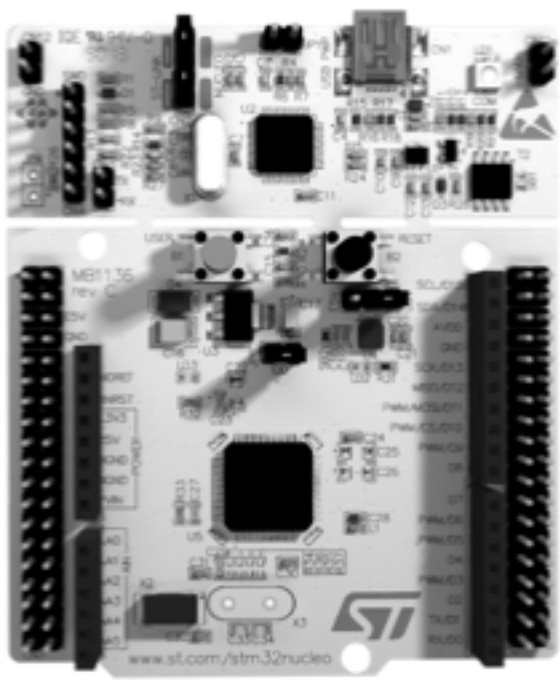
PROJECT #3 – DIGITAL MULTIMETER

Cal Poly SLO | EE 329 - 01 | Professor Paul Hummel



Ethan Clark Najmy

Spring Quarter | June 6, 2022

[1] Photo taken from STM32 Lab Manual

1. Behavior Description

The digital multimeter created in this project consists of a low pass filter, the STM32L4A6ZGT6 board, and an on-board ADC and comparator. The project functions as an AC and DC digital multimeter that can measure various voltages and waveforms. The multimeter will display the voltage in DC mode and will display the peak-to-peak voltage, RMS voltage, and frequency in AC mode. The multimeter displays the output on a computer's serial display.

The multimeter will power up into the DC mode and will begin immediately measuring the DC voltage and displaying it on a bar graph. This bar graph and voltage will be displayed on the computer's serial display. At any point the multimeter may be changed into AC mode by typing "A" and pressing enter on the terminal.

When in AC mode, the multimeter will measure frequency and voltage and display it on the serial display. The terminal will display frequency, peak-to-peak voltage, and true RMS voltage. Also displayed is a bar graph representing the RMS voltage. At any time, the user may switch back to DC mode by typing "D" and pressing enter.

2. System Specifications

Table 1. System Specifications

STM32L4A6ZGT6 Power Specifications	
Supply Voltage	-0.3V – 4V
Current Draw	150mA (max.)
Power Consumption	0.6W (max.)
Power Connection	Micro USB cable (not included)
Digital Multimeter Specifications	
Input Voltage Range	0V – 3V
Frequency Range	1Hz – 1kHz
RMS Calculation	True RMS
Terminal Specifications	
Serial Baud Rate	115200
Set DC Mode	“D” + Enter
Set AC Mode	“A” + Enter

** Serial baud rate must match between personal computer and digital multimeter.*

3. System Schematic

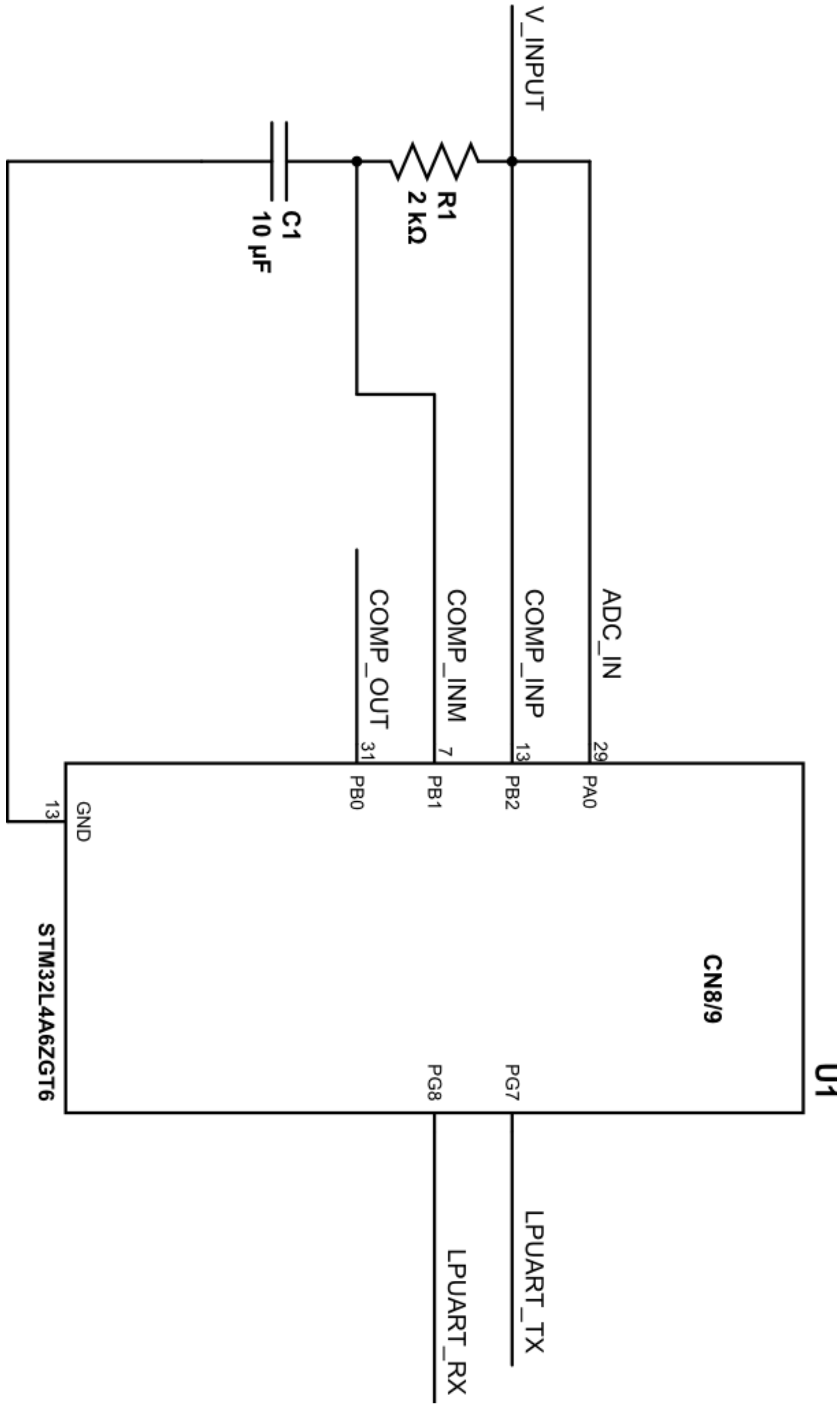


Figure 1. System Schematic

4. Software Architecture

4.1. Overview

The system functions through the use of a software-implemented Finite State Machine (FSM), built-in timers, interrupts, the LPUART, and the built-in analog-to-digital converter (ADC). When powered on, the system initializes the global variables, creates the FSM and states, initializes the ADC, LPUART, comparator, and timers, then enters the FSM. The system will toggle between the FSM and interrupt handler while powered on and will switch between the AC and DC FSM states when commanded to while constantly displaying the voltages to the LPUART. The key aspect of this system functioning is use of the built-in timers and interrupts to measure frequency and to sample voltages. These will be covered in a later section.

4.1.1 Circuitry

This project functions using a comparator as mentioned previously. The comparator takes in two inputs, V_{in}, which is the signal that is measured, and V_{REF}, which is a reference voltage for the comparator to use and create the square wave.

To obtain this V_{REF} value, a low pass filter was constructed using a 2kΩ resistor and a 10µF capacitor. The V_{in} to the comparator and ADC was connected to the open end of the resistor and the V_{REF} of the comparator was connected in between the resistor and capacitor. This allowed for proper V_{REF} voltage and eliminated the need for software to calculate the voltage values.

4.1.2. DC Mode

This project works in 2 main phases, either DC or AC mode. DC mode is more straight forward compared to AC mode. The system powers up in DC mode, and then begins by setting the global IRQ state variable, `stateVal`. This variable used to let the IRQ handler perform operations depending on which mode is currently set (`stateVal = 1` – DC mode, `stateVal = 2` – AC mode). Then the system sets the CCR1 timer to operate every 120,000 clock cycles. The STM board operates at 24MHz, so with the CCR1 timer operating every 120,000 clock cycles and the program obtaining 200 samples in DC mode, it takes approximately 1.6ms to obtain these samples. The system then calls the `ADCConvert()` function which is runs a conversion and obtains a voltage every time CCR1 is triggered. After ~1.6ms, it will obtain it's 200 values, find the approximate DC voltage, and output this to the terminal.

4.1.3. AC Mode

In AC mode, the system first measures frequency. This frequency measurement is done through the use of TIM2 CCR4 which is directly linked to the comparator output. Every rising edge of the comparator, CCR4 saves that value. The system waits until 2 rising edges of the comparator output have been captured, saves those values, then subtracts them determining the period of the wave in clock cycles. This period is then converted into frequency by dividing the clock speed, 24MHz by this period.

The program then moves on and captures voltages, similarly to the DC mode. The difference however is the CCR1 value that triggers the ADC conversion is set to the period of the input waveform to allow adequate voltage measurements over one period of the wave.

4.2. Initialization

The system initializes the global variables and the FSM when booted up. Global variables are used in this system to allow the interrupt handler function access to these necessary variables. The FSM is created through the use of the `typedef` keyword (source code can be found in *Appendix B*), therefore creating a custom data type and creating the individual states. After initialization of the FSM, the system initializes the ADC, LPUART, and comparator by using `ADC_init()`, `LPUART_init()` and `Comparator_init()` functions. In each of these functions, each peripheral is initialized as appropriate.

4.2.1. ACD Initialization

The ADC is initialized through the use of the `ADC_init()` function. In this initialization, the on-board ADC is configured for our use and follows a strict setup procedure.

First, the ADC clock is enabled and set. Then the ADC must be powered on and taken out of deep power down mode. Then the program must wait a minimum of 20µs before continuing.

Then the program continues by calibrating the ADC. This calibration occurs automatically by setting `ADC_CR_ADCAL` bit in the `ADC_CR` register.

