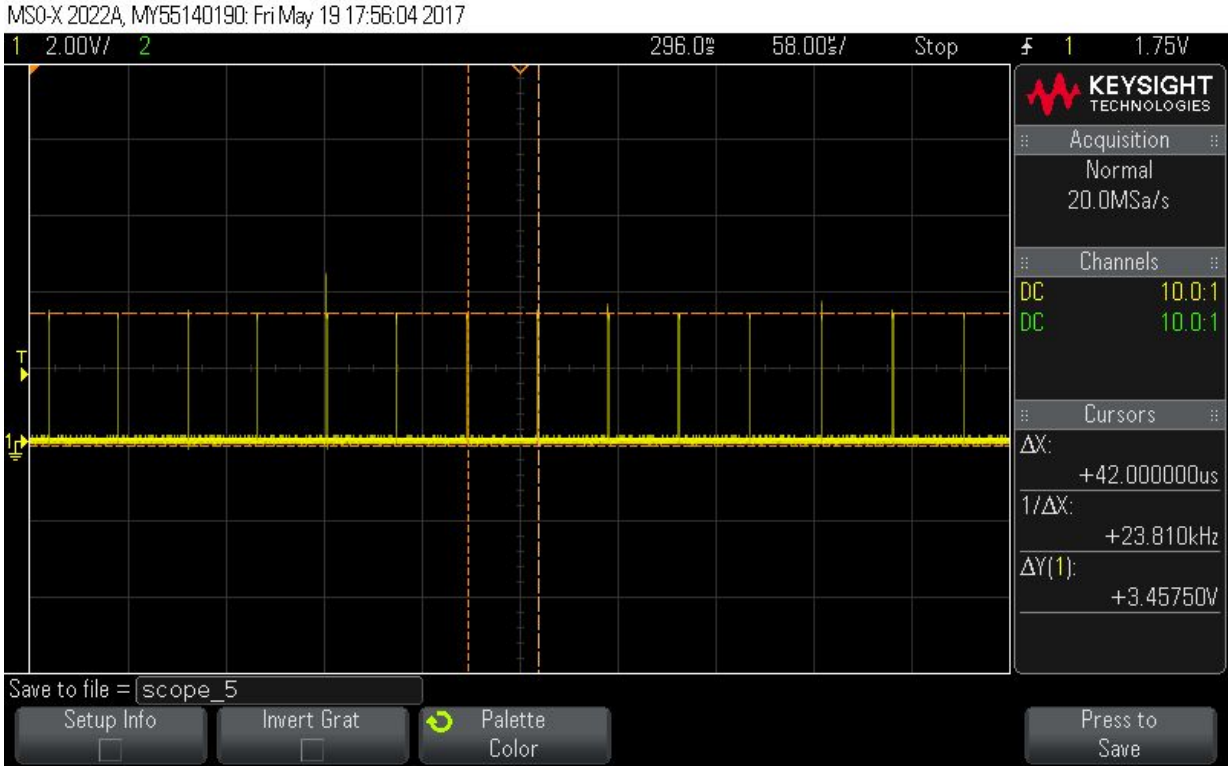



**Figure 9: The Sampling Rate of the ADC**

Moving all time-intensive operations into the ISR made the program much more accurate. In a previous code iteration, all terminal values, including the rms and period, were calculated inside the main function after 20,000 samples were recorded, and they were sent to the terminal via UART shortly after. Separating the code in this way made the measured frequency to be wildly inconsistent, varying by up to 20 Hz in some cases. This is caused by an inconsistency in the amount of samples being taken every 1.2 seconds. One call to the function that sends values to the terminal takes 50.8 ms, as shown below in Figure 10. Samples are taken roughly every 40 us, so this is roughly 1270 samples that are ignored. In the case of higher frequencies, the input voltage could have oscillated past the DC offset many times and the program would have missed every one of them. A UART transmission could potentially be made twice during one timer 32 sampling period, so the number of edges could vary dramatically depending on when these two processes restarted.

