P3 – Multimeter

Zarek Lazowski

EE 329-03, Spring 2021

Dr. Paul Hummel

June 4, 2021

Behaviour Description

This device is able to measure average, root-mean-square, and peak-to-peak voltage, as well as frequency.

There are two modes: DC and AC.

DC mode is the default mode and will display the value and a bar graph representing the average voltage of the input. If one is not in DC mode, they can access it by typing ‘D’ on the terminal.

AC mode requires the input to be periodic and will display the value and a bar graph representing the root-mean-square voltage, the value of the peak-to-peak voltage of the waveform, and the value of the frequency of the waveform. If one is not in AC mode, they can access it by typing ‘A’ on the terminal. If, however, one were to attempt to access AC mode with a non-periodic input, the device will time out after 5 seconds and reset, which puts the device back into DC mode.

System Specification

Table 1: System Specifications

Max Input Voltage	3 V
Min Input Voltage	0 V
Max Input Frequency	1000 Hz
Min Input Frequency	0 Hz
Min Peak to Peak Voltage	0.5 V
DC Mode Refresh Rate	11 Hz
AC Mode Refresh Rate	0.3 Hz
Screen Resolution (characters)	58x20
Power supply	3.3V 11mA

System Schematic

Figure 1 below shows how the system was assembled.

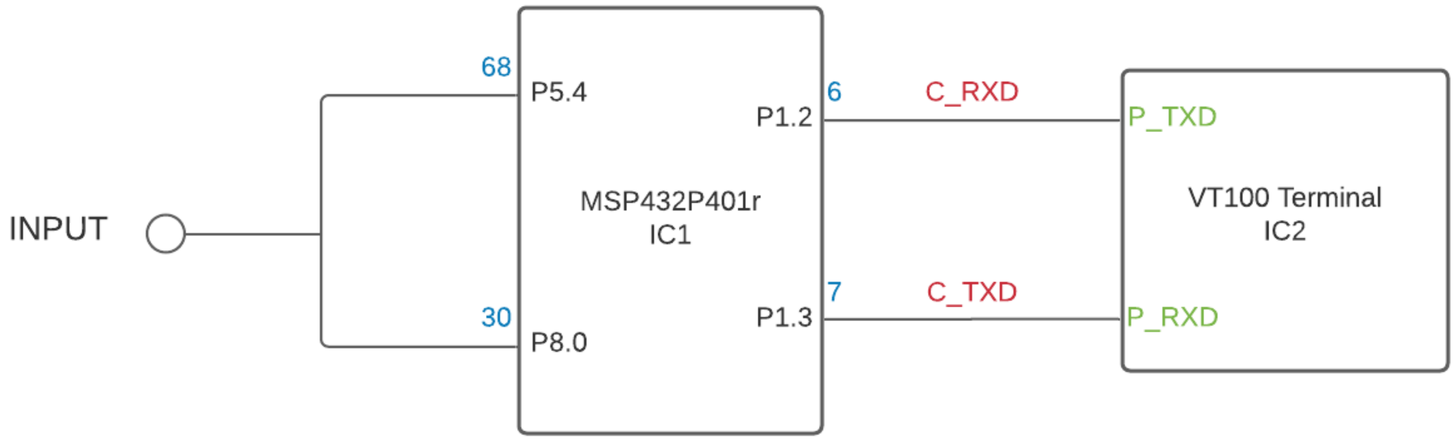

Figure 1: Pinout of the Function Generator

Software Architecture

Figure 2 below shows the user interface when the device is operating in DC mode. Figure 3 shows the user interface when operating in AC mode.

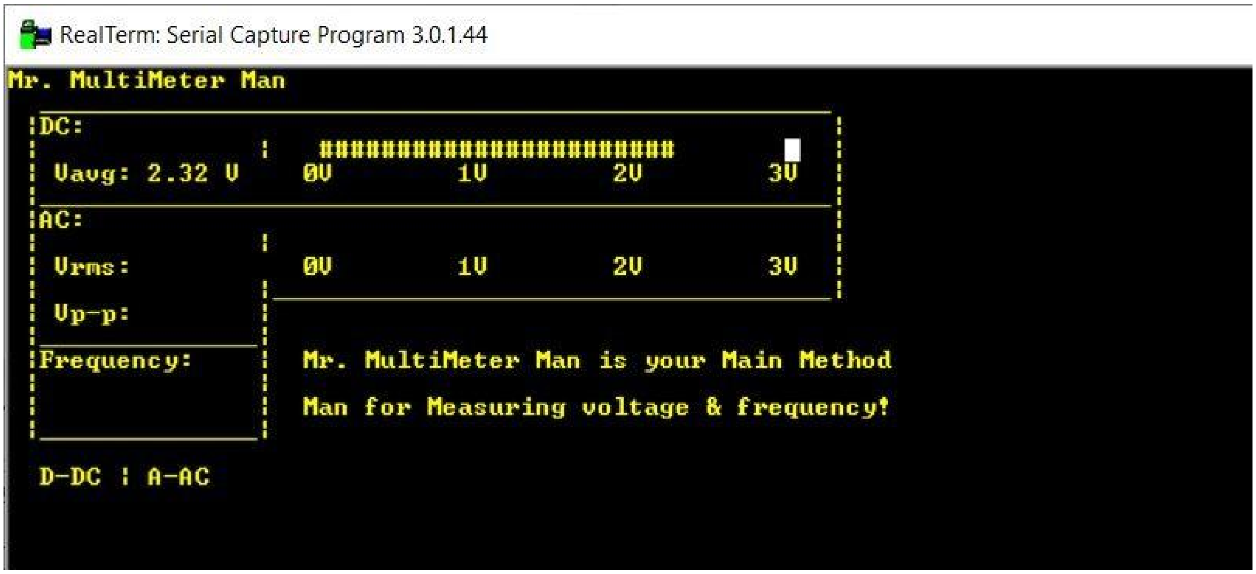

Figure 2: UI in DC Mode

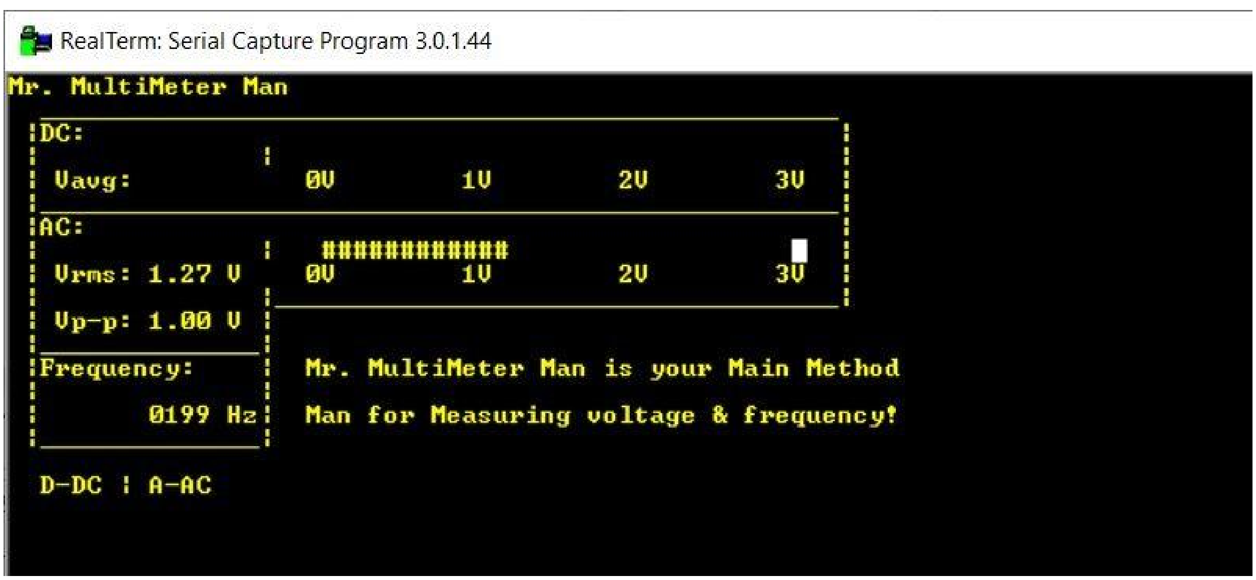

Figure 3: UI in AC Mode

Upon powering up, the system sets the system clock to run at 24 MHz, four states for a state machine are defined, and the necessary peripherals are initialized. This system uses the MSP432's Comparator E module, UART mode for the eUSCI-A module, the 14-bit ADC module, and two Timer A modules. A flowchart describing the main loop can be found below in figure 4.

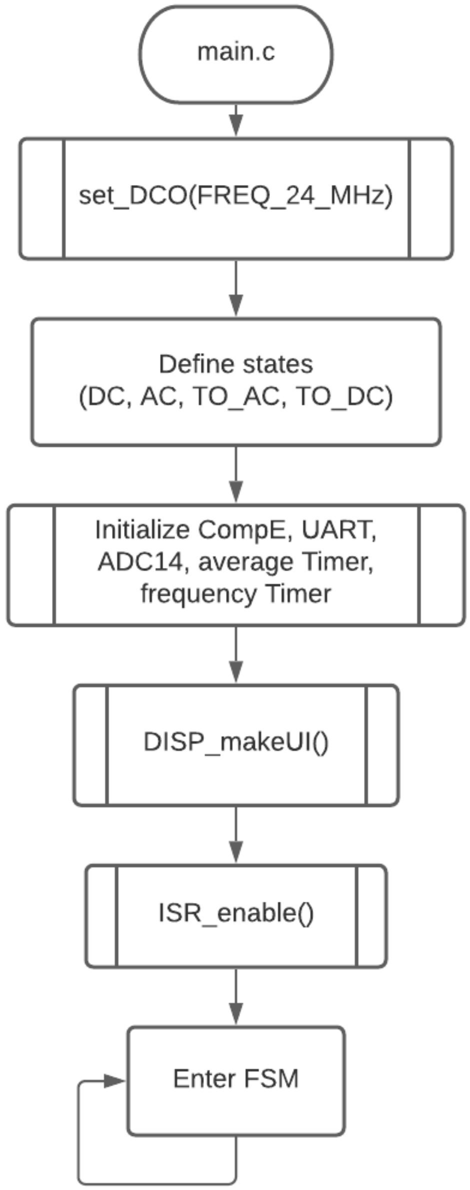

Figure 4: Flowchart of System Main Loop

Figure 5 below outlines the flow of initialization for eUSCI-A's UART mode. In order to modify the required registers, the module needs to be put in software reset. Afterwards the control register is modified to operate in UART mode, selects the 24 MHz SMCLK as a source, and selects the two-stop-bit option. In order to use 16-bit oversampling, the option is selected and precalculated values are set for the clock's prescaler, frequency, and modulation options. These values are selected to run the UART module with a baud rate of 115.2 kbps.

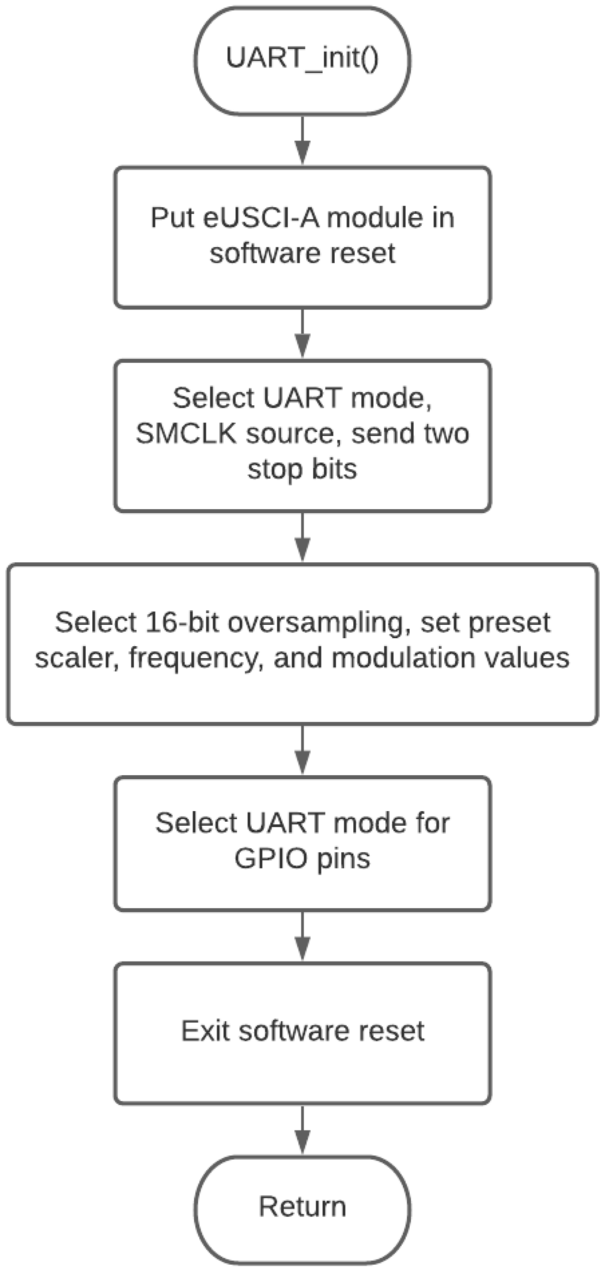

Figure 5: Flowchart of UART initialization

Figure 6 details the flow of 14-bit ADC module. The desired GPIO pins are first selected and are configured to act as input and output for the eUSCI-A module. Afterwards the ADC14 control register is configured to sample a voltage for four clock cycles, read from a single channel and do a single conversion, and selects the 24 MHz SMCLK as the source. Additionally, the ADC module is turned on and “sample and hold” mode is selected. A 14-bit conversion mode is selected, a 3.3 V reference is selected, and the GPIO pin configured previously is set as the input channel.

