Guitar Tuner

Diego Renato Curiel

1 June 2024

Behavior Description

The device is a standard six string guitar tuner. The STM32 microcontroller is running on FreeRTOS and is configured to take 2048 samples at 2048 Hz, achieving a frequency resolution of 1 Hz up to 1024 kHz. To minimize power consumption, FreeRTOS is configured in tickless mode with a wait-to-sleep time of 50 ms. The user can communicate with the device through USART on a terminal.

The device starts in sleep mode. From here, the user can press the on-board blue button to wake it up, at which point instructions for use are displayed on the terminal. The only commands a user can make are inputting which note the user would like to tune to. Once the user wakes the board up and inputs the desired note, the LEDs become the user's reference point for whether or not they are in tune. The LED's on the breadboard determine how close you are to the desired frequency. This threshold allows for +/- 2 Hz on each side; after crossing these boundaries, each LED corresponds to +/- 4 Hz above or below the desired note's frequency. Once the user is done tuning, the on-board blue button can be pressed to put the board to sleep, thus minimizing the power consumption.

The analog input comes from a microphone on a breadboard. This signal is then passed through a signal conditioning circuit, which consists of an amplifier and a fifth order Butterworth low-pass filter with a 3dB cutoff at 600 Hz to minimize unwanted frequencies coming through. This circuit is powered by the microcontroller's 3.3V supply, as are the LEDs that serve as a visual representation of whether or not the user is in tune.

System Specifications

Specification	
Supply	3.3V
Power consumption when asleep	11.06 mW
Power consumption when awake	119.39 mW
Power savings (compared to no low power mode, only when asleep, power consumption when awake is the same)	45.1%

Table 1: Power Specifications
**power calculations are further down, this is a summarization of results*

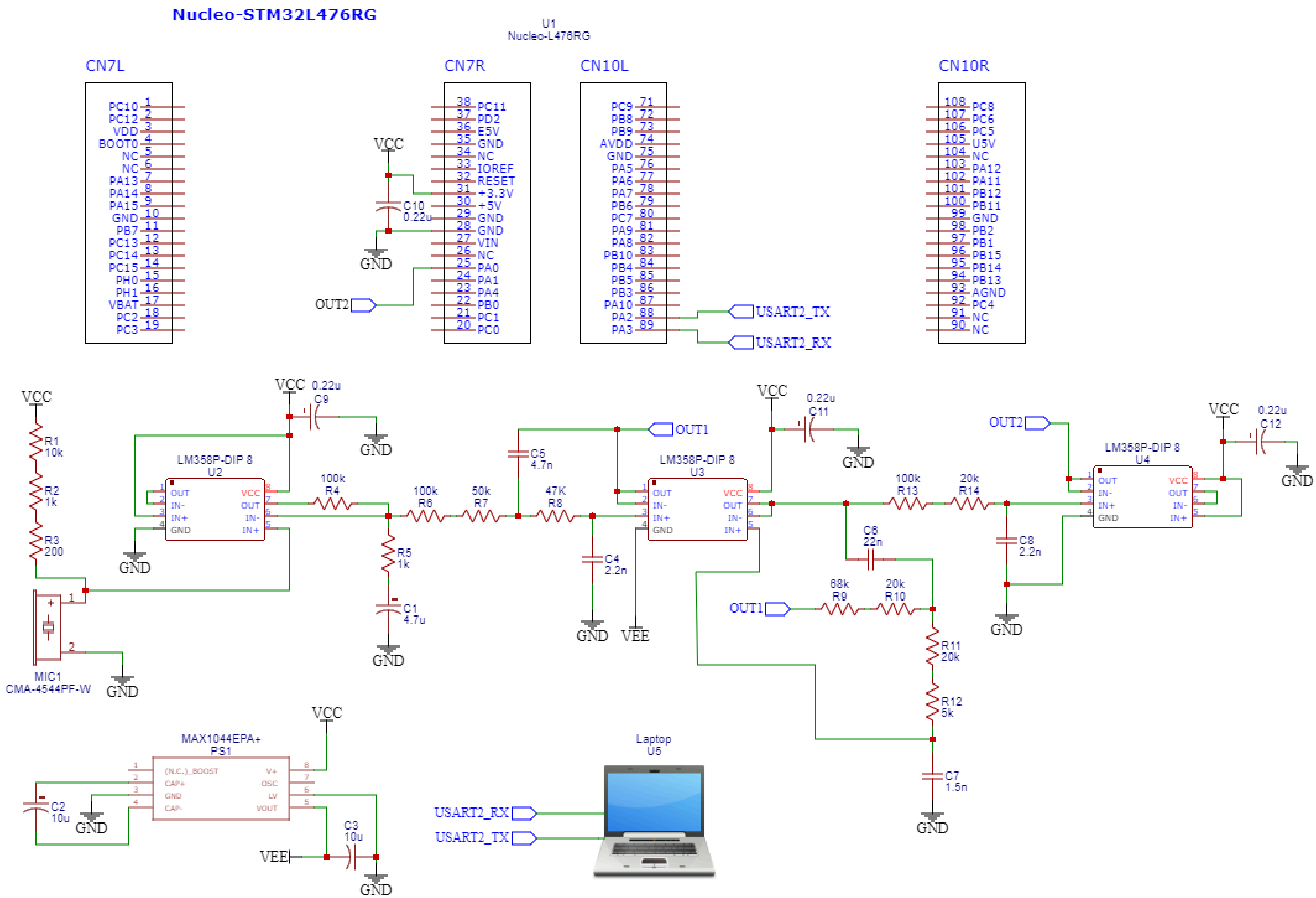
Analog Signal Conditioning Circuitry Specifications	
Supply Voltage	3.3V
Vdd (Dual supply for active filter using op-amp)	-3.3V
Microphone Range	20Hz – 20 kHz
Microphone Signal Bias	1.1V
Gain from Amplifier	~101 mV/mV
Low-Pass Filter Order	5
Low-Pass Filter Topology	Cascaded Sallen-Key
Low-Pass Filter 3dB Cutoff	600Hz

Table 2: Analog Circuitry Specifications

MCU Frequency Detection Program Specification (FFT)	
MCU Clock Speed	32MHz
ADC Resolution	12 bit
Sampling Frequency	2048 Hz
FFT Buffer Size (# of Samples)	2048
Frequency Resolution (Fs/N)	1 Hz
Maximum Frequency Detectable	1024 Hz
Time Between Frequency Calculations	1 s

Table 3: Frequency Detection System Specifications

Schematic



Nucleo-STM32L476RG
U1
Nucleo-L476RG
CN7L
CN7R
CN10L
CN10R
PC10 PC12 VDD BOOT0 NC NC PA13 PA14 PA15 GND PB7 PC13 PC14 PC15 PH0 PH1 VBAT PC2 PC3
PC11 PD2 E5V GND NC IOREF RESET +3.3V +5V GND GND VIN NC PA0 PA1 PA4 PB0 PC1 PC0
PC9 PB8 PB9 AVDD GND PA5 PA6 PA7 PB6 PC7 PA9 PA8 PB10 PB4 PB5 PB3 PA10 PA2 PA3
PC8 PC6 PC5 U5V NC PA12 PA11 PB12 PB11 GND PB2 PB1 PB15 PB14 PB13 AGND PC4 NC NC
USART2_TX
USART2_RX
OUT2
VCC
GND
C10 0.22u
C9 0.22u
C11 0.22u
C12 0.22u
LM358P-DIP 8 U2
LM358P-DIP 8 U3
LM358P-DIP 8 U4
OUT1
R1 10k
R2 1k
R3 200
R4 100k
R5 1k
R6 100k
R7 50k
R8 47K
R9 68k
R10 20k
R11 20k
R12 5k
R13 100k
R14 20k
C1 4.7u
C2 10u
C3 10u
C4 2.2n
C5 4.7n
C6 22n
C7 1.5n
C8 2.2n
MIC1 CMA-4544PF-W
VEE
MAX1044EPA+ PS1
(N.C.)_BOOST CAP+ GND CAP- V+ OSC LV VOUT
Laptop U5

On the schematic:

Above is a detailed hardware schematic of the system. It is powered solely from the microcontroller's 3.3V supply. It begins with a microphone (MIC1) capturing the audio signals. Only one operational amplifier, U2 (LM358P-DIP 8), is used in the amplification stage to condition the signal. The signal is then processed through a cascaded Sallen-Key low pass filter, comprising U3 and U4, also operational amplifiers. This filter has a cutoff frequency of 600Hz, designed to isolate the desired frequency range. Additionally, a charge pump IC (MAX1044EPA+) is incorporated, supplying the negative voltage necessary for the dual supply of the operational amplifiers. This configuration caters specifically to the frequencies relevant for guitar tuning.

OUT1 refers to the output of the first stage of filtering. From OUT1, we see the second and third stages of filtering, which produces OUT2, the final analog signal that is sent to the STM32 microcontroller for processing. The USART2_TX and USART3_RX data lines are routed to the machine running the terminal application that processes the frequency output by the microcontroller.

To save space and to make the schematic cleaner, the LEDs were left out: PC0-PC10 are used, though the user can configure any GPIO pins they would like for the LED outputs, so long as the proper modifications to the code are made. I decided to use the same bank for all of them so that setting one of the LEDs high would be simply done by using a mask.

All circuitry was verified in simulation. The component values used in the system were selected to come as close to the exact values determined when doing filter component calculations. Below are the schematics and simulation results.