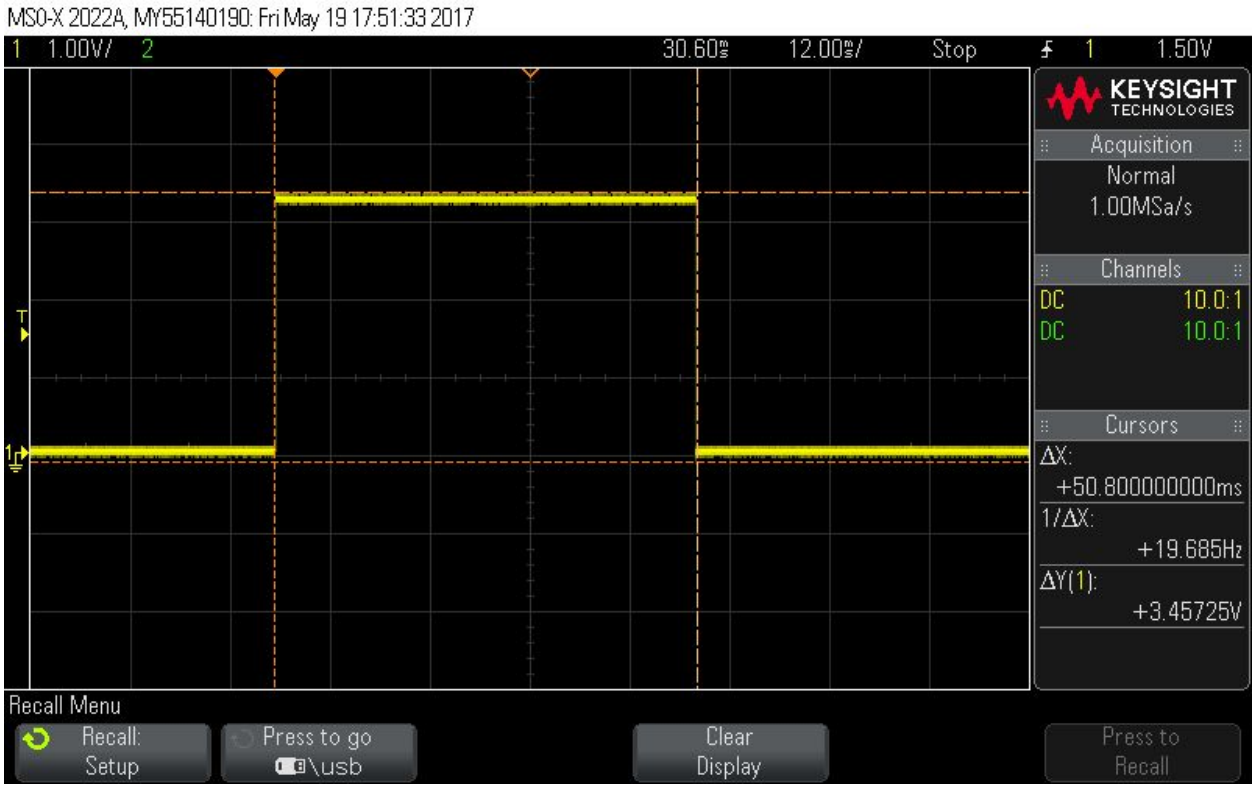



**Figure 10: Update UART function time**

Identifying a point where a input voltage crossed over the DC offset line was a large obstacle to calculating a steady frequency. To identify one of these points, the last voltage read and the current one read were both compared to the DC offset. If the last was greater than it and the current is less than it, it is on a falling edge, and vice versa for a rising edge. One thing learned from past projects is that the output from the ADC is not steady: its value in decimal can vary by up to 100. An epsilon-equals was used to combat this variance, and a buffer of 50 mV

was written into the conditional statement to allow the ADC values room for error. However, this led to another problem. For lower frequencies, they could potentially have multiple values that fall in this epsilon region, which would cause the program to think multiple edges have been found. To combat this, an enable flag was created. When the flag is 0, the program will check for edges as normal, but when it finds one, it sets the flag to 1, temporarily disabling edge checking. The disable equal flag will only return low once both the current and previous voltage values are confirmed to have left the epsilon region around DC offset. The small segment of code that makes this run is shown below. The variable de_flag is the disable equals flag, and half_periods is the running counter of edges, with each edge indicating half a period has passed.

```
/* If the voltage has passed the halfway threshold, checks for both rising or
 * falling edge */
if ((de_flag == 0) &&
  ((last_voltage+epsilon > freq_ref && voltage-epsilon < freq_ref) ||
  (last_voltage-epsilon < freq_ref && voltage+epsilon > freq_ref))){
      de_flag = 1;
      half_periods++;
}

/* Verifies that the current voltage has left the epsilon range before looking
 * for another rising/falling edge */
else if((de_flag==1) &&
  ((last_voltage+epsilon < freq_ref && voltage+epsilon < freq_ref) ||
  (last_voltage-epsilon > freq_ref && voltage-epsilon > freq_ref))){
      de_flag = 0;
}
```

Writing the resulting values to the terminal was the final step in this project. Although the UART communication had already been learned in previous projects, using the VT100 terminal commands to control the placement of the text was a new step. Each VT100 command takes up 8 lines of code (4 characters to send, plus a while loop to wait for the UART transfer to complete) so functions were written to make this process go much smoother. Functions were made to move the cursor to a specific location on the screen, write a string of text, send a command to the terminal, and draw a bar graph at a specified location and length. The use of these functions made the code written in other parts of the file look much cleaner and more readable. The terminal display setup is shown below in Figure 11.