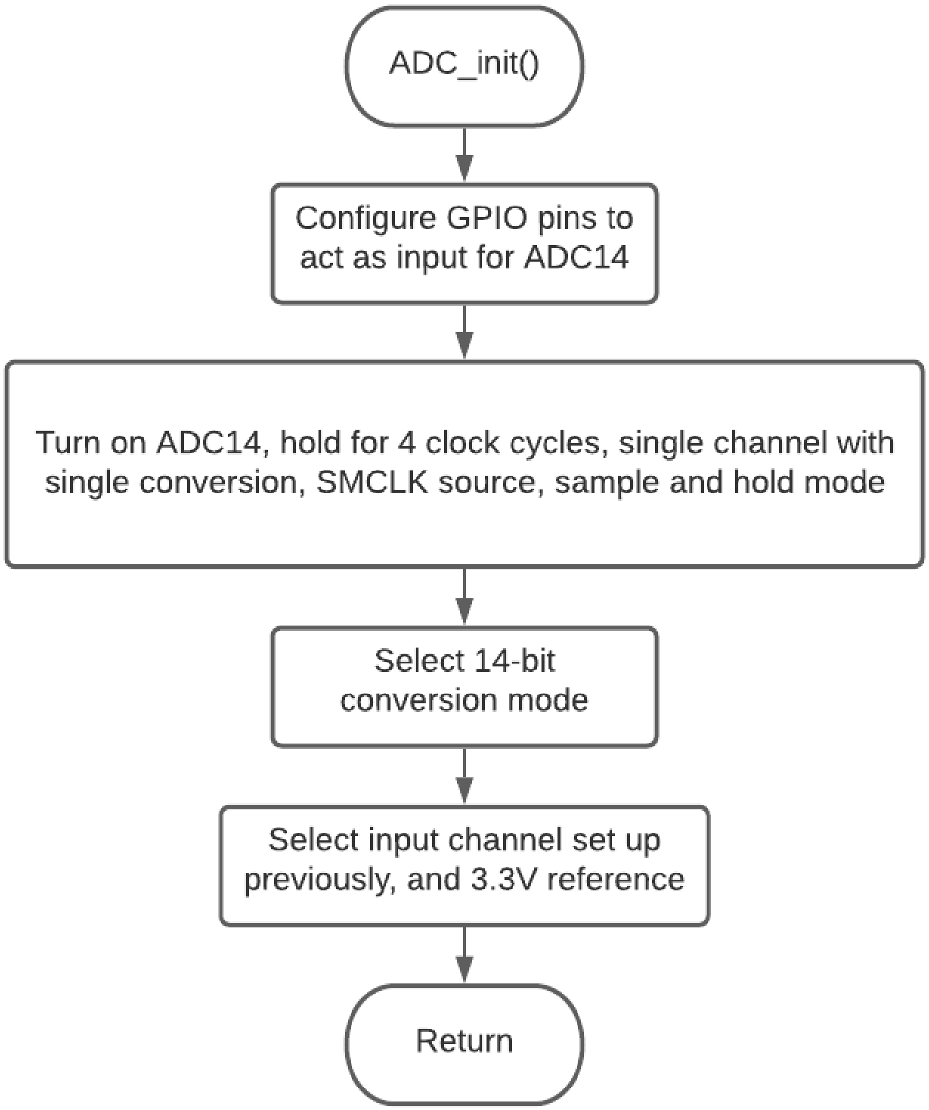

Figure 6: Flowchart of ADC initialization

Figure 7 describes the flow of the Comparator E module. The input channels are selected and the respective GPIO pins are configured to act as inputs. The comparator and output filter are turned on, and a 3000ns delay is selected for the filter. The included reference voltage generator is used to create threshold voltages for hysteresis. The 3.3 V rail is selected as the source for the reference voltage generator and the reference voltages are applied to the negative terminal of the comparator.

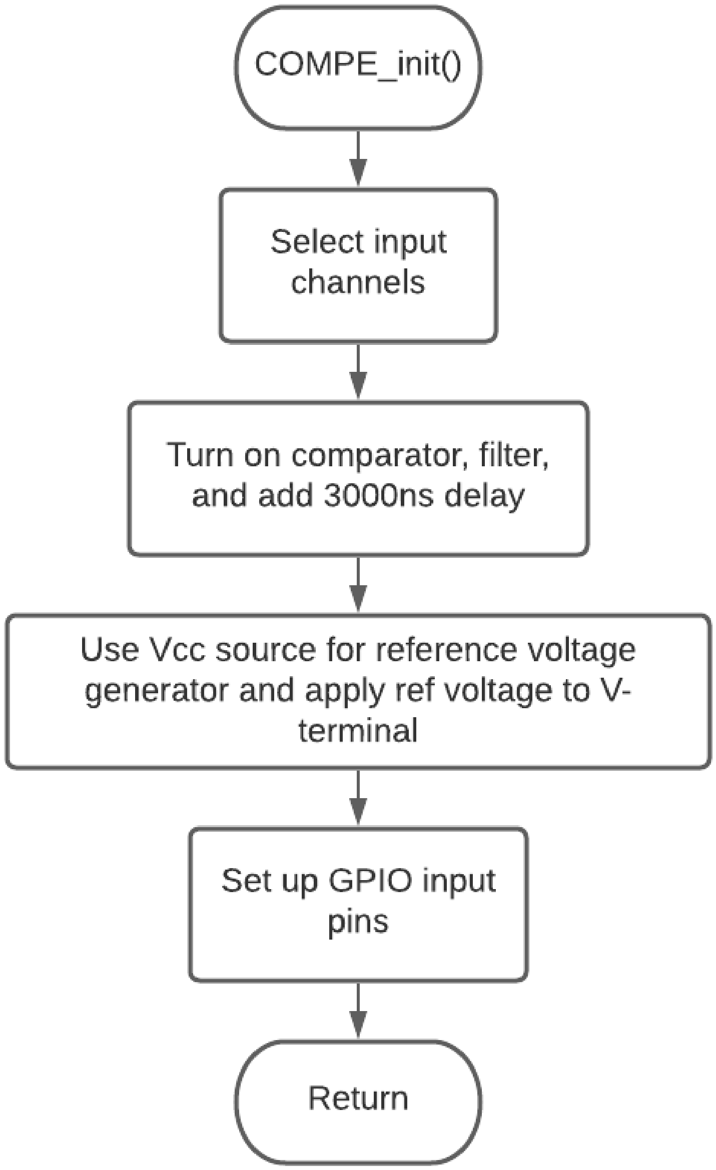

Figure 7: Flowchart of CompE initialization

Figure 8 describes both Timer A initializations, as they're fairly similar. Both use the 24 MHz and their timers are stopped until they are needed. The timer used for frequency measurement however uses additional options, in which capture mode is selected, the internal signal CCIB is selected as the input signal, and is set to capture on the rising edge of the input.

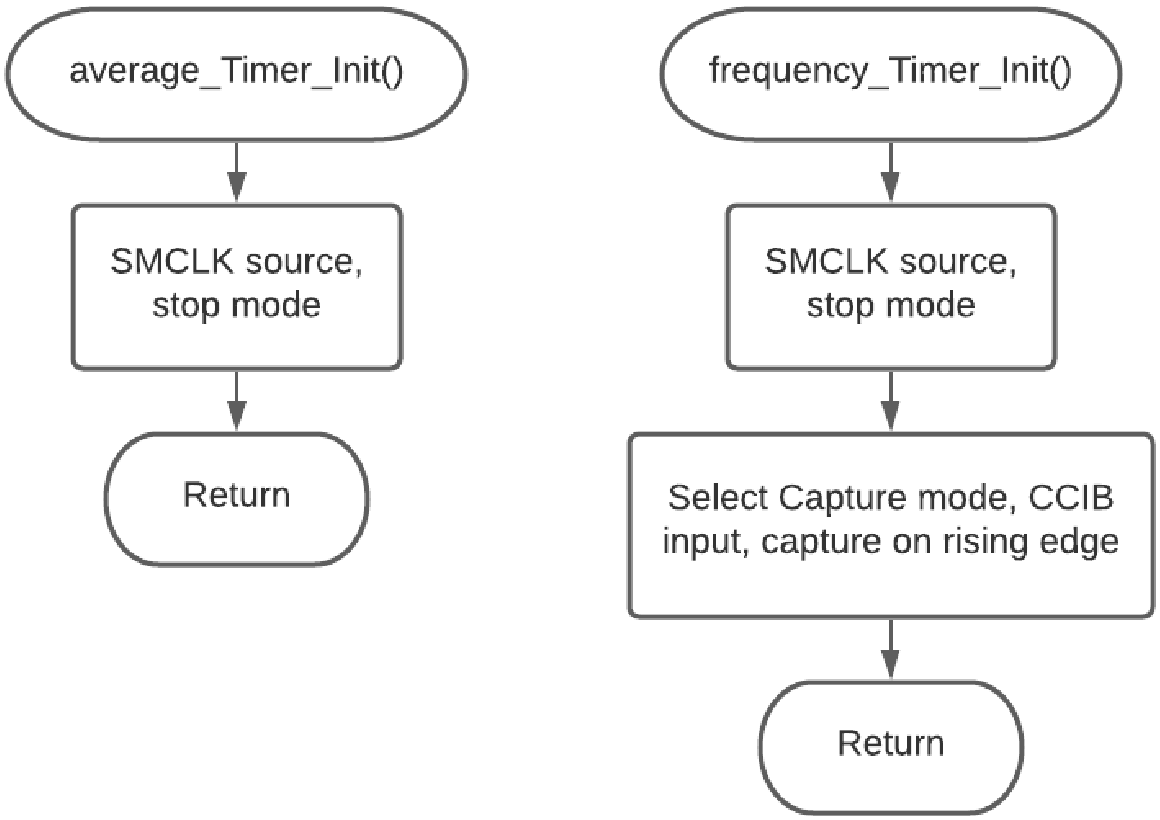

Figure 8: Flowchart of TimerA initializations

Figure 9 describes the flow of the function that makes the skeleton of the UI. After clearing the screen, the system prints a title, an option prompt, and a message at their respective positions on the screen. Then the system creates borders for each box, which displays different information for each mode of operation. At the end of the function, labels for each box are created. This is done once at the beginning and functions that update values in the UI go to their location before printing the respective values.

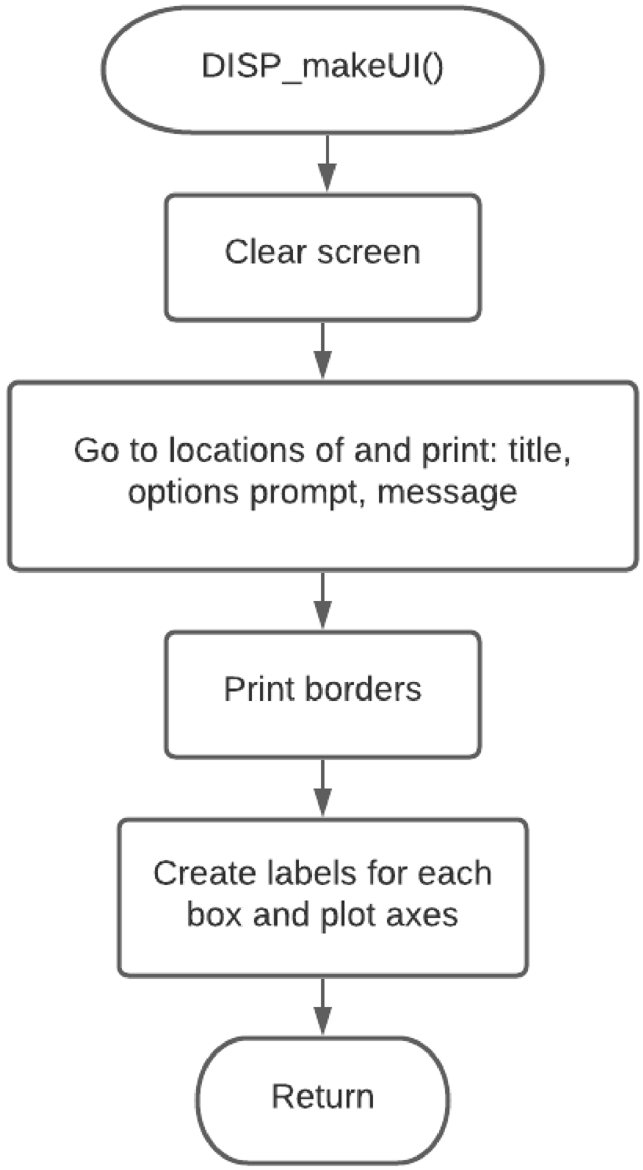

Figure 9: Flowchart of UI Initialization

Figure 10 describes the flow of the UART functions that print to the screen. These functions are used throughout the display library in order to print to the terminal. To print a string, one can call the UART_print_str() function with a char pointer argument. The function then goes through each entry of the array until the nul-byte is encountered. If a non-nul character is encountered, the function calls UART_print_char() with said character as the argument. This function waits until the transmit buffer is empty, and enters the given character into the buffer.

