**FACULDADE DE TECNOLOGIA DE TAUBATÉ**

**DISCIPLINA - PROGRAMAÇÃO DE MICROCONTROLADORES**

**PROF. MOZART FRAGA**

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamenteGráfico, Linha do tempo

Descrição gerada automaticamente

Sumário

[1 – Introdução ao Git Hub 3](#_Toc188387676)

[1.2 – Principais conceitos Git 3](#_Toc188387677)

[1.3 – Principais comando Git 3](#_Toc188387678)

[1.4 – Git GUI 5](#_Toc188387679)

[2 – Revisão Lógica de Programação 6](#_Toc188387680)

[2.1 – Variáveis e tipos de dados 6](#_Toc188387681)

[2.1.1 - Tipos de Dados Básicos 6](#_Toc188387682)

[2.1.2 – Operadores e expressões 8](#_Toc188387683)

[2.1.3 – Estruturas Condicionais 8](#_Toc188387684)

[2.1.4 – Estruturas de repetição 9](#_Toc188387685)

[3 - Entendendo os Microcontroladores 10](#_Toc188387686)

[4 – STM32CUBE IDE 14](#_Toc188387687)

[5 - Estrutura Básica do STM32 18](#_Toc188387688)

[6 – Entrada e Saída Digital (GPIO) 19](#_Toc188387689)

[7 – Comunicação Serial 21](#_Toc188387690)

[7.1 – UART: Configuração e envio/recepção de dados 21](#_Toc188387691)

[7.2 – I2C: Comunicação com sensores e dispositivos 22](#_Toc188387692)

[7.3 – SPI: Comunicação de alta velocidade 22](#_Toc188387693)

[8 – Temporizadores e PWM 23](#_Toc188387694)

[8.1 – Configuração de temporizadores básicos 23](#_Toc188387695)

[8.2 – Geração de sinais PWM para controle de cargas 24](#_Toc188387696)

[9 – Conversores Analógico-Digital (ADC) 25](#_Toc188387697)

[9.1 – Leitura de sensores analógicos 25](#_Toc188387698)

[9.2 – Configuração de múltiplos canais e DMA 26](#_Toc188387699)

[10 – Interrupções 28](#_Toc188387700)

[11 – Debugging e Otimização 30](#_Toc188387701)

[11.1 – Uso do depurador integrado 30](#_Toc188387702)

[11.2 – Monitoramento de variáveis em tempo real 30](#_Toc188387703)

[11.3 – Técnicas de otimização de código 31](#_Toc188387704)

[12 – Uso de RTOS no STM32F103C8T6 32](#_Toc188387705)

# 1 – Introdução ao Git Hub

O GitHub é uma plataforma de hospedagem de código-fonte e controle de versão que utiliza o Git, um sistema amplamente utilizado para gerenciar projetos de software. Ele permite que desenvolvedores colaborem em projetos, rastreiem alterações no código e integrem contribuições de maneira eficiente. Além disso, o GitHub oferece ferramentas para gerenciamento de projetos, como **issues** e **pull requests**, facilitando a organização e comunicação entre equipes. A plataforma também é usada para compartilhar projetos de código aberto, possibilitando que a comunidade contribua e aprenda com o trabalho de outros desenvolvedores.

## 1.2 – Principais conceitos Git

Para utilizar o Git de forma eficaz, é essencial compreender alguns conceitos principais que estruturam seu funcionamento. Aqui estão os mais importantes:

**Repositório (Repository):** É o local onde o código e seu histórico de versões são armazenados. Pode ser local (em seu computador) ou remoto (em plataformas como GitHub).

**Commit:** Representa uma "foto" do estado do código em um momento específico. Cada commit é identificado por um hash único e deve incluir uma mensagem que descreva as alterações realizadas.

**Branch:** São ramificações do projeto que permitem desenvolver novas funcionalidades ou corrigir bugs sem afetar a linha principal de desenvolvimento (geralmente chamada de main ou master).

**Merge:** O processo de integrar mudanças de uma branch para outra, combinando seus históricos e atualizando o código.

**Clone:** Copia um repositório remoto para o ambiente local, permitindo que você trabalhe em um projeto já existente.

**Pull:** Atualiza seu repositório local com as mudanças feitas no repositório remoto.

**Push:** Envia suas alterações (commits) do repositório local para o repositório remoto.

**Fetch:** Obtém informações sobre mudanças no repositório remoto sem integrá-las automaticamente ao seu repositório local.

**Conflict:** Ocorre quando alterações em uma mesma linha de código são feitas em diferentes branches e precisam ser resolvidas manualmente durante um merge.

## 1.3 – Principais comando Git

**1.3.1 - Configuração Inicial**

**git config**: Configura opções globais ou específicas para um repositório.

git config --global user.name "Seu Nome": Define o nome do usuário.

git config --global user.email "seuemail@exemplo.com": Define o e-mail do usuário.

**1.3.2 - Criação e Inicialização de Repositórios**

**git init:** Inicializa um novo repositório Git em um diretório.

**git clone:** Clona um repositório remoto para o ambiente local.

**1.3.3 - Rastreamento e Controle de Arquivos**

**git status:** Mostra o estado atual do repositório, incluindo arquivos rastreados, não rastreados e prontos para commit.

**git add:** Adiciona arquivos ao stage (área de preparação).

git add arquivo.txt: Adiciona um arquivo específico.

git add . : Adiciona todas as alterações no diretório atual.

**git rm:** Remove arquivos do repositório e do sistema de arquivos.

**1.3.4 - Confirmar Alterações**

**git commit:** Cria um snapshot das alterações no repositório.

git commit -m "Mensagem do commit": Confirma com uma mensagem descritiva.

git commit -a -m "Mensagem do commit": Adiciona e confirma todas as alterações rastreadas.

**1.3.5 - Branches (Ramificações)**

**git branch:** Gerencia branches.

**git branch:** Lista as branches existentes.

**git checkout -b new-branch-name:** Cria uma nova branch.

**git checkout <branch\_name>:** Troca de branch ou restaura arquivos.

**git checkout -- <file>:** Restaura um arquivo ao estado do último commit.

**1.3.6 - Integração de Alterações**

**git merge branch\_name:** Combina mudanças de uma branch para outra.

**git rebase ‘branch’:** Reaplica commits de uma branch em outra base, criando um histórico linear.

**1.3.7 - Sincronização com Repositórios Remotos**

**git fetch branch:** Baixa informações de um repositório remoto sem aplicar mudanças.

**git pull origin branch\_name**: Atualiza o repositório local com alterações do remoto (fetch + merge).

**git push:** Envia commits locais para o repositório remoto.

**1.3.8 - Solução de Conflitos e Modificações Temporárias**

**git stash:** Armazena temporariamente alterações não confirmadas.

**git stash apply:** Restaura as alterações salvas.

**git log:** Mostra o histórico de commits.

**git diff:** Compara mudanças entre commits, branches ou arquivos.

**1.3.9 - Reverter Alterações**

**git reset**: Reverte alterações no histórico ou no stage.

**git revert:** Cria um novo commit que desfaz as alterações de um commit específico.

## 1.4 – Git GUI

Um recurso muito útil para facilitar o versionamento de software é utilizar as **Interfaces Gráficas de Usuário (GUI)**. Essas ferramentas fornecem uma visualização humanizada para o Git, facilitando o gerenciamento de sua base de código.

**Melhores Clientes Git GUI para Windows**

**1. Sourcetree**

Sourcetree é um cliente Git GUI gratuito que funciona tanto no Windows quanto no Mac. Essa ferramenta é simples de usar, mas também é muito poderosa. Sua interface permite que você navegue sem esforços e de uma maneira agradável. Ela suporta arquivos grandes de Git e deixa que você os visualize usando diagramas de ramificação detalhados. A pesquisa de commits locais permite que você encontre mudanças de arquivos, commits e ramificações. Enquanto isso, o gerente de repositórios remotos permite que você pesquise e clone repositórios remotos dentro do próprio Sourcetree.