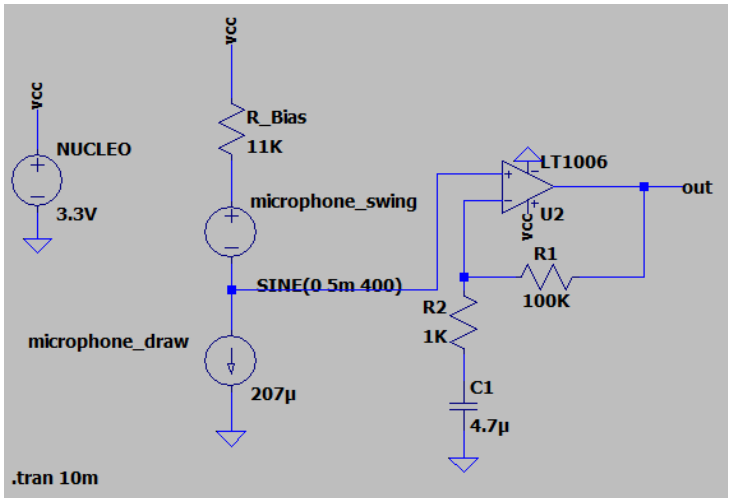
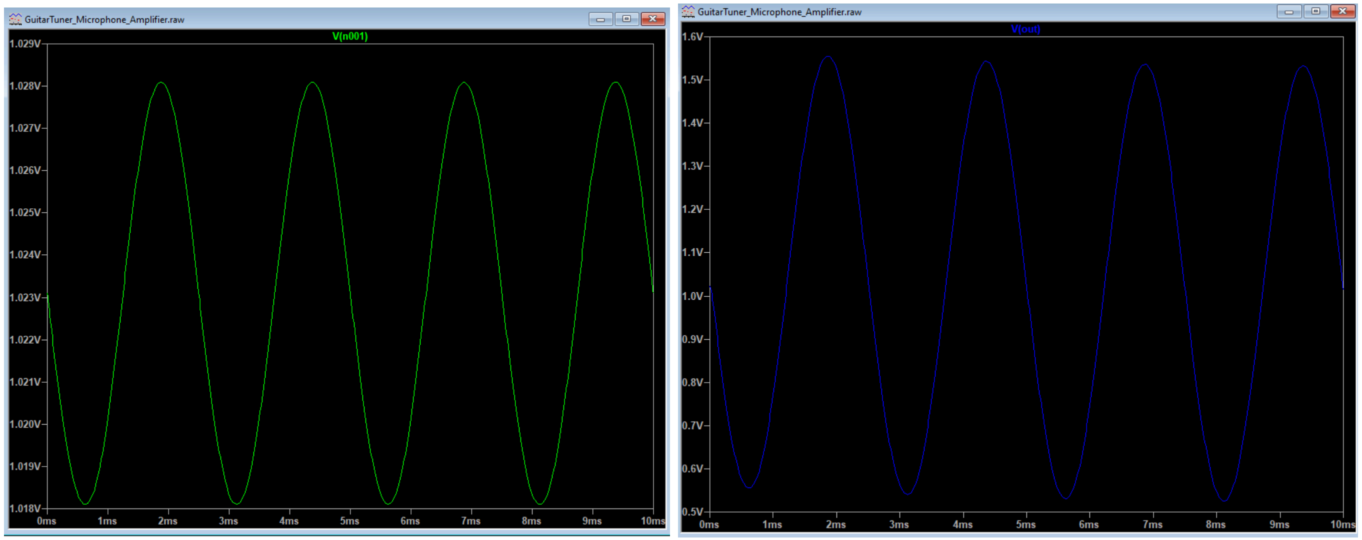


Figure 1: LTSpice Schematic of Microphone/Amplification



Figures 2 and 3: Simulation Results (left 10mVpp, right 1Vpp)

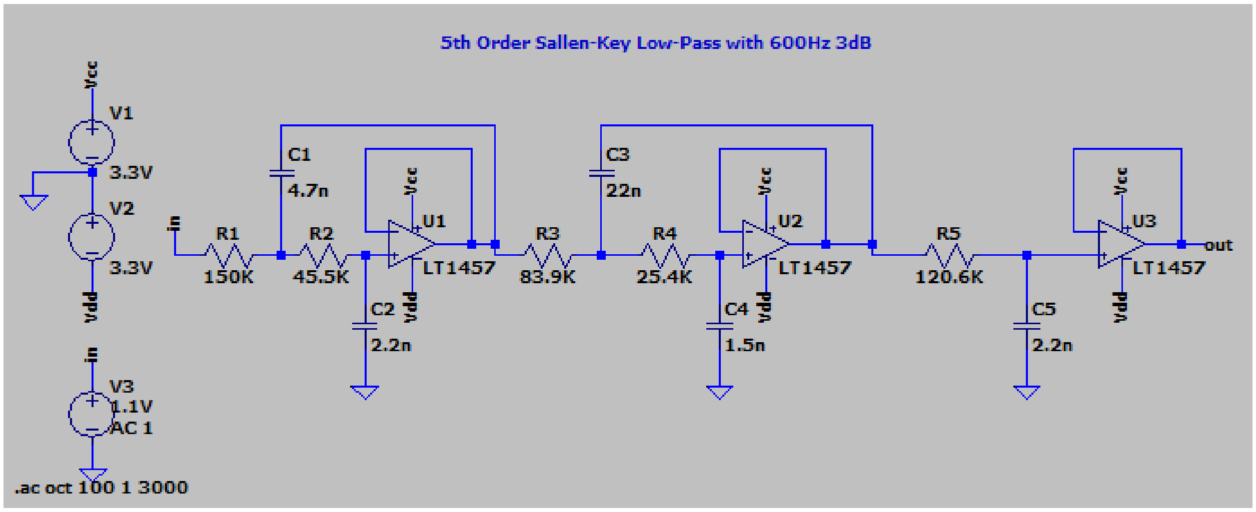


Figure 4: LTSpice Schematic of Low-Pass Filter

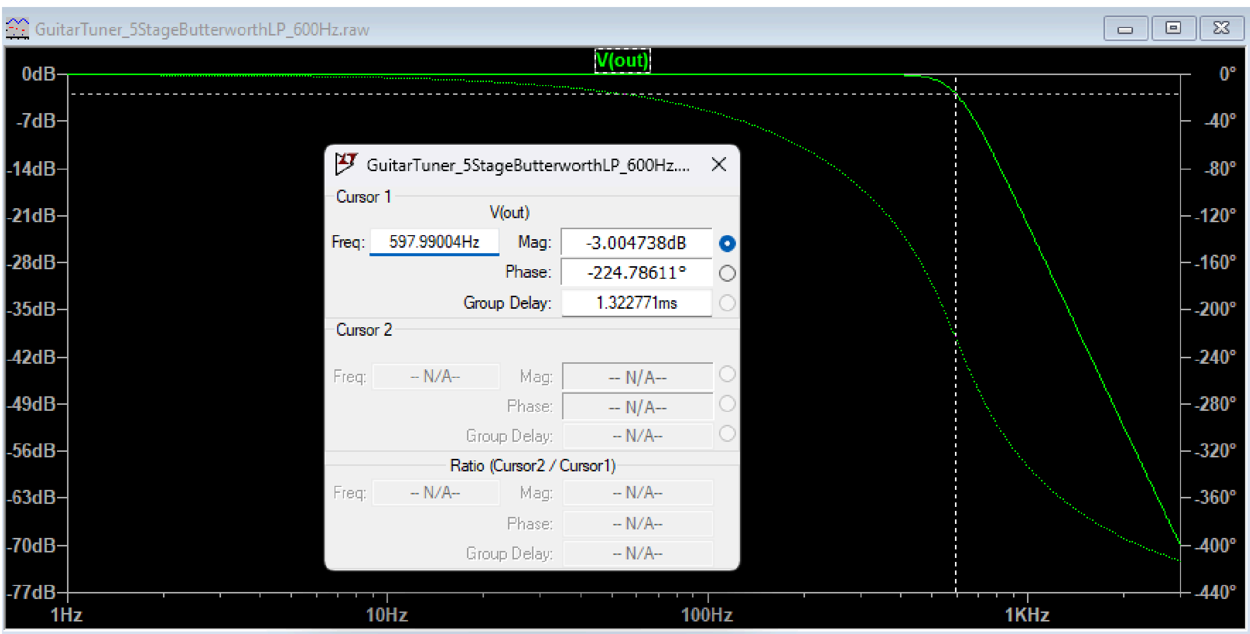


Figure 5: Simulated Frequency Response (3dB cutoff @ 600Hz)

Figures 1-5 all display the process of creating the circuitry for the amplifier and filter. It was Simulated and tested in LTSpice before all the components were ordered and the circuit constructed.