Once this calibration occurs, the program sets the ADC sampling rate. In our case, we want the ADC to sample as quick as possible, so we set it to 2.5 clock cycles per sample. The program then configures the ADC for regular sequence, single conversion, 12-bit resolution, right aligned, software triggered. Lastly, ADC interrupts are enabled.

Once the ADC itself has been configured, the GPIO PA0 (`ADCINN5`) is configured in analog mode with low speed. The initialization program is then finished.

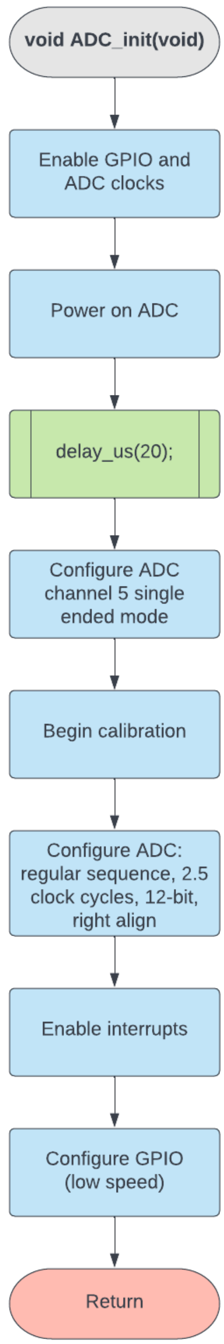


Figure 2. ADC Initialization Flowchart

4.2.2. LPUART Initialization

The LPUART configuration is relatively simple and consists mainly of setting GPIO ports. The STM board uses RS-232 for LPUART communications. RS-232 allows for asynchronous communication between the STM board and an external peripheral, in our case a computer with a USB virtual COM port.

To enable this asynchronous communication, the STM GPIO TX and RX pins must be enabled and configured correctly. This project uses GPIO PG7 and PG8 for RS-232 TX and RX communications.

To enable these pins, the GPIO G clock and LPUART clock must be enabled. Next, LPUART is powered on and the GPIO_AF registers are configured for alternate function mode. Next, the GPIO pins are set to push-pull mode, low speed, with no pull-up or pull-down resistors. Then, the GPIO pins are set to alternate function mode.

(NOTE: GPIO pins must be set to AF mode last.)

Next, the LPUART must be configured. Fortunately, much of the configuration does not need to be changed from the reset value and only the baud rate must be set. The baud rate is determined using the following formula:

$$\text{Tx/Rx baud} = \frac{256 \times f_{CK}}{\text{LPUARTDIV}}$$

Our clock speed is 24MHz and Tx/Rx baud is 115,200 bits/second, therefore we have a baud rate of 53,333.

Lastly, interrupts and LPUART data TX and RX is enabled.

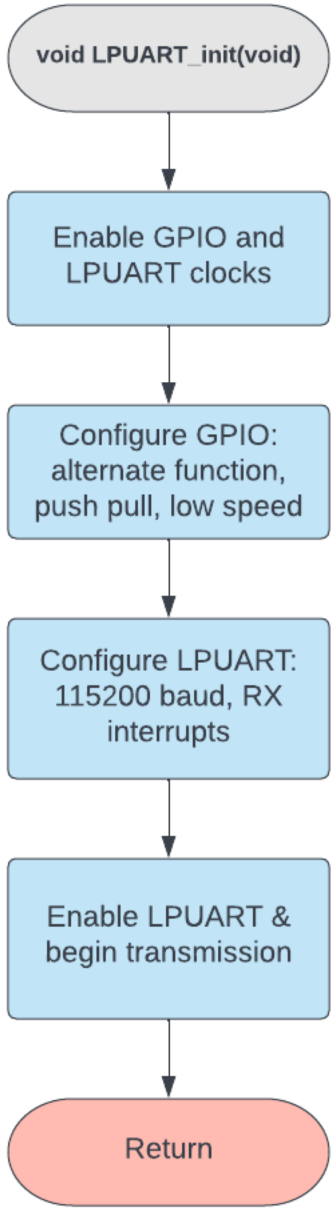


Figure 3. LPUART Initialization Flowchart

4.2.2. Comparator Initialization

The comparator used in this project follows a heavy initialization due to its implementation in conjunction with the STM timers. The STM board has a built-in comparator that takes in two signals and will output a 3.3V square wave. This square wave will follow the input signal's frequency but will only output 0V – 3.3V. The STM also has built in timers that work with this comparator. In our case, we will connect the comparator output to a timer that will capture each rising edge of the square wave. This functionality will be used to determine the frequency of our input signal.

The initialization begins by enabling the GPIO B clock and setting `GPIO_AFR` to the proper alternate function mode for GPIO PB0 to output the comparator square wave. Then PB1 and PB2 get configured as analog mode, where PB1 will be used as the V_{REF} (comparator reference signal) and PB2 will be used as V_{in} (input signal to comparator). The GPIO is configured to push-pull mode, low speed, and no pull-up or pull-down resistor.

Moving to the comparator initialization, PB2 is set as the + input to the comparator and PB1 is set as the – input. Hysteresis and filtering are then enabled to prevent noise on the comparator output, allowing for the timer to capture the rising edge more accurately.

Timer #2's CCR4 capture/compare register will be used as it can link directly to the comparator's output. CCR4 is configured as capture mode, rising edge triggered.

Interrupts will be used for this timer to capture frequency; however, this will be covered in a later section.

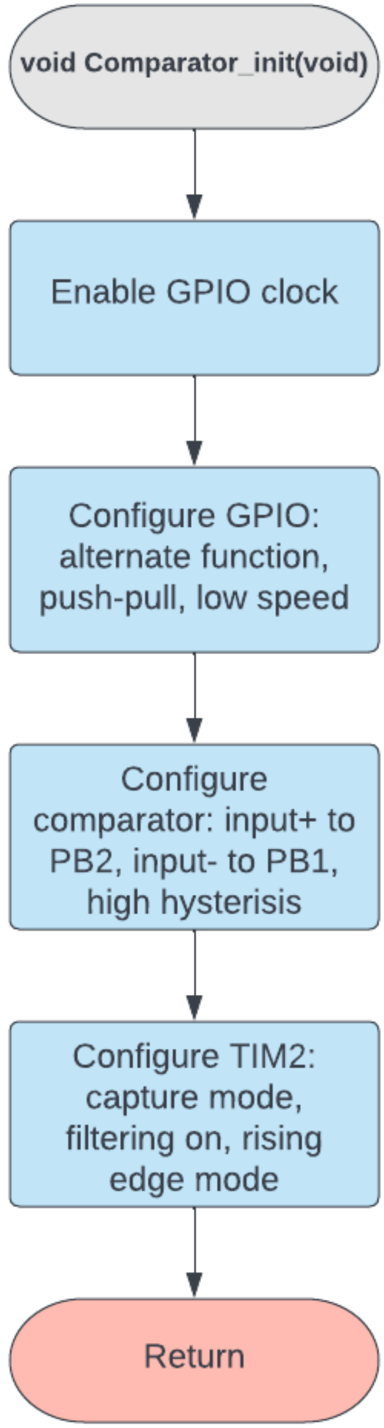


Figure 4. Comparator Initialization Flowchart

4.3. Timers & Interrupts

This system uses timers and interrupts extensively and these are the backbone of proper operation and voltage measurements.

4.3.1. Timing Overview

As mentioned previously, the system uses timers to trigger the interrupt handler depending on which mode is selected. To measure voltages, CCR1 is used. In DC mode, this value is set to 120,000 clock cycles which gives 200 measurements in ~1.6ms. In AC mode, this value is calculated based of the following formula:

```
CCR_Value = CCR_Period / (samples * 10);
```

This formula uses the CCR period and divides it by the # of samples we want – 200. It was determined through calibration that multiplying the samples by 10 gives a more accurate reading. This value is then inputted into CCR1 and is used to trigger the ADC to begin captures values and allows for adequate amount of voltage measurements over the period of a wave, especially very low frequency waves.

4.3.2. Interrupts Overview

The interrupts for this system provide the main functionality of the digital multimeter.

`TIM2_IRQHandler()`

This IRQ handler is used to measure voltages and to measure frequencies. If in voltage measurement mode (`stateVal = 1`), the system will disable CCR1 interrupts and then increment CCR1. This interrupt will only occur inside the `ADCConvert()` function, so after this interrupt occurs the ADC will begin converting values and then reenable interrupts once it is ready to read voltage values again, then the cycle repeats.

To measure frequency (`stateVal = 2`), the system will then trigger on CCR4 interrupts. This is due to CCR4 being linked directly to the rising edge capture of the comparator. When this is triggered, the system captures the first timing value and then saves it to a variable. It will then capture another value saving it to a separate variable. Once these two timing values have been captured, it will disable interrupts and then return to the AC mode FSM state where the system will proceed to calculate the frequency and so on.

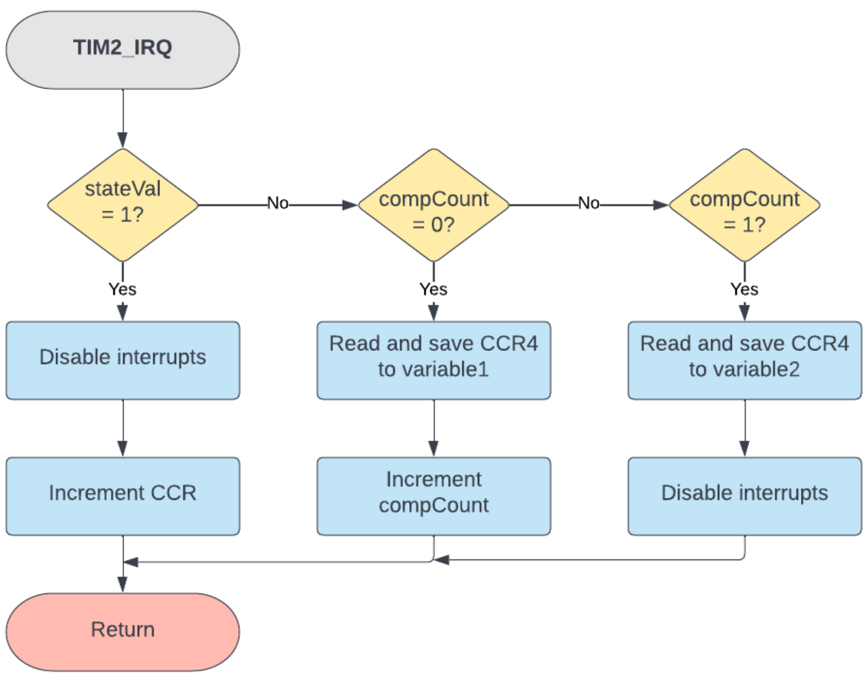


Figure 5. TIM2 IRQ Flowchart

4.4. FSM States

The FSM in this system has 3 states: ST_DC, ST_AC, and ST_SEL. Each state represents an important part of the digital multimeter.

