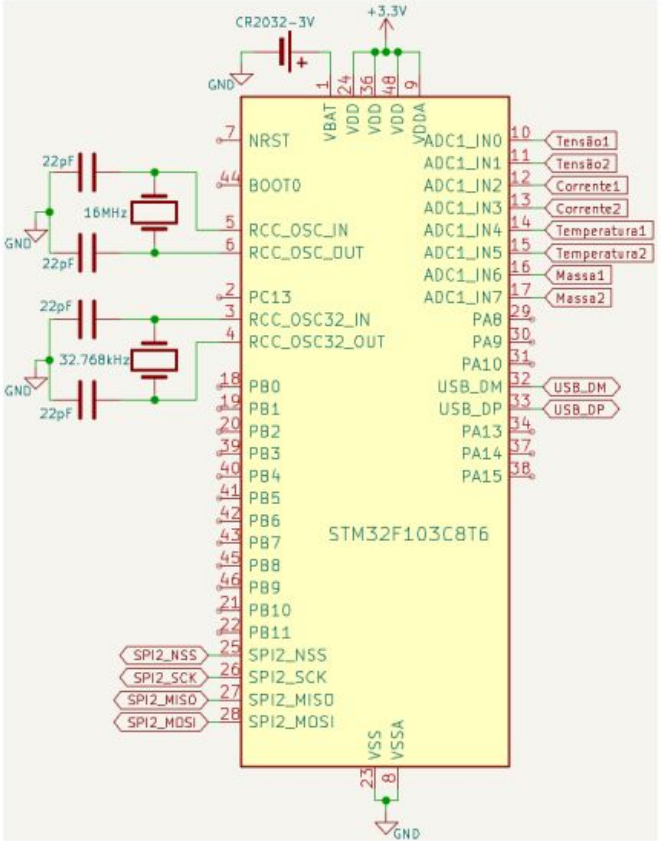
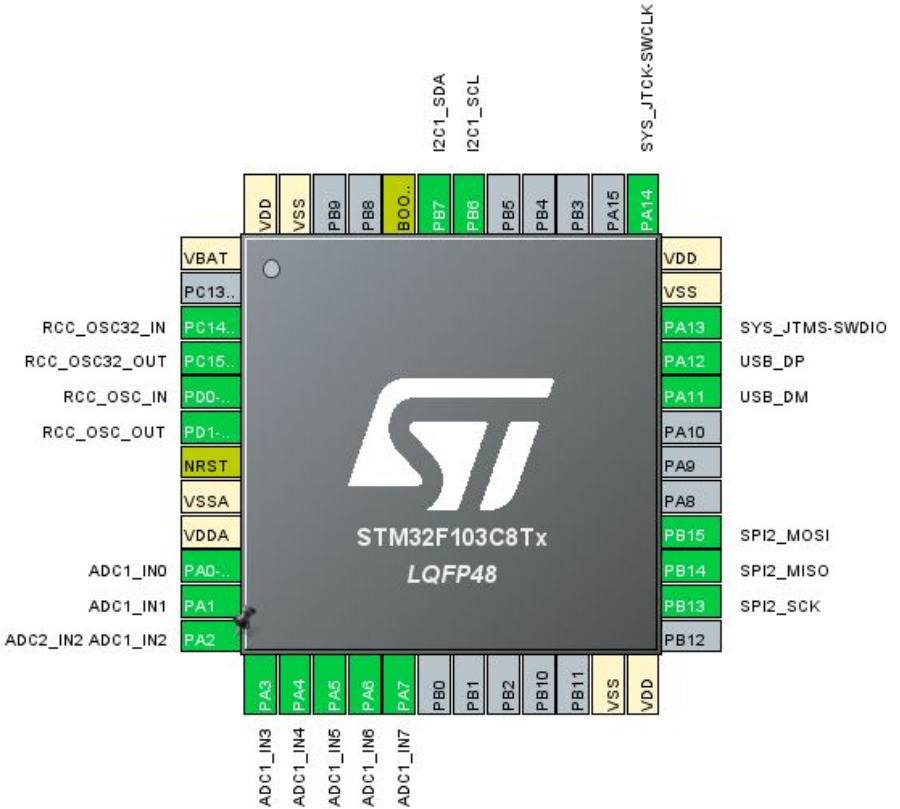
Programação do STM32F103C8Tx

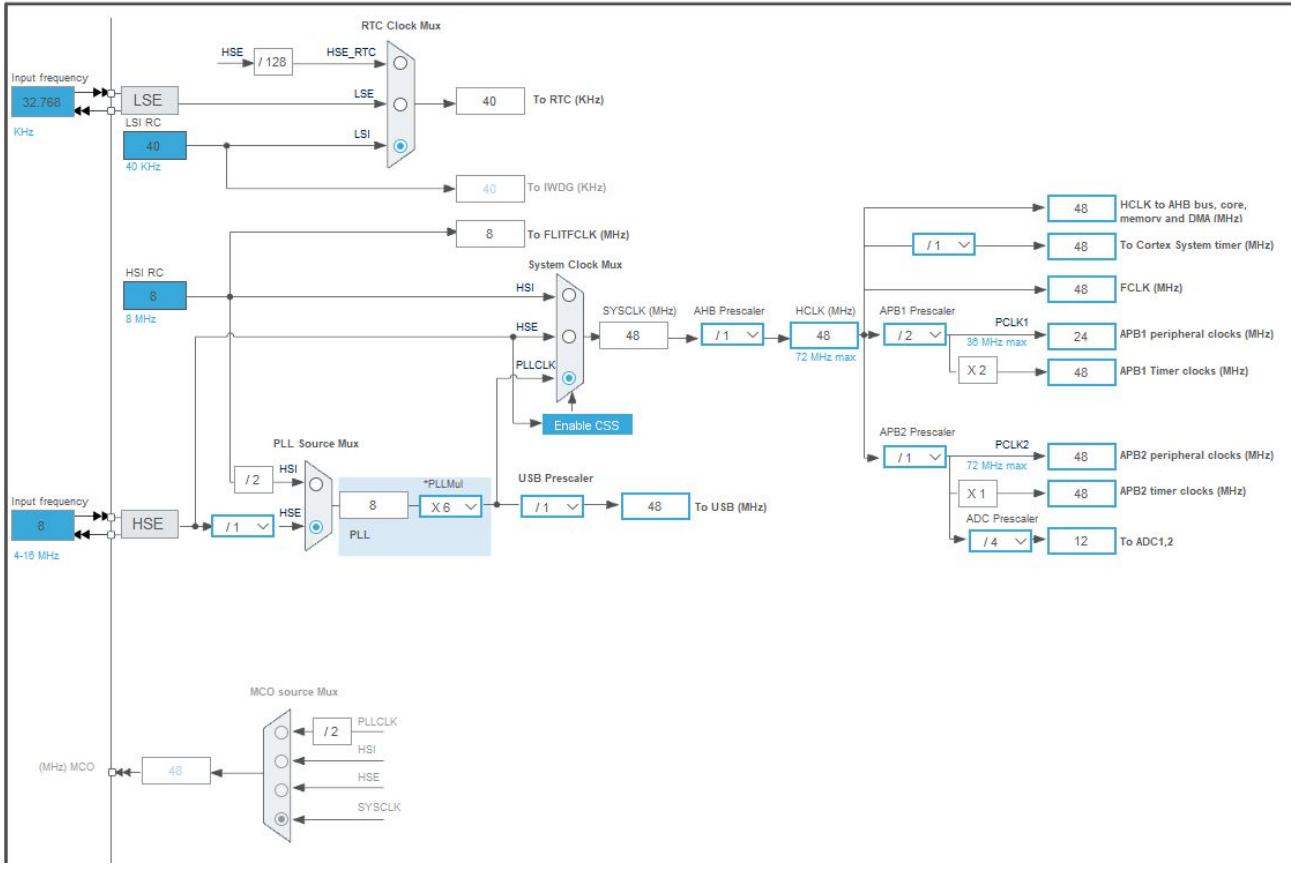
https://github.com/FeruMaga/SistemasEmbarcados2DataLogger.git