Software Architecture

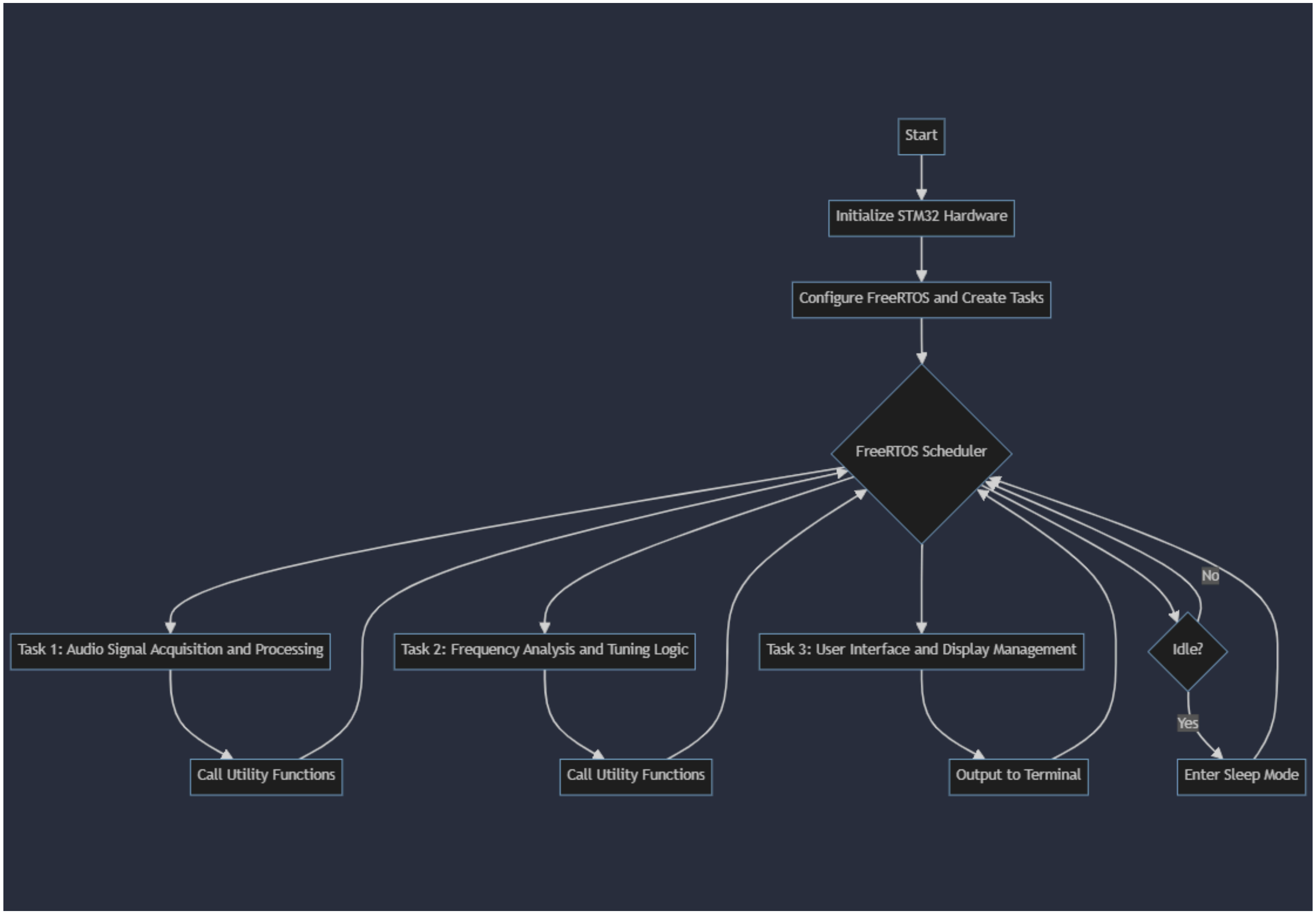


Figure 6: High Level Software Flowchart

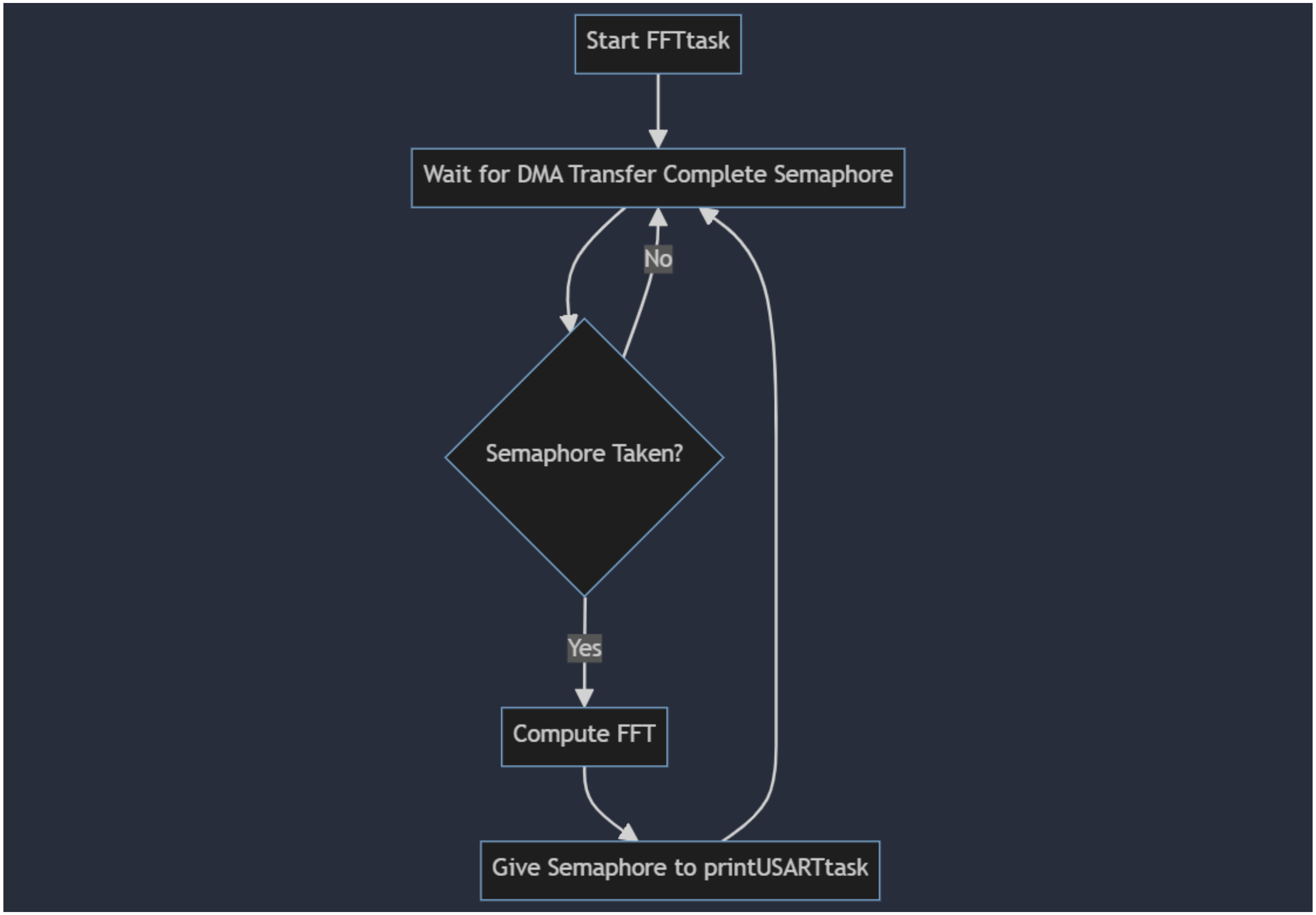


Figure 7: FFT Task Flowchart

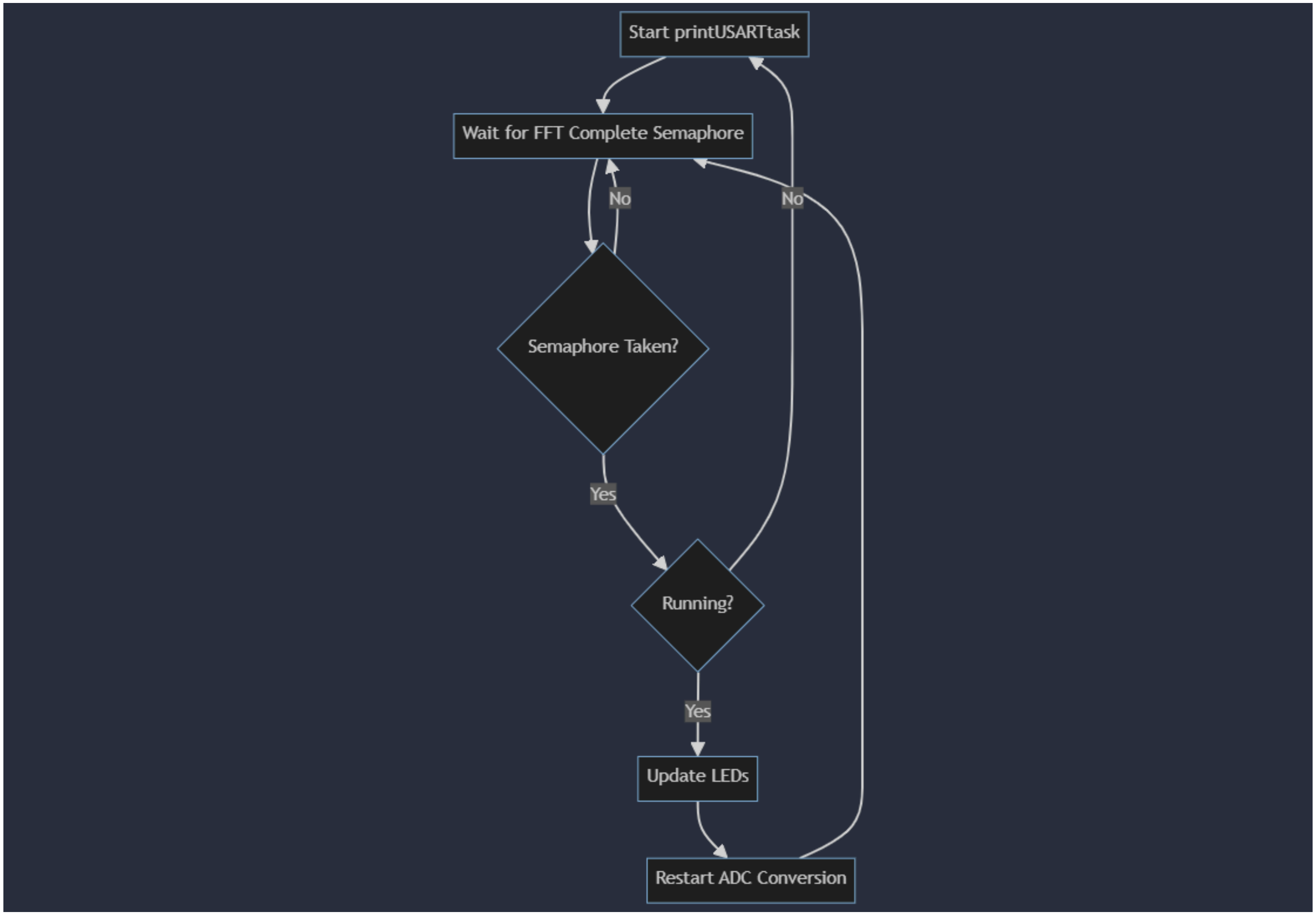


Figure 8: USART Task Flowchart

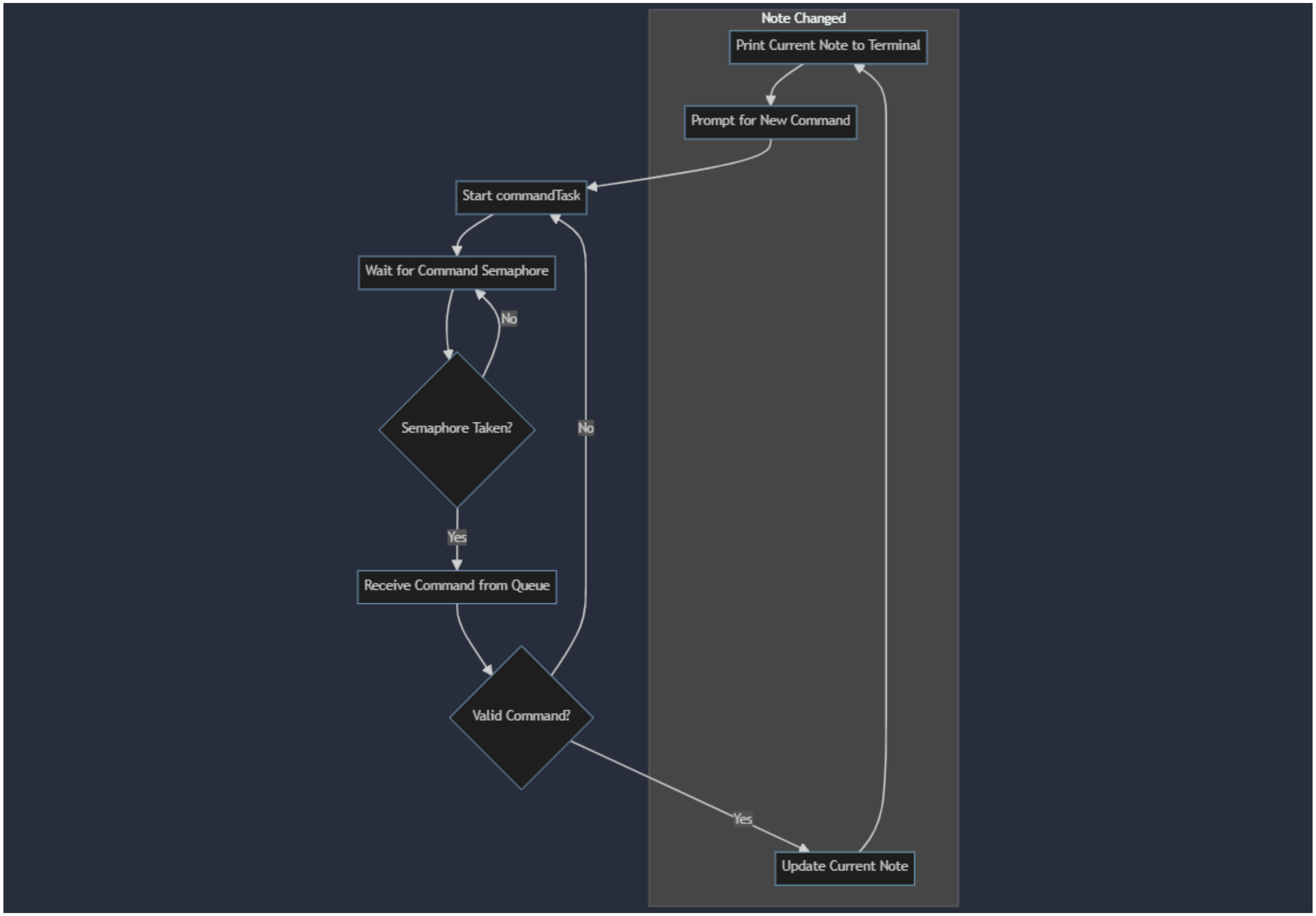


Figure 9: Command Task Flowchart

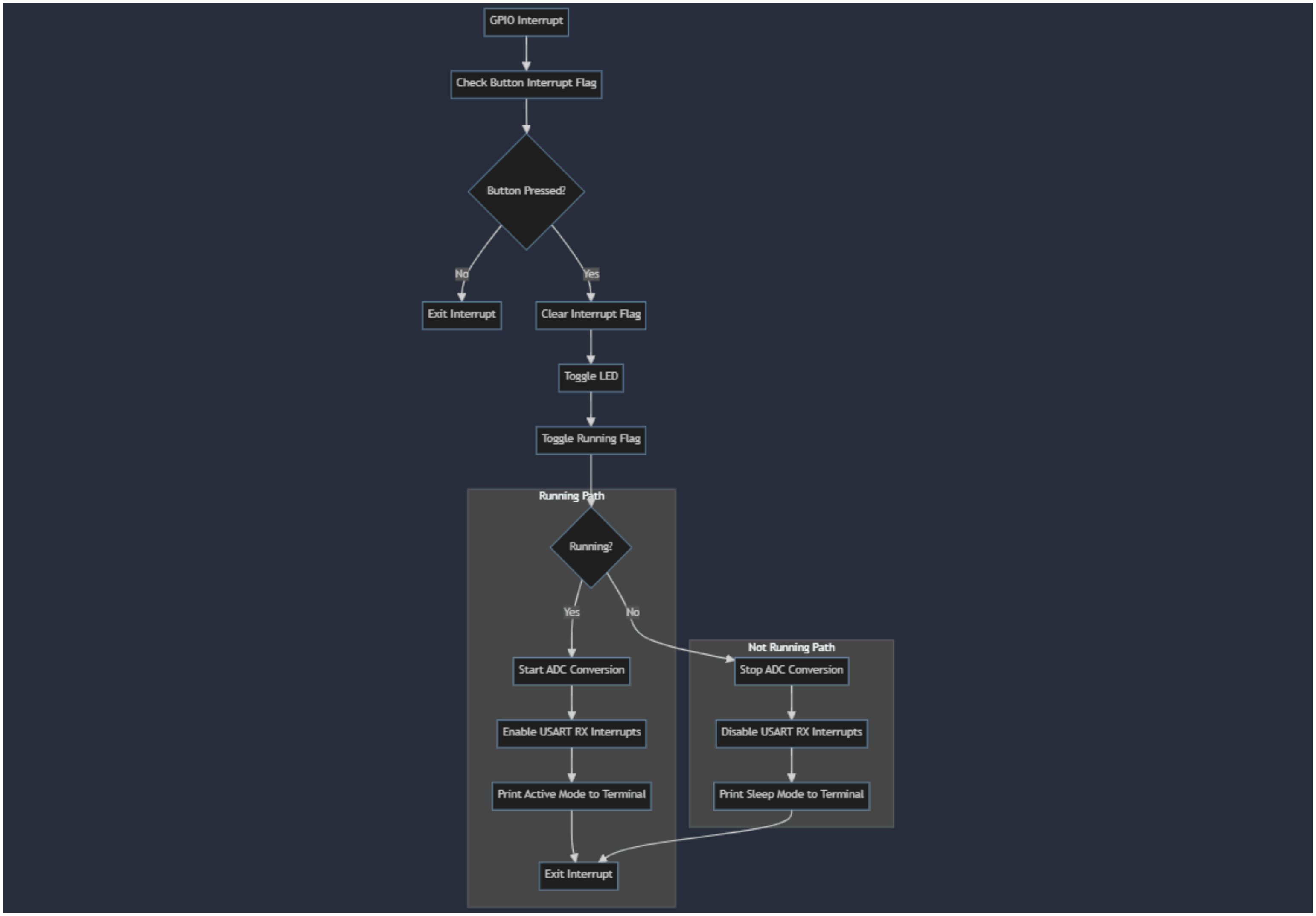


Figure 10: GPIO Interrupt Flowchart

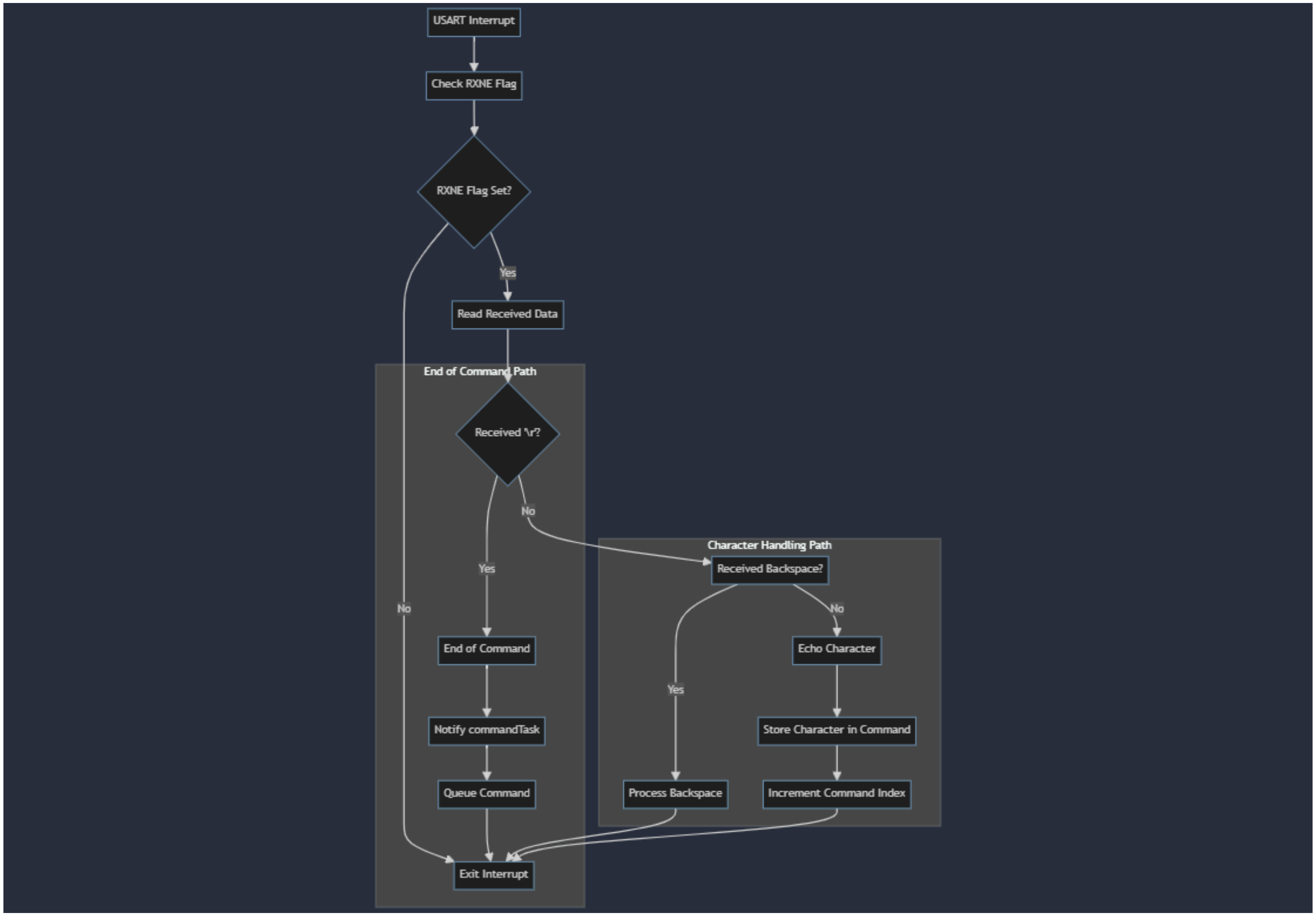
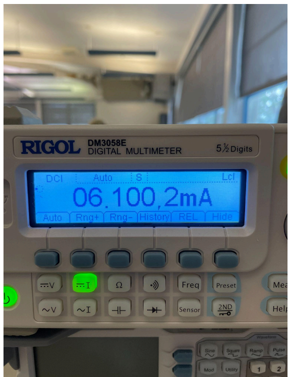
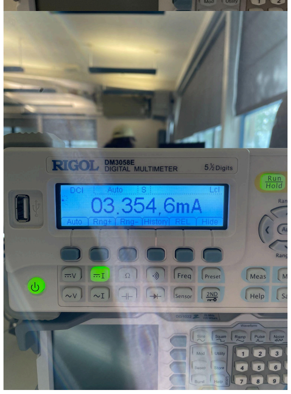


Figure 11: USART Interrupt Flowchart

Power Calculations





These two pictures show the current consumption of the microcontroller when idling. In the top photo, tickless mode in FreeRTOS is not enabled. In the bottom one, tickless mode is enabled. As is shown in the photos, current consumption almost halves when in sleep mode as compared to just idling. Calculations for power consumption are as follows:

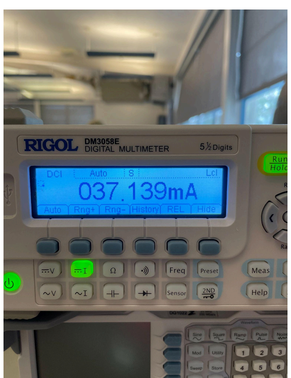
$Power = V * I$

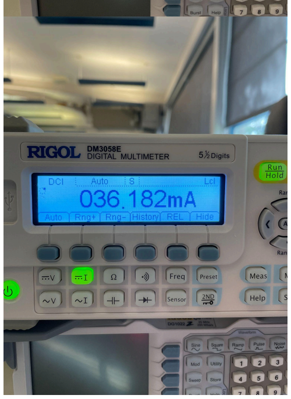
Idle:

$Power = 3.3V * 6.1mA = 20.13\, mW$

Low Power:

$Power = 3.3V * 3.35mA = 11.06\, mW$





$Power = V * I$

Idle:

$Power = 3.3V * 37.14mA = 122.56\, mW$

Low Power:

$Power = 3.3V * 36.18mA = 119.39\, mW$

It's interesting to note that in tickless mode, regular operation of the microcontroller consumes about one mA less current. Total power savings are around 45%, with most of them coming from when the microcontroller is idling.

User Manual

Setup and Configuration:

1. Make proper connections from circuit to microcontroller.
