UNIVERSIDADE FEDERAL DA BAHIA (UFBA)
DEPARTAMENTO DE ENGENHARIA ELÉTRICA E COMPUTAÇÃO (DEEC)

PROGRAMAÇÃO EM TEMPO REAL PARA SISTEMAS EMBARCADOS (ENGD33)

PROFESSOR JESS DE JESUS FIAIS

Projeto de Controle de Motores Omnidirecionais com FreeRTOS

GABRIEL CORREIA DOS SANTOS (219215605)
MÁRCIO COSTA BARROS JUNIOR (221216662)
HUGO THOMÁS DE ALMEIDA E MENDES (218115287)

AGOSTO 2024
SALVADOR - BA

SUMÁRIO

1. INTRODUÇÃO

Os sistemas em tempo real são fundamentais para aplicações em sistemas embarcados, onde a precisão e a previsibilidade são cruciais. Esses sistemas garantem que tarefas críticas sejam executadas dentro de prazos rigorosos, o que é vital no controle de motores, como realizado neste projeto. Foi utilizado o FreeRTOS, um sistema operacional em tempo real amplamente adotado, para o desenvolvimento de um controle de motores omnidirecionais com o controlador STM32F4X1 Blackpill.

O FreeRTOS oferece uma série de funcionalidades que facilitam o desenvolvimento de sistemas embarcados complexos, como a criação e gerenciamento de múltiplas tarefas (tasks), que permitem a execução aparentemente concorrente de diferentes partes do código. Cada tarefa pode ser atribuída a uma prioridade específica, garantindo que as operações mais críticas sejam executadas de maneira oportuna. Além disso, o FreeRTOS inclui mecanismos de sincronização como filas (queues) e semáforos (semaphores), que permitem a comunicação e coordenação entre as tarefas, assegurando que os recursos compartilhados, sejam utilizados de forma eficiente e segura.

Outro recurso importante do FreeRTOS são os timers, que permitem a execução de ações periódicas ou agendadas, como o monitoramento de sensores ou o ajuste de controles de velocidade. No contexto deste projeto, os timers foram empregados para garantir que o controle dos motores seja ajustado continuamente, com base nas leituras dos encoders e sensores de corrente, além do desejo do usuario (input de do deslocamento desejado).

Combinando as funcionalidades do FreeRTOS com as características avançadas da plataforma STM32, este projeto demonstra a aplicação eficiente de conceitos de sistemas em tempo real em um sistema de controle de robô onidirecional, procurando garantir desempenho e confiabilidade em operações críticas.

2. OBJETIVOS DO PROJETO

- **Desenvolver e implementar um sistema de controle para um robô omnidirecional** : utilizando a plataforma STM32 Blackpill, integrando o FreeRTOS como sistema operacional em tempo real para gerenciar as tarefas críticas do sistema e técnicas PID.

- **Construir o código com a aplicação de funcionalidades do FreeRTOS**: criar um sistema modular e eficiente, capaz de realizar múltiplas operações simultaneamente, como controle de motores, leitura de sensores e comunicação com outros dispositivos utilizando de tasks, semáforos, filas e timer.

- **Assegurar a robustez do sistema:** garantir que o microcontrolador mantenha a funcionalidade mesmo em caso de falhas, prevenindo assim estados de erro prolongados.

- **Avaliar e validar o desempenho do sistema desenvolvido:** incluindo a análise do código implementado, sua funcionalidade e as expectativas de desempenho em uma aplicação prática. Além disso, identificar e discutir os elementos que precisam ser ajustados ao aplicar o projeto em um ambiente real.

3. MATERIAIS INDICADOS NO PROGRAMA

Abaixo segue a lista de materiais considerados para a escrita do programa criado.

- **STM32**: Placa de desenvolvimento compatível com FreeRTOS.

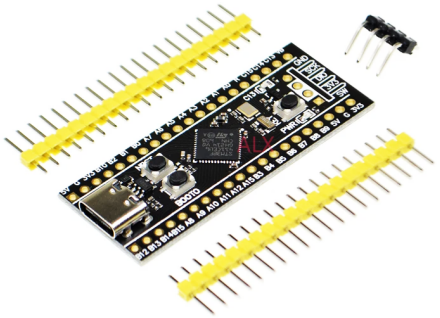

Figura 1: Placa do sistema de núcleo STM32F4X1

- **Motores com Encoders numa estrutura ommidercional** : Três motores para o robô. Na foto aqui inserida eles já estão acoplados em uma base.

Figura 2: CIM Motor (am-0255), encoder E4P miniature e base e rodas para estrutura omnidirecional.

- **Drivers dos Motores**: Com o objetivo de aplicar uma tensão de 12 V e dois sinais PWM será utilizado o seguinte driver.

Figura 3: Driver BTS7960 para cada motor.

- **LCD I2C**: Display para exibir informações sobre a velocidade dos motores.

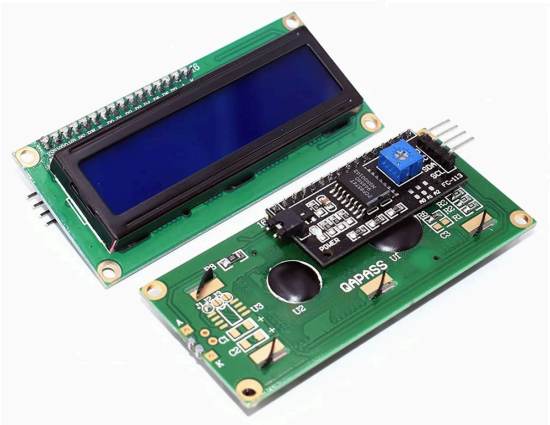

Figura 4: LCD I2C

- **Sensor de Corrente**: um sensor de corrente comum é o **ACS712**, que fornece uma tensão analógica proporcional à corrente que passa pelo sensor. É possível usar uma porta ADC (Conversor Analógico-Digital) para ler o valor desse sensor.

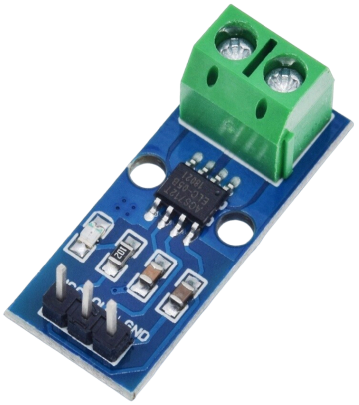

Figura 5: Sensor ACS712

- **Fontes de Alimentação**: Fonte de 12V .

Figura 6 : Bateria 12V.

4. CONTROLE PID

O controle PID (Proporcional-Integral-Derivativo) é amplamente utilizado em sistemas de controle de motores devido à sua capacidade de ajustar a resposta dinâmica de forma precisa. Esse tipo de controle é fundamental para aplicações onde se requer um alto grau de precisão na velocidade ou na posição do motor, como em robôs, veículos autônomos e sistemas industriais.

4.1. Funcionamento do controle PID aplicado aos motores:

O PID controla um motor ajustando o sinal de controle (normalmente uma tensão ou PWM) para alcançar a velocidade ou posição desejada. Esse ajuste é feito com base em três componentes principais

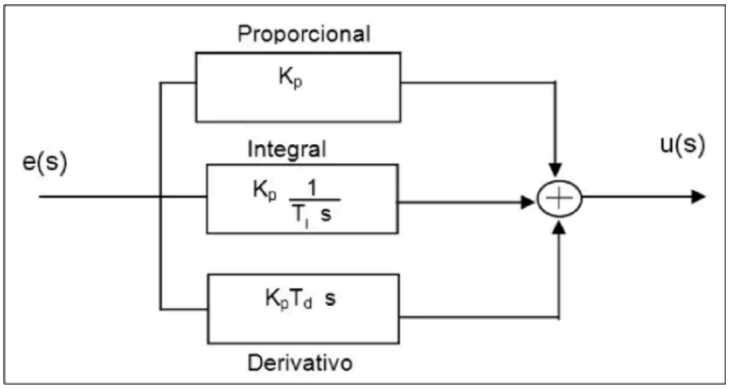

Figura 7 : Controlador PID

Proporcional (Kp): Ajusta a resposta do motor com base na diferença entre o valor desejado (setpoint) e o valor medido (feedback). Se a diferença for grande, o controlador aplica uma correção proporcionalmente maior.

Integral (Ti): Considera o histórico da diferença acumulada ao longo do tempo, corrigindo erros persistentes que o componente proporcional sozinho não consegue eliminar.

Derivativo (Td): Reage às mudanças na diferença, prevendo o comportamento futuro do sistema e ajustando a resposta para suavizar a ação do controle.

5. CINEMÁTICA

A cinemática de um robô com rodas omnidirecionais de três rodas envolve a análise dos movimentos lineares e angulares do robô, considerando a orientação e as contribuições vetoriais de cada roda para a movimentação da plataforma.

5.1. Disposição das Rodas

As rodas omnidirecionais são montadas em uma configuração triangular, com cada roda orientada de forma que seu eixo de rotação aponte para o centro do robô, formando um ângulo de 120° entre si. Esta configuração permite que o robô se mova em qualquer direção sem a necessidade de girar primeiro, oferecendo uma manobrabilidade rica e simplificação no controle. Podemos visualizar a configuração do motor e suas rodas, apresentado por [4], na Fig. 8.

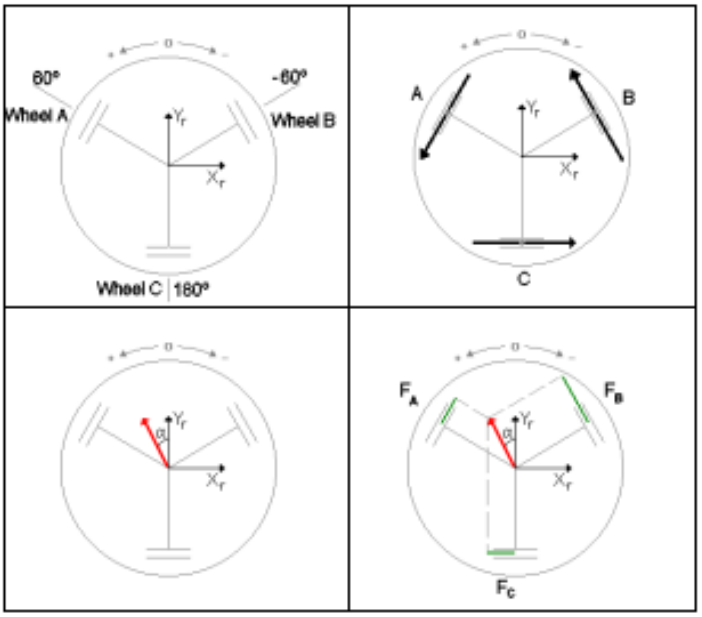

Figura 8: Configuração das rodas e componentes vetoriais

5.2. Contribuição dos Motores

Com a configuração apresentada podemos utilizar dos componentes vetoriais para definir as contribuições de cada motor, apresentado por [4], para os movimentos desejados. As equações abaixo mostram as contribuições.

$$F_A = velocidade \times cos(150 - DireçãoDesejada) \quad (1)$$

$$F_B = velocidade \times cos(30 - DireçãoDesejada) \quad (2)$$

$$F_C = velocidade \times cos(270 - DireçãoDesejada) \quad (3)$$

Então vemos que a contribuição (F) de cada motor (A,B,C) é dada pela velocidade desejada e o cossenso entre a posição do motor e a direção angular desejada. O que simplifica a lógica de programação aplicada.

Assim podemos obter **movimento linear** pela soma vetorial das contribuições de cada uma das três rodas. A direção do movimento desejado é projetada nos eixos de cada uma das rodas, resultando em componentes de velocidade para cada motor. Para **movimentos angulares**, onde o robô precisa girar em torno de seu centro, todas as rodas são acionadas na mesma direção e com a mesma velocidade. A velocidade angular do robô é determinada pela velocidade linear periférica das rodas dividida pelo raio da plataforma. Isso permite ao robô realizar rotações precisas, mantendo a estabilidade do movimento.

Em situações onde o robô precisa se mover linearmente enquanto rotaciona, **é possível combinar os movimentos linear e angular**. O resultado é a soma dos vetores de velocidade linear e angular, ajustando-se para garantir que as velocidades combinadas não excedam a capacidade máxima dos motores.

Neste trabalho vamos nos conter no controle dos motores, então imaginamos o recebimento das contribuições em velocidade de cada motor e aplicamos controle de tração e de velocidade. O momento em que os motores param ou suas contribuições são alteradas são inputs externos ao nosso sistema. A partir deste input realizamos o controle individualizado dos motores.

6. CONCEPÇÃO GERAL DO PROJETO

O código desenvolvido implementa o controle de motores omnidirecionais utilizando um microcontrolador STM32, com o auxílio do sistema operacional em tempo real FreeRTOS. O objetivo principal é controlar três motores de um robô omnidirecional, garantindo que ele se mova conforme as velocidades desejadas nos eixos x e y, além de uma rotação ao redor de seu próprio eixo. Para isso, foi utilizado o controle PID (Proporcional, Integral, Derivativo) para cada motor, permitindo que as velocidades dos motores sejam ajustadas conforme a necessidade, levando em consideração a leitura dos encoders e dos sensores de corrente.

O código foi inicialmente adaptado para a plataforma STM32, substituindo as bibliotecas específicas do Arduino por bibliotecas próprias para o STM32, como ***stm32f4xx_hal.h*** e ***FreeRTOS.h.*** Essas bibliotecas são essenciais para configurar os periféricos do microcontrolador e para gerenciar tarefas em tempo real, respectivamente. O microcontrolador STM32 utilizado suporta múltiplos canais PWM, que foram configurados para controlar a velocidade dos motores. Cada motor possui dois PWM para controle da velocidade e um pino para definir a direção do giro, além de pinos para os encoders, responsáveis por fornecer feedback sobre a posição e a velocidade do motor. As pinagens foram escolhidas a imagem do padrão definido pela disciplina, como visto na figura 9.

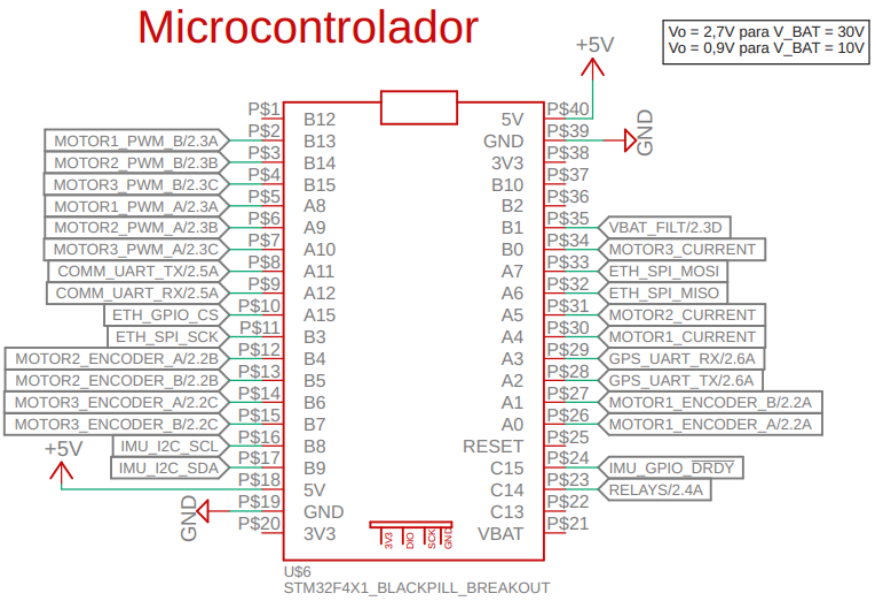

Figura 9 : Configuração dos pinos do controlador

Este projeto foca no controle dos motores, o que significa que nem todos os pinos do microcontrolador serão utilizados pelo projeto. No entanto, a maioria deles é dedicada justamente a esse controle. Alguns pinos digitais estão conectados aos encoders, enquanto outros são pinos PWM e outros configurados como ADC para monitoramento de corrente.