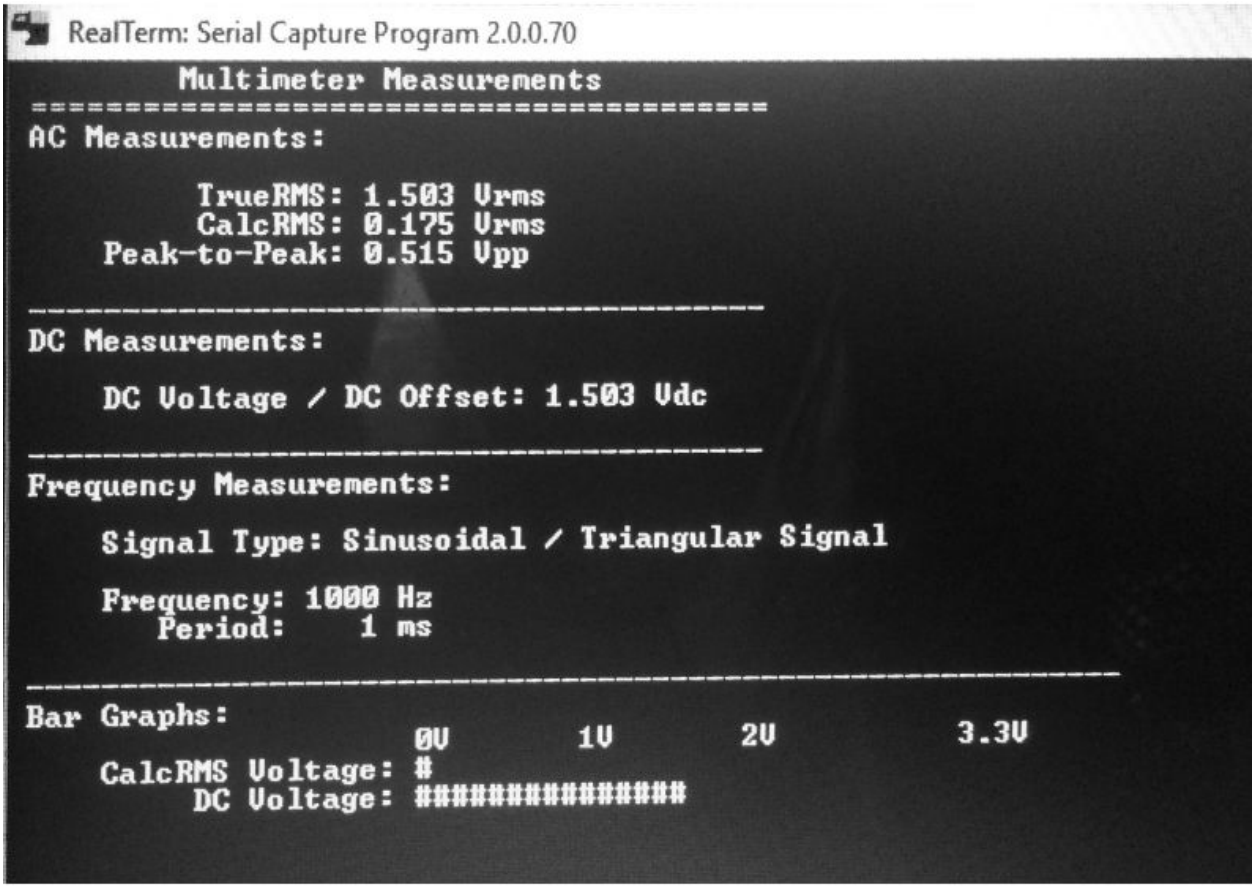



**Figure 8: An Example of the Terminal Display**

Calculating the RMS of the signal was the most complex calculation of all the values required by this project. There is two RMS values being calculated in this project: TrueRMS, which uses the ADC output for the RMS calculation, and the CalcRMS, which subtracts the DC offset from the ADC output value before inputting it in the RMS function. The formulas for both calculations are shown below.

$$TrueRMS = \sqrt{\frac{1}{n}\sum_{i=0}^{n} x_i^2}$$

$$CalcRMS = \sqrt{\frac{1}{n}\sum_{i=0}^{n} (x_i - DC)^2}$$

To find the $x^2$ values, the current voltage value was squared and added to a running sum. The number of samples was iterated upwards with every sample taken. For CalcRMS, the DC offset was subtracted from the current voltage squared, and added to a separate running sum. At the end of a sampling period, the running sum was divided by the number of samples, and square-rooted to find the respective RMS.

## *Bill of Materials:*

| *Component:* | *Qty:* | *Distributor:* | *Part Number:* | *Price per Unit:* | *Total Price:* |
|---|---|---|---|---|---|
| MSP432P401R w/ USB cable | 1 | Texas Instruments | 296-39653-ND | $13.03 | $13.03 |
| MSP432 Board Holder | 1 | Cal Poly IEEE | 534 | $5.00 | $5.00 |
| Total: | | | | | $18.03 |

**Table 2: Bill of Materials**

## *System Integration:*

At the beginning, the project was a lot for the students to wrap their heads around. Being able to calculate so many things seemed like a huge task that was just not possible. Splitting the whole project up into smaller tasks made it seem much more manageable. Initially, it was evident that there would have to be some sort of timer to measure the frequency and period. Whether that timer would measure one full period, or a certain amount of rising edges, or something else, remained to be decided as of yet. Because the voltage measurements seemed to be easier to implement than the frequency measurements, those were started first.

Making the program run fast was a priority, because there is many calculations that have to be done in a short amount of time. The frequency was increased to the maximum of 48 MHz. There are two options in the MSP432 to generate a 48MHz waveform: the DCO, or an external crystal oscillator. The crystal is much more precise than the DCO, so the high frequency crystal oscillator had to be used for the first time in this project. Also, because the frequency was increased to 48 MHz from 3 MHz, the baud rate for the UART transfer was changed as well. To change the eUSCI port's baud rate, a constant has to be found that is the result of the frequency being divided by the intended baud rate. This constant is then written into the eUSCI's BRW register. The terminal being used, RealTerm, has a few pre-selected baud rates available, and the project specs require the baud rate to be greater than 9600. Each option greater than 9600 was divided by 48*10^6, until a whole number was found. In this case, the baud rate found is 38,400, and the constant written into EUSCI_A0->BRW was 1250.