**2. GitHub Desktop**

O GitHub desktop é um cliente Git GUI gratuito e de código aberto. Ele tem uma interface intuitiva que permite gerenciar os códigos sem a necessidade de digitar comandos. Você pode criar ou adicionar repositórios locais e executar operações Git com facilidade.

**3. TortoiseGit**

Esse software gratuito e de código aberto é uma interface shell do Windows para Git. Ele pode ser usado num ambiente comercial, assim como ser desenvolvido para a sua própria versão customizada. O Tortoise Git pode ser usado com outras ferramentas de desenvolvimento e com qualquer tipo de arquivo.

# 2 – Revisão Lógica de Programação

## 2.1 – Variáveis e tipos de dados

As variáveis são usadas para armazenar dados que podem ser alterados durante a execução do programa. Cada variável está associada a um tipo de dado especifico escolhido pelo programador de acordo com a necessidade do projeto.

Os tipos de dados comuns em C incluem:

## 2.1.1 - Tipos de Dados Básicos

Esses são os tipos de dados padrão do C/C++ suportados pela arquitetura ARM Cortex:

**Tipos Inteiros:**

**int8\_t**: Inteiro com sinal de 8 bits.

**uint8\_t**: Inteiro sem sinal de 8 bits.

**int16\_t**: Inteiro com sinal de 16 bits.

**uint16\_t**: Inteiro sem sinal de 16 bits.

**int32\_t:** Inteiro com sinal de 32 bits.

**uint32\_t**: Inteiro sem sinal de 32 bits.

**int64\_t**: Inteiro com sinal de 64 bits (em ARMv7 e versões posteriores).

**uint64\_t**: Inteiro sem sinal de 64 bits (em ARMv7 e versões posteriores).

**Tipos de Ponto Flutuante:**

**float:** Ponto flutuante de 32 bits (Cortex-M4, M7 ou série A).

**double**: Ponto flutuante de 64 bits (em arquiteturas que suportam, como ARMv7-A e ARMv8-A).

**Tipos de Caracteres:**

**char:** Tipo de caractere de 8 bits, pode ser com ou sem sinal dependendo do compilador.

**Tipo Booleano**

**bool:** Um valor booleano de um único bit.

**Tipos de Ponteiros**

Todos os processadores Cortex suportam ponteiros, que podem referenciar locais de memória. O tamanho depende da arquitetura:

Ponteiros de 32 bits para ARMv7 (Cortex-M e Cortex-A).

Ponteiros de 64 bits para ARMv8-A.

**Tipos de Dados Compostos**

**Arrays:** Coleções de dados do mesmo tipo.

Exemplo:

int numeros[5] = {1, 2, 3, 4, 5};

char mensagem[10] = "Olá Mundo";

**Structs:** Tipos de dados personalizados que combinam vários campos de diferentes tipos.

Exemplo:

struct Pessoa

{

char nome [50];

int idade;

float altura;

};

struct Pessoa joao = {"João", 25, 1.75};

**Union:** Tipos de dados eficientes em memória, compartilhando o mesmo espaço entre vários campos.

Exemplo:

union Data {

int inteiro; // Ocupa 4 bytes

float pontoFlutuante; // Ocupa 4 bytes

char texto[4]; // Ocupa 4 bytes

};

union Data dado;

dado.inteiro = 65;

dado.texto[0] = 'O';

dado.texto[1] = 'K';

dado.texto[2] = '\0';

dado.pontoFlutuante = 3.14;

**Tipos Enumerados**

**enum:** Enumerações definidas pelo usuário para maior legibilidade e manutenção do código.

Exemplo:

enum DiaDaSemana {Domingo, Segunda, Terca, Quarta, Quinta, Sexta, Sabado};

enum DiaDaSemana hoje = Terca;

## 2.1.2 – Operadores e expressões

Os operadores são usados para realizar operações em variáveis e valores. Alguns operadores comuns incluem:

* **Aritméticos:** +, -, \*, /, %
* **Relacionais:** ==, !=, >, <, >=, <=
* **Lógicos:** &&, ||, !
* **Bit a bit:** &, |, ^, ~, <<, >>

Exemplo:

int a = 5, b = 10;

int soma = a + b; // Soma

int produto = a \* b; // Produto

int maior = (a > b); // Verifica se a é maior que b

int bitwise\_and = a & b; // AND bit a bit

## 2.1.3 – Estruturas Condicionais

As estruturas condicionais são usadas para executar diferentes blocos de código com base em uma condição avaliada como verdadeira ou falsa.

**2.1.3.1 – if / else if / else**

Permite tomar decisões baseadas em condições lógicas.

if (condicao) {

// Executa se a condição for verdadeira

} else if (outraCondicao) {

// Executa se a outra condição for verdadeira

} else {

// Executa se nenhuma das condições anteriores for verdadeira

}

**2.1.3.2 – switch**

Ideal para comparar uma variável com múltiplos valores.

switch (variavel) {

case valor1:

// Executa o bloco de código para valor1

break;

case valor2:

// Executa o bloco de código para valor2

break;

default:

// Executa se nenhum caso for correspondido

break;

}

## 2.1.4 – Estruturas de repetição

Os loops permitem executar repetidamente um bloco de código enquanto uma condição for verdadeira.

**2.1.4.1 – Laço For**

Usado quando o número de iterações é conhecido.

for (inicializacao; condicao; incremento) {

// Bloco de código a ser repetido

}

**2.1.4.2 – while**

Executa o bloco de código enquanto a condição for verdadeira.

while (condicao) {

// Bloco de código a ser repetido

}

**2.1.4.3 – do/while**

Semelhante ao while, mas garante que o bloco de código seja executado pelo menos uma vez.

do {

// Bloco de código a ser repetido

} while (condicao);

**COMANDO BREAK**

O comando break encerra o processamento de estruturas de repetição ou a execução de um

comando switch.

**COMANDO RETURN**

O comando return é o mecanismo de saída de uma função, retornando um valor para a função que a chamou. Sua sintaxe é: return (expressão\_de\_retorno);

void função\_qualquer(int valor)

{

int Resultado = valor\*2;

Return(Resultado);

}

# 3 - Entendendo os Microcontroladores

Diagrama, Esquemático

Descrição gerada automaticamente

Figura 1 – Arquitetura Microcontrolador ARM

Um microcontrolador é um circuito integrado que combina um processador, memória e periféricos em um único chip, projetado para executar uma ou mais tarefas específicas em sistemas embarcados. Esses dispositivos são amplamente utilizados em aplicações como automação residencial, eletroeletrônicos, automotivo, médico, entre outros.

Os periféricos de um microcontrolador são componentes internos ou externos que permitem a interação com o ambiente ou com outros sistemas. Eles desempenham funções essenciais para o funcionamento do microcontrolador e a implementação de suas funcionalidades. Os principais periféricos de um microcontrolador podem ser classificados em diversas categorias:

**Periféricos de Entrada e Saída (I/O)**

* GPIO (General-Purpose Input/Output): Pinos configuráveis como entrada ou saída para interagir com dispositivos externos.
* Teclados e Botões: Permitem ao usuário enviar comandos ao sistema.
* LEDs e Displays: Usados para exibir informações ou indicar estados do sistema.

**Periféricos de Comunicação**

* UART (Universal Asynchronous Receiver-Transmitter): Protocolo para comunicação serial.
* I2C (Inter-Integrated Circuit): Interface para comunicação entre dispositivos em um barramento de dois fios.
* SPI (Serial Peripheral Interface): Protocolo para comunicação de alta velocidade entre dispositivos.
* CAN (Controller Area Network): Usado principalmente em sistemas automotivos.
* USB (Universal Serial Bus): Interface padrão para conectar dispositivos periféricos.