Para garantir que o sistema permaneça funcional em caso de falhas inesperadas, foi implementado um Watchdog Timer (WDT). O Watchdog é um temporizador que precisa ser resetado periodicamente pelo software; caso isso não ocorra, o microcontrolador é reinicializado automaticamente, o que previne que o sistema trave ou entre em um estado inconsistente por um longo período. No código, o Watchdog foi configurado para um tempo específico, de modo que se qualquer tarefa demorar demais para ser executada, o microcontrolador será reinicializado, garantindo a robustez do sistema.

O controle dos motores é realizado através de tarefas específicas do FreeRTOS, que permitem a execução concorrente de diferentes partes do código. Cada motor possui uma tarefa dedicada para controlar sua velocidade e direção. Essas tarefas utilizam as leituras dos encoders para calcular a velocidade atual do motor e ajustar a saída do controlador PID de acordo com o erro entre a velocidade desejada e a velocidade real.

Além do controle PID, o código implementa uma verificação da tração das rodas, utilizando a leitura dos sensores de corrente. Esses sensores detectam o consumo de corrente dos motores, que pode indicar se uma roda está derrapando (alta corrente) ou se não está oferecendo resistência (baixa corrente). Com base nessa leitura, o código ajusta o sinal PWM para cada motor, reduzindo a potência em caso de derrapagem ou aumentando-a se for necessário.

Outra parte importante do código é a tarefa de exibição de informações no display LCD. Essa tarefa é responsável por mostrar, de forma contínua, as contagens dos encoders de dois dos motores, proporcionando uma interface simples para monitoramento do sistema em tempo real. O acesso ao LCD é sincronizado por um semáforo binário do FreeRTOS, que garante que apenas uma tarefa por vez possa acessar o display, evitando conflitos e corrupção de dados.

Por fim, o setup do microcontrolador configura todas as interrupções para a leitura dos encoders, inicializa os canais PWM para controlar os motores, e cria as tarefas necessárias para o funcionamento do sistema. O FreeRTOS gerencia essas tarefas, alternando entre elas de acordo com as prioridades definidas, permitindo que o sistema opere de forma eficiente e responsiva.

Essa concepção do código para o STM32 com FreeRTOS e Watchdog Timer garante um sistema robusto, capaz de controlar com precisão os motores de um robô omnidirecional, enquanto se protege contra possíveis falhas e mantém uma interface de monitoramento em tempo real.

7. PARÂMETROS INICIAIS DO PROJETO

7.1. Descrição resumitiva das malhas de controle

Malha de Corrente pelo Sensor de Corrente (1ms):

- Sensor de Corrente nos terminais do motor: 3 dados com período de 1ms e direção Tx
- Ações de controle de tração: 3 dados com período de 1ms e direção Tx
- Ganhos do controlador de tração (Kp, Ki, Kd): 9 dados com período de 1ms e direção Tx
- Ganhos do controlador de tração (Kp, Ki, Kd): 9 dados aperiódicos e direção Rx

Malha de Velocidade pelo Sensor Encoder (10ms):

- Velocidade Angular (eixo do motor): 3 dados com período de 10ms e direção Tx
- Aceleração linear (Ax, Ay, Az): 3 dados com período de 10ms e direção Tx
- Velocidade angular (Gx, Gy, Gz, giroscópio): 3 dados com período de 10ms e direção Tx
- Ações de controle de velocidade: 3 dados com período de 10ms e direção Tx
- Setpoint de velocidade: 3 dados com período de 10ms e direção Tx
- Ganhos do controlador de velocidade (Kp, Ki, Kd): 9 dados com período de 10ms e direção Tx
- Ganhos do controlador de velocidade (Kp, Ki, Kd): 9 dados aperiódicos e direção Rx

Malha de Posição (100ms):

- Ações de controle de posição: 3 dados de 100ms e direção Tx
- Ganhos do controlador da posição (Kp, Ki, Kd): 3 dados de período 100ms e direção Tx
- Ganhos do controlador da posição (Kp, Ki, Kd): 3 dados aperiódicos e direção Rx
- Ângulos de rotação da base (Roll, Pitch, Yaw): 9 dados aperiódicos e direção Rx

7.2. Parâmetros Do Encoder

Tem-se 1440 pulsos por rotação total da roda. Com tempo de amostragem de 25ms da malha, cada rotação/segundo produz 36 pulsos. Então a velocidade da roda será definida por:

$$Velocidade\ Rotacional\ = \frac{Contagem\ de\ pulsos}{36} \qquad (4)$$

7.3. Parâmetros Do Controle Do Projeto

- Extensão positiva e negativa do sinal de controle: Saturar entre -1 e +1.
- Controle igual a zero: PWM para frente e para trás igual a zero.
- Controle positivo: PWM para frente = sinal Controle * 100%.
- Controle negativo: PWM para trás = sinal Controle * 100%.
- Sinal de controle fora da faixa [-1, 1]: Saturar em -1 ou +1.

7.4. Configuração Do Pinos Do Microcontrolador

O projeto vai utilizar um STM32F4X1 BlackPill Breakout com programa a ser compilado no STM32CubeIDE. Os pinos do microcontrolador estarão com:

B13 - MOTOR1_PWM_B
B14 - MOTOR2_PWM_B
B15 - MOTOR3_PWM_B
A8 - MOTOR1_PWM_A
A9 - MOTOR2_PWM_A
A10 - MOTOR3_PWM_A
A11 - COMM_UART_TX
A12 - COMM_UART_RX
A15 - ETH_GPIO_CS
B3 - ETH_SPI_SCK
B4 - MOTOR2_ENCODER_A
B5 - MOTOR2_ENCODER_B
B6 - MOTOR3_ENCODER_A
B7 - MOTOR3_ENCODER_B
B8 - IMU_I2C_SCL
B9 - IMU_I2C_SDA
B10 - RELAYS

B1 - VBAT_FILT
B0 - MOTOR3_CURRENT
A7 - ETH_SPI_MOSI
A6 - ETH_SPI_MISO
A5 - MOTOR2_CURRENT
A4 - MOTOR1_CURRENT
A3 - GPS_UART_RX
A2 - GPS_UART_TX
A1 - MOTOR1_ENCODER_B
A0 - MOTOR1_ENCODER_A
RESET -
C15 -
C14 -
C13 - IMU_GPIO_DRDY
VBAT -
5V@2 - 5V_MCU

Assim como o SV7 está com os pinos 2 e 3 no 5V_MCU e o pino 1 no +5V. Lembrando que Vo=2,7V para V_BAT=30V e Vo=0,9V para V_BAT=10V

8. DESCRIÇÃO DO PROJETO UTILIZANDO FREE RTOS

8.1. Importação de Bibliotecas

```
// Importação de Bibliotacas
#include "main.h"
#include "cmsis_os.h"
#include "adc.h"
#include "tim.h"
#include "gpio.h"
#include "iwdg.h"
#include "stm32f4xx_hal.h"
#include <math.h>
#include <stdint.h>
#include <stdbool.h>
```

Esses arquivos de cabeçalho são incluídos para fornecer as declarações e definições necessárias para o código. Eles têm as seguintes funções:

- **"main.h"**: geralmente contém declarações de funções e variáveis que são usadas em main.c e outros arquivos do projeto.
- **"cmsis_os.h"**: inclui funções e definições relacionadas ao gerenciamento de sistemas operacionais em tempo real (RTOS), como FreeRTOS.
- **"adc.h"**: fornece declarações para as funções relacionadas ao módulo de conversão analógica-digital (ADC).
- **"tim.h"**: inclui funções e definições para a configuração e uso dos temporizadores (Timers) do microcontrolador.
- **"gpio.h"**: contém funções e definições para o controle dos pinos de entrada/saída gerais (GPIO).
- **"iwdg.h"**: define funções para a configuração e uso do watchdog timer (IWDG), que ajuda a monitorar e recuperar o sistema em caso de falhas.
- **"stm32f4xx_hal.h"**: fornece a interface de Hardware Abstraction Layer (HAL) para o microcontrolador STM32F4, simplificando o acesso aos periféricos e funções do hardware.
- **<math.h>**: inclui funções matemáticas padrão da biblioteca C.
- **<stdint.h>**: fornece definições para tipos inteiros de largura fixa.
- **<stdbool.h>**: define o tipo booleano bool e os valores true e false para facilitar a manipulação de condições lógicas.

Cada um desses arquivos e bibliotecas é crucial para a configuração e operação do sistema em um ambiente STM32, facilitando o desenvolvimento ao fornecer funções e definições prontas para uso.

8.2. Definições Iniciais do Programa

```
// Definição das Queues
osMessageQueueId_t QueueCorrenteHandle;
osMessageQueueId_t QueueVelocidadeHandle;
osMessageQueueId_t QueuePosicaoHandle;
```

As declarações do início definem identificadores para as filas de mensagens no sistema de tempo real FreeRTOS. As filas são usadas para facilitar a comunicação entre diferentes tarefas. No código:

- QueueCorrenteHandle é utilizado para mensagens relacionadas ao controle de corrente.
- QueueVelocidadeHandle é utilizado para mensagens relacionadas ao controle de velocidade.
- QueuePosicaoHandle é utilizado para mensagens relacionadas ao controle de posição.

```
// Definição dos PID Controllers
PID_Controller pid_traction;
PID_Controller pid_velocity;
PID_Controller pid_position;

// Definição dos PID Controllers
PID_TypeDef PID_Motor1, PID_Motor2, PID_Motor3;
```

Essa parte diz respeito sobre a declaração dos três controladores PID:

- pid_traction é usado para controlar a tração.
- pid_velocity é usado para controlar a velocidade.
- pid_position é usado para controlar a posição.
- PID_Motor1, PID_Motor2 e PID_Motor3 são controladores PID específicos para cada motor, gerenciando individualmente o controle dos motores.

```
// Definição das variáveis globais
volatile int32_t encoderReading1 = 0;
volatile int32_t encoderReading2 = 0;
volatile int32_t encoderReading3 = 0;
```

```
double Roll, Pitch, Yaw;
float speed, traction;
double Motor1_Setpoint, Motor2_Setpoint, Motor3_Setpoint;
double Motor1_Input, Motor2_Input, Motor3_Input;
double Motor1_Output, Motor2_Output, Motor3_Output;
```

Estas são variáveis globais usadas em diferentes partes do programa:

- encoderReading1, encoderReading2, encoderReading3 armazenam as leituras dos encoders dos motores. A palavra-chave volatile indica que esses valores podem ser alterados por interrupções ou tarefas concorrentes.
- Roll, Pitch, Yaw armazenam os ângulos de rotação do sistema, que viriam dos comandos de orientação do usuário.
- speed, traction armazenam a velocidade e a tração atuais.
- Motor1_Setpoint, Motor2_Setpoint, Motor3_Setpoint definem os pontos de ajuste desejados para cada motor.
- Motor1_Input, Motor2_Input, Motor3_Input armazenam os valores de entrada para os controladores PID dos motores.
- Motor1_Output, Motor2_Output, Motor3_Output armazenam os valores de saída dos controladores PID, usados para ajustar os motores.

```
// Prototipação das Funções das Tasks
void TaskControleCorrente(void *argument);
void TaskControleVelocidade(void *argument);
void TaskControlePosicao(void *argument);
void TaskControlePID(void *argument);
void TaskEnvioDadosUART(void *argument);
```

Estas são declarações das funções que representam as tarefas do FreeRTOS:

- TaskControleCorrente gerencia o controle da corrente.
- TaskControleVelocidade gerencia o controle da velocidade.
- TaskControlePosicao gerencia o controle da posição.
- TaskControlePID gerencia o controle PID.
- TaskEnvioDadosUART é responsável pelo envio de dados via comunicação UART.

```
// Protótipos de funções
void PID_Compute(PID_TypeDef *pid);
void PID_Init(PID_TypeDef *pid, double kp, double ki, double kd);
double Read_Angle_Roll(void);
```

```
double Read_Angle_Pitch(void);
double Read_Angle_Yaw(void);
void Error_Handler(void);
```

Estas são declarações de funções usadas no código:

- PID_Compute calcula a saída do controlador PID.
- PID_Init inicializa um controlador PID com os parâmetros proporcionais, integrais e derivados.
- Read_Angle_Roll, Read_Angle_Pitch, Read_Angle_Yaw lêem os ângulos de rotação do sistema.
- Error_Handler lida com situações de erro no sistema.

```
// Função para Inicialização de Periféricos
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
static void MX_ADC2_Init(void);
static void MX_ADC3_Init(void);
static void MX_TIM1_Init(void);
static void MX_TIM2_Init(void);
static void MX_TIM3_Init(void);
static void MX_IWDG_Init(void);
static void MX_USART2_UART_Init(void);
```

Estas funções são responsáveis pela configuração inicial dos periféricos do microcontrolador:

1. SystemClock_Config configura o sistema de clock.
2. MX_GPIO_Init inicializa a configuração dos pinos GPIO.
3. MX_ADC1_Init, MX_ADC2_Init, MX_ADC3_Init inicializam os conversores analógicos-digitais.
4. MX_TIM1_Init, MX_TIM2_Init, MX_TIM3_Init inicializam os temporizadores.
5. MX_IWDG_Init inicializa o watchdog timer.
6. MX_USART2_UART_Init inicializa a comunicação UART.

