ECE 395 Final Report

"Colossal Avian" 30lb Combat Robot Telemetry System

Matthew Zhang (mjzhang4)

ECE 395

5/7/2024

Abstract

The iRobotics Combots teams design 30-lb combat robots that fight each other in an arena. The robot that I have been working on with the team is called "Colossal Avian." In order to design a robust robot, it is helpful to collect information from fights; both in real-time and in post. The telemetry system prototype that was designed over the duration of the semester integrates a microcontroller which interfaces with the various electronic speed controllers (ESCs) that control the robot to gather telemetry info and output that telemetry via a LoRa radio signal.

Background

30lb Combat Robot - Colossal Avian

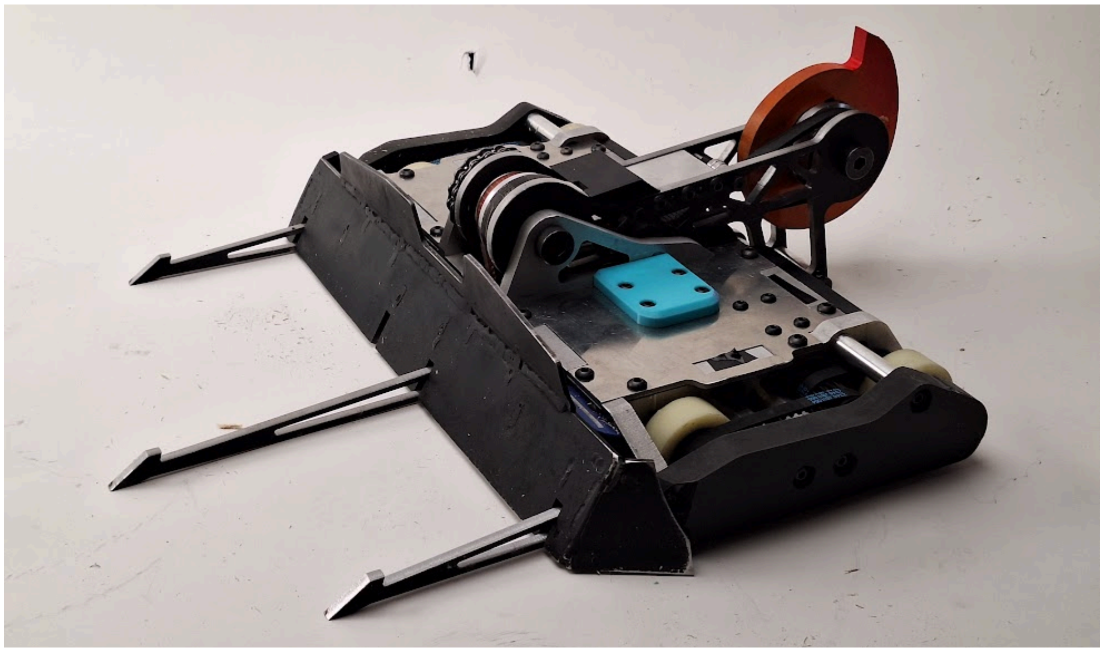

Figure 1: Image of Colossal Avian

Colossal Avian is a 30lb combat robot which features a high speed spinning disk mounted to the end of a pivoting arm, intended to deal impact damage to the tops of opposing robots. The design of the physical hardware and selection of the robot components were done out of ECE395, thus the entire robot was physically constructed and functional at the start of the class and the technical information presented here.

Robot Electronics

The whole robot is powered off of two 6 cell (22.2v nominal) lithium polymer batteries in parallel to reach a total capacity of 3000 mAh. There are a total of 4 brushless motors contained within the robot which are each individually driven by an electronic speed controller, or ESC. These ESCs are all capable of outputting telemetry data in varying formats (more detail in later sections).

- 22.2v 3000 mAh battery pack
- 2x AM32 Drive ESC
- 2x Castle Mamba X ESC
- FS2A Radio Reciever

Subsystem Overview

Block Diagram

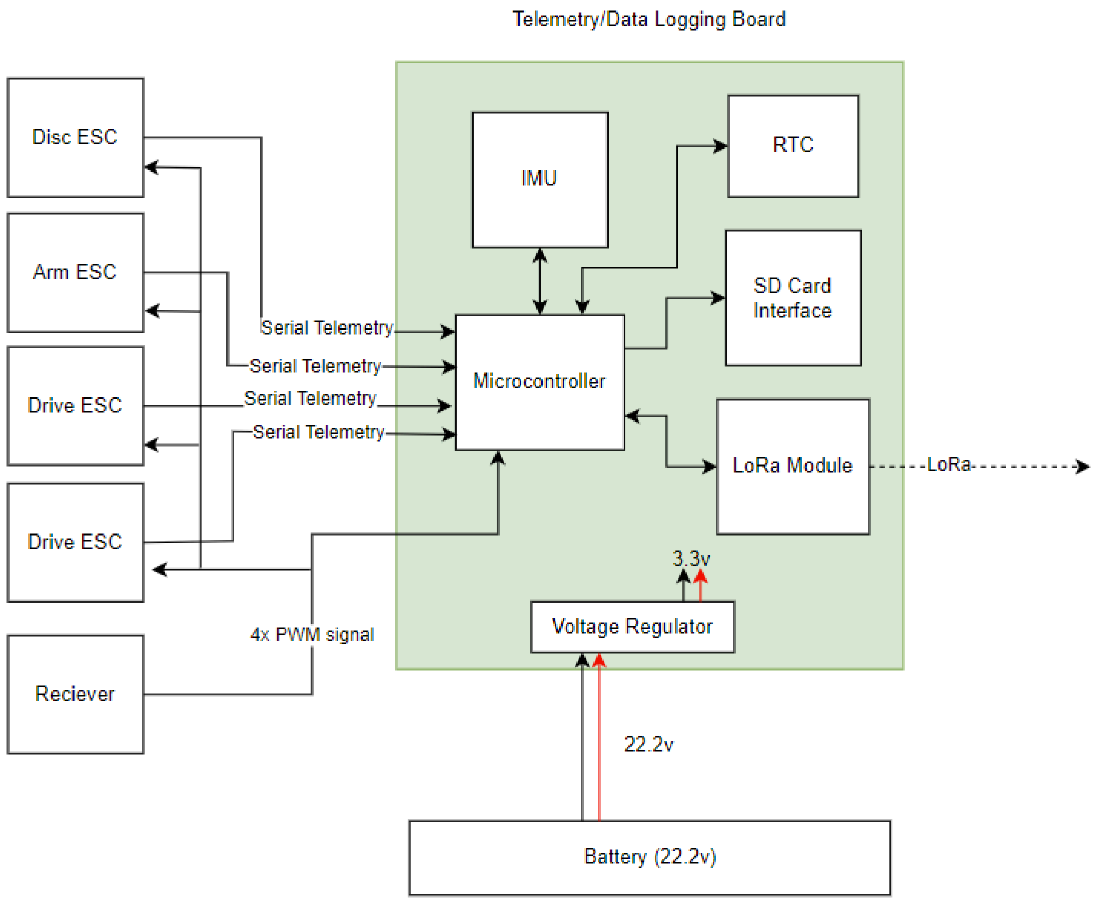

Figure 2: Block Diagram including board components and robot components

AM32 Drive Electronic Speed Controller (ESC) Telemetry

The drive motors utilize AM32 drive ESCs that are designed for drones. These ESCs operate using the KISS telemetry protocol (Outlined at this link). This protocol is a one wire serial UART which transmits a 10 byte frame consisting of the ESC temperature, input voltage, current, total battery consumption, RPM, and a CRC value.

Reading in the serial telemetry from the AM32 ESCs involves reading in the telemetry via a UART RX line on the STM32 microcontroller.

Castle Arm/Disk ESC Telemetry

The disk and arm motors both utilize a Castle Mamba X ESC. These ESCs are also capable of providing a telemetry signal over the receiver wire using a proprietary communication protocol called “Castle Live Link” which requires passing control back and forth over the receiver connection, using the timing between low pulses to communicate information. To simplify communication for the purposes of the project, a “Castle Serial Link” was used which decodes this proprietary signal, and provides a digital serial interface (either UART, I2C, or SPI) while also allowing the receiver signal to pass-through the device. This is a product developed by Castle which was purchased from them.

The original intent was to utilize the UART interface with the Castle Serial Link. However, after designing the entire board, I realized that the UART interface also requires a TX line to talk to the Serial Link to request data. Instead of UART the Castle Serial Links were bodged to the IMU’s i2c test points, taking advantage of the addressed bus to communicate between multiple devices.

LoRa Radio

In order to reliably transmit data, the board utilizes a LoRa module to send information over radio. LoRa is particularly suited for long range, low energy and low bandwidth signals. Given that the telemetry module only needs to send a small packet of numerical data, with reliability being a larger focus, LoRa was an attractive choice.

A RFM95W module was utilized and interfaced with the STM32 microcontroller via a SPI interface.