Configuração do STM32CubeIDE

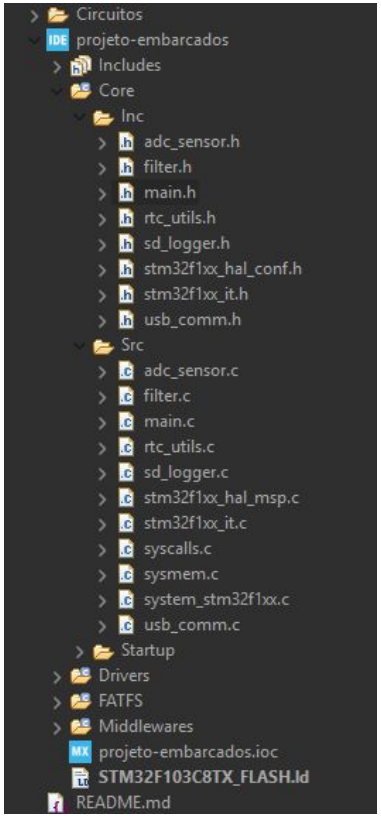




Clock Configuration



Divisão de Arquivos por Responsabilidades



main.c: Contém a função main( ) que inicializa todos os periféricos, gerencia o loop infinito e orquestra as chamadas para outras funções do sistema.

filter.h/.c: Implementam o filtro digital Butterworth passa-baixa de 2ª ordem para suavizar as leituras dos sensores (Tensão, Corrente, Temperatura, Massa) e reduzir ruídos, fornecendo dados mais estáveis para o sistema.

rtc_utils.h/.c: Define e implementa funções utilitárias para o uso do Real-Time Clock (RTC), permitindo obter e formatar a data e hora atuais.

adc_sensor.h/.c: Define a interface e implementa as funções para leitura e processamento dos sinais analógicos vindos dos sensores via ADC.

sd_logger.h/.c: Define a interface e implementa as funções para a inicialização e escrita de dados no cartão SD, utilizando o sistema de arquivos FATFS.

usb_comm.h/.c: Define a interface e implementa as funções para comunicação serial via USB, permitindo a troca de dados entre o microcontrolador e um computador.

Main

inicializa o hardware e os módulos de software do sistema. Em seu loop infinito, ela coleta dados de sensores (ADC), registra a hora (RTC) e escreve essas informações formatadas no cartão SD. Também gerencia erros do sistema.

```c
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_DMA_Init();
  MX_ADC1_Init();
  MX_I2C1_Init();
  MX_SPI2_Init();
  MX_RTC_Init();
  MX_FATFS_Init();
  MX_USB_DEVICE_Init();
  /* USER CODE BEGIN 2 */
  ADC_Sensor_Init(&hadc1);
    SD_Logger_Init(&hspi2);
    USB_Init(&hpcd_USB_FS);
    ADC_Sensor_StartConversion();
  /* USER CODE END 2 */
```

Início

```c
while (1) // Loop infinito: o código dentro deste bloco será executado repetidamente
 {
   /* USER CODE END WHILE */

   /* USER CODE BEGIN 3 */
    // Marcadores de código gerados automaticamente para áreas de usuário
    uint32_t now = HAL_GetTick(); // Obtém o tempo atual em milissegundos desde o boot do microcontrolador

       // Verifica se o intervalo de tempo para log de dados foi atingido
       if (now - lastLogTime >= LOG_INTERVAL_MS) {
           lastLogTime = now; // Atualiza a variável 'lastLogTime' com o tempo atual para a próxima checagem

           RTC_TimeTypeDef time; // Estrutura para armazenar a hora do RTC
           RTC_DateTypeDef date; // Estrutura para armazenar a data do RTC
           RTC_GetDateTime(&hrtc, &time, &date); // Obtém a hora e a data atuais do módulo RTC
           RTC_FormatDateTime(&time, &date, timestamp, sizeof(timestamp)); // Formata a hora e a data em uma string

           // Formata os dados do sensor e o carimbo de tempo em uma única string para o log
           snprintf(logBuffer, sizeof(logBuffer),
                    "%s;T1=%.2fV;C1=%.2fA;TEMP1=%.2fC;MASSA1=%.2fkg;T2=%.2fV;C2=%.2fA;TEMP2=%.2fC;MASSA2=%.2fkg\r\n",
                    timestamp,
                    ADC_Sensor_GetTensao1(),
                    ADC_Sensor_GetCorrente1(),
                    ADC_Sensor_GetTemperatura1(),
                    ADC_Sensor_GetMassa1(),
                    ADC_Sensor_GetTensao2(),
                    ADC_Sensor_GetCorrente2(),
                    ADC_Sensor_GetTemperatura2(),
                    ADC_Sensor_GetMassa2());

           SD_Status_t sd_write_status; // Variável para armazenar o status da escrita no SD

           // Verifica a flag global 'g_sd_card_full' ANTES de tentar escrever.
           // Se o cartão já foi marcado como cheio/com problema, evitamos a chamada da função de escrita.
           if (!g_sd_card_full) {
               sd_write_status = SD_Logger_WriteData(logBuffer); // Tenta escrever a string de log no cartão SD
           }
       }
   }
```

