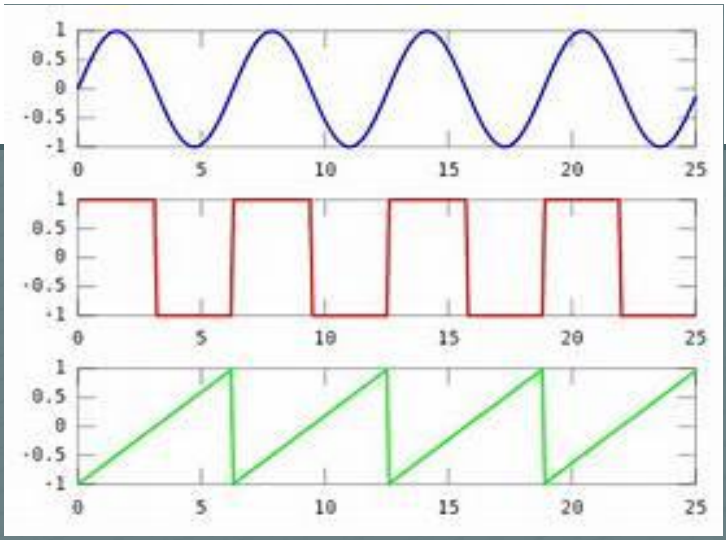
Digital Multimeter

Shawheen Ghezavat

Spring, May 22, 2024

Behavior Description

The Digital Multimeter (DMM) designed in this project measures voltage and frequency using the STM32 NUCLEO-L476RG microcontroller. It provides accurate DC and AC voltage readings and frequency measurements for various waveform types and features a terminal-based interface for real-time data. The system ensures continuous data processing and real-time updates.

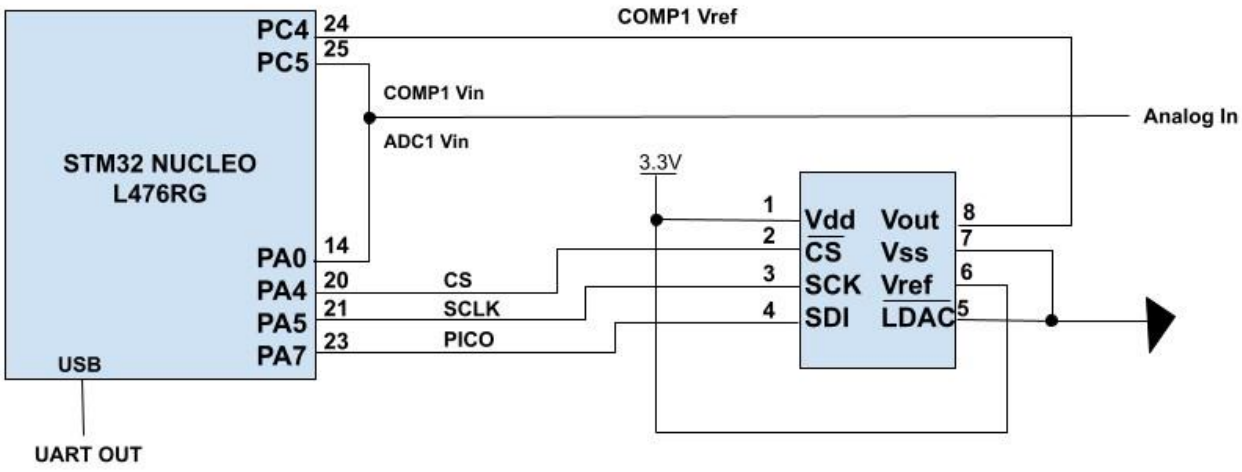


System Specifications

Specification	Details
Power Supply Voltage	3.3V DC
ADC Bit Resolution	12 bits
Battery Life	N/A (powered via USB)
DAC Bit Resolution	12 bits (MCP4921)
Display	Terminal
MCU Clock Speed	24 MHz
Physical Dimensions	75 mm x 54 mm x 15 mm
Weight	Approx. 50 grams
Environmental Tolerance	Operating temperature 0°C to 70°C
Baud Rate	115200
Input Waveform Types	Sine, Square, Sawtooth, and others
Frequency Range	1 Hz-40k+Hz
Max Output Voltage Peak-to-Peak	3.3 V
DC Voltage Reading Range	0-3.3V
AC Voltage Reading Range	0-3V

Table 1

System Schematic



Calculations

To determine the Capture/Compare Register (CCR) value, we start with the desired sample rate and the internal clock frequency:

$$\text{Desired Sample Rate} = 10\,\text{kHz} = 10,000\,\text{Hz}$$

$$\text{MCU Clock} = 24,000,000\,\text{Hz}$$

The CCR value can be calculated using the formula:

$$\text{CCR} = \frac{\text{MCU Clock}}{\text{Desired Sample Rate}}$$

Substituting the given values:

$$\text{CCR} = \frac{24,000,000\,\text{Hz}}{10,000\,\text{Hz}}$$

Simplifying the division:

$$\text{CCR} = 2400$$

Therefore, to achieve a 10 kHz sample rate with a 24 MHz internal clock, the Capture/Compare Register (CCR) value should be:

$$\text{CCR} = 2400$$

Software Architecture

How It Works

The frequency is measured using the TIM2 timer's input capture feature. When an input capture event occurs on channel 4, the TIM2_IRQHandler function is triggered, and this function captures the current timer count (curr) and calculates the time difference (curr - prev) between consecutive captures. Using the clock frequency (CLK_FREQ), the frequency is then computed as freq = CLK_FREQ / (curr – prev) + 1. The calculated frequency is stored in the freq variable and is displayed on the terminal.

Voltage measurements are performed using the ADC (Analog-to-Digital Converter). The ADC1_2_IRQHandler function is triggered upon the end of a conversion (EOC) event. The converted ADC data is read and stored in the adc_data variable, and the adc_flag is set to indicate that new ADC data is available. In the main loop, if adc_flag is set, the adc_data is processed: it updates the maximum (adc_max) and minimum (adc_min) recorded ADC values. The adc_data is also stored in the voltVals array for further calculations.

When the number of samples reaches NUM_SAMPLES, the true root mean square (True RMS) and peak-to-peak voltages are calculated. The calculate_rms function computes the RMS value by summing the squares of the voltage samples, taking the average, and then applying the square root. The calculate_peak_to_peak function determines the peak-to-peak voltage by subtracting the minimum ADC value from the maximum. These calculated values are displayed on the terminal through the update_terminal function. Additionally, the average of the maximum and minimum ADC values is calculated and written to the DAC (Digital-to-Analog Converter) to set a reference voltage for the comparator.

The system continuously converts new ADC samples and updates the frequency and voltage readings, ensuring accurate real-time measurements.