8.3. Loop Principal do Programa

```
// Inicialização do hardware, do sistema e dos periféricos
```

```
HAL_Init();
SystemClock_Config();
MX_GPIO_Init();
MX_ADC1_Init();
MX_ADC2_Init();
MX_ADC3_Init();
MX_TIM1_Init();
MX_TIM2_Init();
MX_TIM3_Init();
MX_IWDG_Init();
MX_USART2_UART_Init();
```

Esta seção é responsável pela configuração inicial do hardware e dos periféricos do microcontrolador:

- HAL_Init(): Inicializa a Biblioteca de Abstração de Hardware (HAL) da STM32. Esta função configura o sistema de clock e inicializa as variáveis e estruturas da HAL.
- SystemClock_Config(): Configura o sistema de clock do microcontrolador, ajustando a frequência do clock de acordo com os requisitos do sistema.
- MX_GPIO_Init(): Inicializa os pinos de entrada e saída (GPIO) configurando-os conforme necessário para o funcionamento do hardware.
- MX_ADC1_Init(), MX_ADC2_Init(), MX_ADC3_Init(): Inicializam os conversores analógicos-digitais (ADC) para a leitura de sinais analógicos.
- MX_TIM1_Init(), MX_TIM2_Init(), MX_TIM3_Init(): Inicializam os temporizadores (Timers), que são usados para controle de tempo e geração de sinais PWM.
- MX_IWDG_Init(): Inicializa o watchdog timer (IWDG), que é um mecanismo de segurança para reiniciar o microcontrolador em caso de falha do software.
- MX_USART2_UART_Init(): Inicializa a interface de comunicação UART (USART2) para comunicação serial com outros dispositivos.

```
// Inicialização das variáveis globais
Roll = 0.0f;
Pitch = 0.0f;
Yaw = 0.0f;
speed = 0.0f;
traction = 0.0f;
Motor1_Setpoint = 0.0f;
Motor2_Setpoint = 0.0f;
Motor3_Setpoint = 0.0f;
```

Nesta etapa, as variáveis globais são inicializadas com valores padrão:

- Roll, Pitch, Yaw: Inicializam os ângulos de rotação em 0.0.
- speed, traction: Inicializam a velocidade e a tração em 0.0.
- Motor1_Setpoint, Motor2_Setpoint, Motor3_Setpoint: Inicializam os pontos de ajuste para cada motor em 0.0, o que indica que os motores inicialmente não têm uma velocidade ou posição alvo definida.

```
// Inicialização das Queues
QueueCorrenteHandle = osMessageQueueNew(3, sizeof(float), NULL); //Tamanho
ajustável da fila em 3
QueueVelocidadeHandle = osMessageQueueNew(3, sizeof(float), NULL);
//Tamanho ajustável da fila em 3
QueuePosicaoHandle = osMessageQueueNew(3, sizeof(float), NULL); //Tamanho
ajustável da fila em 3
```

Aqui, são criadas três filas de mensagens utilizando o FreeRTOS:

- QueueCorrenteHandle: Fila para mensagens relacionadas ao controle de corrente, com capacidade para 3 mensagens do tipo float.
- QueueVelocidadeHandle: Fila para mensagens relacionadas ao controle de velocidade, também com capacidade para 3 mensagens do tipo float.
- QueuePosicaoHandle: Fila para mensagens relacionadas ao controle de posição, com a mesma capacidade.

```
// Inicialização dos Controladores PID

PID_Init(&pid_traction, KP_TRACTION, KI_TRACTION, KD_TRACTION);
PID_Init(&pid_velocity, KP_VELOCITY, KI_VELOCITY, KD_VELOCITY);
PID_Init(&pid_position, KP_POSITION, KI_POSITION, KD_POSITION);
```

Os controladores PID são inicializados com os parâmetros específicos para cada tipo de controle:

- PID_Init(&pid_traction, KP_TRACTION, KI_TRACTION, KD_TRACTION): Inicializa o controlador PID para tração com os ganhos proporcionais (KP), integrais (KI) e derivados (KD) definidos.
- PID_Init(&pid_velocity, KP_VELOCITY, KI_VELOCITY, KD_VELOCITY): Inicializa o controlador PID para velocidade.
- PID_Init(&pid_position, KP_POSITION, KI_POSITION, KD_POSITION): Inicializa o controlador PID para posição.

```
// Criação das Tasks
```

```
const osThreadAttr_t highPriorityTask_attr = {.priority = (osPriority_t) osPriorityHigh};  // Definir a prioridade das tasks de controle como alta

osThreadNew(TaskControleCorrente, NULL, &highPriorityTask_attr); // Task Controle de Corrente - leitura dos sensores de corrente no ADC

osThreadNew(TaskControleVelocidade, NULL, &highPriorityTask_attr); // Task Controle de Velocidade - leitura do Encoder

osThreadNew(TaskControlePosicao, NULL, &highPriorityTask_attr); // Task Controle de Posição - leitura do GPS

osThreadNew(TaskControlePID, NULL, NULL);  // Task PID com prioridade padrão

osThreadNew(TaskEnvioDadosUART, NULL, NULL);  // Task de envio de dados UART com prioridade padrão
```

Nesta etapa, as tarefas (tasks) do FreeRTOS são criadas:

- highPriorityTask_attr: Define a prioridade alta para as tarefas de controle.
- osThreadNew(TaskControleCorrente, NULL, &highPriorityTask_attr): Cria a tarefa para controle de corrente, com alta prioridade.
- osThreadNew(TaskControleVelocidade, NULL, &highPriorityTask_attr): Cria a tarefa para controle de velocidade, também com alta prioridade.
- osThreadNew(TaskControlePosicao, NULL, &highPriorityTask_attr): Cria a tarefa para controle de posição, com alta prioridade.
- osThreadNew(TaskControlePID, NULL, NULL): Cria a tarefa para o controle PID com prioridade padrão.
- osThreadNew(TaskEnvioDadosUART, NULL, NULL): Cria a tarefa para envio de dados via UART, com prioridade padrão.

Essas tarefas são responsáveis por diferentes funções do sistema e operam em paralelo.

```
    // Inicia o Scheduler do FreeRTOS
    osKernelStart();
```

Esta função inicia o scheduler do FreeRTOS, que começa a gerenciar a execução das tarefas criadas. O scheduler gerencia o tempo de CPU entre as tarefas e garante que cada uma delas seja executada conforme sua prioridade e requisitos.

```
    // Caso o Scheduler do FreeRTOS falhe
```

```
    while (1) {}
```

Este é um loop infinito executado se o scheduler do FreeRTOS falhar ao iniciar. Normalmente, este código nunca é executado se o sistema estiver funcionando corretamente, e serve como uma medida de segurança para indicar que o sistema não está operando conforme o esperado.

Cada uma dessas etapas configura e inicia o sistema, garantindo que o hardware, os periféricos e o software de controle estejam prontos para operar de maneira coordenada..

8.4. Tarefa Controle de Corrente

```
// Task para controle da Corrente (Leitura do Sensor)
void TaskControleCorrente(void *argument) {
    float correnteMotor[3] = {0.0f, 0.0f, 0.0f};
    const float referenciaADC = 3.3f;  // Tensão de referência do ADC
    const float resolucaoADC = 4096.0f; // Resolução do ADC de 12 bits
    const float vccDiv2 = referenciaADC / 2.0f; // Vcc/2 = 1.65V para
ACS712
    const float sensibilidade = 0.185f; // Sensibilidade de 185 mV/A para
ACS712-05B

    for (;;) {
        // Leitura dos sensores de corrente (3 dados com período de 1ms)
        HAL_ADC_Start(&hadc1);
        HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
        float valorADC = HAL_ADC_GetValue(&hadc1);
        float vout = (valorADC / resolucaoADC) * referenciaADC;
        correnteMotor[0] = (vout - vccDiv2) / sensibilidade;

        HAL_ADC_Start(&hadc2);
        HAL_ADC_PollForConversion(&hadc2, HAL_MAX_DELAY);
        valorADC = HAL_ADC_GetValue(&hadc2);
        vout = (valorADC / resolucaoADC) * referenciaADC;
        correnteMotor[1] = (vout - vccDiv2) / sensibilidade;

        HAL_ADC_Start(&hadc3);
        HAL_ADC_PollForConversion(&hadc3, HAL_MAX_DELAY);
        valorADC = HAL_ADC_GetValue(&hadc3);
        vout = (valorADC / resolucaoADC) * referenciaADC;
        correnteMotor[2] = (vout - vccDiv2) / sensibilidade;
```

```
        // Verificação de espaço na fila antes de enviar os dados e Envio
dos dados para a Queue
        if (osMessageQueueGetSpace(QueueCorrenteHandle) > 0) {
            osMessageQueuePut(QueueCorrenteHandle, &correnteMotor[0], 0,
0);
        }

        // Aguarda 1ms antes de executar novamente
        osDelay(1);
    }
}
```

De início, é importante já ressaltar que toda a leitura dos encoders é realizada com base em tensão, logo, é feita a correção para o equivalente valor de corrente para a tarefa de controle PID. Foram usados como parâmetros o sensor ACS712-05B, cuja tensão de referência é de 3,3V e a sensibilidade é de 185mV/A.

A TaskControleCorrente é uma tarefa que realiza a leitura contínua dos sensores de corrente e o envio dos dados coletados para uma fila de mensagens no FreeRTOS. Inicialmente, a tarefa define um array correnteMotor com três elementos, todos inicializados com 0.0, para armazenar as leituras dos sensores de corrente.

Dentro de um loop infinito, a tarefa procede com a leitura dos sensores de corrente utilizando conversões analógicas para digitais. Para cada um dos três sensores conectados aos ADCs (Conversores Analógicos para Digitais) do sistema, a tarefa inicia a conversão com HAL_ADC_Start, aguarda a conclusão da conversão com HAL_ADC_PollForConversion e então obtém o valor convertido com HAL_ADC_GetValue. Esses valores são armazenados nos elementos correspondentes do array correnteMotor.

Após obter as leituras dos três sensores, a tarefa verifica se há espaço disponível na fila de mensagens QueueCorrenteHandle utilizando osMessageQueueGetSpace. Se houver espaço disponível, os dados dos sensores (correnteMotor[0], correnteMotor[1], correnteMotor[2]) são enviados para a fila com osMessageQueuePut.

Finalmente, a tarefa aguarda por 1 milissegundo utilizando osDelay, o que limita a taxa de leitura dos sensores e previne a sobrecarga do processador. Este atraso é crucial para garantir que a tarefa execute periodicamente.

8.5. Tarefa Controle de Velocidade

```
#define WINDOW_SIZE 5

// Task para controle da Velocidade (Leitura do Encoder)
void TaskControleVelocidade(void *argument) {
    static float leituraBuffer[3][WINDOW_SIZE] = {0};
    static uint8_t bufferIndex = 0;
    static uint32_t lastSampleTime = 0;
    float velocidadeMotor[3] = {0.0f};
    int32_t encoderReading[3] = {0};
    int32_t lastEncoderReading[3] = {0};

    for (;;) {
        uint32_t currentTime = osKernelGetTickCount();
        uint32_t elapsedTime = currentTime - lastSampleTime;

        if (elapsedTime >= SAMPLE_PERIOD_MS) {
            // Leitura dos encoders atuais
            encoderReading[0] = __HAL_TIM_GET_COUNTER(&htim1);
            encoderReading[1] = __HAL_TIM_GET_COUNTER(&htim2);
            encoderReading[2] = __HAL_TIM_GET_COUNTER(&htim3);

            // Calcular a velocidade bruta
            float velocidadeBruta[3];
            for (int i = 0; i < 3; i++) {
                velocidadeBruta[i] = (float)(encoderReading[i] -
lastEncoderReading[i]) / 36.0f;
                leituraBuffer[i][bufferIndex] = velocidadeBruta[i];
                lastEncoderReading[i] = encoderReading[i];
            }

            // Atualizar o índice do buffer
            bufferIndex = (bufferIndex + 1) % WINDOW_SIZE;
            lastSampleTime = currentTime;
        }

        // Calcular a média móvel usando as leituras mais recentes
disponíveis
        for (int i = 0; i < 3; i++) {
            float soma = 0.0f;
            for (uint8_t j = 0; j < WINDOW_SIZE; j++) {
                soma += leituraBuffer[i][j];
            }
            velocidadeMotor[i] = soma / WINDOW_SIZE;
        }

        // Verificação de espaço na fila antes de enviar os dados
        if (osMessageQueueGetSpace(QueueVelocidadeHandle) > 0) {
            osMessageQueuePut(QueueVelocidadeHandle, &velocidadeMotor[0],
0, 0);
```

```
        }

        // Aguarda 10ms antes de executar novamente
        osDelay(10);
    }
}
```

Nesta parte do código, é implementada uma tarefa para controle da velocidade dos motores, utilizando a leitura dos encoders para calcular e suavizar a velocidade. Inicialmente, é definido um buffer de tamanho WINDOW_SIZE (5), que armazena as leituras de velocidade recentes para cada motor. As variáveis estáticas, incluindo o índice do buffer (bufferIndex) e o tempo da última amostra (lastSampleTime), são utilizadas para manter o estado entre as iterações da tarefa. As leituras dos encoders atuais e anteriores são armazenadas nas variáveis encoderReading e lastEncoderReading, respectivamente.

Durante cada iteração do loop da tarefa, o tempo atual é obtido e comparado com o tempo da última amostra para garantir que o período de amostragem definido (SAMPLE_PERIOD_MS) tenha sido cumprido. Se o período foi alcançado, a tarefa lê os valores dos encoders de cada motor. A diferença entre as leituras atuais e anteriores dos encoders é utilizada para calcular a velocidade bruta dos motores, que é então armazenada no buffer correspondente. Após atualizar o buffer, o índice do buffer é incrementado e o tempo da última amostra é atualizado.

Após a coleta dos dados, a tarefa calcula a média móvel das velocidades dos motores. A média móvel é obtida somando as leituras armazenadas no buffer para cada motor e dividindo o resultado pelo tamanho da janela (WINDOW_SIZE). Este cálculo proporciona uma versão suavizada da velocidade, reduzindo o impacto de variações rápidas e ruídos nas leituras.

Depois de calcular a média móvel, a tarefa verifica se há espaço disponível na fila de mensagens (QueueVelocidadeHandle) antes de inserir os dados de velocidade na fila, garantindo assim que os dados não sejam perdidos. Finalmente, a tarefa aguarda 10 milissegundos antes de iniciar a próxima iteração, o que assegura uma periodicidade de execução de 10ms. Este atraso regula a frequência de execução da tarefa, equilibrando a carga de processamento e o tempo de amostragem.

Em resumo, a tarefa TaskControleVelocidade realiza a leitura dos encoders, calcula e suaviza a velocidade dos motores usando uma média móvel e envia os dados para uma fila de mensagens, mantendo uma operação periódica de 10ms.

8.6. Tarefa Controle de Posição

```
// Task para controle da Posição (Ideia de Leitura Advinda do GPS)
void TaskControlePosicao(void *argument) {
    float posicaoBase[3] = {0.0f, 0.0f, 0.0f};
    for (;;) {
        // Leitura dos ângulos
        Roll = Read_Angle_Roll();
        Pitch = Read_Angle_Pitch();
        Yaw = Read_Angle_Yaw();

        // Leitura dos ângulos de rotação da base (3 dados com período de
100ms)
        posicaoBase[0] = Roll;   // Função para ler o ângulo Roll
        posicaoBase[1] = Pitch;  // Função para ler o ângulo Pitch
        posicaoBase[2] = Yaw;    // Função para ler o ângulo Yaw

        // Verificação de espaço na fila antes de enviar os dados
        if (osMessageQueueGetSpace(QueuePosicaoHandle) > 0) {
            osMessageQueuePut(QueuePosicaoHandle, &posicaoBase[0], 0, 0);
        }

        // Aguarda 100ms antes de executar novamente
        osDelay(100);
    }
}
```

A função `TaskControlePosicao` é responsável por monitorar e atualizar a posição do sistema com base em leituras de ângulos, simulando a leitura de dados de um sistema GPS ou sensores de orientação. A seguir está uma descrição detalhada de cada etapa dessa tarefa:

1. **Inicialização do Array de Posições:** A tarefa começa com a definição de um array `posicaoBase` com três elementos, todos inicializados com 0.0. Este array é usado para armazenar os ângulos de rotação da base do sistema, onde cada elemento representa uma direção diferente: `posicaoBase[0]` para o ângulo de Roll, `posicaoBase[1]` para o ângulo de Pitch e `posicaoBase[2]` para o ângulo de Yaw.
2. **Loop Infinito:** Em seguida, a tarefa entra em um loop infinito (`for (;;)`). Este loop é a estrutura básica que permite à tarefa executar repetidamente suas operações enquanto o sistema estiver em funcionamento.
3. **Leitura dos Ângulos:** Dentro do loop, a tarefa realiza a leitura dos ângulos de rotação utilizando funções específicas:
    - `Roll` é atualizado com o valor retornado pela função `Read_Angle_Roll()`.
    - `Pitch` é atualizado com o valor retornado pela função `Read_Angle_Pitch()`.
    - `Yaw` é atualizado com o valor retornado pela função `Read_Angle_Yaw()`.