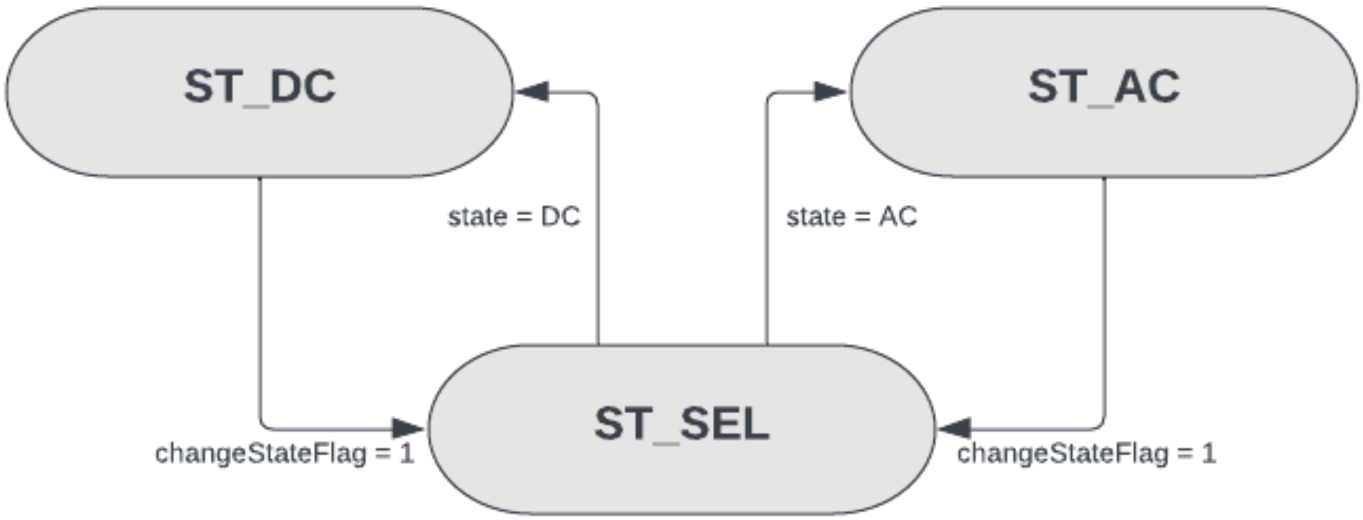


Figure 6. FSM State Diagram

4.4.1. ST_DC

This state is used to measure DC voltages. It begins by setting the CCR value to obtain 200 measurements in ~1.6ms (CCR1 = 120,000). It will then call ADCConvert() and save the voltages from ADCConvert. It will then print to the LPUART. An example output is shown below:

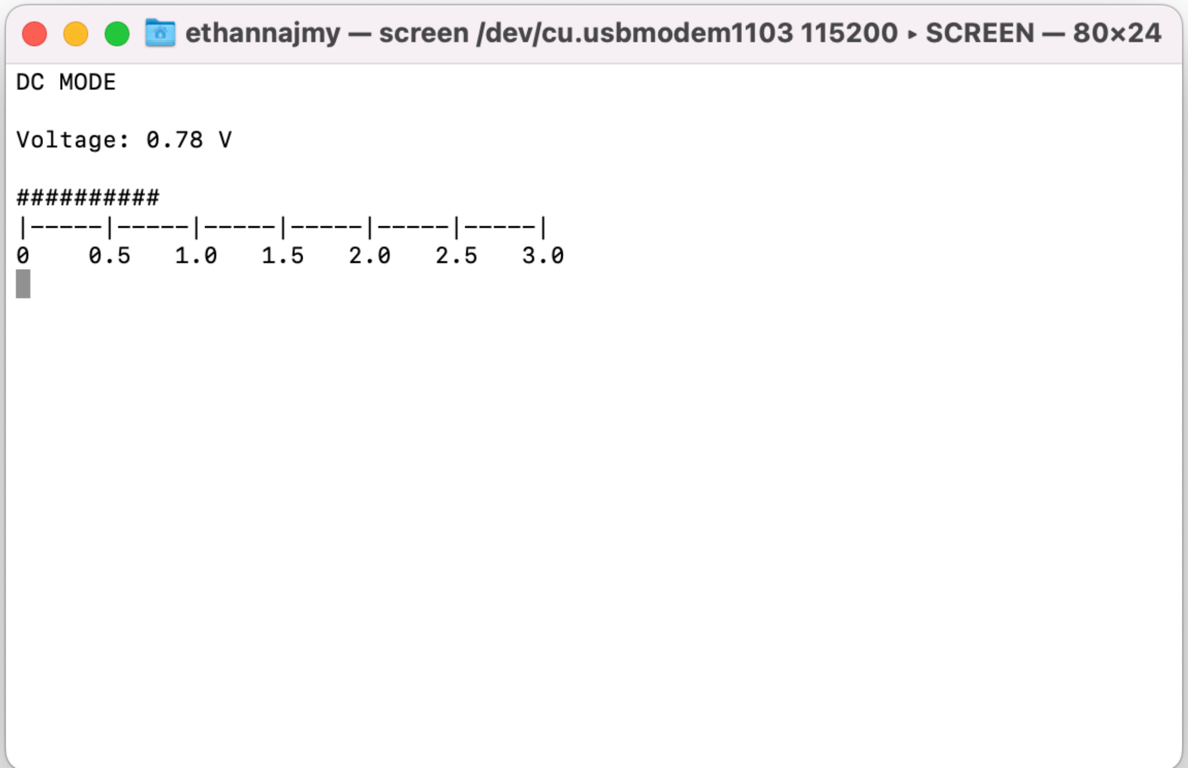


Figure 7. DC Mode Serial Terminal Output

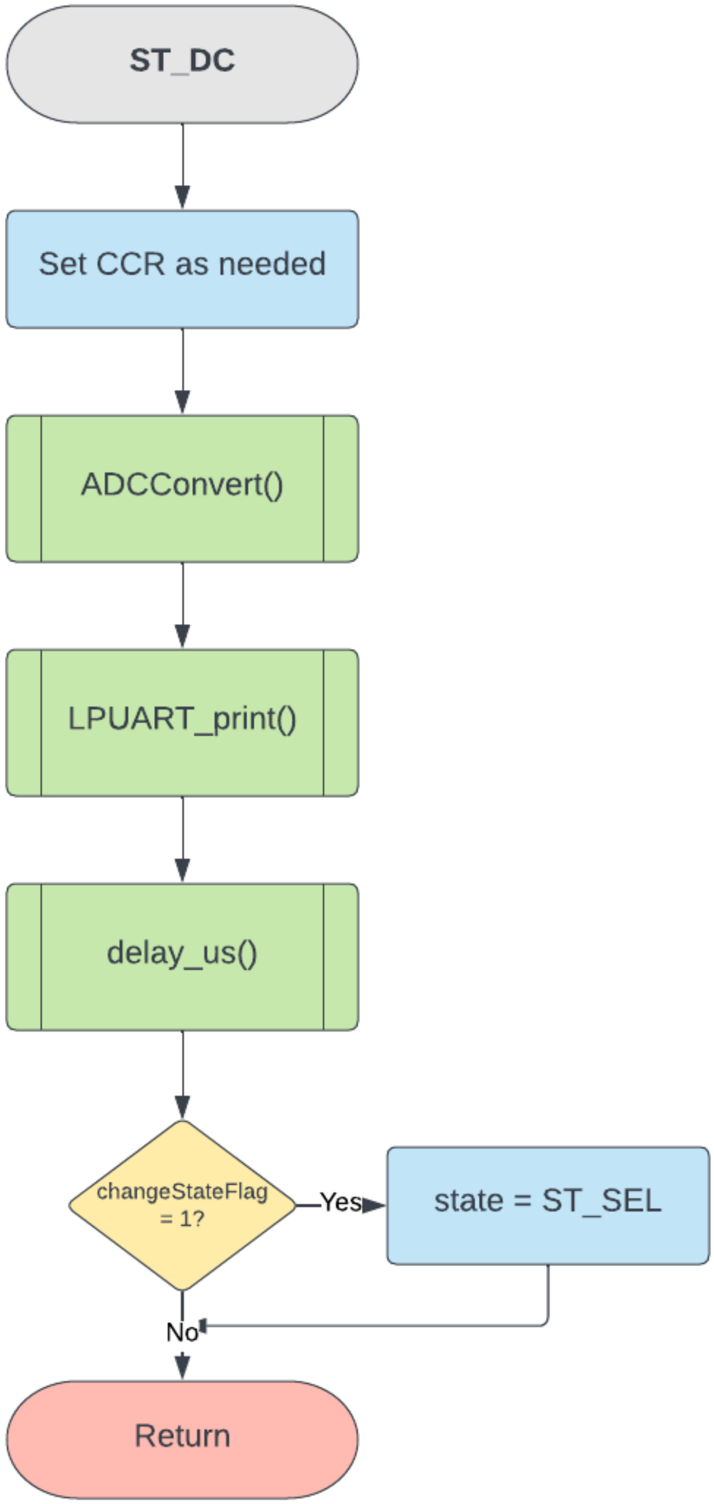


Figure 8. ST_DC Flowchart

4.4.2. ST_AC

In this state, the system will enable CCR4 interrupts, set `stateVal = 2` (for proper IRQ handling), and wait until two edges of the square wave comparator output have been captured. This is to ensure the period can be calculated using the CCR4 captured values. It will then disable CCR4 interrupts, calculate the frequency and transition into voltage measurements. It will set CCR1 to a value calculated using the following formula:

```
CCR_Value = CCR_Period / (samples * 10);
```