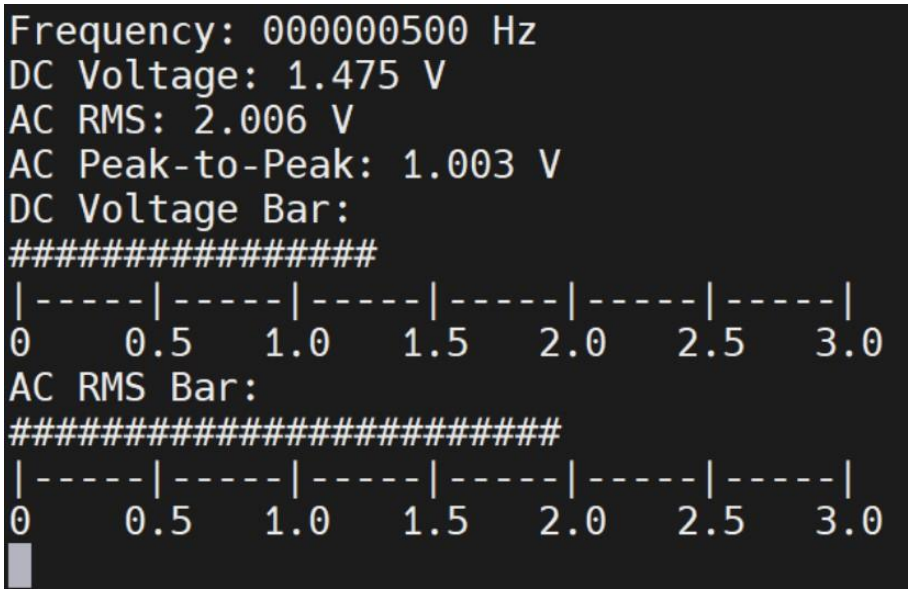
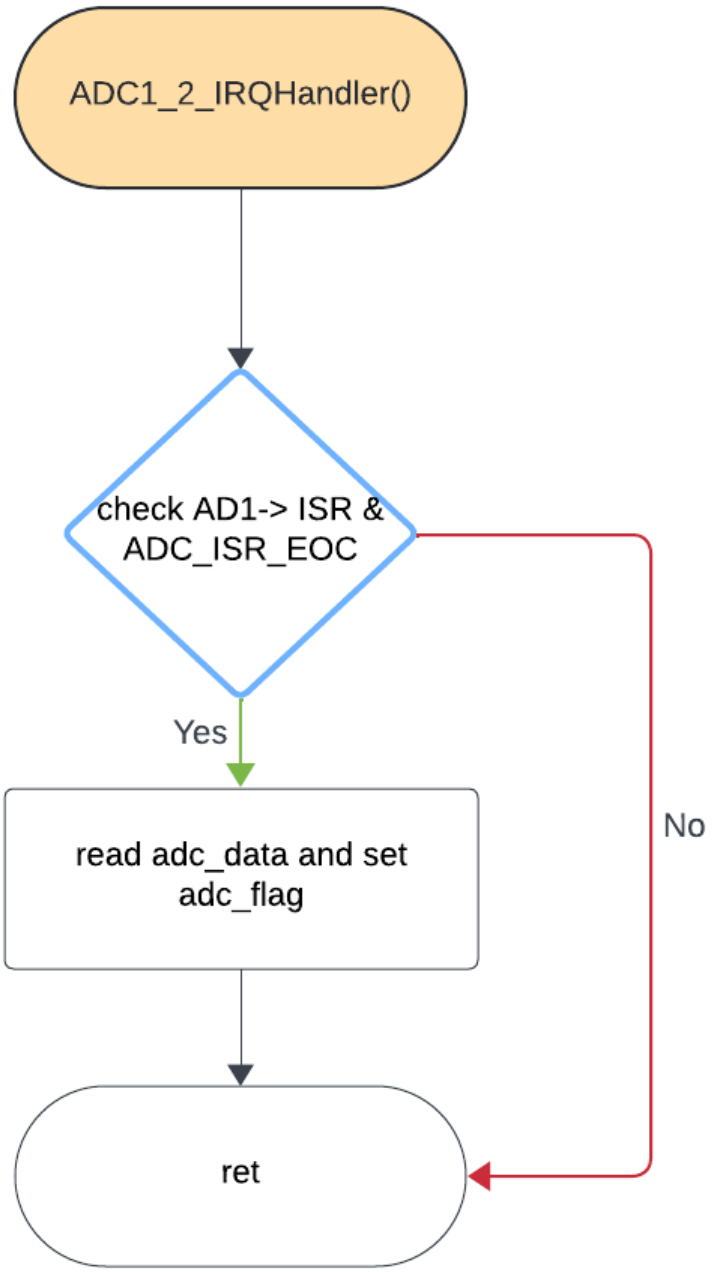
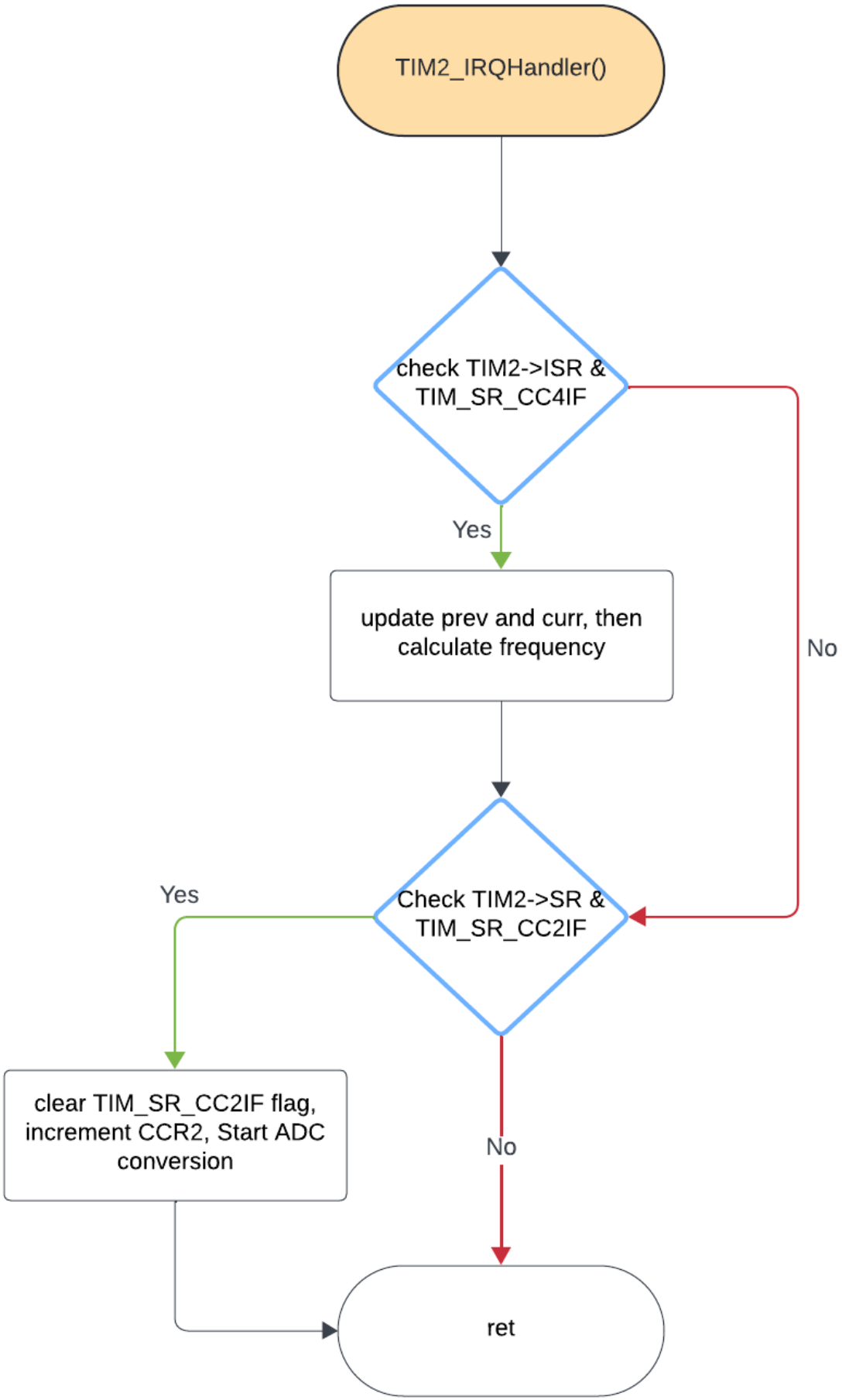
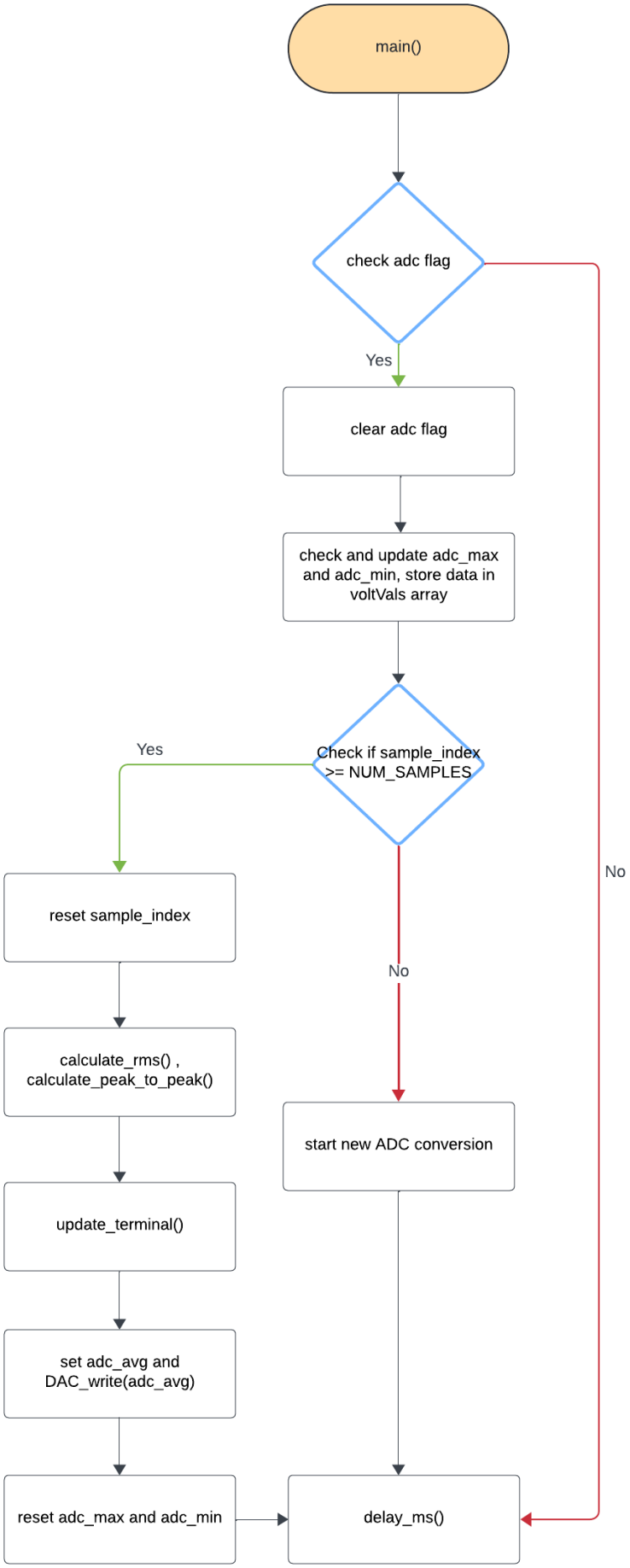
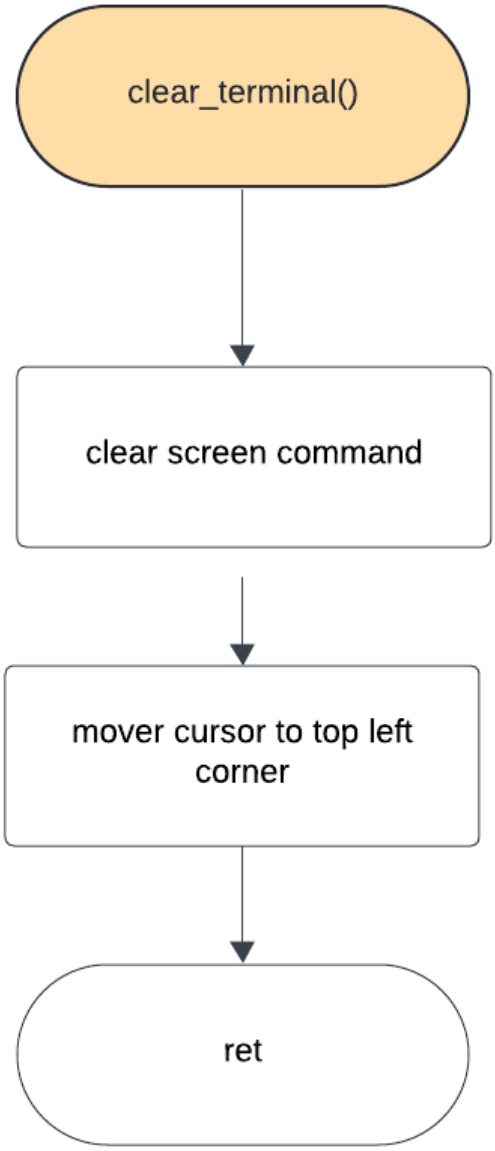