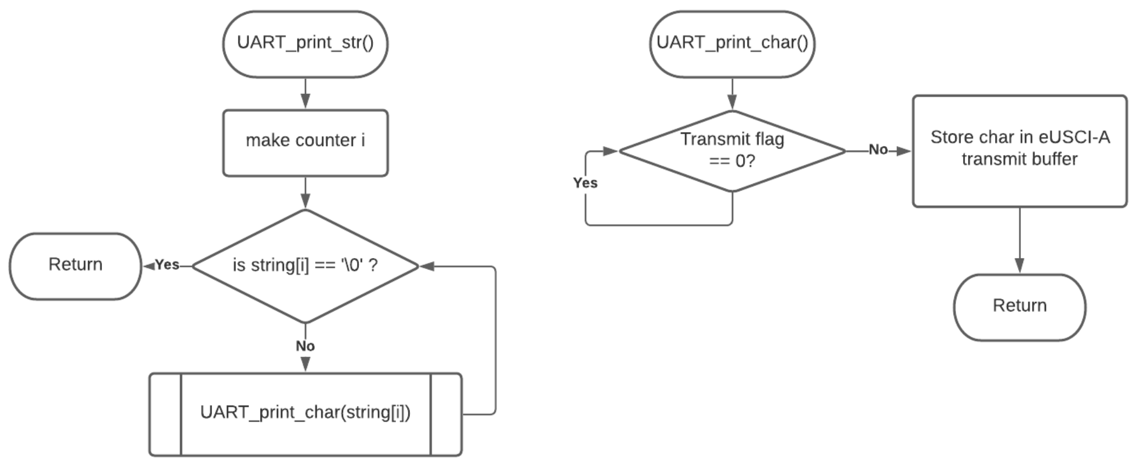

Figure 10: Flowchart of UART Print Functions

Figure 11 describes how VT100 escape codes are handled in the UART library. Macros are predefined for specific actions, some of which are provided in the UART header file. These macros are strings which are used as arguments in UART_esc_code. This function prints the escape character to the terminal, and then UART_print_str is called with the given macro as the argument.

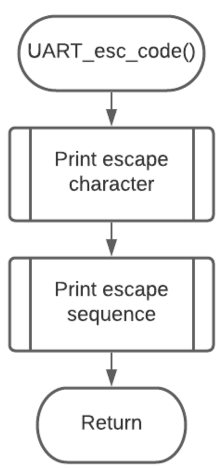

Figure 11: Flowchart of UART_esc_code

Figure 12 describes the process in which the ISRs are enabled. For each interrupt used, they need to be enabled in their respective control register and in the NVIC module. This is done for the eUSCI-A module, TA0 and TA1 modules, and the ADC14 module. Interrupts are then enabled for the MSP432, conversions for the ADC14 are enabled, and the function returns.

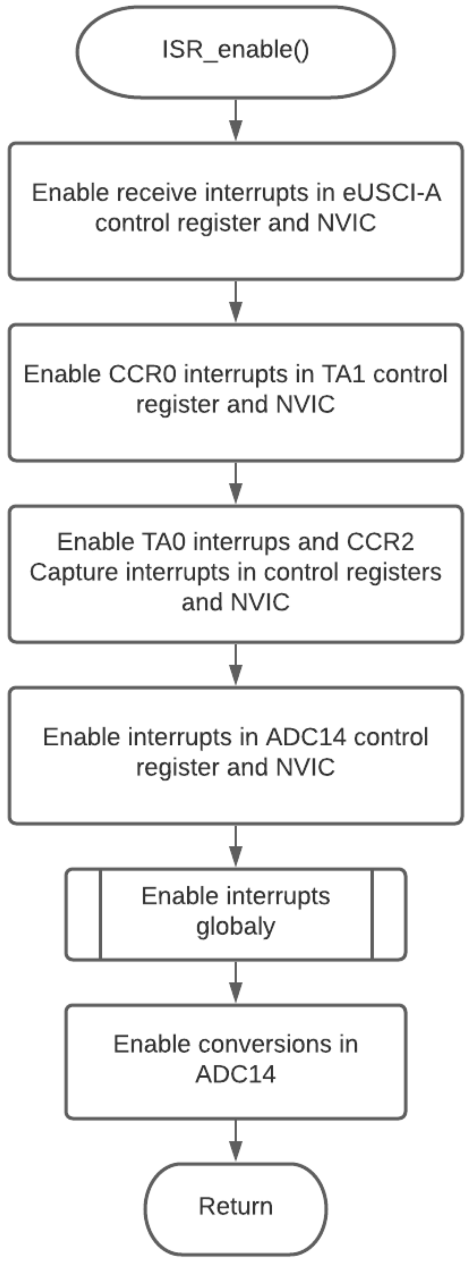

Figure 12: Flowchart of ISR_enable

Figure 13 describes the flow of the eUSCI-A received interrupt. This interrupt saves whatever is in the received buffer, and saves it into a global variable until it is needed.

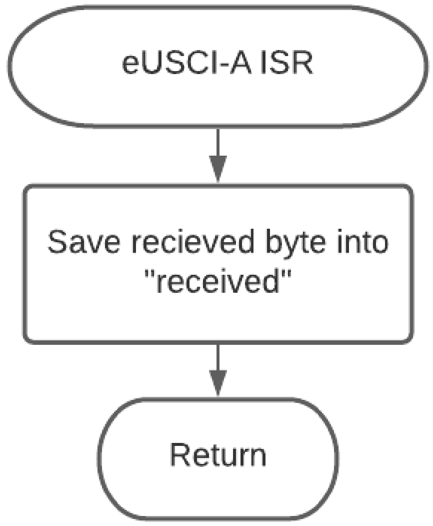

Figure 13: Flowchart of eUSCI-A Received ISR

Figure 14 describes the flow of the TimerA1 ISR. This ISR clears the interrupt flag and sets the start bit for the ADC14 module.

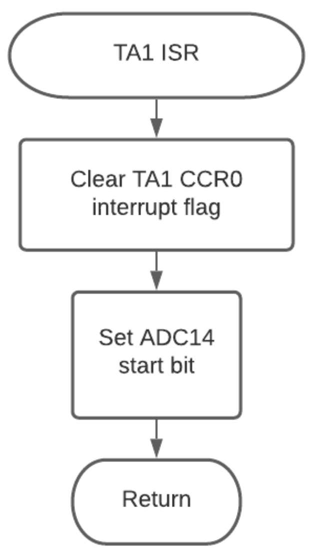

Figure 14: Flowchart of TA1 CCR0 ISR

Figure 15 describes the flow of the ADC14 ISR. This ISR sets a global flag used in any function that needs to take measurements via the ADC, clears the interrupt flag, and saves the output into a global variable to be accessed later.

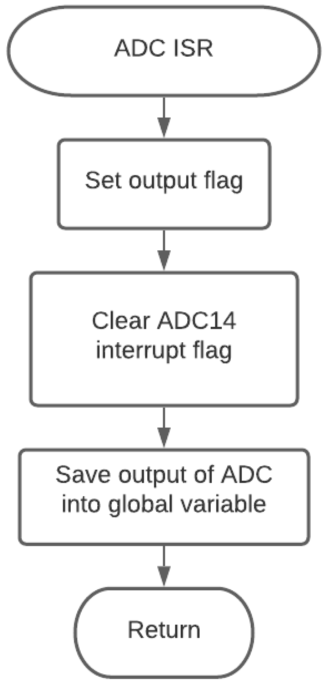

Figure 15: Flowchart of ADC14 ISR

Figure 16 describes the flow of the TA0 interrupt. This ISR is only used in measuring the frequency of the comparator, and looks for two flags.

If the TA0 CCR2 interrupt flag is set, the flag is first cleared and the capture overflow bit is checked. If capture overflow did not occur, the CCR value is saved into a global variable, a global continue flag is set, and a global stop flag is checked. If this global stop flag is set, the ISR stops TA0 from counting. If unset, the number of timer overflows is set to 0, as this is the first CCR value checked. However if capture overflow did occur, the capture overflow flag is cleared.

If this interrupt was called for timer overflow, the flag is cleared and the overflow counter is incremented once.

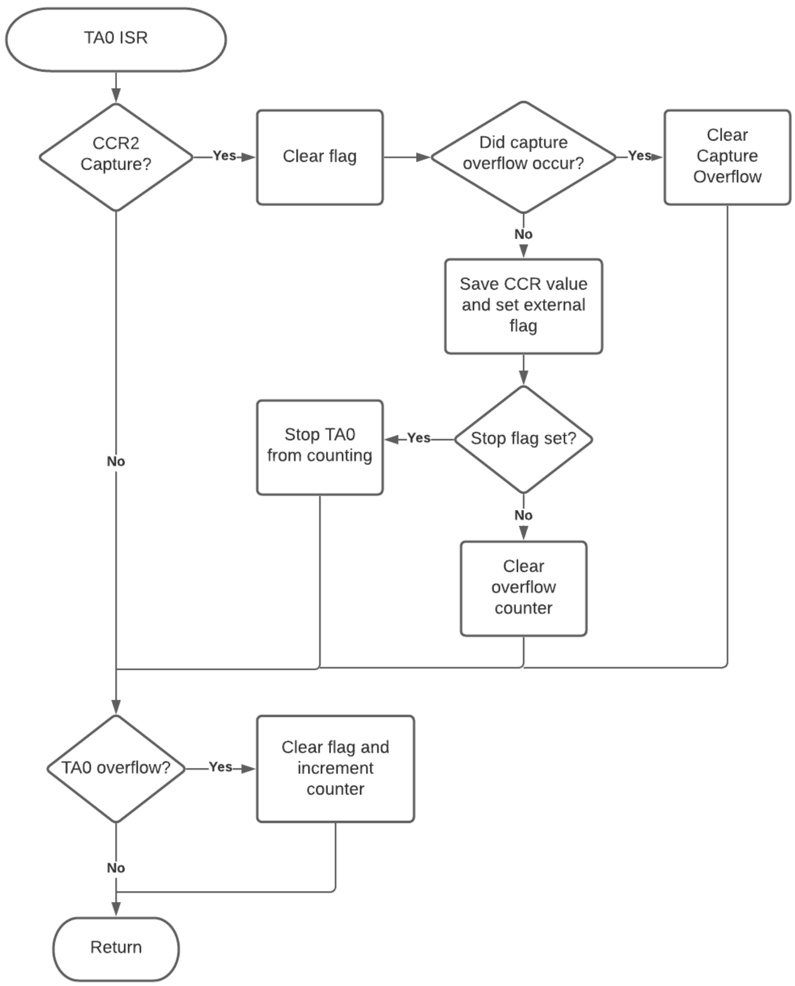

Figure 16: Flowchart of ADC14 ISR

Figure 17 describes the operation of the system's state machine. There are four states: *DC*, *AC*, *TO_AC*, and *TO_DC*. The *DC* and *AC* states are the two main modes of the system, with *DC* being for periodic or non-periodic waveforms, and *AC* being for strictly periodic waveforms. If a user were to want to switch between modes, the system enters the respective *TO_XC* state before entering the desired state.

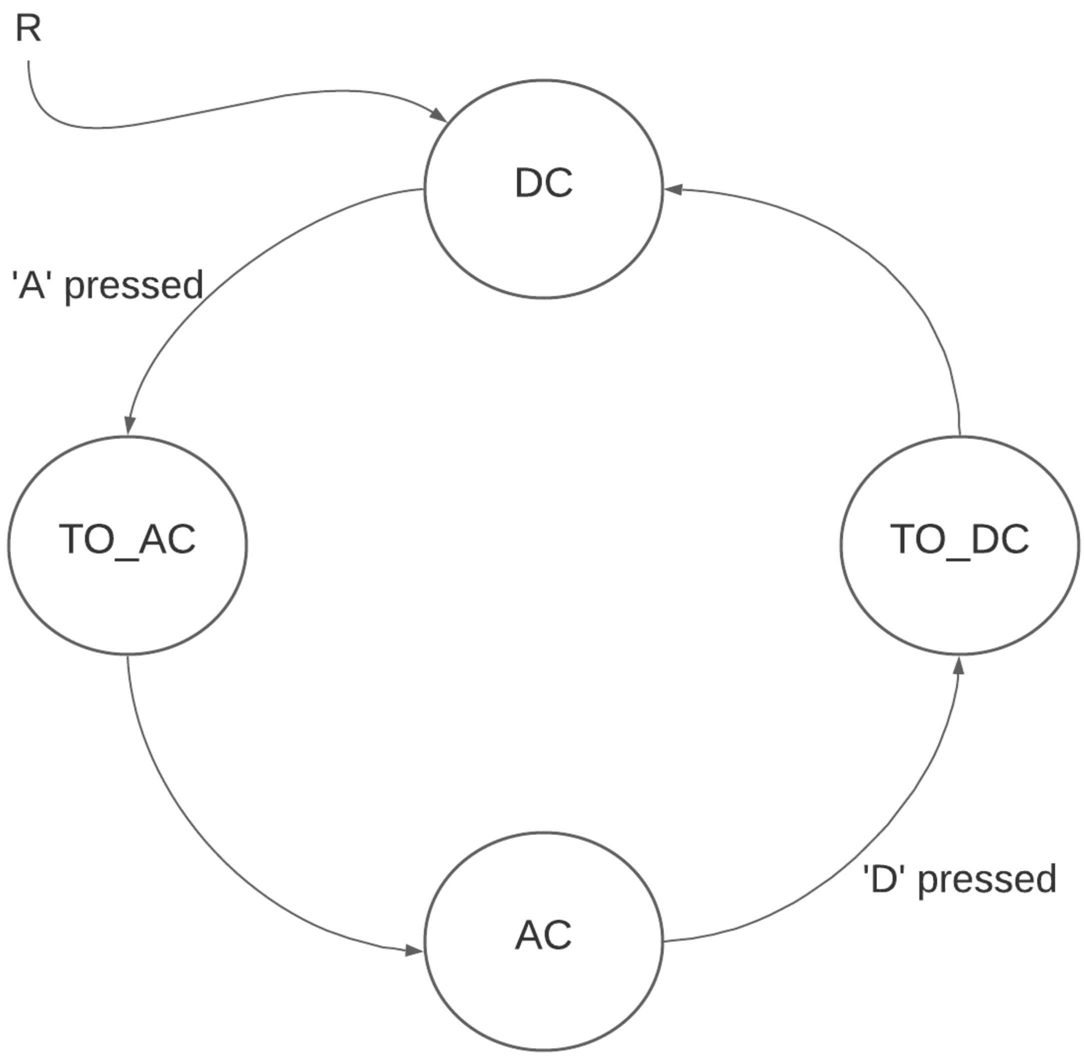

Figure 17: Diagram of System State Machine

Figure 18 describes the flow of the two transition states: *TO_AC* and *TO_DC*. Both states are fairly similar, as they clear the parts of the display previously used. This is done so that the user does not get confused about what the device is trying to convey. *TO_AC* will clear the DC mode values from the display, and *TO_DC* will clear the AC mode values from the display.