As mentioned previously, this value was determined in order to allow full sampling of a wave for a whole period of the wave. This allows the system to obtain very low and high frequency waves properly. It will then go about voltage conversion process the same way as in DC mode. The output of the AC mode on the terminal is shown below:

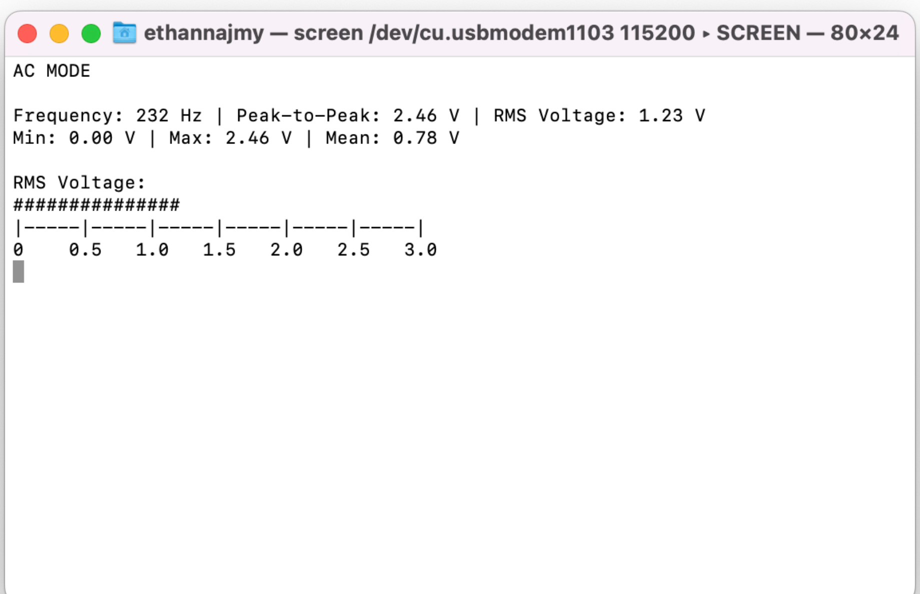


Figure 9. AC Mode Serial Terminal Output

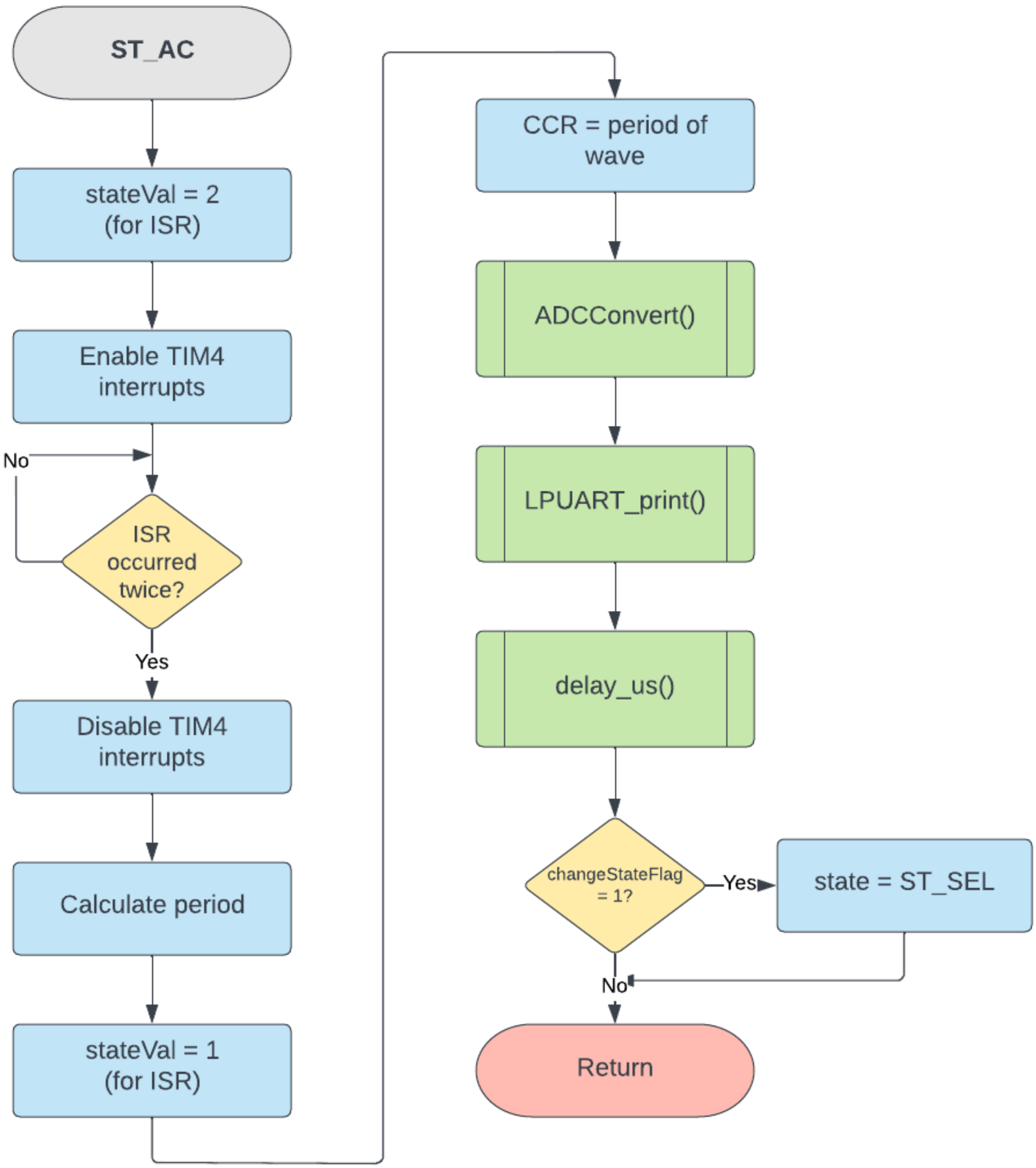


Figure 10. ST_AC Flowchart

4.4.3. ST_SEL

The FSM enters this state if the LPUART IRQ sets the changeStateFlag. If an "A" or "D" (AC or DC) is received from the LPUART, it will set the global flag and the FSM will check if this flag has been triggered. If so, the state gets set depending on the value received from the LPUART. It will also enable or disable the comparator to save power depending on the mode selected.

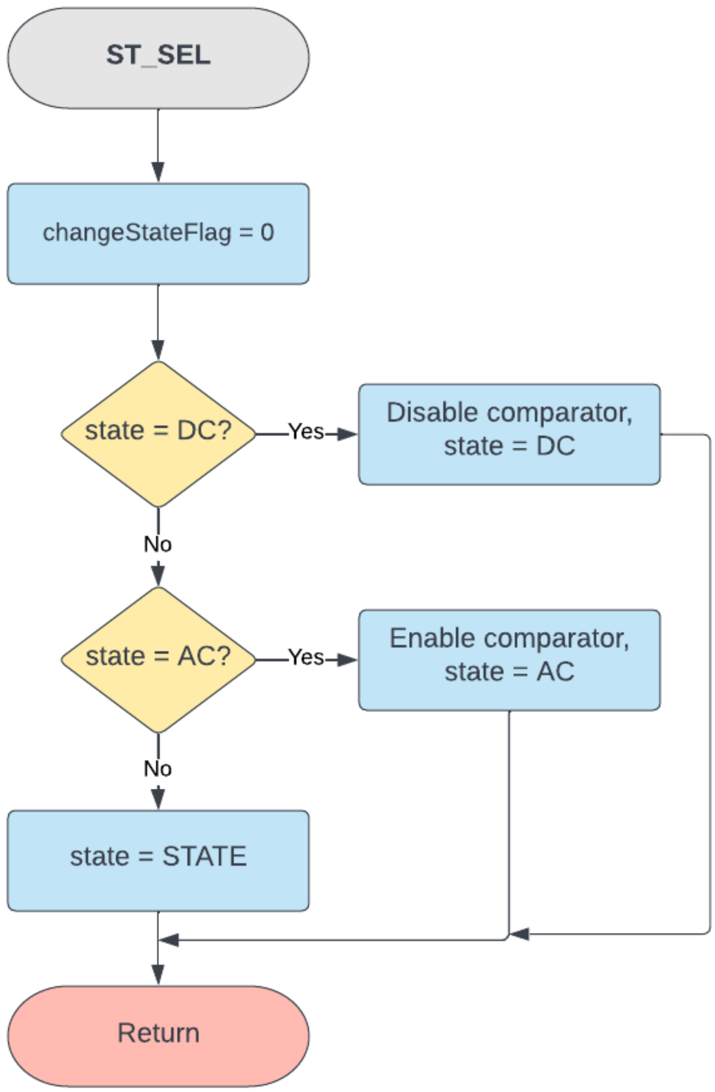


Figure 11. ST_SEL Flowchart

4.5. Subroutines and Functions

In each state, subroutines/functions are used to operate different aspects of the system. For example, reading the voltages requires the use of `ADCConvert()`. If you want to get a print to the terminal, you can use the subroutine `LPUART_print()`. In this section each function will be explained.

4.5.1. ADC Functions

`void ADCConvert(uint32_t *voltageValues, uint8_t state)`

This function works by triggering 200 conversions of the ADC dependent on the time set in CCR1. The function will wait until CCR1 triggers, begin a single conversion, and save it to the pointer `voltageValues`. It will do this until the required number of samples are obtained.

It will then disable interrupts, calculate the min, max, and mean voltage. If in AC mode, it will calculate RMS, if not it will skip this step to save time.

It will then convert the digital values obtained from the ADC to voltages through the following formulas. These formulas were obtained through calibration of the ADC following the technical note in the lab manual.

```
 min = ((min * 812) - 1314) / 10000;
 max = ((max * 812) - 1314) / 10000;
mean = ((mean * 812) - 1314) / 10000;
```

Lastly, it calculates the peak to peak value by taking the max – min and saves all these values: min, max, mean, peak-peak, and rms to the `voltageValues` array.