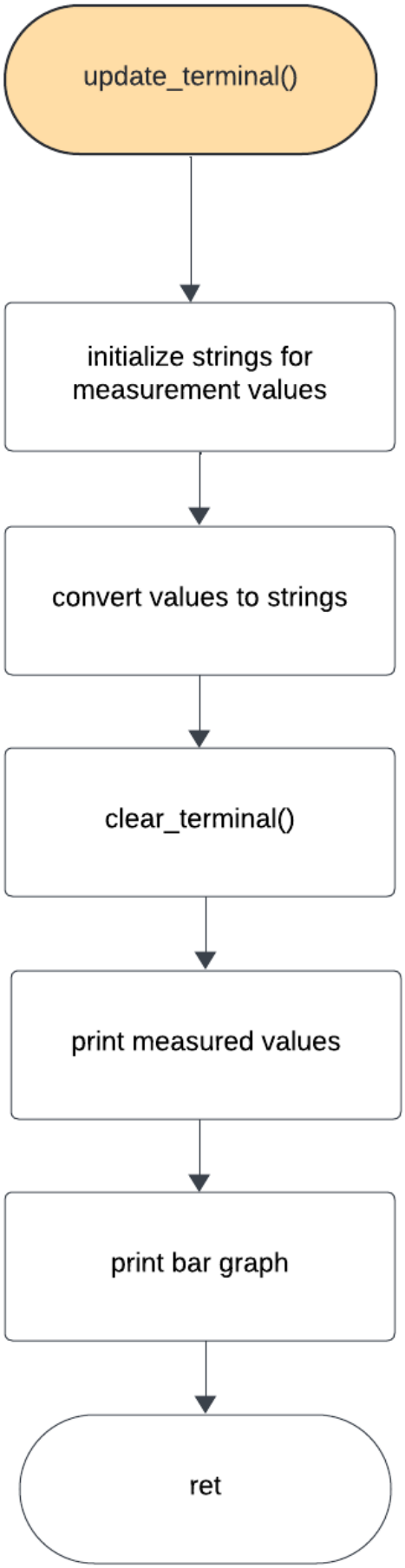
Figure 1: Output Terminal Display for 500Hz AC Signal with 1.00Vpp, 2V DC offset

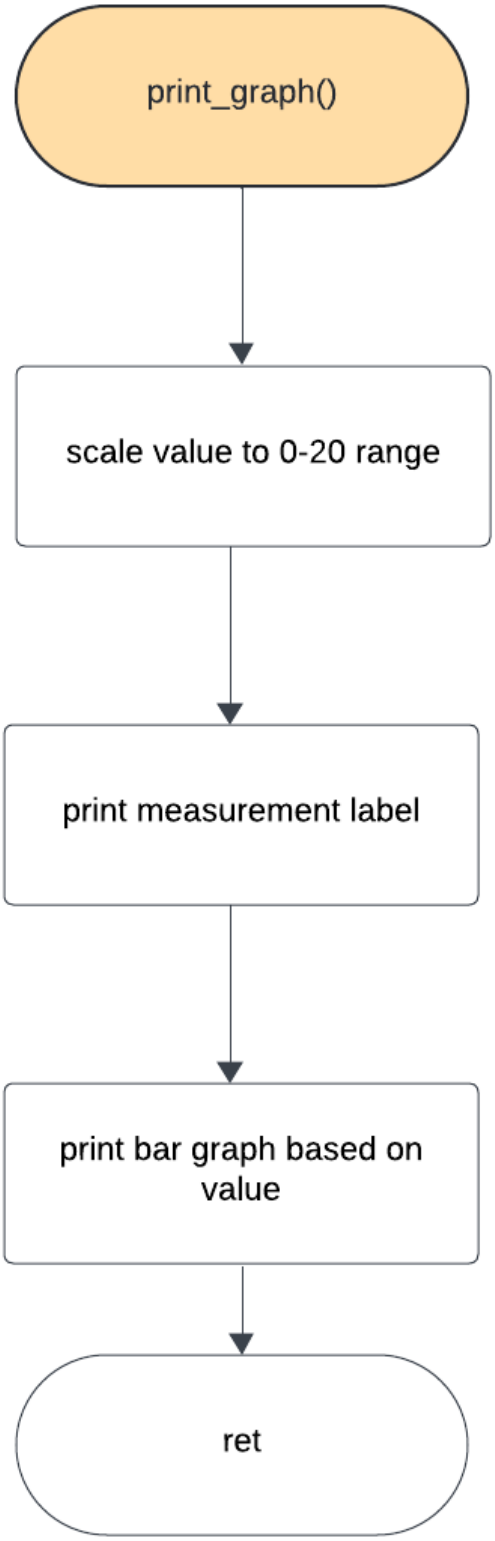

ADC1_2_IRQHandler()
check AD1-> ISR &
ADC_ISR_EOC
Yes
No
read adc_data and set
adc_flag
ret

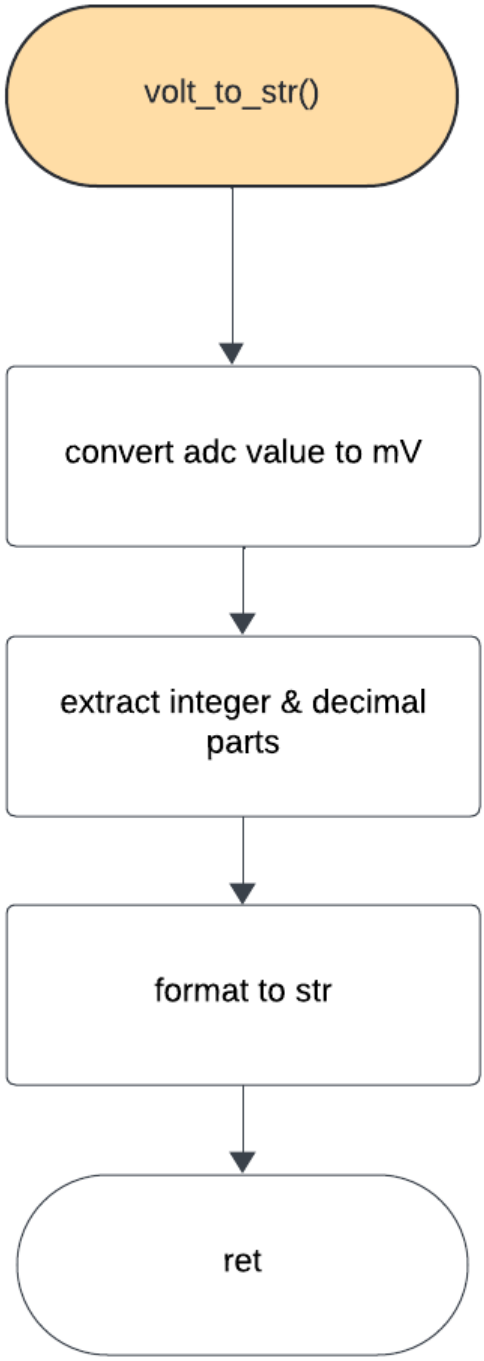

TIM2_IRQHandler()
check TIM2->ISR & TIM_SR_CC4IF
Yes
No
update prev and curr, then calculate frequency
Check TIM2->SR & TIM_SR_CC2IF
Yes
No
clear TIM_SR_CC2IF flag, increment CCR2, Start ADC conversion
ret