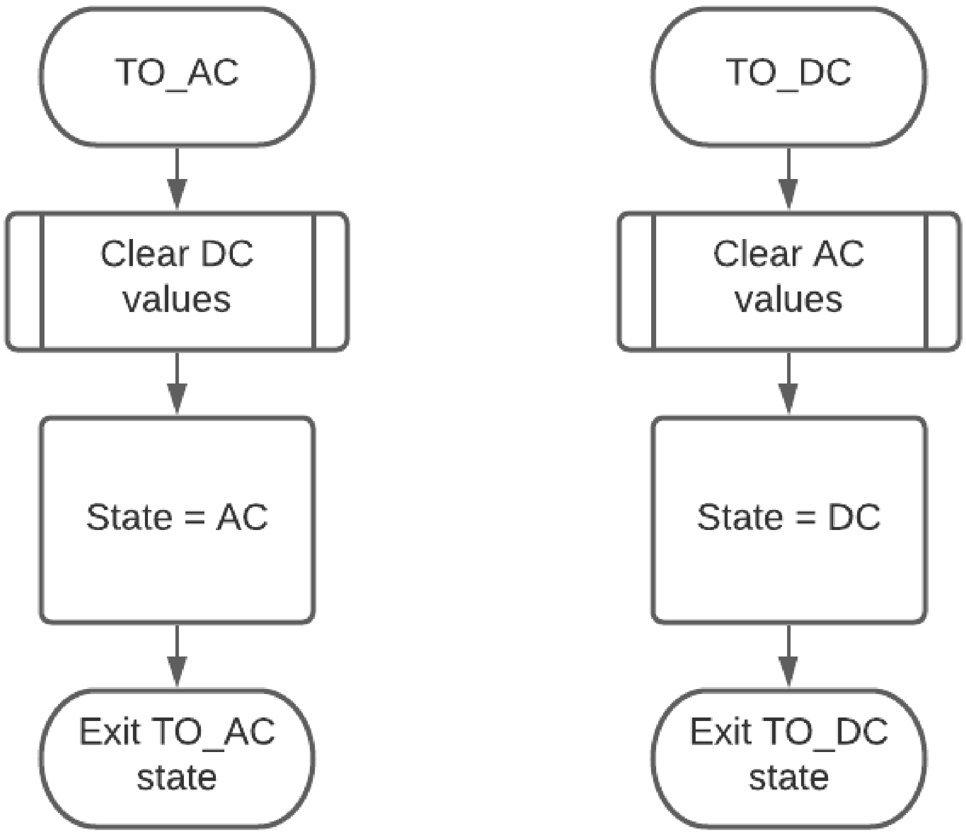

Figure 18: Flowchart of *TO_AC* and *TO_DC* states

Figure 19 describes the flow of the DISP_clear_DC and DISP_clear_AC functions used in the *TO_AC* and *TO_DC* functions respectively. Both of these functions clear the DC and AC portions of the UI, respectively. As such, both of these functions are fairly similar. Clearing the DC portions involves clearing the V_{AVG} value and bar graph. Clearing the AC portions involves clearing the V_{RMS} value and bar graph and the V_{PP} value.

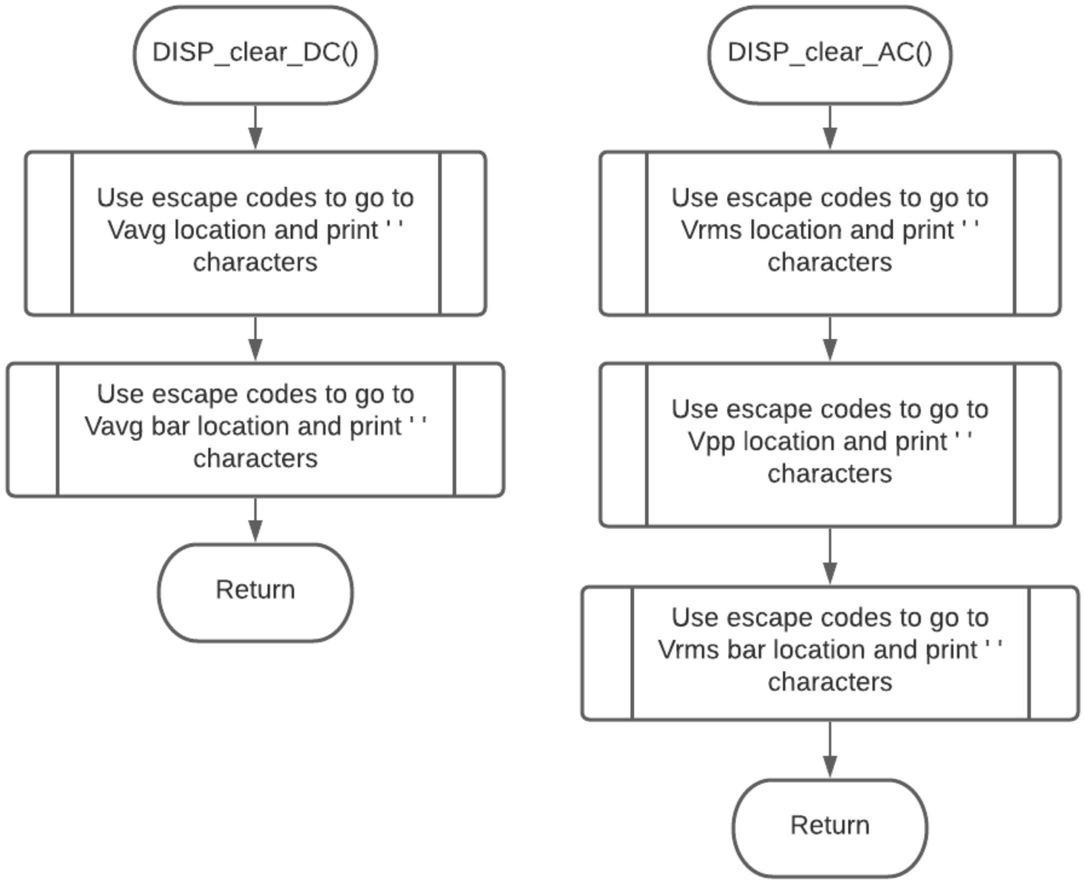

Figure 19: Flowchart of DISP_clear_DC and DISP_clear_AC functions

Figure 20 describes the *DC* state. This state takes a fast average of the input signal and updates both the V_{AVG} value and bar graph. Then the system checks to see if the character ‘A’ was received. If so, the next state is selected to be *TO_AC*. Otherwise the state variable is left alone and the system reenters the state machine in the *DC* state.

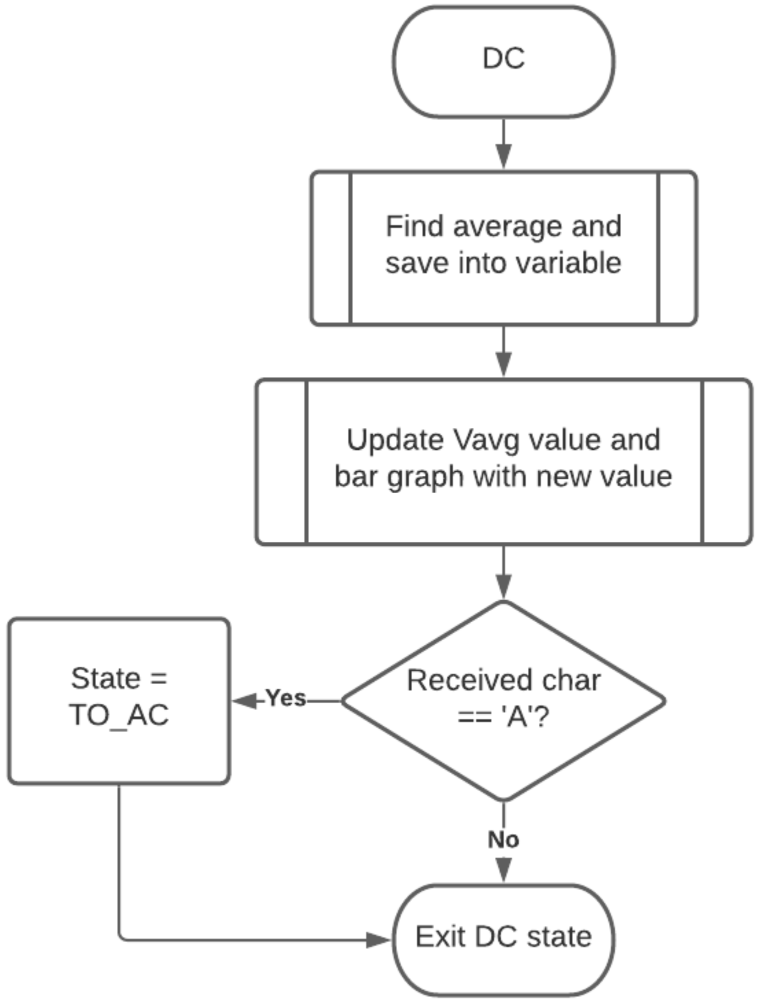

Figure 20: Flowchart of *DC* state

Figure 21 describes both averaging functions, as they are functionally identical. The function find_Fast_Average is used in the DC state to average takes 180 samples over 90 ms, while find_Offset is used in the AC state and takes 10,000 samples over the course of a full second. Both functions set the CCR value of TA1 to their respective values and start the TA1 counter. The fast average function takes a sample every 0.5 ms while the offset average function takes a sample every 0.1 ms. Both functions convert the 14-bit ADC outputs into a voltage and creates a running sum of the voltages. After taking all the samples, the timer is stopped, the sum is divided by the number of samples taken, and the average value is returned to the caller.