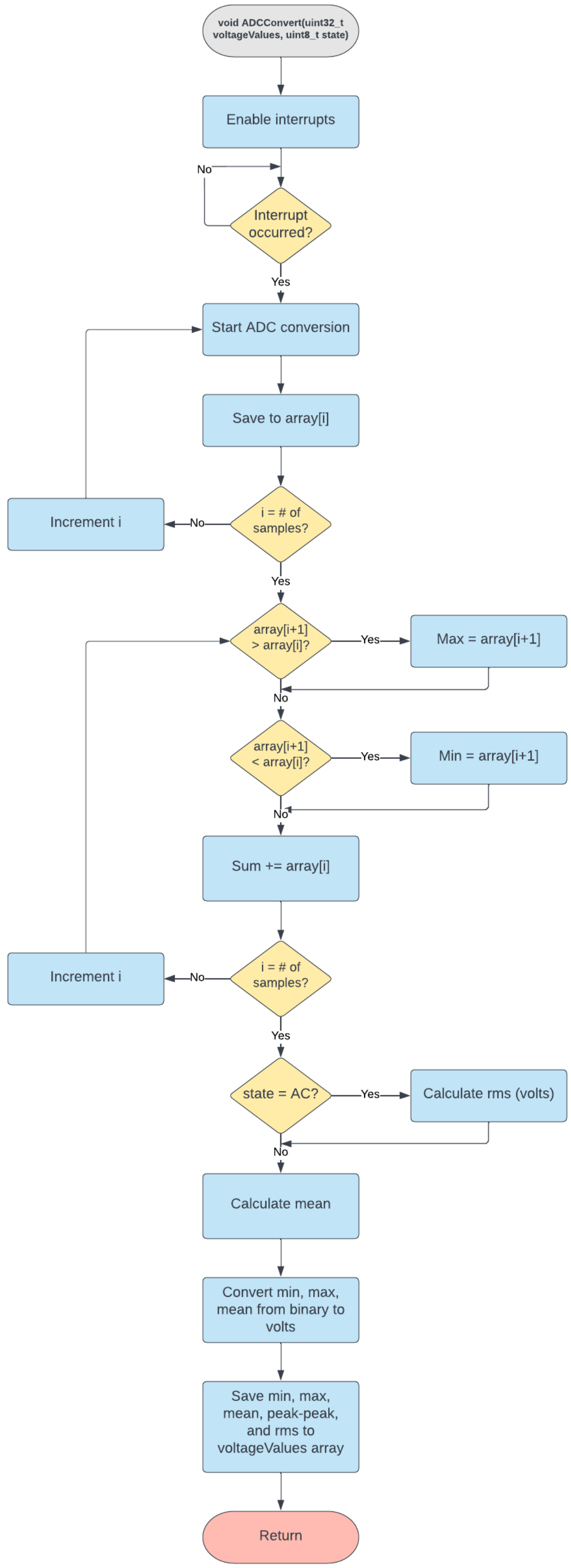


Figure 12. ADCConvert() Flowchart

`uint16_t ADCVoltageToString(char *string, uint32_t value)`

This function works by taking in a pointer that is in the form of the desired output. For example, when the system outputs voltages, it outputs it in the form “x.xx” therefore the pointer has value “0.00”.

The function will then increment through each part of the pointer and save a single digit from `value` to that spot. Each digit will have `0x30` added to it so it is converting to a `char` and able to be printed on the LPUART.

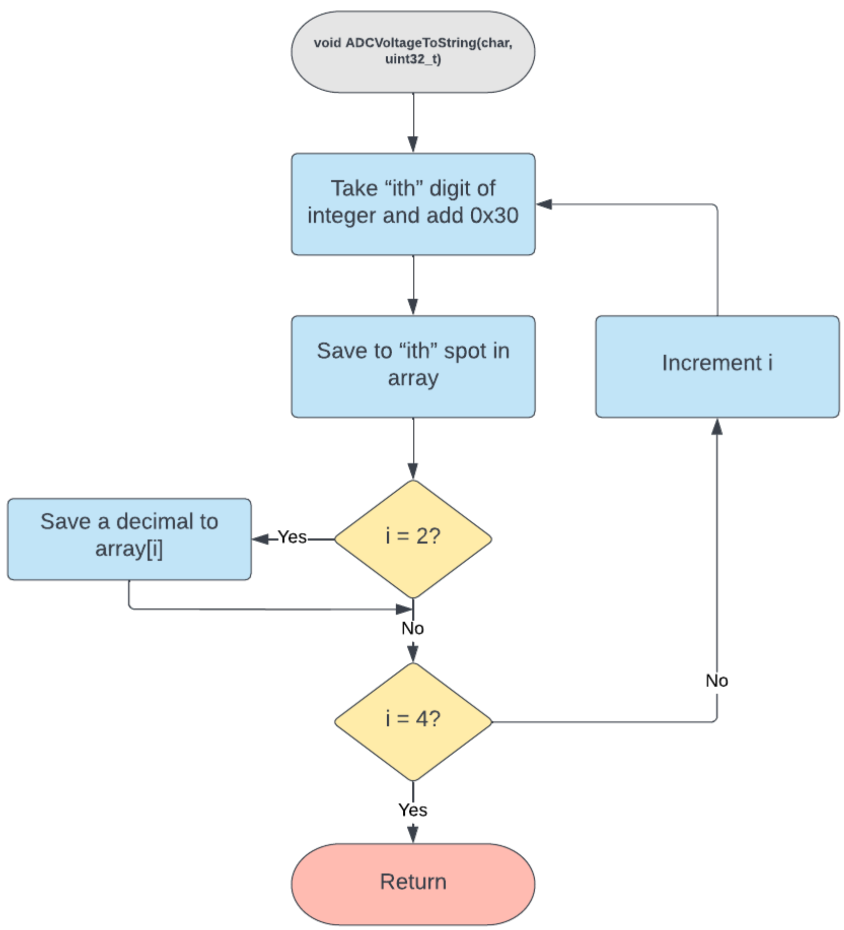


Figure 13. DAC_volt_conv() Flowchart

4.5.2. LPUART Functions

`void LPUART_print(const char *str)`

This function works by starting at the first character in the pointer `str` and checking if the LPUART is ready to accept data. If so, it will send this character to the LPUART and then increment the counter. The function will then increment through each character in the array until the string is terminated.

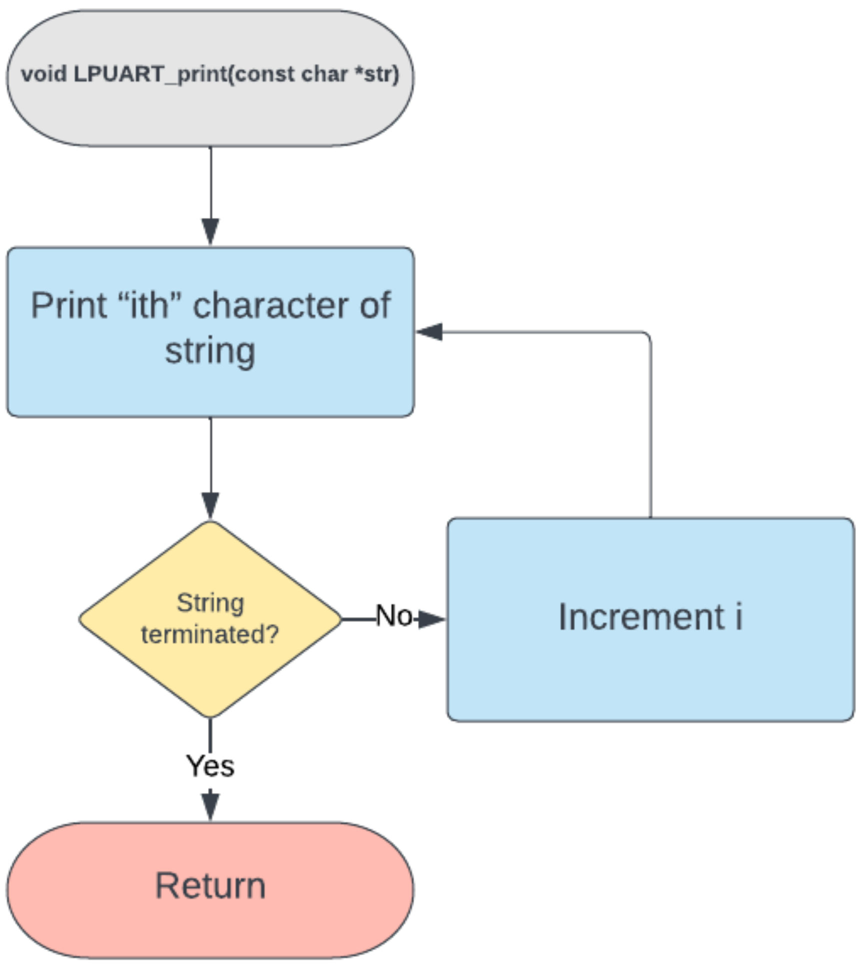


Figure 14. LPUART_print() Flowchart

`void LPUART_write_ESC(void)`

This function works by checking if the LPUART is ready to receive data, and if so it will output 0x1B to the LPUART. 0x1B signifies the beginning of an escape character and will allow for the LPUART to perform functions such as moving the cursor or changing text color.

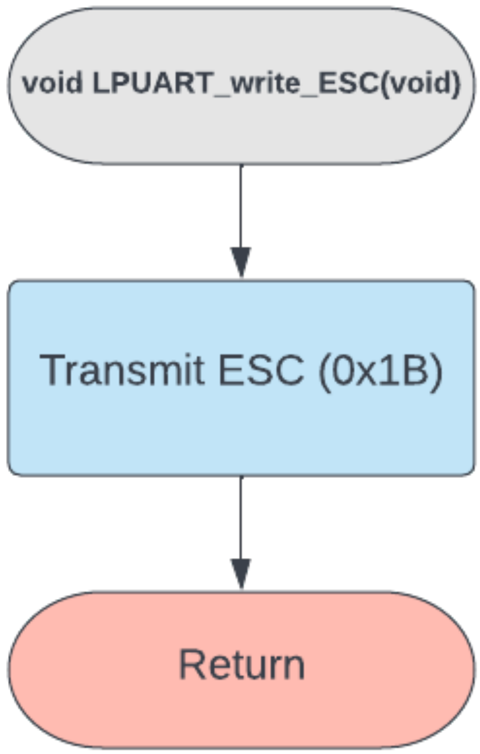


Figure 15. LPUART_write_ESC Flowchart

`void LPUART_ESC_Code(const char *str)`

This function simply combines `LPUART_write_ESC()` and `LPUART_print()`. It will call `LPUART_write_ESC()` then send `str` to `LPUART_print()` and print the escape code to the terminal. This function is used as it facilitates the best data transmission between the STM board and LPUART rather than calling each function individually.