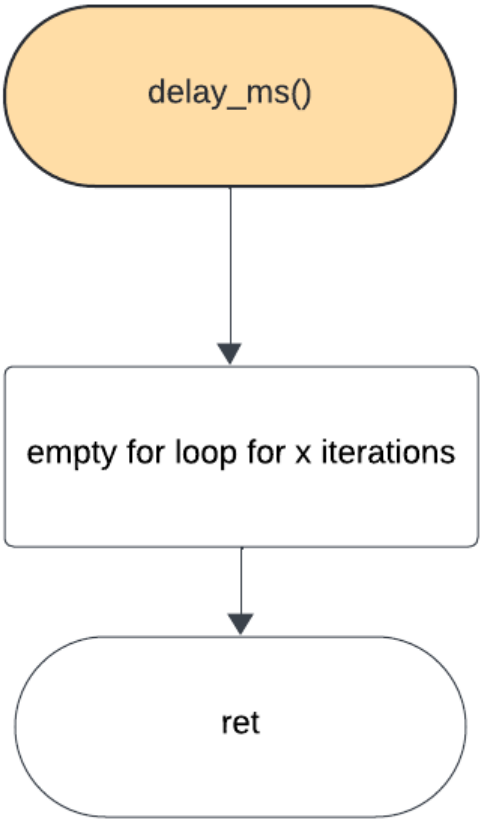

main()
check adc flag
Yes
No
clear adc flag
check and update adc_max and adc_min, store data in voltVals array
Check if sample_index >= NUM_SAMPLES
Yes
No
reset sample_index
calculate_rms() , calculate_peak_to_peak()
update_terminal()
set adc_avg and DAC_write(adc_avg)
reset adc_max and adc_min
start new ADC conversion
delay_ms()

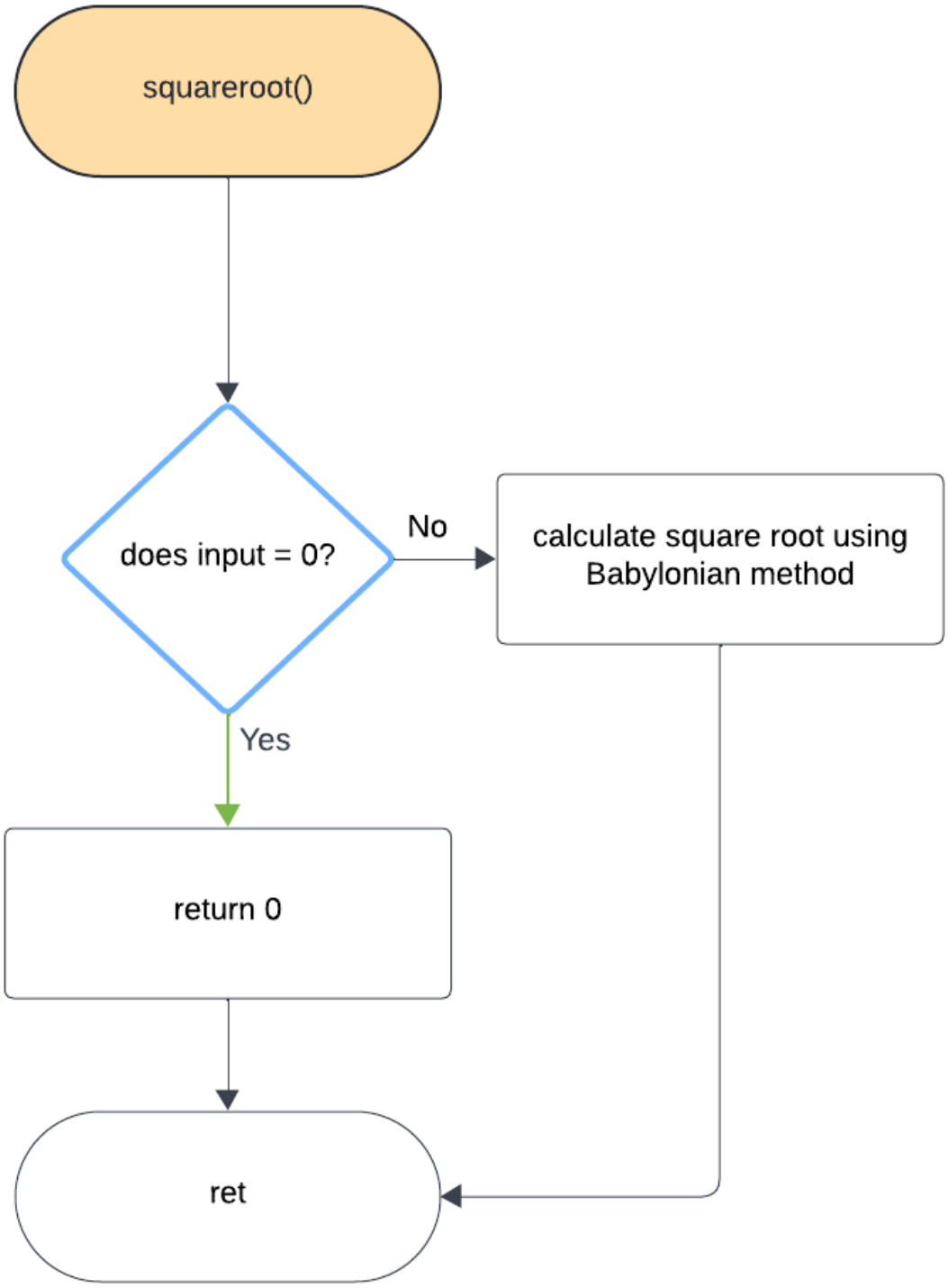

clear_terminal()
clear screen command
mover cursor to top left corner
ret

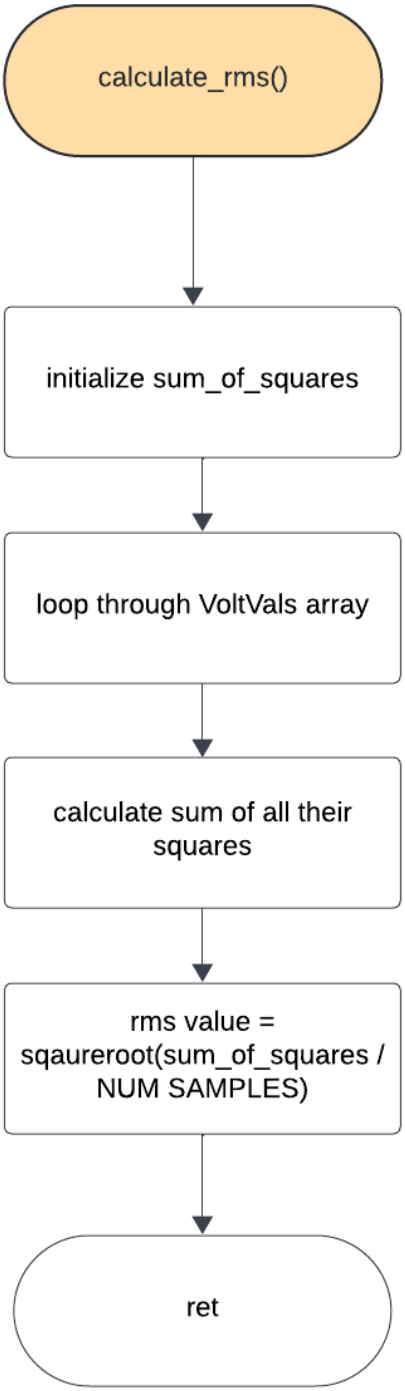

update_terminal()
initialize strings for measurement values
convert values to strings
clear_terminal()
print measured values
print bar graph
ret

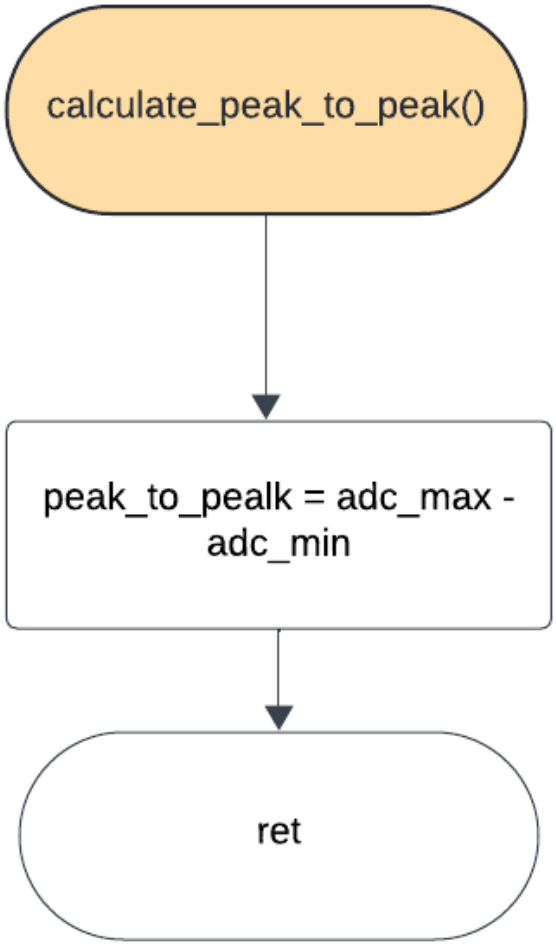

print_graph()
scale value to 0-20 range
print measurement label
print bar graph based on value
ret