RTC

rtc_utils.h

Interface para o Real-Time Clock (RTC). Ele inclui funções para obter, definir e formatar a hora e a data, permitindo ao sistema gerenciar carimbos de tempo.

```c
/*
 * rtc_utils.h
 *
 *  Created on: Jun 1, 2025
 *      Author: FeruMaga
 */

#ifndef INC_RTC_UTILS_H_
#define INC_RTC_UTILS_H_


#include "main.h"
#include <stdint.h>
#include <stdio.h>

// Obtém a hora e a data atuais do RTC. Armazena os valores nas estruturas 'sTime' e 'sDate'.
void RTC_GetDateTime(RTC_HandleTypeDef* hrtc, RTC_TimeTypeDef* sTime, RTC_DateTypeDef* sDate);

// Define uma nova hora e data para o RTC. Permite configurar o RTC com um valor específico.
void RTC_SetDateTime(RTC_HandleTypeDef* hrtc, uint8_t hours, uint8_t minutes, uint8_t seconds, uint8_t day, uint8_t month, u

// Formata a hora e a data (obtidas das estruturas 'sTime' e 'sDate') em uma string de texto legível.
void RTC_FormatDateTime(RTC_TimeTypeDef* sTime, RTC_DateTypeDef* sDate, char* buffer, size_t bufferSize);


#endif /* INC_RTC_UTILS_H_ */
```

rtc_utils.c

implementa as funções para obter, definir e formatar a hora e a data do Real-Time Clock (RTC) do sistema. Ele permite que o microcontrolador acesse e gerencie informações de tempo de forma prática para registro ou exibição.

```c
#include "rtc_utils.h"

void RTC_GetDateTime(RTC_HandleTypeDef* hrtc, RTC_TimeTypeDef* sTime, RTC_DateTypeDef* sDate) {
    HAL_RTC_GetTime(hrtc, sTime, RTC_FORMAT_BIN);
    HAL_RTC_GetDate(hrtc, sDate, RTC_FORMAT_BIN);
}

// Implementação da função para obter a hora e a data atuais do RTC.
void RTC_SetDateTime(RTC_HandleTypeDef* hrtc, uint8_t hours, uint8_t minutes, uint8_t seconds, uint8_t day, uint8_t month, uint16_t year) {
    RTC_TimeTypeDef sTime = {0};
    RTC_DateTypeDef sDate = {0};

    sTime.Hours = hours;
    sTime.Minutes = minutes;
    sTime.Seconds = seconds;
    sDate.Date = day;
    sDate.Month = month;
    sDate.Year = year - 2000;

    HAL_RTC_SetTime(hrtc, &sTime, RTC_FORMAT_BIN);
    HAL_RTC_SetDate(hrtc, &sDate, RTC_FORMAT_BIN);
}

// Implementação da função para definir a hora e a data no RTC.
void RTC_FormatDateTime(RTC_TimeTypeDef* sTime, RTC_DateTypeDef* sDate, char* buffer, size_t bufferSize) {
    snprintf(buffer, bufferSize, "%02d/%02d/20%02d %02d:%02d:%02d",
             sDate->Date, sDate->Month, sDate->Year,
             sTime->Hours, sTime->Minutes, sTime->Seconds);
}
```

Filtro

filter.h

Define a interface para um filtro digital Butterworth passa-baixa de 2ª ordem. Ele especifica as estruturas para coeficientes e estado do filtro, além das funções para inicializar e aplicar a filtragem de amostras.

```c
/*
 * filter.h
 *
 *  Created on: Jun 10, 2025
 *      Author: FeruMaga
 */

#ifndef INC_FILTER_H_
#define INC_FILTER_H_

// Estrutura para armazenar os coeficientes de um filtro Butterworth de 2ª ordem.
typedef struct {
    double B0, B1, B2;
    double A1, A2;
} FilterCoefficients_t;

// Estrutura para armazenar as variáveis de estado de um filtro Butterworth de 2ª ordem.
typedef struct {
    double x1, x2; // Amostras de entrada anteriores (x[n-1], x[n-2])
    double y1, y2; // Amostras de saída anteriores (y[n-1], y[n-2])
} Butter2State_t;

// Protótipos das funções do filtro.


// Inicializa o estado de um filtro Butterworth de 2ª ordem.
void butter2_init(Butter2State_t* s);

// Aplica o filtro Butterworth de 2ª ordem a uma nova amostra.
double butter2_apply(Butter2State_t* s, double x, const FilterCoefficients_t* coeffs);

#endif /* INC_FILTER_H_ */
#endif /* INC_FILTER_H_ */
```

filter.c

inicializa o filtro Butterworth para zero e aplica a filtragem digital a novas amostras. Ele usa os coeficientes para calcular a saída e atualiza os estados internos para o próximo processamento.

```c
/*
 * filter.c
 *
 *  Created on: Jun 12, 2025
 *      Author: FeruMaga
 */
#include "filter.h"

// Implementação da função para inicializar o estado do filtro.
void butter2_init(Butter2State_t* s) {
    s->x1 = s->x2 = s->y1 = s->y2 = 0.0;
}

// Implementação da função para aplicar o filtro a uma nova amostra.
void butter2_apply(Butter2State_t* s, double x, const FilterCoefficients_t* coeffs) {
    // A equação de diferença do filtro digital IIR de 2ª ordem.
    // y[n] = B0*x[n] + B1*x[n-1] + B2*x[n-2] - A1*y[n-1] - A2*y[n-2]
    const double y = coeffs->B0 * x + coeffs->B1 * s->x1 + coeffs->B2 * s->x2
                   - coeffs->A1 * s->y1 - coeffs->A2 * s->y2;

    // Atualiza os estados de atraso para a próxima amostra.
    s->x2 = s->x1; // x[n-2] <- x[n-1]
    s->x1 = x;     // x[n-1] <- x[n]
    s->y2 = s->y1; // y[n-2] <- y[n-1]
    s->y1 = y;     // y[n-1] <- y[n]

    return y; // Retorna o novo valor filtrado.
}
```