2. Make sure LEDs are visible and correctly configured.

Usage:

1. Power on and Init:
    a. Connect the board to the PC via USB.
    b. Open a terminal application (I use PuTTY) and connect to the correct COM port with a baud rate of 460800.
    c. Wake the board up by pressing the on-board blue button.
2. Tuning mode:
    a. Type the desired note to tune to in the terminal and press enter.
    b. Tune the guitar, watch the LEDs! Once in tune, the green LED will hang high for five seconds.
    c. Switch notes at any time by repeating step 2A.
3. Sleep mode:
    a. Deactivate the tuning mode by pressing the blue button again. The terminal will display “Sleeping…”
    b. Reactivate the tuning mode by pressing the blue button again and following the steps in 2.

Code

main.h:

```
/* Define to prevent recursive inclusion
--------------------------------------*/
#ifndef __MAIN_H
#define __MAIN_H
#ifdef __cplusplus
extern "C" {
#endif
/* Includes
------------------------------------------------------------------*/
#include "stm32l4xx_hal.h"
/* Private includes
----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "arm_math.h"
//#include "init.h"
//#include "utils.h"
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <string.h>
#include <float.h>
#include <math.h>
#include <complex.h>
/* USER CODE END Includes */
/* Exported types
------------------------------------------------------------*/
/* USER CODE BEGIN ET */
typedef struct {
      char command[50];
      uint8_t len;
} command_t;
/* USER CODE END ET */
/* Exported constants
--------------------------------------------------------*/