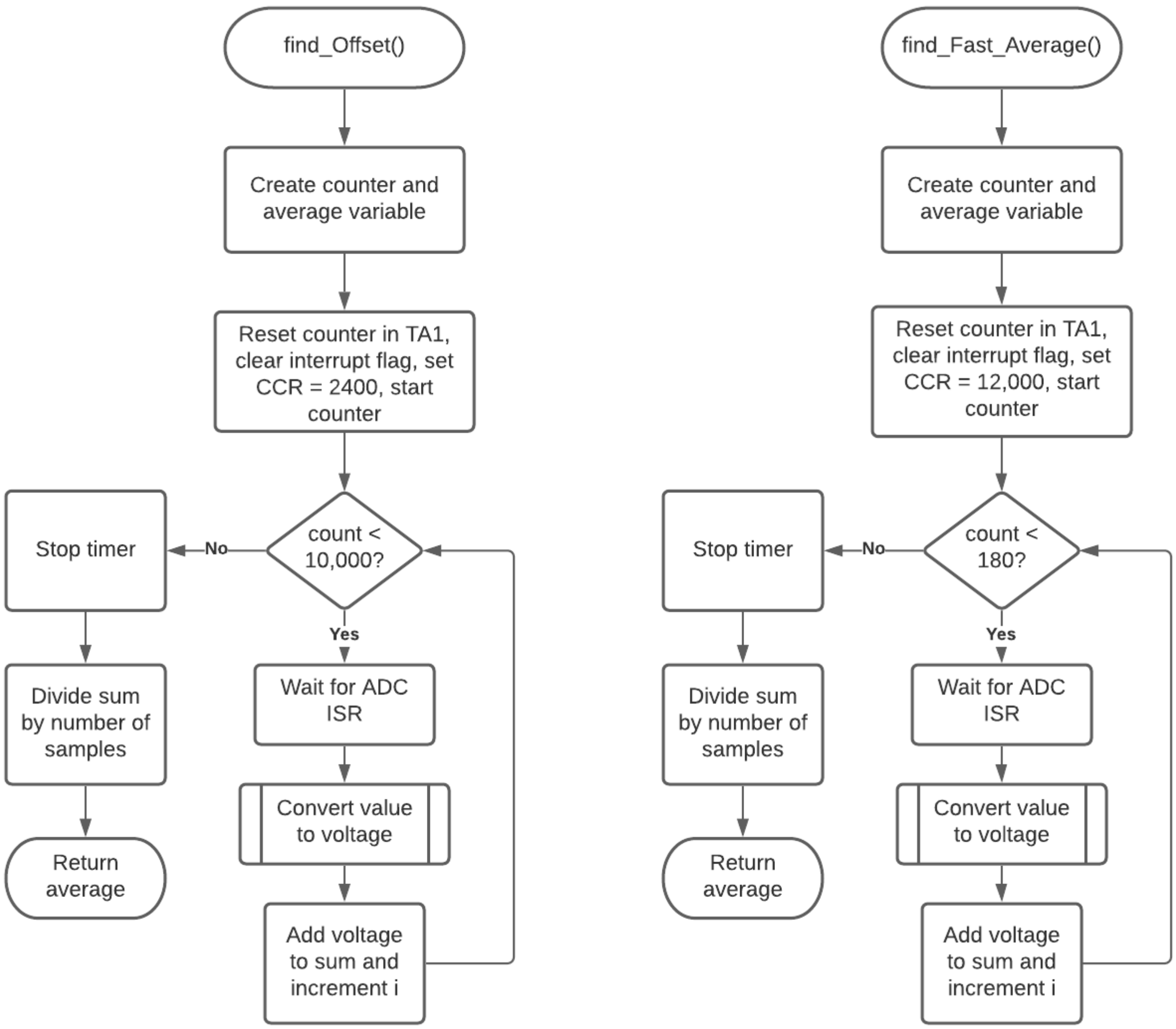

Figure 21: Flowchart of find_Offset and find_Fast_Average

Figure 22 describes the conversion process for turning a 14-bit number into a voltage. This function follows a $y = mx + b$ relationship, as determined by a linear best fit model and adjusted based on several readings in a real world environment. Currently, $m = 2$ and $b = -280$. This value is then scaled down by 100 to get voltage in terms of 10s of millivolts. Based on readings, the value 334 was found to be the average 14-bit value used to represent 0 V. This value is then used as a lower limit, and any input values below this will be converted to 0 V.

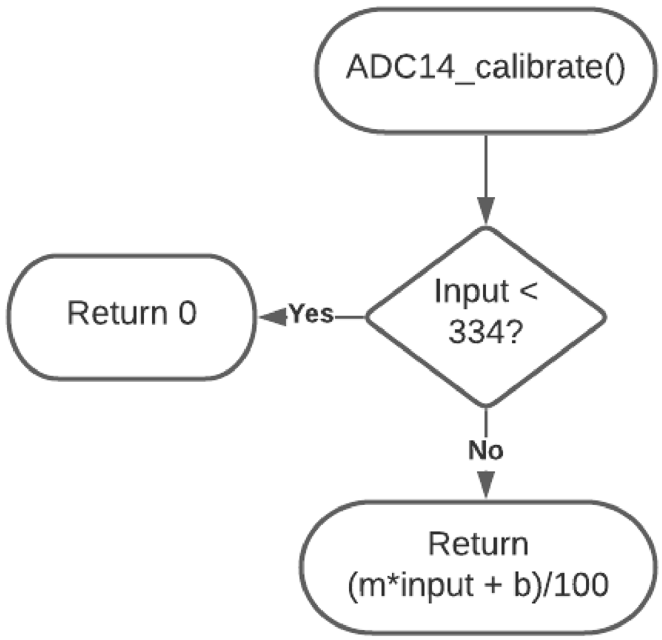

Figure 22: Flowchart of ADC14_calibrate

The other new functions in the *DC* state are the DISP_update_Value family of functions. As they are all functionally identical, figure 23 will cover DISP_update_VAVG, DISP_update_VRMS, DISP_update_VPP, and DISP_update_FREQ. Each function goes to their respective location and overwrites whatever characters may be there with the given argument.

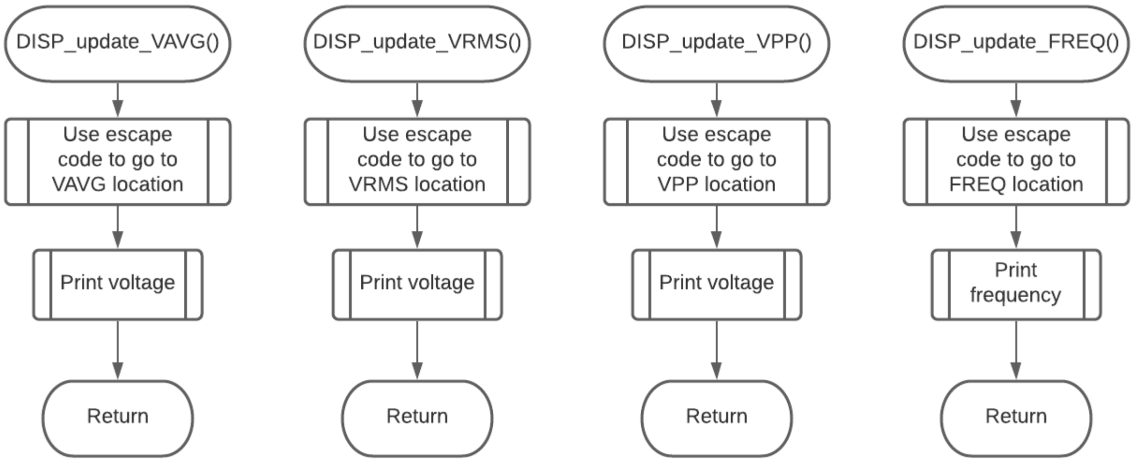

Figure 23: Flowchart of DISP_update_VAVG, DISP_update_VRMS, DISP_update_VPP, and DISP_update_FREQ

Similarly to above, DISP_printVolts and DISP_printFreq are functionally similar. Figure 24 describes the flow of these two functions. Voltages only have three values to print, and therefore only need to separate the value into hundreds, tens, and ones values. Frequencies however need to be separated into thousands, hundreds, tens, and ones values. Then each value is printed in descending order of magnitude.

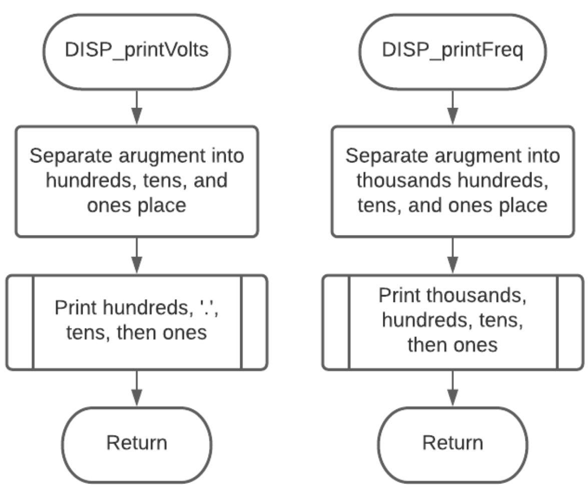

Figure 24: Flowchart of DISP_printVolts and DISP_printFreq

Figure 25 describes the flow of the AC state. This state consists of finding a usable average with the find_Offset function, converting this into a 5-bit value to find an upper and lower threshold for the comparator, calculating the frequency of the resulting square wave and updating the terminal with the measured value, then using this value to take equally spaced samples to find the min, max, and RMS voltages. The RMS and peak-to-peak voltages are then updated on the terminal, the received character is checked. If the character is ‘D’, then the next state is set to *TO_DC*.

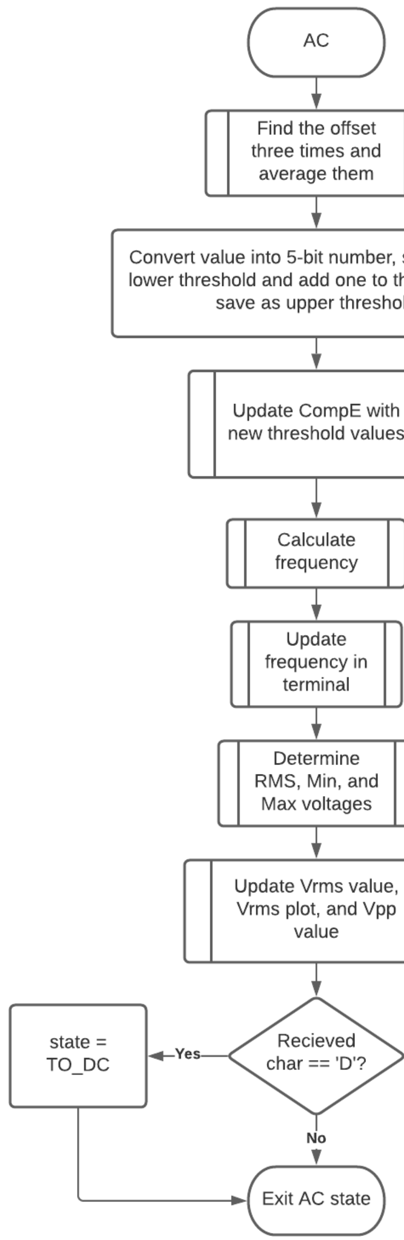

Figure 25: Flowchart of *AC* state

Figure 26 describes how the system updates the threshold voltages of the comparator. The arguments are first cleaned so nothing is unintentionally written over. The old reference voltages are cleared, and the new arguments are stored in their place.

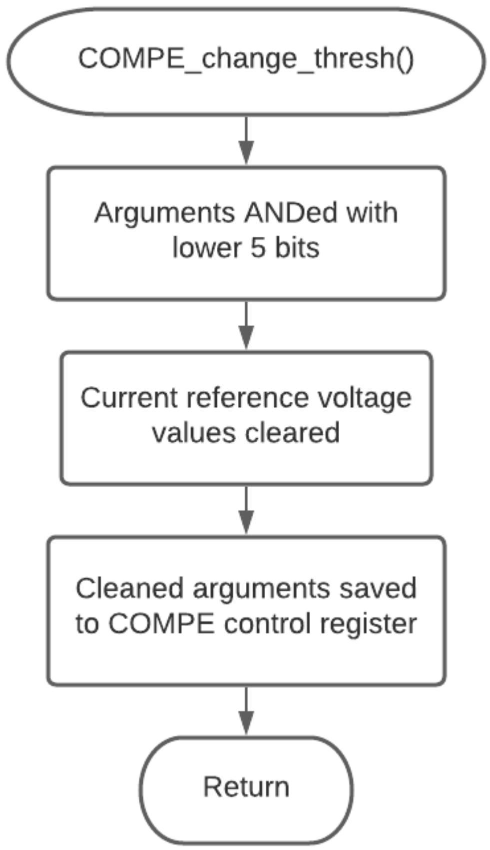

Figure 26: Flowchart of COMPE_change_thresh

Figure 27 describes how frequency is calculated using calculate_frequency. Timer values and variables are initialized and a watchdog timer is started to ensure the system does not hang. The watchdog will reset the system after five seconds have passed. The system waits until the first TA0 ISR is triggered, stores the value into a variable, and sets a flag that will tell the TA0 ISR to stop the TA0 counter. Once the second value is received, it is stored in a variable, the watchdog timer is turned off, and the difference between the two values is found. If value 1 is greater than value 2, the difference will account for an overflow and 1 is subtracted from the overflow counter. The value of the overflow is right shifted by 16 to find the number of overflow clock cycles. The difference and the overflow count is added together to find the total amount of clock cycles between the two rising edges. The clock frequency is divided by this value to find the frequency of the input waveform, and is returned to the caller.

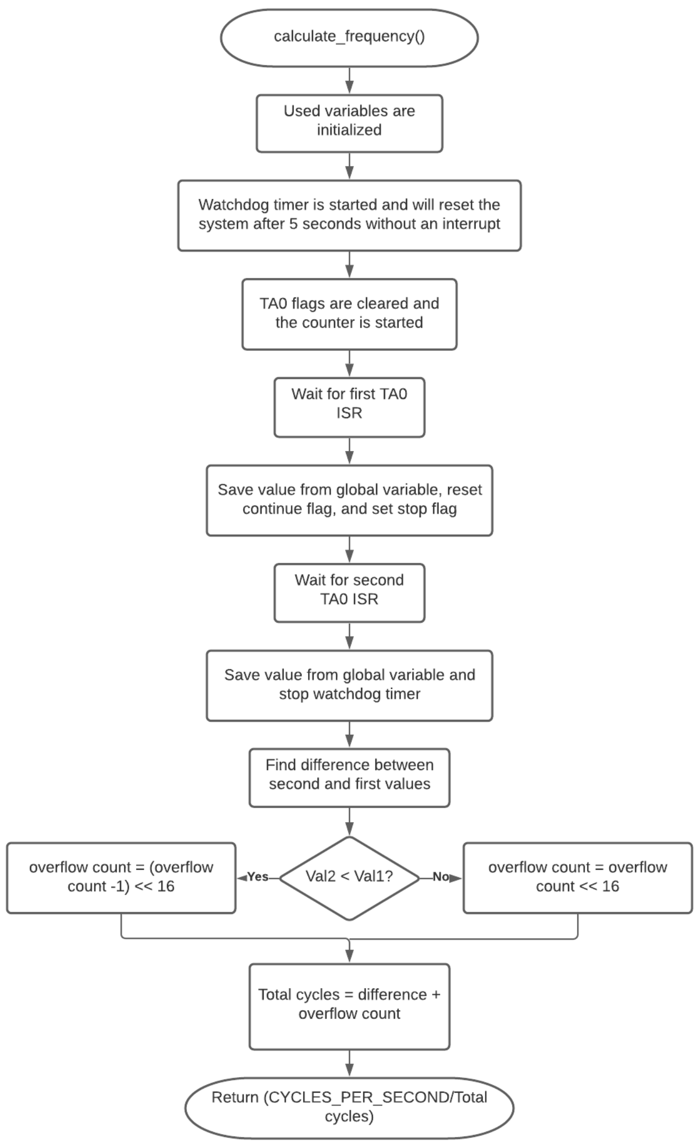

Figure 27: Flowchart of calculate_frequency

Figure 28 describes the flow of the analyze_wave function. Using the measured frequency, the number of cycles need to have an evenly spaced 500 samples is calculated. This value is stored in TA1 and the timer is started. After collecting 500 samples, the timer is stopped and min, max, and RMS variables are initialized. The 500 samples are iterated through again, finding the lowest and highest values in the set, as well as calculating the sum of all the squares. After iterating through the samples a second time, the RMS is divided by the number of samples taken, and the square root is calculated to find the RMS. The min, max, and RMS values are converted to voltages, and the function returns.