4. Essas funções (`Read_Angle_Roll`, `Read_Angle_Pitch`, `Read_Angle_Yaw`) são presumivelmente responsáveis por capturar os ângulos atuais da base do sistema, possivelmente a partir de um sensor de orientação ou uma leitura simulada.
5. **Atualização do Array de Posições:** Os ângulos lidos são então atribuídos ao array `posicaoBase`:
    - `posicaoBase[0]` é definido como o valor de `Roll`.
    - `posicaoBase[1]` é definido como o valor de `Pitch`.
    - `posicaoBase[2]` é definido como o valor de `Yaw`.
6. **Envio dos Dados para a Fila:** A tarefa verifica se há espaço disponível na fila de mensagens `QueuePosicaoHandle` utilizando a função `osMessageQueueGetSpace`. Se houver espaço suficiente, a tarefa envia os dados do array `posicaoBase` para a fila usando a função `osMessageQueuePut`.
7. **Atraso entre Iterações:** Após enviar os dados, a tarefa aguarda por 100 milissegundos com a função `osDelay`. Esse atraso define o intervalo entre as leituras sucessivas dos ângulos e o envio dos dados para a fila, permitindo uma atualização periódica dos dados de posição a cada 100 milissegundos.

Em resumo, a função `TaskControlePosicao` é responsável por ler os ângulos de rotação da base do sistema, atualizar um array com esses valores, e enviar os dados para uma fila de mensagens, tudo isso a cada 100 milissegundos. Essa tarefa é crucial para o controle e monitoramento da posição do sistema em tempo real.

8.7. Tarefa para enviar e receber os dados via Comunicação Serial

```
// Adicionar uma estrutura para armazenar os parâmetros PID dinâmicos
typedef struct {
    double kp;
    double ki;
    double kd;
} PID_Params;

PID_Params params_traction = {KP_TRACTION, KI_TRACTION, KD_TRACTION};
PID_Params params_velocity = {KP_VELOCITY, KI_VELOCITY, KD_VELOCITY};
PID_Params params_position = {KP_POSITION, KI_POSITION, KD_POSITION};

// Mutex para sincronização
```

```
osMutexId_t mutexPIDParamsHandle;
osMutexId_t mutexSetpointsHandle;

// Semáforos para sincronização
osSemaphoreId_t semPIDParamsUpdateHandle;
osSemaphoreId_t semSetpointsUpdateHandle;

// Task para processamento da UART e atualização de setpoints
void TaskEnvioDadosUART(void *argument) {
    char buffer[256];
    int len;
    float new_kp, new_ki, new_kd;
    float new_traction_setpoint[3];
    float new_velocity_setpoint[3];
    float new_position_setpoint[3];
    char command[64];

    for (;;) {
        // Envio dos dados via UART
        len = snprintf(buffer, sizeof(buffer),
            "Roll: %.2f, Pitch: %.2f, Yaw: %.2f, Speed: %.2f, Traction:
%.2f\n",
            Roll, Pitch, Yaw, speed, traction);
        HAL_UART_Transmit(&huart2, (uint8_t*)buffer, len, HAL_MAX_DELAY);

        // Receber comando via UART
        HAL_UART_Receive(&huart2, (uint8_t*)command, sizeof(command) - 1,
HAL_MAX_DELAY);

        // Processar o comando para atualizar os parâmetros PID
        if (sscanf(command, "KP_TRACTION:%f KI_TRACTION:%f KD_TRACTION:%f
KP_VELOCITY:%f KI_VELOCITY:%f KD_VELOCITY:%f KP_POSITION:%f KI_POSITION:%f
KD_POSITION:%f",
                   &new_kp, &new_ki, &new_kd,
                   &params_velocity.kp, &params_velocity.ki,
&params_velocity.kd,
                   &params_position.kp, &params_position.ki,
&params_position.kd) == 9) {
            osMutexWait(mutexPIDParamsHandle, osWaitForever);
            params_traction.kp = new_kp;
            params_traction.ki = new_ki;
            params_traction.kd = new_kd;
            osMutexRelease(mutexPIDParamsHandle);
```

```
            // Sinaliza que os parâmetros PID foram atualizados
            osSemaphoreRelease(semPIDParamsUpdateHandle);
        }

        // Atualização dos setpoints via UART
        if (sscanf(command, "TRACTION_SETPOINT:%f %f %f
VELOCITY_SETPOINT:%f %f %f POSITION_SETPOINT:%f %f %f",
                   &new_traction_setpoint[0], &new_traction_setpoint[1],
&new_traction_setpoint[2],
                   &new_velocity_setpoint[0], &new_velocity_setpoint[1],
&new_velocity_setpoint[2],
                   &new_position_setpoint[0], &new_position_setpoint[1],
&new_position_setpoint[2]) == 9) {
            osMutexWait(mutexSetpointsHandle, osWaitForever);
            memcpy(traction_setpoint, new_traction_setpoint,
sizeof(new_traction_setpoint));
            memcpy(velocity_setpoint, new_velocity_setpoint,
sizeof(new_velocity_setpoint));
            memcpy(position_setpoint, new_position_setpoint,
sizeof(new_position_setpoint));
            osMutexRelease(mutexSetpointsHandle);

            // Sinaliza que os parâmetros PID foram atualizados
            osSemaphoreRelease(semPIDParamsUpdateHandle);
        }

        // Aguarda 500ms antes de enviar novamente
        osDelay(500);
    }
}
```

A função TaskEnvioDadosUART é responsável por enviar dados para um dispositivo externo via comunicação serial, utilizando o UART (Universal Asynchronous Receiver/Transmitter). Esta tarefa é essencial para monitorar e depurar o sistema, permitindo a transmissão de informações em tempo real sobre o estado do sistema.

No início da função, um buffer de caracteres (char buffer[256]) é definido para armazenar os dados que serão enviados via UART. O tamanho do buffer é configurado para 256 bytes, o que é suficiente para acomodar a mensagem formatada que será enviada. O

buffer é utilizado para construir uma string formatada contendo as informações a serem transmitidas.

A função snprintf é então utilizada para formatar a string que será enviada. Ela preenche o buffer com uma string que inclui as variáveis Roll, Pitch, Yaw, speed e traction. Estes são valores que representam o ângulo de rotação em três eixos (Roll, Pitch, Yaw), a velocidade do sistema e a tração, respectivamente. O formato da string é especificado como "Roll: %.2f, Pitch: %.2f, Yaw: %.2f, Speed: %.2f, Traction: %.2f\n", onde cada valor é formatado com duas casas decimais.

O snprintf retorna o comprimento da string formatada, armazenado na variável len. Esse comprimento é usado para informar a função HAL_UART_Transmit sobre a quantidade de dados que deve ser transmitida. A função HAL_UART_Transmit é então chamada para enviar os dados via UART. O primeiro parâmetro é o identificador da UART (&huart2), o segundo parâmetro é o buffer contendo os dados a serem enviados, o terceiro parâmetro é o comprimento dos dados a serem enviados (len), e o quarto parâmetro é o tempo máximo de espera (HAL_MAX_DELAY), indicando que a função deve aguardar indefinidamente até que a transmissão seja concluída.

Após o envio dos dados, a função HAL_UART_Receive é utilizada para receber comandos enviados via UART. Esses comandos podem ser utilizados para atualizar dinamicamente os parâmetros PID do sistema. A função espera indefinidamente até que um comando seja recebido, armazenando-o no array command.

O comando recebido é então processado utilizando a função sscanf, que analisa a string recebida e extrai os valores dos parâmetros PID para tração, velocidade e posição. Se todos os nove valores esperados forem recebidos corretamente, os parâmetros PID para tração (params_traction.kp, params_traction.ki, params_traction.kd) são atualizados com os novos valores recebidos. O processamento do comando permite que o sistema seja ajustado dinamicamente em tempo real, conforme necessário, sem a necessidade de recompilar ou reprogramar o código.

Além de atualizar os parâmetros PID, a função também processa comandos para atualizar os setpoints de tração, velocidade e posição. Esses novos setpoints são armazenados nas variáveis correspondentes (traction_setpoint, velocity_setpoint, position_setpoint). O acesso a essas variáveis é protegido por mutexes (mutexSetpointsHandle), que garantem que as atualizações sejam feitas de maneira segura e sem conflitos entre tarefas.

Após enviar e receber os dados, a função faz uma pausa de 500 milissegundos usando osDelay(500). Isso controla a frequência com que os dados são enviados, evitando o envio contínuo e permitindo uma transmissão em intervalos regulares. O osDelay é uma função do FreeRTOS que suspende a execução da tarefa por um período especificado.

Em resumo, a função TaskEnvioDadosUART constrói uma string formatada com informações sobre o sistema e a envia periodicamente via UART, enquanto também processa comandos recebidos para atualizar dinamicamente os parâmetros PID e os setpoints. A comunicação serial permite a visualização dos dados do sistema em tempo real, facilitando a monitoração e análise do comportamento do sistema, além de possibilitar ajustes dinâmicos nos parâmetros de controle.

A sincronização dos parâmetros PID e dos setpoints entre as tarefas é gerenciada usando mutexes e semáforos. O mutex mutexPIDParamsHandle é usado para garantir acesso exclusivo aos parâmetros PID durante a atualização e leitura. O semáforo semPIDParamsUpdateHandle é utilizado para sinalizar que os parâmetros PID foram atualizados, permitindo que outras tarefas saibam que novos valores estão disponíveis. De maneira semelhante, o mutex mutexSetpointsHandle protege o acesso aos setpoints, e o semáforo semSetpointsUpdateHandle indica que novos setpoints foram definidos. A gestão cuidadosa desses mecanismos de sincronização evita condições de corrida e garante a consistência dos dados entre as diferentes partes do sistema.

8.8. Tarefa do Controle PID dos Motores do Programa

```
// Task para controle PID dos motores
void TaskControlePID(void *argument) {
    float correnteMotor[3];
    float velocidadeMotor[3];
    float posicaoBase[3];

    float traction_setpoint[3];
    float velocity_setpoint[3];
    float position_setpoint[3];

    for (;;) {
        // Aguardar que os parâmetros PID e setpoints sejam atualizados
        osSemaphoreAcquire(semPIDParamsUpdateHandle, osWaitForever);
        osSemaphoreAcquire(semSetpointsUpdateHandle, osWaitForever);
```

```
        // Leitura dos dados das filas
        osMessageQueueGet(QueueCorrenteHandle, &correnteMotor[0], NULL,
osWaitForever);
        osMessageQueueGet(QueueVelocidadeHandle, &velocidadeMotor[0],
NULL, osWaitForever);
        osMessageQueueGet(QueuePosicaoHandle, &posicaoBase[0], NULL,
osWaitForever);

        // Copiar os setpoints atuais
        osMutexWait(mutexSetpointsHandle, osWaitForever);
        memcpy(traction_setpoint, global_traction_setpoint,
sizeof(traction_setpoint));
        memcpy(velocity_setpoint, global_velocity_setpoint,
sizeof(velocity_setpoint));
        memcpy(position_setpoint, global_position_setpoint,
sizeof(position_setpoint));
        osMutexRelease(mutexSetpointsHandle);

        // Controle PID para cada motor
        float dt = 0.01; // Intervalo de amostragem em segundos

        // Controle de tração
        float traction_command[3];
        for (int i = 0; i < 3; i++) {
            traction_command[i] = PID_Compute(&pid_traction,
traction_setpoint[i], correnteMotor[i], dt);
        }

        // Controle de velocidade
        float velocity_command[3];
        for (int i = 0; i < 3; i++) {
            velocity_command[i] = PID_Compute(&pid_velocity,
velocity_setpoint[i], velocidadeMotor[i], dt);
        }

        // Controle de posição
        float position_command[3];
        for (int i = 0; i < 3; i++) {
            position_command[i] = PID_Compute(&pid_position,
position_setpoint[i], posicaoBase[i], dt);
        }
```

```
        // Aplicação dos comandos aos motores
        Set_Motor_Commands(traction_command, velocity_command,
position_command);

        // Enviar dados via UART
        for (int i = 0; i < 3; i++) {
            Send_Motor_Data_UART(velocidadeMotor[i], traction_command[i],
posicaoBase[i]);
        }

        // Aguarda 10ms antes de executar novamente
        osDelay(10);
    }
}
```

A função TaskControlePID é responsável pela execução do controle PID para os motores e pelo envio de dados através da interface UART. O código começa declarando três arrays de variáveis float: correnteMotor, velocidadeMotor e posicaoBase, que são utilizados para armazenar as leituras dos sensores correspondentes aos motores e à base do sistema.

Dentro do loop infinito, a tarefa inicia com a espera pelos sinais dos semáforos que indicam que os parâmetros PID e os setpoints foram atualizados. A função osSemaphoreAcquire é chamada para aguardar indefinidamente até que semPIDParamsUpdateHandle e semSetpointsUpdateHandle sejam liberados, garantindo que os parâmetros e setpoints mais recentes estejam disponíveis antes de proceder com o controle.

A leitura dos dados dos sensores é então realizada através das filas de mensagens. A função osMessageQueueGet é utilizada para recuperar os valores de corrente dos motores, velocidade e posição da base das filas QueueCorrenteHandle, QueueVelocidadeHandle e QueuePosicaoHandle, respectivamente. O parâmetro osWaitForever garante que a tarefa espere indefinidamente até que os dados estejam disponíveis, assegurando a atualização correta das variáveis.

Após a leitura dos dados, os setpoints atuais são copiados para variáveis locais. Isso é feito através da função osMutexWait, que garante acesso exclusivo ao mutex mutexSetpointsHandle durante a cópia dos valores. A função memcpy é utilizada para copiar os setpoints de tração, velocidade e posição das variáveis globais global_traction_setpoint, global_velocity_setpoint e global_position_setpoint para as variáveis locais traction_setpoint,

velocity_setpoint e position_setpoint. Após a cópia, o mutex é liberado com osMutexRelease, permitindo que outras tarefas acessem os setpoints.

O controle PID é então calculado para cada motor. O intervalo de amostragem é definido como 0.01 segundos, o que corresponde a 10 milissegundos. A função PID_Compute é chamada para calcular os comandos de controle PID com base nos valores de corrente, velocidade e posição dos motores. Para cada tipo de controle (tração, velocidade e posição), um array de comandos (traction_command, velocity_command, position_command) é preenchido com os valores calculados pelos respectivos controladores PID. Os setpoints são usados como referência, enquanto as leituras dos sensores fornecem o feedback necessário para o cálculo do erro e ajuste do controle.

Uma vez que os comandos de controle PID para tração, velocidade e posição são calculados, a função Set_Motor_Commands é chamada para aplicar esses comandos aos motores. Esta função ajusta os sinais de controle dos motores de acordo com os valores calculados, assegurando que os motores respondam corretamente aos comandos de tração, velocidade e posição.