/* USER CODE BEGIN EC */
/* USER CODE END EC */
/* Exported macro
------------------------------------------------------------*/
```

```
/* USER CODE BEGIN EM */
/* USER CODE END EM */
/* Exported functions prototypes
---------------------------------------------*/
void Error_Handler(void);
/* USER CODE BEGIN EFP */
/* USER CODE END EFP */
/* Private defines
-----------------------------------------------------------*/
/* USER CODE BEGIN Private defines */
#define CLK_FREQUENCY 32000000 // system clock
#define SAMPLING_FREQUENCY 2048 // sampling frequency
#define NUM_SAMPLES 2048 // number of samples
#define DELTA_F (float)SAMPLING_FREQUENCY/(float)NUM_SAMPLES // frequency
resolution
#define BAUD_RATE 460800 // baud rate for USART
#define GPIOC_PIN_MASK 0x07FF // mask for pins PC0 to PC10
extern volatile unsigned long ulHighFrequencyTimerTicks; // used for task
runtime stats
extern q15_t digitalSampleArray[NUM_SAMPLES]; // array to hold the samples
from ADC
extern q15_t FFTBuffer[NUM_SAMPLES * 2]; // array to hold values of FFT
extern float maxFrequency ; // to hold maxFrequency for printing
extern float desiredFrequency; // to hold current desired frequency
extern uint16_t dcOffset; // to hold offset for printing
extern arm_rfft_instance_q15 S; // FFT structure
extern uint8_t running; // flag to determine "on/off"
extern char command[50]; // to hold global command
extern uint8_t commandIndex; // for indexing into global command
extern char currentNote[50]; // for the current note to tune to
/* USER CODE END Private defines */
#ifdef __cplusplus
}
#endif
#endif /* __MAIN_H */
```