Gathering input from the ADC was no issue at this point, but now the real test was being able to analyze the data in a way that can isolate things such as maximum values, or RMS. Most of this turned out to be simple math. As described in an above section, the RMS values were found by computing a running sum of the input over a given time interval. The maximum and

minimum values were found by setting the first value in a given sample period to equal both max and min, and then comparing each successive value to both of the two. If a new value exceeded max or fell below min, the respective variable was updated. The DC offset is the average between the maximum and minimum. The peak-to-peak voltage is the maximum minus the minimum. Although the structure of the code changed over the course of this project, these calculations remained the same throughout.

To make the values appear more steady, and to be able to calculate an accurate maximum and minimum, many values from the ADC had to be sampled before the terminal values are updated. In this case, this group settled on 25,000 values. There was really no specific reason for this number, other than that it was sufficiently large, and considering how slow this code was running at the time, it would exceed the period of a 1Hz wave. It was officially measured at 1.34 seconds. This iteration of code did in fact generate fairly steady voltage values, because of the length of the sampling interval. If the sampling interval is greater than 1 second, it is guaranteed that the maximum and minimum values will have been read, and a decently accurate RMS value should have been calculated due to the large amount of samples input.

The real issue with this code began to arise when frequency calculation was attempted. At this point it looked like that it had to be kept separate from the rest of the voltage calculations, due to its time sensitive nature. Over a specified period of time, calculate the amount of full periods occur, and divide that by the time interval to get the frequency. It seemed straight forward, but deciding when a full period had passed is the difficult part. Setting a certain threshold to compare to the current voltage values seemed like a good idea, so that is the general idea that was implemented. First, this threshold was initially set to be the first value found in the sampling period. When this generated inconsistent results, the maximum voltage was given a shot as the threshold. Eventually, the DC offset was decided upon as the threshold for which to compare the input voltage to identify edges. The full implementation of this solution is described in the Component Design section above.

There was one idea for frequency that was unique from the others, and it took up a good amount work hours before eventually being abandoned. It was found that Timer A could be configured in “capture mode”, which is a very useful setting that attaches Timer A’s input to an external pin, and raises its interrupt flag every time that pin sees a rising edge. This seemed like the ultimate solution, because the timing could be made so exact, with minimal software solution. However, the input voltage had to be within a very specific set of values to trigger the timer properly, else it would do nothing. All of the group members’ EE knowledge was put to the test to figure out this issue.

It was found that in order for capture mode to work properly, the peak-to-peak voltage to to be approximately 2.5 V or greater. Given that the system specification require the smallest input signal to be at 0.5 Vpp, it was decided that an amplifier should be built in order to amplify the 0.5 V signal to something greater than 2.5 V. Initially, a LM301AN operational amplifier was used to attempt this amplification, but had no success in doing so. Using the design skills

and proper equipment from EE 308, another attempt was made at amplifying the signal using a common emitter amplifier biased by a current mirror, cascaded with a common collector. This would have allowed for the signal to be amplified properly and the common collector would have prevented the load from stealing any current from the main circuitry of the amplifier.

However, due to the unaccounted large output voltage swing (around 9 Vpp) when the input signal was maxed out at 3 Vpp, the amplifying method proved to be dangerous to use on the board. Even when attempting to put a 0V and 3.3 V rails on the output, the signal proved to be too strong for the rails to handle, making it exceed past the expected potential. In the end, this idea had to be abandoned since the amplifier turned out to be more complicated than expected and the MSP became damaged due to excessive voltage inputs through trial and error. When the due date of the project neared, a software solution for the frequency was instead discovered. This allowed for the hardware project to be scrapped and a software solution to be implemented.

Although it's not as creative as a hardware solution, the software frequency measurement still produced measurements of remarkable accuracy. The Timer 32 was used to count down a 1.2 second period. 1.2 seconds was chosen because it gives time for even the longest period, 1 second, to complete before calculating the frequency. Across this time period, the amount of times the input voltage crosses the DC offset level was counted. The code for identifying each rising or falling edge is detailed above in the component design section. Each count means a half period has progressed. At the end of the 1.2 second period, the total half-periods was counted, divided by 2, and then divided by 1.2 seconds to find frequency. The reciprocal of frequency is the period. Both values were sent to the terminal to be displayed.

As described above in component design, all of the voltage measurements were eventually moved into the Timer 32's ISR, calculating new values and sending them to the terminal every 1.2 seconds. It was also found that for lower frequencies, the 1.2 seconds of sampling was not quite enough to gather an accurate measure of frequency. If the frequency was below 175 Hz, the program would then sample values over 2.4 seconds, and if the frequency was below 20 Hz, the program would sample across 4.8 seconds. Because there are less rising or falling edges in a lower frequency signal, a small change in the amount of edges calculated can have a big effect on the resulting calculation, so a larger sample size was taken for these values to mitigate this risk.

## *Conclusion:*

The project preceding this was to create a function generator, so it makes sense that the next logical step is to create a multimeter to read those generated voltages. The use of an ADC is essential to this project, or else the analog values would not be useful to a digital circuit such as the MSP432. An ADC works sort of like a reverse DAC, but it requires a little more thought than a DAC does. A configurable sample and hold time makes timing the ADC conversions a crucial

part of the code, and because analog signals from an unreliable source tend to vary minutely, the resulting digital value also has a good amount of error to it. Managing this error is important to having a steady output value. Taking averages, or computing values over a longer sample period is the technique used here to mitigate this problem.