Hardware Design

System Requirements:

- Operable on max battery voltage (25.2 VDC)
- Reads telemetry from 4 different ESCs
- Can pass-through and read PWM command signals to ESCs
- Resistant to supply voltage spikes/dips
- Wireless communication is reliable in a noisy/enclosed environment inside robot
- Interfaces with Laptop
- Capable of logging data via SD Card
- USB connection for debugging via serial print statements
- Compact enough to fit within the robot

Schematic

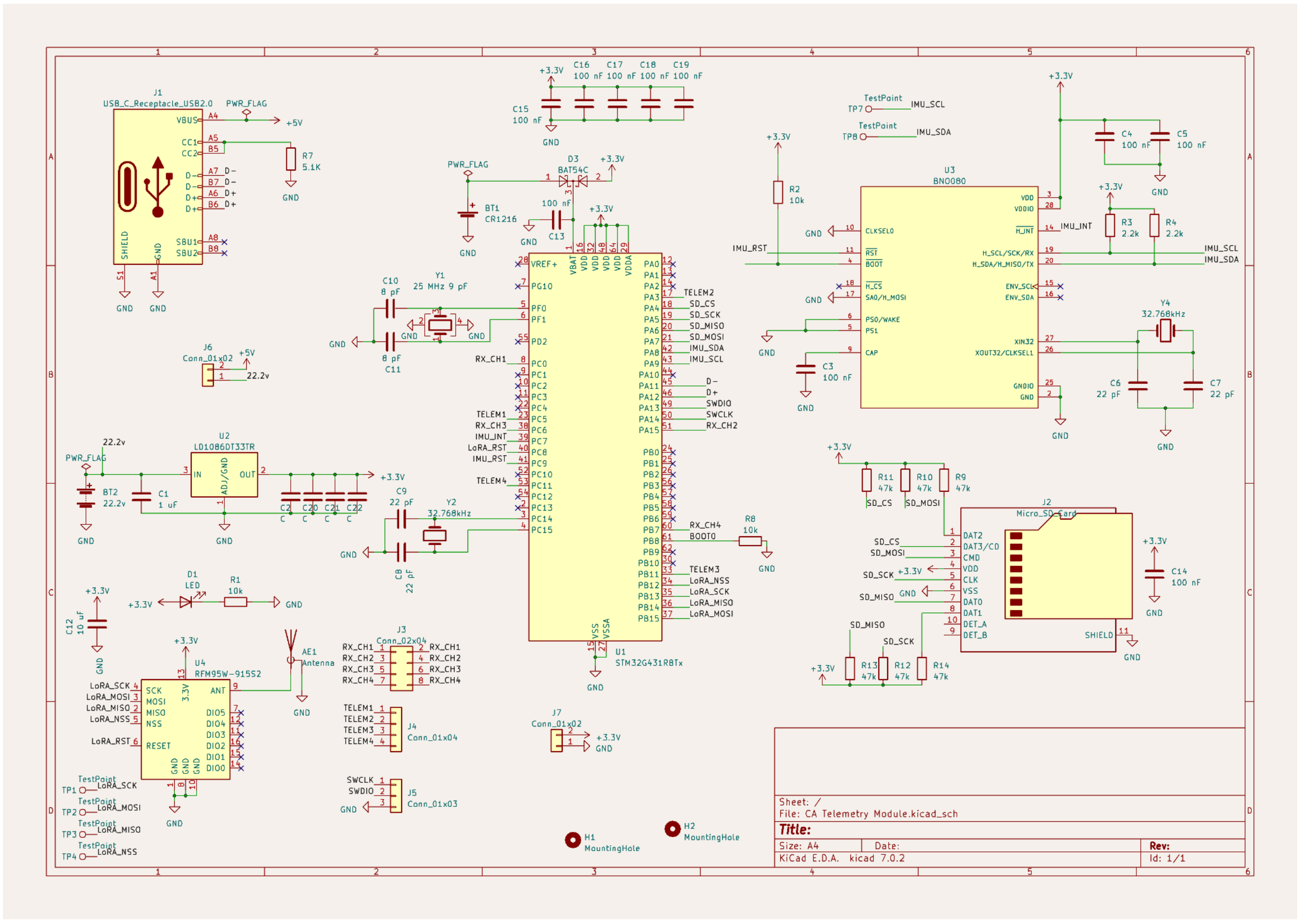

Figure 3: Board Schematic

PCB Layout

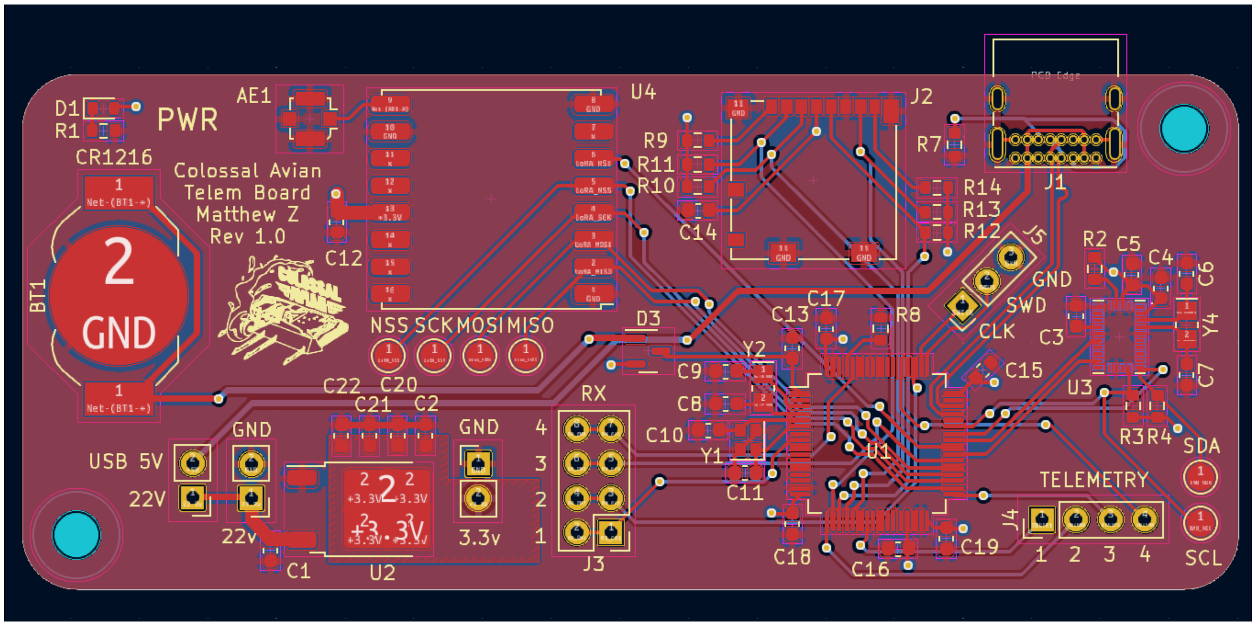

Figure 3: Board Layout

Design Features:

- Capacitor bank at LDO output to prevent voltage spikes
- Power-on LED
- M3 mounting points for mounting within robot
- USB-C connection for USB serial debugging output
- Test points and headers for major connections
- External pull-ups for i2c and SD card SPI interface

Major Components:
Microcontroller: STM32G431RBT6

- Cortex M4 provides plenty of processing overhead for the task
- Provides the required number of peripherals (timers, UARTs, USB, etc) for the required connectivity

LDO: LD1086DT33TR

- Large voltage range, 4.5v - 50v, provides flexibility for many different power sources including battery power, 5V, or other
- Large current capacity (1.5A) provides lots of overhead to not run out of current

LoRa Module: RFM95W

- Common in many projects involving LoRa radio
- 915 MHz

IMU: BNO085

- Provides on-board 9 axis sensor fusion of an accelerometer, gyro, and magnetometer, simplifying the process of getting sensor values
- I2c Interface

Learnings from First Prototype

LDO heat dissipation

- Excessive heat production when running off of battery voltage (22.2v) from dropping the voltage down to 3.3v, especially when the radio is active and drawing up to 100 mA
- Consider choosing a different type of regulator

Castle ESC Telemetry

- UART interface requires a transmit line as well to requires telemetry data from the ESC, had to bodge to IMU i2c
- Provide a dedicated i2c connection in the future

IO Protection

- During testing of the telemetry board in competition, a drive ESC went up in flames. This failure likely killed the STM32 in the board by sending a bad signal through the telemetry line.
- Future revisions could benefit from some sort of protection at the inputs to prevent component damage

LoRa signal strength

- During competition, the integrity of received LoRa signals was poor.
- A better antenna could be used on the receiver side (currently using a simple wire antenna)

Connectors

- Pin headers worked OK, but could be more compact. Within the robot space is limited, and headers were too tall in some cases
- Using other smaller connectors could be preferred

Firmware

The firmware for the telemetry board was written using STM32's CubeIDE, making use of the ST HAL and CubeIDE's pre-generated setup code.

Telemetry (I2C, UART)

The drive ESC telemetry system utilizes the UART peripherals in the STM32 in interrupt mode to capture the telemetry data. Upon reading a complete telemetry frame over UART, a callback function is triggered and used to store the telemetry data that was read.

Error handling became a necessity, as sometimes the UART interface would hit an "overrun" error, meaning that the microcontroller had become out of sync with the UART data coming in. In this case, the UART would lock up and no longer provide data. Since a single lost data point is not critical, the solution was to catch that the error had occurred, clear the error flag, and then re-enable the interrupt so that the next telemetry frame would be captured correctly.

For the Castle ESCs used for the arm and disk, telemetry data is acquired via register access through i2c. The program requests temperature, current, RPM and voltage through the i2c interface by sending the device address, then the register address, and then a checksum. The information is then sent back as a 16 bit value, accompanied by another checksum.

Error handling was important for the Castle I2C interface as well, sometimes hitting no-acknowledge errors and locking up the bus. To resolve this issue, the code would detect errors thrown during a transmission, and restart the I2C peripheral, which would clear the error and allow the next set of data to be read in correctly.

LoRa Radio (SPI)

To communicate with the LoRa module, a SPI interface was used to read and write to various registers within the RFM95W module.

In order to set up the RFM95W to transmit, the following steps are taken:

1. Write to the RFM to put into LoRa mode
2. Set the FIFO address to 0
3. Put the RFM into standby mode
4. Write the signal bandwidth, coding rate, and spreading factor to the RFM
5. Write the desired signal frequency (915 MHz)
6. Set the desired transmit power

To transmit data, the following steps are taken:

1. Write the header as the first part of the LoRa transmission. This includes:
    a. The address of the device to send to
    b. The address of the device the signal is coming from
    c. The message ID
    d. Additional flags for the message
2. Then, at the FIFO address, burst write over SPI all of the data to send. In our case this will be the telemetry data
3. Switch the mode to transmit mode
4. Wait for the transmit done flag to be raised
5. Clear the flag, and set the RFM mode back to standby.

The above process is repeated as quickly as possible to send telemetry data out via the RFM module.

Receiver PWM Capture

To capture receiver PWM signals, the timer peripherals are used in input capture mode with interrupts. Upon seeing a rising and falling edge, the input capture callback is triggered and the duty cycle is read into the program from the timer peripheral as the time elapsed between rising and falling edges of the PWM.

Incomplete Subsystems

SD Card

To record telemetry data locally without always needing a laptop connection, an SD card over SPI was designed into the board. In the interest of time and completion of the project within the semester, I followed a tutorial with example code by a user named “kiwih” posted online for using a SD card over SPI and the STM32 HAL (Tutorial located at this link). This implementation utilized the FatFS middleware along with a custom SPI driver to talk to the SD card. I was unsuccessful in getting the SD Card to mount, with an error code indicating inability to interface with the SD card. Potential issues include that the SD card I used is unable to write over SPI, or a potential code issue. I did not have time to fully investigate before the end of the project.