USB Communication

usb_comm.h

Interface para a comunicação USB CDC (Virtual Serial Port). Ele estabelece as funções para inicializar USB, receber comandos (como calibração) e enviar dados para um host, facilitando a interação e configuração do dispositivo.

```c
/*
 * usb.h
 *
 *  Created on: Jun 12, 2025
 *      Author: FeruMaga
 */

#ifndef INC_USB_COMM_H_
#define INC_USB_COMM_H_

// Define o tamanho do buffer de recepção USB.
#define USB_RX_BUFFER_SIZE 256

void USB_Init(PCD_HandleTypeDef* hpcd);
// Inicializa o periférico USB Device (PCD) do STM32.

void USB_Receive(uint8_t* Buf, uint32_t Len);
// Função callback chamada quando dados são recebidos via USB.
// Ela copia e processa os dados de calibração.

bool USB_ProcessCalibrationCommand(uint8_t* data, uint32_t len, CalibrationData_t* cal
// Analisa uma string de comando para extrair e aplicar parâmetros de calibração.

void USB_Send(char* str);
// Envia uma string de texto para o host via USB.



#endif /* INC_USB_COMM_H_ */
```

usb_comm.c

Este módulo inicializa a comunicação USB e processa comandos de calibração recebidos. Ele permite ao usuário enviar parâmetros para ajustar os sensores via USB e envia feedback.

```c
#include "usb_comm.h"
#include "usbd_cdc_if.h"

extern CalibrationData_t gCalibrationData;

void USB_Init(PCD_HandleTypeDef* hpcd) {
    // Inicia o periférico USB Device (PCD).
    // Prepara o hardware USB para comunicação.
    HAL_PCD_Start(hpcd);
}

void USB_Receive(uint8_t* Buf, uint32_t Len) {
    char rx_buffer_copy[USB_RX_BUFFER_SIZE]; // Buffer local para copiar os dados recebidos.

    // Garante que o comprimento dos dados não exceda o tamanho do buffer local.
    if (Len >= USB_RX_BUFFER_SIZE) {
        Len = USB_RX_BUFFER_SIZE - 1; // Deixa espaço para o terminador nulo.
    }
    // Copia os dados recebidos do buffer da USB para o buffer local.
    memcpy(rx_buffer_copy, Buf, Len);
    // Adiciona o terminador nulo para tratar a cópia como uma string C.
    rx_buffer_copy[Len] = '\0';

    // Tenta processar a string como um comando de calibração.
    if (USB_ProcessCalibrationCommand((uint8_t*)rx_buffer_copy, Len, &gCalibrationData)) {
        // Se a calibração foi bem-sucedida, marca o sistema como calibrado.
        gCalibrationData.isCalibrated = true;
        // Envia uma mensagem de sucesso de volta ao host.
        USB_Send("Calibracao aplicada com sucesso!\r\n");
    } else {
        // Se o comando de calibração for inválido.
        if (gCalibrationData.isCalibrated) {
            // Se já havia calibração, informa que o comando foi inválido, mas mantém os dados existentes.
            USB_Send("Comando de calibracao invalido. Usando valores ja calibrados.\r\n");
        } else {
            // Se não havia calibração e o comando foi inválido, informa que usará valores padrão.
            USB_Send("Calibracao invalida ou ausente. Usando valores mocados.\r\n");
        }
    }
}
```

```c
bool USB_ProcessCalibrationCommand(uint8_t* data, uint32_t len, CalibrationData_t* calData) {
    char *token;       // Ponteiro para cada pedaço da string separada por vírgula.
    // Aloca memória dinamicamente para uma cópia da string de entrada,
    // pois strtok_r modifica a string original.
    char *data_copy = (char*)malloc(len + 1);
    if (data_copy == NULL) return false; // Retorna false se a alocação falhar.
    memcpy(data_copy, data, len);        // Copia os dados.
    data_copy[len] = '\0';               // Garante a terminação nula.

    char *rest = data_copy;              // Ponteiro para o restante da string a ser analisada.
    bool receivedAny = false;            // Flag para verificar se algum parâmetro foi lido.

    // Itera sobre a string, dividindo-a em tokens (partes separadas por vírgulas).
    while ((token = strtok_r(rest, ",", &rest)) != NULL) {
        // Tenta parsear cada token para um parâmetro de calibração específico usando sscanf.
        // O retorno '1' de sscanf indica que um valor foi lido e atribuído com sucesso.
        if (sscanf(token, "T1_OFF=%f", &calData->tensao1_offset) == 1) receivedAny = true;
        else if (sscanf(token, "T1_COEFF=%f", &calData->tensao1_coeff) == 1) receivedAny = true;
        else if (sscanf(token, "T2_OFF=%f", &calData->tensao2_offset) == 1) receivedAny = true;
        else if (sscanf(token, "T2_COEFF=%f", &calData->tensao2_coeff) == 1) receivedAny = true;
        else if (sscanf(token, "C1_OFF=%f", &calData->corrente1_offset) == 1) receivedAny = true;
        else if (sscanf(token, "C1_COEFF=%f", &calData->corrente1_coeff) == 1) receivedAny = true;
        else if (sscanf(token, "C2_OFF=%f", &calData->corrente2_offset) == 1) receivedAny = true;
        else if (sscanf(token, "C2_COEFF=%f", &calData->corrente2_coeff) == 1) receivedAny = true;
        else if (sscanf(token, "TEMP1_OFF=%f", &calData->temperatura1_offset) == 1) receivedAny = true;
        else if (sscanf(token, "TEMP1_COEFF=%f", &calData->temperatura1_coeff) == 1) receivedAny = true;
        else if (sscanf(token, "TEMP2_OFF=%f", &calData->temperatura2_offset) == 1) receivedAny = true;
        else if (sscanf(token, "TEMP2_COEFF=%f", &calData->temperatura2_coeff) == 1) receivedAny = true;
        else if (sscanf(token, "MASSA1_OFF=%f", &calData->massa1_offset) == 1) receivedAny = true;
        else if (sscanf(token, "MASSA1_COEFF=%f", &calData->massa1_coeff) == 1) receivedAny = true;
        else if (sscanf(token, "MASSA2_OFF=%f", &calData->massa2_offset) == 1) receivedAny = true;
        else if (sscanf(token, "MASSA2_COEFF=%f", &calData->massa2_coeff) == 1) receivedAny = true;
    }

    free(data_copy); // Libera a memória alocada.
    return receivedAny; // Retorna verdadeiro se algum parâmetro foi lido.
}

void USB_Send(char* str) {
    // Envia a string para o host via USB CDC (Virtual Serial Port).
    CDC_Transmit_FS((uint8_t*)str, strlen(str));
}
```