The timing of the different calculations and function calls is also a large part of making a successful working product. Some operations take longer than others, like a square root for example. Operations like this, and function calls like the one that updates all new values to the terminal, should be grouped together, so that the rest of the program can be written to schedule its processes around this large time gap. If the 50 ms long terminal write function was called in between a pair of ADC samples, then all the voltages that should have been sampled in that 50 ms interval are lost, and the resulting calculations will be inaccurate. Timing is more important in this project than it has in any project preceding this.

*Understanding how to create and utilize samples from an ADC is essential for interfacing a microcontroller to an analog circuit, without the usage of any extra circuitry components. Timing the samples and calculations is integral to the generation of accurate, stable values, and poor timing is the cause of many problems that occurred in this project. Timing will continue to be prominent in future microcontroller uses, and understanding how to efficiently manage operations is crucial. The terminal can be a useful tool for displaying a program output, but because writing data to it is such a slow operation, it is not ideal for small debugging operations and its use in the program must be managed carefully.*

## *Appendices:*

### *Source Code:*

**Main.c:**

```
/*****************************************************************************
*
 * The main code for allowing the MSP to be used as a Multimeter and displaying
 * onto a terminal using VT100 Protocol. Pin 5.4 is used to input the signal
 * into the ADC of the MSP in order t=for the MSP to process the information
 * and display certain values onto the terminal.
 *
 * Authors: Andrew McGuan, Denis Pyryev
 * Date: 5/19/2017
 */

#include <ADCdriver.h>
#include "msp.h"
#include <math.h>
#include "UART.h"

#define DC_SIGNAL 0
#define RECT_SIGNAL 1
#define OTHER_SIGNAL 2

void ADC14_init(void);
float average_max(float find_max[]);
float average_min(float find_min[]);
void update_UART_values(float nums[]);

/* Values instantiated in order to communicate with the
 * Interrupt Service Routines */
unsigned int half_periods = 0;  // Counts threshold crosses for frequency
float freq_ref = 1.5;   // Initial threshold set for half-periods
float frequency = 0;
int period = 0;
int interrupts = 0;   // Used to refresh the DC offset value

// The maximum and minimum voltage values across a period
float max = 0;
float min = 0;

float dc_off;    // DC Offset value
```

```
int samples = 0;  // Used to calculates RMS values
double square_sum = 0;  // True RMS
double calc_square_sum = 0;  // Calculated RMS
float voltage, last_voltage;
int de_flag = 0;      // Used for frequency precision
int signal_type = 0;  // Used to determine the signal type

int main(void) {
    volatile unsigned int i;

    // RGB LED Initialization
    P2->SEL1 &= ~0x07;        /* configure P2.1 as simple I/O */
    P2->SEL0 &= ~0x07;
    P2->DIR = 0x07;           /* P2.1 set as output pin */

    WDT_A->CTL = WDT_A_CTL_PW |       // Stop WDT
                 WDT_A_CTL_HOLD;

    P5->SEL1 |= BIT4;                 // Configure P5.4 for ADC
    P5->SEL0 |= BIT4;

    UART0_init();
    ADC14_init();

    /* Transition to VCORE Level 1: AM0_LDO --> AM1_LDO */
    while ((PCM->CTL1 & PCM_CTL1_PMR_BUSY));
     PCM->CTL0 = PCM_CTL0_KEY_VAL | PCM_CTL0_AMR_1;
    while ((PCM->CTL1 & PCM_CTL1_PMR_BUSY));

    /* Configure Flash wait-state to 1 for both banks 0 & 1 */
    FLCTL->BANK0_RDCTL = (FLCTL->BANK0_RDCTL &
     ~(FLCTL_BANK0_RDCTL_WAIT_MASK)) | FLCTL_BANK0_RDCTL_WAIT_1;
    FLCTL->BANK1_RDCTL = (FLCTL->BANK0_RDCTL &
     ~(FLCTL_BANK1_RDCTL_WAIT_MASK)) | FLCTL_BANK1_RDCTL_WAIT_1;

    /* Configure DCO to 48MHz, ensure MCLK uses DCO as source*/
    CS->KEY = CS_KEY_VAL ; // Unlock CS module for register access
    CS->CTL0 = 0; // Reset tuning parameters
    CS->CTL0 = CS_CTL0_DCORSEL_5; // Set DCO to 48MHz
    /* Select MCLK = DCO, no divider */
    CS->CTL1 = CS->CTL1 & ~(CS_CTL1_SELM_MASK | CS_CTL1_DIVM_MASK) |
     CS_CTL1_SELM_3;
    CS->KEY = 0; // Lock CS module from unintended accesses

    __disable_irq();    // Turn global interrupts on

    NVIC_EnableIRQ(ADC14_IRQn);
    NVIC_EnableIRQ(T32_INT1_IRQn);
```

```
__enable_irq();    // Turn global interrupts on

SCB->SCR &= ~SCB_SCR_SLEEPONEXIT_Msk;   // Wake up on exit from ISR

setADCval(0);       // Initialize the ADC value to 0
setADCFlag(1);      // Set flag to enter if statement in while loop

// Initialize the Timer_32 settings
TIMER32_1->CONTROL |= TIMER32_CONTROL_ENABLE |
                      TIMER32_CONTROL_SIZE | TIMER32_CONTROL_IE;

int adcval;      // Used for temporary ADC output storage

display_init();  // Display the Initials headings for the UART

float epsilon = 0.05;  // A difference of 0.05 in voltage
int intervals = 0;     // Used to wait for determining signal type
int wait = 10000;      // The initial wait amount of intervals

TIMER32_1->LOAD =  115200000;  // Initial Timer_32 set for 1.2 seconds

while (1)
{
    if(getADCFlag()){

        adcval = getADCval(); // Get the current ADC value

        // Update the DC Offset every 5 interrupts generated
        if(interrupts == 5) {
            freq_ref = dc_off;
            interrupts = 0;
        }

        // Convert ADC value to approximate voltage value
        voltage = adcval * 0.0002;

        /* If the voltage has passed the halfway threshold,
         * checks for both rising or falling edge */
        if ((de_flag == 0) && ((last_voltage+epsilon > freq_ref &&
                                voltage-epsilon < freq_ref) ||
                               (last_voltage-epsilon < freq_ref &&
                                voltage+epsilon > freq_ref))) {
            de_flag = 1;     // Set threshold pass signal high
            half_periods++;  // Increment the amount of threshold passes
        }

        /* Verifies that the current voltage has left the epsilon range