IMU
The BNO085 IMU used in this project is set to communicate to the microcontroller via i2c. However, it uses a proprietary method to communicate over this i2c interface called the Sensor Hub Transport Protocol or SHTP. Understanding this protocol took significant resources as opposed to a simple register access. In the time constraints of the semester, I was not able to fully understand and implement the communication between the microcontroller and the IMU.

RTC + Coin Battery
This feature was never explored much, although the hardware exists on the board for a coin cell battery to power the RTC while the device is idling. The RTC is included within the STM32, and needs some firmware code to ensure it works.

Laptop

LoRa Receiver

An Adafruit Feather 32u4 board with a pre-installed LoRa module was utilized as a dongle to allow the laptop to receive the LoRa signal. This was used over a custom solution as it was quick and easy, especially with the provided code and documentation for the device. The board is programmed to send any LoRa telemetry signals that it receives over a USB serial COM port, so that it may be read easily by a computer.

Telemetry Application

To view the telemetry data live, a python GUI was constructed to view the data in real-time using the PyQt interface. This interface features a live readout of all telemetry values, the ability to record and save data to a CSV, and live plots that update with the current telemetry values.

Appendix A: Low-Level Code
Main.c

```c
/* USER CODE BEGIN Header */
/**
 ******************************************************************************
 * @file           : main.c
 * @brief          : Main program body
 ******************************************************************************
 * @attention
 *
 * Copyright (c) 2024 STMicroelectronics.
 * All rights reserved.
 *
 * This software is licensed under terms that can be found in the LICENSE file
 * in the root directory of this software component.
 * If no LICENSE file comes with this software, it is provided AS-IS.
 *
 ******************************************************************************
 */
/* USER CODE END Header */
/* Includes ------------------------------------------------------------------*/
#include "main.h"
#include "app_fatfs.h"
#include "usb_device.h"
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include <stdio.h>
#include <string.h>
#include <stdarg.h> //for va_list var arg functions
/* USER CODE END Includes */
/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */
/* USER CODE END PTD */
/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */
/* USER CODE END PD */
/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */
/* USER CODE END PM */
/* Private variables ---------------------------------------------------------*/
I2C_HandleTypeDef hi2c2;
SPI_HandleTypeDef hspi1;
SPI_HandleTypeDef hspi2;
TIM_HandleTypeDef htim1;
TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;
```

```c
UART_HandleTypeDef huart4;
UART_HandleTypeDef huart1;
UART_HandleTypeDef huart2;
UART_HandleTypeDef huart3;
/* USER CODE BEGIN PV */
uint8_t CH1_Val = 0;
uint8_t CH2_Val = 0;
uint8_t CH3_Val = 0;
uint8_t CH4_Val = 0;
uint16_t uart_err = 0;
uint8_t temp3;
float voltage3;
float current3;
uint16_t consumption3;
uint16_t RPM3;
uint8_t UART3_rxBuffer[10];
uint8_t UART4_rxBuffer[10];
uint8_t LoRaBuffer[251] = {1, 2, 3};
uint8_t txSize = 34;
uint8_t UART3Err = 0;
uint8_t UART4Err = 0;
uint8_t SpiErr = 0;
uint8_t CastleAddr1 = 8;
uint8_t CastleAddr2 = 9;
uint8_t BNOAddr = 0x4A << 1;
uint8_t errorcode = 0;
uint8_t BNOready = 0;
/* USER CODE END PV */
/* Private function prototypes -----------------------------------------------*/
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_SPI1_Init(void);
static void MX_SPI2_Init(void);
static void MX_TIM1_Init(void);
static void MX_TIM2_Init(void);
static void MX_TIM3_Init(void);
static void MX_TIM4_Init(void);
static void MX_UART4_Init(void);
static void MX_USART1_UART_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_USART3_UART_Init(void);
static void MX_I2C2_Init(void);
/* USER CODE BEGIN PFP */
extern uint8_t CDC_Transmit_FS(uint8_t* Buf, uint16_t Len);
/* USER CODE END PFP */
/* Private user code ---------------------------------------------------------*/
/* USER CODE BEGIN 0 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
```

```
{
 if(GPIO_Pin == GPIO_PIN_7) {
        //try to read if possible
        BNOready = 1;
 } else {
 }
}
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim) {
       if (htim->Instance == TIM3) {
              CH3_Val = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
       }
       else if(htim->Instance == TIM1){
              CH1_Val = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
       }
       else if(htim->Instance == TIM2){
              CH2_Val = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_2);
       }
       else if(htim->Instance == TIM4){
              CH4_Val = HAL_TIM_ReadCapturedValue(htim, TIM_CHANNEL_1);
       }
}
void LoRaSPIWrite(uint8_t addr, uint8_t data){
       HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_RESET); //pull SS low
       uint8_t transmit_buffer[2] = {addr|0x80u , data};
       if(HAL_SPI_Transmit(&hspi2, &transmit_buffer[0] , 1, 100) != HAL_OK){
              SpiErr++;
       }
       if(HAL_SPI_Transmit(&hspi2, &transmit_buffer[1] , 1, 100) != HAL_OK){
              SpiErr++;
       }
       HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_SET); //pull SS high to end tx
}
void LoRaSPIWriteBurst(uint8_t addr, uint8_t* data, uint8_t len){
       HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_RESET); //pull SS low
       uint8_t spi_addr = addr|0x80u;
       if(HAL_SPI_Transmit(&hspi2, &spi_addr, 1, 100) != HAL_OK){
              SpiErr++;
       }
       if(HAL_SPI_Transmit(&hspi2, data , len, 100) != HAL_OK){
              SpiErr++;
       }
       HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_SET); //pull SS high to end tx
}
uint8_t LoRaSPIRead(uint8_t addr){
       uint8_t buff = addr & 0x7F;
       HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_RESET); //pull SS low
       if(HAL_SPI_Transmit(&hspi2, &buff, 1, 100)!= HAL_OK){
              SpiErr++;
	}
	if(HAL_SPI_Receive(&hspi2, &buff, 1, 100) != HAL_OK){
		SpiErr++;
	}
	HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_SET); //pull SS high to end tx
	return buff;
}
uint8_t LoRaTxDone(){
	uint8_t IRQFlags = LoRaSPIRead(0x12);//Check IRQ flags
	IRQFlags = (IRQFlags & 0x8 )>>3;
	return IRQFlags;
}
void process_telemetry(uint8_t *telemBuffer){
	temp3 = telemBuffer[0];
	voltage3 = (256*telemBuffer[1] + telemBuffer[2])/100.0;
	current3 = (256*telemBuffer[3] + telemBuffer[4])/100.0;
	consumption3 = 256*telemBuffer[5] + telemBuffer[6];
	RPM3 = ((256*telemBuffer[7] + telemBuffer[8])*100)/(24/2);
}
void try_UART3_IT(){
	if(HAL_UART_Receive_IT (&huart3, UART3_rxBuffer, 10) != HAL_OK){
		UART3Err = 1;
	}
	else{
		UART3Err = 0;
	}
}
void try_UART4_IT(){
	if(HAL_UART_Receive_IT (&huart4, UART4_rxBuffer, 10) != HAL_OK){
		UART4Err = 1;
	}
	else{
		UART4Err = 0;
	}
}
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
	if(huart->Instance == USART3){
		try_UART3_IT();
	}
	else if(huart->Instance == UART4){
		try_UART4_IT();
	}
}
void HAL_UART_ErrorCallback(UART_HandleTypeDef *huart) {
	if(huart->Instance == USART3){
		UART3Err = 1;
	}
	else if(huart->Instance == UART4){
		UART4Err = 1;
	}
}
void writeAM32TelemLoRa(){
	//drive esc 1 telem
	for (int i=0; i<9; i++){
		LoRaBuffer[i] = UART3_rxBuffer[i];
	}
	//drive esc 2 telem
	for (int i=0; i<9; i++){
		LoRaBuffer[i+9] = UART4_rxBuffer[i];
	}
	/*
	 */
}
/*
void startAccelerometer(){
	HAL_I2C_Master_Transmit(&hi2c2, BNOAddr ,TX_Buffer,1,100);
}
void startBNO(){
	HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, GPIO_PIN_RESET); //hit reset
	HAL_Delay(10);
	HAL_GPIO_WritePin(GPIOC, GPIO_PIN_9, GPIO_PIN_SET); //unreset
	HAL_Delay(4000); //wait for bno
	//enable accel
	uint8_t txbuf[21] = {0x15, 0, 0x02, 0, 0xFD, 0x1, 0, 0, 0, 0x60, 0xEA};
	HAL_I2C_Master_Transmit_IT(&i2c2, BNOAddr, txBuf, 21);
}
void readBNO(){
	if(BNOready == 1){ //check if there is data availible
		uint8_t headerBuf[4] = {0};
	 HAL_I2C_Master_Receive_IT(&i2c2, BNOAddr, headerBuf, 4 , 100); //read in the
header
	 uint8_t dataLen = ((headerBuf[0]) + (headerBuf[1]<<8) - 4); //get length from
header msb/lsb, double check this
	 HAL_I2C_Master_Receive_IT(&i2c2, BNOAddr, bnoDataBuf, dataLen , 100); //read
everythign else
	}
	BNOready = 0;
}*/
uint16_t readCastle(uint8_t esc_addr, uint8_t reg_addr){
   uint8_t buf[3] = {0};
   //int8_t tx_buf[4] = {reg_addr, 0, 0, -(reg_addr + (esc_addr<<1))};//try computing
correct checksum to send back
   uint8_t tx_buf[4] = {reg_addr, 0, 0, (~(reg_addr + (esc_addr<<1)))+1 };//try
computing correct checksum to send back
   //uint8_t tx_buf[4] = {reg_addr, 0, 0, 0xf0};
```

```c
   HAL_StatusTypeDef ret1;
   HAL_StatusTypeDef ret2;
       ret1 = HAL_I2C_Master_Transmit(&hi2c2, (esc_addr<<1), tx_buf, 4, 100);
   ret2 = HAL_I2C_Master_Receive(&hi2c2, (esc_addr<<1), buf, 3, 100);