SD

sd_logger.h

interface para o módulo de log no cartão SD, incluindo tipos de status de operação. Ele declara funções para inicializar a comunicação com o SD e para escrever dados no arquivo de log, além de uma flag de "cartão cheio".

```
/*
 * sd_logger.h
 *
 *  Created on: Jun 12, 2025
 *      Author: FeruMaga
 */

#ifndef INC_SD_LOGGER_H_
#define INC_SD_LOGGER_H_


#include "main.h"
#include "fatfs.h"
#include "string.h"
#include "usb_comm.h"


typedef enum {
    SD_OK       = 0,  // Status: Operação de cartão SD bem-sucedida.
    SD_OPEN_ERR = 1,  // Status: Erro ao abrir ou criar o arquivo no cartão SD.
    SD_WRITE_ERR = 2  // Status: Erro ao escrever dados no arquivo do cartão SD.
} SD_Status_t; // Definição de um tipo enumerado para representar o status das operações do cartão SD.

// Prototipo da função que escreve dados no cartão SD.
SD_Status_t SD_Logger_WriteData(char* data);

// Adicione uma variável global (extern) para o estado do SD
extern bool g_sd_card_full;

// Define uma macro para o nome do arquivo de log no cartão SD.
#define LOG_FILE_NAME "log.txt"

// Prototipo da função de inicialização do módulo de log no cartão SD.
void SD_Logger_Init(SPI_HandleTypeDef* hspi);

// Prototipo original da função de escrita de dados no cartão SD.
void SD_Logger_WriteData(char* data);

#endif /* INC_SD_LOGGER_H_ */
```

sd_logger.c

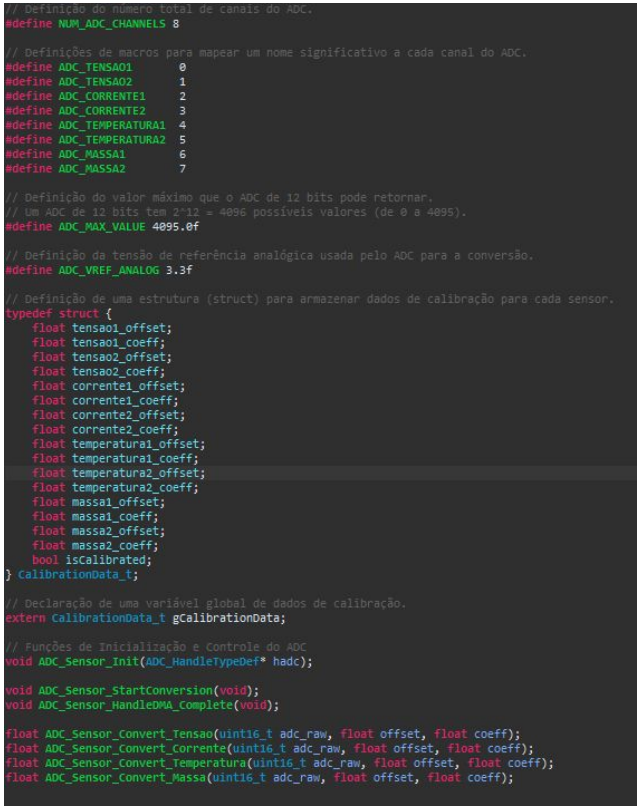
Implementa o logging em cartão SD, gerenciando a inicialização e montagem do sistema de arquivos FATFS. Ele permite escrever dados em um arquivo de log, tratando erros como cartão cheio ou problemas de escrita para garantir a robustez do sistema.

```c
// Objeto de sistema de arquivos (File System Object) para a biblioteca FATFS.
FATFS fs;
// Objeto de arquivo (File Object) para a biblioteca FATFS.
FIL file;

// Variável para armazenar o número de bytes escritos em uma operação de escrita.
UINT byteswritten;
// Inicializa a flag que indica se o cartão SD está cheio.
bool g_sd_card_full = false;

void SD_Logger_Init(SPI_HandleTypeDef* hspi) {
    // Redefine a flag de cartão cheio para falso na inicialização.
    g_sd_card_full = false;

    // Tenta vincular o driver de hardware do cartão SD (definido como USER_Driver)
    if (FATFS_LinkDriver(&USER_Driver, USERPath) != 0) {
        // Se houver um erro ao vincular o driver, envia uma mensagem via USB
        // e chama o Error_Handler do sistema.
        USB_Send("Erro ao linkar driver FATFS\r\n");
        Error_Handler();
    }

    // Tenta montar o sistema de arquivos FATFS no volume lógico.
    // O "" indica o volume padrão, e "1" indica que a montagem deve ser forçada (útil em alguns cenários).
    FRESULT fr_mount = f_mount(&fs, "", 1);
    if (fr_mount != FR_OK) {
        // Se a montagem falhar, o cartão SD pode estar ausente, não formatado ou corrompido.
        USB_Send("Erro ao montar SD\r\n");
        // Em caso de falha na montagem, assume-se que o cartão está inacessível para escrita.
        // A flag 'g_sd_card_full' é definida como true para evitar tentativas futuras de escrita.
        g_sd_card_full = true;
        // Chama o Error_Handler do sistema para tratamento de erro crítico.
        Error_Handler(); // Dependendo da sua estratégia de erro, pode parar aqui.
    }
}
```

```c
// Implementação da função de escrita de dados no cartão SD.
// Retorna um status indicando o resultado da operação.
SD_Status_t SD_Logger_WriteData(char* data) {
    FRESULT fr; // Variável para armazenar o resultado das operações da biblioteca FATFS.

    // Verifica se a flag de cartão cheio está ativa.
    // Se estiver, evita novas tentativas de escrita para otimizar o desempenho.
    if (g_sd_card_full) {
        return SD_WRITE_ERR; // Retorna um erro indicando que o cartão não pode ser escrito.
                             // Poderia ser um novo status como SD_ALREADY_FULL para maior clareza.
    }

    // Tenta abrir o arquivo de log (LOG_FILE_NAME).
    // FA_OPEN_APPEND: Abre o arquivo para escrita, posicionando o cursor no final.
    // Se o arquivo não existir, ele será criado.
    // FA_WRITE: Habilita a permissão de escrita.
    fr = f_open(&file, LOG_FILE_NAME, FA_OPEN_APPEND | FA_WRITE);

    // Verifica se a operação de abertura do arquivo foi bem-sucedida.
    if (fr == FR_OK) {
        // Se o arquivo foi aberto com sucesso, tenta escrever os dados.
        // 'data': Ponteiro para os dados a serem escritos.
        // 'strlen(data)': Número de bytes a serem escritos (tamanho da string).
        // '&bytesWritten': Ponteiro para a variável onde o número de bytes realmente escritos será armazenado.
        fr = f_write(&file, data, strlen(data), &bytesWritten);

        // Verifica se a escrita foi bem-sucedida (fr == FR_OK) E se todos os bytes foram escritos.
        if (fr == FR_OK && bytesWritten == strlen(data)) {
            f_close(&file); // Fecha o arquivo para salvar as alterações e liberar recursos.
            return SD_OK; // Retorna status de sucesso.
        } else {
            // Se houve um erro durante a escrita ou se nem todos os bytes foram escritos.
            // Isso pode ocorrer por disco cheio, erro físico, ou outros problemas.
            USB_Send("Erro de escrita no SD ou disco cheio.\r\n");
            // Define a flag de cartão cheio como true.
            // Isso é uma medida preventiva para evitar novas tentativas de escrita que falhariam.
            g_sd_card_full = true;
            // Tenta fechar o arquivo mesmo com erro, para liberar recursos.
            f_close(&file);
            return SD_WRITE_ERR; // Retorna status de erro de escrita.
        }
    } else {
        // Se houver um erro ao tentar abrir o arquivo.
        // Isso pode ser causado por: cartão removido, cartão corrompido,
        // ou o próprio cartão SD estar fisicamente cheio (FATFS pode retornar FR_DENIED ou FR_DISK_ERR).
        USB_Send("Erro ao abrir arquivo de log no SD. (Cartão cheio ou problema?)\r\n");
        // Assume que o cartão está cheio ou inacessível para escrita e define a flag.
        g_sd_card_full = true;
        return SD_OPEN_ERR; // Retorna status de erro de abertura.
    }
}
```