Após aplicar os comandos, a função Send_Motor_Data_UART é utilizada para enviar os dados dos motores via UART. Para cada motor, a função transmite a velocidade, o comando de tração e a posição atual, permitindo a monitoração e análise remota do estado dos motores.

Finalmente, a tarefa aguarda 10 milissegundos antes de iniciar a próxima iteração do loop, utilizando a função osDelay. Esse atraso define a frequência com que a tarefa é executada, permitindo uma atualização periódica das leituras e dos comandos. O intervalo de 10 milissegundos é ajustável e deve ser escolhido com base nas necessidades do sistema e na precisão requerida para o controle.

Em resumo, TaskControlePID gerencia o controle PID dos motores lendo dados dos sensores, calculando os comandos de controle com base nos valores lidos, aplicando esses comandos aos motores e transmitindo e recebendo informações via UART. A tarefa opera em um ciclo contínuo com um intervalo de 10 milissegundos, garantindo uma atualização periódica e eficiente do sistema de controle. A sincronização com semáforos e mutexes assegura a consistência dos dados e evita condições de corrida entre as tarefas que manipulam os parâmetros PID e os setpoints.

8.9. Tarefa do Controle PID dos Motores do Programa

```
// Função para aplicar comandos aos motores
void Set_Motor_Commands(float traction_command[3], float
velocity_command[3], float position_command[3]) {
    // Definir limites máximos e mínimos para os comandos dos motores
    const float MAX_PWM = 255.0f;  // Valor máximo do PWM (ajustar
conforme necessário)
    const float MIN_PWM = 0.0f;    // Valor mínimo do PWM (ajustar
conforme necessário)

    // Variáveis para os comandos finais dos motores
    float motor_command[3][2]; // [0] para frente, [1] para trás

    // Ajustar os comandos finais dos motores com base nas combinações dos
comandos de tração e velocidade
    for (int i = 0; i < 3; i++) {
        float total_command = traction_command[i] + velocity_command[i] +
position_command[i];

        // Saturar o comando para garantir que ele esteja dentro da faixa
[-1, 1]
        if (total_command > 1.0f) total_command = 1.0f;
        if (total_command < -1.0f) total_command = -1.0f;

        // Determinar PWM para frente e para trás
        if (total_command > 0) {
            motor_command[i][0] = total_command * MAX_PWM;
            motor_command[i][1] = 0;
        } else {
            motor_command[i][0] = 0;
            motor_command[i][1] = -total_command * MAX_PWM;
        }
    }

    // Aplicar os comandos PWM aos motores (ajustar conforme necessário)
    __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1,
(uint32_t)motor_command[0][0]);
    __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2,
(uint32_t)motor_command[1][0]);
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1,
(uint32_t)motor_command[2][0]);
```

```
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2,
(uint32_t)motor_command[0][1]);
    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1,
(uint32_t)motor_command[1][1]);
    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2,
(uint32_t)motor_command[2][1]);
}
```

A função Set_Motor_Commands é projetada para aplicar comandos de controle aos motores, ajustando os sinais de PWM para cada motor com base nos comandos de tração, velocidade e posição. A função começa definindo os limites máximos e mínimos para os sinais de PWM. O valor máximo de PWM é estabelecido em 255.0, enquanto o mínimo é 0.0. Esses valores são ajustáveis conforme as necessidades do sistema.

Em seguida, a função declara um array bidimensional motor_command de tamanho 3x2. Este array é usado para armazenar os comandos finais de PWM para cada motor, sendo que o índice [0] refere-se ao comando de PWM para frente e o índice [1] refere-se ao comando de PWM para trás.

Para cada motor (o loop percorre de 0 a 2, representando os três motores), a função calcula um total_command, que é a soma dos comandos de tração, velocidade e posição para o motor atual. Este valor é então saturado para garantir que fique dentro do intervalo permitido de [-1, 1]. Se total_command exceder 1.0, ele é ajustado para 1.0. Se for menor que -1.0, é ajustado para -1.0. Esta saturação é importante para garantir que os comandos não excedam os limites operacionais dos motores.

Depois de ajustar o total_command, a função determina o comando de PWM para frente e para trás para cada motor. Se o total_command for positivo, o comando de PWM para frente é definido como total_command multiplicado pelo valor máximo de PWM (MAX_PWM), enquanto o comando de PWM para trás é definido como 0. Se o total_command for negativo, o comando de PWM para frente é definido como 0 e o comando de PWM para trás é ajustado para o valor absoluto de total_command multiplicado por MAX_PWM. Isso garante que o motor receba o sinal correto para mover-se na direção desejada.

Finalmente, a função aplica os comandos de PWM aos motores utilizando a função __HAL_TIM_SET_COMPARE. Esta função é responsável por ajustar o valor de comparação para os canais PWM dos temporizadores. Os comandos de PWM são configurados para os canais apropriados dos temporizadores associados aos motores. Os canais de PWM são distribuídos conforme o motor e a direção (para frente ou para trás).

Em resumo, Set_Motor_Commands ajusta os sinais de PWM dos motores com base nos comandos combinados de tração, velocidade e posição, garantindo que os comandos estejam dentro dos limites operacionais e aplicando os sinais adequados para a movimentação correta dos motores.

8.10. Função para configurar o sistema de clock do Microcontrolador

```
// Funções de Configuração do Hardware
void SystemClock_Config(void) {
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    RCC_OscInitStruct.HSEState = RCC_HSE_ON;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    RCC_OscInitStruct.PLL.PLLM = 8;
    RCC_OscInitStruct.PLL.PLLN = 336;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
    RCC_OscInitStruct.PLL.PLLQ = 7;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
        Error_Handler();
    }

    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK |
RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) !=
HAL_OK) {
        Error_Handler();
    }
}
```

A função SystemClock_Config é responsável pela configuração do sistema de clock do microcontrolador, assegurando que o relógio do sistema e os diversos periféricos sejam configurados corretamente para operar com as frequências desejadas. Esta função é crucial para garantir que o microcontrolador funcione dentro das especificações e com a precisão necessária para o correto funcionamento dos aplicativos.

A função começa definindo e inicializando duas estruturas de configuração: RCC_OscInitTypeDef e RCC_ClkInitTypeDef. Essas estruturas são utilizadas para configurar o oscilador e o clock do sistema.

Primeiramente, a estrutura RCC_OscInitTypeDef é configurada para definir o tipo de oscilador e as configurações do PLL (Phase-Locked Loop). O campo OscillatorType é definido como RCC_OSCILLATORTYPE_HSE, indicando que o oscilador de alta frequência externo (HSE - High-Speed External) será utilizado. O campo HSEState é configurado para RCC_HSE_ON, o que significa que o oscilador HSE está ligado. Em seguida, o PLL é ativado através do campo PLLState configurado para RCC_PLL_ON. A fonte do PLL é definida como o oscilador HSE, através de PLLSource configurado para RCC_PLLSOURCE_HSE.

Os parâmetros do PLL são ajustados para configurar a frequência do clock do sistema. PLL_M é definido como 8, o que configura o divisor do oscilador HSE. PLL_N é definido como 336, especificando o multiplicador do PLL. PLL_P é configurado como RCC_PLLP_DIV2, indicando que o clock do sistema será dividido por 2. Finalmente, PLL_Q é definido como 7, ajustando o divisor de saída do PLL.

Após configurar os parâmetros do PLL, a função HAL_RCC_OscConfig é chamada para aplicar essas configurações. Se a função retornar um erro, o Error_Handler é acionado para lidar com a falha.

Em seguida, a estrutura RCC_ClkInitTypeDef é configurada para definir a fonte e os divisores dos clocks dos diversos barramentos e sistemas. O campo ClockType é configurado para incluir todos os tipos de clock relevantes: RCC_CLOCKTYPE_HCLK, RCC_CLOCKTYPE_SYSCLK, RCC_CLOCKTYPE_PCLK1 e RCC_CLOCKTYPE_PCLK2. SYSCLKSource é definido como RCC_SYSCLKSOURCE_PLLCLK, indicando que o clock do sistema será fornecido pelo PLL. Os divisores dos clocks são então configurados: AHBCLKDivider é definido como RCC_SYSCLK_DIV1, significando que o clock do barramento AHB é igual ao clock do sistema; APB1CLKDivider é definido como RCC_HCLK_DIV4, configurando o divisor do clock do barramento APB1; e APB2CLKDivider é configurado como RCC_HCLK_DIV2, ajustando o divisor do clock do barramento APB2.

A função HAL_RCC_ClockConfig é então designada por aplicar essas configurações de clock. O parâmetro FLASH_LATENCY_5 é passado para configurar a latência de memória flash para 5 ciclos, garantindo que a memória flash opere corretamente com a nova configuração de clock. Se a configuração falhar, a função Error_Handler é chamada.

Em resumo, a função SystemClock_Config configura o sistema de clock do microcontrolador, definindo a fonte e os parâmetros do PLL, ajustando os divisores dos clocks dos diversos barramentos e aplicando as configurações. Isso é essencial para garantir que o microcontrolador opere nas frequências corretas para o desempenho ideal do sistema.

8.11. Função para configurar o GPIO do Microcontrolador

```
// Inicialização dos GPIOs
static void MX_GPIO_Init(void) {
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();

    // Configuração dos pinos dos motores e outros periféricos
    GPIO_InitStruct.Pin = GPIO_PIN_12 | GPIO_PIN_13 | GPIO_PIN_14 |
GPIO_PIN_15;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    GPIO_InitStruct.Alternate = GPIO_AF1_TIM1;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = GPIO_PIN_8 | GPIO_PIN_9 | GPIO_PIN_10 |
GPIO_PIN_11;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    GPIO_InitStruct.Alternate = GPIO_AF1_TIM1;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}
```

A função MX_GPIO_Init é responsável pela configuração inicial dos pinos de entrada e saída (GPIOs) do microcontrolador, configurando-os para operar de acordo com as necessidades específicas do sistema, como controle de motores e outros periféricos. Esta configuração é crucial para garantir que os pinos estejam corretamente preparados para suas funções desejadas, como geração de sinais PWM ou comunicação com outros dispositivos.

O processo começa com a habilitação dos clocks para os portos GPIO necessários. A função __HAL_RCC_GPIOA_CLK_ENABLE(), __HAL_RCC_GPIOB_CLK_ENABLE(), e __HAL_RCC_GPIOC_CLK_ENABLE() são chamadas para ativar os relógios para os portos GPIO A, B e C, respectivamente. Esta etapa é essencial, pois o microcontrolador só pode usar os periféricos GPIO se o clock correspondente estiver habilitado.

Após habilitar os clocks, a configuração dos pinos é realizada utilizando a estrutura GPIO_InitTypeDef, que é preenchida com os parâmetros de configuração necessários. Inicialmente, a estrutura GPIO_InitStruct é configurada com zeros para garantir que todos os campos sejam limpos antes de definir os parâmetros específicos.

Os pinos do porto GPIOB são configurados primeiro. A configuração define que os pinos 12, 13, 14 e 15 serão usados para funções alternativas (AF) e que operarão no modo de push-pull (modo de saída de alta corrente). O parâmetro GPIO_InitStruct.Mode é definido como GPIO_MODE_AF_PP, indicando que os pinos serão configurados para operar em modo de função alternativa de push-pull. O parâmetro GPIO_InitStruct.Pull é configurado como GPIO_NOPULL, o que significa que nenhum resistor de pull-up ou pull-down será utilizado. A velocidade de operação dos pinos é configurada como GPIO_SPEED_FREQ_LOW, o que define a frequência de operação como baixa. O campo GPIO_InitStruct.Alternate é configurado como GPIO_AF1_TIM1, especificando que os pinos serão associados à função alternativa 1, que neste caso é o temporizador TIM1. A função HAL_GPIO_Init(GPIOB, &GPIO_InitStruct) aplica essas configurações aos pinos do porto GPIOB.

A configuração dos pinos do porto GPIOA segue um processo semelhante. Os pinos 8, 9, 10 e 11 são configurados para operar em modo de função alternativa, sem resistores de pull-up ou pull-down, com uma velocidade de operação baixa e associados à função alternativa 1 do temporizador TIM1. A configuração é aplicada utilizando a função HAL_GPIO_Init(GPIOA, &GPIO_InitStruct).

Em resumo, a função MX_GPIO_Init realiza a configuração inicial dos pinos GPIO, incluindo a habilitação dos clocks dos portos GPIO necessários e a definição dos parâmetros de configuração para os pinos de controle dos motores e outros periféricos. Esta configuração prepara os pinos para operar em modo de função alternativa, associado ao temporizador TIM1, com uma frequência de operação baixa e sem resistores de pull-up ou pull-down.

8.12. Função para configurar o ADC do Microcontrolador

```
// Inicialização do ADC1
static void MX_ADC1_Init(void) {
    ADC_ChannelConfTypeDef sConfig = {0};
    hadc1.Instance = ADC1;
    hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
    hadc1.Init.Resolution = ADC_RESOLUTION_12B;
    hadc1.Init.ScanConvMode = ENABLE;
```

```
    hadc1.Init.ContinuousConvMode = DISABLE;
    hadc1.Init.DiscontinuousConvMode = DISABLE;
    hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc1.Init.NbrOfConversion = 1;
    hadc1.Init.DMAContinuousRequests = DISABLE;
    hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    if (HAL_ADC_Init(&hadc1) != HAL_OK) {
        Error_Handler();
    }
    sConfig.Channel = ADC_CHANNEL_6;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
    if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) {
        Error_Handler();
    }
}
```

A função MX_ADC1_Init realiza a configuração do conversor analógico-digital (ADC) do microcontrolador, especificamente o ADC1. Este processo é crucial para garantir que o ADC funcione corretamente para a leitura de sinais analógicos. A função configura o ADC com parâmetros específicos que determinam seu comportamento e características.

O processo inicia com a definição e configuração da estrutura ADC_ChannelConfTypeDef, chamada sConfig. Esta estrutura é usada para configurar os canais do ADC e seu comportamento durante a conversão.

Primeiramente, o ponteiro hadc1.Instance é configurado para apontar para o ADC1, especificando que a configuração será aplicada ao ADC1. Em seguida, a configuração do ADC é realizada por meio da estrutura hadc1.Init, que é preenchida com os parâmetros necessários.

O campo ClockPrescaler é configurado como ADC_CLOCK_SYNC_PCLK_DIV4, o que define o prescaler do relógio do ADC, indicando que a frequência do clock do ADC será dividida por 4 em relação ao clock do periférico. Isso ajusta a velocidade de amostragem do ADC.

O campo Resolution é configurado como ADC_RESOLUTION_12B, especificando que o ADC terá uma resolução de 12 bits. Isso significa que o valor digital obtido após a conversão analógica será um número de 12 bits.

O campo ScanConvMode é configurado como ENABLE, indicando que o ADC está em modo de varredura, o que permite a conversão de múltiplos canais em sequência. No entanto, a configuração hadc1.Init.NbrOfConversion é definida como 1, o que significa que apenas um canal será convertido por vez.

O campo ContinuousConvMode é configurado como DISABLE, o que desativa o modo de conversão contínua. Isso significa que o ADC não continuará convertendo automaticamente após a conclusão de uma conversão.