   uint16_t val = (buf[0]<<8) + buf[1];
   if (ret1 != HAL_OK){
       val = HAL_I2C_GetError(&hi2c2);
       __HAL_I2C_DISABLE(&hi2c2);      // stop i2c
       __HAL_I2C_ENABLE(&hi2c2);               // start i2c
   }
   /*
   if (ret2 != HAL_OK){
       val = 2;
   }*/
   return val;
}
void writeLoRaCastle(){
       uint16_t castle0Temp = readCastle(8, 6);
       LoRaBuffer[18] = castle0Temp>>8;
       LoRaBuffer[19] = castle0Temp & 0xFF;
       uint16_t castle0Voltage = readCastle(8, 0);
       LoRaBuffer[20] = castle0Voltage>>8;
       LoRaBuffer[21] = castle0Voltage & 0xFF;
       uint16_t castle0Current = readCastle(8, 2);
       LoRaBuffer[22] = castle0Current>>8;
       LoRaBuffer[23] = castle0Current & 0xFF;
       uint16_t castle0RPM = readCastle(8, 5);
       LoRaBuffer[24] = castle0RPM>>8;
       LoRaBuffer[25] = castle0RPM & 0xFF;
       uint16_t castle1Temp = readCastle(9, 6);
       LoRaBuffer[26] = castle1Temp>>8;
       LoRaBuffer[27] = castle1Temp & 0xFF;
       uint16_t castle1Voltage = readCastle(9, 0);
       LoRaBuffer[28] = castle1Voltage>>8;
       LoRaBuffer[29] = castle1Voltage & 0xFF;
       uint16_t castle1Current = readCastle(9, 2);
       LoRaBuffer[30] = castle1Current>>8;
       LoRaBuffer[31] = castle1Current & 0xFF;
       uint16_t castle1RPM = readCastle(9, 5);
       LoRaBuffer[32] = castle1RPM>>8;
       LoRaBuffer[33] = castle1RPM & 0xFF;
}
void myprintf(const char *fmt, ...) {
 static char buffer[256];
 va_list args;
 va_start(args, fmt);
 vsnprintf(buffer, sizeof(buffer), fmt, args);
 va_end(args);
 int len = strlen(buffer);
 CDC_Transmit_FS((uint8_t*)buffer, len);
}
/*
uint16_t clearCastle(uint8_t esc_addr){
   uint8_t tx_buf[5] = {0, 0, 0, 0, 0};
       HAL_I2C_Master_Transmit(&hi2c2, esc_addr<<1, tx_buf, 5, HAL_MAX_DELAY);
}*/
/* USER CODE END 0 */
/**
 * @brief  The application entry point.
 * @retval int
 */
int main(void)
{
 /* USER CODE BEGIN 1 */
 /* USER CODE END 1 */
 /* MCU Configuration--------------------------------------------------------*/
 /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
 HAL_Init();
 /* USER CODE BEGIN Init */
 /* USER CODE END Init */
 /* Configure the system clock */
 SystemClock_Config();
 /* USER CODE BEGIN SysInit */
 /* USER CODE END SysInit */
 /* Initialize all configured peripherals */
 MX_GPIO_Init();
 MX_SPI1_Init();
 MX_SPI2_Init();
 MX_TIM1_Init();
 MX_TIM2_Init();
 MX_TIM3_Init();
 MX_TIM4_Init();
 MX_UART4_Init();
 MX_USART1_UART_Init();
 MX_USART2_UART_Init();
 MX_USART3_UART_Init();
 //MX_USB_Device_Init();
 MX_I2C2_Init();
 if (MX_FATFS_Init() != APP_OK) {
   Error_Handler();
 }
 /* USER CODE BEGIN 2 */
 HAL_TIM_IC_Start_IT(&htim1, TIM_CHANNEL_1); // Primary channel - rising edge
 HAL_TIM_IC_Start(&htim1, TIM_CHANNEL_2);    // Secondary channel - falling edge
 HAL_TIM_IC_Start_IT(&htim2, TIM_CHANNEL_1); // Primary channel - rising edge
 HAL_TIM_IC_Start(&htim2, TIM_CHANNEL_2);    // Secondary channel - falling edge
 HAL_TIM_IC_Start_IT(&htim3, TIM_CHANNEL_1); // Primary channel - rising edge
 HAL_TIM_IC_Start(&htim3, TIM_CHANNEL_2);    // Secondary channel - falling edge
 HAL_TIM_IC_Start_IT(&htim4, TIM_CHANNEL_1); // Primary channel - rising edge
 HAL_TIM_IC_Start(&htim4, TIM_CHANNEL_2);    // Secondary channel - falling edge
 HAL_UART_Receive_IT(&huart3, UART3_rxBuffer, 10); //start receiving telemetry on ch3
 HAL_UART_Receive_IT(&huart4, UART4_rxBuffer, 10); //start receiving telemetry on ch4
 HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_SET); //pull SS high by default
 //for printing
 uint8_t printBuffer[100] = {'\0'};
 //init LoRa
 LoRaSPIWrite(0x01, 0x00); //Put in sleep
 HAL_Delay(10);
 LoRaSPIWrite(0x01, 0x80); //Put in LoRa mode and in sleep
 HAL_Delay(10);
 uint8_t reg = LoRaSPIRead(0x01);
 if(reg != 0x80){
             sprintf(printBuffer, "Lora Failed, %u %u\n", reg, SpiErr);
             //CDC_Transmit_FS(printBuffer,sizeof(printBuffer));
 }
 else{
             sprintf(printBuffer, "Lora Success");
             //CDC_Transmit_FS(printBuffer,sizeof(printBuffer));
 }
 LoRaSPIWrite(0x0E, 0); //Write 0 to RegFifoTxBaseAddr
 LoRaSPIWrite(0x0F, 0); //Write 0 to RegFifoRxBaseAddr
 HAL_Delay(10);
 LoRaSPIWrite(0x01, 0b10000001); //Put in LoRa mode and in standby
 HAL_Delay(10);
 LoRaSPIWrite(0x1D, 0x72); // (0b01110010) write RegModemConfig 1 for signal
bandwidth (125 khz -> 0111), coding rate (4/5 001), Explicit header??
 LoRaSPIWrite(0x1E, 0x74); // (0b01110100 ) write RegModemConfig 2 for
SpreadingFactor (7 -> 0111), TxContinuousMode (off, 0), RxPayloadCrcOn ON (1),
SymbTimeout(9:8)
 //Fstep = 61.03515625hz, so to make 915 mhz, write 14991360 to frequency registers
 uint32_t LoRaFreq= 14991360;
 LoRaSPIWrite(0x06, (LoRaFreq&0xFF0000)>>16); //Write MSB
 LoRaSPIWrite(0x07, (LoRaFreq&0x00FF00)>>8); //Write middle
 LoRaSPIWrite(0x08, (LoRaFreq&0x0000FF)); //Write LSB
 LoRaSPIWrite(0x09, 0x80 | (18-2));//Set TX power to 18 (PA Select = 1) (boost)
 HAL_Delay(10);
 LoRaSPIWrite(0x0D, 0); // Position at the beginning of the FIFO
 LoRaSPIWrite(0x00, 255); //WRite  (to) to fifo
 LoRaSPIWrite(0x00, 1); //WRite header (from) to fifo
 LoRaSPIWrite(0x00, 0); //WRite header (ID) to flafifo
 LoRaSPIWrite(0x00, 0); //WRite header (flags) to fifo
 //sprintf(LoRaBuffer, "MODULE POWER ON");
 LoRaSPIWriteBurst(0x00, LoRaBuffer, txSize);
 //LoRaSPIWrite(0x00, 97); //WRite char 97 to fifo
 /*LoRaSPIWrite(0x00, 98); //WRite char 97 to fifo*/
 LoRaSPIWrite(0x22, txSize+4); //write length
 HAL_Delay(50);
 LoRaSPIWrite(0x01, 0b10000011); //Put in LoRa mode and in transmit */
 //u_int8_t buffer[] = "Hello World";
 uint8_t DebugMode = 0;
 //======================TESTING FOR SD ==============

/*
NOTE THIS IS NOT WRITTEN BY ME, THIS IS TAKEN FROM A STM32 SPI SD CARD INTERFACE
TUTORIAL LOCATED AT https://01001000.xyz/2020-08-09-Tutorial-STM32CubeIDE-SD-card/

 myprintf("\r\n~ SD card demo by kiwih ~\r\n\r\n");
 HAL_Delay(1000); //a short delay is important to let the SD card settle
 //some variables for FatFs
 FATFS FatFs;       //Fatfs handle
 FIL fil;           //File handle
 FRESULT fres; //Result after operations
 //Open the file system
 fres = f_mount(&FatFs, "", 1); //1=mount now
 if (fres != FR_OK) {
      myprintf("f_mount error (%i)\r\n", fres);
      while(1);
 }
 //Let's get some statistics from the SD card
 DWORD free_clusters, free_sectors, total_sectors;
 FATFS* getFreeFs;
 fres = f_getfree("", &free_clusters, &getFreeFs);
 if (fres != FR_OK) {
      myprintf("f_getfree error (%i)\r\n", fres);
      while(1);
 }
 //Formula comes from ChaN's documentation
 total_sectors = (getFreeFs->n_fatent - 2) * getFreeFs->csize;
 free_sectors = free_clusters * getFreeFs->csize;
 myprintf("SD card stats:\r\n%10lu KiB total drive space.\r\n%10lu KiB
available.\r\n", total_sectors / 2, free_sectors / 2);
 //Now let's try to open file "test.txt"
 fres = f_open(&fil, "test.txt", FA_READ);
 if (fres != FR_OK) {
      myprintf("f_open error (%i)\r\n");
      while(1);
 }
 myprintf("I was able to open 'test.txt' for reading!\r\n");
 //Read 30 bytes from "test.txt" on the SD card
 BYTE readBuf[30];
 //We can either use f_read OR f_gets to get data out of files
 //f_gets is a wrapper on f_read that does some string formatting for us
```

```
TCHAR* rres = f_gets((TCHAR*)readBuf, 30, &fil);
if(rres != 0) {
      myprintf("Read string from 'test.txt' contents: %s\r\n", readBuf);
} else {
      myprintf("f_gets error (%i)\r\n", fres);
}
//Be a tidy kiwi - don't forget to close your file!
f_close(&fil);
//Now let's try and write a file "write.txt"
fres = f_open(&fil, "write.txt", FA_WRITE | FA_OPEN_ALWAYS | FA_CREATE_ALWAYS);
if(fres == FR_OK) {
      myprintf("I was able to open 'write.txt' for writing\r\n");
} else {
      myprintf("f_open error (%i)\r\n", fres);
}
//Copy in a string
strncpy((char*)readBuf, "a new file is made!", 19);
UINT bytesWrote;
fres = f_write(&fil, readBuf, 19, &bytesWrote);
if(fres == FR_OK) {
      myprintf("Wrote %i bytes to 'write.txt'!\r\n", bytesWrote);
} else {
      myprintf("f_write error (%i)\r\n");
}
//Be a tidy kiwi - don't forget to close your file!
f_close(&fil);
//We're done, so de-mount the drive
f_mount(NULL, "", 0);
*/
/* USER CODE END 2 */
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
        /*
      sprintf(printBuffer, "CH1 RX: %d CH2 RX: %d CH3 RX: %d CH4 RX: %d \r\n",
CH1_Val, CH2_Val, CH3_Val, CH4_Val);
      CDC_Transmit_FS(printBuffer,sizeof(printBuffer));*/
        //HAL_Delay(50);
      if(DebugMode == 0){
            if(LoRaTxDone()){
                  LoRaSPIWrite(0x01, 0b10000001); //Put in LoRa mode and in standby
                  LoRaSPIWrite(0x12, 0xFF); //Clear IRQ
                  //write updated telemetry data to buffer
                  writeAM32TelemLoRa();
                  writeLoRaCastle();
```

```
                    LoRaSPIWrite(0x0D, 0); // Position at the beginning of the FIFO
                    LoRaSPIWrite(0x00, 255); //WRite  (to) to fifo
                    LoRaSPIWrite(0x00, 1); //WRite header (from) to fifo
                    LoRaSPIWrite(0x00, 0); //WRite header (ID) to flafifo