```

```
 * before looking for another rising/falling edge */
else if((de_flag==1) && ((last_voltage+epsilon < freq_ref &&
                              voltage+epsilon < freq_ref) ||
                          (last_voltage-epsilon > freq_ref &&
                              voltage-epsilon > freq_ref))) {
    de_flag = 0;
}

// Set the last voltage value equal to the current voltage
last_voltage = voltage;

if(voltage > max) max = voltage;  // Update current max value
if(voltage < min) min = voltage;  // Update current min value
dc_off = (max + min)/2;           // Calculate DC Offset

// Add current voltage square to the square sum
square_sum += pow(voltage, 2);

// Add current voltage minus offset squared to the tsquare sum
calc_square_sum += pow(voltage-dc_off, 2);

// Add to the current sample counter for RMS Calculations
samples++;

// Check to see what type of signal is bing input into the ADC
intervals++;
if (intervals > wait) {
    if (max - min < 0.2)   // DC if there is small peak-to-peak
        signal_type = DC_SIGNAL;
    else if ((voltage + 0.025 > max && voltage - 0.025 < max) ||
             (voltage + 0.025 > min && voltage - 0.025 < min) &&
             !(voltage < max - 0.025 && voltage > min + 0.025))
        signal_type = RECT_SIGNAL;   // Rectangle if no middle
    else
        signal_type = OTHER_SIGNAL; // Either Sine or Triangle
    intervals = 0;
}

// Light up Corresponding LED colors to signal types
if (signal_type == DC_SIGNAL) {
    P2->OUT &= ~0x07;   // Make the RGB LED green if DC
    P2->OUT |= 4;       // Output Blue to LED
    wait = 10000;
} else if (signal_type == RECT_SIGNAL) {
    P2->OUT &= ~0x07;   // Make the RGB LED green if rectangle
    P2->OUT |= 2;       // Output Green to LED
    wait = 10000;
} else if (signal_type == OTHER_SIGNAL)  {
```

```
                P2->OUT &= ~0x07;   // Make the RGB LED red if sine or triangle
                P2->OUT |= 5;       // Output Purple to LED
                wait = 52873;
            }

            // Reset the ADC setting to allow for next round of sampling
            ADC14->IER0 |= ADC14_IER0_IE0;
            ADC14->CTL0 |= ADC14_CTL0_ENC | ADC14_CTL0_SC;
            setADCFlag(0);
        }
    }
}

// ADC14 interrupt service routine
void ADC14_IRQHandler(void) {
    // Stores current ADC value into a variable
    setADCval(ADC14->MEM[0]);
    setADCFlag(1);
}

void T32_INT1_IRQHandler(void){
    float nums[7];                // Hold values to send to UART
    static float timer_len = 1.2;   // Real time of the Timer_32

    interrupts ++;  // For DC Offset

    // Calculate the current frequency, round to the nearest ones
    frequency = (half_periods/2)/timer_len + 0.6;

    // Calculate Period from Frequency and round to nearest int
    period = 1000.0/((int)frequency);
    half_periods = 0;

    // Values that are sent to UART
    nums[6] = sqrt(square_sum/samples);       // Calculate TRMS
    nums[5] = sqrt(calc_square_sum/samples);  // Calculate CRMS
    nums[0] = max;    // Maximum Voltage
    nums[1] = min;    // Minimum Voltage
    nums[2] = frequency * 1.006; // Frequency with adjustment
    nums[3] = period;            // Period
    nums[4] = dc_off + 0.01;     // DC Offset with adjustment
    update_UART_values(nums);

    // Reset the values for the next sampling period
    samples = square_sum = calc_square_sum = 0;
    max = min = voltage;

    TIMER32_1->INTCLR = 1;    // Clear interrupt flag
```

```
    if(frequency < 20){
        // Sample across 4.8 seconds for low frequencies
        TIMER32_1->LOAD =  230400000;
        timer_len = 4.8;
    }
    if(frequency < 175){
        // Sample across 2.4 seconds for frequencies between 20 and 175
        TIMER32_1->LOAD = 115200000;
        timer_len = 2.4;
    }
   else{
       // Otherwise, reset the countdown for another 1.2 seconds
       TIMER32_1->LOAD = 57600000;
       timer_len = 1.2;
   }
}

void update_UART_values(float nums[]) {
    // Update the Values on the Terminal display

    // Move cursor to home
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 0x1B;
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 0x5B;
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 'H';

    // Moe cursors around and display the new values in proper spots
    move_UART_cursor(5, 20);
    send_UART_decimal(nums[6]);       // TRMS value
    move_UART_cursor(6, 20);
    send_UART_decimal(nums[5]);       // CRMS value
    move_UART_cursor(7, 20);
    if (signal_type == DC_SIGNAL)
        send_UART_decimal(0.000);   // Vpp value of 0 if DC signal
    else
        send_UART_decimal(nums[0] - nums[1]);   // Vpp value
    move_UART_cursor(12, 30);
    send_UART_decimal(nums[4]);         // DC Offset value
    move_UART_cursor(17, 19);

    // Update signal type on terminal display
    if (signal_type == DC_SIGNAL)
        write_UART_string("DC Signal                    ");
    else if (signal_type == RECT_SIGNAL)
        write_UART_string("Rectangular Signal           ");
    else if (signal_type == OTHER_SIGNAL)
```

```
        write_UART_string("Sinusoidal / Triangular Signal");
    move_UART_cursor(19, 17);
    send_UART_integer(nums[2]);    // Frequency value
    move_UART_cursor(20, 17);
    send_UART_integer(nums[3]);    // Period value

    // Updating the display for the voltage bar graphs
    draw_UART_bargraph(25, 23, (int)(nums[5]*10));
    draw_UART_bargraph(26, 23, (int)(nums[4]*10));

    // Carriage Return to end transmission
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 13;       /* Send carriage return */
}

void ADC14_init(void){
    // Sampling time, S&H=16, ADC14 on, run on MCLK (48MHz)
    ADC14->CTL0 = ADC14_CTL0_SHT0_2 | ADC14_CTL0_SHP
            | ADC14_CTL0_ON | ADC14_CTL0_SSEL__MCLK;

    // Use sampling timer, 14-bit conversion results
    ADC14->CTL1 = ADC14_CTL1_RES_3;

    // A1 ADC input select; Vref=AVCC
    ADC14->MCTL[0] |= ADC14_MCTLN_INCH_1;

    // Enable ADC conv complete interrupt
    ADC14->IER0 |= ADC14_IER0_IE0;
}
/*******************************************************************************
/
```