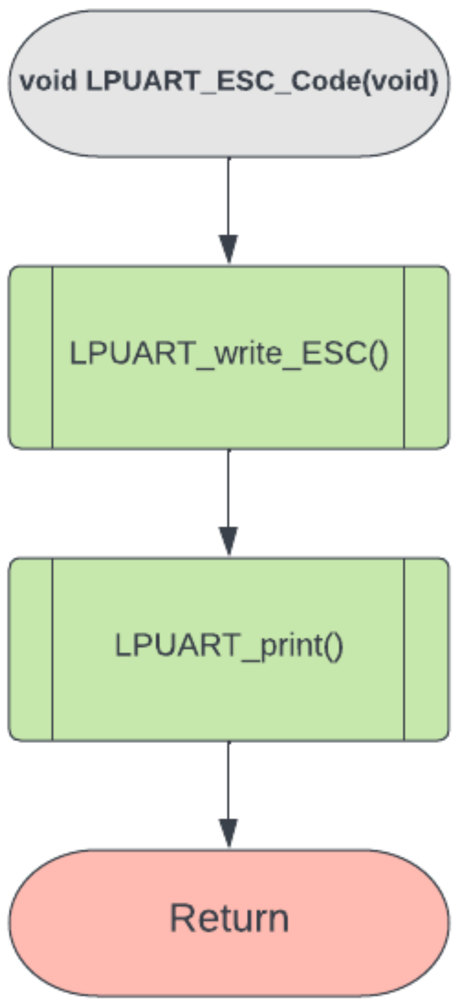


Figure 16. LPUART_ESC_Code() Flowchart

4.6.3. Comparator Functions

`void FreqToString(char *string, uint32_t value)`

This function works very similarly to `ADCVoltageToString()`. It takes in the frequency value and will output it in the same format as the pointer `string`. The format of `string` is “000” so the frequency will be displayed as so.

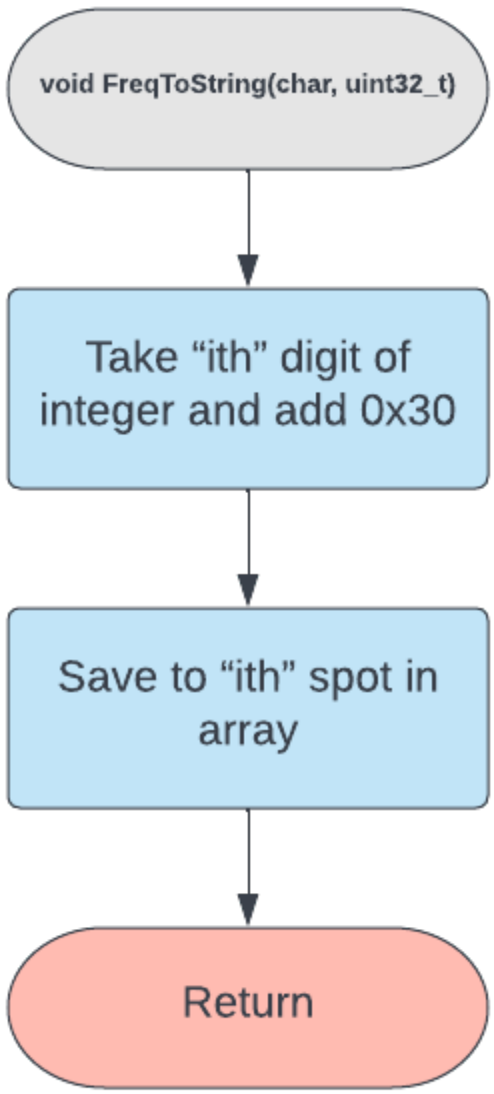


Figure 17. FreqToString () Flowchart

5. Appendices

Appendix A – References

[1] P. Hummel and J. Green, “STM32 Lab Manual,” *Google Docs*. [Online]. Available: https://docs.google.com/document/d/1NdE5188B2JWkEPdFAOdvxrEYo0R90Cg7wsG6oqHYOwk/edit#. [Accessed: 20-Apr-2022].

[2] “ST32L476 Data Sheet,” *ST.com*. [Online]. Available: https://www.st.com/resource/en/datasheet/stm32l476je.pdf. [Accessed: 20-Apr-2022].

[3] “ST32L476 Reference Manual,” *ST.com*. [Online]. Available: https://www.st.com/resource/en/reference_manual/rm0351-stm32l47xxx-stm32l48xxx-stm32l49xxx-and-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf. [Accessed: 20-Apr-2022].

[4] M. Tsoi, “Incomplete Guide to C (With A Focus on Embedded Systems Development),” 29-Mar-2021. .

[5] “MCP4901/4911/4921 Datasheet.” Microship Technology Inc., 2010.