                    LoRaSPIWrite(0x00, 0); //WRite header (flags) to fifo
                    LoRaSPIWriteBurst(0x00, LoRaBuffer, txSize);
                    LoRaSPIWrite(0x01, 0b10000011); //Put in LoRa mode and in
transmit */
             }
        }
        else{
             process_telemetry(UART3_rxBuffer);
             sprintf(printBuffer, "%u C , %.2f V , %.2f A , %u mAh , %u RPM, ch3: %u
%u \r\n",  temp3, voltage3, current3, consumption3, RPM3, CH3_Val, uart_err);
             CDC_Transmit_FS(printBuffer,sizeof(printBuffer));
             HAL_Delay(100);
        }
        /*HAL_GPIO_TogglePin(GPIOB,GPIO_PIN_12);
        HAL_Delay(500);*/
        //transmit address and then data to LoRa
        //handle UART errors
        if(UART3Err){
             //sprintf(printBuffer, "attempting to recover");
             //CDC_Transmit_FS(printBuffer,sizeof(printBuffer));
             uart_err++;
          __HAL_UART_FLUSH_DRREGISTER(&huart3);
             __HAL_UART_CLEAR_FLAG(&huart3, UART_CLEAR_OREF);
             try_UART3_IT();
        }
        if(UART4Err){
             uart_err++;
          __HAL_UART_FLUSH_DRREGISTER(&huart4);
             __HAL_UART_CLEAR_FLAG(&huart4, UART_CLEAR_OREF);
             try_UART4_IT();
        }
 }
 /* USER CODE END 3 */
}
/**
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
 RCC_OscInitTypeDef RCC_OscInitStruct = {0};
 RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};
 /** Configure the main internal regulator output voltage
 */
 HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1);
 /** Initializes the RCC Oscillators according to the specified parameters
 * in the RCC_OscInitTypeDef structure.
 */
 RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI|RCC_OSCILLATORTYPE_HSI48;
 RCC_OscInitStruct.HSIState = RCC_HSI_ON;
 RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
 RCC_OscInitStruct.HSI48State = RCC_HSI48_ON;
 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_HSI;
 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();
 }
}
/**
 * @brief I2C2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_I2C2_Init(void)
{
 /* USER CODE BEGIN I2C2_Init 0 */
 /* USER CODE END I2C2_Init 0 */
 /* USER CODE BEGIN I2C2_Init 1 */
 /* USER CODE END I2C2_Init 1 */
 hi2c2.Instance = I2C2;
 hi2c2.Init.Timing = 0x00303D5B;
 hi2c2.Init.OwnAddress1 = 0;
 hi2c2.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
 hi2c2.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
 hi2c2.Init.OwnAddress2 = 0;
 hi2c2.Init.OwnAddress2Masks = I2C_OA2_NOMASK;
 hi2c2.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
 hi2c2.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
 if (HAL_I2C_Init(&hi2c2) != HAL_OK)
 {
   Error_Handler();
 }
 /** Configure Analogue filter
 */
 if (HAL_I2CEx_ConfigAnalogFilter(&hi2c2, I2C_ANALOGFILTER_ENABLE) != HAL_OK)
 {
   Error_Handler();
 }
 /** Configure Digital filter
 */
 if (HAL_I2CEx_ConfigDigitalFilter(&hi2c2, 0) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN I2C2_Init 2 */
 /* USER CODE END I2C2_Init 2 */
}
/**
 * @brief SPI1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_SPI1_Init(void)
{
 /* USER CODE BEGIN SPI1_Init 0 */
 /* USER CODE END SPI1_Init 0 */
 /* USER CODE BEGIN SPI1_Init 1 */
 /* USER CODE END SPI1_Init 1 */
 /* SPI1 parameter configuration*/
 hspi1.Instance = SPI1;
 hspi1.Init.Mode = SPI_MODE_MASTER;
 hspi1.Init.Direction = SPI_DIRECTION_2LINES;
 hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
 hspi1.Init.CLKPolarity = SPI_POLARITY_LOW;
 hspi1.Init.CLKPhase = SPI_PHASE_1EDGE;
 hspi1.Init.NSS = SPI_NSS_SOFT;
 hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_64;
 hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
 hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
 hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
 hspi1.Init.CRCPolynomial = 7;
 hspi1.Init.CRCLength = SPI_CRC_LENGTH_DATASIZE;
 hspi1.Init.NSSPMode = SPI_NSS_PULSE_ENABLE;
 if (HAL_SPI_Init(&hspi1) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN SPI1_Init 2 */
 /* USER CODE END SPI1_Init 2 */
}
/**
 * @brief SPI2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_SPI2_Init(void)
{
 /* USER CODE BEGIN SPI2_Init 0 */
 /* USER CODE END SPI2_Init 0 */
 /* USER CODE BEGIN SPI2_Init 1 */
 /* USER CODE END SPI2_Init 1 */
 /* SPI2 parameter configuration*/
 hspi2.Instance = SPI2;
 hspi2.Init.Mode = SPI_MODE_MASTER;
 hspi2.Init.Direction = SPI_DIRECTION_2LINES;
 hspi2.Init.DataSize = SPI_DATASIZE_8BIT;
 hspi2.Init.CLKPolarity = SPI_POLARITY_LOW;
 hspi2.Init.CLKPhase = SPI_PHASE_1EDGE;
 hspi2.Init.NSS = SPI_NSS_SOFT;
 hspi2.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_8;
 hspi2.Init.FirstBit = SPI_FIRSTBIT_MSB;
 hspi2.Init.TIMode = SPI_TIMODE_DISABLE;
 hspi2.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
 hspi2.Init.CRCPolynomial = 7;
 hspi2.Init.CRCLength = SPI_CRC_LENGTH_DATASIZE;
 hspi2.Init.NSSPMode = SPI_NSS_PULSE_DISABLE;
 if (HAL_SPI_Init(&hspi2) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN SPI2_Init 2 */
 /* USER CODE END SPI2_Init 2 */
}
/**
 * @brief TIM1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM1_Init(void)
{
 /* USER CODE BEGIN TIM1_Init 0 */
 /* USER CODE END TIM1_Init 0 */
 TIM_SlaveConfigTypeDef sSlaveConfig = {0};
 TIM_IC_InitTypeDef sConfigIC = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};
 /* USER CODE BEGIN TIM1_Init 1 */
 /* USER CODE END TIM1_Init 1 */
 htim1.Instance = TIM1;
 htim1.Init.Prescaler = 160;
 htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
 htim1.Init.Period = 2000;
 htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
 htim1.Init.RepetitionCounter = 0;
 htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
 if (HAL_TIM_IC_Init(&htim1) != HAL_OK)
 {
   Error_Handler();
 }
 sSlaveConfig.SlaveMode = TIM_SLAVEMODE_RESET;
 sSlaveConfig.InputTrigger = TIM_TS_TI1FP1;
 sSlaveConfig.TriggerPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
 sSlaveConfig.TriggerPrescaler = TIM_ICPSC_DIV1;
 sSlaveConfig.TriggerFilter = 0;
 if (HAL_TIM_SlaveConfigSynchro(&htim1, &sSlaveConfig) != HAL_OK)
 {
   Error_Handler();
 }
 sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
 sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
 sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
 sConfigIC.ICFilter = 0;
 if (HAL_TIM_IC_ConfigChannel(&htim1, &sConfigIC, TIM_CHANNEL_1) != HAL_OK)
 {
   Error_Handler();
 }
 sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
 sConfigIC.ICSelection = TIM_ICSELECTION_INDIRECTTI;
 if (HAL_TIM_IC_ConfigChannel(&htim1, &sConfigIC, TIM_CHANNEL_2) != HAL_OK)
 {
   Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterOutputTrigger2 = TIM_TRGO2_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN TIM1_Init 2 */
 /* USER CODE END TIM1_Init 2 */
}
/**
 * @brief TIM2 Initialization Function
 * @param None
```

```
 * @retval None
 */
static void MX_TIM2_Init(void)
{
 /* USER CODE BEGIN TIM2_Init 0 */
 /* USER CODE END TIM2_Init 0 */
 TIM_SlaveConfigTypeDef sSlaveConfig = {0};
 TIM_IC_InitTypeDef sConfigIC = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};
 /* USER CODE BEGIN TIM2_Init 1 */
 /* USER CODE END TIM2_Init 1 */
 htim2.Instance = TIM2;
 htim2.Init.Prescaler = 160;
 htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
 htim2.Init.Period = 2000;
 htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
 htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
 if (HAL_TIM_IC_Init(&htim2) != HAL_OK)
 {
   Error_Handler();
 }
 sSlaveConfig.SlaveMode = TIM_SLAVEMODE_RESET;
 sSlaveConfig.InputTrigger = TIM_TS_TI1FP1;
 sSlaveConfig.TriggerPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
 sSlaveConfig.TriggerPrescaler = TIM_ICPSC_DIV1;
 sSlaveConfig.TriggerFilter = 0;
 if (HAL_TIM_SlaveConfigSynchro(&htim2, &sSlaveConfig) != HAL_OK)
 {
   Error_Handler();
 }
 sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
 sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
 sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
 sConfigIC.ICFilter = 0;
 if (HAL_TIM_IC_ConfigChannel(&htim2, &sConfigIC, TIM_CHANNEL_1) != HAL_OK)
 {
   Error_Handler();
 }
 sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
 sConfigIC.ICSelection = TIM_ICSELECTION_INDIRECTTI;
 if (HAL_TIM_IC_ConfigChannel(&htim2, &sConfigIC, TIM_CHANNEL_2) != HAL_OK)
 {
   Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN TIM2_Init 2 */
 /* USER CODE END TIM2_Init 2 */
}
/**
 * @brief TIM3 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM3_Init(void)
{
 /* USER CODE BEGIN TIM3_Init 0 */
 /* USER CODE END TIM3_Init 0 */
 TIM_SlaveConfigTypeDef sSlaveConfig = {0};
 TIM_IC_InitTypeDef sConfigIC = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};
 /* USER CODE BEGIN TIM3_Init 1 */
 /* USER CODE END TIM3_Init 1 */
 htim3.Instance = TIM3;
 htim3.Init.Prescaler = 160 -1 ;
 htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
 htim3.Init.Period = 3999;
 htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
 htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
 if (HAL_TIM_IC_Init(&htim3) != HAL_OK)
 {
   Error_Handler();
 }