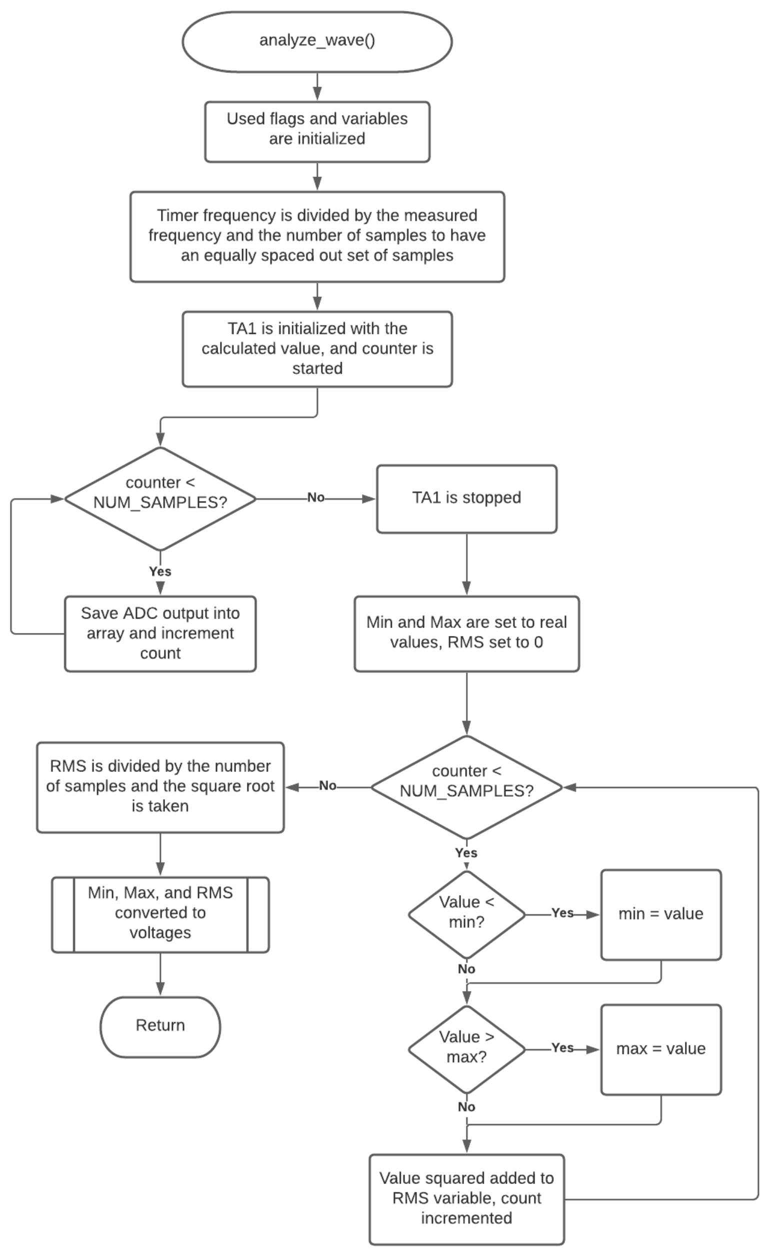

Figure 28: Flowchart of analyze_wave

Appendix

main.c

```
#include "msp.h"
#include "dco.h"
#include "display.h"
#include "uart.h"
#include "adc.h"
#include "compe.h"
#include <math.h>

#define SLOW_TIMER        12000
#define AVG_TIMER         0x0960

#define SLOW_SAMPLES      180
#define AVG_SAMPLES       10000
#define ANALYSIS_SAMPLES  500

#define MVOLT_FIVEBIT     103
#define TIMER_FREQ        24000000
#define OFFSET_ERROR      2

uint8_t output_flag, CCflag, STPflag, received;
uint16_t ADCoutput, CCoutput, min, max;
uint32_t overflow_count;
uint64_t rms;

/* - - - - - ISRs - - - - - */

/*ISR reads received UART character and saves it in a
 * global variable */
void EUSCIA0_IRQHandler (void)
{
	/*Read the character*/
	received = EUSCI_A0->RXBUF;
}

/*ISR for timerA1, CCR0. Begins ADC conversion*/
void TA1_0_IRQHandler(void)
{
	TIMER_A1->CCTL[0] &= ~(TIMER_A_CCTLN_CCIFG);   //Clear the interrupt flag
	ADC14->CTL0 |= ADC14_CTL0_SC;                  //Set the start bit
}

/*TA0 general and CCRN interrupts*/
void TA0_N_IRQHandler(void)
```

```
{
	/*TA0_CCR2 interrupt*/
	if(TIMER_A0-> CCTL[2] & TIMER_A_CCTLN_CCIFG)
	{
		TIMER_A0-> CCTL[2] &= ~(TIMER_A_CCTLN_CCIFG);		//Clear flag

		/*Capture overflow did not occur*/
		if(!(TIMER_A0-> CCTL[2] & TIMER_A_CCTLN_COV))
		{
			CCoutput = TIMER_A0->CCR[2];			//Save output to
			CCflag = 1;					//Set continue flag

			if(STPflag)
				TIMER_A0->CTL &= ~(TIMER_A_CTL_MC_MASK);	// Stop counting and
interrupts
			else
				overflow_count = 0; //Start counting after we get first CCR val
	}

	/*Capture overflow occurred*/
	else
		TIMER_A0-> CCTL[2] &= ~ (TIMER_A_CCTLN_COV);  // Clear Capture Overflow
	}

	/*TA0 general interrupt*/
	if(TIMER_A0-> CTL & TIMER_A_CTL_IFG)
	{
		TIMER_A0-> CTL &= ~(TIMER_A_CTL_IFG);	//Clear flag
		overflow_count++;			//Increment overflow count
	}
}


/*ISR for ADC, saves value and sets flag*/
void ADC14_IRQHandler(void)
{
	output_flag = 1;				//Mark flag for Vavg function
	ADC14->CLRIFGR0 |= ADC14_CLRIFGR0_CLRIFG0;	//Clear interrupt flag
	ADCoutput = ADC14->MEM[0];			//Save most recent value
}

/*Enable interrupts*/
void ISR_enable(void)
{
	/*Interrupts used in receiving characters from the terminal*/
	EUSCI_A0->IE |= EUSCI_A_IE_RXIE;		//Enable receive interrupts
	NVIC->ISER[0] = 1 << EUSCIA0_IRQn;		//Enable interrupts in NVIC
```

```
	/*Interrupts used in TA1 (averaging)*/
	TIMER_A1->CCTL[0] = TIMER_A_CCTLN_CCIE;       //Enable timer interrupts for CCIFG
	NVIC->ISER[0] = 1 << TA1_0_IRQn;              //TA1_CCTL0.CCIFG is 11th in the list

	/*Interrupts used in TA0 (timing square wave)*/
	TIMER_A0->CTL |= TIMER_A_CTL_IE;              //Enable TA0 general interrupts
	TIMER_A0->CCTL[2] |= (TIMER_A_CCTLN_CCIE);  //Enable Capture interrupts in TA0_CCR2
	NVIC->ISER[0] = 1 << TA0_N_IRQn;              // Timer A Interrupt

	/*Interrupts used to read value in ADC14*/
	ADC14->IER0 |= ADC14_IER0_IE0;                //Enable ADC conv complete interrupt
	NVIC->ISER[0] = 1 << ADC14_IRQn;              //Enable ADC interrupts in NVIC

	__enable_irq();                               //Enable Global interrupts

	/*Enable conversions in ADC*/
	ADC14->CTL0 |= ADC14_CTL0_ENC;
}

/* - - - - - Timer Inits - - - - - */

/*This function sets up TA1 to count to 2400. This is done 10,000 times
 * so that we take one full second to */
void average_Timer_Init(void)
{
	/*Set up TimerA*/
	TIMER_A1->CTL = ( TIMER_A_CTL_SSEL__SMCLK    |   //Select SMCLK at 24MHz
	                  TIMER_A_CTL_MC__STOP);     //Halt timer until needed
}

/*This function initializes TA0_CCR2 to capture the rising edge of the comparator*/
void frequency_Timer_Init(void)
{
	TIMER_A0->CTL = ( TIMER_A_CTL_TASSEL_2    |    // SMCLK source
	                  TIMER_A_CTL_ID_0         |
	                  TIMER_A_EX0_TAIDEX_0     |    // No dividers
	                  TIMER_A_CTL_MC__STOP);        // Stop Mode

	TIMER_A0->CCTL[2] = (  TIMER_A_CCTLN_CM__RISING  |   // Capture on rising edge
	                       TIMER_A_CCTLN_CCIS__CCIB   |      // Capture/compare input select CCIxB
	                       TIMER_A_CCTLN_CAP);               // Capture Mode
}

/* - - - - - General Functions - - - - - */

/*This function calculates a quick average
```

```
 * when the multimeter is in DC mode.*/
uint16_t find_Fast_Average(void)
{
	uint8_t i;
	uint32_t average;

	TIMER_A1->R = 0;                                    //Reset timer count
	TIMER_A1->CCR[0] = SLOW_TIMER;                      //Have timer count for 0.5ms
	TIMER_A1->CCTL[0] &= ~(TIMER_A_CCTLN_CCIFG);        //Clear the interrupt flag
	TIMER_A1->CTL |= TIMER_A_CTL_MC__UP;                //Have timer count up

	average = 0; //Reset average

	/*Loop 180 times*/
	for(i = 0; i < SLOW_SAMPLES; i++)
	{
		while(!(output_flag));                      //Wait to capture next value
		average += ADC14_calibrate(ADCoutput);       //Convert value to voltage, add to average
		output_flag = 0;                            //Unset flag
	}

	TIMER_A1->CTL &= ~(TIMER_A_CTL_MC_MASK);            //Set timer in stop mode

	average /= SLOW_SAMPLES;   //Divide by total samples to get average

	return average;
}

/*This function is called to calculate the
 * average voltage of the input signal.
 *
 * It will take 10,000 samples over the
 * course of a second, and average it*/
uint16_t find_Offset(void)
{
	uint16_t i, value;
	uint32_t average;

	TIMER_A1->R = 0;                                    //Reset timer count
	TIMER_A1->CCR[0] = AVG_TIMER;                       //Have timer count up to 2400
	TIMER_A1->CCTL[0] &= ~(TIMER_A_CCTLN_CCIFG);        //Clear the interrupt flag
	TIMER_A1->CTL |= TIMER_A_CTL_MC__UP;                //Have timer count up

	average = 0; //Reset average

	/*Loop 10,000 times*/
	for(i = 0; i < AVG_SAMPLES; i++)
```

```
	{
		while(!(output_flag));				//Wait to capture next value
		value = ADC14_calibrate(ADCoutput);	//Convert value to voltage
		average += value;					//Add val into average
		output_flag = 0;					//Unset flag
	}

	TIMER_A1->CTL &= ~(TIMER_A_CTL_MC_MASK);	//Set timer in stop mode

	average /= AVG_SAMPLES;   //Divide by total samples to get average

	return average;
}

/*This function times the rising edge of the comparator rising edge
 * and calculates the frequency of the input wave*/
uint16_t calculate_frequency(void)
{
	uint16_t value1, value2, diff;
	uint32_t value;

	/*Initialize variables used*/
	CCflag = 0;
	STPflag = 0;

	WDT_A->CTL = WDT_A_CTL_PW		|	//Watchdog password
		WDT_A_CTL_SSEL__SMCLK	|	//SMCLK source
		WDT_A_CTL_IS_1;			//Reset after 5 seconds


	TIMER_A0->R = 0;						//Reset Timer A0 count
	TIMER_A0->CCTL[2] &= ~(TIMER_A_CCTLN_CCIFG);	//Clear Capture flag
	TIMER_A0->CTL &= ~(TIMER_A_CTL_IFG);		//Clear TA0 General flag
	TIMER_A0-> CCTL[2] &= ~ (TIMER_A_CCTLN_COV);	//Clear capture overflow just in case
	TIMER_A0->CTL |= (TIMER_A_CTL_MC__CONTINUOUS);	//Count up continuously

	while(!CCflag);	//Wait until we capture CCR value
	value1 = CCoutput;	//Save output into variable
	CCflag = 0;		//Reset continue flag
	STPflag = 1;		//Notify ISR next value is last

	while(!CCflag);	//Wait until we capture CCR value
	value2 = CCoutput;	//Save output into variable

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

	diff = value2 - value1;
```

```
	if(value2 < value1)
		overflow_count = ((overflow_count - 1) << 16);
	else
		overflow_count = (overflow_count << 16);

	value = diff + overflow_count;

	return (TIMER_FREQ/value) + 1;
}

/*This function analyzes the input waveform. After
 * taking 600 samples over the course of one period
 * (as determined by the measured frequency), the
 * RMS, min, and max voltages are calculated*/
void analyze_wave(uint16_t freq)
{
	uint16_t i, values[ANALYSIS_SAMPLES];
	uint32_t CCcount;

	/*Calculate value CCR to count to*/
	CCcount = (TIMER_FREQ / freq);
	CCcount /= ANALYSIS_SAMPLES;

	TIMER_A1->R = 0;                                //Reset timer count
	TIMER_A1->CCR[0] = CCcount;                     //Have timer count up to calculated value
	TIMER_A1->CCTL[0] &= ~(TIMER_A_CCTLN_CCIFG);    //Clear the interrupt flag
	TIMER_A1->CTL |= TIMER_A_CTL_MC__UP;            //Have timer count up

	/*Loop 500 times, save into array for sampling*/
	for(i = 0; i < ANALYSIS_SAMPLES; i++)
	{
		while(!(output_flag));    //Wait to capture next value
		values[i] = ADCoutput;    //Save value into array
		output_flag = 0;          //Unset flag
	}

	TIMER_A1->CTL &= ~(TIMER_A_CTL_MC_MASK);        //Set timer in stop mode

	/*Initialize voltages*/
	min = values[0];
	max = values[0];
	rms = 0;

	/*Iterate through values again*/
	for(i = 0; i < ANALYSIS_SAMPLES; i++)
	{
```

```
		/*Grab new max and min values*/
		if(values[i] < min)
			min = values[i];

		if(values[i] > max)
			max = values[i];

		/*Save squares*/
		rms += (values[i] * values[i]);
	}

	rms /= ANALYSIS_SAMPLES;  //divide by number of samples

	rms = sqrt(rms); //Take the square root

	/*Convert values to voltages*/
	min = ADC14_calibrate(min);
	max = ADC14_calibrate(max);
	rms = ADC14_calibrate(rms);
}

/**
 * main.c
 */
void main(void)
{
	uint8_t FiveBitLo, FiveBitHi, i;
	uint16_t offset, freq, Vpp;

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

	/*Set system clock to 24MHz*/
	set_DCO(FREQ_24_MHz);

	typedef enum {
		DC,
		AC,
		TO_AC,
		TO_DC
	} states;

	states state = DC;

	/*Initializer peripherals*/
	UART_init();
	ADC14_init();
	COMPE_init();
```

```
    average_Timer_Init();
    frequency_Timer_Init();

    DISP_makeUI();

    /*Initialize global variables*/
    output_flag = 0;

    /*Enable interrupts*/
    ISR_enable();

  while(1)
  {
    switch(state)
    {
          /*DC Mode, find an average over the course of 90 ms
          * Display this average on the terminal in the given
          * field.*/
          case DC:
                 offset = find_Fast_Average();   //Calculate average

                DISP_update_VAVG(offset);        //Update Vavg in UI

                DISP_update_DCBAR(offset);       //Update the plot

                /*Check for input character*/
                if(received == 'A')
                    state = TO_AC;

                break;

          /*AC Mode, find a stable offset, calculate threshold
          * voltages for the comparator, measure frequency of
          * the comparator output, use this frequency to calculate
          * Vrms and Vpp, display frequency, Vrms, and Vpp in their
          * respective fields*/
          case AC:
                offset = 0;

                for(i = 0; i < 3; i++)
                      offset += find_Offset();

                offset /= 3;

                /*Calculate threshold voltages*/
                FiveBitLo = ((offset * 10) / MVOLT_FIVEBIT);        //High to low
                FiveBitHi = ((offset * 10) / MVOLT_FIVEBIT) + 1;   //Low to high
```

```
                COMPE_change_thresh(FiveBitLo, FiveBitHi);    //Change the threshold voltages

                freq = calculate_frequency();   //Calculate frequency
                DISP_update_FREQ(freq);          //Update frequency on UI

                analyze_wave(freq);       //Analyze period of wave at measured frequency

                Vpp = max - min;    //Calculate peak to peak voltage

                /*Update UI with new values*/
                DISP_update_VRMS(rms);
                DISP_update_VPP(Vpp);
                DISP_update_ACBAR(rms);

                /*Check for input character*/
                if(received == 'D')
                    state = TO_DC;

                break;

            /*Transition from AC to DC mode, clears AC fields*/
            case TO_DC:
                DISP_clear_AC();
                state = DC;
                break;

            /*Transition from DC to AC mode, clears DC fields*/
            case TO_AC:
                DISP_clear_DC();
                state = AC;
                break;

            default:
                state = DC;
        }
    }
}
```