## UART.h:

```
/*******************************************************************************
*
 * Header File for the UART source code
 */

#ifndef UART_H_
#define UART_H_

void UART0_init(void);
void write_UART_string(char string[]);
void display_init(void);
void send_UART_decimal(float num);
void send_UART_integer(int num);
```

```
void move_UART_cursor(int row, int col);
void draw_UART_bargraph(int startrow, int startcol, int length);

#endif /* UART_H_ */
/*****************************************************************************
/
```

## UART.c:

```
/*****************************************************************************
/
#include "msp.h"
#include <stdio.h>

/**
 * Moves the cursor on the terminal to
 * the specified row and column used VT100 protocol
 */
void move_UART_cursor(int row, int col){
    int temp;

    // Moves the cursor to the proper row specified
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 0x1B;
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 0x5B;
    if(row >= 10){
        temp = row / 10;
        while(!(EUSCI_A0->IFG & 0x02)) { }
        EUSCI_A0->TXBUF = temp+48;
        temp = row-temp*10;
        while(!(EUSCI_A0->IFG & 0x02)) { }
        EUSCI_A0->TXBUF = temp+48;
    }
    else{
        while(!(EUSCI_A0->IFG & 0x02)) { }
        EUSCI_A0->TXBUF = row+48;
    }

    // Moves the cursor to the proper column specified
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = ';';
    if(col >= 10){
        temp = col / 10;
        while(!(EUSCI_A0->IFG & 0x02)) { }
        EUSCI_A0->TXBUF = temp+48;
        temp = col-temp*10;
```

```
        while(!(EUSCI_A0->IFG & 0x02)) { }
        EUSCI_A0->TXBUF = temp+48;
    }
    else{
        while(!(EUSCI_A0->IFG & 0x02)) { }
        EUSCI_A0->TXBUF = col+48;
    }

    // Bring cursor back home
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 'H';
}

/**
 * Sends a command to the terminal,
 * with the two arguments as characters
 */
void send_UART_command(char first, char second){
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 0x1B;
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 0x5B;
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = first;
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = second;
}

/**
 * Writes a string to the terminal. Horizontal only
 */
void write_UART_string(char string[]){
    int i = 0;
    while(string[i] != '\0'){
        while(!(EUSCI_A0->IFG & 0x02)) { }
        EUSCI_A0->TXBUF = string[i++];
    }
}

/**
 * Writes certain number of hashes to represent
 * bar graphs onto the terminal with each hash = 0.1V
 */
void draw_UART_bargraph(int startrow, int startcol, int length){
    int i = 0;
    move_UART_cursor(startrow, startcol);
    while(i<33){
        while(!(EUSCI_A0->IFG & 0x02)) { }
```

```
        if(i<length)
            // Fill graph with hashes with corresponding voltage
            EUSCI_A0->TXBUF = '#';
        else
            // Refresh hashes off screen if next value is smaller
            EUSCI_A0->TXBUF = ' ';
        i++;
    }
}

/**
 * Initializes the UART settings
 */
void UART0_init(void)
{
    EUSCI_A0->CTLW0 |= 1;    /* Put in reset mode for config */
    EUSCI_A0->MCTLW = 0;     /* Disable oversampling */

    /* 1 stop bit, no parity, SMCLK, 8-bit data */
    EUSCI_A0->CTLW0 = 0x0081;

    EUSCI_A0->BRW = 1250;    /* 48,000,000 / 38400 = 1250 , Baudrate */
    P1->SEL0 |= 0x0C;        /* P1.3, P1.2 for UART */
    P1->SEL1 &= ~0x0C;
    EUSCI_A0->CTLW0 &= ~1;   /* Take UART out of reset mode */
}

/**
 * Writes the header information for data display
 */
void display_init() {
    // Clears screen
    send_UART_command('2', 'J');

    // Puts the cursor home
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 0x1B;
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 0x5B;
    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = 'H';

    // Everything below is just moving a cursor to a designated
    // spot and printing the corresponding header with spaces and
    // units for the values to them be filled in later
    move_UART_cursor(1, 10);
    write_UART_string("Multimeter Measurements");
    move_UART_cursor(2, 2);
```

```
    int i = 0;
    for(; i<40; i++)
        write_UART_string("=");
    move_UART_cursor(3, 2);
    write_UART_string("AC Measurements:");