O campo DiscontinuousConvMode é configurado como DISABLE, indicando que o ADC não usará o modo de conversão descontínua, que é útil para dividir a conversão em várias fases.

O campo ExternalTrigConvEdge é configurado como ADC_EXTERNALTRIGCONVEDGE_NONE, o que indica que a conversão do ADC não será acionada por um sinal externo.

O campo DataAlign é configurado como ADC_DATAALIGN_RIGHT, definindo que os dados de conversão serão alinhados à direita, o que significa que os bits menos significativos da conversão serão os primeiros a serem armazenados.

O campo NbrOfConversion é configurado como 1, o que significa que apenas um canal será convertido durante cada operação de conversão.

O campo DMAContinuousRequests é configurado como DISABLE, o que indica que o ADC não fará solicitações contínuas ao DMA (Direct Memory Access).

O campo EOCSelection é configurado como ADC_EOC_SINGLE_CONV, o que define que o sinal de fim de conversão (End of Conversion, EOC) será gerado após a conclusão de uma única conversão.

Após a configuração da estrutura hadc1.Init, a função HAL_ADC_Init(&hadc1) é chamada para inicializar o ADC com os parâmetros especificados. Se a inicialização falhar, a função Error_Handler() é chamada para lidar com o erro.

Em seguida, a estrutura sConfig é configurada para especificar o canal do ADC. O campo Channel é configurado como ADC_CHANNEL_6, indicando que o canal 6 do ADC será usado para a conversão. O campo Rank é definido como 1, o que define a ordem de conversão no modo de varredura (se vários canais fossem usados). O campo SamplingTime é

configurado como ADC_SAMPLETIME_3CYCLES, definindo o tempo de amostragem do canal como 3 ciclos de clock do ADC.

A função HAL_ADC_ConfigChannel(&hadc1, &sConfig) é então chamada para configurar o canal do ADC com os parâmetros especificados. Se a configuração do canal falhar, a função Error_Handler() é chamada para tratar o erro.

Em resumo, a função MX_ADC1_Init configura o ADC1 do microcontrolador definindo seu modo de operação, resolução, e características específicas de conversão. Após configurar o ADC, o canal 6 é selecionado e configurado para a leitura, garantindo que o ADC esteja pronto para realizar a conversão dos sinais analógicos conforme necessário.

8.13. Função para configurar o Timer do Microcontrolador

```
// Inicialização do Timer 1
static void MX_TIM1_Init(void) {
    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};
    TIM_OC_InitTypeDef sConfigOC = {0};

    htim1.Instance = TIM1;
    htim1.Init.Prescaler = 0;
    htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim1.Init.Period = 0xFFFF;
    htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim1.Init.RepetitionCounter = 0;
    htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim1) != HAL_OK) {
        Error_Handler();
    }

    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK)
{
        Error_Handler();
    }

    if (HAL_TIM_PWM_Init(&htim1) != HAL_OK) {
        Error_Handler();
    }
```

```
    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) !=
HAL_OK) {
        Error_Handler();
    }

    sConfigOC.OCMode = TIM_OCMODE_PWM1;
    sConfigOC.Pulse = 0;
    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
    sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1) !=
HAL_OK) {
        Error_Handler();
    }

    HAL_TIM_MspPostInit(&htim1);
}
```

A função `MX_TIM1_Init` é responsável pela configuração do Timer 1 no microcontrolador. O Timer 1 é um periférico que pode ser usado para uma variedade de aplicações, como temporização, geração de PWM (Pulse Width Modulation), e captura de sinais. A função realiza uma série de passos para configurar o Timer 1 e prepará-lo para uso.

O processo começa com a definição e inicialização das estruturas necessárias para a configuração do Timer 1. A função utiliza três estruturas principais: `TIM_ClockConfigTypeDef`, `TIM_MasterConfigTypeDef`, e `TIM_OC_InitTypeDef`. Essas estruturas são usadas para configurar o timer, sua fonte de clock, e o modo de operação do PWM.

Primeiramente, a estrutura `htim1.Instance` é configurada para apontar para o Timer 1, que é o periférico que será configurado. A estrutura `htim1.Init` é então preenchida com os parâmetros de configuração do timer. O campo `Prescaler` é definido como 0, o que significa que o valor do prescaler do timer é 1, e o timer operará com a frequência do clock principal. O campo `CounterMode` é configurado como `TIM_COUNTERMODE_UP`, o que indica que o timer contará de 0 até o valor do período.

O campo `Period` é definido como `0xFFFF`, o que significa que o timer contará até o valor máximo de 16 bits (65535) antes de reiniciar. O campo `ClockDivision` é configurado

como `TIM_CLOCKDIVISION_DIV1`, o que define a divisão do clock como 1, ou seja, sem divisão adicional do clock do timer. O campo `RepetitionCounter` é configurado como 0, o que significa que não há repetição de contagem. O campo `AutoReloadPreload` é configurado como `TIM_AUTORELOAD_PRELOAD_DISABLE`, indicando que o preload do valor de recarga está desativado.

Após a configuração da estrutura `htim1.Init`, a função `HAL_TIM_Base_Init(&htim1)` é chamada para inicializar o Timer 1 com as configurações especificadas. Se a inicialização falhar, a função `Error_Handler()` é chamada para lidar com o erro.

Em seguida, a configuração da fonte de clock do timer é realizada. A estrutura `sClockSourceConfig` é configurada com `ClockSource` definido como `TIM_CLOCKSOURCE_INTERNAL`, indicando que a fonte de clock do timer será interna. A função `HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig)` é chamada para aplicar essa configuração. Se a configuração falhar, a função `Error_Handler()` é chamada.

Após configurar a fonte de clock, o modo PWM (Pulse Width Modulation) do Timer 1 é inicializado. A função `HAL_TIM_PWM_Init(&htim1)` é chamada para preparar o timer para operar no modo PWM. Se a inicialização falhar, a função `Error_Handler()` é chamada.

A configuração mestre do timer é realizada a seguir. A estrutura `sMasterConfig` é configurada com `MasterOutputTrigger` definido como `TIM_TRGO_RESET`, o que indica que o timer não gerará um sinal de disparo para outros periféricos. O campo `MasterSlaveMode` é configurado como `TIM_MASTERSLAVEMODE_DISABLE`, indicando que o timer não está em modo mestre/slave. A função `HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig)` é chamada para aplicar essa configuração. Se a configuração falhar, a função `Error_Handler()` é chamada.

Finalmente, a configuração do modo de saída comparadora (Output Compare) do timer é realizada. A estrutura `sConfigOC` é configurada para o modo PWM1, com o campo `Pulse` definido como 0, o que significa que o comprimento do pulso inicial é 0. O campo `OCPolarity` é configurado como `TIM_OCPOLARITY_HIGH`, o que define a polaridade do sinal de saída como alta. O campo `OCNPolarity` é configurado como `TIM_OCNPOLARITY_HIGH`, e `OCFastMode` é configurado como `TIM_OCFAST_DISABLE`, desativando o modo rápido. Os campos `OCIdleState` e `OCNIdleState` são configurados como `TIM_OCIDLESTATE_RESET` e `TIM_OCNIDLESTATE_RESET`, respectivamente, indicando que o estado de inatividade é resetado.

A função `HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1)` é chamada para aplicar a configuração do modo PWM ao canal 1 do Timer 1. Se a configuração falhar, a função `Error_Handler()` é chamada.

Por fim, a função `HAL_TIM_MspPostInit(&htim1)` é chamada para realizar qualquer configuração adicional necessária após a inicialização do timer. Isso pode incluir configurações específicas de hardware ou ajustes finais no periférico.

Em resumo, a função `MX_TIM1_Init` configura o Timer 1 para operar no modo PWM, define a fonte de clock e aplica as configurações necessárias para garantir que o timer esteja pronto para uso em aplicações que exigem temporização ou modulação por largura de pulso.

8.14. Função para configurar o USART do Microcontrolador

```
// Inicialização da comunicação Serial
void MX_USART2_UART_Init(void)
{
    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_TX_RX;
    huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart2.Init.OverSampling = UART_OVERSAMPLING_16;
    if (HAL_UART_Init(&huart2) != HAL_OK){
        Error_Handler();
    }
}
```

A função MX_USART2_UART_Init tem a responsabilidade de configurar e inicializar o módulo USART2 do microcontrolador para comunicação serial. O objetivo desta configuração é garantir que o periférico UART funcione corretamente de acordo com os requisitos da aplicação, permitindo a transmissão e recepção de dados seriais. O processo é realizado em várias etapas, cada uma com uma configuração específica para o funcionamento da UART.

Primeiramente, a função define a instância do periférico UART que será configurado. Neste caso, huart2.Instance é definido como USART2, indicando que a configuração será aplicada ao módulo USART2 do microcontrolador. huart2 é uma estrutura do tipo UART_HandleTypeDef que contém todas as informações necessárias para gerenciar a UART, incluindo o estado e a configuração do periférico.

Em seguida, são configurados vários parâmetros essenciais para a operação da UART. Estes parâmetros incluem a taxa de transmissão, o comprimento da palavra, o número de bits

de parada, o tipo de paridade, o modo de operação, o controle de fluxo de hardware e a sobreamostragem. Cada um desses parâmetros é ajustado para atender às necessidades específicas da comunicação serial:

- **Taxa de Transmissão (Baud Rate):** A configuração huart2.Init.BaudRate = 115200; define a taxa de transmissão dos dados como 115200 bps (bits por segundo). Esta taxa é uma escolha comum para comunicação serial e determina a velocidade com que os dados serão transmitidos e recebidos pelo módulo UART.
- **Comprimento da Palavra:** O comprimento de cada palavra de dados é configurado com huart2.Init.WordLength = UART_WORDLENGTH_8B;. Isso define que cada palavra de dados terá 8 bits. O comprimento da palavra é importante para garantir que os dados sejam interpretados corretamente tanto pelo transmissor quanto pelo receptor.
- **Número de Bits de Parada:** A configuração huart2.Init.StopBits = UART_STOPBITS_1; determina que haverá um bit de parada na comunicação. O bit de parada é utilizado para marcar o final de uma transmissão de dados e garantir que o receptor saiba quando os dados terminam.
- **Paridade:** A configuração huart2.Init.Parity = UART_PARITY_NONE; define que não será utilizada verificação de paridade na comunicação. A paridade é uma forma de verificação de erros, e neste caso, está desativada, significando que a comunicação não incluirá bits de paridade para verificar a integridade dos dados transmitidos.
- **Modo de Operação:** A configuração huart2.Init.Mode = UART_MODE_TX_RX; define que a UART será configurada para operar tanto na transmissão (TX) quanto na recepção (RX) de dados. Isso permite que o periférico UART envie e receba dados simultaneamente.
- **Controle de Fluxo de Hardware:** A configuração huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE; especifica que o controle de fluxo de hardware, que normalmente envolve sinais como CTS (Clear to Send) e RTS (Request to Send), não será utilizado. O controle de fluxo de hardware é opcional e, nesse caso, está desativado.
- **Sobreamostragem:** A configuração huart2.Init.OverSampling = UART_OVERSAMPLING_16; define que a UART usará uma sobreamostragem de 16 vezes. A sobreamostragem é uma técnica que melhora a precisão da comunicação serial, especialmente em altas taxas de transmissão, ao amostrar o sinal mais vezes do que a taxa de baud.

Após definir todos esses parâmetros, a função chama HAL_UART_Init(&huart2) para inicializar o módulo UART com as configurações especificadas. Se a inicialização falhar por

qualquer motivo, a função Error_Handler() é chamada para lidar com a falha. Esta função de tratamento de erros é responsável por implementar medidas apropriadas quando ocorre uma falha na inicialização, garantindo que o sistema possa responder adequadamente a problemas.

Em resumo, a função MX_USART2_UART_Init configura e inicializa o módulo USART2 do microcontrolador para comunicação serial com parâmetros específicos de taxa de transmissão, comprimento da palavra, bits de parada, paridade, modo de operação, controle de fluxo de hardware e sobreamostragem. A função garante que a UART esteja pronta para transmitir e receber dados conforme configurado, e trata possíveis falhas durante o processo de inicialização.

8.15. Função para configurar WATCHDOG e ERRO do Microcontrolador

```
// Inicialização do IWDG (Independent Watchdog Timer)
static void MX_IWDG_Init(void) {
    hiwdg.Instance = IWDG;
    hiwdg.Init.Prescaler = IWDG_PRESCALER_64;
    hiwdg.Init.Reload = 4095;
    if (HAL_IWDG_Init(&hiwdg) != HAL_OK) {
        Error_Handler();
    }
}

// Função de tratamento de erro
void Error_Handler(void) {
    __disable_irq();
    while (1) {}
}
```

A função MX_IWDG_Init realiza a configuração e inicialização do temporizador watchdog independente (IWDG) do microcontrolador, um componente essencial para a garantia da operação confiável do sistema, uma vez que o IWDG ajuda a reiniciar o microcontrolador em caso de falhas ou travamentos do software.

Para iniciar, a função define a instância do watchdog por meio da estrutura hiwdg, configurando o campo Instance com IWDG, o que indica que o módulo específico do hardware IWDG será utilizado. Em seguida, o código configura o prescaler do IWDG, ajustando o valor de Prescaler para IWDG_PRESCALER_64. Este fator de prescaler é responsável por reduzir a frequência do clock do watchdog, dividindo-a por 64. Essa configuração ajusta a velocidade com a qual o temporizador conta, influenciando diretamente o período durante o qual o watchdog deve ser reiniciado para evitar uma reinicialização automática do sistema.

O campo Reload é então definido com o valor 4095, que é o valor máximo que um temporizador de 12 bits pode armazenar. Esse valor determina o intervalo de tempo máximo que o watchdog pode contar antes de gerar um evento de reinício. Com o valor de reload ajustado para 4095, o temporizador terá o maior período possível antes de expirar, garantindo que o sistema tenha bastante tempo para reiniciar o watchdog antes que ocorra um reinício não desejado.

Depois de definir esses parâmetros, a função chama HAL_IWDG_Init(&hiwdg), que aplica as configurações definidas e inicializa o watchdog. Caso a inicialização falhe, a função retorna um erro e chama Error_Handler, uma função projetada para lidar com situações de falha crítica.

A função Error_Handler atua como um mecanismo de tratamento de erros, desabilitando todas as interrupções globais do microcontrolador com __disable_irq(). Esse comando assegura que nenhuma interrupção adicional possa interferir ou agravar a situação durante o tratamento do erro. Em seguida, o código entra em um loop infinito com while (1) {}. Esse comportamento é uma medida de segurança que impede a execução adicional de código, mantendo o sistema em um estado estático e evitando que continue operando em uma condição instável. O loop infinito garante que o sistema não continue a executar operações que poderiam causar mais problemas, servindo como uma forma de bloqueio seguro em caso de erros críticos.

Portanto, a função MX_IWDG_Init configura o temporizador watchdog independente com um prescaler de 64 e um valor de reload de 4095, e a função Error_Handler é utilizada para gerenciar falhas críticas ao desabilitar interrupções e manter o sistema em um estado de espera contínua. Essa abordagem ajuda a garantir a estabilidade e a recuperação do sistema em situações de erro.

9. CONCLUSÃO

O desenvolvimento do protótipo de código para o controle PID de motores com encoders e sensores de corrente foi concluído. No entanto, o código gerado é apenas uma etapa inicial e precisa ser testado em conjunto com o controlador e o hardware completo para verificar sua eficácia real.