**Periféricos de Temporização**

* Timers: Utilizados para medição de tempo, geração de delays ou sinais PWM (Pulse Width Modulation).
* Watchdog Timer: Monitora o funcionamento do sistema e reinicia o microcontrolador em caso de falha.

**Periféricos Analógicos**

* Conversores ADC (Analog-to-Digital Converter): Convertem sinais analógicos em digitais.
* Conversores DAC (Digital-to-Analog Converter): Convertem sinais digitais em analógicos.
* Comparadores Analógicos\*\*: Comparam tensões e geram saídas lógicas com base no resultado.

**Periféricos de Memória**

* Memória Flash\*\*: Armazena o programa e dados não voláteis.
* RAM (Random Access Memory): Utilizada para armazenamento temporário de dados durante a execução do programa.
* EEPROM (Electrically Erasable Programmable Read-Only Memory): Armazena dados que podem ser alterados durante a execução do sistema.

**Periféricos Especiais**

* Controladores DMA (Direct Memory Access): Permitem transferências diretas de dados entre memória e periféricos sem o uso do processador.
* Módulos de Captura e Comparação: Utilizados em aplicações que exigem medição precisa de eventos externos.
* Sensores Integrados: Alguns microcontroladores possuem sensores de temperatura ou acelerômetros integrados.

**Arquitetura ARM**

A arquitetura ARM (“Advanced RISC Machine”) é um conjunto de instruções e uma família de processadores baseada no conceito RISC (Reduced Instruction Set Computer). Desenvolvida inicialmente pela ARM Holdings, essa arquitetura é conhecida por seu baixo consumo de energia e alta eficiência, tornando-se ideal para dispositivos embarcados, como smartphones, tablets, dispositivos IoT e microcontroladores.

Os processadores ARM utilizam uma abordagem modular e escalável, permitindo que diferentes fabricantes personalizem seus designs para atender a necessidades específicas. Características como pipelines eficientes, suporte a múltiplos núcleos e compatibilidade com instruções de 32 e 64 bits contribuem para seu amplo uso em aplicações de alto desempenho e baixo custo. Além disso, sua ampla comunidade de desenvolvedores e ecossistema de ferramentas tornam a arquitetura ARM uma das mais populares e versáteis no mercado de sistemas embarcados.

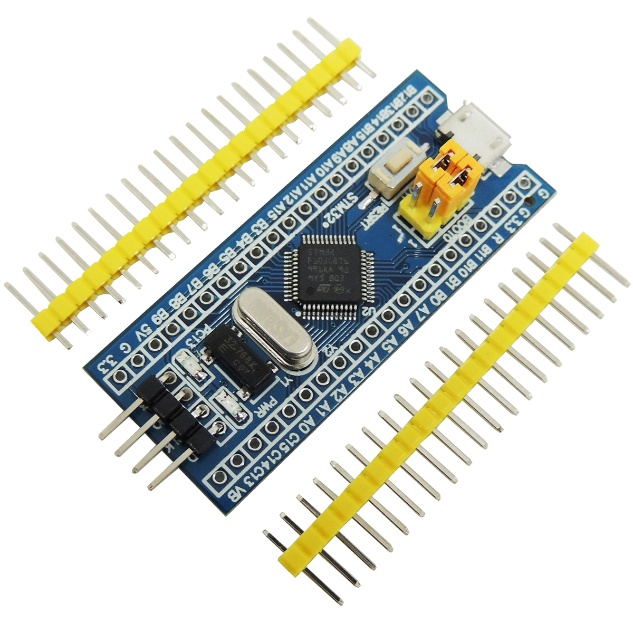


Figura 2 – Placa de desenvolvimento com o microcontrolador STM32F013C8T6 a ser usado no curso

Texto

Descrição gerada automaticamente com confiança média

**Pinagem da Placa de Desenvolvimento Blue Pill**

**Gráfico, Linha do tempo

Descrição gerada automaticamente**

Entendendo a IDE

IDE (“Integrated Development Environment”) é um ambiente de desenvolvimento integrado que combina um conjunto de ferramentas essenciais para a criação de software em uma única interface. As IDEs têm como objetivo simplificar e agilizar o processo de desenvolvimento, oferecendo recursos que facilitam a escrita, edição, depuração e compilação de códigos.

Os principais componentes de uma IDE incluem:

1. **Editor de Código**: Um editor de texto otimizado para a escrita de código-fonte, geralmente com destaque de sintaxe, autocompletação e sugestões.
2. **Compilador ou Interpretador**: Ferramentas que traduzem o código escrito em linguagem de programação para um formato executável pelo computador.
3. **Depurador (Debugger)**: Um utilitário para identificar e corrigir erros no código, permitindo a execução passo a passo e a inspeção de variáveis.
4. **Gerenciamento de Projetos**: Recursos para organizar arquivos, bibliotecas e dependências dentro de um projeto.
5. **Ferramentas de Integração**: Plugins ou extensões para integração com sistemas de controle de versão (como Git) ou plataformas de implantação.

Exemplos populares de IDEs incluem Visual Studio, Eclipse, PyCharm, Arduino IDE, STMCUBE IDE e muitos outros.

# 4 – STM32CUBE IDE

O STM32CubeIDE é um sistema de desenvolvimento completo para desenvolver código para quase todos os microcontroladores baseados em STM32 da ST Microelectronics. Como o nome sugere, é um Ambiente de Desenvolvimento Integrado (IDE) que inclui essencialmente a ferramenta de configuração de HW GUI STMCubeMx e um compilador completo.

Após instalar CubeIDE, para iniciar um novo projeto crie um repositório em um local de sua preferência e direcione para selecionar seu workspace padrão, clique em launch para poder finalizar a escolha do repositório.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Figura 5 - Seleção de diretório para workspace

O próximo passo será criar um projeto.

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

Figura 6 - Criando um novo projeto

Neste momento irá abrir uma tela com todas as famílias de microcontroladores da família STM32 atualmente suportadas pela CubeIDE. Para poder achar nosso microcontrolador, iremos no campo de busca “part number search” e digitaremos “STM32F103C8T6”.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Figura 7 – Seleção do MCU STM32F103C8T6

Neste momento iremos ver a tela onde será possível dar o nome do projeto, definimos com o nome “Projeto XXXXXX”, não sendo necessário modificar mais nada, apenas clicar em next e finish.

Segue-se para a tela de carregamento do CubeMX, uma interface gráfica para a configuração do microcontrolador.

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

Figura 8 - Tela inicial para configuração gráfica do CubeMX

Nota-se que nenhuma configuração foi feita até o presente momento, e sabendo que iremos utilizar a placa Blue pill, precisamos configurar o uso do cristal externo de 8 MHz que possui conectada fisicamente, então nosso primeiro passo é configurar o uso do cristal externo da nossa placa de desenvolvimento. Existe uma opção no lado esquerdo, chamada “System Core”, ao clicar nela, será aberto uma lista de periféricos para configuração, iremos configurar o periférico “RCC”, para isso é preciso clicar 2x no mesmo.

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamenteFigura 9 - Tela de configuração do periférico RCC

O próximo passo é ir na opção “High Speed Clock (HSE)” e marcar a opção, “Crystal/Ceramic Resonator”.

[Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente](https://embarcados.com.br/wp-content/uploads/2019/08/image14-1.png)

Figura 10 - Habilitação do HSE no RCC.