uart.h

```
/*This library contains macros and functions necessary
 * to utilize the UART module on the MSP432.
 *
 * The baud rate chosen for this library is 115.2 kb/s
 *
 * REQUIRES INPUT CLK OF 3MHZ
 *
 *     Author: Zarek Lazowski & Spencer Stone
 */

#ifndef UART_H_
#define UART_H_

#ifndef MSP_H_
#include "msp.h"
#endif

#define UART_BR_SCALER  13
#define UART_BR_FREQ      0
#define UART_BR_STAGE   0x25

#define UART_PORT P1
#define UART_RXD BIT2
#define UART_TXD BIT3

#define UART_ESC 0x1B

/*List of useful escape commands*/
//Movement
#define UART_U1     "[1A"   //up
#define UART_U5     "[5A"
#define UART_D1     "[1B"   //down
#define UART_D5     "[5B"
#define UART_R1     "[1C"   //right
#define UART_R5     "[5C"
#define UART_L1     "[1D"   //left
#define UART_L5     "[5D"
#define UART_HOME    "[H"   //top left

//Effects
#define UART_EOFF    "[0m"   //effects off
#define UART_CLEAR   "[2J"   //clear screen
#define UART_BLINK   "[5m"   //following text blinks

//RGB text
```

```
#define UART_CRED   "[31m"
#define UART_CGREEN "[32m"
#define UART_CBLUE  "[34m"
#define UART_CWHITE "[37m"

void UART_init(void);
void UART_esc_code(uint8_t *code);
void UART_print_str(uint8_t *str);
void UART_print_char(uint8_t letter);

#endif /* UART_H_ */
```

uart.c

```
#include "uart.h"

/*This function initializes the MSP432's eUSCI-A
 * options to run at 115.2 kb/s and enables
 * the UART GPIO pins*/
void UART_init(void)
{
	EUSCI_A0->CTLW0 |= EUSCI_A_CTLW0_SWRST; /*Put module into software reset*/

	/*Configure eUSCI-A*/
	EUSCI_A0->CTLW0 = ( EUSCI_A_CTLW0_SWRST       | //Software reset
	                    EUSCI_A_CTLW0_MODE_0      | //Select UART mode
	                    EUSCI_A_CTLW0_SSEL__SMCLK   | //Select SMCLK
	                    EUSCI_A_CTLW0_SPB);       //Send two stop bits just in case

	EUSCI_A0->BRW = UART_BR_SCALER;

	EUSCI_A0->MCTLW = ( EUSCI_A_MCTLW_OS16                          | //Enable oversampling
	                    (UART_BR_FREQ << EUSCI_A_MCTLW_BRF_OFS)     | //Set Baud Rate Frequency
	                    (UART_BR_STAGE << EUSCI_A_MCTLW_BRS_OFS));    //Set Baud Rate Modulation
Stage

	/*Configure UART pins*/
	UART_PORT->SEL0 |= (UART_RXD | UART_TXD); /*Select UART for RXD and TXD pins*/
	UART_PORT->SEL1 &= ~(UART_RXD | UART_TXD);

	EUSCI_A0->CTLW0 &= ~(EUSCI_A_CTLW0_SWRST); /*Exit software reset*/
}

void UART_esc_code(uint8_t *code)
{
	/*Send escape character to terminal*/
	UART_print_char(UART_ESC);

	UART_print_str(code);
}

void UART_print_str(uint8_t *str)
{
	int i;

	/*Continue calling the print char function with contents of string until
	* we reach a nul byte*/
	for(i = 0; str[i] != '\0'; i++)
	UART_print_char(str[i]);
```

```
}

void UART_print_char(uint8_t letter)
{
	/*Wait for transmit ifg*/
	while(!(EUSCI_A0->IFG & EUSCI_A_IFG_TXIFG));

	/*Transmit byte*/
	EUSCI_A0->TXBUF = letter;
}
```

adc.h

```c
/*This library offers an initialization function
 * that configures the MSP432's 14-bit ADC unit to
 * operate with a single channel, doing single conversions
 * based on values read from P5.4.
 *
 * This library uses the 3.3 V rail as a reference,
 * and samples the input for 4 clock cycles.
 *
 *  Created on: May 20, 2021
 *     Author: Zarek & Armin
 */

#ifndef ADC_H_
#define ADC_H_

#ifndef MSP_H_
#include "msp.h"
#endif

#define ADC_PORT P5
#define ADC_INPUT BIT4

/*Calibration values (y = mx + b)*/
#define ADC_M 2
#define ADC_B -280
#define ADC_ZOOM 100 //Division to get in terms of 10*millivolts

void ADC14_init(void);
uint16_t ADC14_calibrate(uint16_t num);

#endif /* ADC_H_ */
```

adc.c

```c
#include "adc.h"

/*This function initializes the input pins for the ADC on the
 * MSP432 and configures the control register to the below options*/
void ADC14_init(void)
{
	ADC_PORT->SEL0 |= ADC_INPUT;   // Configure P5.4 for ADC analog input pin
	ADC_PORT->SEL1 |= ADC_INPUT;
	ADC_PORT->DIR &= ~(ADC_INPUT);

	ADC14->CTL0 = (ADC14_CTL0_ON            //Turns on the ADC
		|  ADC14_CTL0_SHT0_0       //Sample and hold for 4 clk cycles
		|  ADC14_CTL0_CONSEQ_0     //Single channel, single conversion
		|  ADC14_CTL0_SSEL__SMCLK   //Select SMCLK source
		|  ADC14_CTL0_DIV__1       //No clock dividing
		|  ADC14_CTL0_SHP);        //Sample and hold mode

	ADC14->CTL1 = (ADC14_CTL1_RES__14BIT);  //14 bit conversion mode

	//Single ended mode (ADC14DIF = 0) input: A1 (5.4)
	//3.3 V reference
	ADC14->MCTL[0] = (ADC14_MCTLN_INCH_1 | ADC14_MCTLN_VRSEL_0);
}

uint16_t ADC14_calibrate(uint16_t num)
{
	/*Special case: calibrated 0 to start at 334*/
	if(num < 334)
	return 0;

	/*y = mx + b*/
	else
	return ( ((ADC_M*num) + ADC_B) / ADC_ZOOM );
}
```

compe.h

```c
/*This library provides useful macros and functions
 * for interacting with the Comparator E 0 module.
 *
 * This module takes an input clock at 24 MHz, takes
 * and input signal at P8.0 and is compared to
 * voltage generated by the included reference voltage
 * generator.
 *
 * The output of this comparator can be observed on P7.1
 *
 *  Created on: May 29, 2021
 *     Author: Janelle & Zarek
 */

#ifndef COMPE_H_
#define COMPE_H_

#ifndef MSP_H_
#include "msp.h"
#endif

/*Port for + input (C1) of CE0*/
#define COMPE_INPORT P8
#define COMPE_INPIN  BIT0

/*Port for analyzing the output of CE0*/
#define COMPE_OUTPORT  P7
#define COMPE_OUTPIN   BIT1

/*Used for anding with desired threshold voltages*/
#define COMPE_FIVEBITS 0x1F

void COMPE_init(void);
void COMPE_change_thresh(uint8_t lo, uint8_t hi);
void COMPE_debug(void);

#endif /* COMPE_H_ */
```