 sSlaveConfig.SlaveMode = TIM_SLAVEMODE_RESET;
 sSlaveConfig.InputTrigger = TIM_TS_TI1FP1;
 sSlaveConfig.TriggerPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
 sSlaveConfig.TriggerPrescaler = TIM_ICPSC_DIV1;
 sSlaveConfig.TriggerFilter = 0;
 if (HAL_TIM_SlaveConfigSynchro(&htim3, &sSlaveConfig) != HAL_OK)
 {
   Error_Handler();
 }
 sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
 sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
 sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
 sConfigIC.ICFilter = 0;
 if (HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_1) != HAL_OK)
 {
   Error_Handler();
 }
 sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
 sConfigIC.ICSelection = TIM_ICSELECTION_INDIRECTTI;
 if (HAL_TIM_IC_ConfigChannel(&htim3, &sConfigIC, TIM_CHANNEL_2) != HAL_OK)
 {
   Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN TIM3_Init 2 */
 /* USER CODE END TIM3_Init 2 */
}
/**
 * @brief TIM4 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM4_Init(void)
{
 /* USER CODE BEGIN TIM4_Init 0 */
 /* USER CODE END TIM4_Init 0 */
 TIM_SlaveConfigTypeDef sSlaveConfig = {0};
 TIM_IC_InitTypeDef sConfigIC = {0};
 TIM_MasterConfigTypeDef sMasterConfig = {0};
 /* USER CODE BEGIN TIM4_Init 1 */
 /* USER CODE END TIM4_Init 1 */
 htim4.Instance = TIM4;
 htim4.Init.Prescaler = 160;
 htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
 htim4.Init.Period = 2000;
 htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
 htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
 if (HAL_TIM_IC_Init(&htim4) != HAL_OK)
 {
   Error_Handler();
 }
 sSlaveConfig.SlaveMode = TIM_SLAVEMODE_RESET;
 sSlaveConfig.InputTrigger = TIM_TS_TI2FP2;
 sSlaveConfig.TriggerPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
 sSlaveConfig.TriggerPrescaler = TIM_ICPSC_DIV1;
 sSlaveConfig.TriggerFilter = 0;
 if (HAL_TIM_SlaveConfigSynchro(&htim4, &sSlaveConfig) != HAL_OK)
 {
   Error_Handler();
 }
 sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
 sConfigIC.ICSelection = TIM_ICSELECTION_INDIRECTTI;
```

```
 sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
 sConfigIC.ICFilter = 0;
 if (HAL_TIM_IC_ConfigChannel(&htim4, &sConfigIC, TIM_CHANNEL_1) != HAL_OK)
 {
   Error_Handler();
 }
 sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
 sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
 if (HAL_TIM_IC_ConfigChannel(&htim4, &sConfigIC, TIM_CHANNEL_2) != HAL_OK)
 {
   Error_Handler();
 }
 sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
 sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
 if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN TIM4_Init 2 */
 /* USER CODE END TIM4_Init 2 */
}
/**
 * @brief UART4 Initialization Function
 * @param None
 * @retval None
 */
static void MX_UART4_Init(void)
{
 /* USER CODE BEGIN UART4_Init 0 */
 /* USER CODE END UART4_Init 0 */
 /* USER CODE BEGIN UART4_Init 1 */
 /* USER CODE END UART4_Init 1 */
 huart4.Instance = UART4;
 huart4.Init.BaudRate = 115200;
 huart4.Init.WordLength = UART_WORDLENGTH_8B;
 huart4.Init.StopBits = UART_STOPBITS_1;
 huart4.Init.Parity = UART_PARITY_NONE;
 huart4.Init.Mode = UART_MODE_RX;
 huart4.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart4.Init.OverSampling = UART_OVERSAMPLING_16;
 huart4.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart4.Init.ClockPrescaler = UART_PRESCALER_DIV1;
 huart4.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart4) != HAL_OK)
 {
   Error_Handler();
 }
 if (HAL_UARTEx_SetTxFifoThreshold(&huart4, UART_TXFIFO_THRESHOLD_1_8) != HAL_OK)
```

```
  {
    Error_Handler();
  }
  if (HAL_UARTEx_SetRxFifoThreshold(&huart4, UART_RXFIFO_THRESHOLD_1_8) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_UARTEx_DisableFifoMode(&huart4) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN UART4_Init 2 */
  /* USER CODE END UART4_Init 2 */
}
/**
  * @brief USART1 Initialization Function
  * @param None
  * @retval None
  */
static void MX_USART1_UART_Init(void)
{
  /* USER CODE BEGIN USART1_Init 0 */
  /* USER CODE END USART1_Init 0 */
  /* USER CODE BEGIN USART1_Init 1 */
  /* USER CODE END USART1_Init 1 */
  huart1.Instance = USART1;
  huart1.Init.BaudRate = 115200;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  huart1.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
  huart1.Init.ClockPrescaler = UART_PRESCALER_DIV1;
  huart1.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_UARTEx_SetTxFifoThreshold(&huart1, UART_TXFIFO_THRESHOLD_1_8) != HAL_OK)
  {
    Error_Handler();
  }
  if (HAL_UARTEx_SetRxFifoThreshold(&huart1, UART_RXFIFO_THRESHOLD_1_8) != HAL_OK)
  {
    Error_Handler();
  }
```

```
 if (HAL_UARTEx_DisableFifoMode(&huart1) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN USART1_Init 2 */
 /* USER CODE END USART1_Init 2 */
}
/**
 * @brief USART2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_USART2_UART_Init(void)
{
 /* USER CODE BEGIN USART2_Init 0 */
 /* USER CODE END USART2_Init 0 */
 /* USER CODE BEGIN USART2_Init 1 */
 /* USER CODE END USART2_Init 1 */
 huart2.Instance = USART2;
 huart2.Init.BaudRate = 115200;
 huart2.Init.WordLength = UART_WORDLENGTH_8B;
 huart2.Init.StopBits = UART_STOPBITS_1;
 huart2.Init.Parity = UART_PARITY_NONE;
 huart2.Init.Mode = UART_MODE_RX;
 huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart2.Init.OverSampling = UART_OVERSAMPLING_16;
 huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart2.Init.ClockPrescaler = UART_PRESCALER_DIV1;
 huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart2) != HAL_OK)
 {
   Error_Handler();
 }
 if (HAL_UARTEx_SetTxFifoThreshold(&huart2, UART_TXFIFO_THRESHOLD_1_8) != HAL_OK)
 {
   Error_Handler();
 }
 if (HAL_UARTEx_SetRxFifoThreshold(&huart2, UART_RXFIFO_THRESHOLD_1_8) != HAL_OK)
 {
   Error_Handler();
 }
 if (HAL_UARTEx_DisableFifoMode(&huart2) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN USART2_Init 2 */
 /* USER CODE END USART2_Init 2 */
}
```

```c
/**
 * @brief USART3 Initialization Function
 * @param None
 * @retval None
 */
static void MX_USART3_UART_Init(void)
{
 /* USER CODE BEGIN USART3_Init 0 */
 /* USER CODE END USART3_Init 0 */
 /* USER CODE BEGIN USART3_Init 1 */
 /* USER CODE END USART3_Init 1 */
 huart3.Instance = USART3;
 huart3.Init.BaudRate = 115200;
 huart3.Init.WordLength = UART_WORDLENGTH_8B;
 huart3.Init.StopBits = UART_STOPBITS_1;
 huart3.Init.Parity = UART_PARITY_NONE;
 huart3.Init.Mode = UART_MODE_RX;
 huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
 huart3.Init.OverSampling = UART_OVERSAMPLING_16;
 huart3.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
 huart3.Init.ClockPrescaler = UART_PRESCALER_DIV1;
 huart3.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
 if (HAL_UART_Init(&huart3) != HAL_OK)
 {
   Error_Handler();
 }
 if (HAL_UARTEx_SetTxFifoThreshold(&huart3, UART_TXFIFO_THRESHOLD_1_8) != HAL_OK)
 {
   Error_Handler();
 }
 if (HAL_UARTEx_SetRxFifoThreshold(&huart3, UART_RXFIFO_THRESHOLD_1_8) != HAL_OK)
 {
   Error_Handler();
 }
 if (HAL_UARTEx_DisableFifoMode(&huart3) != HAL_OK)
 {
   Error_Handler();
 }
 /* USER CODE BEGIN USART3_Init 2 */
 /* USER CODE END USART3_Init 2 */
}
/**
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void)
{
 GPIO_InitTypeDef GPIO_InitStruct = {0};
/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */
 /* GPIO Ports Clock Enable */
 __HAL_RCC_GPIOC_CLK_ENABLE();
 __HAL_RCC_GPIOF_CLK_ENABLE();
 __HAL_RCC_GPIOA_CLK_ENABLE();
 __HAL_RCC_GPIOB_CLK_ENABLE();
 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(SD_CS_GPIO_Port, SD_CS_Pin, GPIO_PIN_SET);
 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOB, GPIO_PIN_12, GPIO_PIN_SET);
 /*Configure GPIO pin Output Level */
 HAL_GPIO_WritePin(GPIOC, GPIO_PIN_8|GPIO_PIN_9, GPIO_PIN_SET);
 /*Configure GPIO pin : SD_CS_Pin */
 GPIO_InitStruct.Pin = SD_CS_Pin;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_PULLUP;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
 HAL_GPIO_Init(SD_CS_GPIO_Port, &GPIO_InitStruct);
 /*Configure GPIO pin : PB12 */
 GPIO_InitStruct.Pin = GPIO_PIN_12;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_PULLUP;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_MEDIUM;
 HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
 /*Configure GPIO pin : PC7 */
 GPIO_InitStruct.Pin = GPIO_PIN_7;
 GPIO_InitStruct.Mode = GPIO_MODE_IT_FALLING;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
 /*Configure GPIO pin : PC8 */
 GPIO_InitStruct.Pin = GPIO_PIN_8;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_PULLUP;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
 /*Configure GPIO pin : PC9 */
 GPIO_InitStruct.Pin = GPIO_PIN_9;
 GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
 GPIO_InitStruct.Pull = GPIO_NOPULL;
 GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
 HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
 /* EXTI interrupt init*/
 HAL_NVIC_SetPriority(EXTI9_5_IRQn, 0, 0);
 HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);
/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
```

```
}
/* 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 */
}
#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 */
```