Após isso, vamos na opção “Clock configuration” para definir a frequência do cristal para 8 MHz e habilitar a multiplicação interna do RCC para HSE. Após modificar estes dois parâmetros, vamos até a opção “HCLK(MHz)” e vamos digitar 72 MHz e teclar enter, depois “OK”, neste momento, o próprio CubeMX irá fazer todos os cálculos de multiplicação de PLL e configuração para que os periféricos e clock principal, possam ser rodados na frequência de 72 MHz (exceto o APB1 peripheral clock que é limitado a 36 MHz).

[Diagrama

Descrição gerada automaticamente](https://embarcados.com.br/wp-content/uploads/2019/08/image10-1.png)Figura 11 - Configuração de PLL, HSE e Frequência de opção em 72 Mhz.

Nosso próximo passo é voltar para opção “Pinout & Configuration” no menu principal da barra superior, clicar novamente em “System Core” e realizar a configuração dos pinos e periféricos que serão utilizados.

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamenteFigura 12 - Configuração da Serial Wire para debug.

Para gerar a configuração final do projeto, na opção “Save ALL” que é basicamente o ícone de um disquete aninhado, ao clicar nela, será perguntado se queremos gerar o código inicial a partir do CubeMX, iremos escolher a opção “Yes”.

[Interface gráfica do usuário, Texto, Aplicativo, chat ou mensagem de texto

Descrição gerada automaticamente](https://embarcados.com.br/wp-content/uploads/2019/08/image5-2.png)Figura 13 - Passo para salvar e gerar estrutura de códigos.

# 5 - Estrutura Básica do STM32

ARM Cortex-M Core:

Os microcontroladores STM32 são baseados na arquitetura ARM Cortex-M, projetada para alto desempenho com baixo consumo de energia. Dependendo da família, eles podem utilizar núcleos como Cortex-M0, M3, M4, ou M7, cada um com diferentes capacidades e características.

**Memória:**

* Flash: Usada para armazenar o código do programa.
* RAM: Armazena dados temporários e variáveis em tempo de execução.
* EEPROM (em alguns modelos): Permite armazenar dados não voláteis.

**Barramentos Internos:**

Os barramentos (como AHB e APB) conectam o núcleo aos periféricos e à memória, garantindo uma comunicação eficiente.

**GPIO e Multiplexação de Funções**

**General-Purpose Input/Output (GPIO):**

Pinos configuráveis como entrada ou saída digital, essenciais para interagir com o mundo externo.

**Multiplexação de Funções:**

Muitos pinos possuem múltiplas funções (ex.: GPIO, UART, SPI). A seleção da função é feita por meio de registros de configuração.

**Periféricos Internos**

Os STM32 possuem diversos periféricos integrados, que podem variar conforme o modelo:

**Relógio (RCC - Reset and Clock Control):**

Responsável por configurar as fontes de clock do sistema (HSE, LSE, HSI, etc.) e a frequência de operação dos periféricos e do núcleo.

**Temporizadores (Timers):**

Usados para medir intervalos de tempo, gerar sinais PWM ou controlar eventos periódicos.

**ADC e DAC:**

* ADC (Analog-to-Digital Converter): Converte sinais analógicos (ex.: sensores) em valores digitais.
* DAC (Digital-to-Analog Converter): Converte sinais digitais em analógicos.

**Comunicação:**

Interfaces como UART, SPI, I2C, CAN e USB, que permitem a troca de dados com outros dispositivos.

**Controladores DMA (Direct Memory Access):**

Permitem transferir dados diretamente entre periféricos e memória, liberando o núcleo para outras tarefas.

# 6 – Entrada e Saída Digital (GPIO)

Este tópico é crucial para ensinar como interagir com o mundo externo usando os pinos de propósito geral (GPIO) do microcontrolador STM32. Os GPIOs permitem que o microcontrolador se comunique com sensores, atuadores, botões, LEDs, e outros dispositivos digitais.

**6.1 – Configuração Básica dos GPIOs**

Para usar um GPIO, é necessário configurá-lo corretamente. Isso é feito por meio de registros específicos ou por ferramentas como o STM32CubeMX. A configuração básica envolve os seguintes passos:

1. **Selecionar o pino**: Identificar o pino físico no microcontrolador que será usado.
2. **Configurar o modo do pino**: Escolher entre entrada, saída, ou modos alternativos.
3. **Configurar a velocidade**: Determinar a frequência de operação do pino (baixa, média, alta, ou muito alta).
4. **Habilitar o pull-up/pull-down**: Definir resistores internos para estabilizar o sinal em entradas digitais.

**6.2 – Modos de Operação dos GPIOs**

Os pinos GPIO do STM32 podem ser configurados nos seguintes modos:

* **Entrada Digital**: Para ler sinais digitais de botões, sensores, etc.
* **Saída Digital**: Para acionar LEDs, relés, ou outros dispositivos.
* **Modo Alternativo**: Usado quando o pino está vinculado a um periférico (ex.: UART, SPI, PWM).
* **Entrada Analógica**: Para ler sinais analógicos em combinação com o ADC.
* **Entrada/Saída com Interrupção**: Permite que o pino gere uma interrupção quando seu estado mudar.

**6.3 – Exemplos de Uso**

**Acender um LED (Saída Digital)**

1. Configurar o pino GPIO como saída digital.
2. Enviar um nível lógico alto (HIGH) para ligar o LED e baixo (LOW) para desligá-lo.

HAL\_GPIO\_WritePin(GPIOA, GPIO\_PIN\_5, GPIO\_PIN\_SET); // Liga o LED

HAL\_GPIO\_WritePin(GPIOA, GPIO\_PIN\_5, GPIO\_PIN\_RESET); // Desliga o LED

**Ler o Estado de um Botão (Entrada Digital)**

1. Configurar o pino GPIO como entrada com pull-up interno.
2. Ler o estado do pino para determinar se o botão está pressionado.

if (HAL\_GPIO\_ReadPin(GPIOC, GPIO\_PIN\_13) == GPIO\_PIN\_RESET)

{

// Botão pressionado

}

**Interrupções com GPIO**

* **O Que São:** Interrupções permitem que o microcontrolador reaja a mudanças de estado em um pino sem precisar verificar constantemente (polling).
* **Exemplo**: Um botão pressionado pode gerar uma interrupção para executar uma ação específica.

void HAL\_GPIO\_EXTI\_Callback(uint16\_t GPIO\_Pin) {

if (GPIO\_Pin == GPIO\_PIN\_13) {

// Código a ser executado quando o botão for pressionado

}

}

# 7 – Comunicação Serial

Os microcontroladores STM32 possuem diversas interfaces para comunicação serial, permitindo a troca de dados com outros dispositivos, como sensores, displays e módulos externos. Este capítulo aborda a configuração e o uso das principais interfaces disponíveis no STM32F103C8T6.

## 7.1 – UART: Configuração e envio/recepção de dados

**Introdução**A UART (Universal Asynchronous Receiver/Transmitter) é amplamente utilizada para comunicação serial simples e eficiente. No STM32F103C8T6, ela é configurada através do periférico USART.

**Passos para configuração:**

1. Configuração do clock: Ative o clock para o periférico USART no RCC (Clock Control).
2. Configuração dos pinos: Configure os pinos correspondentes no GPIO como:

* TX: saída alternada push-pull.
* RX: entrada flutuante ou com pull-up.

1. Configuração da UART:

* Defina a taxa de transmissão (baud rate).
* Configure o número de bits de dados, paridade e stop bits.
* Habilite a transmissão e recepção.

1. Envio de dados: Utilize as funções HAL ou escreva diretamente nos registros para enviar bytes pelo registro USART\_DR.
2. Recepção de dados: Monitore a flag de recebimento e leia os dados no registro USART\_DR.