É importante ressaltar que o sucesso do sistema depende das propostas e especificações fornecidas pelas outras equipes, e os valores usados no código protótipo precisarão ser ajustados para refletir o protótipo real e suas limitações. A próxima fase do projeto envolverá a integração do sistema e a realização de testes práticos para ajustar o controle PID e garantir que ele atenda às necessidades do sistema final.

O projeto nos ofereceu a oportunidade de explorar a estrutura de programação para sistemas embarcados e entender como simular ambientes para proporcionar experiências em tempo real para os usuários. Enfrentamos diversas complicações durante o desenvolvimento, mas o uso do FreeRTOS, uma estrutura amplamente empregada, foi crucial para superar esses desafios. Este projeto foi fundamental para o aprimoramento do nosso conhecimento na área de sistemas embarcados, permitindo-nos aplicar e consolidar os conceitos aprendidos em sala de aula de forma prática e efetiva.

10. REFERÊNCIAS BIBLIOGRÁFICAS

[1] **FIAIS**, Jesse de Jesus. **Materiais de Suporte para o Desenvolvimento do Projeto Final utilizando FreeRTOS**. Escola Politécnica da Universidade Federal da Bahia (UFBA). Departamento de Engenharia Elétrica e da Computação (DEEC). Salvador (BA), Julho de 2024.

[2] **IGINO**, Wellington Passos. **Ensino de Sistemas Embarcados Baseado em Projeto: Exemplo Aplicado à Robótica**. Dissertação de Mestrado. Departamento de Engenharia Elétrica e da Computação (DEEC) da Universidade Federal da Bahia (UFBA). Salvador (BA). Dezembro, 2023.

[3] **RIBEIRO**, Fernando; **MOUTINHO**, I. **SILVA**, Pedro; **FRAGA**, Carlos; **PEREIRA**, Nino. **Three omni-directional wheels control on a mobile robot.** Grupo de Automação e Robótica, Departamento de Eletrotécnica Industrial da Universidade do Minho. Campus Azurém, Guimarães, Portugal. Setembro, 2004.

[4] **MESTIRI**, Youssef. **Mobile Manipulator Robot: Omni 3 Wheels Manipulator Robot**. Dissertação de Mestrado. Instituto Politécnico de Bragança. 2021.

11. ANEXO - PROJETO COMPLETO

```
// Universidade Federal da Bahia (UFBA)
// Departamento de Engenharia Elétrica e da Computação (DEEC)
// Equipe: Gabriel, Márcio e Hugo
// Matéria: Programação em Tempo Real para Sist. Embarcados
// Professor: Jess Fiais

// CONTROLE DE VELOCIDADE DOS MOTORES DE UM ROBÔ OMNIDIRECIONAL

// Importação de Bibliotacas
#include "main.h"
#include "cmsis_os.h"
#include "adc.h"
#include "tim.h"
#include "gpio.h"
#include "iwdg.h"
#include "stm32f4xx_hal.h"
#include <math.h>
#include <stdint.h>
#include <stdbool.h>

// Definição das Queues
osMessageQueueId_t QueueCorrenteHandle;
osMessageQueueId_t QueueVelocidadeHandle;
osMessageQueueId_t QueuePosicaoHandle;

// Definição dos PID Controllers
PID_Controller pid_traction;
PID_Controller pid_velocity;
PID_Controller pid_position;

// Definição dos PID Controllers
PID_TypeDef PID_Motor1, PID_Motor2, PID_Motor3;

// Definição das variáveis globais
volatile int32_t encoderReading1 = 0;
volatile int32_t encoderReading2 = 0;
volatile int32_t encoderReading3 = 0;
double Roll, Pitch, Yaw;
float speed, traction;
double Motor1_Setpoint, Motor2_Setpoint, Motor3_Setpoint;
double Motor1_Input, Motor2_Input, Motor3_Input;
```

```c
double Motor1_Output, Motor2_Output, Motor3_Output;

// Prototipação das Funções das Tasks
void TaskControleCorrente(void *argument);
void TaskControleVelocidade(void *argument);
void TaskControlePosicao(void *argument);
void TaskControlePID(void *argument);
void TaskEnvioDadosUART(void *argument);

// Protótipos de funções
void PID_Compute(PID_TypeDef *pid);
void PID_Init(PID_TypeDef *pid, double kp, double ki, double kd);
double Read_Angle_Roll(void);
double Read_Angle_Pitch(void);
double Read_Angle_Yaw(void);
void Error_Handler(void);

// Função para Inicialização de Periféricos
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_ADC1_Init(void);
static void MX_ADC2_Init(void);
static void MX_ADC3_Init(void);
static void MX_TIM1_Init(void);
static void MX_TIM2_Init(void);
static void MX_TIM3_Init(void);
static void MX_IWDG_Init(void);
static void MX_USART2_UART_Init(void);

int main(void) {
    // Inicialização do hardware, do sistema e dos periféricos
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_ADC1_Init();
    MX_ADC2_Init();
    MX_ADC3_Init();
    MX_TIM1_Init();
    MX_TIM2_Init();
    MX_TIM3_Init();
    MX_IWDG_Init();
    MX_USART2_UART_Init();
```

```
    // Inicialização das variáveis globais
    Roll = 0.0f;
    Pitch = 0.0f;
    Yaw = 0.0f;
    speed = 0.0f;
    traction = 0.0f;
    Motor1_Setpoint = 0.0f;
    Motor2_Setpoint = 0.0f;
    Motor3_Setpoint = 0.0f;

    // Inicialização das Queues
    QueueCorrenteHandle = osMessageQueueNew(3, sizeof(float), NULL);
//Tamanho ajustável da fila em 3
    QueueVelocidadeHandle = osMessageQueueNew(3, sizeof(float), NULL);
//Tamanho ajustável da fila em 3
    QueuePosicaoHandle = osMessageQueueNew(3, sizeof(float), NULL);
//Tamanho ajustável da fila em 3

    // Inicialização dos Controladores PID
    PID_Init(&pid_traction, KP_TRACTION, KI_TRACTION, KD_TRACTION);
    PID_Init(&pid_velocity, KP_VELOCITY, KI_VELOCITY, KD_VELOCITY);
    PID_Init(&pid_position, KP_POSITION, KI_POSITION, KD_POSITION);

    // Criação das Tasks
    const osThreadAttr_t highPriorityTask_attr = {.priority =
(osPriority_t) osPriorityHigh};  // Definir a prioridade das tasks de
controle como alta
    osThreadNew(TaskControleCorrente, NULL, &highPriorityTask_attr); //
Task Controle de Corrente - leitura dos sensores de corrente no ADC
    osThreadNew(TaskControleVelocidade, NULL, &highPriorityTask_attr); //
Task Controle de Velocidade - leitura do Encoder
    osThreadNew(TaskControlePosicao, NULL, &highPriorityTask_attr); //
Task Controle de Posição - leitura do GPS
    osThreadNew(TaskControlePID, NULL, NULL);  // Task PID com prioridade
padrão
    osThreadNew(TaskEnvioDadosUART, NULL, NULL);  // Task de envio de
dados UART com prioridade padrão

    // Inicia o Scheduler do FreeRTOS
    osKernelStart();

    // Caso o Scheduler do FreeRTOS falhe
    while (1) {}
```

```
}

// Task para controle da Corrente (Leitura do Sensor)
void TaskControleCorrente(void *argument) {
    float correnteMotor[3] = {0.0f, 0.0f, 0.0f};
    const float referenciaADC = 3.3f;  // Tensão de referência do ADC
    const float resolucaoADC = 4096.0f; // Resolução do ADC de 12 bits
    const float vccDiv2 = referenciaADC / 2.0f; // Vcc/2 = 1.65V para
ACS712
    const float sensibilidade = 0.185f; // Sensibilidade de 185 mV/A para
ACS712-05B

    for (;;) {
        // Leitura dos sensores de corrente (3 dados com período de 1ms)
        HAL_ADC_Start(&hadc1);
        HAL_ADC_PollForConversion(&hadc1, HAL_MAX_DELAY);
        float valorADC = HAL_ADC_GetValue(&hadc1);
        float vout = (valorADC / resolucaoADC) * referenciaADC;
        correnteMotor[0] = (vout - vccDiv2) / sensibilidade;

        HAL_ADC_Start(&hadc2);
        HAL_ADC_PollForConversion(&hadc2, HAL_MAX_DELAY);
        valorADC = HAL_ADC_GetValue(&hadc2);
        vout = (valorADC / resolucaoADC) * referenciaADC;
        correnteMotor[1] = (vout - vccDiv2) / sensibilidade;

        HAL_ADC_Start(&hadc3);
        HAL_ADC_PollForConversion(&hadc3, HAL_MAX_DELAY);
        valorADC = HAL_ADC_GetValue(&hadc3);
        vout = (valorADC / resolucaoADC) * referenciaADC;
        correnteMotor[2] = (vout - vccDiv2) / sensibilidade;

        // Verificação de espaço na fila antes de enviar os dados e Envio
dos dados para a Queue
        if (osMessageQueueGetSpace(QueueCorrenteHandle) > 0) {
            osMessageQueuePut(QueueCorrenteHandle, &correnteMotor[0], 0,
0);
        }

        // Aguarda 1ms antes de executar novamente
        osDelay(1);
    }
}
```

```c
#define WINDOW_SIZE 5

// Task para controle da Velocidade (Leitura do Encoder)
void TaskControleVelocidade(void *argument) {
    static float leituraBuffer[3][WINDOW_SIZE] = {0};
    static uint8_t bufferIndex = 0;
    static uint32_t lastSampleTime = 0;
    float velocidadeMotor[3] = {0.0f};
    int32_t encoderReading[3] = {0};
    int32_t lastEncoderReading[3] = {0};

    for (;;) {
        uint32_t currentTime = osKernelGetTickCount();
        uint32_t elapsedTime = currentTime - lastSampleTime;

        if (elapsedTime >= SAMPLE_PERIOD_MS) {
            // Leitura dos encoders atuais
            encoderReading[0] = __HAL_TIM_GET_COUNTER(&htim1);
            encoderReading[1] = __HAL_TIM_GET_COUNTER(&htim2);
            encoderReading[2] = __HAL_TIM_GET_COUNTER(&htim3);

            // Calcular a velocidade bruta
            float velocidadeBruta[3];
            for (int i = 0; i < 3; i++) {
                velocidadeBruta[i] = (float)(encoderReading[i] -
lastEncoderReading[i]) / 36.0f;
                leituraBuffer[i][bufferIndex] = velocidadeBruta[i];
                lastEncoderReading[i] = encoderReading[i];
            }

            // Atualizar o índice do buffer
            bufferIndex = (bufferIndex + 1) % WINDOW_SIZE;
            lastSampleTime = currentTime;
        }

        // Calcular a média móvel usando as leituras mais recentes
disponíveis
        for (int i = 0; i < 3; i++) {
            float soma = 0.0f;
            for (uint8_t j = 0; j < WINDOW_SIZE; j++) {
                soma += leituraBuffer[i][j];
            }
            velocidadeMotor[i] = soma / WINDOW_SIZE;
        }

        // Verificação de espaço na fila antes de enviar os dados
        if (osMessageQueueGetSpace(QueueVelocidadeHandle) > 0) {
            osMessageQueuePut(QueueVelocidadeHandle, &velocidadeMotor[0],
0, 0);
```

```
        }

        // Aguarda 10ms antes de executar novamente
        osDelay(10);
    }
}

// Task para controle da Posição (Ideia de Leitura Advinda do GPS)
void TaskControlePosicao(void *argument) {
    float posicaoBase[3] = {0.0f, 0.0f, 0.0f};
    for (;;) {
        // Leitura dos ângulos
        Roll = Read_Angle_Roll();
        Pitch = Read_Angle_Pitch();
        Yaw = Read_Angle_Yaw();

        // Leitura dos ângulos de rotação da base (3 dados com período de
100ms)
        posicaoBase[0] = Roll;   // Função para ler o ângulo Roll
        posicaoBase[1] = Pitch;  // Função para ler o ângulo Pitch
        posicaoBase[2] = Yaw;    // Função para ler o ângulo Yaw

        // Verificação de espaço na fila antes de enviar os dados
        if (osMessageQueueGetSpace(QueuePosicaoHandle) > 0) {
            osMessageQueuePut(QueuePosicaoHandle, &posicaoBase[0], 0, 0);
        }

        // Aguarda 100ms antes de executar novamente
        osDelay(100);
    }
}

// Adicionar uma estrutura para armazenar os parâmetros PID dinâmicos
typedef struct {
    double kp;
    double ki;
    double kd;
} PID_Params;

PID_Params params_traction = {KP_TRACTION, KI_TRACTION, KD_TRACTION};
PID_Params params_velocity = {KP_VELOCITY, KI_VELOCITY, KD_VELOCITY};
PID_Params params_position = {KP_POSITION, KI_POSITION, KD_POSITION};
```

```
// Mutex para sincronização
osMutexId_t mutexPIDParamsHandle;
osMutexId_t mutexSetpointsHandle;

// Semáforos para sincronização
osSemaphoreId_t semPIDParamsUpdateHandle;
osSemaphoreId_t semSetpointsUpdateHandle;

// Task para processamento da UART e atualização de setpoints
void TaskEnvioDadosUART(void *argument) {
    char buffer[256];
    int len;
    float new_kp, new_ki, new_kd;
    float new_traction_setpoint[3];
    float new_velocity_setpoint[3];
    float new_position_setpoint[3];
    char command[64];

    for (;;) {
        // Envio dos dados via UART
        len = snprintf(buffer, sizeof(buffer),
            "Roll: %.2f, Pitch: %.2f, Yaw: %.2f, Speed: %.2f, Traction:
%.2f\n",
            Roll, Pitch, Yaw, speed, traction);
        HAL_UART_Transmit(&huart2, (uint8_t*)buffer, len, HAL_MAX_DELAY);

        // Receber comando via UART
        HAL_UART_Receive(&huart2, (uint8_t*)command, sizeof(command) - 1,
HAL_MAX_DELAY);

        // Processar o comando para atualizar os parâmetros PID
        if (sscanf(command, "KP_TRACTION:%f KI_TRACTION:%f KD_TRACTION:%f
KP_VELOCITY:%f KI_VELOCITY:%f KD_VELOCITY:%f KP_POSITION:%f KI_POSITION:%f
KD_POSITION:%f",
                   &new_kp, &new_ki, &new_kd,
                   &params_velocity.kp, &params_velocity.ki,
&params_velocity.kd,
                   &params_position.kp, &params_position.ki,
&params_position.kd) == 9) {
            osMutexWait(mutexPIDParamsHandle, osWaitForever);
            params_traction.kp = new_kp;
            params_traction.ki = new_ki;
            params_traction.kd = new_kd;
```

```
            osMutexRelease(mutexPIDParamsHandle);

            // Sinaliza que os parâmetros PID foram atualizados
            osSemaphoreRelease(semPIDParamsUpdateHandle);
        }

        // Atualização dos setpoints via UART
        if (sscanf(command, "TRACTION_SETPOINT:%f %f %f
VELOCITY_SETPOINT:%f %f %f POSITION_SETPOINT:%f %f %f",
                   &new_traction_setpoint[0], &new_traction_setpoint[1],
&new_traction_setpoint[2],
                   &new_velocity_setpoint[0], &new_velocity_setpoint[1],
&new_velocity_setpoint[2],
                   &new_position_setpoint[0], &new_position_setpoint[1],
&new_position_setpoint[2]) == 9) {
            osMutexWait(mutexSetpointsHandle, osWaitForever);
            memcpy(traction_setpoint, new_traction_setpoint,
sizeof(new_traction_setpoint));
            memcpy(velocity_setpoint, new_velocity_setpoint,
sizeof(new_velocity_setpoint));
            memcpy(position_setpoint, new_position_setpoint,
sizeof(new_position_setpoint));
            osMutexRelease(mutexSetpointsHandle);

            // Sinaliza que os parâmetros PID foram atualizados
            osSemaphoreRelease(semPIDParamsUpdateHandle);
        }

        // Aguarda 500ms antes de enviar novamente
        osDelay(500);
    }
}

// Task para controle PID dos motores
void TaskControlePID(void *argument) {
    float correnteMotor[3];
    float velocidadeMotor[3];
    float posicaoBase[3];

    float traction_setpoint[3];
    float velocity_setpoint[3];
    float position_setpoint[3];
```

```
    for (;;) {
        // Aguardar que os parâmetros PID e setpoints sejam atualizados
        osSemaphoreAcquire(semPIDParamsUpdateHandle, osWaitForever);
        osSemaphoreAcquire(semSetpointsUpdateHandle, osWaitForever);

        // Leitura dos dados das filas
        osMessageQueueGet(QueueCorrenteHandle, &correnteMotor[0], NULL,
osWaitForever);
        osMessageQueueGet(QueueVelocidadeHandle, &velocidadeMotor[0],
NULL, osWaitForever);
        osMessageQueueGet(QueuePosicaoHandle, &posicaoBase[0], NULL,
osWaitForever);

        // Copiar os setpoints atuais
        osMutexWait(mutexSetpointsHandle, osWaitForever);
        memcpy(traction_setpoint, global_traction_setpoint,
sizeof(traction_setpoint));
        memcpy(velocity_setpoint, global_velocity_setpoint,
sizeof(velocity_setpoint));
        memcpy(position_setpoint, global_position_setpoint,
sizeof(position_setpoint));
        osMutexRelease(mutexSetpointsHandle);

        // Controle PID para cada motor
        float dt = 0.01; // Intervalo de amostragem em segundos

        // Controle de tração
        float traction_command[3];
        for (int i = 0; i < 3; i++) {
            traction_command[i] = PID_Compute(&pid_traction,
traction_setpoint[i], correnteMotor[i], dt);
        }

        // Controle de velocidade
        float velocity_command[3];
        for (int i = 0; i < 3; i++) {
            velocity_command[i] = PID_Compute(&pid_velocity,
velocity_setpoint[i], velocidadeMotor[i], dt);
        }

        // Controle de posição
        float position_command[3];
        for (int i = 0; i < 3; i++) {
```

```
            position_command[i] = PID_Compute(&pid_position,
position_setpoint[i], posicaoBase[i], dt);
        }

        // Aplicação dos comandos aos motores
        Set_Motor_Commands(traction_command, velocity_command,
position_command);

        // Enviar dados via UART
        for (int i = 0; i < 3; i++) {
            Send_Motor_Data_UART(velocidadeMotor[i], traction_command[i],
posicaoBase[i]);
        }

        // Aguarda 10ms antes de executar novamente
        osDelay(10);
    }
}

// Função para aplicar comandos aos motores
void Set_Motor_Commands(float traction_command[3], float
velocity_command[3], float position_command[3]) {
    // Definir limites máximos e mínimos para os comandos dos motores
    const float MAX_PWM = 255.0f;  // Valor máximo do PWM (ajustar
conforme necessário)
    const float MIN_PWM = 0.0f;    // Valor mínimo do PWM (ajustar
conforme necessário)

    // Variáveis para os comandos finais dos motores
    float motor_command[3][2]; // [0] para frente, [1] para trás

    // Ajustar os comandos finais dos motores com base nas combinações dos
comandos de tração e velocidade
    for (int i = 0; i < 3; i++) {
        float total_command = traction_command[i] + velocity_command[i] +
position_command[i];

        // Saturar o comando para garantir que ele esteja dentro da faixa
[-1, 1]
        if (total_command > 1.0f) total_command = 1.0f;
        if (total_command < -1.0f) total_command = -1.0f;

        // Determinar PWM para frente e para trás
```

```
        if (total_command > 0) {
            motor_command[i][0] = total_command * MAX_PWM;
            motor_command[i][1] = 0;
        } else {
            motor_command[i][0] = 0;
            motor_command[i][1] = -total_command * MAX_PWM;
        }
    }

    // Aplicar os comandos PWM aos motores (ajustar conforme necessário)
    __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_1,
(uint32_t)motor_command[0][0]);
    __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_2,
(uint32_t)motor_command[1][0]);
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1,
(uint32_t)motor_command[2][0]);
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_2,
(uint32_t)motor_command[0][1]);
    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_1,
(uint32_t)motor_command[1][1]);
    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_2,
(uint32_t)motor_command[2][1]);
}

// Função para enviar os dados via comunicação serial (APLICAÇÃO SEM O
FREERTOS)
//void Send_Motor_Data_UART(float speed, float traction, float position) {
//    char uart_buffer[100];
//    int len = snprintf(uart_buffer, sizeof(uart_buffer), "Speed: %.2f,
Traction: %.2f, Position: %.2f\r\n", speed, traction, position);
//
//    if (len > 0) {
//        HAL_StatusTypeDef status = HAL_UART_Transmit(&huart2,
(uint8_t*)uart_buffer, len, HAL_MAX_DELAY);
//        if (status != HAL_OK) {
//            // Tratar o erro de transmissão, se necessário
//            Error_Handler();  // Ou algum outro tratamento de erro
apropriado
//        }
//    }
//}

// Funções de Configuração do Hardware
```

