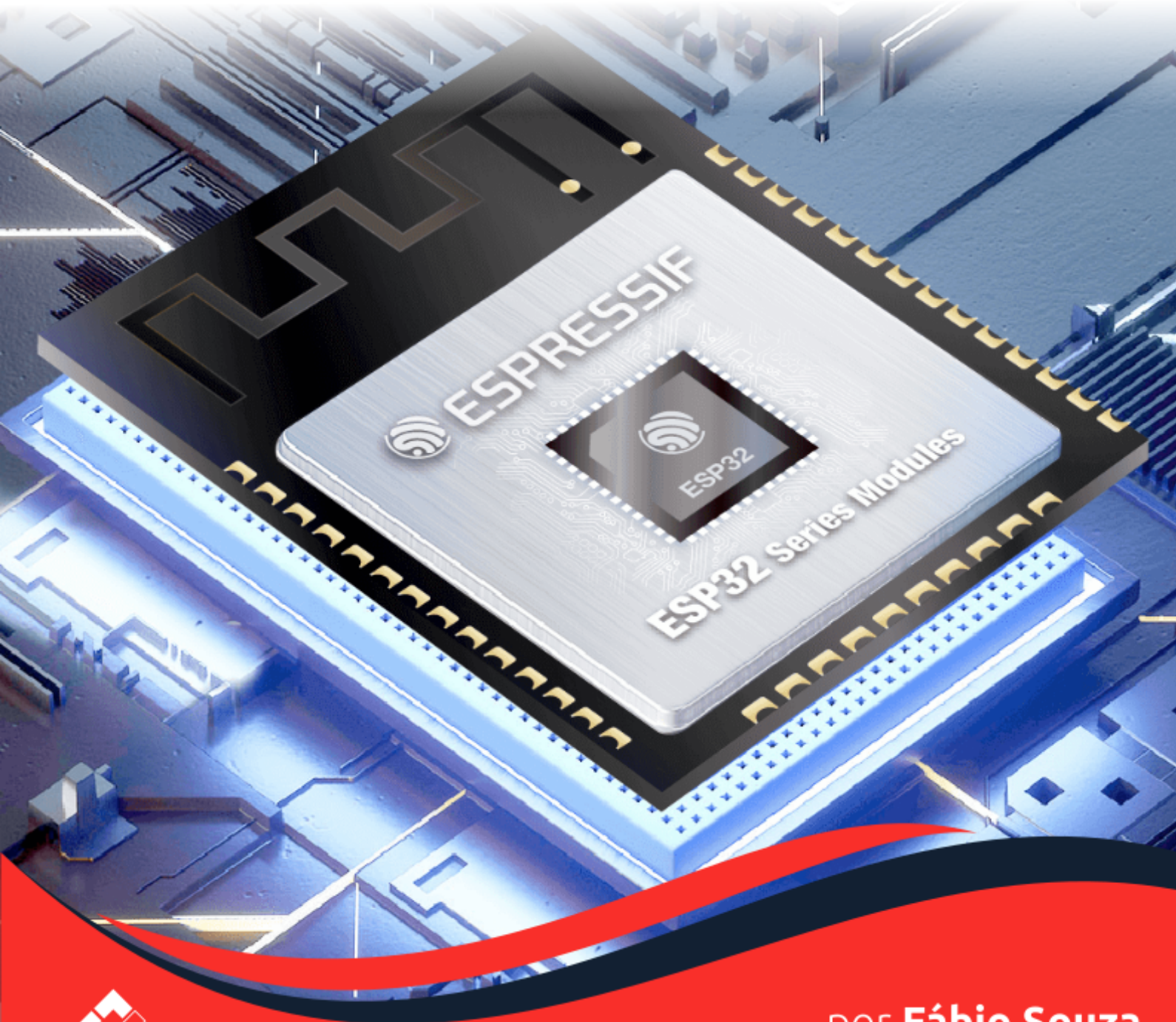
Explorando o Potencial do

ESP32

Guia de Iniciação ao ESP-IDF 5



EMBARCADOS

por **Fábio Souza**

Aviso legal

Este eBook foi escrito com fins didáticos, e com todos os esforço para que ele ficasse o mais claro e didático possível.

O objetivo deste eBook é educar. O autor não garante que as informações contidas neste eBook estão totalmente completas e não deve ser responsável por quaisquer erros ou omissões.

O autor não será responsabilizado com qualquer pessoa ou entidade com relação a qualquer perda, ou dano causado, ou alegado a ser causado direta, ou indiretamente por este eBook.

Se achar algum erro, ou tiver sugestões de tópicos, ou melhorias, me envie um e-mail: contato@embarcados.com.br

Sobre o Embarcados

O Portal Embarcados tem como foco inspirar qualidade e inovação tecnológica, disseminando o conhecimento da área de sistemas embarcados.

Atualmente, o Portal Embarcados é um grande site com conteúdo técnico sobre hardware, firmware e assunto relacionados a Embarcados e IoT, e boa parte desse conteúdo é criado pela comunidade, por pessoas que compartilham seu conhecimento na área de sistemas eletrônicos. Em Novembro de 2022 cerca de 190 mil pessoas visualizam páginas do Embarcados, o que confirma o site como uma fonte popular de referência nos assuntos referentes a sistemas embarcados, sistemas eletrônicos e IoT (Internet das Coisas).

Estamos formando uma grande base de conhecimento acessível a todos e viabilizando a troca de informações e experiências que ajudam a crescer e desenvolver nossa capacidade como profissionais da área de sistemas embarcados. O Portal convida a todos, incluindo makers e curiosos, a consultarem os textos e a compartilharem seu conhecimento através de tutoriais, videos e projetos. Envie o seu texto pro Embarcados no link.

O Portal Embarcados também realiza eventos com a comunidade e também webinars, que são veiculados pelo menos 2 vezes por mês com o apoio de empresas e com pessoas de notório saber em assuntos relacionados a sistemas embarcados.

- Para acessar os Webinars do Embarcados, acesse o link.
- Para acessar o evento Seminário de Sistemas Embarcados e IoT 2020, acesse o link.
- Para acessar o evento Embarcados Experience 2020, acesse o link.
- Para acessar o evento Embarcados Experience 2022, acesse o link.

A plataforma Embarcados Contest recebe competições e programas que permitem com que pessoas e empresas desenvolvam seus projetos com um suporte e atenção especial das empresas envolvidas nesses desafios. Para acessar os concursos já realizados, acesse o link.

Canal do Youtube – Lives, Webinars, Videos Técnicos e Podcasts

Canal do Youtube – Lives, Webinars, Videos Técnicos e Podcasts

Possuímos uma sessão com vídeos de reviews, lives, unboxing e ensino em sistemas embarcados, e onde concentramos os vídeos de nossos eventos e Webinars passados. Toda quarta-feira a tarde, às 17:00, temos o Café com Embarcados, com entrevistas com convidados especiais que trabalham na área. Toda sexta-feira à tarde, às 17:00, temos a Bancada do Embarcados, com assuntos técnicos, relacionados a Firmware, Hardware e ferramentas de nuvem. Para acessar o EmbarcadosTV clique aqui.