Exemplo com HAL:

UART\_HandleTypeDef huart1;

void UART\_Init() {

\_\_HAL\_RCC\_USART1\_CLK\_ENABLE();

huart1.Instance = USART1;

huart1.Init.BaudRate = 9600;

huart1.Init.WordLength = UART\_WORDLENGTH\_8B;

huart1.Init.StopBits = UART\_STOPBITS\_1;

huart1.Init.Parity = UART\_PARITY\_NONE;

huart1.Init.Mode = UART\_MODE\_TX\_RX;

HAL\_UART\_Init(&huart1);

}

void UART\_Send(char \*message) {

HAL\_UART\_Transmit(&huart1, (uint8\_t \*)message, strlen(message), HAL\_MAX\_DELAY);

}

void UART\_Receive(char \*buffer, uint16\_t size) {

HAL\_UART\_Receive(&huart1, (uint8\_t \*)buffer, size, HAL\_MAX\_DELAY);

}

## 7.2 – I2C: Comunicação com sensores e dispositivos

**Introdução**  
O protocolo I2C (Inter-Integrated Circuit) é utilizado para comunicação entre dispositivos usando dois fios: SDA (dados) e SCL (clock).

**Passos para configuração:**

Configuração do clock: Habilite o clock do periférico I2C no RCC.

Configuração dos pinos: Configure SDA e SCL como saída alternada open-drain com pull-up.

Configuração do periférico I2C:

Defina o endereço do dispositivo mestre/escravo.

Configure a velocidade do clock (geralmente 100 kHz ou 400 kHz).

Envio e recepção de dados: Use as funções HAL ou manipule diretamente os registros.

**Exemplo com HAL:**

I2C\_HandleTypeDef hi2c1;

void I2C\_Init() {

\_\_HAL\_RCC\_I2C1\_CLK\_ENABLE();

hi2c1.Instance = I2C1;

hi2c1.Init.ClockSpeed = 100000;

hi2c1.Init.DutyCycle = I2C\_DUTYCYCLE\_2;

hi2c1.Init.OwnAddress1 = 0x00;

hi2c1.Init.AddressingMode = I2C\_ADDRESSINGMODE\_7BIT;

hi2c1.Init.DualAddressMode = I2C\_DUALADDRESS\_DISABLE;

hi2c1.Init.GeneralCallMode = I2C\_GENERALCALL\_DISABLE;

hi2c1.Init.NoStretchMode = I2C\_NOSTRETCH\_DISABLE;

HAL\_I2C\_Init(&hi2c1);

}

void I2C\_Write(uint16\_t DevAddress, uint8\_t \*data, uint16\_t size) {

HAL\_I2C\_Master\_Transmit(&hi2c1, DevAddress, data, size, HAL\_MAX\_DELAY);

}

void I2C\_Read(uint16\_t DevAddress, uint8\_t \*buffer, uint16\_t size) {

HAL\_I2C\_Master\_Receive(&hi2c1, DevAddress, buffer, size, HAL\_MAX\_DELAY);

}

## 7.3 – SPI: Comunicação de alta velocidade

**Introdução**  
O protocolo SPI (Serial Peripheral Interface) é utilizado para comunicações rápidas entre dispositivos, como displays, sensores e memórias flash.

**Passos para configuração:**

1. Configuração do clock: Habilite o clock do periférico SPI no RCC.
2. Configuração dos pinos: Configure os pinos MOSI, MISO e SCK como saída alternada.

* NSS (opcional): configurado como entrada ou controlado por software.

1. Configuração do periférico SPI:

* Configure o modo (mestre ou escravo).
* Defina a frequência do clock.
* Configure o formato dos dados (8 ou 16 bits) e a polaridade/fase do clock.

1. Envio e recepção de dados: Use as funções HAL ou manipule os registros.

**Exemplo com HAL:**

SPI\_HandleTypeDef hspi1;

void SPI\_Init() {

\_\_HAL\_RCC\_SPI1\_CLK\_ENABLE();

hspi1.Instance = SPI1;

hspi1.Init.Mode = SPI\_MODE\_MASTER;

hspi1.Init.Direction = SPI\_DIRECTION\_2LINES;

hspi1.Init.DataSize = SPI\_DATASIZE\_8BIT;

hspi1.Init.CLKPolarity = SPI\_POLARITY\_LOW;

hspi1.Init.CLKPhase = SPI\_PHASE\_1EDGE;

hspi1.Init.NSS = SPI\_NSS\_SOFT;

hspi1.Init.BaudRatePrescaler = SPI\_BAUDRATEPRESCALER\_16;

hspi1.Init.FirstBit = SPI\_FIRSTBIT\_MSB;

HAL\_SPI\_Init(&hspi1);

}

void SPI\_Transmit(uint8\_t \*data, uint16\_t size) {

HAL\_SPI\_Transmit(&hspi1, data, size, HAL\_MAX\_DELAY);

}

void SPI\_Receive(uint8\_t \*buffer, uint16\_t size) {

HAL\_SPI\_Receive(&hspi1, buffer, size, HAL\_MAX\_DELAY);

}

# 8 – Temporizadores e PWM

Os temporizadores (Timers) são periféricos essenciais no STM32, permitindo o controle preciso de tempo, geração de sinais periódicos e modulação por largura de pulso (PWM). Neste capítulo, abordaremos a configuração de temporizadores básicos e o uso do PWM para controle de cargas.

## 8.1 – Configuração de temporizadores básicos

**Introdução**  
Os temporizadores básicos no STM32F103C8T6 (como TIM2 e TIM3) são utilizados para contagem de tempo, geração de interrupções e outras funções baseadas em tempo.

**Passos para configuração de temporizadores básicos:**

1. Ativação do clock: Ative o clock do temporizador desejado no RCC.
2. Configuração do temporizador:

* Defina o valor do *prescaler* para ajustar a frequência do contador.
* Configure o valor do *auto-reload* (ARR) para definir o período do temporizador.
* Habilite o modo contínuo ou uma única execução.

1. Interrupções (opcional):

* Habilite a interrupção no temporizador.
* Implemente a rotina de tratamento de interrupção (ISR).

1. Início do temporizador: Inicie o temporizador e, se necessário, ative a geração de eventos.

**Exemplo com HAL:**

TIM\_HandleTypeDef htim2;

void Timer\_Init() {

\_\_HAL\_RCC\_TIM2\_CLK\_ENABLE();

htim2.Instance = TIM2;

htim2.Init.Prescaler = 7200 - 1; // 72 MHz / 7200 = 10 kHz

htim2.Init.CounterMode = TIM\_COUNTERMODE\_UP;

htim2.Init.Period = 10000 - 1; // 10 kHz / 10000 = 1 Hz

htim2.Init.ClockDivision = TIM\_CLOCKDIVISION\_DIV1;

HAL\_TIM\_Base\_Init(&htim2);

}

void Timer\_Start() {

HAL\_TIM\_Base\_Start\_IT(&htim2); // Inicia o temporizador com interrupções

}

void TIM2\_IRQHandler() {

HAL\_TIM\_IRQHandler(&htim2);

}

void HAL\_TIM\_PeriodElapsedCallback(TIM\_HandleTypeDef \*htim) {

if (htim->Instance == TIM2) {

// Código a ser executado a cada período

}

}

## 8.2 – Geração de sinais PWM para controle de cargas

**Introdução**  
O PWM (Pulse Width Modulation) é amplamente utilizado para controle de cargas como LEDs, motores e ventiladores. No STM32, o PWM é gerado por temporizadores configurados em modo de saída.

**Passos para configuração do PWM:**

1. Ativação do clock: Ative o clock do temporizador usado para o PWM e configure o pino GPIO como saída alternada.
2. Configuração do temporizador:

* Configure o *prescaler* e o *auto-reload* para definir a frequência do PWM.
* Habilite o modo de saída PWM no canal desejado.