Sensores ADC

adc_sensor.h Interface do módulo ADC, mapeando canais e valores de referência. Ele especifica a estrutura de dados de calibração para múltiplos sensores e declara funções para inicializar, converter e obter leituras dos sensores.



```c
// Definição do número total de canais do ADC.
#define NUM_ADC_CHANNELS 8

// Definições de macros para mapear um nome significativo a cada canal do ADC.
#define ADC_TENSAO1       0
#define ADC_TENSAO2       1
#define ADC_CORRENTE1     2
#define ADC_CORRENTE2     3
#define ADC_TEMPERATURA1  4
#define ADC_TEMPERATURA2  5
#define ADC_MASSA1        6
#define ADC_MASSA2        7

// Definição do valor máximo que o ADC de 12 bits pode retornar.
// Um ADC de 12 bits tem 2^12 = 4096 possíveis valores (de 0 a 4095).
#define ADC_MAX_VALUE 4095.0f

// Definição da tensão de referência analógica usada pelo ADC para a conversão.
#define ADC_VREF_ANALOG 3.3f

// Definição de uma estrutura (struct) para armazenar dados de calibração para cada sensor.
typedef struct {
    float tensao1_offset;
    float tensao1_coeff;
    float tensao2_offset;
    float tensao2_coeff;
    float corrente1_offset;
    float corrente1_coeff;
    float corrente2_offset;
    float corrente2_coeff;
    float temperatura1_offset;
    float temperatura1_coeff;
    float temperatura2_offset;
    float temperatura2_coeff;
    float massa1_offset;
    float massa1_coeff;
    float massa2_offset;
    float massa2_coeff;
    bool isCalibrated;
} CalibrationData_t;

// Declaração de uma variável global de dados de calibração.
extern CalibrationData_t gCalibrationData;

// Funções de Inicialização e Controle do ADC
void ADC_Sensor_Init(ADC_HandleTypeDef* hadc);

void ADC_Sensor_StartConversion(void);
void ADC_Sensor_HandleDMA_Complete(void);

float ADC_Sensor_Convert_Tensao(uint16_t adc_raw, float offset, float coeff);
float ADC_Sensor_Convert_Corrente(uint16_t adc_raw, float offset, float coeff);
float ADC_Sensor_Convert_Temperatura(uint16_t adc_raw, float offset, float coeff);
float ADC_Sensor_Convert_Massa(uint16_t adc_raw, float offset, float coeff);
```

```c
float ADC_Sensor_GetTensao1(void);
float ADC_Sensor_GetTensao2(void);
float ADC_Sensor_GetCorrente1(void);
float ADC_Sensor_GetCorrente2(void);
float ADC_Sensor_GetTemperatura1(void);
float ADC_Sensor_GetTemperatura2(void);
float ADC_Sensor_GetMassa1(void);
float ADC_Sensor_GetMassa2(void);
#endif /* INC_ADC_SENSOR_H_ */
```

adc_sensors.c

inicializa o ADC e dados de calibração, gerencia a conversão de leituras de sensores via DMA. Ele aplica filtros passa-baixa nas leituras brutas, convertendo-as para unidades de engenharia para uso do sistema.

```
CalibrationData_t gCalibrationData;

static uint16_t adcBuffer[NUM_ADC_CHANNELS];
static ADC_HandleTypeDef* pAdcHandle;

// Variáveis estáticas para armazenar os valores dos sensores em unidades de e
static float tensao1Eng = 0.0f;
static float tensao2Eng = 0.0f;
static float corrente1Eng = 0.0f;
static float corrente2Eng = 0.0f;
static float temperatura1Eng = 0.0f;
static float temperatura2Eng = 0.0f;
static float massa1Eng = 0.0f;
static float massa2Eng = 0.0f;

// --- Instâncias de estado do filtro Butterworth para CADA SENSOR ---
static Butter2State_t tensao1FilterState;
static Butter2State_t tensao2FilterState;
static Butter2State_t corrente1FilterState;
static Butter2State_t corrente2FilterState;
```

```
// --- Instâncias de coeficientes de filtro para CADA SENSOR ---
static const FilterCoefficients_t tensaoFilterCoeffs = {
    .B0 = 1.95669352e-03, .B1 = 3.91338703e-03, .B2 = 1.95669352e-03,
    .A1 = -1.90562624e+00, .A2 = 9.13453273e-01
};
static const FilterCoefficients_t tensao2FilterCoeffs = {
    .B0 = 1.95669352e-03, .B1 = 3.91338703e-03, .B2 = 1.95669352e-03,
    .A1 = -1.90562624e+00, .A2 = 9.13453273e-01
};

static const FilterCoefficients_t correnteFilterCoeffs = {
    .B0 = 1.95669352e-03, .B1 = 3.91338703e-03, .B2 = 1.95669352e-03,
    .A1 = -1.90562624e+00, .A2 = 9.13453273e-01
};
static const FilterCoefficients_t corrente2FilterCoeffs = {
    .B0 = 1.95669352e-03, .B1 = 3.91338703e-03, .B2 = 1.95669352e-03,
    .A1 = -1.90562624e+00, .A2 = 9.13453273e-01
};

static const FilterCoefficients_t temperaturaFilterCoeffs = {
    .B0 = 1.57912440e-07, .B1 = 3.15824879e-07, .B2 = 1.57912440e-07,
    .A1 = -1.99842095e+00, .A2 = 9.98421045e-01
};
static const FilterCoefficients_t temperatura2FilterCoeffs = {
    .B0 = 1.57912440e-07, .B1 = 3.15824879e-07, .B2 = 1.57912440e-07,
    .A1 = -1.99842095e+00, .A2 = 9.98421045e-01
};

static const FilterCoefficients_t massaFilterCoeffs = {
    .B0 = 3.94553258e-06, .B1 = 7.89106516e-06, .B2 = 3.94553258e-06,
    .A1 = -1.98418047e+00, .A2 = 9.84206979e-01
};
static const FilterCoefficients_t massa2FilterCoeffs = {
    .B0 = 3.94553258e-06, .B1 = 7.89106516e-06, .B2 = 3.94553258e-06,
    .A1 = -1.98418047e+00, .A2 = 9.84206979e-01
};

static bool firstAdcRead = true;
```