Appendix B: Python GUI Code

```
import sys
from PyQt5.QtWidgets import QApplication, QWidget, QLabel, QVBoxLayout,
QHBoxLayout, QGridLayout, QPushButton, QComboBox
from PyQt5.QtGui import QFont
from PyQt5.QtCore import QTimer, QThread, pyqtSignal

from PyQt5.QtWidgets import QApplication, QMainWindow, QVBoxLayout, QWidget
import pyqtgraph as pg
```

```python
import csv
import serial
from serial.tools import list_ports
from datetime import datetime
import random

# font styles
font_family = 'Bahnschrift'
avian_font = QFont(font_family, 20, QFont.Bold)
esc_label_font = QFont(font_family, 12, QFont.Bold)
measurement_font = QFont(font_family, 10)
value_font = QFont(font_family, 28, QFont.Bold)
min_max_font = QFont(font_family, 10)

# consts
DRIVE_ESC_1 = 'Drive ESC 1'
DRIVE_ESC_2 = 'Drive ESC 2'
WEAPON_ESC = 'Weapon ESC'
ARM_ESC = 'Arm ESC'

TEMP = 'Temp'
RPM = 'RPM'
CURRENT = 'Current'
CONSUMPTION = 'Consumption'
VOLTAGE = 'Voltage'

BATTERY_VOLTAGE = 'Battery Voltage'
TOTAL_CURRENT = 'Total Current'
TOTAL_CONSUMPTION = 'Total Consumption'
SIGNAL_STRENGTH = 'Signal Strength'


class SerialReaderThread(QThread):
    new_data = pyqtSignal(str)
    timestamps = []
    raw_data = []

    def __init__(self, port, parent=None):
        super().__init__(parent)
        self.baudrate = 115200
        self.serial_port = serial.Serial(port, self.baudrate)
        self.serial_port.flushInput()

    def set_port(self, port):
        self.serial_port = serial.Serial(port, self.baudrate)
        self.serial_port.flushInput()

    def run(self):
        while True:
            if self.serial_port.in_waiting:
                line = self.serial_port.readline().decode().strip()
                self.new_data.emit(line)
                self.timestamps.append(datetime.now())
                self.raw_data.append(line)
                print(line)

    def export_raw_data(self):
        now = datetime.now().strftime('%Y_%m_%d_%H_%M_%S')
        file_name = f"avian_data_{now}_raw.csv"
        with open(file_name, "w", newline="") as csv_file:
            writer = csv.writer(csv_file)
            writer.writerows(self.raw_data)


class Avian():
    def __init__(self, serial_port):
        self.data = {}
        if not serial_port == None:
            self.serial_reader = SerialReaderThread(serial_port)
            self.serial_reader.new_data.connect(self.handle_data)
            self.serial_reader.start()

        self.data_timestamps = []

        self.esc_names = [DRIVE_ESC_1, DRIVE_ESC_2, WEAPON_ESC, ARM_ESC]
        self.esc_measurement_names = [TEMP, RPM, CURRENT, CONSUMPTION, VOLTAGE]
        self.esc_measurement_units = ["°C", "", "A", "mAh", "V"]
        for esc_name in self.esc_names:
            esc_data = {}
            for measurement_name in self.esc_measurement_names:
                esc_data[measurement_name] = {
                    'values': [],
                    'min': None,
                    'max': None,
                }
            self.data[esc_name] = esc_data

        self.robot_measurement_names = [
            BATTERY_VOLTAGE, TOTAL_CURRENT, TOTAL_CONSUMPTION, SIGNAL_STRENGTH
        ]

        for measurement_name in self.robot_measurement_names:
            self.data[measurement_name] = {
                'values': [],
                'min': None,
                'max': None,
            }

    def get_esc_names(self):
        return self.esc_names

    def get_esc_measurement_names(self):
        return self.esc_measurement_names

    def get_esc_measurement_units(self):
        return self.esc_measurement_units

    def get_displayed_esc_measurement_names(self):
        return self.esc_measurement_names[:4]

    def get_displayed_esc_measurement_units(self):
        return self.esc_measurement_units[:4]

    def get_robot_measurement_names(self):
        return self.robot_measurement_names
```

```python
    def get_all_values(self, measurement, esc=None):
        obj = (self.data if esc == None else self.data[esc])
        return obj[measurement]['values']

    def get_last_n_values(self, measurement, n, esc=None):
        obj = (self.data if esc == None else self.data[esc])
        return obj[measurement]['values'][-n:]

    def get_current_value(self, measurement, esc=None):
        obj = (self.data if esc == None else self.data[esc])
        values = obj[measurement]['values']
        return values[-1] if len(values) > 0 else None

    def get_min_value(self, measurement, esc=None):
        obj = (self.data if esc == None else self.data[esc])
        return obj[measurement]['min']

    def get_max_value(self, measurement, esc=None):
        obj = (self.data if esc == None else self.data[esc])
        return obj[measurement]['max']

    def add_value(self, measurement, value, esc=None):
        obj = (self.data if esc == None else self.data[esc])
        obj[measurement]['values'].append(value)
        min_value = self.get_min_value(measurement, esc)
        if min_value == None or value < min_value:
            obj[measurement]['min'] = value
        max_value = self.get_max_value(measurement, esc)
        if max_value == None or value > max_value:
            obj[measurement]['max'] = value

    def export_to_csv(self):
        timestamp = datetime.now().strftime('%Y_%m_%d_%H_%M_%S')
        csv_data = []

        headers = ['Timestamp']
        for esc in self.get_esc_names():
            for measurement in self.get_esc_measurement_names():
                headers.append(f"{esc} {measurement}")
        for measurement in self.robot_measurement_names:
            headers.append(measurement)
        csv_data.append(headers)

        # one row per timestamp
        for i in range(len(self.data_timestamps)):
            data_row = [self.data_timestamps[i].strftime('%H_%M_%S_%f')]
            for esc in self.get_esc_names():
                for measurement in self.get_esc_measurement_names():
                    all_values = self.get_all_values(measurement, esc)
                    data_row.append(all_values[i] if len(
                        all_values) > i else None)

            for measurement in self.robot_measurement_names:
                all_values = self.get_all_values(measurement)
                data_row.append(all_values[i] if len(
                    all_values) > i else None)
            csv_data.append(data_row)

        file_name = f"avian_data_{timestamp}.csv"
        with open(file_name, "w", newline="") as csv_file:
            writer = csv.writer(csv_file)
            writer.writerows(csv_data)

        self.serial_reader.export_raw_data()

    def print_data(self):
        print(self.data)

    def handle_data(self, data):
        if (not "Data:" in data):
            return

        now_timestamp = datetime.now()
        self.data_timestamps.append(now_timestamp)
        data_array = list(map(lambda str: int(str), data.split()[1:]))
        split_data = {
            DRIVE_ESC_1: data_array[0:9],  # first 9
            DRIVE_ESC_2: data_array[9:18],  # next 9
            WEAPON_ESC: data_array[18:26],  # next 8
            ARM_ESC: data_array[26:34],  # last 8
            SIGNAL_STRENGTH: data_array[34]
        }

        def merge_bytes(byte1, byte2):
            return (byte1 << 8) + byte2

        parsed_esc_data = {}
        for esc in [DRIVE_ESC_1, DRIVE_ESC_2]:
            esc_data = split_data[esc]
            parsed_esc_data[esc] = {
                TEMP: esc_data[0],
                VOLTAGE: merge_bytes(esc_data[1], esc_data[2]) / 100,
                CURRENT: merge_bytes(esc_data[3], esc_data[4]) / 100,
                CONSUMPTION: merge_bytes(esc_data[5], esc_data[6]),
                RPM: int(merge_bytes(esc_data[7], esc_data[8]) * 100 / 6)
            }

        for esc in [WEAPON_ESC, ARM_ESC]:
            esc_data = split_data[esc]
            scale_val = 2042
            current = merge_bytes(esc_data[4], esc_data[5]) / scale_val * 50
            delta_time_hours = (
                now_timestamp - self.data_timestamps[-1]).total_seconds() / 3600
            consumption = current * 1000 * delta_time_hours
            parsed_esc_data[esc] = {
                TEMP: merge_bytes(esc_data[0], esc_data[1]) / scale_val * 30,
                VOLTAGE: merge_bytes(esc_data[2], esc_data[3]) / scale_val * 20,
                CURRENT: current,
                CONSUMPTION: consumption,
                RPM: int(merge_bytes(
                    esc_data[6], esc_data[7]) / scale_val * 20416.66 / 7)
            }

        for esc in parsed_esc_data:
            for measurement in parsed_esc_data[esc]:
                parsed_esc_data[esc][measurement] = round(
                    parsed_esc_data[esc][measurement], 2)
                self.add_value(
                    measurement, parsed_esc_data[esc][measurement], esc)

        parsed_robot_data = {}
        parsed_robot_data[BATTERY_VOLTAGE] =
parsed_esc_data[DRIVE_ESC_1][VOLTAGE]
        parsed_robot_data[TOTAL_CURRENT] = sum(
            list(map(lambda esc: parsed_esc_data[esc][CURRENT], self.esc_names)))
        parsed_robot_data[TOTAL_CONSUMPTION] = sum(
            list(map(lambda esc: parsed_esc_data[esc][CONSUMPTION],
self.esc_names)))
        parsed_robot_data[SIGNAL_STRENGTH] = split_data[SIGNAL_STRENGTH]

        for measurement in parsed_robot_data:
            self.add_value(
                measurement, round(parsed_robot_data[measurement], 2)
            )


class TelemetryGUI(QWidget):
    def __init__(self):
        super().__init__()
        self.com_port_dropdown = QComboBox()
        ports = list_ports.comports()
        self.port_names = list(map(lambda port: port.name, ports))
        self.com_port_dropdown.addItems(self.port_names)

        if len(self.port_names) > 0:
            self.avian = Avian(self.port_names[0])