1. Configuração do ciclo de trabalho (*duty cycle*): Ajuste o valor do registrador CCRx (onde x é o número do canal) para controlar o *duty cycle*.
2. Início do PWM: Inicie o temporizador no modo PWM.

**Exemplo com HAL:**

TIM\_HandleTypeDef htim3;

void PWM\_Init() {

\_\_HAL\_RCC\_TIM3\_CLK\_ENABLE();

htim3.Instance = TIM3;

htim3.Init.Prescaler = 720 - 1; // 72 MHz / 720 = 100 kHz

htim3.Init.CounterMode = TIM\_COUNTERMODE\_UP;

htim3.Init.Period = 1000 - 1; // 100 kHz / 1000 = 100 Hz

htim3.Init.ClockDivision = TIM\_CLOCKDIVISION\_DIV1;

HAL\_TIM\_PWM\_Init(&htim3);

TIM\_OC\_InitTypeDef sConfigOC;

sConfigOC.OCMode = TIM\_OCMODE\_PWM1;

sConfigOC.Pulse = 500; // 50% duty cycle

sConfigOC.OCPolarity = TIM\_OCPOLARITY\_HIGH;

sConfigOC.OCFastMode = TIM\_OCFAST\_DISABLE;

HAL\_TIM\_PWM\_ConfigChannel(&htim3, &sConfigOC, TIM\_CHANNEL\_1);

}

void PWM\_Start() {

HAL\_TIM\_PWM\_Start(&htim3, TIM\_CHANNEL\_1); // Inicia PWM no canal 1

}

void PWM\_SetDutyCycle(uint16\_t duty) {

\_\_HAL\_TIM\_SET\_COMPARE(&htim3, TIM\_CHANNEL\_1, duty); // Ajusta o duty cycle

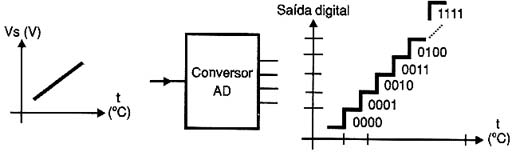
}

# 9 – Conversores Analógico-Digital (ADC)

O STM32F103C8T6 possui um conversor ADC de 12 bits capaz de realizar leituras rápidas e precisas de sinais analógicos. Este capítulo aborda a configuração para leitura de sensores analógicos, uso de múltiplos canais e a integração com o DMA para automação do processo de conversão.

## 9.1 – Leitura de sensores analógicos

**Introdução**  
O ADC converte um sinal analógico (como a saída de um sensor) em um valor digital que pode ser processado pelo microcontrolador. Com resolução de 12 bits, os valores digitais variam de 0 a 4095, representando a tensão de entrada proporcional à tensão de referência.



**Passos para configuração:**

1. Ativação do clock: Habilite o clock do ADC no RCC.
2. Configuração do pino analógico: Configure o pino GPIO usado como entrada analógica (modo analógico, sem resistores pull-up ou pull-down).
3. Configuração do ADC:

* Defina o modo de operação (contínuo ou único).
* Configure o canal de entrada e o tempo de amostragem.
* Calibre o ADC antes do uso para maior precisão.

1. Início da conversão: Inicie o ADC e leia os valores convertidos.

**Exemplo com HAL:**

ADC\_HandleTypeDef hadc1;

void ADC\_Init() {

\_\_HAL\_RCC\_ADC1\_CLK\_ENABLE();

hadc1.Instance = ADC1;

hadc1.Init.ScanConvMode = ADC\_SCAN\_DISABLE; // Modo de canal único

hadc1.Init.ContinuousConvMode = ENABLE; // Conversão contínua

hadc1.Init.DiscontinuousConvMode = DISABLE;

hadc1.Init.ExternalTrigConv = ADC\_SOFTWARE\_START;

hadc1.Init.DataAlign = ADC\_DATAALIGN\_RIGHT;

hadc1.Init.NbrOfConversion = 1;

HAL\_ADC\_Init(&hadc1);

// Configuração do canal

ADC\_ChannelConfTypeDef sConfig;

sConfig.Channel = ADC\_CHANNEL\_0; // Canal 0 (PA0)

sConfig.Rank = 1;

sConfig.SamplingTime = ADC\_SAMPLETIME\_71CYCLES\_5;

HAL\_ADC\_ConfigChannel(&hadc1, &sConfig);

}

uint16\_t ADC\_Read() {

HAL\_ADC\_Start(&hadc1); // Inicia a conversão

HAL\_ADC\_PollForConversion(&hadc1, HAL\_MAX\_DELAY); // Aguarda término

return HAL\_ADC\_GetValue(&hadc1); // Retorna o valor convertido

}

# 9.2 – Configuração de múltiplos canais e DMA

**Introdução**  
O STM32F103C8T6 permite configurar o ADC para ler múltiplos canais em sequência, ideal para aplicações com vários sensores. Além disso, o uso de DMA (Direct Memory Access) automatiza a transferência dos dados convertidos, reduzindo a carga no processador.

O DMA permite a transferência de dados entre periféricos e a memória (ou entre regiões de memória) sem a intervenção do processador. Isso libera a CPU para outras tarefas enquanto os dados são transferidos automaticamente.

Vantagens do DMA:

* Reduz a carga no processador.
* Permite transferências rápidas e automáticas.
* Ideal para aplicações que envolvem grandes volumes de dados, como comunicação serial, leitura de ADCs, ou saída de dados por PWM.

**Passos para configuração:**

1. Ativação do clock: Ative o clock do ADC e do DMA no RCC.
2. Configuração dos pinos: Configure todos os pinos analógicos necessários (em modo analógico).
3. Configuração do ADC:

* Ative o modo de varredura (*scan mode*).
* Defina o número de canais a serem convertidos e seus tempos de amostragem.
* Configure o modo contínuo, se necessário.

1. Configuração do DMA:

* Vincule o ADC ao DMA.
* Configure o DMA para transferir os dados do ADC para uma área de memória.
* Especifique a fonte e o destino dos dados.
* Configure o tamanho dos dados e o modo de operação (normal ou circular).
* Defina as prioridades.

1. Início da conversão: Inicie o ADC no modo contínuo e habilite o DMA.

O DMA transfere automaticamente os valores convertidos para o buffer adcBuffer.

Cada posição do buffer corresponde a um canal:

* adcBuffer[0]: Canal 0 (PA0)
* adcBuffer[1]: Canal 1 (PA1)
* adcBuffer[2]: Canal 2 (PA2)

**Exemplo com HAL:**

ADC\_HandleTypeDef hadc1;

DMA\_HandleTypeDef hdma\_adc1;

uint16\_t adcBuffer[3]; // Buffer para armazenar valores de 3 canais

void ADC\_DMA\_Init() {

\_\_HAL\_RCC\_ADC1\_CLK\_ENABLE();

\_\_HAL\_RCC\_DMA1\_CLK\_ENABLE();

// Configuração do ADC

hadc1.Instance = ADC1;

hadc1.Init.ScanConvMode = ADC\_SCAN\_ENABLE; // Modo de múltiplos canais

hadc1.Init.ContinuousConvMode = ENABLE;

hadc1.Init.DiscontinuousConvMode = DISABLE;

hadc1.Init.ExternalTrigConv = ADC\_SOFTWARE\_START;

hadc1.Init.DataAlign = ADC\_DATAALIGN\_RIGHT;

hadc1.Init.NbrOfConversion = 3; // 3 canais

HAL\_ADC\_Init(&hadc1);

// Configuração dos canais

ADC\_ChannelConfTypeDef sConfig;

sConfig.SamplingTime = ADC\_SAMPLETIME\_71CYCLES\_5;

sConfig.Channel = ADC\_CHANNEL\_0; // Canal 0 (PA0)

sConfig.Rank = 1;