adc_sensors.c

```c
// Inicializa o módulo ADC_Sensor.
void ADC_Sensor_Init(ADC_HandleTypeDef* hadc) {
    pAdcHandle = hadc;

    // --- Configurações de calibração (permanecem as mesmas) ---
    gCalibrationData.tensao1_offset = 2047.0f; gCalibrationData.tensao1_coeff = 0.1075f;
    gCalibrationData.tensao2_offset = 2047.0f; gCalibrationData.tensao2_coeff = 0.1075f;
    gCalibrationData.corrente1_offset = 2047.0f; gCalibrationData.corrente1_coeff = 0.002442f;
    gCalibrationData.corrente2_offset = 2047.0f; gCalibrationData.corrente2_coeff = 0.002442f;
    gCalibrationData.temperatura1_offset = 2047.0f; gCalibrationData.temperatura1_coeff = 0.04884f;
    gCalibrationData.temperatura2_offset = 2047.0f; gCalibrationData.temperatura2_coeff = 0.04884f;
    gCalibrationData.massa1_offset = 2047.0f; gCalibrationData.massa1_coeff = 0.24420f;
    gCalibrationData.massa2_offset = 2047.0f; gCalibrationData.massa2_coeff = 0.24420f;
    gCalibrationData.isCalibrated = false;

    // --- Inicializa os estados dos filtros Butterworth para TODOS os sensores ---
    butter2_init(&tensao1FilterState);
    butter2_init(&tensao2FilterState);
    butter2_init(&corrente1FilterState);
    butter2_init(&corrente2FilterState);
    butter2_init(&temperatura1FilterState);
    butter2_init(&temperatura2FilterState);
    butter2_init(&massa1FilterState);
    butter2_init(&massa2FilterState);

    firstAdcRead = true;
}

// Inicia a conversão do ADC usando DMA.
void ADC_Sensor_StartConversion(void) {
    HAL_ADC_Start_DMA(pAdcHandle, (uint32_t*)adcBuffer, NUM_ADC_CHANNELS);
}
```

```c
// Função de callback (chamada de retorno) invocada quando a transferência DMA do ADC é concluída.
void ADC_Sensor_HandleDMA_Complete(void) {
    // Converte as leituras brutas do ADC para unidades de engenharia.
    float newTensao1 = ADC_Sensor_Convert_Tensao(adcBuffer[ADC_TENSAO1], gCalibrationData.tensao1_offset, gCalibrationData.tensao1_coeff);
    float newTensao2 = ADC_Sensor_Convert_Tensao(adcBuffer[ADC_TENSAO2], gCalibrationData.tensao2_offset, gCalibrationData.tensao2_coeff);
    float newCorrente1 = ADC_Sensor_Convert_Corrente(adcBuffer[ADC_CORRENTE1], gCalibrationData.corrente1_offset, gCalibrationData.corrente1_coe
    float newCorrente2 = ADC_Sensor_Convert_Corrente(adcBuffer[ADC_CORRENTE2], gCalibrationData.corrente2_offset, gCalibrationData.corrente2_coe
    float newTemperatura1 = ADC_Sensor_Convert_Temperatura(adcBuffer[ADC_TEMPERATURA1], gCalibrationData.temperatura1_offset, gCalibrationData.t
    float newTemperatura2 = ADC_Sensor_Convert_Temperatura(adcBuffer[ADC_TEMPERATURA2], gCalibrationData.temperatura2_offset, gCalibrationData.t
    float newMassa1 = ADC_Sensor_Convert_Massa(adcBuffer[ADC_MASSA1], gCalibrationData.massa1_offset, gCalibrationData.massa1_coeff);
    float newMassa2 = ADC_Sensor_Convert_Massa(adcBuffer[ADC_MASSA2], gCalibrationData.massa2_offset, gCalibrationData.massa2_coeff);

    // --- Aplica o filtro Butterworth de 2ª ordem a TODAS as leituras dos sensores ---
    // Usando os coeficientes específicos para cada tipo de sensor.
    tensao1Eng = (float)butter2_apply(&tensao1FilterState, (double)newTensao1, &tensaoFilterCoeffs);
    tensao2Eng = (float)butter2_apply(&tensao2FilterState, (double)newTensao2, &tensao2FilterCoeffs); // Note: Assuming tensao2 also uses tensa
    corrente1Eng = (float)butter2_apply(&corrente1FilterState, (double)newCorrente1, &correnteFilterCoeffs);
    corrente2Eng = (float)butter2_apply(&corrente2FilterState, (double)newCorrente2, &corrente2FilterCoeffs); // Note: Assuming corrente2 also
    temperatura1Eng = (float)butter2_apply(&temperatura1FilterState, (double)newTemperatura1, &temperaturaFilterCoeffs);
    temperatura2Eng = (float)butter2_apply(&temperatura2FilterState, (double)newTemperatura2, &temperatura2FilterCoeffs); // Note: Assuming tem
    massa1Eng = (float)butter2_apply(&massa1FilterState, (double)newMassa1, &massaFilterCoeffs);
    massa2Eng = (float)butter2_apply(&massa2FilterState, (double)newMassa2, &massa2FilterCoeffs); // Note: Assuming massa2 also uses massaFilte

    firstAdcRead = false; // Se a flag for usada para outras finalidades, mantenha e atualize.


    // Reinicia a conversão do ADC via DMA imediatamente após o processamento.
    HAL_ADC_Start_DMA(pAdcHandle, (uint32_t*)adcBuffer, NUM_ADC_CHANNELS);
}

// --- Funções de Conversão (permanecem as mesmas) ---
float ADC_Sensor_Convert_Tensao(uint16_t adc_raw, float offset, float coeff) {
    return ((float)adc_raw - offset) * coeff;
}

float ADC_Sensor_Convert_Corrente(uint16_t adc_raw, float offset, float coeff) {
    return ((float)adc_raw - offset) * coeff;
}

float ADC_Sensor_Convert_Temperatura(uint16_t adc_raw, float offset, float coeff) {
    return ((float)adc_raw - offset) * coeff;
}

float ADC_Sensor_Convert_Massa(uint16_t adc_raw, float offset, float coeff) {
    return ((float)adc_raw - offset) * coeff;
}

// --- Funções Getters (permanecem as mesmas) ---
float ADC_Sensor_GetTensao1(void) { return tensao1Eng; }
float ADC_Sensor_GetTensao2(void) { return tensao2Eng; }
float ADC_Sensor_GetCorrente1(void) { return corrente1Eng; }
float ADC_Sensor_GetCorrente2(void) { return corrente2Eng; }
float ADC_Sensor_GetTemperatura1(void) { return temperatura1Eng; }
float ADC_Sensor_GetTemperatura2(void) { return temperatura2Eng; }
float ADC_Sensor_GetMassa1(void) { return massa1Eng; }
float ADC_Sensor_GetMassa2(void) { return massa2Eng; }
```