        else:
            self.avian = Avian(None)

        self.displayed_data = {}

        # OPTIONS
        self.should_show_plots = True
        self.use_fake_data = False
        self.num_values_to_plot = 50
```

```python
        self.initialize_gui()

    # TODO: hook up properly
    def on_select_port(self, index):
        selected_port = self.port_names[index]
        self.avian = Avian(selected_port)

    def initialize_gui(self):
        self.main_layout = QHBoxLayout()
        self.setLayout(self.main_layout)

        esc_layout = QGridLayout()
        self.main_layout.addLayout(esc_layout)

        for esc_index, esc_name in enumerate(self.avian.get_esc_names()):
            esc_square = QVBoxLayout()

            esc_label = QLabel(esc_name)
            esc_label.setFont(esc_label_font)
            esc_square.addWidget(esc_label)

            measurement_grid = QGridLayout()
            measurement_grid.setSpacing(4)

            displayed_measurements =
self.avian.get_displayed_esc_measurement_names()
            displayed_measurement_units =
self.avian.get_displayed_esc_measurement_units()

            for measurement_index, measurement_name in
enumerate(displayed_measurements):
                units = displayed_measurement_units[
                    measurement_index
                ]
                measurement_square = QWidget()
                measurement_square_style = "background-color: #cccccc;"
                measurement_square.setStyleSheet(measurement_square_style)
                esc_layout.addWidget(measurement_square)
                value_column = QVBoxLayout(measurement_square)
                self.create_measurement_display(
                    value_column, measurement_name, units, esc_name
                )
                measurement_grid.addWidget(
                    measurement_square, measurement_index // 2, measurement_index
% 2
                )

            esc_square.addLayout(measurement_grid)
            esc_layout.addLayout(esc_square, esc_index // 2, esc_index % 2)

        # right side
        robot_column = QVBoxLayout()
        self.main_layout.addLayout(robot_column)

        avian_label = QLabel("Colossal Avian")
        avian_label.setFont(avian_font)
        robot_column.addWidget(avian_label)

        self.create_measurement_display(
            robot_column, BATTERY_VOLTAGE, 'V'
        )
        self.total_current_label = self.create_measurement_display(
            robot_column, TOTAL_CURRENT, 'A'
        )
        self.total_consumption_label = self.create_measurement_display(
            robot_column, TOTAL_CONSUMPTION, 'mAh'
        )
        self.total_consumption_label = self.create_measurement_display(
            robot_column, SIGNAL_STRENGTH, 'dBm'
        )

        dropdown_title = QLabel("COM Port")
        dropdown_title.setFont(measurement_font)
        robot_column.addWidget(dropdown_title)
        robot_column.addWidget(self.com_port_dropdown)

        export_button = QPushButton("Export to CSV")
        export_button.clicked.connect(self.avian.export_to_csv)
        robot_column.addWidget(export_button)

        # flex
        self.main_layout.setStretch(0, 8)
        self.main_layout.setStretch(1, 2)

        self.setWindowTitle('Colossal Avian')
        self.setGeometry(100, 100, 500, 300)

        self.timer = QTimer()
        self.timer.timeout.connect(self.update_gui)
        self.timer.start(100)

    def update_gui(self):
        if (self.use_fake_data):
            print('using fake data')
            for measurement in self.avian.get_robot_measurement_names():
                self.avian.add_value(measurement, random.randint(-100, 0)
                                     if measurement == SIGNAL_STRENGTH
                                     else random.randint(0, 100)
                                     )
            for esc in self.avian.get_esc_names():
                for measurement in
self.avian.get_displayed_esc_measurement_names():
                    self.avian.add_value(
                        measurement, random.randint(0, 100), esc)

        # update labels and plots
        for measurement in self.avian.get_robot_measurement_names():
            self.update_label_and_plot(measurement)
        for esc in self.avian.get_esc_names():
            for measurement in self.avian.get_displayed_esc_measurement_names():
                self.update_label_and_plot(measurement, esc)

    def create_measurement_display(self, layout, measurement, units, esc=None):
        name_label = QLabel(measurement)
        name_label.setFont(measurement_font)
        layout.addWidget(name_label)
        current_value = self.avian.get_current_value(measurement, esc)
        value_label = QLabel(str(current_value))
        value_label.setFont(value_font)
        layout.addWidget(value_label)

        data = self.avian.get_all_values(
            measurement, esc
        )

        min_value = self.avian.get_min_value(measurement, esc)
        max_value = self.avian.get_max_value(measurement, esc)
        min_max_label = QLabel(
            f"{min_value} | {max_value}"
        )
        min_max_label.setFont(min_max_font)
        layout.addWidget(min_max_label)

        if self.should_show_plots and measurement != SIGNAL_STRENGTH:
            pg.setConfigOption('background', 'w')
            # pg.setConfigOption('foreground', 'k')
            plot = pg.PlotWidget()
            # plot.setMaximumHeight(50)
            layout.addWidget(plot)
        else:
            plot = None

        display_data = {'value_label': value_label, 'units': units,
'min_max_label': min_max_label,
                        'plot': plot, 'data': data}
        if esc == None:
            self.displayed_data[measurement] = display_data
        else:
            if not esc in self.displayed_data:
                self.displayed_data[esc] = {}
            self.displayed_data[esc][measurement] = display_data

    def update_label_and_plot(self, measurement, esc=None):
        obj = self.displayed_data[measurement] if esc == None else
self.displayed_data[esc][measurement]
        value = self.avian.get_current_value(measurement, esc)
        value_text = f"{str(value)} {obj['units']}"
        min_max_text = f"{str(self.avian.get_min_value(measurement, esc))} |
{str(self.avian.get_max_value(measurement, esc))}"

        if esc == None:
            self.displayed_data[measurement]['value_label'].setText(value_text)
            if measurement == TOTAL_CONSUMPTION and value != None:
                percent = round(100 * value / 3000, 2)
                self.displayed_data[measurement]['value_label'].setText(
                    f"{value_text} ({percent}%)"
                )
            elif measurement == SIGNAL_STRENGTH and value != None:
                if value < -90:
                    style_sheet = 'background-color: red;'
                elif value < -80:
                    style_sheet = 'background-color: orange;'
                elif value < -70:
                    style_sheet = 'background-color: yellow;'
                else:
                    style_sheet = 'color: black;'
                self.displayed_data[measurement]['value_label'].setStyleSheet(
                    style_sheet
                )

            self.displayed_data[measurement]['min_max_label'].setText(
                min_max_text
            )
        else:
            self.displayed_data[esc][measurement]['value_label'].setText(
                value_text
            )
            if measurement == TEMP and value != None:
                if value >= 80:
                    style_sheet = 'background-color: red;'
                elif value >= 70:
                    style_sheet = 'background-color: orange;'
                elif value >= 60:
                    style_sheet = 'background-color: yellow;'
                else:
                    style_sheet = 'color: black;'

self.displayed_data[esc][measurement]['value_label'].setStyleSheet(
                    style_sheet
                )

            self.displayed_data[esc][measurement]['min_max_label'].setText(
                min_max_text
            )

        # self.should_show_plots = self.displayed_data[measurement][
        #     'ax'] != None if esc == None else
self.displayed_data[esc][measurement]['ax'] != None

        measurements_to_plot_all = [CONSUMPTION,
                                    BATTERY_VOLTAGE, TOTAL_CONSUMPTION, TEMP]
        if self.should_show_plots and measurement != SIGNAL_STRENGTH:
            if measurement in measurements_to_plot_all:
                data = self.avian.get_all_values(measurement, esc)
            else:
                data = self.avian.get_last_n_values(
                    measurement, self.num_values_to_plot, esc
                )

            pen_options = pg.mkPen('k', width=1)
            if esc == None:
                self.displayed_data[measurement]['data'] = data
                self.displayed_data[measurement]['plot'].clear()
                self.displayed_data[measurement]['plot'].plot(
                    self.displayed_data[measurement]['data'],
                    pen=pen_options
                )
            else:
                self.displayed_data[esc][measurement]['data'] = data
                self.displayed_data[esc][measurement]['plot'].clear()
                self.displayed_data[esc][measurement]['plot'].plot(
                    self.displayed_data[esc][measurement]['data'],
                    pen=pen_options
                )

    # def closeEvent(self, event):
        # self.avian.export_to_csv()f


if __name__ == '__main__':
    app = QApplication(sys.argv)
    window = TelemetryGUI()
    window.show()
    sys.exit(app.exec_())
```