HAL\_ADC\_ConfigChannel(&hadc1, &sConfig);

sConfig.Channel = ADC\_CHANNEL\_1; // Canal 1 (PA1)

sConfig.Rank = 2;

HAL\_ADC\_ConfigChannel(&hadc1, &sConfig);

sConfig.Channel = ADC\_CHANNEL\_2; // Canal 2 (PA2)

sConfig.Rank = 3;

HAL\_ADC\_ConfigChannel(&hadc1, &sConfig);

// Configuração do DMA

hdma\_adc1.Instance = DMA1\_Channel1;

hdma\_adc1.Init.Direction = DMA\_PERIPH\_TO\_MEMORY;

hdma\_adc1.Init.PeriphInc = DMA\_PINC\_DISABLE;

hdma\_adc1.Init.MemInc = DMA\_MINC\_ENABLE;

hdma\_adc1.Init.PeriphDataAlignment = DMA\_PDATAALIGN\_HALFWORD;

hdma\_adc1.Init.MemDataAlignment = DMA\_MDATAALIGN\_HALFWORD;

hdma\_adc1.Init.Mode = DMA\_CIRCULAR;

hdma\_adc1.Init.Priority = DMA\_PRIORITY\_LOW;

HAL\_DMA\_Init(&hdma\_adc1);

\_\_HAL\_LINKDMA(&hadc1, DMA\_Handle, hdma\_adc1);

}

void ADC\_DMA\_Start() {

HAL\_ADC\_Start\_DMA(&hadc1, (uint32\_t \*)adcBuffer, 3); // 3 canais

}

# 10 – Interrupções

O STM32F103C8T6 oferece suporte a **interrupções** importantes para melhorar a eficiência e a responsividade do sistema. Este capítulo aborda o conceito de interrupções e como configurá-las.

**O que são interrupções e como usá-las**

Interrupções são eventos que pausam temporariamente a execução normal do código para que o microcontrolador possa tratar eventos específicos, como a recepção de dados ou o término de uma conversão ADC. Após o tratamento, o programa retorna ao ponto onde foi interrompido.

**Vantagens das interrupções:**

* Melhoram a eficiência ao evitar o uso de *polling*.
* Permitem respostas rápidas a eventos externos ou internos.
* Reduzem a carga de processamento em tarefas que não exigem supervisão contínua.

**Passos para configurar interrupções no STM32:**

1. **Habilitação do clock:** Ative o clock do periférico relacionado à interrupção (por exemplo, USART, ADC, etc.).
2. **Configuração do periférico:** Configure o periférico para gerar uma interrupção quando ocorrer o evento desejado.
3. **Habilitação da interrupção no NVIC:** Ative a linha de interrupção no NVIC (Nested Vectored Interrupt Controller).
4. **Implementação da rotina de tratamento:** Crie a função de tratamento (ISR) para processar o evento.

**Exemplo: Configurando uma interrupção para um botão (EXTI)**

* Quando o botão conectado ao pino PA0 é pressionado, uma interrupção é gerada.
* O NVIC direciona a execução para a função EXTI0\_IRQHandler, que chama o *callback* para o tratamento do evento (botão pressionado, sinal digital recebido, etc.).

void EXTI\_Init() {

// Ativar clock do GPIO

\_\_HAL\_RCC\_GPIOA\_CLK\_ENABLE();

// Configurar PA0 como entrada com pull-down

GPIO\_InitTypeDef GPIO\_InitStruct = {0};

GPIO\_InitStruct.Pin = GPIO\_PIN\_0;

GPIO\_InitStruct.Mode = GPIO\_MODE\_IT\_RISING;

GPIO\_InitStruct.Pull = GPIO\_PULLDOWN;

HAL\_GPIO\_Init(GPIOA, &GPIO\_InitStruct);

// Habilitar a linha de interrupção no NVIC

HAL\_NVIC\_SetPriority(EXTI0\_IRQn, 2, 0); // Prioridade

HAL\_NVIC\_EnableIRQ(EXTI0\_IRQn); // Habilitar interrupção

}

// Rotina de tratamento de interrupção (ISR)

void EXTI0\_IRQHandler() {

HAL\_GPIO\_EXTI\_IRQHandler(GPIO\_PIN\_0);

}

void HAL\_GPIO\_EXTI\_Callback(uint16\_t GPIO\_Pin) {

if (GPIO\_Pin == GPIO\_PIN\_0) {

// Código executado quando o botão é pressionado

}

}

# 11 – Debugging e Otimização

Trabalhar com microcontroladores como o STM32F103C8T6 exige técnicas eficientes de depuração e otimização para garantir que o código funcione corretamente e com desempenho adequado. Este capítulo aborda o uso do depurador integrado, o monitoramento de variáveis em tempo real e práticas de otimização de código.

## 11.1 – Uso do depurador integrado

O STM32F103C8T6 suporta depuração por meio de interfaces como SWD (Serial Wire Debug) e JTAG. Essas interfaces permitem a inspeção do estado do microcontrolador em tempo real, além de fornecer ferramentas como pontos de interrupção (*breakpoints*), rastreamento de execução e inspeção de registros.

**Configuração do depurador:**

1. **Ferramentas necessárias:**

* Uma interface de depuração, como ST-Link V2.
* Software de desenvolvimento, como STM32CubeIDE ou Keil uVision.

1. **Conexão física:**

* Conecte o ST-Link ao microcontrolador via SWD (pinos SWDIO, SWCLK, GND e VCC).

1. **Configuração no software:**

* No STM32CubeIDE, crie um novo *debug configuration*:
* Vá em **Run > Debug Configurations**.
* Configure o ST-Link como a interface de depuração.
* Compile o projeto em modo de depuração (geralmente *Debug* ao invés de *Release*).

1. **Uso do depurador:**

* Inicie o depurador e use as ferramentas disponíveis:
* **Breakpoints:** Pause a execução em linhas específicas.
* **Step In/Out/Over:** Execute o código passo a passo.
* **Watch Variables:** Monitore variáveis em tempo real.

## 11.2 – Monitoramento de variáveis em tempo real

O monitoramento em tempo real permite acompanhar o valor de variáveis e registradores sem interromper a execução do código, uma técnica especialmente útil em sistemas embarcados onde a interrupção do programa pode causar problemas.

**Ferramentas disponíveis:**

1. **SWV (Serial Wire Viewer):**

* Disponível em alguns depuradores, como o ST-Link.
* Permite o envio de dados em tempo real via SWD.

1. **ITM (Instrumentation Trace Macrocell):**

Ferramenta do ARM Cortex-M para enviar mensagens de depuração ao host.

1. **Visualização no STM32CubeIDE:**

* Use o *Live Expressions* para monitorar variáveis em tempo real.
* Configure as variáveis no painel **Expressions**.

## 11.3 – Técnicas de otimização de código

**Por que otimizar?**  
A otimização melhora o desempenho, reduz o consumo de memória e energia, e aumenta a confiabilidade do sistema.

**Técnicas de otimização:**

**1. Uso eficiente de periféricos:**

* Prefira **DMA** para transferências de dados ao invés de processá-las no código.
* Utilize **interrupções** para eventos assíncronos, evitando o uso de *polling*.

**2. Configurações de compilação:**

Use o modo de otimização do compilador:

**-O0:** Sem otimização (para depuração).

**-O1:** Otimização básica.

**-O2:** Otimização para desempenho.

**-Os:** Otimização para reduzir o tamanho do código.

* No STM32CubeIDE, configure em **Project Properties > C/C++ Build > Settings > Optimization Level**.

**3. Análise de desempenho:**

* Use ferramentas de perfilamento para identificar gargalos no código.
* No STM32CubeIDE, ative o **Performance Analyzer** para medir tempos de execução.