main.c:

```
/*
****************************************************************************
***
 * @file           : main.c
 * author                : Diego Renato Curiel
 * date created    : 5/29/2024
 * last modified   :
****************************************************************************
***
 * Guitar Tuner description
 * Simple six string, standard guitar tuner. On board peripherals used
include the
 * DMA, ADC, USART2, TIM2, TIM5, and a ton of GPIO pins for LED outputs.
 * The program is written utlizing FreeRTOS in tickless mode, to enable a
 * lower power consumption when the user isn't actively tuning. The way to
 * enter sleep and exit sleep is through the on board blue button. The low
 * power mode utilized is sleep.
****************************************************************************
***
*/
#include "main.h"
#include "cmsis_os.h"
#include "task.h"
#include "semphr.h"
#include "queue.h"
#include "init.h"
#include "utils.h"
SemaphoreHandle_t dmaTransferCompleteSemaphore, fftCompleteSemaphore,
commandSemaphore; // semaphores for tasks
TaskHandle_t FFTHandler, printUSARTHandler, commandHandler; // create
handler names
QueueHandle_t CommandQueue; // queue for passing commands safely from USART
interrupt
void DMA1_Channel1_IRQHandler(void);
void EXTI15_10_IRQHandler(void);
void TIM5_IRQHandler(void);
void USART2_IRQHandler(void);
```

```
void FFTtask(void *argument);
void printUSARTtask(void *argument);
void commandTask(void *argument);
volatile unsigned long ulHighFrequencyTimerTicks; // used for task runtime
stats
q15_t digitalSampleArray[NUM_SAMPLES] = {0}; // array to hold the samples
from ADC
q15_t FFTBuffer[NUM_SAMPLES * 2] = {0}; // array to hold values of FFT
float maxFrequency = 0.0f; // to hold maxFrequency for printing
float desiredFrequency = 0.0f; // to hold current desired frequency
uint16_t dcOffset = 0; // to hold offset for printing
arm_rfft_instance_q15 S; // FFT structure
uint8_t running = 0; // flag to determine "on/off"
uint8_t high = 0; // user input to determine tuning below 200Hz
char command[50] = {0}; // to hold global command
uint8_t commandIndex = 0; // for indexing into global command
char currentNote[50] = {0}; // for the current note to tune to
int main(void)
{
      // INITIALIZE ALL PERIPHERALS
      HAL_Init();
      SystemClock_Config();
      configureTimerForRunTimeStats();
      BankA_Init();
      BankC_Init();
      TIM2_Init();
      USART_Init();
      USART_ESC_Code("[2J"); //clear terminal
      USART_ESC_Code("[H"); //back to top left
      USART_ESC_Code("[?25l"); //hide cursor
      USART_ESC_Code("[37m"); //white text
      //Initialize FFT configuration
      arm_rfft_init_q15(&S, NUM_SAMPLES, 0, 1);
      //Ensure all LEDs are off
      GPIOA->ODR &= ~GPIO_ODR_OD5;
      GPIOC->ODR &= ~(GPIOC_PIN_MASK);
   // Create semaphores
   dmaTransferCompleteSemaphore = xSemaphoreCreateBinary();
   fftCompleteSemaphore = xSemaphoreCreateBinary();
   commandSemaphore = xSemaphoreCreateBinary();
   if (dmaTransferCompleteSemaphore == NULL || fftCompleteSemaphore == NULL
|| commandSemaphore == NULL) {while(1);}
   // Create Command Queue
```

```c
   CommandQueue = xQueueCreate(2, sizeof(command_t));
   if (CommandQueue == NULL) { while(1); }
      // Create tasks
      BaseType_t retVal;
      retVal = xTaskCreate(FFTtask, "FFTtask", configMINIMAL_STACK_SIZE *
4, NULL, tskIDLE_PRIORITY + 6, &FFTHandler);
      if (retVal != pdPASS) { while(1); } // check if task creation failed
      retVal = xTaskCreate(printUSARTtask, "printUSARTtask",
configMINIMAL_STACK_SIZE * 4, NULL, tskIDLE_PRIORITY + 6,
&printUSARTHandler);
      if (retVal != pdPASS) { while(1); } // check if task creation failed
      retVal = xTaskCreate(commandTask, "commandTask",
configMINIMAL_STACK_SIZE * 4, NULL, tskIDLE_PRIORITY + 6, &commandHandler);
      if (retVal != pdPASS) { while(1); } // check if task creation failed
      // Notify user that board is starting in sleep mode
      USART_Print("Sleeping...");
      // START ADC/DMA LAST TO ENSURE ALL RTOS CONFIG COMPLETES BEFORE DMA
INTERRUPT HITS
      DMA_WithADC_Init();
      ADC_Init();
      // Start scheduler
      vTaskStartScheduler();
      printf("Shouldn't be here!\n");
      while (1) {}
}
/* Define Tasks
---------------------------------------------------------------*/
void FFTtask(void *argument)
{
      for(;;)
      {
            if (xSemaphoreTake(dmaTransferCompleteSemaphore, portMAX_DELAY)
== pdTRUE) // take semaphore
            {
                  // compute the FFT!
                  computeFFT();
                  // give semaphore to calculate bins task
                  xSemaphoreGive(fftCompleteSemaphore);
            }
      }
}
void printUSARTtask(void *argument)
{
```

```
	for(;;)
	{
		if (xSemaphoreTake(fftCompleteSemaphore, portMAX_DELAY) ==
pdTRUE)
		{
			if (running)
			{
				// update LEDS
				desiredFrequency = getFrequencyFromNote((const
char*) currentNote);
				updateLEDs(maxFrequency, desiredFrequency);
				// restart ADC for next conversion/FFT!
				ADC1->CR |= ADC_CR_ADSTART;
			}
		}
	}
}
void commandTask(void *argument)
{
	for(;;)
	{
		if (xSemaphoreTake(commandSemaphore, portMAX_DELAY) == pdTRUE)
		{
			command_t cmd;
			if (xQueueReceive(CommandQueue, &cmd, 0) == pdTRUE)
			{
				// parse and execute user command!
				if(cmd.len < 50)
				{
					memcpy(currentNote, cmd.command, cmd.len);
					currentNote[cmd.len] = '\0';
				}
				char buffer[100] = {0};
				snprintf(buffer, sizeof(buffer), "Currently tuning
to note %s\r\n", currentNote);
				USART_Print(buffer);
				USART_Print("$: ");
			}
		}
	}
}
/* INTERRUPT
HANDLERS---------------------------------------------------------------*/
```

```
/* Interrupt Handler for DMA */
void DMA1_Channel1_IRQHandler(void)
{
      if (DMA1->ISR & DMA_ISR_TCIF1) // check transfer complete intr flag
      {
            BaseType_t xHigherPriorityTaskWoken = pdFALSE;
            ADC1->CR |= ADC_CR_ADSTP; // stop conversions
            DMA1->IFCR |= DMA_IFCR_CTCIF1; // clear transfer complete intr
flag
            // Give FFT task semaphore
            xSemaphoreGiveFromISR(dmaTransferCompleteSemaphore,
&xHigherPriorityTaskWoken);
            portYIELD_FROM_ISR(&xHigherPriorityTaskWoken); // yield to next
highest priority task (fft)
      }
}
/* Interrupt Handler for Button Press */
void EXTI15_10_IRQHandler(void)
{
   if (EXTI->PR1 & EXTI_PR1_PIF13) // check button interrupt flag
   {
       BaseType_t xHigherPriorityTaskWoken = pdFALSE;
       EXTI->PR1 = EXTI_PR1_PIF13; // Clear interrupt flag
       GPIOA->ODR ^= GPIO_ODR_OD5; // Toggle LED
       GPIOC->ODR &= ~(GPIOC_PIN_MASK); // Turn off scale LEDs
       running = !running; // toggle global running flag (turn on or off)
       if (running) // if just turned on, start conversion and turn on
USART RX interrupts
       {
                  USART_ESC_Code("[2J"); //clear terminal
                  USART_ESC_Code("[H"); //back to top left
            printHeader();
            USART_Print("$: ");
            ADC1->CR |= ADC_CR_ADSTART;
            USART2->CR1 |= (USART_CR1_RE);      // enable receive for
USART2
            USART2->CR1 |= USART_CR1_RXNEIE;                   // enable
RXNE interrupt on USART2
            USART2->ISR &= ~(USART_ISR_RXNE);                  // clear
interrupt flag
       }
       else // if just turned off, turn off USART RX interrupts so board
can only be woken by button
```

```
        {
                    USART_ESC_Code("[2J"); //clear terminal
                    USART_ESC_Code("[H"); //back to top left
                    USART_Print("Sleeping...");
              USART2->CR1 &= ~USART_CR1_RE;
              // disable receive for USART2
              USART2->CR1 &= ~USART_CR1_RXNEIE;                   // disable
RXNE interrupt on USART2
              USART2->ISR &= ~(USART_ISR_RXNE);                  // clear
interrupt flag
              commandIndex = 0; // reset global index when turned off
        }
        portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
    }
}
void USART2_IRQHandler(void) {
   if (USART2->ISR & USART_ISR_RXNE)
   {
      BaseType_t xHigherPriorityTaskWoken = pdFALSE;
      if (USART2->RDR == '\r')
      {
                  USART_Print("\r\n");
                  command_t cmd_t;
                  memcpy(cmd_t.command, command, commandIndex);
                  cmd_t.command[commandIndex] = '\0';
                  cmd_t.len = commandIndex;
                  xSemaphoreGiveFromISR(commandSemaphore,
&xHigherPriorityTaskWoken);
                  xQueueSendFromISR(CommandQueue, &cmd_t,
&xHigherPriorityTaskWoken);
                  commandIndex = 0; //reset message index
            }
            else if (USART2->RDR == 0x7F)// backspace
            {
                  USART2->TDR = USART2->RDR; // echo
                  if (commandIndex > 0)
                  {
                        commandIndex--;
                  }
            }
      else
      {
            USART2->TDR = USART2->RDR; // copy received char to transmit
```

```
buffer to echo
            command[commandIndex] = USART2->RDR; //put char into string
            commandIndex++; // increment current index
      }
       portYIELD_FROM_ISR(xHigherPriorityTaskWoken);
   }
}
/* Timer 5 is used to collect runtime stats for FreeRTOS tasks*/
void TIM5_IRQHandler(void)
{
      TIM5->SR &= ~(TIM_SR_UIF);
      ulHighFrequencyTimerTicks++;
}
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_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_0) != 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 */
}
```

init.h:

```c
/*
* init.h
*
*  Created on: May 30, 2024
*      Author: thede
*/
#ifndef INC_INIT_H_
#define INC_INIT_H_
void SystemClock_Config(void);
void configureTimerForRunTimeStats(void);
void RTOS_Stats_Timer_Init(void);
void BankA_Init(void);
void BankC_Init(void);
void LEDinit(void);
void TIM2_Init(void);
void DMA_WithADC_Init(void);
void ADC_Init(void);
void USART_Init(void);
#endif /* INC_INIT_H_ */
```