compe.c

```
#include "compe.h"

/*This function initializes the Comparator E 0 module to take
 * an input from Analog Input 1, and compares it to voltages
 * from CE1's reference voltage generator*/
void COMPE_init(void)
{

      COMP_E0->CTL0 |= (COMP_E_CTL0_IPEN      |      // Analog input V+ terminal is enabled
                COMP_E_CTL0_IPSEL_1);           // Choose inchannel C1 (P8.0)

      COMP_E0->CTL1 |= (COMP_E_CTL1_ON        |      // Turns comparator on
                COMP_E_CTL1_FDLY_3  |      // Filter Delay 3000ns
                COMP_E_CTL1_F);                 // Comparator filter on

      COMP_E0->CTL2 |= (COMP_E_CTL2_RS_1      |      // VCC applied to the resistor ladder
                COMP_E_CTL2_RSEL);              // VREF is applied to the V- terminal;

      COMP_E0->INT  |= (COMP_E_INT_IE);              // Comparator output interrupt enable

      COMPE_INPORT->SEL0 |= (COMPE_INPIN);           // Positive Select
      COMPE_INPORT->SEL1 |= (COMPE_INPIN);
}

/*This function sets new voltages for the CE0 threshold
 * voltages. This is done by cleaning the input to ensure
 * only 5 bits are written, unsetting bits in the REF0/REF1
 * portion of the control register, and setting the requested
 * bits.*/
void COMPE_change_thresh(uint8_t lo, uint8_t hi)
{
      uint8_t FiveBitLo, FiveBitHi;

      /*Clean input to be only 5-bit*/
      FiveBitLo = lo & COMPE_FIVEBITS;
      FiveBitHi = hi & COMPE_FIVEBITS;

      /*Reset reference voltage values*/
      COMP_E0->CTL2 &= ~(COMP_E_CTL2_REF1_MASK | COMP_E_CTL2_REF0_MASK);

      /*Sets up hi-to-lo and lo-to-hi voltages for CE0*/
      COMP_E0->CTL2 |= (FiveBitLo << COMP_E_CTL2_REF1_OFS);  //Reference voltage is low
      COMP_E0->CTL2 |= (FiveBitHi << COMP_E_CTL2_REF0_OFS);  //Reference voltage is High
}
```

```
/*This function will set up P7.1 as an output of the comparator
 * E 0 module, for use in debugging*/
void COMPE_debug(void)
{
	/*Setup output to analyze square wave*/
	COMPE_OUTPORT->SEL0 |=	COMPE_OUTPIN;
	COMPE_OUTPORT->SEL1 &=	~(COMPE_OUTPIN);
	COMPE_OUTPORT->DIR  |=	COMPE_OUTPIN;
}
```

display.h

```
/*This library contains functions that set up,
 * edit, and navigate the UI in the terminal.
 *
 *  Created on: Jun 2, 2021
 *      Author: Zarek
 */

#ifndef DISPLAY_H_
#define DISPLAY_H_

#ifndef MSP_H_
#include "msp.h"
#endif

#ifndef UART_H_
#include "uart.h"
#endif

/*Different text in the terminal*/
#define DISP_TITLE        "Mr. MultiMeter Man"
#define DISP_OPTIONS      "D-DC | A-AC"
#define DISP_BAR_LABEL  "0V     1V     2V     3V"
#define DISP_MSG_1        "Mr. MultiMeter Man is your Main Method"
#define DISP_MSG_2        "Man for Measuring voltage & frequency!"

/*Locations on the terminal*/
#define DISP_TOPLEFT      "[2;2H"

#define DISP_DC_BOX       "[3;3H"
#define DISP_VAVG_LABEL "[5;4H"
#define DISP_VAVG_VAL   "[5;10H"
#define DISP_DC_BAR       "[4;21H"
#define DISP_DC_LABEL   "[5;20H"
#define DISP_DC_PLOT      "[3;18H"

#define DISP_AC_BOX       "[7;3H"
#define DISP_VRMS_LABEL "[9;4H"
#define DISP_VRMS_VAL   "[9;10H"
#define DISP_VPP_LABEL  "[11;4H"
#define DISP_VPP_VAL      "[11;10H"
#define DISP_AC_BAR       "[8;21H"
#define DISP_AC_LABEL   "[9;20H"
#define DISP_AC_PLOT      "[7;18H"

#define DISP_FREQ_BOX   "[13;3H"
```

```
#define DISP_FREQ_VAL   "[15;10H"
#define DISP_OPT_LOC      "[18;3H"

#define DISP_MSG_LOC_1  "[13;20H"
#define DISP_MSG_LOC_2  "[15;20H"
#define DISP_MSG_LOC_3  "[17;20H"
#define DISP_MSG_LOC_4  "[19;20H"

/*Starting location of boarders*/
#define DISP_INFO_B1      "[2;3H"
#define DISP_INFO_B2      "[6;3H"
#define DISP_INFO_B3      "[12;3H"
#define DISP_INFO_B4      "[16;3H"
#define DISP_PLOT_B1      "[2;18H"
#define DISP_PLOT_B2      "[6;18H"
#define DISP_PLOT_B3      "[10;18H"
#define DISP_LEFT_BORD  "[3;2H"
#define DISP_MID_BORD   "[2;17H"
#define DISP_RIGHT_BORD "[3;54H"

/*Length of various borders*/
#define DISP_PLOT_LEN   36
#define DISP_BAR_LEN      30
#define DISP_INFO_LEN   14
#define DISP_RIGHT_HT   8
#define DISP_MID_HT       7

void DISP_makeUI(void);

void DISP_update_VAVG(uint16_t val);
void DISP_update_DCBAR(uint16_t val);
void DISP_clear_DC(void);

void DISP_update_VRMS(uint16_t val);
void DISP_update_VPP(uint16_t val);
void DISP_update_ACBAR(uint16_t val);
void DISP_clear_AC(void);

void DISP_update_FREQ(uint16_t val);

void DISP_printDown(uint8_t letter);
void DISP_printVolts(uint16_t val);
void DISP_printFreq(uint16_t val);

#endif /* DISPLAY_H_ */
```

display.c

```c
#include "display.h"

/*This function goes to the predefined location
 * to print the given Vavg*/
void DISP_update_VAVG(uint16_t val)
{
	UART_esc_code(DISP_VAVG_VAL);
	DISP_printVolts(val);
}

/*This function updates the Vavg bar graph*/
void DISP_update_DCBAR(uint16_t val)
{
	uint8_t i, boxes;

	/*Go to beginning of plot*/
	UART_esc_code(DISP_DC_BAR);

	/*Divide by 10 to figure out how many boxes are required*/
	boxes = val / 10;

	for(i = 0; i < DISP_BAR_LEN; i++)
	{
	/*Print character for every 1/10 volt*/
	if(i < boxes)
		UART_print_char('#');
	/*If no more boxes, print blanks*/
	else
		UART_print_char(' ');
	}
}

/*This function blanks out the DC portions of the
 * UI when switching to AC mode*/
void DISP_clear_DC(void)
{
	uint8_t i;

	UART_esc_code(DISP_VAVG_VAL);

	for(i = 0; i < 6; i++)
	UART_print_char(' ');

	UART_esc_code(DISP_DC_BAR);
```

```
	for(i = 0; i < DISP_BAR_LEN; i++)
	UART_print_char(' ');
}

/*This function goes to the predefined location
 * to print the given Vrms*/
void DISP_update_VRMS(uint16_t val)
{
	UART_esc_code(DISP_VRMS_VAL);
	DISP_printVolts(val);
}

/*This function goes to the predefined location
 * to print the given Vpp*/
void DISP_update_VPP(uint16_t val)
{
	UART_esc_code(DISP_VPP_VAL);
	DISP_printVolts(val);
}

/*This function updates the Vrms bar graph*/
void DISP_update_ACBAR(uint16_t val)
{
	uint8_t i, boxes;

	/*Go to beginning of plot*/
	UART_esc_code(DISP_AC_BAR);

	/*Divide by 10 to figure out how many boxes are required*/
	boxes = val / 10;

	for(i = 0; i < DISP_BAR_LEN; i++)
	{
	/*Print character for every 1/10 volt*/
	if(i < boxes)
		UART_print_char('#');
	/*If no more boxes, print blanks*/
	else
		UART_print_char(' ');
	}
}


/*This function goes to the predefined location
 * to print the given frequency*/
void DISP_update_FREQ(uint16_t val)
{
```

```
	UART_esc_code(DISP_FREQ_VAL);
	DISP_printFreq(val);
}

/*This function blanks out the DC portions of the
 * UI when switching to AC mode*/
void DISP_clear_AC(void)
{
	uint8_t i;

	UART_esc_code(DISP_VRMS_VAL);

	for(i = 0; i < 6; i++)
	UART_print_char(' ');

	UART_esc_code(DISP_VPP_VAL);

	for(i = 0; i < 6; i++)
	UART_print_char(' ');

	UART_esc_code(DISP_AC_BAR);

	for(i = 0; i < DISP_BAR_LEN; i++)
	UART_print_char(' ');
}

/*Print character and move to location one below*/
void DISP_printDown(uint8_t letter)
{
	UART_print_char(letter);  //Print char
	UART_esc_code(UART_L1);   //Go back
	UART_esc_code(UART_D1);   //Go down
}

/*Print voltage from converted value. Supplied values are in
 * terms of 10's of millivolts.*/
void DISP_printVolts(uint16_t val)
{
	uint8_t hundred, ten, one;

	hundred = (val/100);                   //Val in 100's place
	ten = (val/10) - (hundred*10);         //Val in 10's place
	one = val - (hundred*100) - (ten*10);   //Val in 1's place

	UART_print_char(hundred + 0x30);
	UART_print_char('.');
	UART_print_char(ten + 0x30);
```

```
	UART_print_char(one + 0x30);
	UART_print_str(" V");
}

/*Print given frequency.*/
void DISP_printFreq(uint16_t val)
{
	uint8_t thousand, hundred, ten, one;

	thousand = val/1000;                                                    //Val in 1000's place
	hundred = (val/100) - (thousand * 10);                                  //Val in 100's place
	ten = (val/10) - (thousand * 100) - (hundred * 10);                     //Val in 10's place
	one = val - (thousand * 1000) - (hundred * 100) - (ten * 10);   //Val in 1's place

	UART_print_char(thousand + 0x30);
	UART_print_char(hundred + 0x30);
	UART_print_char(ten + 0x30);
	UART_print_char(one + 0x30);
	UART_print_str(" Hz");
}

/*This function creates the skeleton that forms
 * the UI*/
void DISP_makeUI(void)
{
	uint8_t i;

	/*Clear the screen*/
	UART_esc_code(UART_HOME);
	UART_esc_code(UART_CLEAR);

	/*Print the title*/
	UART_print_str(DISP_TITLE);

	/*Print mode options*/
	UART_esc_code(DISP_OPT_LOC);
	UART_print_str(DISP_OPTIONS);

	/*Print message*/
	UART_esc_code(DISP_MSG_LOC_1);
	UART_print_str(DISP_MSG_1);

	UART_esc_code(DISP_MSG_LOC_2);
	UART_print_str(DISP_MSG_2);


	/*Start printing borders*/
```

```
/*Left side border*/
UART_esc_code(DISP_LEFT_BORD);

for(i = 0; i < DISP_INFO_LEN; i++)
DISP_printDown('|');

/*Top of information border*/
UART_esc_code(DISP_INFO_B1);

for(i = 0; i < DISP_INFO_LEN; i++)
UART_print_char('_');

/*Second info border*/
UART_esc_code(DISP_INFO_B2);

for(i = 0; i < DISP_INFO_LEN; i++)
UART_print_char('_');

/*Third info border*/
UART_esc_code(DISP_INFO_B3);

for(i = 0; i < DISP_INFO_LEN; i++)
UART_print_char('_');

/*Fourth info border*/
UART_esc_code(DISP_INFO_B4);

for(i = 0; i < DISP_INFO_LEN; i++)
UART_print_char('_');

/*Top of plot border*/
UART_esc_code(DISP_PLOT_B1);

for(i = 0; i < DISP_PLOT_LEN; i++)
UART_print_char('_');

/*Second plot border*/
UART_esc_code(DISP_PLOT_B2);

for(i = 0; i < DISP_PLOT_LEN; i++)
UART_print_char('_');

/*Third plot border*/
UART_esc_code(DISP_PLOT_B3);

for(i = 0; i < DISP_PLOT_LEN; i++)
UART_print_char('_');
```

```
/*Right side border*/
UART_esc_code(DISP_RIGHT_BORD);

for(i = 0; i < DISP_RIGHT_HT; i++)
DISP_printDown('|');

/*Middle border*/
UART_esc_code(DISP_MID_BORD);
DISP_printDown('_');
UART_esc_code(UART_D1);
DISP_printDown('|');
UART_esc_code(UART_D1);
DISP_printDown('_');
UART_esc_code(UART_D1);
DISP_printDown('|');
UART_esc_code(UART_D1);

for(i = 0; i < DISP_MID_HT; i++)
DISP_printDown('|');


/*Make the boxes where information and plots are located*/
/*DC information*/
UART_esc_code(DISP_DC_BOX);
UART_print_str("DC:");          //Print box title

UART_esc_code(DISP_VAVG_LABEL);
UART_print_str("Vavg:");        //Print Vavg label

UART_esc_code(DISP_DC_LABEL);
UART_print_str(DISP_BAR_LABEL); //Print Vavg plot label

/*AC information*/
UART_esc_code(DISP_AC_BOX);
UART_print_str("AC:");          //Print box title

UART_esc_code(DISP_VRMS_LABEL);
UART_print_str("Vrms:");        //Print Vrms label

UART_esc_code(DISP_VPP_LABEL);
UART_print_str("Vp-p:");        //Print Vpp label

UART_esc_code(DISP_AC_LABEL);
UART_print_str(DISP_BAR_LABEL); //Print Vrms plot label

/*Frequency information*/
```

```
	UART_esc_code(DISP_FREQ_BOX);
	UART_print_str("Frequency:");	//Print box title
}
```

References

[1] Texas Instruments, "MSP432P401 Datasheet," June 2019. [Online]. Available: https://www.ti.com/lit/ds/slas826h/slas826h.pdf?ts=1618956869111. [Accessed April 2021].

[2] Texas Instruments, "MSP432P4xx Technical Reference Manual," June 2019. [Online]. Available: https://www.ti.com/lit/ug/slau356i/slau356i.pdf?ts=1618923921408. [Accessed April 2021].