Appendix B – Source Code

Source code is attached on the following pages:

```c
// Created by: Ethan Najmy
// Project 3 - Digital Multimeter
// CPE 329, Professor Hummel, Spring 2022
// June 6, 2022

#include "main.h"
#include "ADC.h"
#include "delay.h"
#include "LPUART.h"
#include "comparator.h"


#define CLK_SPEED 24000000
#define CCR_Std 120000

void SystemClock_Config(void);

// Global variable to keep track of states in ISR
uint8_t stateVal = 0;

// Global interrupt flag
uint8_t interruptFlag = 0;

// Global change state flag
uint8_t changeStateFlag = 0;

// Used to keep track of LPUART RX data
uint8_t stateChangeValue = 0;

// Used to save CCR rising edge value
uint32_t CCR_Capture = 0, CCR_Capture2 = 0;

// CCR1 initial value
uint32_t CCR_Value = 120000;

// Used to count comparator interrupts
uint8_t compCount = 0;

int main(void){

    /* -- Create FSM -- */
    typedef enum {ST_DC, ST_AC, ST_SEL} State_Type;
    State_Type state = ST_DC;


    /* -- Variables -- */
    uint32_t voltageValues[5];
    char tempVoltageString[] = "0.00";
    char tempFreqString[] = "000";
    uint32_t CCR_Period;
    uint32_t frequency = 0;
    uint8_t serialPrintCount = 0;


    /* -- Initialization -- */
    // Initialize System Functions
    HAL_Init();
    SystemClock_Config();

    // Initialize peripherals
    ADC_init();
    LPUART_init();

    // Turn on timer clock & init. comparator
    RCC->APB1ENR1 |= RCC_APB1ENR1_TIM2EN;
    Comparator_init();

    // Set CCR1 to 120,000 (0.005s)
```

```c
    // CCR1 will trigger when to begin converting values
    TIM2->CCR1 = CCR_Std;

    // Update interrupt event!!!!
    TIM2->EGR |= TIM_EGR_UG;

    // Enable TIM2 interrupts
    NVIC->ISER[0] |= (1 << (TIM2_IRQn & 0x1F));

    // Enable global interrupts
    __enable_irq();

    // Start timer
    TIM2->CR1 |= TIM_CR1_CEN;

    // Clear Screen
    LPUART_ESC_Code("[2J");

    while (1) {

        // FSM!
        switch(state){

        // Used to switch between waveforms
        case ST_DC:{

            // Check if should change stage
            if (changeStateFlag){
                state = ST_SEL;
                break;
            }

            // Set global state variable for IRQ
            stateVal = 1;

            CCR_Value = CCR_Std;
            // Set CCR1
            TIM2->CCR1 = CCR_Value;
            // Update interrupt event!!!!
            TIM2->EGR |= TIM_EGR_UG;

            // Start ADC Conversion to capture voltages
            // voltageValues[0] = min
            // voltageValues[1] = max
            // voltageValues[2] = mean (Offset)
            // voltageValues[3] = peak-peak
            ADCConvert(voltageValues, state);

            // Clear Screen
            LPUART_ESC_Code("[2J");
            // Return cursor home
            LPUART_ESC_Code("[H");
            // Print
            LPUART_print("DC MODE");
            // Move down 2 lines
            LPUART_ESC_Code("[2E");
            // Print max value
            LPUART_print("Voltage: ");
            ADCVoltageToString(tempVoltageString, voltageValues[2]);
            LPUART_print(tempVoltageString);
            LPUART_print(" V ");

            LPUART_ESC_Code("[2E");

            serialPrintCount = (voltageValues[2] * 12) / 100;

            for (int i = 0; i <= serialPrintCount; i++)
                LPUART_print("#");
```

```
            LPUART_ESC_Code("[1E");
            LPUART_print("|-----|-----|-----|-----|-----|-----|");
            LPUART_ESC_Code("[1E");
            LPUART_print("0    0.5   1.0   1.5   2.0   2.5   3.0");
            // Move down 1 line
            LPUART_ESC_Code("[1E");


            /* -- Used to delay outputting/measuring -- */
            // Set IRQ state
            stateVal = 0;
            // Set CCR1
            TIM2->CCR1 = 1200000;
            // Update interrupt event!!!!
            TIM2->EGR |= TIM_EGR_UG;
            // Enable CCR1 interrupts
            TIM2->DIER |= (TIM_DIER_CC1IE);
            // Wait until interrupt occurs
            while (!(TIM2->SR & TIM_SR_CC1IF));
            // Disable CCR1 interrupts
            TIM2->DIER &= ~(TIM_DIER_CC1IE);


            // Check if change state again (speeds up transition)
            if (changeStateFlag){
                state = ST_SEL;
                break;
            }

            break;
        }

        case ST_AC:{

            // Disable CCR1 interrupts
            TIM2->DIER &= ~(TIM_DIER_CC1IE);

            // Check if need to change state
            if (changeStateFlag){
                state = ST_SEL;
                break;
            }

            // Set ISR value for freq. measurements
            stateVal = 2;

            // Enable TIM2 CCR4 Interrupts
            TIM2->DIER |= (TIM_DIER_CC4IE);

            // Wait until 2 comparator cycles have occurred
            // Then we know frequency is ready to measure
            while (compCount != 2);

            // Disable Interrupts for CCR4
            TIM2->DIER &= ~(TIM_DIER_CC4IE);

            // Subtract CCR captures to determine period in cycles
            CCR_Period = (CCR_Capture2 - CCR_Capture) - 150;

            // Reset count
            compCount = 0;

            // Set stateVal = 1 for ISR
            stateVal = 1;

            CCR_Value = CCR_Period / (samples * 10);
            // Want 200 samples / period
```

```c
        TIM2->CCR1 = CCR_Value;
        // Update interrupt event!!!!
        TIM2->EGR |= TIM_EGR_UG;

        // Start ADC Conversion to capture voltages
        // voltageValues[0] = min
        // voltageValues[1] = max
        // voltageValues[2] = mean (Offset)
        // voltageValues[3] = peak-peak
        // voltageValues[4] = rms

        ADCConvert(voltageValues, state);

        // Clear Screen
        LPUART_ESC_Code("[2J");
        // Return cursor home
        LPUART_ESC_Code("[H");
        // Print
        LPUART_print("AC MODE");
        // Move down 2 lines
        LPUART_ESC_Code("[2E");
        // Print to the terminal
        LPUART_print("Frequency: ");
        // Used to convert frequency to a string
        FreqToString(tempFreqString, frequency);
        LPUART_print(tempFreqString);
        // Print "Hz"
        LPUART_print(" Hz ");


        // Convert peak-peak value to string
        ADCVoltageToString(tempVoltageString, voltageValues[3]);
        // Print peak-to-peak values
        LPUART_print("| Peak-to-Peak: ");
        LPUART_print(tempVoltageString);
        LPUART_print(" V ");

        // Convert rms value to string
        ADCVoltageToString(tempVoltageString, voltageValues[4]);
        // Print rms values
        LPUART_print("| RMS Voltage: ");
        LPUART_print(tempVoltageString);
        LPUART_print(" V ");

        LPUART_ESC_Code("[1E");
        LPUART_print("Min: ");
        ADCVoltageToString(tempVoltageString, voltageValues[0]);
        LPUART_print(tempVoltageString);
        LPUART_print(" V ");

        LPUART_print("| Max: ");
        ADCVoltageToString(tempVoltageString, voltageValues[1]);
        LPUART_print(tempVoltageString);
        LPUART_print(" V ");

        LPUART_print("| Mean: ");
        ADCVoltageToString(tempVoltageString, voltageValues[2]);
        LPUART_print(tempVoltageString);
        LPUART_print(" V ");

        LPUART_ESC_Code("[2E");
        LPUART_print("RMS Voltage:");
        LPUART_ESC_Code("[E");

        serialPrintCount = (voltageValues[4] * 12) / 100;

        for (int i = 0; i <= serialPrintCount; i++)
            LPUART_print("#");
```

```c

            LPUART_ESC_Code("[E");
            LPUART_print("|-----|-----|-----|-----|-----|-----|");
            LPUART_ESC_Code("[1E");
            LPUART_print("0    0.5   1.0   1.5   2.0   2.5   3.0");
            LPUART_ESC_Code("[1E");

            delay_us(100000);

            // Check if need to change state again (speeds up transition)
            if (changeStateFlag){
                state = ST_SEL;
                break;
            }
            break;
        }

        case ST_SEL:{
            // Clear change state flag
            changeStateFlag = 0;

            // Erase Whole Screen
            LPUART_ESC_Code("[2J");

            // Switch statement to check what was typed
            switch(stateChangeValue){

            // If A change to AC
            case 'A':{
                state = ST_AC;
                // Enable Comparator
                COMP1->CSR |= (COMP_CSR_EN);
                break;
                // If D change to DC
            } case 'D':{
                // Disable Comparator
                COMP1->CSR &= ~(COMP_CSR_EN);
                state = ST_DC;
                break;
                // If none of that do nothing
            } default:{
                break;
            }
            }
        }
        }
    }
}

/* -- Timer 2 Interrupt Handler -- */
void TIM2_IRQHandler(void){

    switch(stateVal){

    // If interrupted before variable set, do nothing
    case 0:{
        break;
    }

    // DC and AC Voltage Measurements
    case 1:{
        // Disable CCR1 interrupts
        TIM2->DIER &= ~(TIM_DIER_CC1IE);

        // Increment CCR1
        TIM2->CCR1 += CCR_Value;

        break;
```

```c
    }

    // Frequency Measurements
    case 2:{
        // Stuff for the comparator
        if (compCount == 0){
            // Read CCR4 and save
            CCR_Capture = TIM2->CCR4;
            compCount++;
        } else if (compCount == 1){
            // Read CCR4 and save
            CCR_Capture2 = TIM2->CCR4;
            compCount++;
            // Disable interrupts
            TIM2->DIER &= ~(TIM_DIER_CC4IE);
        }
        break;
    }
    }
}

void LPUART1_IRQHandler(void){
    // Read the input
    stateChangeValue = LPUART1->RDR;

    // If interrupt occurs, state change may be needed
    changeStateFlag = 1;
}



/* ------------------------------------------------ */
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /** Configure the main internal regulator output voltage
     */
    if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
    {
        Error_Handler();
    }

    /** Initializes the RCC Oscillators according to the specified parameters
     * in the RCC_OscInitTypeDef structure.
     */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_MSI;
    RCC_OscInitStruct.MSIState = RCC_MSI_ON;
    RCC_OscInitStruct.MSICalibrationValue = 0;
    RCC_OscInitStruct.MSIClockRange = RCC_MSIRANGE_9;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
        Error_Handler();
    }

    /** Initializes the CPU, AHB and APB buses clocks
     */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
            |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_MSI;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_1) != HAL_OK)
    {
```

```
        Error_Handler();
    }
}
void Error_Handler(void)
{
    /* USER CODE BEGIN Error_Handler_Debug */
    /* User can add his own implementation to report the HAL error return state */
    __disable_irq();
    while (1)
    {
    }
    /* USER CODE END Error_Handler_Debug */
}
#ifdef  USE_FULL_ASSERT
void assert_failed(uint8_t *file, uint32_t line)
{
    /* USER CODE BEGIN 6 */
    /* User can add his own implementation to report the file name and line number,
     ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
    /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */

```

```
#ifndef SRC_ADC_H_
#define SRC_ADC_H_

// Defines
#define samples 20000

// Function Declarations
void ADC_init(void);
void ADCConvert(uint32_t *, uint8_t);
void ADCVoltageToString(char *, uint32_t);


#endif /* SRC_ADC_H_ */

```

```c
#include "main.h"
#include "ADC.h"
#include "delay.h"
#include <math.h>

/* -- Global Variables for ADC Conv. Process -- */
static uint8_t ADC_EOC_Flag = 0;
static uint16_t ADCData = 0;

/* -- ADC Initialization Function -- */
void ADC_init(void){

    // Turn on ADC Clocks using HCLK (AHB)
        RCC->AHB2ENR |= RCC_AHB2ENR_ADCEN;
        ADC123_COMMON->CCR = (1 << ADC_CCR_CKMODE_Pos);

        // Power up the ADC and voltage regulator
        ADC1->CR &= ~(ADC_CR_DEEPPWD);
        ADC1->CR |= (ADC_CR_ADVREGEN);

        // Wait 20us for the voltage regulator to startup
        delay_us(20);

        // Configure single ended mode for channel 5
        ADC1->DIFSEL &= ~(ADC_DIFSEL_DIFSEL_5);

        // Calibrate ADC - ensure ADEN is 0 and single ended mode
        ADC1->CR &= ~(ADC_CR_ADEN | ADC_CR_ADCALDIF);
        ADC1->CR |= ADC_CR_ADCAL;

        // Wait for ADCAL to be 0
        while (ADC1->CR & ADC_CR_ADCAL);

        // Enable ADC
        // Clear ADRDY bit by writing 1
        ADC1->ISR |= (ADC_ISR_ADRDY);
        ADC1->CR |= ADC_CR_ADEN;

        // Wait for ADRDY to be 1
        while(!(ADC1->ISR & ADC_ISR_ADRDY));

        // Clear the ADRDY bit by writing 1
        ADC1->ISR |= ADC_ISR_ADRDY;

        // Configure SQR for regular sequence
        ADC1->SQR1 = (5 << ADC_SQR1_SQ1_Pos);

        // Configure channel 5 for sampling time (SMP) with 2.5 clocks
        ADC1->SMPR1 = (0b000 << ADC_SMPR1_SMP5_Pos);
        // # determines sampling rate in register reference page
        // SMP5 is GPIO A0 which corresponds with ADCINN5

        // Configure conversion resolution (RES) 12-bit, right align
        // Single Conversion, software trigger
        ADC1->CFGR = 0;

        /* ---- Enable interrupts ---- */
        // End of conversion interrupt enable
        ADC1->IER |= ADC_IER_EOCIE;

        // Clear the flag
        ADC1->ISR |= (ADC_ISR_EOC);

        // Enable interrupts in NVIC
        NVIC->ISER[0] |= (1 << (ADC1_2_IRQn & 0x1F));

        // Enable interrupts globally
        // ** DISABLED HERE DUE TO ENABLE IN MAIN **
```

```c
        //__enable_irq();

        /* ---- GPIO Configure ---- */
        // Configure GPIO for PA0 pin for analog mode
        GPIOA->MODER &= ~(GPIO_MODER_MODE0);
        GPIOA->MODER |= (GPIO_MODER_MODE0);

        // Low Speed
        GPIOA->PUPDR &= ~(GPIO_PUPDR_PUPD0);
}

void ADCConvert(uint32_t *voltageValues, uint8_t state){
    uint32_t min = 0xFFFFFFFF, max = 0,
                sum = 0, mean = 0,
                peakpeak = 0;
    double rms = 0;
    uint32_t voltageArray[samples];


    for(int i = 0; i < (samples); i++)
    {
        // Enable CCR1 interrupts
        TIM2->DIER |= (TIM_DIER_CC1IE);

        // Wait until interrupt occurs
        while (!(TIM2->SR & TIM_SR_CC1IF));

        //Start a conversion
        ADC1->CR |= ADC_CR_ADSTART;

        //Wait for the flag set by a conversion finishing
        while(!(ADC_EOC_Flag));

        //Store the converted value in the array
        voltageArray[i] = ADCData;

        //Clear the conversion finished flag
        ADC_EOC_Flag = 0;
    }

    // Disable CCR1 interrupts
    TIM2->DIER &= ~(TIM_DIER_CC1IE);

    // Find the min, max and mean of the samples
    for(int i = 0; i < samples; i++){
        min = voltageArray[i] < min ? voltageArray[i] : min;
        max = voltageArray[i] > max ? voltageArray[i] : max;
        sum += voltageArray[i];
    }

    // If state = 1 (ST_AC), calc RMS
    if (state == 1){
    // Calculate the RMS value
    for(int i = 0; i < samples; i++){

        // Square the element in the array
        voltageArray[i] = pow(voltageArray[i],2);

        // Add squared element to rms
        rms += voltageArray[i];
    }
    // Divide squared elements by # of array val
    rms /= samples;
    // Take square root
    rms = sqrt(rms);
    rms = ((rms * 812) - 1314) / 10000;
    voltageValues[4] = rms;
    }
```

```c

    // Calculate the mean
    mean = (sum / samples);

    // Convert the values to voltages
    // Turn the 12 bit binary number into a voltage in uV
    // Use calibration equation to get voltage value
    // Then divide by 10000 to get value in 10s of mV
    min = ((min * 812) - 1314) / 10000;
    max = ((max * 812) - 1314) / 10000;
    mean = ((mean * 812) - 1314) / 10000;

    // Prevents min & mean from going very high if negative
    // voltage measured due to min being unsigned.
    if (min > 4000)
        min = 0;
    if (mean > 4000)
        mean = 0;

    // Calculate peak-to-peak from max and min
    peakpeak = max - min;

    // Save min, max, and mean values to array
    voltageValues[0] = min;
    voltageValues[1] = max;
    voltageValues[2] = mean;
    voltageValues[3] = peakpeak;

}

/* -- Function to convert the ADC voltage into a string -- */
// This function was implemented by A7 partner Colt Whitley.
void ADCVoltageToString(char *string, uint32_t value)
{
    // In this application string is always a character array with 4 digits
    // The second element we want to always set to '.'
    for(int i = 0; i < 4; i++)
    {
        // Create and store a character for the ith digit of the number in
        if(i != 2) {
            string[(3) - i] = 0x30 + (value % 10);
            // Shift value over 1 decade
            value /= 10;
        }
        // If we are on the 2nd i we want to add a decimal place
        else string[(3) - i] = '.';
    }
}

void ADC1_2_IRQHandler(void){
    if(ADC1->ISR & ADC_ISR_EOC)
    {
        //Read data into the static variable
        ADCData = ADC1->DR;
        //Set the static flag
        ADC_EOC_Flag = 1;
    }
}