init.c:

```c
/*
* init.c
*
*  Created on: May 30, 2024
*      Author: thede
*/
#include "init.h"
#include "main.h"
void BankA_Init(void)
{
      RCC->AHB2ENR = RCC_AHB2ENR_GPIOAEN; // enable bank A clock
   // Configure PA5 as output (LED)
   GPIOA->MODER &= ~GPIO_MODER_MODE5_Msk;
   GPIOA->MODER |= GPIO_MODER_MODE5_0;
      // Set PA1 to alternate function
      GPIOA->MODER &= ~GPIO_MODER_MODE1;
      GPIOA->MODER |= GPIO_MODER_MODE1_1;
      // Set alternate function type for PA1 (AF1 for TIM2_CH2)
      GPIOA->AFR[0] &= ~GPIO_AFRL_AFSEL1_Msk; //clear bits
      GPIOA->AFR[0] |= GPIO_AFRL_AFSEL1_0; //set bits
      // High speed, no pull up or pull down
      GPIOA->OSPEEDR |= GPIO_OSPEEDR_OSPEED1_1;
      GPIOA->PUPDR &= ~GPIO_PUPDR_PUPD1_Msk;
}
void BankC_Init(void)
{
   // Enable GPIOC peripheral clock
   RCC->AHB2ENR |= RCC_AHB2ENR_GPIOCEN;
   // Init LEDs
   LEDinit();
   // Configure PC13 as input (Button)
   GPIOC->MODER &= ~GPIO_MODER_MODE13;
   // Enable SYSCFG clock
   RCC->APB2ENR |= RCC_APB2ENR_SYSCFGEN;
   // Configure EXTI13 line for PC13
   SYSCFG->EXTICR[3] &= ~SYSCFG_EXTICR4_EXTI13_Msk;
   SYSCFG->EXTICR[3] |= SYSCFG_EXTICR4_EXTI13_PC;
   // Enable EXTI13 interrupt
   EXTI->IMR1 |= EXTI_IMR1_IM13;
   EXTI->RTSR1 |= EXTI_RTSR1_RT13;
```

```c
    // Configure NVIC for EXTI15_10
    NVIC_SetPriority(EXTI15_10_IRQn, 5);
    NVIC_EnableIRQ(EXTI15_10_IRQn);
}
void LEDinit(void)
{
    // Set PC0 to PC10 as outputs
    GPIOC->MODER &= ~(
        GPIO_MODER_MODE0_Msk |
        GPIO_MODER_MODE1_Msk |
        GPIO_MODER_MODE2_Msk |
        GPIO_MODER_MODE3_Msk |
        GPIO_MODER_MODE4_Msk |
        GPIO_MODER_MODE5_Msk |
        GPIO_MODER_MODE6_Msk |
        GPIO_MODER_MODE7_Msk |
        GPIO_MODER_MODE8_Msk |
        GPIO_MODER_MODE9_Msk |
        GPIO_MODER_MODE10_Msk
    );
    GPIOC->MODER |= (
        GPIO_MODER_MODE0_0 |
        GPIO_MODER_MODE1_0 |
        GPIO_MODER_MODE2_0 |
        GPIO_MODER_MODE3_0 |
        GPIO_MODER_MODE4_0 |
        GPIO_MODER_MODE5_0 |
        GPIO_MODER_MODE6_0 |
        GPIO_MODER_MODE7_0 |
        GPIO_MODER_MODE8_0 |
        GPIO_MODER_MODE9_0 |
        GPIO_MODER_MODE10_0
    );
    // Set output type to push-pull (default state)
    GPIOC->OTYPER &= ~(
        GPIO_OTYPER_OT0 |
        GPIO_OTYPER_OT1 |
        GPIO_OTYPER_OT2 |
        GPIO_OTYPER_OT3 |
        GPIO_OTYPER_OT4 |
        GPIO_OTYPER_OT5 |
        GPIO_OTYPER_OT6 |
        GPIO_OTYPER_OT7 |
```

```
        GPIO_OTYPER_OT8 |
        GPIO_OTYPER_OT9 |
        GPIO_OTYPER_OT10
    );
    // Set output speed to high
    GPIOC->OSPEEDR |= (
        GPIO_OSPEEDR_OSPEED0_1 |
        GPIO_OSPEEDR_OSPEED1_1 |
        GPIO_OSPEEDR_OSPEED2_1 |
        GPIO_OSPEEDR_OSPEED3_1 |
        GPIO_OSPEEDR_OSPEED4_1 |
        GPIO_OSPEEDR_OSPEED5_1 |
        GPIO_OSPEEDR_OSPEED6_1 |
        GPIO_OSPEEDR_OSPEED7_1 |
        GPIO_OSPEEDR_OSPEED8_1 |
        GPIO_OSPEEDR_OSPEED9_1 |
        GPIO_OSPEEDR_OSPEED10_1
    );
    // Set no pull-up/pull-down
    GPIOC->PUPDR &= ~(
        GPIO_PUPDR_PUPD0_Msk |
        GPIO_PUPDR_PUPD1_Msk |
        GPIO_PUPDR_PUPD2_Msk |
        GPIO_PUPDR_PUPD3_Msk |
        GPIO_PUPDR_PUPD4_Msk |
        GPIO_PUPDR_PUPD5_Msk |
        GPIO_PUPDR_PUPD6_Msk |
        GPIO_PUPDR_PUPD7_Msk |
        GPIO_PUPDR_PUPD8_Msk |
        GPIO_PUPDR_PUPD9_Msk |
        GPIO_PUPDR_PUPD10_Msk
    );
}
void TIM2_Init(void)
{
  RCC->APB1ENR1 |= RCC_APB1ENR1_TIM2EN; // Enable the TIM2 peripheral clock
  TIM2->CNT = 0; // Ensure count starts at 0
  TIM2->ARR = (CLK_FREQUENCY/SAMPLING_FREQUENCY) - 1; // auto reload for
sampling freq
  TIM2->CCR2 = ((CLK_FREQUENCY/SAMPLING_FREQUENCY) - 1) / 2; // set 50%
duty cycle for PWM mode (ARR/2)
  // Config channel 2 to PWM Mode 1, output
  TIM2->CCMR1 &= TIM_CCMR1_CC2S; // Channel 2 configured as output
```

```c
  TIM2->CCMR1 |= TIM_CCMR1_OC2M_1 | TIM_CCMR1_OC2M_2; // MODE 1; 0110
  // Enable compare mode for Channel 2
  TIM2->CCER |= TIM_CCER_CC2E;
  // Set upcounting, Enable TIM2
  TIM2->CR1 &= ~TIM_CR1_DIR;
  TIM2->CR1 |= TIM_CR1_CEN; // TIMER ENABLE
}
void DMA_WithADC_Init(void)
{
	//Enable DMA1 clock
	RCC->AHB1ENR |= RCC_AHB1ENR_DMA1EN;
	//Set up peripheral to memory mode,
	//sizes of transfer (16 bits), automatic increments on memory
address,
	//circular mode, direction (read from peripheral, ADC1),
	//and interrupts on complete transfer
	DMA1_Channel1->CCR &= ~DMA_CCR_MEM2MEM; //turn mem2mem mode off
	DMA1_Channel1->CCR &= ~DMA_CCR_DIR; //set dir = 0
	DMA1_Channel1->CCR &= ~DMA_CCR_PINC; //turn off peripheral increment
	DMA1_Channel1->CCR |= (DMA_CCR_MSIZE_0 | DMA_CCR_PSIZE_0 |
DMA_CCR_MINC |
						DMA_CCR_CIRC | DMA_CCR_TCIE);
	//Set number of data to transfer
	DMA1_Channel1->CNDTR = NUM_SAMPLES;
	//Set addresses to read/write to/from
	DMA1_Channel1->CPAR = (uint32_t) &(ADC1->DR); //source (DIR = 0)
	DMA1_Channel1->CMAR = (uint32_t) digitalSampleArray; //destination
(DIR = 0)
	//Select DMA Channel
	DMA1_CSELR->CSELR &= ~(DMA_CSELR_C1S); //set C1S (where ADC1 is
connected) to 0000
	//Initialize NVIC to be FreeRTOS safe
	NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4); // set NVIC Priority
Grouping
	NVIC_SetPriority(DMA1_Channel1_IRQn,
NVIC_EncodePriority(NVIC_GetPriorityGrouping(), 5, 0)); // set interrupt
priorities
	NVIC_EnableIRQ(DMA1_Channel1_IRQn); // enable interrupt
	DMA1_Channel1->CCR |= DMA_CCR_EN; //enable DMA1 Channel 1
}
void ADC_Init(void)
{
	RCC->AHB2ENR |= RCC_AHB2ENR_ADCEN; //turn on clock for ADC
```

```
	ADC123_COMMON->CCR = ((ADC123_COMMON->CCR & ~(ADC_CCR_CKMODE)) |
ADC_CCR_CKMODE_0); // set ADC to be clocked at PCLK/2
	ADC1->CR &= ~ADC_CR_ADSTART; //make sure start bit is cleared to be
able to run config
	//power up ADC and voltage regulator
	ADC1->CR &= ~ADC_CR_DEEPPWD;
	ADC1->CR |= ADC_CR_ADVREGEN;
	//ADC Voltage Regulator delay
	for (uint16_t i = 0; i < 1000; i++)
	  for (uint16_t j = 0; j < 100; j++);
	//Calibrate ADC
	ADC1->CR &= ~(ADC_CR_ADEN | ADC_CR_ADCALDIF); //ensure ADC is not
enabled, single ended calibration
	ADC1->CR |= ADC_CR_ADCAL; //start calibration
	while (ADC1->CR & ADC_CR_ADCAL); //wait for calibration to finish
	ADC1->DIFSEL &= ~ADC_DIFSEL_DIFSEL_5; //PA0 is channel 5. Set low for
single ended.
	//ENABLE ADC
	ADC1->ISR |= ADC_ISR_ADRDY; //clear adrdy flag
	ADC1->CR |= ADC_CR_ADEN; //enable ADC1
	while (!(ADC1->ISR & ADC_ISR_ADRDY)); //wait for ADC1 to be ready to
start conversion
	ADC1->ISR |= ADC_ISR_ADRDY; //clear adrdy flag
	//configure ADC
	ADC1->CFGR &= ~ADC_CFGR_CONT; //single conversion
	ADC1->CFGR &= ~(ADC_CFGR_RES); //12 bit resolution
	ADC1->CFGR |= ADC_CFGR_EXTEN_0; //enable hardware trigger on rising
edge
	ADC1->CFGR |= ADC_CFGR_EXTSEL_1 | ADC_CFGR_EXTSEL_0; //enable
hardware trigger source (TIM2_CH2)
	ADC1->CFGR |= ADC_CFGR_DMACFG; //set DMA circular mode
	ADC1->CFGR |= ADC_CFGR_DMAEN; //enable DMA mode
	ADC1->CFGR |= ADC_CFGR_OVRMOD; // enable discard on overrun (most
recent sample always in ADC->DR)
	ADC1->SMPR1 &= ~(0x3) << ADC_SMPR1_SMP5_Pos; //set sample rate to 2.5
clock cycles
	//clear 1st conversion and sequence length bits
	//set 1st conversion to channel 5
	ADC1->SQR1 = (ADC1->SQR1 & ~(ADC_SQR1_SQ1_Msk | ADC_SQR1_L_Msk))
	                        | (5 << ADC_SQR1_SQ1_Pos);
	//configure GPIO pin PA0 (channel 5)
	RCC->AHB2ENR |= RCC_AHB2ENR_GPIOAEN; //enable GPIOA clock
	GPIOA->AFR[0] = (GPIOA->AFR[0] & ~(GPIO_AFRL_AFSEL0)) | (7 <<
```

```
GPIO_AFRL_AFSEL0_Pos);
      GPIOA->MODER |= GPIO_MODER_MODE0; //alternate func
      GPIOA->ASCR |= GPIO_ASCR_ASC0;
}
void USART_Init(void)
{
      RCC->AHB2ENR |= (RCC_AHB2ENR_GPIOAEN);
      GPIOA->AFR[0] &= ~(GPIO_AFRL_AFSEL2 | GPIO_AFRL_AFSEL3);    // mask
AF selection
      GPIOA->AFR[0] |= ((7 << GPIO_AFRL_AFSEL2_Pos) |             // select
USART2 (AF7)
                               (7 << GPIO_AFRL_AFSEL3_Pos));             //
for PA2 and PA3
      GPIOA->MODER &= ~(GPIO_MODER_MODE2 | GPIO_MODER_MODE3);     // enable
alternate function
      GPIOA->MODER |= (GPIO_MODER_MODE2_1 | GPIO_MODER_MODE3_1);  // for
PA2 and PA3
      // Configure USART2 connected to the debugger virtual COM port
      RCC->APB1ENR1 |= RCC_APB1ENR1_USART2EN;            // enable USART by
turning on system clock
      USART2->CR1 &= ~(USART_CR1_M1 | USART_CR1_M0);     // set data to 8
bits
      USART2->BRR = CLK_FREQUENCY / BAUD_RATE;           // baudrate
      USART2->CR1 |= USART_CR1_UE;                       // enable USART
      USART2->CR1 |= (USART_CR1_TE);      // enable transmit for USART
      // Set interrupt priority
      NVIC_SetPriorityGrouping(NVIC_PRIORITYGROUP_4); // set NVIC Priority
Grouping
      NVIC_SetPriority(USART2_IRQn,
NVIC_EncodePriority(NVIC_GetPriorityGrouping(), 5, 0)); // set interrupt
priorities
   NVIC_EnableIRQ(USART2_IRQn);
}
/* Configure Timer to interrupt 100 kHz (100 times every Tick) */
void RTOS_Stats_Timer_Init(void)
{
      RCC->APB1ENR1 |= (RCC_APB1ENR1_TIM5EN);   // turn on TIM5
      TIM5->DIER |= (TIM_DIER_UIE);             // enable interrupts
      TIM5->SR  &= ~(TIM_SR_UIF);                     // clear interrupt
flag
      TIM5->ARR = CLK_FREQUENCY/100000 - 1;
      TIM5->CR1 |= TIM_CR1_CEN;                     // start timer
      // enable interrupts
```

```
      NVIC->ISER[0] = (1 << (TIM5_IRQn & 0x1F));
}
/* Built in functions for using FreeRTOS runtime stats need to be defined*/
void configureTimerForRunTimeStats(void)
{
   ulHighFrequencyTimerTicks = 0;
   RTOS_Stats_Timer_Init();
}
unsigned long getRunTimeCounterValue(void)
{
      return ulHighFrequencyTimerTicks;
}
```