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

    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
    RCC_OscInitStruct.HSEState = RCC_HSE_ON;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
    RCC_OscInitStruct.PLL.PLLM = 8;
    RCC_OscInitStruct.PLL.PLLN = 336;
    RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
    RCC_OscInitStruct.PLL.PLLQ = 7;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK) {
        Error_Handler();
    }

    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK |
RCC_CLOCKTYPE_SYSCLK | RCC_CLOCKTYPE_PCLK1 | RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV4;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV2;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_5) !=
HAL_OK) {
        Error_Handler();
    }
}

// Inicialização dos GPIOs
static void MX_GPIO_Init(void) {
    GPIO_InitTypeDef GPIO_InitStruct = {0};

    __HAL_RCC_GPIOA_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();
    __HAL_RCC_GPIOC_CLK_ENABLE();

    // Configuração dos pinos dos motores e outros periféricos
    GPIO_InitStruct.Pin = GPIO_PIN_12 | GPIO_PIN_13 | GPIO_PIN_14 |
GPIO_PIN_15;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
```

```
    GPIO_InitStruct.Alternate = GPIO_AF1_TIM1;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = GPIO_PIN_8 | GPIO_PIN_9 | GPIO_PIN_10 |
GPIO_PIN_11;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    GPIO_InitStruct.Alternate = GPIO_AF1_TIM1;
    HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
}

// Inicialização do ADC1
static void MX_ADC1_Init(void) {
    ADC_ChannelConfTypeDef sConfig = {0};
    hadc1.Instance = ADC1;
    hadc1.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
    hadc1.Init.Resolution = ADC_RESOLUTION_12B;
    hadc1.Init.ScanConvMode = ENABLE;
    hadc1.Init.ContinuousConvMode = DISABLE;
    hadc1.Init.DiscontinuousConvMode = DISABLE;
    hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc1.Init.NbrOfConversion = 1;
    hadc1.Init.DMAContinuousRequests = DISABLE;
    hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    if (HAL_ADC_Init(&hadc1) != HAL_OK) {
        Error_Handler();
    }
    sConfig.Channel = ADC_CHANNEL_6;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
    if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK) {
        Error_Handler();
    }
}

// Inicialização do ADC2
static void MX_ADC2_Init(void) {
    ADC_ChannelConfTypeDef sConfig = {0};
    hadc2.Instance = ADC2;
    hadc2.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
    hadc2.Init.Resolution = ADC_RESOLUTION_12B;
```

```
    hadc2.Init.ScanConvMode = ENABLE;
    hadc2.Init.ContinuousConvMode = DISABLE;
    hadc2.Init.DiscontinuousConvMode = DISABLE;
    hadc2.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    hadc2.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc2.Init.NbrOfConversion = 1;
    hadc2.Init.DMAContinuousRequests = DISABLE;
    hadc2.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    if (HAL_ADC_Init(&hadc2) != HAL_OK) {
        Error_Handler();
    }
    sConfig.Channel = ADC_CHANNEL_7;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
    if (HAL_ADC_ConfigChannel(&hadc2, &sConfig) != HAL_OK) {
        Error_Handler();
    }
}

// Inicialização do ADC3
static void MX_ADC3_Init(void) {
    ADC_ChannelConfTypeDef sConfig = {0};
    hadc3.Instance = ADC3;
    hadc3.Init.ClockPrescaler = ADC_CLOCK_SYNC_PCLK_DIV4;
    hadc3.Init.Resolution = ADC_RESOLUTION_12B;
    hadc3.Init.ScanConvMode = ENABLE;
    hadc3.Init.ContinuousConvMode = DISABLE;
    hadc3.Init.DiscontinuousConvMode = DISABLE;
    hadc3.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
    hadc3.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc3.Init.NbrOfConversion = 1;
    hadc3.Init.DMAContinuousRequests = DISABLE;
    hadc3.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
    if (HAL_ADC_Init(&hadc3) != HAL_OK) {
        Error_Handler();
    }
    sConfig.Channel = ADC_CHANNEL_8;
    sConfig.Rank = 1;
    sConfig.SamplingTime = ADC_SAMPLETIME_3CYCLES;
    if (HAL_ADC_ConfigChannel(&hadc3, &sConfig) != HAL_OK) {
        Error_Handler();
    }
}
```

```
// Inicialização do Timer 1
static void MX_TIM1_Init(void) {
    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};
    TIM_OC_InitTypeDef sConfigOC = {0};

    htim1.Instance = TIM1;
    htim1.Init.Prescaler = 0;
    htim1.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim1.Init.Period = 0xFFFF;
    htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim1.Init.RepetitionCounter = 0;
    htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim1) != HAL_OK) {
        Error_Handler();
    }

    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK)
{
        Error_Handler();
    }

    if (HAL_TIM_PWM_Init(&htim1) != HAL_OK) {
        Error_Handler();
    }

    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) !=
HAL_OK) {
        Error_Handler();
    }

    sConfigOC.OCMode = TIM_OCMODE_PWM1;
    sConfigOC.Pulse = 0;
    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
    sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
```

```
    if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_1) !=
HAL_OK) {
        Error_Handler();
    }

    HAL_TIM_MspPostInit(&htim1);
}

// Inicialização do Timer 2
static void MX_TIM2_Init(void) {
    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};
    TIM_OC_InitTypeDef sConfigOC = {0};

    htim2.Instance = TIM2;
    htim2.Init.Prescaler = 0;
    htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim2.Init.Period = 0xFFFF;
    htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim2) != HAL_OK) {
        Error_Handler();
    }

    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
{
        Error_Handler();
    }

    if (HAL_TIM_PWM_Init(&htim2) != HAL_OK) {
        Error_Handler();
    }

    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) !=
HAL_OK) {
        Error_Handler();
    }

    sConfigOC.OCMode = TIM_OCMODE_PWM1;
    sConfigOC.Pulse = 0;
```

```
    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
    sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) !=
HAL_OK) {
        Error_Handler();
    }

    HAL_TIM_MspPostInit(&htim2);
}

// Inicialização do Timer 3
static void MX_TIM3_Init(void) {
    TIM_ClockConfigTypeDef sClockSourceConfig = {0};
    TIM_MasterConfigTypeDef sMasterConfig = {0};
    TIM_OC_InitTypeDef sConfigOC = {0};

    htim3.Instance = TIM3;
    htim3.Init.Prescaler = 0;
    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim3.Init.Period = 0xFFFF;
    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
    if (HAL_TIM_Base_Init(&htim3) != HAL_OK) {
        Error_Handler();
    }

    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
    if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
{
        Error_Handler();
    }

    if (HAL_TIM_PWM_Init(&htim3) != HAL_OK) {
        Error_Handler();
    }

    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) !=
HAL_OK) {
```

```
        Error_Handler();
    }

    sConfigOC.OCMode = TIM_OCMODE_PWM1;
    sConfigOC.Pulse = 0;
    sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCNPolarity = TIM_OCNPOLARITY_HIGH;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
    sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
    if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_1) !=
HAL_OK) {
        Error_Handler();
    }

    HAL_TIM_MspPostInit(&htim3);
}

// Inicialização da comunicação Serial
void MX_USART2_UART_Init(void)
{
    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_TX_RX;
    huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart2.Init.OverSampling = UART_OVERSAMPLING_16;
    if (HAL_UART_Init(&huart2) != HAL_OK){
        Error_Handler();
    }
}

// Inicialização do IWDG (Independent Watchdog Timer)
static void MX_IWDG_Init(void) {
    hiwdg.Instance = IWDG;
    hiwdg.Init.Prescaler = IWDG_PRESCALER_64;
    hiwdg.Init.Reload = 4095;
    if (HAL_IWDG_Init(&hiwdg) != HAL_OK) {
        Error_Handler();
    }
}
```

```
// Função de tratamento de erro
void Error_Handler(void) {
    __disable_irq();
    while (1) {}
}

// -> Funções de Leitura dos Ângulos (Não sei exatamente como implementar)
float Read_Angle_Roll(void) {
    // Função fictícia para leitura do ângulo Roll
    return 0.0f;
}
float Read_Angle_Pitch(void) {
    // Função fictícia para leitura do ângulo Pitch
    return 0.0f;
}
float Read_Angle_Yaw(void) {
    // Função fictícia para leitura do ângulo Yaw
    return 0.0f;
}

// Implementação da função PID_Compute
void PID_Compute(PID_TypeDef *pid) {
    double error = pid->Setpoint - pid->Input;
    pid->ITerm += (pid->Ki * error);
    if(pid->ITerm > pid->outMax) pid->ITerm = pid->outMax;
    else if(pid->ITerm < pid->outMin) pid->ITerm = pid->outMin;
    double dInput = pid->Input - pid->lastInput;

    pid->Output = pid->Kp * error + pid->ITerm - pid->Kd * dInput;
    if(pid->Output > pid->outMax) pid->Output = pid->outMax;
    else if(pid->Output < pid->outMin) pid->Output = pid->outMin;

    pid->lastInput = pid->Input;
}

// Implementação da função PID_Init
void PID_Init(PID_TypeDef *pid, double kp, double ki, double kd) {
    pid->Kp = kp;
    pid->Ki = ki;
    pid->Kd = kd;
    pid->Integral = 0;
    pid->LastError = 0;
```

```
}

// Estrutura para o controle PID
typedef struct {
    double Setpoint;
    double Input;
    double Output;
    double Kp;
    double Ki;
    double Kd;
    double Integral;
    double LastError;
} PID_TypeDef;
```