```

```
/* ---- comparator.h ---- */
#ifndef SRC_COMPARATOR_H_
#define SRC_COMPARATOR_H_

// Defines
#define COMPAR1 GPIOB

// Function declarations / prototypes
void Comparator_init(void);
void FreqToString(char *string, uint32_t value);


#endif /* SRC_COMPARATOR_H_ */

```

```c
/* ---- comparator.c ---- */
#include "main.h"
#include "comparator.h"
#include <string.h>

void Comparator_init(void){
    /* -- GPIOB Enable -- */
        // Turn on GPIOB Clock
        RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN;

        // Set AFR to AF12 for use of PB0 as comparator out
        COMPAR1->AFR[0] &= ~(GPIO_AFRL_AFSEL0);
        COMPAR1->AFR[0] |= (0b1100 << GPIO_AFRL_AFSEL0_Pos);

        // Set PB0 to AF Mode for comparator output
        COMPAR1->MODER &= ~(GPIO_MODER_MODE0);
        COMPAR1->MODER |= (0b10 << GPIO_MODER_MODE0_Pos);

        // Set PB1 and PB2 to Analog Mode
        COMPAR1->MODER &= ~(GPIO_MODER_MODE1
                | GPIO_MODER_MODE2);
        COMPAR1->MODER |= ((0b11 << GPIO_MODER_MODE1_Pos)
                | (0b11 << GPIO_MODER_MODE2_Pos));

        // Push-Pull
        COMPAR1->OTYPER &= ~(GPIO_OTYPER_OT0
                | GPIO_OTYPER_OT1
                | GPIO_OTYPER_OT2);

        // Low Speeds
        COMPAR1->OSPEEDR &= ~(GPIO_OSPEEDR_OSPEED0
            | GPIO_OSPEEDR_OSPEED1
            | GPIO_OSPEEDR_OSPEED2);

        // No PUPD
        COMPAR1->PUPDR &= ~(GPIO_PUPDR_PUPD0
                | GPIO_PUPDR_PUPD1
                | GPIO_PUPDR_PUPD2);

        /* -- Comparator Enable -- */
        // Configure Input+ of comparator for PB2
        COMP1->CSR &= ~(COMP_CSR_INPSEL);
        COMP1->CSR |= (0b1 << COMP_CSR_INPSEL_Pos);

        // Configure Input- of comparator for PB1
        COMP1->CSR &= ~(COMP_CSR_INMSEL);
        COMP1->CSR |= (0b110 << COMP_CSR_INMSEL_Pos);

        // Enable Hysteresis for Noise Reduction
        COMP1->CSR |= (0b11 << COMP_CSR_HYST_Pos);

        /* -- Timer 2 CCR4 Enable -- */
        // Connect CCR4 to COMP1_OUT
        TIM2->OR1 |= (0b1 << TIM2_OR1_TI4_RMP_Pos);

        // Set Capture/Compare Selection
        TIM2->CCMR2 |= (0b1 << TIM_CCMR2_CC4S_Pos);

        // Set Capture Buffer to prevent double counting period
        TIM2->CCMR2 |= (0b11 << TIM_CCMR2_IC4F_Pos);

        // Set rising edge
        TIM2->CCER &= ~(TIM_CCER_CC4NP);
        TIM2->CCER &= ~(TIM_CCER_CC4P);

        // Enable capture mode
        TIM2->CCER &= ~(TIM_CCER_CC4E);
        TIM2->CCER |= (0b1 << TIM_CCER_CC4E_Pos);
```

```c

        // Enable Capture/Compare Generation
        TIM2->EGR &= ~(TIM_EGR_CC4G);
        TIM2->EGR |= (0b1 << TIM_EGR_CC4G_Pos);
}


// Used to convert digits to strings for UART
void FreqToString(char *string, uint32_t value){

    // for the length of the string
    for(int i = 0; i < strlen(string); i++){

        // add hex 30 to the value and save in last array spot
        string[((strlen(string) - 1) - i)] = 0x30 + (value % 10);

        // divide value by 10 to add to next array slot
        value /= 10;
    }
}

```

```c
/* ---- LPUART.h ---- */
#ifndef SRC_LPUART_H_
#define SRC_LPUART_H_

// Include #defines
#define LPUART GPIOG
#define BAUD_RATE_HEX 53333
#define ESC 0x1B


// Include function definitions / prototypes
void LPUART_init(void);
void LPUART_write_ESC(void);
void LPUART_print(const char *str);
void LPUART_ESC_Code(const char *str);

#endif /* SRC_LPUART_H_ */

```

```c
/* ----------- LPUART.c ------------ */
#include "main.h"
#include "LPUART.h"
#include "delay.h"

// Function to initialize the LPUART
void LPUART_init(void){

    /* -- GPIOG Enable -- */
    // Turn on GPIOG Clock
    RCC->AHB2ENR |= RCC_AHB2ENR_GPIOGEN;
    RCC->APB1ENR2 |= RCC_APB1ENR2_LPUART1EN;

    // Power on GPIOG/LPUART
    PWR->CR2 |= PWR_CR2_IOSV;

    // Set AF Mode Registers to AF8
    LPUART->AFR[0] &= ~(GPIO_AFRL_AFSEL7);
    LPUART->AFR[0] |= (GPIO_AFRL_AFSEL7_3);

    LPUART->AFR[1] &= ~(GPIO_AFRH_AFSEL8);
    LPUART->AFR[1] |= (GPIO_AFRH_AFSEL8_3);

    // Set PG7 and PG8 to Push Pull
    LPUART->OTYPER &= ~(GPIO_OTYPER_OT7
            | GPIO_OTYPER_OT8);

    // Set PG7 and PG8 to Low Speed
    LPUART->OTYPER &= ~(GPIO_OSPEEDR_OSPEED7
            | GPIO_OSPEEDR_OSPEED8);

    // Set PG7 and PG8 to No PUPD
    LPUART->PUPDR &= ~(GPIO_PUPDR_PUPD7
            | GPIO_PUPDR_PUPD8);

    // Set PG7 (TX) and PG8 (RX) to AF Mode
    LPUART->MODER &= ~(GPIO_MODER_MODE7
            | GPIO_MODER_MODE8);
    LPUART->MODER |= ((2 << GPIO_MODER_MODE7_Pos)
            | (2 << GPIO_MODER_MODE8_Pos));

    /* -- LPUART Enable -- */
    // 8 bit mode, No Parity, 1 Stop Bit
    // All set by default, no configuration needed

    // Baud Rate = 115200
    LPUART1->BRR = BAUD_RATE_HEX;

    // Interrupts Enable
    LPUART1->CR1 |= USART_CR1_RXNEIE;

    NVIC->ISER[2] |= (1<<(LPUART1_IRQn & 0x1F));

    // ** DISABLED HERE DUE TO ENABLE IN MAIN **
    //__enable_irq();


    // Enable LPUART
    LPUART1->CR1 |= (USART_CR1_UE);

    /* --- Begin Data Transmission --- */
    LPUART1->CR1 |= (USART_CR1_TE | USART_CR1_RE);
}

void LPUART_print(const char *str){
    // Initialize counting variable
    uint8_t i = 0;

```

```
    // While the input str is not terminated, print.
    while (str[i] != '\0'){
        while(!(LPUART1->ISR & USART_ISR_TXE));

        LPUART1->TDR = (str[i++]);
    }
}

void LPUART_write_ESC(void){
    // If the TXE flag is high, meaning transmission over,
    // write to the transmit data register
    while (!(LPUART1->ISR & USART_ISR_TXE)){
    }
    LPUART1->TDR = ESC;
}

void LPUART_ESC_Code(const char *str){
    LPUART_write_ESC();
    LPUART_print(str);
}
```