adc_sensor.c

```c
// Inicia a conversão do ADC usando DMA.
// O DMA transfere os resultados da conversão diretamente para o 'adcBuffer'.
void ADC_Sensor_StartConversion(void) {
    // Chama a função da HAL para iniciar a conversão do ADC em modo DMA.
    HAL_ADC_Start_DMA(pAdcHandle, (uint32_t*)adcBuffer, NUM_ADC_CHANNELS);
}


// Função de callback (chamada de retorno) invocada quando a transferência DMA do ADC é concluída.
void ADC_Sensor_HandleDMA_Complete(void) {
    // Converte as leituras brutas do ADC para unidades de engenharia usando os offsets e coeficientes.
    // adcBuffer[ADC_TENSAO1] acessa a leitura bruta do canal de Tensão 1, e assim por diante.
    float newTensao1 = ADC_Sensor_Convert_Tensao(adcBuffer[ADC_TENSAO1], gCalibrationData.tensao1_offset, gCalibrationData.tensao1_coeff);
    float newTensao2 = ADC_Sensor_Convert_Tensao(adcBuffer[ADC_TENSAO2], gCalibrationData.tensao2_offset, gCalibrationData.tensao2_coeff);
    float newCorrente1 = ADC_Sensor_Convert_Corrente(adcBuffer[ADC_CORRENTE1], gCalibrationData.corrente1_offset, gCalibrationData.corrente1_coeff);
    float newCorrente2 = ADC_Sensor_Convert_Corrente(adcBuffer[ADC_CORRENTE2], gCalibrationData.corrente2_offset, gCalibrationData.corrente2_coeff);
    float newTemperatura1 = ADC_Sensor_Convert_Temperatura(adcBuffer[ADC_TEMPERATURA1], gCalibrationData.temperatura1_offset, gCalibrationData.temperatura1_coeff);
    float newTemperatura2 = ADC_Sensor_Convert_Temperatura(adcBuffer[ADC_TEMPERATURA2], gCalibrationData.temperatura2_offset, gCalibrationData.temperatura2_coeff);
    float newMassa1 = ADC_Sensor_Convert_Massa(adcBuffer[ADC_MASSA1], gCalibrationData.massa1_offset, gCalibrationData.massa1_coeff);
    float newMassa2 = ADC_Sensor_Convert_Massa(adcBuffer[ADC_MASSA2], gCalibrationData.massa2_offset, gCalibrationData.massa2_coeff);

    // Lógica de filtragem:
    // Se for a primeira leitura, os valores de engenharia são inicializados diretamente com as novas leituras.
    // Isso evita que o filtro comece de um valor zero abrupto, o que poderia causar um pico inicial.
    if (firstAdcRead) {
        tensao1Eng = newTensao1;
        tensao2Eng = newTensao2;
        corrente1Eng = newCorrente1;
        corrente2Eng = newCorrente2;
        temperatura1Eng = newTemperatura1;
        temperatura2Eng = newTemperatura2;
        massa1Eng = newMassa1;
        massa2Eng = newMassa2;
        firstAdcRead = false;
    } else {
        // Para leituras subsequentes, aplica um filtro passa-baixa (Lowpass) para suavizar os dados.
        // Filter_Lowpass(valor_anterior, novo_valor, alpha) calcula um novo valor filtrado.
        tensao1Eng = Filter_Lowpass(tensao1Eng, newTensao1, FILTER_ALPHA_TENSAO);
        tensao2Eng = Filter_Lowpass(tensao2Eng, newTensao2, FILTER_ALPHA_TENSAO);
        corrente1Eng = Filter_Lowpass(corrente1Eng, newCorrente1, FILTER_ALPHA_CORRENTE);
        corrente2Eng = Filter_Lowpass(corrente2Eng, newCorrente2, FILTER_ALPHA_CORRENTE);
        temperatura1Eng = Filter_Lowpass(temperatura1Eng, newTemperatura1, FILTER_ALPHA_TEMPERATURA);
        temperatura2Eng = Filter_Lowpass(temperatura2Eng, newTemperatura2, FILTER_ALPHA_TEMPERATURA);
        massa1Eng = Filter_Lowpass(massa1Eng, newMassa1, FILTER_ALPHA_MASSA);
        massa2Eng = Filter_Lowpass(massa2Eng, newMassa2, FILTER_ALPHA_MASSA);
    }

    // Reinicia a conversão do ADC via DMA imediatamente após o processamento.
    HAL_ADC_Start_DMA(pAdcHandle, (uint32_t*)adcBuffer, NUM_ADC_CHANNELS);
}


// Converte um valor bruto do ADC para tensão em Volts.
float ADC_Sensor_Convert_Tensao(uint16_t adc_raw, float offset, float coeff) {
    // Fórmula de conversão linear: (valor_bruto_ADC - offset) * coeficiente
    return ((float)adc_raw - offset) * coeff;
}
```

```c
// Converte um valor bruto do ADC para massa em Quilogramas.
float ADC_Sensor_Convert_Massa(uint16_t adc_raw, float offset, float coeff) {
    return ((float)adc_raw - offset) * coeff;
}

// Funções "getter" que retornam os valores atuais dos sensores em unidades de engenharia.
float ADC_Sensor_GetTensao1(void) { return tensao1Eng; }
float ADC_Sensor_GetTensao2(void) { return tensao2Eng; }
float ADC_Sensor_GetCorrente1(void) { return corrente1Eng; }
float ADC_Sensor_GetCorrente2(void) { return corrente2Eng; }
float ADC_Sensor_GetTemperatura1(void) { return temperatura1Eng; }
float ADC_Sensor_GetTemperatura2(void) { return temperatura2Eng; }
float ADC_Sensor_GetMassa1(void) { return massa1Eng; }
float ADC_Sensor_GetMassa2(void) { return massa2Eng; }
```