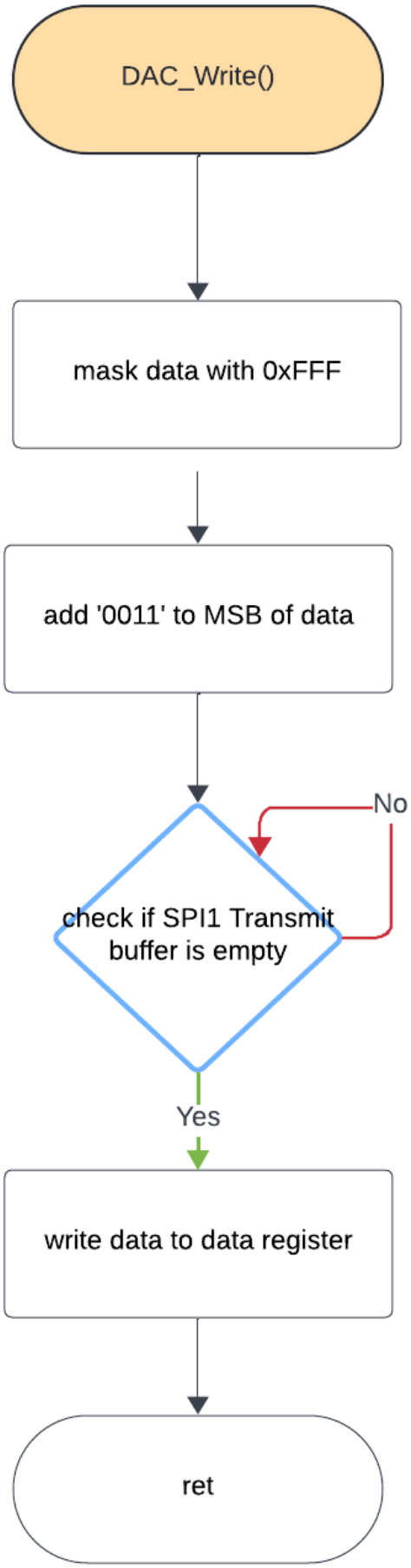

volt_to_str()
convert adc value to mV
extract integer & decimal parts
format to str
ret

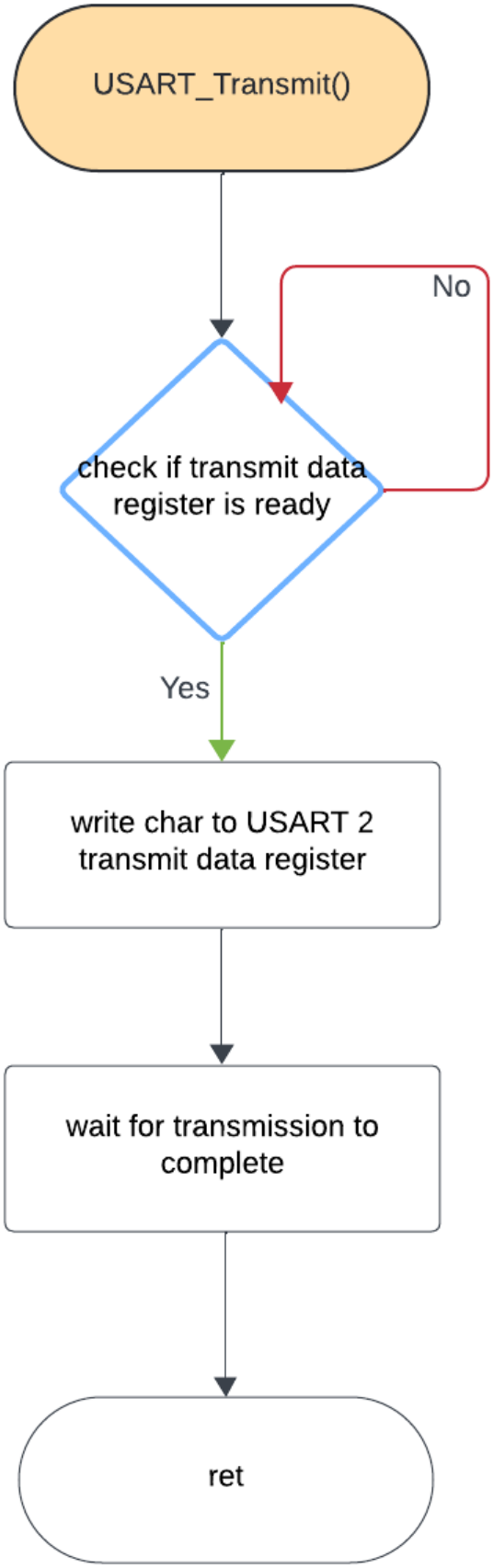

delay_ms()
empty for loop for x iterations
ret

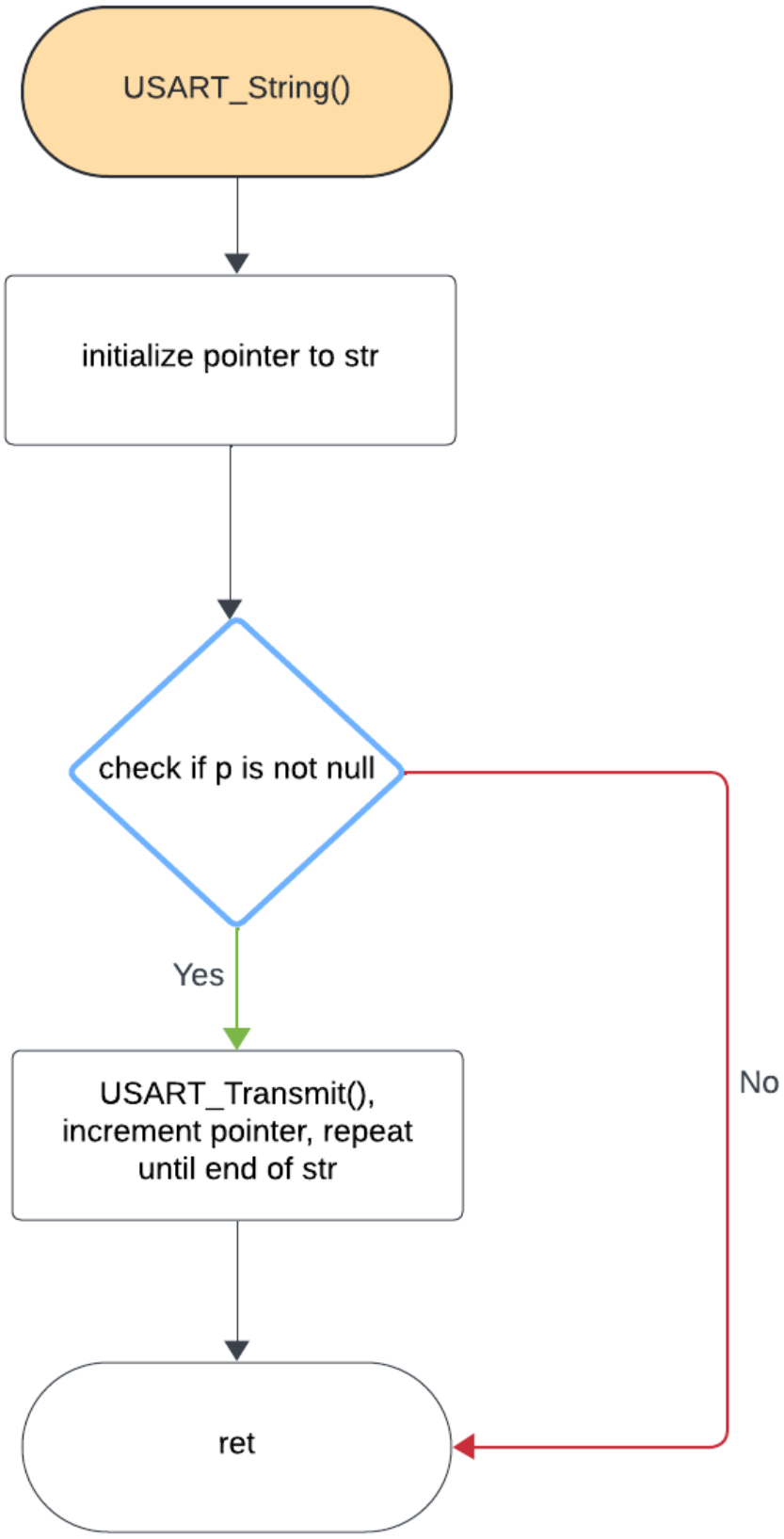

squareroot()
does input = 0?
No
calculate square root using Babylonian method
Yes
return 0
ret

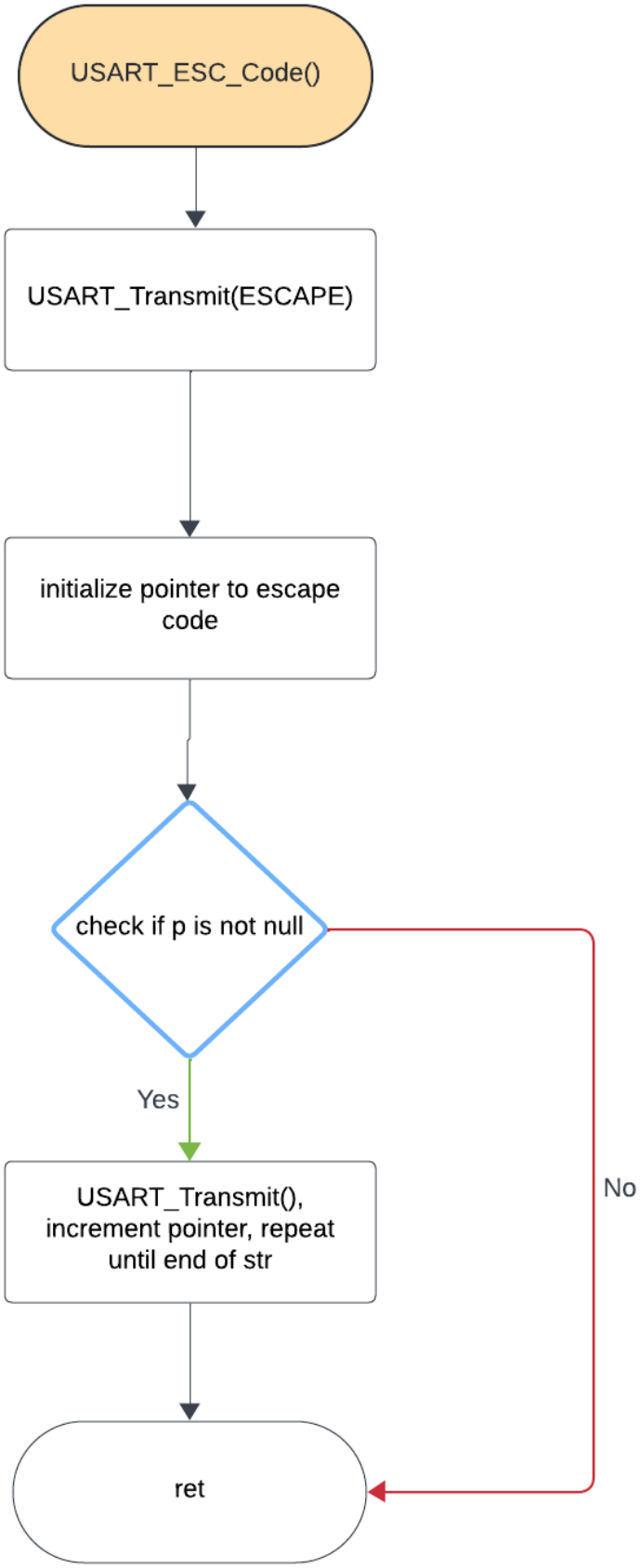

calculate_rms()
initialize sum_of_squares
loop through VoltVals array
calculate sum of all their squares
rms value = sqaureroot(sum_of_squares / NUM SAMPLES)
ret


calculate_peak_to_peak()
peak_to_pealk = adc_max - adc_min
ret


DAC_Write()
mask data with 0xFFF
add '0011' to MSB of data
check if SPI1 Transmit buffer is empty
No
Yes
write data to data register
ret


USART_Transmit()
No
check if transmit data register is ready
Yes
write char to USART 2 transmit data register
wait for transmission to complete
ret


USART_String()
initialize pointer to str
check if p is not null
Yes
No
USART_Transmit(), increment pointer, repeat until end of str
ret


USART_ESC_Code()
USART_Transmit(ESCAPE)
initialize pointer to escape code
check if p is not null
Yes
No
USART_Transmit(), increment pointer, repeat until end of str
ret

Code

main.c

```
#include "main.h"
#include "globals.h"
#include "terminal.h"
#include "calculations.h"
#include "ADC.h"
#include "comparator.h"
#include "timer.h"
#include "DAC.h"
#include "USART.h"
#include <stdint.h>

/* Define ------------------------------------------------------------------*/
#define ADC_SAMPLE_MAX 10000 // samples needed to calc Vref (1 s with 10 kHz sample rate)
#define CLK_FREQ 24000000 // internal clock freq
//#define NUM_SAMPLES 200 // gives good readings :)

/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);

/* Globals ------------------------------------------------------------------*/
uint16_t adc_data; // adc data var init
uint8_t adc_flag; // flag for adc conversion
uint32_t freq; // frequency var init
uint32_t voltVals[NUM_SAMPLES]; // array holds voltage samples
uint32_t rms_value = 0; // rms var init
volatile uint32_t adc_count = 0;
volatile int32_t adc_max = -1; // max adc sample, start with impossibly low value
volatile int32_t adc_min = 4097; // min adc sample, start with impossibly high value
volatile uint16_t peak_to_peak = 0; // init peak to peak var
volatile uint32_t sample_index = 0; // init sample idx var

int main(void)
{
  /* MCU Configuration--------------------------------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();

  /* Configure the system clock */
  SystemClock_Config();
  uint16_t adc_avg = (adc_max+adc_min)>>1; // avg adc value to use as Vref

  /* Initialize all configured peripherals */
  USART_Init();
  ADC1_Init();
  COMP1_Init();
  DAC_Init();
  TIM2_Init();
  /* Infinite loop */
  while (1)
  {
      if (adc_flag)
      {
          adc_flag = 0; // clear ADC global flag
          if(adc_data > adc_max){
              adc_max = adc_data;
          }
```

```
            if(adc_data < adc_min){
                adc_min = adc_data;
            }

            voltVals[sample_index++] = adc_data;

            if (sample_index >= NUM_SAMPLES)
            {
                // calculations and update terminal when max samples reached
                sample_index = 0;
                calculate_rms();
                calculate_peak_to_peak();
                update_terminal();

                // update dac with average value for comparator reference voltage
                adc_avg = (adc_max + adc_min) >> 1;
                DAC_Write(adc_avg);

                // reset max and min values
                adc_max = -1;
                adc_min = 4097;
            }

            // start another conversion
            ADC1->CR |= ADC_CR_ADSTART;
        }

        // delay
        delay_ms(1);
    }
}

void TIM2_IRQHandler(void) {

        static uint32_t prev=0;
        static uint32_t curr=1;

        // input capture happened
        if(TIM2->SR & TIM_SR_CC4IF) {
              prev = curr; // move last count to previous
              curr = TIM2->CCR4; // read captured count, clears CC4 flag automatically
              freq = CLK_FREQ / (curr - prev) + 1; // calc new freq
        }
        // CCR2 reached clocks for next ADC sample
        if(TIM2->SR & TIM_SR_CC2IF) {
              TIM2->SR &= ~ TIM_SR_CC2IF; // clear interrupt flag
              TIM2->CCR2 += TIM_SAMPLE_CC; // inc CCR to next milestone
              ADC1->CR |= ADC_CR_ADSTART; // start new conversion
        }
}

void ADC1_2_IRQHandler(void) {
        /* ADC ISR for EOC */
        if(ADC1->ISR & ADC_ISR_EOC) {
              adc_data = ADC1->DR; // read data, clears EOC flag automatically
              adc_flag = 1; // set global flag
        }
}
```

---

calculations.c

```c
#include "calculations.h"
#include "globals.h"

// peak to peak calculation
void calculate_peak_to_peak(void)
{
      peak_to_peak = adc_max - adc_min;
}


// rms calculation
void calculate_rms(void)
{
    uint32_t sum_of_squares = 0;
    for (int i = 0; i < NUM_SAMPLES; i++)
    {
        sum_of_squares += voltVals[i] * voltVals[i];
    }
    rms_value = squareroot(sum_of_squares / NUM_SAMPLES);
}

// sqrt function for efficiency
// babylonian algorithm
uint16_t squareroot(uint32_t value)
{
    if (value == 0)
      return 0;
    uint32_t x = value;
    uint32_t y = (x + 1) / 2;
    while (y < x)
    {
        x = y;
        y = (x + value / x) / 2;
    }
    return (uint16_t)x;
}

//delay
void delay_ms(uint32_t ms)
{
    for (volatile uint32_t i = 0; i < ms * 3000; i++){}
}
```

---

calculations.h

```c
#ifndef SRC_CALCULATIONS_H_
#define SRC_CALCULATIONS_H_
#include <stdint.h>

void calculate_rms(void);
void delay_ms(uint32_t ms);
uint16_t squareroot(uint32_t value);
void calculate_peak_to_peak(void);


#endif /* SRC_CALCULATIONS_H_ */
```

Terminal.c

```c
#include "USART.h"
#include "terminal.h"
#include "globals.h"

void volt_to_str(uint32_t adc_value, uint8_t *str)
{
    // convert adc value to mv
    uint32_t millivolts = (adc_value * 3300) / 4096;

    // extract integer and decimal
    uint32_t integer_part = millivolts / 1000;
    uint32_t decimal_part = millivolts % 1000;

    // format
    str[0] = '0' + integer_part;       // convert integer part
    str[1] = '.';                      //decimal point
    str[2] = '0' + (decimal_part / 100); // first decimal
    decimal_part %= 100;
    str[3] = '0' + (decimal_part / 10);  // second decimal
    str[4] = '0' + (decimal_part % 10);  // third decimal
    str[5] = '\0';                      // null-terminator
}

// prints bar graph
void print_graph(const char* label, uint16_t value) {
    int num = (value * 20) / 4096; // Scale value to 0-20 range for bar graph
    USART_String(label);
    for (int i = 0; i < 20; i++) {
        if (i < num) {
            USART_String("##");
        } else {
            USART_String(" ");
        }
    }
    USART_String("|\r\n|-----|-----|-----|-----|-----|-----|\r\n");
    USART_String("0    0.5   1.0   1.5   2.0   2.5   3.0\r\n");
}

void update_terminal(void)
{
      // initialize strs for values
    char str[10];
    uint8_t minVoltStr[10];
    uint8_t maxVoltStr[10];
    uint8_t rmsVoltStr[10];
    uint8_t peakToPeakStr[10];

    // store values in str format
    volt_to_str(adc_min, minVoltStr);
    volt_to_str(adc_max, maxVoltStr);
    volt_to_str(rms_value, rmsVoltStr);
    volt_to_str(peak_to_peak, peakToPeakStr);

    clear_terminal();

    // display frequency
    USART_String("Frequency: ");
    num_to_str(freq, str, 10);
    USART_String(str);
    USART_String(" Hz\r\n");
```

```
    // display dc voltage
    USART_String("DC Voltage: ");
    USART_String((char *)minVoltStr);
    USART_String(" V\r\n");

    // display ac rms
    USART_String("AC RMS: ");
    USART_String((char *)rmsVoltStr);
    USART_String(" V\r\n");

    // display ac peak to peak
    USART_String("AC Peak-to-Peak: ");
    USART_String((char *)peakToPeakStr);
    USART_String(" V\r\n");

    // display bar graphs
    print_graph("DC Voltage Bar: \r\n", adc_min); // dc voltage bar graph
    print_graph("AC RMS Bar: \r\n", rms_value); // ac rms bar graph
    }


void clear_terminal(void) {
    USART_String("\033[2J"); // clear screen
    USART_String("\033[H");  // move cursor top left corner
}
```

Terminal.h

```c
#ifndef SRC_TERMINAL_H_
#define SRC_TERMINAL_H_

#include "stm32l4xx_hal.h"
void volt_to_str(uint32_t adc_value, uint8_t *str);
void print_graph(const char* label, uint16_t value);
void update_terminal(void);
void clear_terminal(void);

#endif /* SRC_TERMINAL_H_ */
```

Globals.h

```
#ifndef SRC_GLOBALS_H_
#define SRC_GLOBALS_H_


#include <stdint.h>
#define NUM_SAMPLES 200 // gives good readings :)
extern uint16_t adc_data;
extern uint8_t adc_flag;
extern uint32_t freq;
extern uint32_t voltVals[NUM_SAMPLES];
extern uint32_t rms_value;
extern volatile uint32_t adc_count;
extern volatile int32_t adc_max;
extern volatile int32_t adc_min;
extern volatile uint16_t peak_to_peak;
extern volatile uint32_t sample_index;

#endif /* SRC_GLOBALS_H_ */
```

ADC.c

```c
#include "ADC.h"

void ADC1_Init(void) {
	/* configure ADC1 */

	// configure clocks
	RCC->AHB2ENR |= RCC_AHB2ENR_ADCEN;
	// set clock for synchronous hclk
	ADC123_COMMON->CCR = (1<<ADC_CCR_CKMODE_Pos);

	// power up ADC and enable voltage regulator
	ADC1->CR &= ~ADC_CR_DEEPPWD;
	ADC1->CR |= ADC_CR_ADVREGEN;

	// wait 20 us
	for(uint32_t i = 0; i<80; i++);

	// calibrate ADC (single-ended input)
	ADC1->CR &= ~(ADC_CR_ADEN | ADC_CR_ADCALDIF);
	ADC1->CR |= ADC_CR_ADCAL; // start calibration
	while(ADC1->CR & ADC_CR_ADCAL); // wait for calibration to end

	// configure channels for single ended mode
	ADC1->DIFSEL &= ~(ADC_DIFSEL_DIFSEL_5);

	// enable ADC
	ADC1->ISR |= ADC_ISR_ADRDY; // clear bit with 1
	ADC1->CR |= ADC_CR_ADEN; // set enable bit
	while(!(ADC1->ISR & ADC_ISR_ADRDY)); // wait for bit to be 1 (ready)
	ADC1->ISR |= ADC_ISR_ADRDY; // clear bit with 1 again

	// ADC single mode, 12-bit resolution, right aligned
	ADC1->CFGR = 0;
	ADC1->SMPR1 = 0 << ADC_SMPR1_SMP5_Pos; // 2.5 clock sampling (fastest)

	// sequence has length 1, channel 5
	ADC1->SQR1 = (5<<ADC_SQR1_SQ1_Pos);

	// enable interrupts
	ADC1->IER |= ADC_IER_EOCIE; // end of conversion interrupt
	NVIC->ISER[0] |= (1<<(ADC1_2_IRQn & 0x1F)); // enable interrupts in NVIC
	__enable_irq();

	// configure GPIO PA0 for analog input
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;
	GPIOA->MODER |= GPIO_MODER_MODE0; // analog mode = 0b11
	GPIOA->ASCR |= GPIO_ASCR_ASC0; // connect analog switch to ADC input
}
```

ADC.h

```c
#ifndef SRC_ADC_H_
#define SRC_ADC_H_

#include "stm32l4xx_hal.h"

void ADC1_Init(void);
void ADC1_2_IRQHandler(void);

#endif /* SRC_ADC_H_ */
```

comparator.c

```c
#include "comparator.h"

void COMP1_Init(void) {
	/*
	 * COMP1_INP = PC5 (V in) - analog input
	 * COMP1_INM --> PC4 (V ref) - analog input
	 * COMP1_OUT = PB0 (V out) - AF 12
	 */

	/* configure PB0 as AF12 for comparator output */
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOBEN;
	GPIOB->MODER &= ~GPIO_MODER_MODE0;
	GPIOB->MODER |= GPIO_MODER_MODE0_1; // PB0 AF
	GPIOB->OTYPER &= ~GPIO_OTYPER_OT0; // PP output for PB0
	GPIOB->OSPEEDR |= GPIO_OSPEEDR_OSPEED0; // very fast for PB0
	GPIOB->PUPDR &= ~GPIO_PUPDR_PUPD0; // no PUPD
	GPIOB->AFR[0] &= ~GPIO_AFRL_AFSEL0;
	GPIOB->AFR[0] |= 12 << GPIO_AFRL_AFSEL0_Pos; // PB0 -> AF 12

	/* configure PC4 and PC5 as analog inputs */
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOCEN;
	GPIOC->MODER |= GPIO_MODER_MODE5 | GPIO_MODER_MODE4;

	/* configure COMP */
	COMP1->CSR |= COMP_CSR_HYST; // high hysteresis
	COMP1->CSR |= COMP_CSR_POLARITY; // yes inverted
	COMP1->CSR &= ~COMP_CSR_INPSEL; // pos input: use PC5 for Vin
	COMP1->CSR |= COMP_CSR_INMSEL; // neg input: PC4
	COMP1->CSR &= ~COMP_CSR_PWRMODE; // high power
	COMP1->CSR |= COMP_CSR_EN; // enable comparator
}
```

---

comparator.h

```c
#ifndef SRC_COMPARATOR_H_
#define SRC_COMPARATOR_H_

#include "stm32l4xx_hal.h"

void COMP1_Init(void);

#endif /* SRC_COMPARATOR_H_ */
```

DAC.c

```c
#include "DAC.h"

void DAC_Init(void) {
	/* clock for GPIOA */
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;

	/* GPIO SPI: AF5 SCK PA5, MOSI PA7 */
	SPI_PORT->MODER &= ~(GPIO_MODER_MODER5 | GPIO_MODER_MODER7);
	SPI_PORT->MODER |= GPIO_MODER_MODER5_1 | GPIO_MODER_MODER7_1;
	SPI_PORT->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR5 | GPIO_OSPEEDER_OSPEEDR7;
	SPI_PORT->OTYPER &= ~(GPIO_OTYPER_OT_5 | GPIO_OTYPER_OT_7);
	SPI_PORT->PUPDR &= ~(GPIO_PUPDR_PUPDR5 | GPIO_PUPDR_PUPDR7);

	/* GPIO SPI: AF5 NSS PA15 - NEEDED?? */
	SPI_PORT->MODER &= ~(GPIO_MODER_MODER15);
	SPI_PORT->MODER |= GPIO_MODER_MODER15_1;
	SPI_PORT->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR15;
	SPI_PORT->OTYPER &= ~(GPIO_OTYPER_OT_15);
	SPI_PORT->PUPDR &= ~(GPIO_PUPDR_PUPDR15);

	/* GPIO SPI: Output Mode NSS PA4 */
	SPI_PORT->MODER &= ~GPIO_MODER_MODER4;
	SPI_PORT->MODER |= GPIO_MODER_MODER4_0;
	SPI_PORT->OSPEEDR |= GPIO_OSPEEDER_OSPEEDR4;
	SPI_PORT->OTYPER &= ~GPIO_OTYPER_OT_4;
	SPI_PORT->PUPDR &= ~GPIO_PUPDR_PUPDR4;

	/* SCK AF5PA5 */
	SPI_PORT->AFR[0] &= ~((0x000F << GPIO_AFRL_AFSEL5_Pos));
	SPI_PORT->AFR[0] |=  ((0x0005 << GPIO_AFRL_AFSEL5_Pos));

	/* NSS AF5PA15 */
	SPI_PORT->AFR[1] &= ~((0x000F << GPIO_AFRH_AFSEL15_Pos));
	SPI_PORT->AFR[1] |=  ((0x0005 << GPIO_AFRH_AFSEL15_Pos));

	/* MOSI AF5PA7 */
	SPI_PORT->AFR[0] &= ~((0x000F << GPIO_AFRL_AFSEL7_Pos));
	SPI_PORT->AFR[0] |=  ((0x0005 << GPIO_AFRL_AFSEL7_Pos));

	/* clock for SPI */
	RCC->APB2ENR |= RCC_APB2ENR_SPI1EN;
	SPI1->CR1 &= ~(SPI_CR1_BR);

	/* 0:0 polarity + phase */
	SPI1->CR1 &= ~( SPI_CR1_CPOL | SPI_CR1_CPHA );

	/* TX RX mode */
	SPI1->CR1 &= ~( SPI_CR1_RXONLY );

	/* MSB mode */
	SPI1->CR1 &= ~( SPI_CR1_LSBFIRST );

	/* STM as master */
	SPI1->CR1 |= SPI_CR1_MSTR;

	SPI1->CR2 |= SPI_CR2_DS; // 16 bit data
	SPI1->CR2 |= SPI_CR2_SSOE; // enable hardware management of CS to avoid mode fault
	SPI1->CR2 &= ~( SPI_CR2_FRF); // motorola mode

	/* disable interrupts for transmit empty and receive not empty */
```

```
	SPI1->CR2 &= ~( SPI_CR2_TXEIE | SPI_CR2_RXNEIE );
	SPI1->CR2 &= ~SPI_CR2_FRXTH;

	/* enable SPI config */
	SPI1->CR1 |= SPI_CR1_SPE;
}

void DAC_Write(uint16_t data) {
	/*
	 * Assumes Pin at CS_MASK of SPI_PORT is GPIO output acting as CS
	 */

	/* 12 bit data from KeypadRead*/
	data &= DAC_DATA_MASK;

	/* add 0011 to MSB */
	uint16_t msg = DAC_CONTROLS | data;

	/* SET CS LOW */
	SPI_PORT->ODR &= ~GPIO_ODR_OD4;

	/* ensure all previous data is already sent */
	while (!(SPI1->SR & SPI_SR_TXE));

	/* write message */
	SPI1->DR = msg;

	/* wait for this data to be sent */
	while (SPI1->SR & SPI_SR_BSY);

	/* SET CS HIGH */
	SPI_PORT->ODR |= GPIO_ODR_OD4;
}
```

DAC.h

```c
#ifndef SRC_DAC_H_
#define SRC_DAC_H_

#include "stm32l4xx_hal.h"
#include <stdint.h>

#define SPI_PORT        GPIOA
#define DAC_DATA_MASK   0xFFF
#define DAC_CONTROLS    0x3 << 12

void DAC_Init(void);
void DAC_Write(uint16_t);
#endif /* SRC_DAC_H_ */
```

Timer.c

```c
#include "timer.h"
void TIM2_Init(void) {
	/*
	 * Call this function LAST when initializing peripherals because it enables interrupts globally.
	 */
	/* enable timer clock */
	RCC->APB1ENR1 |= RCC_APB1ENR1_TIM2EN;
	/* set up TIM2 to count up mode */
	TIM2->PSC = 0; // no prescaling
	TIM2->ARR = TIM_MAX; // timer counts up to specified maximum
	TIM2->CCR2 = TIM_SAMPLE_CC-1; // CCR2 = specified value for sample rate
	TIM2->SR &= ~(TIM_SR_CC4IF | TIM_SR_CC2IF); // clear interrupt flags
	/* input capture compare for CCR4 */
	TIM2->CCMR2 &= ~TIM_CCMR2_CC4S;
	TIM2->CCMR2 |= TIM_CCMR2_CC4S_0; // configure CCR1 as input connected to TI1
	TIM2->CCMR2 &= ~TIM_CCMR2_IC4F;
	TIM2->CCMR2 |= 15 << TIM_CCMR2_IC4F_Pos; // filter
	TIM2->CCER &= ~(TIM_CCER_CC4P | TIM_CCER_CC4NP); // non inverted, rising edge
	TIM2->CCMR2 &= ~TIM_CCMR2_IC4PSC; // no prescaler
	TIM2->CCER |= TIM_CCER_CC4E; // enable input capture control
	TIM2->OR1 = TIM2_OR1_TI4_RMP_0; // connect CCR4 input to COMP1
	/* enable interrupt on input capture */
	TIM2->DIER |= TIM_DIER_CC4IE | TIM_DIER_CC2IE; // enable interrupts in timer
	NVIC->ISER[0] |= (1 << (TIM2_IRQn & 0x1F)); // enable interrupt in NVIC
	__enable_irq(); // enable interrupts globally
	/* start timer counting */
	TIM2->CR1 |= TIM_CR1_CEN;
}
```

Timer.h

```
#ifndef SRC_TIMER_H_
#define SRC_TIMER_H_

#include "stm32l4xx_hal.h"

#define TIM_MAX                0xFFFFFFFF // number for ARR
#define TIM_SAMPLE_CC   2400; // number for CCR2 (clocks between samples)
                                          // this is 10 kHz sample rate with internal clock
= 24 MHz

void TIM2_Init(void);
void TIM2_IRQHandler(void);

#endif /* SRC_TIMER_H_ */
```

USART.c

```c
#include "USART.h"

void USART_Init(void) {
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN; // Enable Port A CLK for USART2 TX and RX
	RCC->APB1ENR1 |= RCC_APB1ENR1_USART2EN; // Enable USART2 Peripheral CLK

	GPIOA->MODER &= ~(GPIO_MODER_MODE2 | GPIO_MODER_MODE3); // Clear the mode
	GPIOA->MODER |= (GPIO_MODER_MODE2_1 | GPIO_MODER_MODE3_1); // Set Alternate Function
mode
	GPIOA->OSPEEDR |= (GPIO_OSPEEDR_OSPEED2 | GPIO_OSPEEDR_OSPEED3); // high speed
	GPIOA->AFR[0] |= ((7 << GPIO_AFRL_AFSEL2_Pos) | (7 << GPIO_AFRL_AFSEL3_Pos));


	USART2 -> CR1 |= (USART_CR1_RXNEIE);
	USART2->BRR = USARTDIV; // Baud rate 115200
	USART2 -> CR1 |= (USART_CR1_TE | USART_CR1_RE | USART_CR1_UE);


	NVIC -> ISER[1] |= (1<<(USART2_IRQn & 0x1F));
	__enable_irq();
}

void USART_Transmit(char c) {
	while (!(USART2->ISR & USART_ISR_TXE));
	USART2->TDR = c;
	// Wait for the transmission to complete
	while (!(USART2->ISR & USART_ISR_TC));
}

void USART_String(char* str) {
	// Send the escape code sequence
	for (char* p = str; *p != '\0'; p++) {
		USART_Transmit(*p);
	}
}

void USART_ESC_Code(char* escape_code) {
	// Send the escape character
	USART_Transmit(ESCAPE);
	// Send the escape code sequence
	for (char* p = escape_code; *p != '\0'; p++) {
		USART_Transmit(*p);
	}
}

char* num_to_str(uint32_t num, char* buff, uint32_t buff_len) {
	/*
	 * convert an integer to a string, up to 4 digits (left pad with 0)
	 */
	int32_t i = buff_len-1;
	buff[i--]='\0';
	while(num>=0 && i>=0) {
		buff[i--]=num%10+'0';
		num/=10;
	}
	return buff+i+1;
}
```

USART.h

```
#ifndef SRC_USART_H_
#define SRC_USART_H_

#include "stm32l4xx_hal.h"

#define USARTDIV 208          // 115.2 kbps
#define ESCAPE '\033'

void USART_Init(void);
void USART_Transmit(char);
void USART_ESC_Code(char*);
void USART_String(char*);
void USART2_IRQHandler(void);
char* num_to_str(uint32_t num, char* buff, uint32_t buff_len);

#endif /* SRC_USART_H_ */
```

References

STMicroelectronics. “UM1724 User manual - STM32 Nucleo-64 boards (MB1136).” STMicroelectronics, https://www.st.com/resource/en/user_manual/um1724-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf.

STMicroelectronics. “STM32L476xx Ultra-low-power Arm® Cortex®-M4 32-bit MCU+FPU, 100DMIPS, up to 1MB Flash, 128 KB SRAM, USB OTG FS, LCD, ext. SMPS.” STMicroelectronics, https://www.st.com/resource/en/datasheet/stm32l476rg.pdf.

STMicroelectronics. “RM0351 Reference manual - STM32L47xxx, STM32L48xxx, STM32L49xxx, and STM32L4Axxx advanced Arm®-based 32-bit MCUs.” STMicroelectronics, https://www.st.com/resource/en/reference_manual/rm0351-stm32l47xxx-stm32l48xxx-stm32l49xxxand-stm32l4axxx-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

Microchip Technology Inc. MCP4921/4922 12-Bit DAC with SPI Interface. 2007. 21897B.pdf (microchip.com)