    move_UART_cursor(5, 11);
    write_UART_string("TrueRMS:       Vrms");
    move_UART_cursor(6, 11);
    write_UART_string("CalcRMS:       Vrms");
    move_UART_cursor(7, 6);
    write_UART_string("Peak-to-Peak:       Vpp");
    move_UART_cursor(9, 2);
    for(i=0; i<40; i++)
        write_UART_string("-");
    move_UART_cursor(10, 2);
    write_UART_string("DC Measurements:");
    move_UART_cursor(12, 6);
    write_UART_string("DC Voltage / DC Offset:       Vdc");
    move_UART_cursor(14, 2);
    for(i=0; i<40; i++)
        write_UART_string("-");
    move_UART_cursor(15, 2);
    write_UART_string("Frequency Measurements:");
    move_UART_cursor(17, 6);
    write_UART_string("Signal Type:");
    move_UART_cursor(19, 6);
    write_UART_string("Frequency:      Hz");
    move_UART_cursor(20, 9);
    write_UART_string("Period:      ms");
    move_UART_cursor(22, 2);
    for(i=0; i<60; i++)
           write_UART_string("-");
    move_UART_cursor(23, 2);
    write_UART_string("Bar Graphs:");
    move_UART_cursor(24, 23);
    write_UART_string("0V       1V       2V          3.3V");
    move_UART_cursor(25, 6);
    write_UART_string("CalcRMS Voltage:");
    move_UART_cursor(26, 11);
    write_UART_string("DC Voltage:");
}

/**
 * Function used to send the Decimal voltage value to the terminal
 */
void send_UART_decimal(float num){
    char write = (char) num;
    char digit;
```

```
    /* Wait for transmit buffer to be full */
    while(!(EUSCI_A0->IFG & 0x02)) { }
    /* Send the ones digit to the terminal */
    EUSCI_A0->TXBUF = write+48;

    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = '.';    /* Send the decimal point */

    /* Shift over old number */
    digit = write * 10;
    /* Shift new number and find the difference */
    write = (char) (num * 10) - digit;

    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = write+48;  /* Send the tenths digit */

    /* Shift over old number plus the old-old */
    digit = (write+digit)*10;
    /* Shift new number and find the difference */
    write = (char) (num*100) - digit;

    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = write+48; /* Send the hundredths digit */

    /* Shift over old number plus the old-old */
    digit = (write+digit)*10;
    /* Shift new number and find the difference */
    write = (char) (num*1000) - digit;

    while(!(EUSCI_A0->IFG & 0x02)) { }
    EUSCI_A0->TXBUF = write+48; /* Send the thousandths digit */
}

/**
 * Initializes the UART settings
 */
void send_UART_integer(int num) {
   char string[10];
   if(num > 1000)   // No need to display a number greater than 1000
       return;
   sprintf(string, "%4d", num);  // Convert number to string
   int i = 0;
   while(string[i] != '\0' ){// Send string to terminal one char at a time
       while(!(EUSCI_A0->IFG & 0x02)) { }
       EUSCI_A0->TXBUF = string[i++];
   }
}
```

```
/*****************************************************************************
/
```

**ADCDriver.h:**

```
/*****************************************************************************
*
 * Header File for the ADC driver source code
 */

#ifndef ADCDRIVER_H_
#define ADCDRIVER_H_

#define DC_MODE 0
#define AC_MODE 1

void setADCFlag(char n);
char getADCFlag();
char getADCval();
void setADCval(unsigned int n);

#endif /* ADCDRIVER_H_ */
/*****************************************************************************
/
```

**ADCDriver.c:**

```
/*****************************************************************************
/
static char ADCflag;
static unsigned int adcval;

/**
 * Flag is set high when the ADC has read in another value
 */
char getADCFlag(){
    return ADCflag;
}

/**
 * Changes the current value of the ADC ready flag
 */
void setADCFlag(int n){
    ADCflag = n;
}

/**
```

```
 * Returns the current digital value of the ADC
 */
unsigned int getADCval(){
    return adcval;
}

/**
 * Used to update the current digital value of the ADC
 */
void setADCval(unsigned int n){
    adcval = n;
}
/*****************************************************************************
/
```

***References:***


Program Examples :
http://microdigitaled.com/ARM/MSP432_ARM/Code/MSP432_red_codes.htm
MSP432401R Datasheet:
http://www.ti.com/lit/ds/symlink/msp432p401m.pdf
VT-100 Protocol:
http://ascii-table.com/ansi-escape-sequences-vt-100.php