utils.h:

```c
/*
* utils.h
*
*  Created on: May 30, 2024
*      Author: thede
*/
#ifndef INC_UTILS_H_
#define INC_UTILS_H_
#include "main.h"
#define NOTE_E2     83.0
#define NOTE_A   110.0
#define NOTE_D    147.0
#define NOTE_G   196.0
#define NOTE_B    247.0
#define NOTE_E3  330.0
typedef struct {
   float frequency;
   const char *note;
} FrequencyNotePair;
// Lookup table
extern FrequencyNotePair sixStringStandard[];
float getFrequencyFromNote(const char *note);
void USART_Print(const char* message);
void USART_ESC_Code(const char* message);
void computeFFT(void);
void printHeader(void);
void printFrequency(void);
//void processInput(command_t *cmd);
void updateLEDs(float currentFrequency, float desiredFrequency);
void USART_MoveCursor(int row, int col);
int _write(int file, char *ptr, int len);
#endif /* INC_UTILS_H_ */
```

utils.c:

```c
/*
* utils.c
*
*  Created on: May 30, 2024
*      Author: thede
*/
//#include "main.h"
#include "utils.h"
#include "FreeRTOS.h"
#include "task.h"
FrequencyNotePair sixStringStandard[] = {
   { NOTE_E2, "E2" },
   { NOTE_A, "A" },
   { NOTE_D, "D" },
   { NOTE_G, "G" },
   { NOTE_B, "B" },
   { NOTE_E3, "E3" }
};
float getFrequencyFromNote(const char *note)
{
      for(int i = 0; i < 6; i++)
      {
            if (strcasecmp(note, sixStringStandard[i].note) == 0)
            {
                  return sixStringStandard[i].frequency;
            }
      }
      return 0.0;
}
void computeFFT(void)
{
      //Step 1: FFT
      arm_rfft_q15(&S, digitalSampleArray, FFTBuffer);
      //Step 2: Find Max Magnitude index
      uint16_t maxIndex = 0;
      uint16_t end = NUM_SAMPLES / 2; // only care about first half
      float maxMag = 0.0f; // to hold max magnitude for dB scaling
      for(uint16_t i = 1; i < end; i++) {
        //calculate magnitude using real and imaginary parts
        float rPart = (float) FFTBuffer[i * 2];
```

```
            float iPart = (float) FFTBuffer[(i * 2) + 1];
            float curMag = sqrtf(rPart * rPart + iPart * iPart);
            //update maxMag, index if true
            if (curMag > maxMag) {
                  maxMag = curMag;
                  maxIndex = i;
            }
        }
        //Step 3: Find corresponding frequency
        maxFrequency = ((float) maxIndex) * DELTA_F;
        //Step 4: Digital signal conditioning
        //For frequencies below ~200 Hz, the fundamental frequency
        //registers with a lower magnitude than the harmonics of
        //the same frequency. These numbers were determined
        //experimentally.
        if(desiredFrequency < 120.0)
        {
              maxFrequency = maxFrequency / 3.0;
        }
        else if (desiredFrequency < 200)
        {
              maxFrequency = maxFrequency / 2.0;
        }
        // DEBUG
//      char debugBuf[100] = {0};
//      snprintf(debugBuf, sizeof(debugBuf), "maxF: %.2f desF: %.2f curNote:
%s\r\n", maxFrequency, desiredFrequency, currentNote);
//      USART_Print(debugBuf);
}
void updateLEDs(float currentFrequency, float desiredFrequency)
{
   float difference = currentFrequency - desiredFrequency;
   uint32_t ledMask = 0x0;
   // Calculate which LED to turn on based on the frequency difference
   if (fabs(difference) <= 2) // Tight boundaries for green
   {
       ledMask = (1 << 5); // Turn on the green LED (PC5)
   }
   else if (difference > 0)
   {
       int ledIndex = 6 + ((int)(difference - 4) / 4);
       if (ledIndex > 10) ledIndex = 10;
       ledMask = (1 << ledIndex);
```

```
    }
    else
    {
        int ledIndex = 4 + ((int)(difference + 4) / 4);
        if (ledIndex < 0) ledIndex = 0;
        ledMask = (1 << ledIndex);
    }
    // Set the appropriate LED
    GPIOC->ODR = (GPIOC->ODR & ~GPIOC_PIN_MASK) | ledMask;
    if (ledMask == (1 <<5))
    {
      vTaskDelay(pdMS_TO_TICKS(5000)); // delay on green for five seconds
    }
}
void printHeader(void)
{
      USART_Print("Welcome to Guitar Tuner v1.0!\r\n\tUsage:\r\n\t");
      USART_Print("This is a standard tuner for a six-string guitar:
E2-A-D-G-B-E3\r\n\t");
      USART_Print("When prompted, please input the note you would like to
tune to.\r\n\t");
      USART_Print("At any moment, simply input the new note you want to
tune to. Happy tuning!\r\n\r\n");
}
void USART_MoveCursor(int row, int col)
{
      //init string
      char command[10] = {0};
      //make command
      snprintf(command, sizeof(command), "[%d;%dH", row, col);
      //USART_ESC_Code goes here
      USART_ESC_Code(command);
}
void USART_Print(const char* message)
{
  uint32_t i;
  for (i = 0; message[i] != 0; i++)
  {                // check for terminating NULL character
      while (!(USART2->ISR & USART_ISR_TXE));       // wait for transmit
buffer to be empty
      USART2->TDR = message[i];                      // transmit character
to USART
  }
```

```c
}
void USART_ESC_Code(const char* message)
{
      while (!(USART2->ISR & USART_ISR_TXE));
      USART2->TDR = 0x1b; // code for 'ESC'
      USART_Print(message);
}
void printFrequency(void)
{
        char freqString[15];
        snprintf(freqString, sizeof(freqString), "%.2f", maxFrequency);
        USART_Print(freqString);
}
int _write(int file, char *ptr, int len) // FOR PRINTF DEBUGGING
{
  for(int i=0; i<len; i++)
     ITM_SendChar((*ptr++));
  return len;
}
```