**4. Redução de consumo de memória:**

* Prefira variáveis locais em vez de globais.
* Use tipos de dados menores sempre que possível (ex.: uint8\_t ao invés de uint32\_t para valores pequenos).

**5. Escrita de código eficiente:**

* Evite loops desnecessários e funções recursivas.
* Utilize funções *inline* para pequenos trechos de código.
* Substitua cálculos repetitivos por constantes ou pré-calculados.

**6. Uso de bibliotecas otimizadas:**

* Utilize as bibliotecas HAL ou LL (Low Layer) da STMicroelectronics, ajustando o nível de abstração às necessidades de desempenho.

# 12 – Uso de RTOS no STM32F103C8T6

O uso de um sistema operacional em tempo real (RTOS) em microcontroladores como o STM32F103C8T6 permite a execução eficiente de múltiplas tarefas, gerenciamento de recursos e sincronização entre eventos. Este capítulo aborda os conceitos básicos de RTOS, sua configuração no STM32 e exemplos práticos com o FreeRTOS, um dos RTOS mais populares e amplamente suportados.

**12.1 – O que é um RTOS?**

Um RTOS é um sistema operacional projetado para garantir que tarefas críticas sejam executadas dentro de limites de tempo bem definidos. Ele organiza a execução de tarefas em **threads** ou **tasks** e oferece serviços como:

* **Gerenciamento de tarefas:** Controle da criação, priorização e troca de contexto entre tarefas.
* **Sincronização:** Ferramentas como semáforos para coordenar o acesso a recursos compartilhados.
* **Temporização:** Funções para atrasos precisos e execução periódica de tarefas.
* **Comunicação:** Fila de mensagens e outros mecanismos para troca de dados entre tarefas.

**Por que usar um RTOS?**

* Facilita o desenvolvimento de sistemas complexos.
* Melhora a previsibilidade e organização do código.
* Garante respostas rápidas e determinísticas para eventos.

**12.2 – Configuração do FreeRTOS no STM32F103C8T6**

O FreeRTOS é uma escolha comum para RTOS no STM32 devido à sua simplicidade, suporte e integração com ferramentas como STM32CubeMX e STM32CubeIDE.

**Passos para configurar o FreeRTOS:**

1. **Instalação e Configuração no STM32CubeMX:**

* Abra o STM32CubeMX e configure o projeto para o STM32F103C8T6.
* Vá até a aba **Middleware** e habilite o **FreeRTOS**.
* Configure os parâmetros principais:
* **Heap Size:** Memória para alocação dinâmica (ex.: 1024 bytes).
* **Tick Rate:** Frequência de execução do kernel (geralmente 1ms).
* Adicione tarefas iniciais e configure suas prioridades.

1. **Geração do código:**

* Gere o código no STM32CubeIDE. O STM32CubeMX adicionará automaticamente os arquivos necessários do FreeRTOS ao projeto.

1. **Criação de tarefas no código:**

* Utilize a função xTaskCreate para criar tarefas.
* Defina as prioridades de cada tarefa para garantir que tarefas críticas tenham precedência.

**12.3 – Exemplo prático: Criando tarefas no FreeRTOS**

Exemplo: Alternar LEDs usando FreeRTOS

**Objetivo:** Alternar dois LEDs com diferentes períodos utilizando tarefas.

* Duas tarefas (LED\_Task1 e LED\_Task2) alternam LEDs com diferentes períodos.
* vTaskDelay é usado para criar atrasos precisos sem bloquear o processador.

#include "FreeRTOS.h"

#include "task.h"

#include "stm32f1xx\_hal.h"

void LED\_Task1(void \*pvParameters) {

while (1) {

HAL\_GPIO\_TogglePin(GPIOC, GPIO\_PIN\_13); // Alterna o LED no pino PC13

vTaskDelay(pdMS\_TO\_TICKS(500)); // Atraso de 500ms

}

}

void LED\_Task2(void \*pvParameters) {

while (1) {

HAL\_GPIO\_TogglePin(GPIOB, GPIO\_PIN\_0); // Alterna o LED no pino PB0

vTaskDelay(pdMS\_TO\_TICKS(1000)); // Atraso de 1000ms

}

}

int main(void) {

HAL\_Init();

SystemClock\_Config();

// Configuração de GPIOs

\_\_HAL\_RCC\_GPIOC\_CLK\_ENABLE();

\_\_HAL\_RCC\_GPIOB\_CLK\_ENABLE();

GPIO\_InitTypeDef GPIO\_InitStruct = {0};

GPIO\_InitStruct.Pin = GPIO\_PIN\_13;

GPIO\_InitStruct.Mode = GPIO\_MODE\_OUTPUT\_PP;

GPIO\_InitStruct.Speed = GPIO\_SPEED\_FREQ\_LOW;

HAL\_GPIO\_Init(GPIOC, &GPIO\_InitStruct);

GPIO\_InitStruct.Pin = GPIO\_PIN\_0;

HAL\_GPIO\_Init(GPIOB, &GPIO\_InitStruct);

// Criação das tarefas

xTaskCreate(LED\_Task1, "LED1", 128, NULL, 1, NULL);

xTaskCreate(LED\_Task2, "LED2", 128, NULL, 1, NULL);

// Iniciar o agendador do FreeRTOS

vTaskStartScheduler();

// O programa nunca deve chegar aqui

while (1);

}

**12.4 – Sincronização entre tarefas**

Em sistemas RTOS, é comum que várias tarefas compartilhem recursos ou precisem ser sincronizadas. O FreeRTOS fornece ferramentas como:

**1. Semáforos:**

Usados para sinalizar eventos ou gerenciar acesso a recursos compartilhados.

**Exemplo:** Sincronizar tarefas com semáforos binários:

#include "semphr.h"

SemaphoreHandle\_t xSemaphore;

void Task1(void \*pvParameters) {

while (1) {

xSemaphoreGive(xSemaphore); // Libera o semáforo

vTaskDelay(pdMS\_TO\_TICKS(1000));

}

}

void Task2(void \*pvParameters) {

while (1) {

if (xSemaphoreTake(xSemaphore, portMAX\_DELAY)) { // Espera pelo semáforo

// Executa algo após receber o semáforo

}

}

}

**2. Filas:**

Permitem a troca de dados entre tarefas de forma segura.

Exemplo: Enviar e receber dados por fila:

#include "queue.h"

QueueHandle\_t xQueue;

void Sender\_Task(void \*pvParameters) {

int value = 0;

while (1) {

xQueueSend(xQueue, &value, portMAX\_DELAY);

value++;

vTaskDelay(pdMS\_TO\_TICKS(500));

}

}

void Receiver\_Task(void \*pvParameters) {

int receivedValue;

while (1) {

if (xQueueReceive(xQueue, &receivedValue, portMAX\_DELAY)) {

// Processa o valor recebido

}

}

}

**12.5 – Dicas de otimização com RTOS**

1. **Evite prioridades iguais:** Prioridades iguais podem causar inversão de prioridades e ineficiência.
2. **Minimize o uso de atrasos longos:** Use filas ou semáforos para sincronizar tarefas em vez de atrasos extensos.
3. **Reduza o uso de alocação dinâmica:** Sempre que possível, use alocação estática para evitar fragmentação de memória.
4. **Perfilamento e análise:** Use ferramentas de análise como o Tracealyzer para monitorar o desempenho do RTOS.

**12.6 – Benefícios e limitações do RTOS no STM32F103C8T6**

**Benefícios:**

* Organização do código em tarefas independentes.
* Gerenciamento eficiente de eventos assíncronos.
* Melhor escalabilidade para projetos maiores.

**Limitações:**

* Consumo adicional de memória RAM.
* Requer planejamento cuidadoso para evitar problemas como *deadlocks*.

# Configuração para geração dos binários

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente