Project 2 - Digital Multimeter

Kieran Valino

May 24, 2023

Behavior Description

This digital multimeter (DMM) reads in either the voltage of a direct current (DC) or the voltage of an alternating current (AC) in the form of a sinusoidal, triangle, sawtooth, and square waveform. This digital multimeter accepts voltages between 0 to 3.3V, and any higher will end up damaging the device. When in DC mode, the digital multimer will provide the measurement of the average voltage (V_{AVG}) and output it to the voltage graph. When in AC mode, the digital multimeter will provide the measurement of the AC wave frequency (Hz), the root-mean-squared voltage (V_{RMS}), the peak-to-peak voltage (V_{PP}), and the wave's DC offset. When in DC mode, a graphical representation will be outputted showcasing V_{AVG}. When in DC mode, a graphical representation will be outputted showcasing V_{RMS}. Switching to AC mode can be accomplished by inputting the "A" key on the keyboard into the terminal. Switching to DC mode can be accomplished by inputting the "D" key on the keyboard into the terminal.

System Specifications:

Board:

TYPE	DESCRIPTION
Name	STM32L476RG
Manufacturer	STMicroelectronics
Series	STM32L4
Core	ARM Cortex M4
Mounting Type	MCU 32 bit
Operating Frequency	32 MHz
Flash Memory Size	1 Mbyte
Operating Supply Voltage	3.3V
Package Pin Count	64 pins
Interface Type	USB
Unit Weight	10.582189 oz.

Table 1: Board specifications

System Schematic:

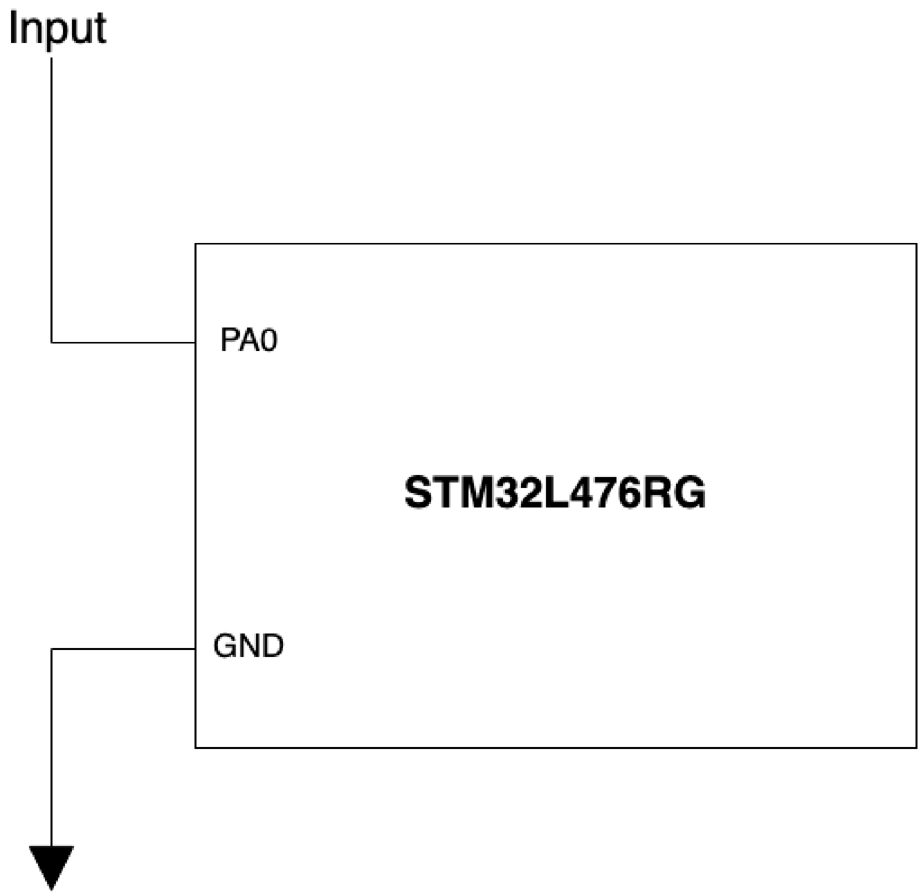

Figure 1: Digital Multimeter Schematic showcasing how the STM32L476RG board is connected to the voltage input that is to be read by the Analog-to-Digital Converter (ADC) on the STM32L476RG board and displayed to the terminal through the STM32L476RG board's Universal Asynchronous Receiver/Transmitter (UART). Physical input to handle switching between DC mode and AC mode is also handled through UART on the STM32L476RG board.

Note: Handling of the frequency calculation for AC mode is handled using a Fast Fourier Transform method, via software. The only physical hardware required is the GPIO pin, the voltage input source, and the wires.

Software Architecture:

Main function:

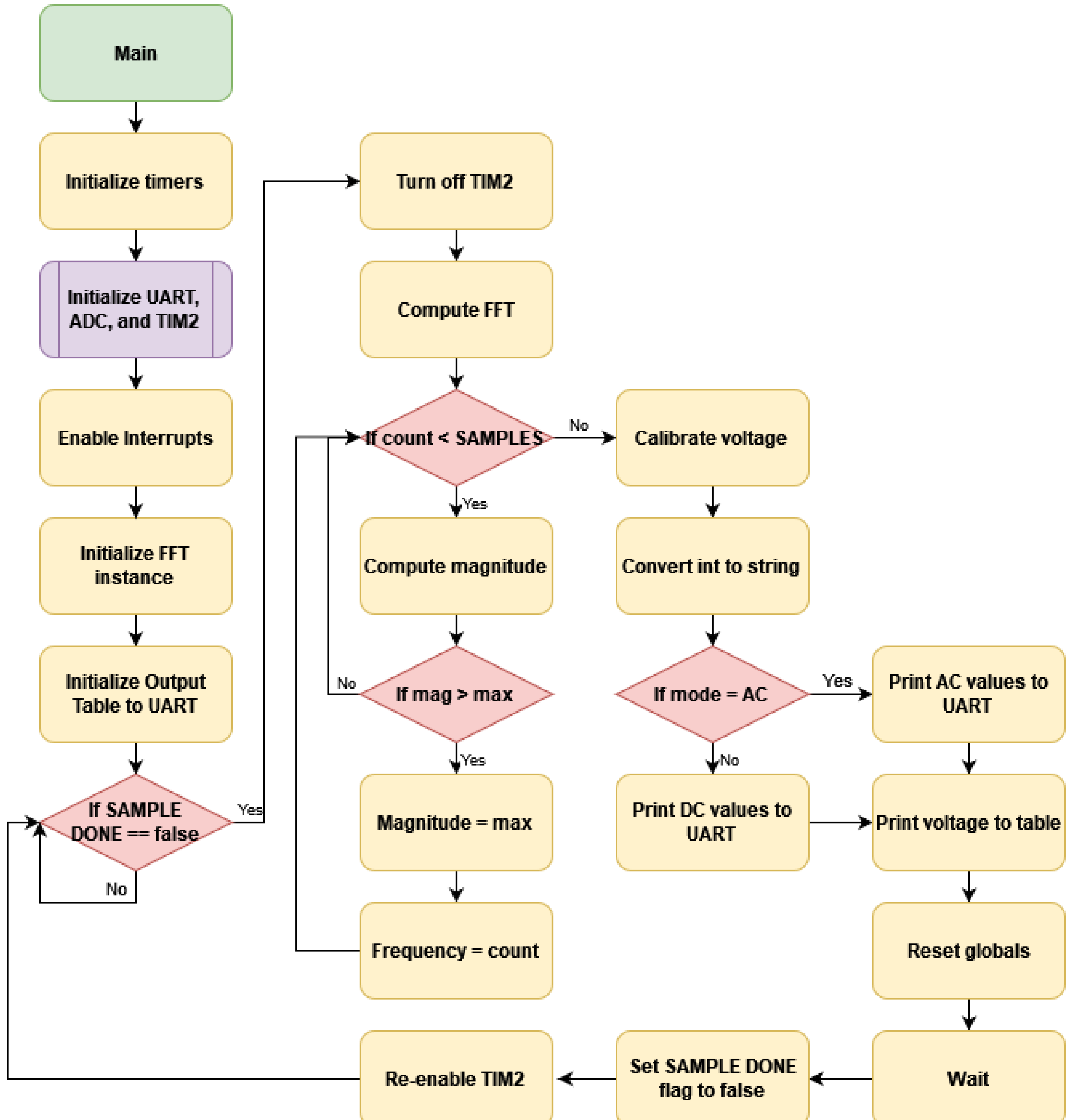

Figure 2: Main function flowchart demonstrating the main process of the Digital Multimeter. After initialization, the program checks to see if sampling through the ADC is done. If so, it will turn off the timer and compute the frequency through the Fast Fourier Transform Method. It will then calibrate the voltage measurements and output depending on the mode. The program will then re-enable the timer and tell the ADC that sampling is not done.
(see appendix for code details)

ADC Interrupt Handler function:

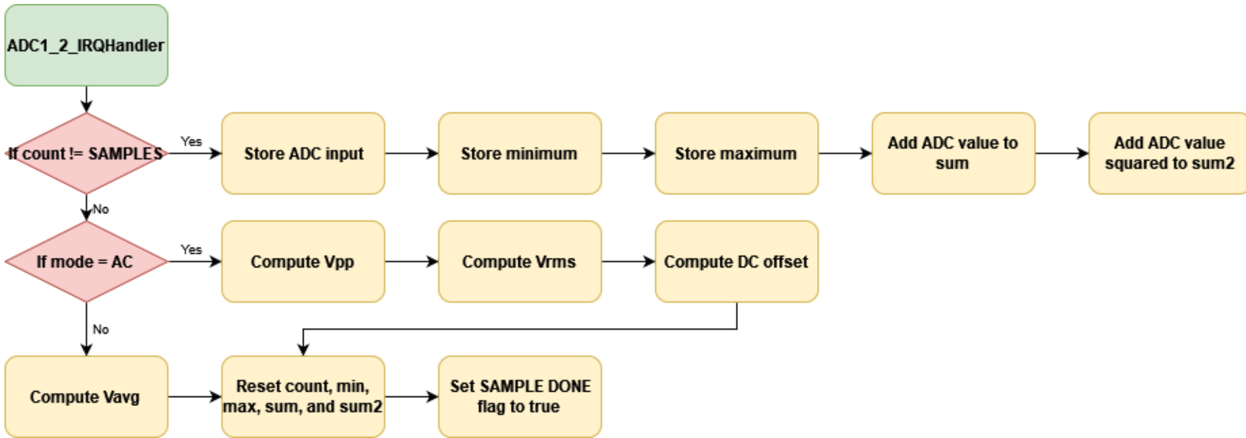

Figure 3: Interrupt handler function flowchart for the ADC demonstrating the ADC interrupt process of the Digital Multimeter. During the interrupt, the function will check if the ADC has measured the number of samples needed. If not, it will input the ADC value and adjust the measurements based on if the ADC value is either a minimum or a maximum of the total number of samples measured. From there, it will add the value to the total sum and add the value squared for the sum squared. Once the number of samples has been reached, the interrupt handler will compute V_{PP}, V_{AVG}, V_{RMS}, and the DC offset depending on the mode. The total sum will be used to compute V_{AVG} for DC mode and the sum squared will be used to compute V_{RMS} for AC mode. The interrupt handler will reset the local values and tell the main function that sampling is done.
(see appendix for code details)

TIM2 Interrupt Handler function:

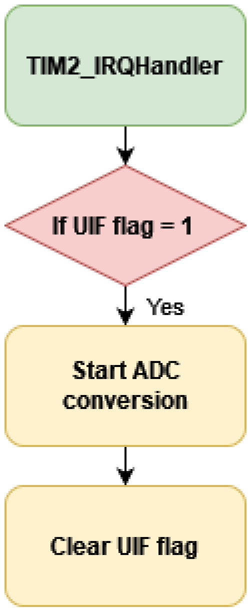

Figure 4: Interrupt handler function flowchart for the Timer 2 demonstrating the Timer 2 interrupt process of the Digital Multimeter. Once the UIF flag is set by hardware, the interrupt handler tells the ADC to start another ADC conversion and then clears the UIF flag.
(see appendix for code details)

UART Interrupt Handler function:

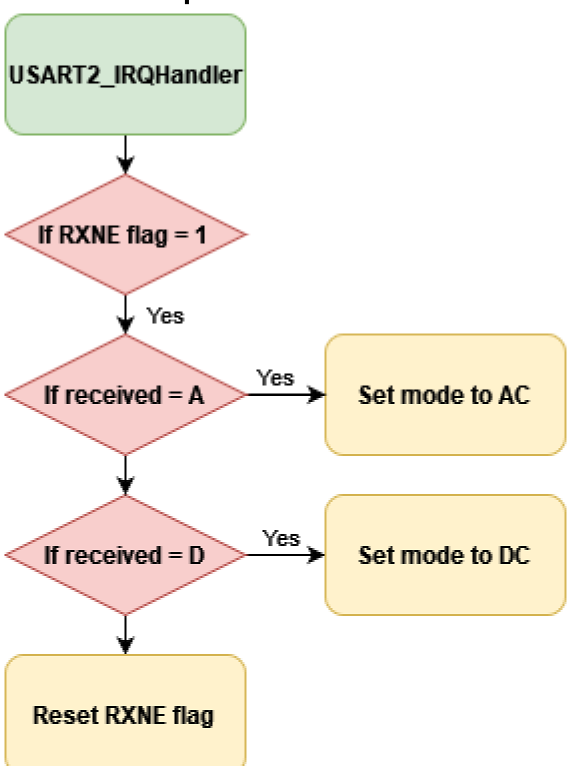

Figure 5: Interrupt handler function flowchart for UART demonstrating the UART interrupt process of the Digital Multimeter. Once a character has been received, toggling the RXNE bit, the interrupt handler checks if the character is an “A” or a “D”. If the character is an “A”, the mode is set to AC mode. If the character is a “D”, the mode is set to DC mode. The handler then clears the RXNE bit.
(see appendix for code details)

UART functions:

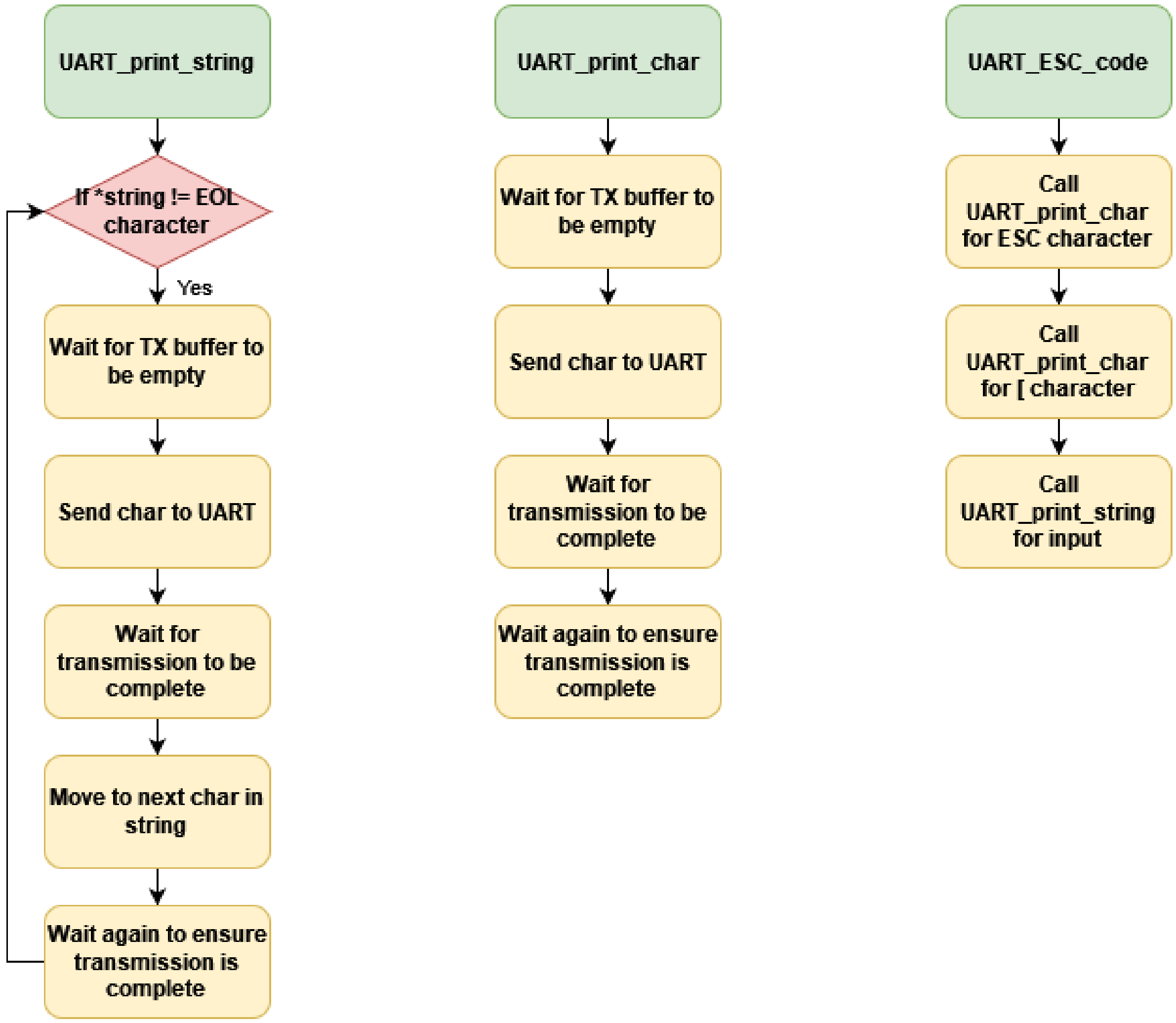

Figure 6: UART functions flowchart for UART to assist in outputting data to the terminal for the Digital Multimeter.

UART_print_string checks if the string has not reached the End-Of-Line character ('\0'). If it has not yet, the function will check if the TX buffer is empty, told by if the TXE bit is clear. The function will then output the character to UART and wait if the transmission is complete through the TC bit. The function will then increment the string pointer to the next character, wait, and begin the cycle again until the End-Of-Line character.

UART_print_char is the same code as UART_print_string but without checking for the End-Of-Line character and adjusting the character pointer because the function only runs once.

UART_ESC_code combines UART_print_string and UART_print char to output escape codes to UART.
(see appendix for code details)

ADC Voltage Conversion function:

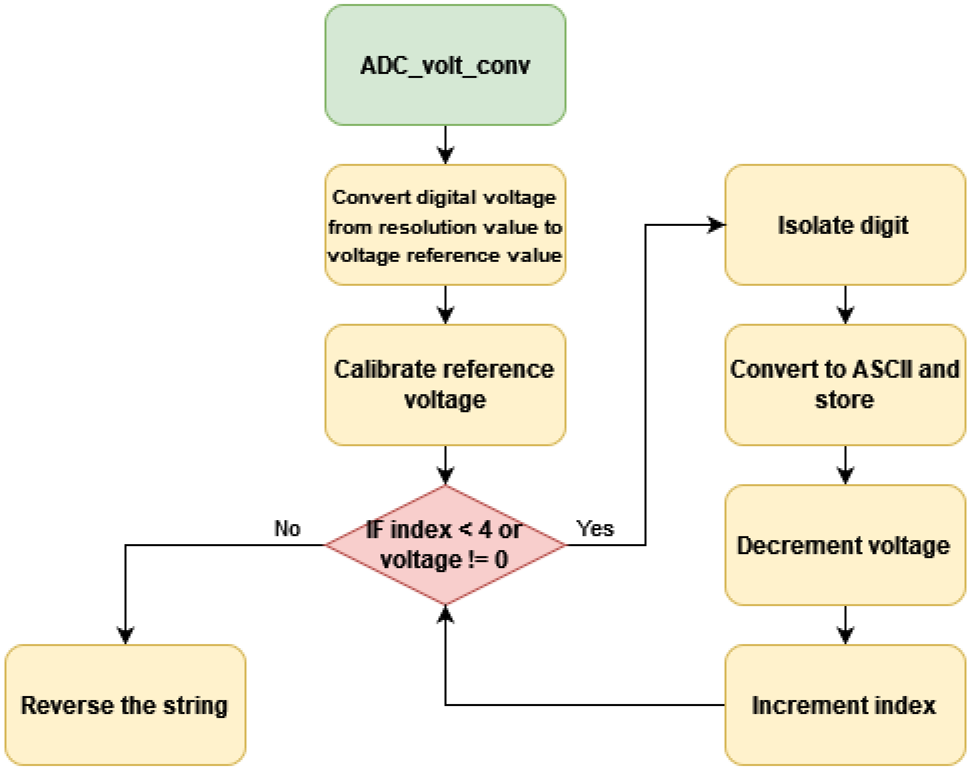

Figure 7: ADC voltage conversion flowchart for the ADC to calibrate the ADC data into millivolts for the Digital Multimeter. The function will first convert the digital voltage value from the 4096-bit resolution to the 3.3 V reference voltage. From there, it will calibrate the voltage to be closer to the true voltage value. Next, the function will convert the millivolt integer value to a character array, or string, by isolating the last digit, converting the digit to ASCII, storing it in the character array, decrementing the voltage by removing that digit, and incrementing the index until the voltage is 0 or the index is out of range of the 4 digit millivolt value. However, since the function removed the last digit first, the string will need to be reversed.
(see appendix for code details)

Print Graph function:

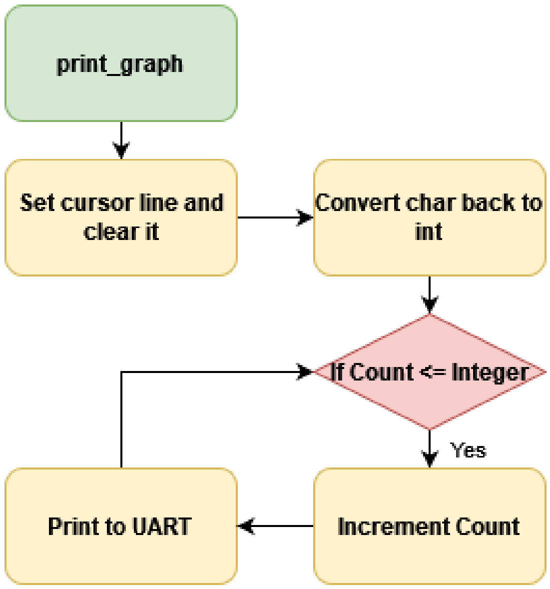

Figure 8: Voltage printing flowchart to assist in outputting voltages to the graph for the Digital Multimeter. After moving the cursor to the desired location, the function will convert the character array voltages back into an integer. It will then print the integer to UART for the graph by incrementing the integer value by 100 to fit within the graph.
(see appendix for code details)

Print Voltage function:

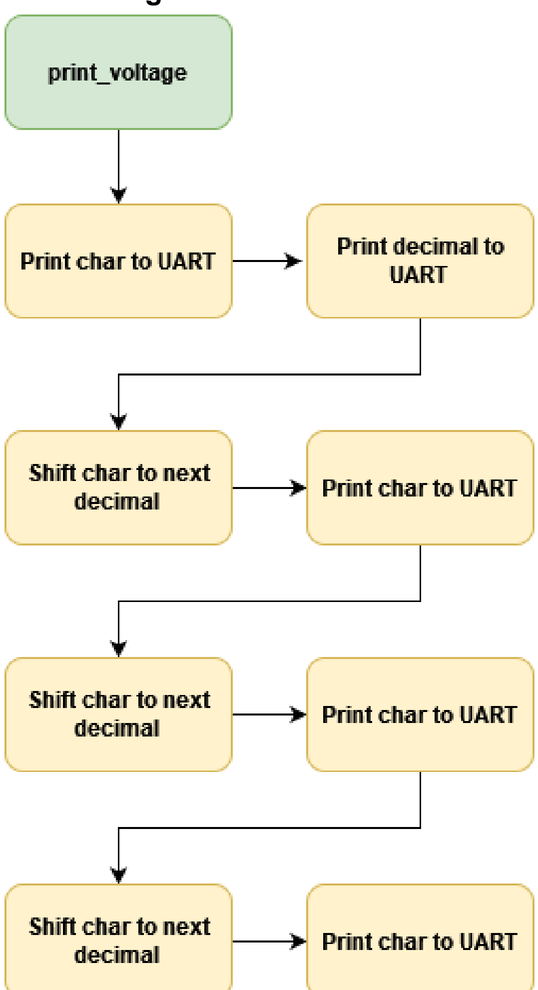

Figure 9: Voltage printing flowchart to assist in outputting voltages for the Digital Multimeter. The function prints out each millivolt byte to UART, but converts millivolts to volts by placing a decimal after the first digit.
(see appendix for code details)

Uint16 to String Function:

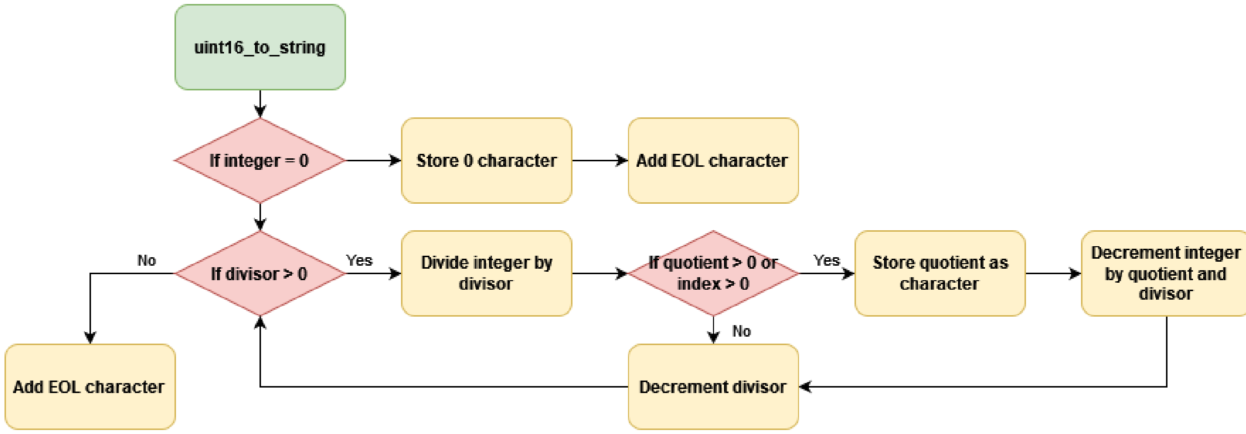

Figure 10: Unsigned 16-bit integer to string flowchart to assist in outputting the frequency for the Digital Multimeter. The function first handles if the integer is a zero by storing the zero character in the character array and ending with the End of Line character ('\0'). The function will then loop through by checking if the divisor is greater than zero. If so, it will divide the integer by the divisor and check if the quotient or index is greater than 0. If so, it will store the quotient as and ASCII character into the character away and decrement the divisor and the integer. Once the loop is no longer satisfied, it will end the character array with the End-of-Line character.
(see appendix for code details)

Appendix:

main.h

```
#ifndef __MAIN_H
#define __MAIN_H

#ifdef __cplusplus
extern "C" {
#endif


#define DC 0
#define AC 1

#define ADC_min_calib 20
#define ADC_max_calib 5

void SystemClock_Config(void);
void print_voltage(char*);
void print_graph(char*);

/* Includes ------------------------------------------------------------------*/
#include "stm32l4xx_hal.h"

/* Private includes ----------------------------------------------------------*/
void Error_Handler(void);


#ifdef __cplusplus
}
#endif

#endif /* __MAIN_H */
```

main.c

```c
#include "main.h"
#include "arm_math.h"
#include "adc.h"
#include "uart.h"
#include "math.h"

// globals
float32_t samples[SAMPLES] = {0};
char SAMPLE_DONE = FALSE;
char mode = DC;
uint16_t voltage;
uint16_t vpp;
uint16_t dc_offset;

int main(void)
{
  HAL_Init();
  SystemClock_Config();
  ADC_init();
  UART_init();

  float32_t fft[SAMPLES];     // FFT data array
  uint16_t frequency;
  uint64_t max;
  uint64_t magnitude;

  // Character Printouts
  char cfreq[4];
  char cvolt[4];
  char cvolt2[4];
  char cvolt3[4];

  // Configure TIM2 for interrupt; ARR may change due to calibration
  RCC->APB1ENR1 |= (RCC_APB1ENR1_TIM2EN);
  TIM2->DIER |= (TIM_DIER_UIE);
  TIM2->ARR = 15750;
  TIM2->CR1 |= TIM_CR1_CEN;

  NVIC->ISER[0] = (1 << (TIM2_IRQn & 0x1F));
  __enable_irq();

  // instantiate fast Fourier transform struct
  arm_rfft_fast_instance_f32 fft_instance;
  arm_rfft_fast_init_f32(&fft_instance, SAMPLES);

  // Initialize table
  UART_ESC_code("2J");
  UART_ESC_code("8:0H");
  UART_print_string("|----|----|----|----|----|----|----");
```

```
UART_ESC_code("9:0H");
UART_print_string("0   0.5  1.0  1.5  2.0  2.5  3.0");

while (1)
{
  // wait for sample data to be collected and disable clock
  while(SAMPLE_DONE == FALSE);
  TIM2->CR1                  &= ~(TIM_CR1_CEN);


  arm_rfft_fast_f32(&fft_instance, samples, fft, 0);

  // 0 is DC Offset
  // Find the max of the real, imaginary pairs
  // Ex: [2][3] = 1 Hz, [4][5] = 2 Hz

  max = SAMPLES;      // 1024 to prevent floating
  magnitude = 0;
     frequency = 0;

  for (uint16_t i = 2; i < SAMPLES; i += 2) {
            magnitude = (fft[i] * fft[i]) + (fft[i+1] * fft[i+1]);
            if (magnitude >= max){
              max = magnitude;
              frequency = i;// / 2;
            }
  }

  // Calibrate voltage
  ADC_volt_conv(voltage, cvolt);
  // Convert frequency to string
  uint16_to_string(frequency, cfreq);

  switch (mode)
  {
  case AC:
      ADC_volt_conv(vpp, cvolt2);                // calibrate peak-to-peak voltage
      ADC_volt_conv(dc_offset, cvolt3);          // calibrate DC offset
      UART_ESC_code("1:0H");                     // set cursor to line 1
      UART_ESC_code("2K");                       // clear line
      UART_print_string("Mode: AC");             // print mode
      UART_ESC_code("2:0H");                     // set cursor to line 2
      UART_ESC_code("2K");                       // clear line
      UART_print_string("Frequency: ");          // print frequency
      UART_print_string(cfreq);
      UART_ESC_code("3:0H");                     // set cursor to line 3
      UART_ESC_code("2K");                       // clear line
      UART_print_string("Voltage (RMS): ");      // print Vrms
      print_voltage(cvolt);
      UART_ESC_code("4:0H");                     // set cursor to line 4
      UART_ESC_code("2K");                       // clear line
      UART_print_string("Voltage (VPP): ");      // print Vpp
```

```
            print_voltage(cvolt2);
            UART_ESC_code("5:0H");                      // set cursor to line 4
            UART_ESC_code("2K");                        // clear line
            UART_print_string("Voltage Offset: ");  // print Vpp
            print_voltage(cvolt3);
            break;
        case DC:
            UART_ESC_code("1:0H");                      // set cursor to line 1
            UART_ESC_code("2K");                        // clear line
            UART_print_string("Mode: DC");              // print mode
            UART_ESC_code("2:0H");                      // set cursor to line 2
            UART_ESC_code("2K");                        // clear line
            UART_print_string("Voltage (AVG): ");       // print Vavg
            print_voltage(cvolt);
            UART_ESC_code("3:0H");                      // clear Vrms line
            UART_ESC_code("2K");
            UART_ESC_code("4:0H");                      // clear Vpp line
            UART_ESC_code("2K");
            UART_ESC_code("5:0H");                      // clear offset line
            UART_ESC_code("2K");
            break;
        }
        // print voltage to graph
        print_graph(cvolt);

        // reset globals
        voltage = 0;
        vpp = 0;
        dc_offset = 0;

        // wait
        for (int i = 0; i < 5000000; i++);
        SAMPLE_DONE = FALSE;
        TIM2->CR1 |= TIM_CR1_CEN;                       // re-enable clock
    }
}

void ADC1_2_IRQHandler(void) {
       if (ADC1 -> ISR & ADC_ISR_EOC) {
       //flag is reset by reading data
       static uint32_t i = 0;
       static uint32_t ADC_min = 0;
       static uint32_t ADC_max = 0;
       static uint32_t sum = 0;
       static float64_t sum2 = 0;
       if (i != SAMPLES){
                samples[i] = (float) ADC1 -> DR;
                if (samples[i] < ADC_min)           // check if less than current min
                {
                       ADC_min = samples[i];
                }
                if (samples[i] > ADC_max)           // check if greater than current max
```

```
                {
                        ADC_max = samples[i];
                }
                // add to sum for average
                sum += samples[i];

                // add to sum^2 for rms
                sum2 += (float64_t)(samples[i] * samples[i]);
                i++;
        }
        else {
                switch(mode)
                {
                case AC:
                        // compute peak-to-peak voltage
                        vpp = (ADC_max - ADC_max_calib) - (ADC_min + ADC_min_calib);
                        // compute Vrms
                        voltage = sqrt(sum2/SAMPLES);
                        dc_offset = (ADC_max + ADC_min) / 2;
                        break;
                case DC:
                        voltage = sum / SAMPLES;;
                        break;
                }
                i = 0;
                ADC_min = ADC_max;
                ADC_max = 0;
                sum = 0;
                sum2 = 0;
                SAMPLE_DONE = TRUE;
        }
      }
}


void TIM2_IRQHandler(void)
{
        // check status register for update event flag
        if (TIM2->SR & TIM_SR_UIF) {
                ADC1 -> CR |= ADC_CR_ADSTART;
                TIM2 -> SR &= ~(TIM_SR_UIF);
        }
}

void USART2_IRQHandler(void)
{
    // Check if character has been received
    if (USART2->ISR & USART_ISR_RXNE)
    {
        uint8_t received = USART2->RDR;

        switch(received)
```

```
        {
                case ('A' | 'a'):
                        mode = AC;     // set mode to AC
                        break;
                case ('D' | 'd'):
                        mode = DC;     // set mode to DC
                        break;
        }
    }
    // Reset received data flag
    USART2->ISR &= ~(USART_ISR_RXNE);

}

void print_graph(char* volt)
{
    UART_ESC_code("7:0H");
    UART_ESC_code("2K");      // clear line
    uint16_t i;
    uint16_t intvalue = 0;

    // convert char array back to int
    for (i = 0; i < 4; i++) {
        intvalue = intvalue * 10 + (volt[i] - '0');
    }
    // print to graph
    for (i = 0; i <= intvalue; i+=100){
        UART_print_char('X');
    }
    UART_ESC_code("10:0H");
}

void print_voltage(char* volt)
{
    char c = *volt;
    UART_print_char(c);                     // print first digit
    UART_print_char(0x2e);          // add decimal
    c = *(volt+1);                  // shift to first decimal
    UART_print_char(c);                     // print next digit
    c = *(volt+2);                  // shift to second decimal
    UART_print_char(c);                     // print next digit
    c = *(volt+3);                  // shift to third decimal
    UART_print_char(c);                     // print next digit
}


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

  /** Configure the main internal regulator output voltage
```

```c
  */
  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_10;
  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();
  }
}

/* USER CODE BEGIN 4 */

/* USER CODE END 4 */

/**
  * @brief  This function is executed in case of error occurrence.
  * @retval None
  */
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 */
}
```

```c
#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source line number
  *            where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
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 */
```

uart.h

```c
#ifndef INC_UART_H_
#define INC_UART_H_

#include "stm32l4xx_hal.h"

#define ESC 0x1B
#define lbracket 0x5B

void UART_init();
void UART_ESC_code(char *str);
void UART_print_string(char *str);
void UART_print_char(char c);
void uint16_to_string(uint16_t integer, char* str);


#endif /* INC_UART_H_ */
```

uart.c

```c
#include "uart.h"

#define SYS_CLK 4000000
#define BAUD 115200


void UART_init()
{
    //Enable USART timer
    RCC->APB1ENR1 |= RCC_APB1ENR1_USART2EN;

    //Enable GPIO timer
    RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;
    GPIOA->MODER &= ~(GPIO_MODER_MODE2 | GPIO_MODER_MODE3);
    GPIOA->MODER |= (GPIO_MODER_MODE2_1 | GPIO_MODER_MODE3_1);
    GPIOA->AFR[0] &= ~(GPIO_AFRL_AFRL2 | GPIO_AFRL_AFRL3);
    GPIOA->AFR[0] |= (7 << GPIO_AFRL_AFSEL2_Pos);   // Enable TX on PA2
    GPIOA->AFR[0] |= (7 << GPIO_AFRL_AFSEL3_Pos);   // Enable RX on PA3


    USART2->CR1 &= ~(USART_CR1_UE);                    // Turn off USART
    USART2->CR1 |= (USART_CR1_RE | USART_CR1_TE);   // Enable transmitter and receiver
    USART2->CR1 &= ~(USART_CR1_OVER8);                 // Set to oversample by 16
    USART2->BRR = 277;                                 // 32MHz / 115.2kbps ~ 277
    USART2->CR2 &= ~(USART_CR2_STOP);                  // Set 1 bit stop
    USART2->CR1 &= ~(USART_CR1_M1 | USART_CR1_M0);  // Set to 8 data bits
    USART2->CR1 &= ~(USART_CR1_PCE);                   // Set no parity
    USART2->CR1 |= (USART_CR1_UE);                     // Enable USART


    // Enable interrupt in NVIC
    USART2->CR1 |= USART_CR1_RXNEIE;
    NVIC->ISER[1] = (1 << (USART2_IRQn & 0x1F));


}

void UART_print_string(char *str)
{
       while (*str != '\0') {
       // Wait for the TX buffer to be empty
       while (!(USART2->ISR & USART_ISR_TXE));

       // Send the character
       USART2->TDR = (*str);

       // Wait for transmission complete
       while (!(USART2->ISR & USART_ISR_TC));
```

```c
        // Move to the next character
        str++;

        // Add a delay to ensure previous transmission is complete
        for (int i = 0; i < 1000; i++);
        }
}

void UART_print_char(char c)
{
        while (!(USART2->ISR & USART_ISR_TXE));

        // Send the character
        USART2->TDR = c;

        // Wait for transmission complete
        while (!(USART2->ISR & USART_ISR_TC));

        // Add a delay to ensure previous transmission is complete
        for (int i = 0; i < 1000; i++);
}

void UART_ESC_code(char *str)
{
         UART_print_char(ESC); // ESC character
         UART_print_char(lbracket); // [ character
         UART_print_string(str);
}


void uint16_to_string(uint16_t integer, char* str)
{
        uint16_t divisor = 10000;
        int index = 0;
        // Handle the case where the value is zero
        if (integer == 0) {
            str[index++] = '0';
            str[index] = '\0';
            return;
        }
        // Handle the case where the value is greater than zero
        while (divisor > 0) {
            uint16_t quotient = integer / divisor;
            if (quotient > 0 || index > 0) {
                str[index++] = quotient + '0';
                 integer -= quotient * divisor;
            }
            divisor /= 10;
        }
        str[index] = '\0';
}
```

adc.h

```c
#ifndef INC_ADC_H_
#define INC_ADC_H_

#include "stm32l4xx_hal.h"

void ADC_init(void);
void ADC_volt_conv(uint16_t, char*);

#define TRUE 1
#define FALSE 0

#define SAMPLES 1024

#define VREF 3300
#define RES 4096

#define CALIBRATED_MULT 1014
#define CALIBRATED_OFFSET 6684

#endif /* INC_ADC_H_ */
```

adc.c

```c
#include "adc.h"

void ADC_init()
{
    // Configure PA0 for ADC
    RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN;
    GPIOA->MODER &= ~(GPIO_MODER_MODE0);
    GPIOA->MODER |= (GPIO_MODER_MODE0);
    GPIOA->ASCR |= (GPIO_ASCR_ASC0);   // connect PA0 analog switch to ADC input

    //enable ADC Clock
    RCC->AHB2ENR |= RCC_AHB2ENR_ADCEN;
    // ADC will run at same speed as CPU
    ADC123_COMMON->CCR = (1 << ADC_CCR_CKMODE_Pos);

    //power up ADC and turn on voltage regulator
    ADC1->CR &= ~(ADC_CR_DEEPPWD);
    ADC1->CR |= (ADC_CR_ADVREGEN);
    for (uint32_t i = 0; i < 640; i++);    // wait 20 ms for regulator to power up
                                                                    // 4 Mhz -> 80
                                                                    // 32 MHZ -> 640
                                                                    // 80 Mhz -> 1600
    //calibrate ADC
    //ensure ADC is not enabled, single ended calibration
    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 finish

    //configure single ended mode for channel 5 before enabling ADC
    ADC1->DIFSEL &= ~(ADC_DIFSEL_DIFSEL_5);

    //enable ADC
    ADC1->ISR |= (ADC_ISR_ADRDY);           // clear ready flag with a 1
    ADC1->CR |= (ADC_CR_ADEN);                       // enable ADC
    while (!(ADC1->ISR & ADC_ISR_ADRDY));        // wait for ADC ready flag
    ADC1->ISR |= (ADC_ISR_ADRDY);           // clear ready flag with a 1

    //configure ADC
    //set sequence to 1 conversion on channel 5
    ADC1->SQR1 = (5 << ADC_SQR1_SQ1_Pos);

    //configure sampling time of 2.5 clock cycles for channel 5
    ADC1->SMPR1 &= ~(ADC_SMPR1_SMP5);

    //ADC configuration 12-bit software trigger
    // right align
    ADC1->CFGR = 0;

    // start conversion
```

```c
    ADC1->CR |= (ADC_CR_ADSTART);

    //enable interrupts for ADC
    ADC1->IER |= (ADC_IER_EOCIE);            // interrupt on end of conversion
    ADC1->ISR &= ~(ADC_ISR_EOC);             // clear EOC flag

    NVIC->ISER[0] = (1 << (ADC1_2_IRQn & 0x1F));    // enable interrupt in NVIC
}


void ADC_volt_conv(uint16_t dig_mvolt, char* retval)
// Calibrate millivolts and convert to a string for UART
{
    uint16_t mvolt;
    int64_t calib = (VREF * dig_mvolt) / RES;
    int64_t uvolt = CALIBRATED_MULT * calib - CALIBRATED_OFFSET;
    if (uvolt < 0) {mvolt = calib;}
    else {mvolt = uvolt / 1000;}

    uint8_t index = 0;

    while(index < 4 || mvolt != 0)      // make sure value is 4 digits
    {
        uint16_t digit = mvolt % 10;    // isolate digit
        retval[index] = digit + '0';    // '0' converts to ASCII character
        mvolt /= 10;
        index++;
    }

    // Reverse the string
    uint8_t i = 0, j = index - 1;
    while (i < j)
    {
       char temp = retval[i];
       retval[i] = retval[j];
       retval[j] = temp;
       i++;
       j--;
    }
}
```