Siga o Embarcados na Redes Sociais

https://www.facebook.com/osembarcados/
https://www.instagram.com/portalembarcados/
https://www.youtube.com/embarcadostv/
https://www.linkedin.com/company/embarcados/
https://twitter.com/embarcados

Publique seus textos no Embarcados

Clique aqui e veja como mandar textos para o Embarcados.

Seção de oportunidade de trabalho na área de Embarcados

Confira aqui as melhores vagas na área de sistemas eletrônicos embarcados.

Sobre o Autor

Fabio Souza é um engenheiro com ampla experiência no desenvolvimento de projetos eletrônicos. Atualmente, como diretor do portal Embarcados, dedica-se a promover a área de desenvolvimento de projetos eletrônicos, sistemas embarcados e IoT no Brasil.

Com seu profundo conhecimento em eletrônica e programação, Fabio atua como professor de graduação e pós-graduação, bem como ministra cursos livres e exclusivos para empresas. Ele é um entusiasta do movimento maker, da cultura DIY e do compartilhamento de conhecimento, publicando diversos artigos, projetos open hardware e sendo autor de livros da área.

Por meio de iniciativas como o projeto Franzininho e outros projetos na área de educação, leva a cultura maker para o Brasil, capacitando e incentivando professores e alunos a usarem a tecnologia em suas vidas. Sua dedicação é fundamental para impulsionar a inovação e o empreendedorismo no país.

Conecte-se com Fábio Souza:

Redes Sociais:

Youtube:

Seus Artigos no Embarcados

Sumário

Introdução

Seja bem-vindo ao mundo empolgante do ESP32 e ao ecossistema poderoso do ESP-IDF! Se você é um entusiasta de eletrônica, um desenvolvedor de sistemas embarcados ou está apenas começando a explorar o vasto campo da Internet das Coisas (IoT), este guia foi criado especialmente para você.

O ESP32, desenvolvido pela Espressif Systems, é um SoC Wi-Fi e Bluetooth, altamente versátil e repleto de recursos. Sua popularidade crescente deve-se à sua capacidade de oferecer conectividade sem fio combinada com um processador de alto desempenho e uma variedade de interfaces, tornando-o ideal para uma ampla gama de projetos.

O objetivo deste ebook é fornecer um ponto de partida sólido para você dar os primeiros passos com o ESP-IDF, o framework de desenvolvimento de software oficial da Espressif para toda a família do ESP32 (que hoje está bem grande). Neste guia, vamos aprender como extrair todo potencial do ESP32, aprendendo a configurar o ambiente de desenvolvimento e a aproveitar os recursos poderosos do ESP-IDF para criar projetos.

O que você encontrará neste ebook:

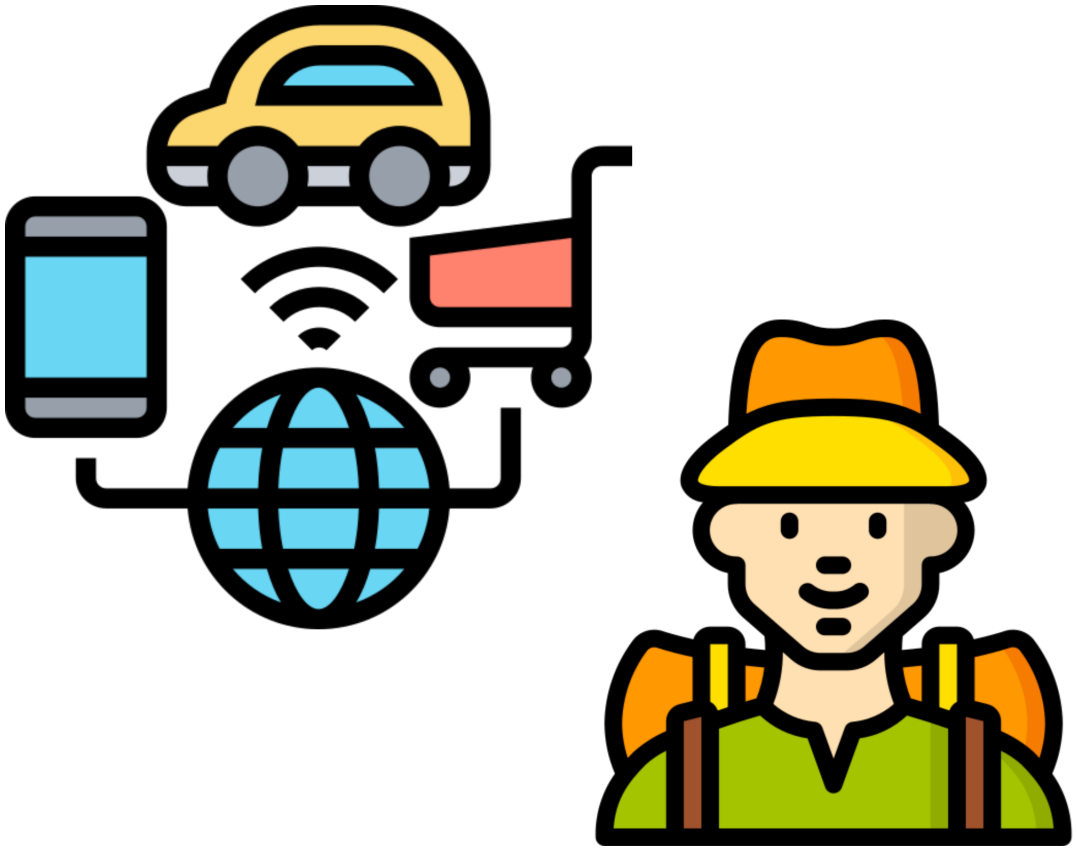
1. **Visão Geral do ESP32:** Uma visão geral das especificações e recursos que tornam o ESP32 uma escolha popular para projetos de IoT.
2. **Conhecendo o ESP-IDF:** Entenderemos o que é o ESP-IDF e como ele se relaciona com o ESP32. Veremos os principais componentes e funcionalidades do framework, que nos permitirão explorar todo o potencial do chip ESP32.
3. **Configurando o Ambiente de Desenvolvimento:** Aprenderemos a configurar o ambiente de desenvolvimento, passo a passo, para começar a escrever e compilar nossos próprios projetos com o ESP-IDF.
4. **Explorando o ESP-IDF:** Criaremos nosso primeiro projeto “Hello World” para garantir que tudo esteja configurado corretamente.
5.

6. **Exemplos de aplicações** com:

    1. GPIO
    2. PWM
    3. ADC
    4. WiFi

Então, se você está pronto para mergulhar de cabeça no ESP-IDF e desbloquear o potencial do ESP32, vamos começar a explorar este mundo fascinante de desenvolvimento de sistemas embarcados e IoT!

Você encontra todos os exemplos apresentados no livro em:
https://github.com/FBSeletronica/exemplos-ebook-esp32-com-espidf



ESP32

O ESP32 é uma família de SoC (**System-on-a-Chip**) desenvolvida pela Espressif Systems, uma empresa chinesa especializada em soluções de conectividade sem fio e sistemas embarcados. O primeiro SoC ESP32, foi lançado em 2016, e rapidamente conquistou o mundo da eletrônica e do IoT (Internet das Coisas) devido à sua combinação única de recursos avançados, baixo custo e ampla versatilidade. O ESP32 é uma das plataformas mais utilizadas atualmente, conforme exibido na pesquisa de mercado do Embarcados: Relatório da Pesquisa sobre o Mercado Brasileiro de Sistemas Embarcados e IoT 2023

Características e Especificações

O ESP32 é baseado, atualmente, nas arquiteturas: Xtensa LX6, Xtensa® 32-bit LX7 e RISC-V 32-bit. Com um rico conjunto de recursos aliado a um baixo custo, o ESP32 possibilita desenvolver diversas aplicações de sistemas embarcados. Sua família de SoCs (Sistemas em um Chip) possui diferentes categorias de comunicações sem fio e periféricos poderosos.

Atualmente, a linha de SoCs ESP32 é composta por quatro variações principais: ESP32, ESP32-S, ESP32-C e ESP32-H. Cada variação ainda possui diferentes versões.

A seguir são apresentadas as famílias com os respectivos recursos.

ESP32

- ESP32: MCU de 32 bits e Wi-Fi de 2,4 GHz e Bluetooth/Bluetooth LE:
    - Dois ou um núcleo(s) de CPU com frequência de clock ajustável, variando de 80 MHz a 240 MHz

- Potência de saída de +19,5 dBm garante uma boa cobertura física
- Bluetooth Clássico para conexões legadas, com suporte para L2CAP, SDP, GAP, SMP, AVDTP, AVCTP, A2DP (SNK) e AVRCP (CT)
- Suporte para perfis de Bluetooth de Baixa Energia (Bluetooth LE), incluindo L2CAP, GAP, GATT, SMP e perfis baseados em GATT como BluFi, SPP-like, etc
- Bluetooth de Baixa Energia (Bluetooth LE) conecta-se a smartphones e transmite beacons de baixa energia para facilitar a detecção
- Corrente de repouso é inferior a 5 µA, tornando-o adequado para aplicações com bateria e dispositivos eletrônicos vestíveis (wearables)
- Periféricos incluem sensores capacitivos de toque, sensor Hall, interface de cartão SD, Ethernet, SPI de alta velocidade, UART, I2S e I2C.

ESP32-S

- ESP32-S2: MCU de 32 bits e Wi-Fi de 2,4 GHz.
  - CPU single-core de alto desempenho de 240 MHz
  - Desempenho ultra baixo em consumo de energia: controle fino de clock, escalonamento dinâmico de tensão e frequência
  - Recursos de segurança: eFuse, criptografia de flash, inicialização segura, verificação de assinatura, algoritmos AES, SHA e RSA integrados
  - Periféricos incluem 43 GPIOs, 1 interface USB OTG de alta velocidade, SPI, I2S, UART, I2C, PWM para LED, interface LCD, interface de câmera, ADC, DAC, sensor de toque, sensor de temperatura
  - Disponibilidade de agentes de conectividade em nuvem comuns e recursos de produto comuns reduzem o tempo de lançamento no mercado.
- ESP32-S3: MCU de 32 bits e Wi-Fi de 2,4 GHz e Bluetooth 5 (LE)
  - Processador dual-core Xtensa® de 32 bits LX7 que opera a até 240 MHz
  - 512 KB de SRAM e 384 KB de ROM no chip, e interfaces SPI, Dual SPI, Quad SPI, Octal SPI, QPI e OPI que permitem a conexão com memórias flash e RAM externas
  - Suporte adicional para instruções vetoriais no MCU, que proporciona aceleração para computação de redes neurais e carga de trabalho de processamento de sinais

- Periféricos incluem 45 GPIOs programáveis, SPI, I2S, I2C, PWM, RMT, ADC e UART, host SD/MMC e TWAI™
- Recursos de segurança confiáveis garantidos por inicialização segura baseada em RSA, criptografia de flash baseada em AES-XTS, a inovadora assinatura digital e o periférico HMAC, “World Controller”.

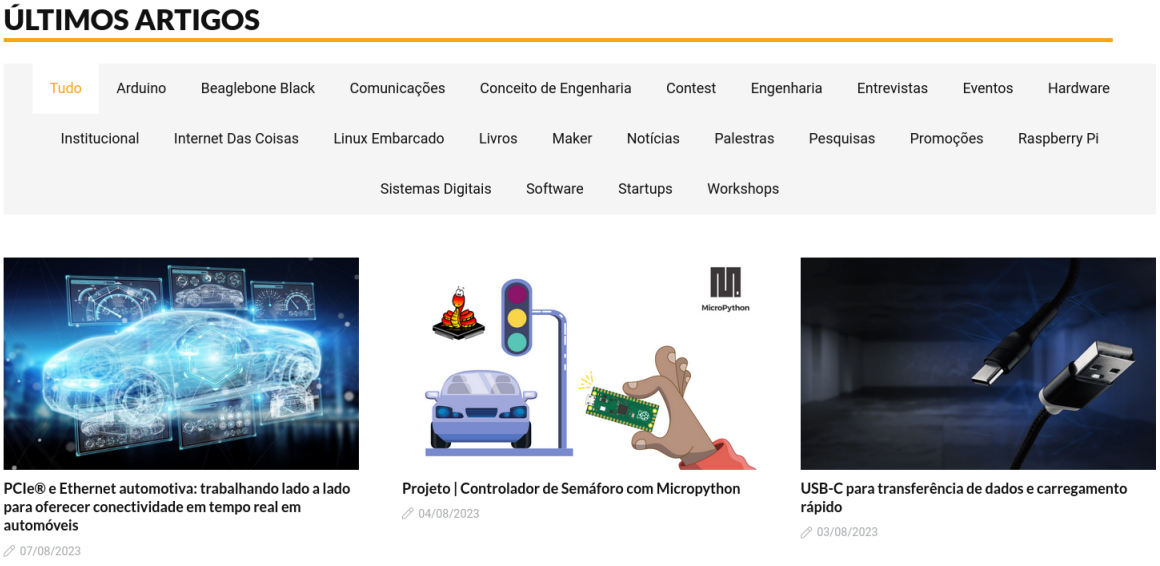
ESP32-C

- ESP32-C2: MCU RISC-V de 32 bits e Wi-Fi de 2,4 GHz e Bluetooth 5 (LE)
  - Processador RISC-V single-core de 32 bits que opera a até 120 MHz
  - Desempenho de potência e RF de última geração
  - 576 KB de ROM, 272 KB de SRAM (16 KB para cache) no chip
  - 14 GPIOs programáveis: SPI, UART, I2C, controlador PWM para LED, controlador GDMA (General DMA), ADC SAR, sensor de temperatura
- ESP32-C3: MCU RISC-V de 32 bits e Wi-Fi de 2,4 GHz e Bluetooth 5 (LE)
  - Processador RISC-V single-core de 32 bits com pipeline de quatro estágios que opera a até 160 MHz
  - Desempenho de potência e RF de última geração
  - 400 KB de SRAM e 384 KB de ROM no chip, e interfaces SPI, Dual SPI, Quad SPI e QPI que permitem a conexão com memórias flash
  - Recursos de segurança confiáveis garantidos por inicialização segura baseada em RSA-3072, criptografia de flash baseada em AES-128-XTS, a inovadora assinatura digital e o periférico HMAC, suporte para aceleração de hardware de algoritmos criptográficos
  - Conjunto rico de interfaces periféricas e GPIOs, ideal para várias situações e aplicações complexas
- ESP32-C6: MCU RISC-V de 32 bits e Wi-Fi 6 de 2,4 GHz e Bluetooth 5 (LE) e IEEE 802.15.4
  - Processador RISC-V single-core de 32 bits que opera a até 160 MHz
  - Desempenho de potência e RF de última geração
  - 320 KB de ROM, 512 KB de SRAM, 16 KB de SRAM de baixo consumo no chip, e compatibilidade com memória flash externa

- 30 GPIOs (QFN40) ou 22 GPIOs (QFN32) programáveis, com suporte para SPI, UART, I2C, I2S, RMT, TWAI e PWM

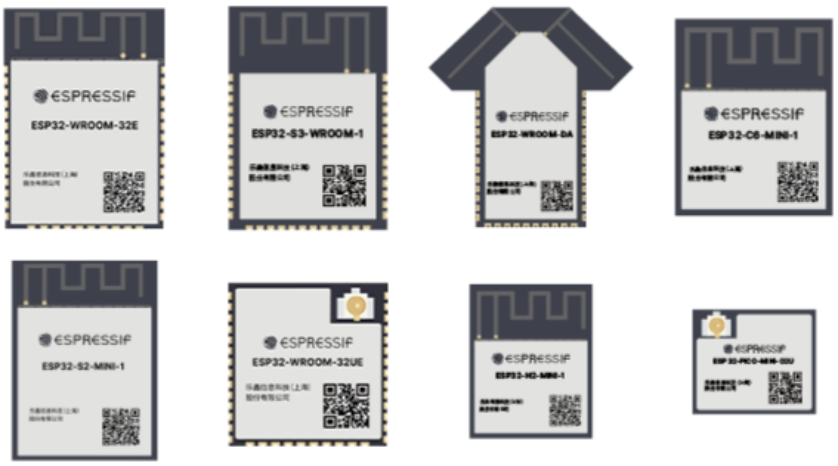
ESP32-H

- ESP32-H2: MCU RISC-V de 32 bits e Bluetooth 5 (LE) e IEEE 802.15.4
  - Processador RISC-V single-core de 32 bits que opera a até 96 MHz
  - 320 KB de SRAM, 128 KB de ROM, 4 KB de memória LP, e compatibilidade com memória flash externa
  - 19 GPIOs programáveis, com suporte para UART, SPI, I2C, I2S, Periférico de Controle Remoto, PWM para LED, Controlador USB Serial/JTAG de alta velocidade, GDMA, MCPWM
  - Pode ser usado para construir dispositivos finais da rede Thread, bem como roteador de borda Thread e ponte Matter combinando-o com o SoC ESP Wi-Fi.



Módulos

Cada família de SoCs apresentada acima possui uma variedade de módulos totalmente certificados. Esses módulos facilitam o desenvolvimento de aplicações com o ESP32 e aceleram o desenvolvimento dos projetos. Alguns módulos possuem antenas integradas e/ou conectores para antenas externas.



ESP Product Selector

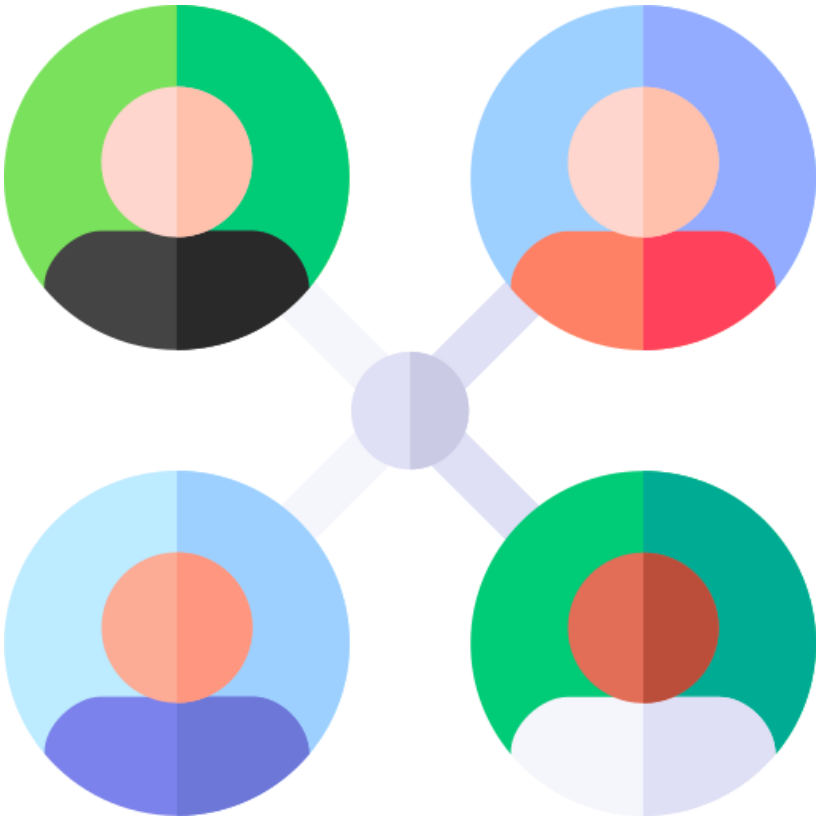
Para escolher o melhor ESP32 para seu projeto, recomendo o uso da ferramenta **ESP Product Selector**. Com ela é possível filtrar por recursos, e fazer comparações entre dispositivos. Assim é possível chegar mais facilmente ao Part Number que atenda as especificações do projeto.
Acesse em: [ESP Product Selector]

Popularidade e Comunidade

Devido à sua poderosa combinação de recursos e acessibilidade, o ESP32 ganhou uma comunidade ativa e apaixonada de desenvolvedores em todo o mundo. A Espressif Systems também fornece um extenso suporte de documentação, tutoriais e fóruns, tornando o aprendizado e o desenvolvimento com o ESP32 uma experiência colaborativa.

Com o ESP32 em suas mãos, você está pronto para embarcar em uma emocionante jornada para explorar o mundo da IoT, criar projetos inteligentes e transformar suas ideias em realidade. Nos próximos capítulos, vamos mergulhar no kit de desenvolvimento de software oficial do ESP32, o ESP-IDF, e aprender como tirar o máximo proveito dessa poderosa plataforma.

Participe da Comunidade Embarcados



ESP-IDF

O ESP-IDF (Espressif IoT Development Framework) é o framework oficial de desenvolvimento de IoT da Espressif para as séries de SoCs ESP32, ESP32-S, ESP32-C e ESP32-H. Ele fornece um SDK autossuficiente para desenvolvimento usando linguagens de programação como C e C++.

O ESP-IDF está disponível gratuitamente no GitHub. A maioria dos componentes do ESP-IDF está disponível na forma de fonte sob a licença Apache 2.0. Os componentes de terceiros estão disponíveis sob uma licença permissiva compatível.

A Importância do ESP-IDF

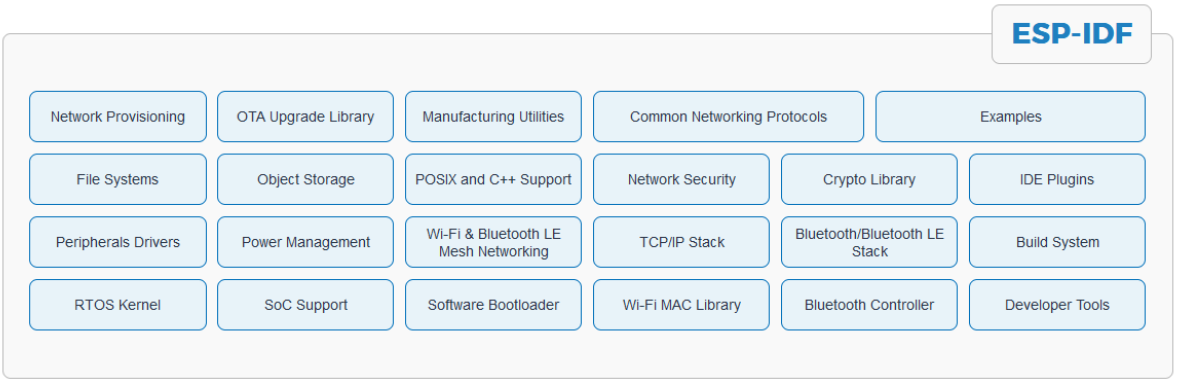
Embora o ESP32 possa ser programado com várias linguagens e ambientes, o ESP-IDF é a escolha ideal para desenvolver aplicações profissionais com recursos avançados e longo tempo de suporte e manutenção. Por ser o framework principal, todas as atualizações chegam primeiro no ESP-IDF e com ele é possível tirar o máximo proveito dos recursos e periféricos de toda a família ESP32, além de fornecer uma API bem documentada e facilidade na portabilidade de projetos entre as famílias.

Além disso, há uma política de suporte a longo prazo das versões, dando segurança no desenvolvimento de um projeto que precisa de suporte a longo prazo. O ecossistema de desenvolvimento vem com diversas ferramentas que auxiliam no desenvolvimento e para explorar recursos avançados dos ESP-32.

Com o ESP-IDF, os desenvolvedores têm acesso total aos recursos do ESP32, como WiFi, Bluetooth, GPIOs, PWM, ADC e muito mais, permitindo que criem projetos altamente personalizados e interativos. Além disso, o ESP-IDF é um projeto de código aberto com uma licença permissiva. Isso permite que as empresas possam usar livremente o ESP-IDF, e ainda toda a comunidade pode contribuir para seu desenvolvimento contínuo, aprimorando ainda mais suas funcionalidades.

Componentes do ESP-IDF

O ESP-IDF oferece suporte a um grande número de componentes de software e ferramentas, incluindo, compiladores, RTOS, drivers periféricos, pilha de rede, várias implementações de protocolo e auxiliares para casos de uso de aplicativos comuns. Esses componentes ajudam os desenvolvedores a se concentrar na lógica de negócios, enquanto o SDK fornece a maioria dos blocos de construção necessários para aplicativos típicos. Ferramentas de desenvolvedor de código aberto e disponíveis gratuitamente, bem como IDEs Eclipse e VSCode com suporte oficial, garantem facilidade de uso para os desenvolvedores.



Documentação e Suporte

O ESP-IDF vem com uma extensa documentação para seus componentes de software, não apenas no nível de uso, mas também no nível de design. Isso ajuda os desenvolvedores a entender completamente o que o ESP-IDF oferece e selecionar o que melhor se adapta às suas aplicações. O ESP-IDF contém mais de 100 exemplos que explicam o uso de seus componentes, bem como seus periféricos e recursos de hardware. Esses exemplos bem testados e bem mantidos fornecem um excelente ponto de partida para seus aplicativos.

Além disso, existem fóruns online ativos onde os membros da comunidade podem trocar experiências, tirar dúvidas e compartilhar soluções.

Compatibilidade e Versões

O ESP-IDF é constantemente atualizado e aprimorado pela equipe da Espressif Systems e pela comunidade de desenvolvedores.

Ele possui um processo de lançamento bem definido e uma política de suporte que garante que os clientes possam escolher um lançamento estável e que continuem recebendo correções importantes para seu aplicativo. Cada versão estável passa por um rigoroso processo de controle de qualidade que garante a prontidão da produção.

É importante estar atento às versões compatíveis e garantir que você esteja usando a versão adequada para sua aplicação.

Agora que conhecemos o papel fundamental do ESP-IDF no desenvolvimento para o ESP32, vamos mergulhar no processo de configuração do ambiente de desenvolvimento e começar a explorar as diversas funcionalidades oferecidas por esse poderoso kit de desenvolvimento.

Webinar gravado: Controle de motores BLDC e de indução trifásico

Webinar Gravado: Explore o potencial dos FPGAs

03/07/2023 Sistemas Digitais, Webinar, Hardware

Webinar Gravado: Projeto de hardware utilizando Conversores DC/DC

Setup do Ambiente de Desenvolvimento ESP-IDF

Antes de começarmos a explorar todo o potencial do ESP32 com o ESP-IDF, é necessário instalar e configurar corretamente o ambiente de desenvolvimento. Neste capítulo, você aprenderá o passo a passo para instalar as ferramentas necessárias e preparar o seu ambiente de trabalho.

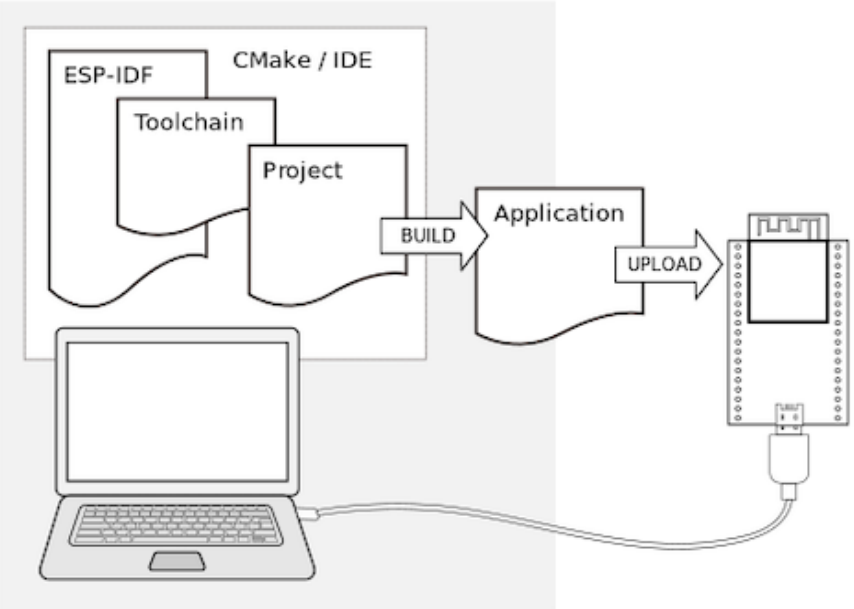
Para esse eBook vamos fazer as instalações das ferramentas necessárias e usar o ESP-IDF por linha de comandos. Não trabalharemos com VS Code ou Eclipse. Porém, após a instalação é possível configurar facilmente o ESP-IDF dentro de editores usando as extensões:

- Eclipse Plugin
- VSCode Extension

Para trabalhar com o ESP-IDF precisaremos das seguintes ferramentas:

- Toolchain - para compilar código para ESP32
- Ferramentas de compilação - CMake e Ninja para criar a aplicação completa para ESP32
- ESP-IDF que contém essencialmente API (bibliotecas de software e código-fonte) para ESP32 e scripts para operar o Toolchain

A figura a seguir dá uma visão geral do conjunto de ferramentas para desenvolver com ESP-IDF:



Instalação do ESP-IDF no Windows

O ESP-IDF requer a instalação de alguns pré-requisitos, como Python, Git, cross-compilers, ferramentas de compilação CMake e Ninja, para que você possa criar firmware para os SoCs suportados.

⚠️ Limitações:

- O caminho de instalação do ESP-IDF e do ESP-IDF Tools não deve ter mais de 90 caracteres. Caminhos de instalação muito longos podem resultar em falha na construção.
- O caminho de instalação do Python ou ESP-IDF não deve conter espaços em branco ou parênteses.
- O caminho de instalação do Python ou ESP-IDF não deve conter caracteres especiais (não ASCII), a menos que o sistema operacional esteja configurado com suporte a “Unicode UTF-8”.

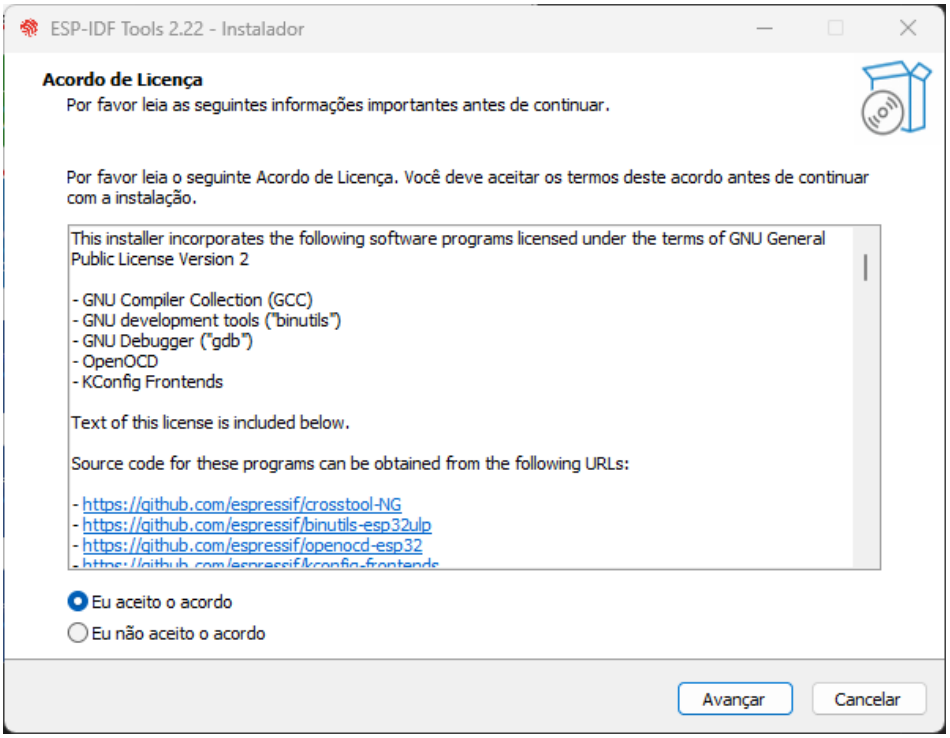
O instalador para Windows já faz a instalação dos pré-requisitos e todo o conjunto de ferramentas e softwares do ESP-IDF.

Baixe o instalador em: Windows Installer Download. Escolha o Universal Online Installer (é necessário ter acesso à internet durante a instalação):

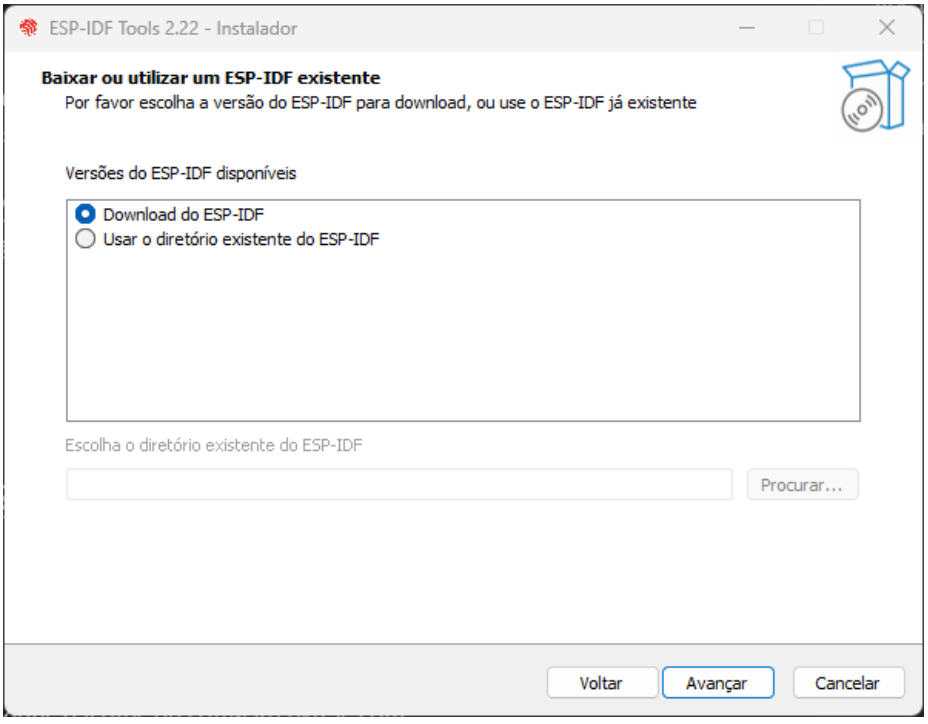
Execute o instalador.

O processo de instalação é bem simples, bastando seguir as configurações básicas. A seguir alguns pontos de atenção necessários:

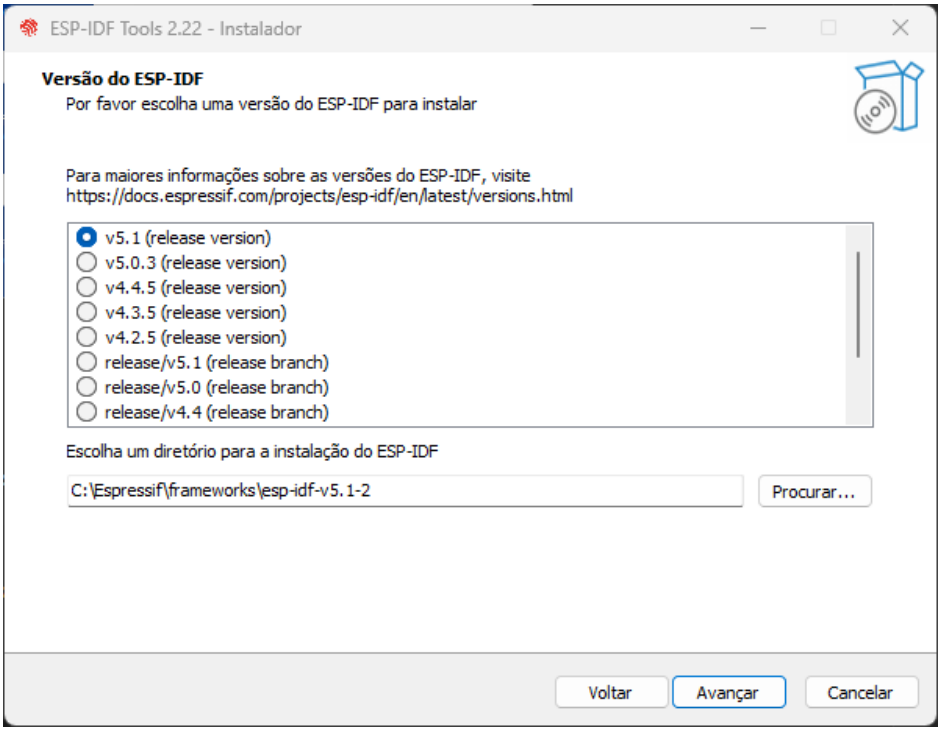
- Aceite os termos da licença para seguir com a instalação:



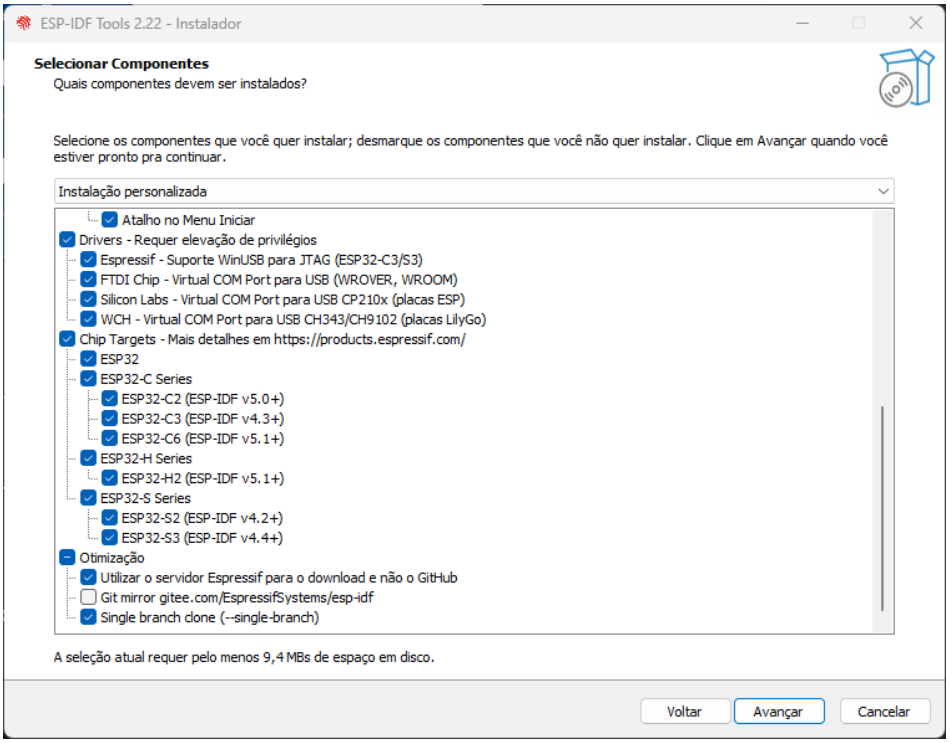
- Escolha a opção de Download do ESP-IDF



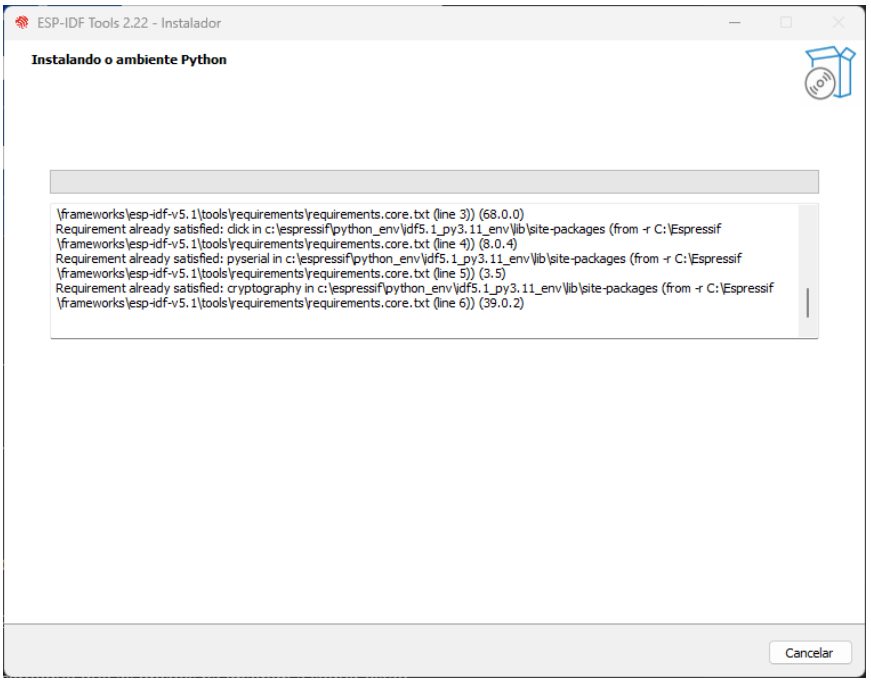
- Selecione a versão mais atual do ESP-IDF (Ou quando necessário uma versão específica):



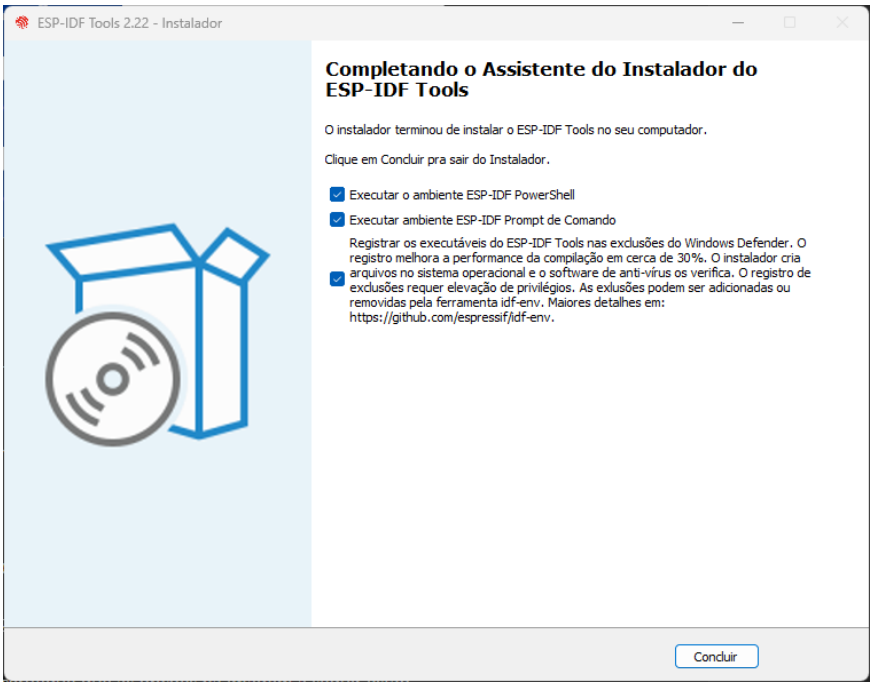
- Selecione os drivers necessários e os ESP32 que deseja trabalhar. Na dúvida selecione todos:



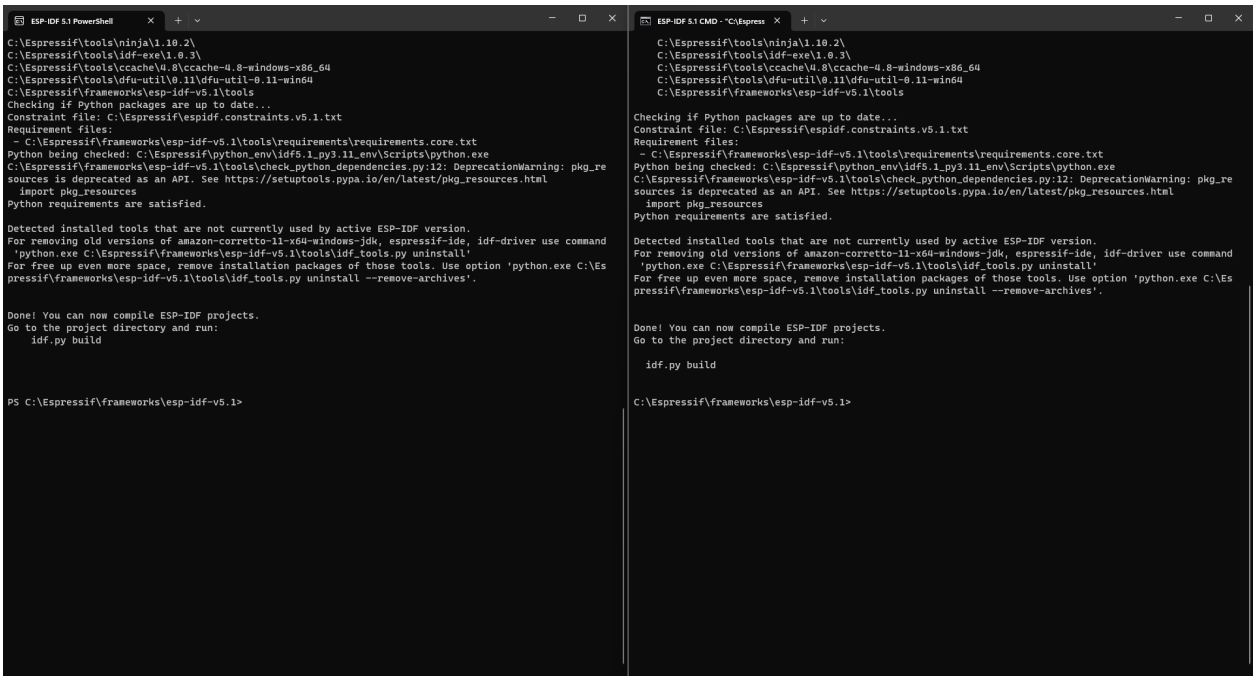
- Agora é só seguir com a instalação. Acompanhe o processo para verificar se houve erros ou alguma mensagem para permissão de instalação. Esse processo pode demorar um pouco:



- Ao final certifique que as opções na imagem a seguir estão selecionadas:

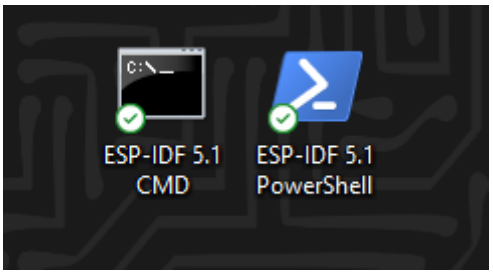


- Pronto, agora é só clicar em concluir. Será aberto o prompt de comandos e PowerShell já com o ESP-IDF. Use um dos 2 terminais para os próximos passos com o ESP-IDF.



Pronto. O ESP-IDF está instalado e pronto para seguirmos na criação de projetos. No próximo capítulo, vamos explorar os comandos e criar nosso primeiro projeto no ESP-IDF.

> **ℹ️ Sempre que for trabalhar com o ESP-IDF abra o ícone (CMD ou PowerShell) criado na área de trabalho. Ele já carrega todas as configurações necessárias para trabalhar com o ESP-IDF no Windows.**



Instalação do ESP-IDF no Linux e MacOS

Nessa seção vamos ver os passos para instalação do ESP-IDF no Linux e MacOS. Se você estiver usando o Windows pode pular para a próxima etapa.

Instalação dos pré-requisitos

Para usar o ESP-IDF com o ESP32, você precisa instalar alguns pacotes de software extras com base no seu sistema operacional.

Usuários do Linux

Para compilar usando o ESP-IDF, você precisará obter os seguintes pacotes. O comando a ser executado depende de qual distribuição do Linux você está usando:

Ubuntu e Debian:

```
Unset
sudo apt-get install git wget flex bison gperf python3 python3-venv cmake ninja-build ccache libffi-dev libssl-dev dfu-util libusb-1.0-0
```

CentOS 7 & 8:

```
Unset
sudo yum -y update && sudo yum install git wget flex bison gperf python3 cmake ninja-build ccache dfu-util libusbx
```

Arch:

```
Unset
sudo pacman -S --needed gcc git make flex bison gperf python cmake ninja ccache dfu-util libusb
```

Usuários macOS

O ESP-IDF usará a versão do Python instalada por padrão no macOS.

- Instale a compilação CMake & Ninja:

Se você tiver o HomeBrew, poderá executar:

```
Unset
brew install cmake ninja dfu-util
```

Se você tiver MacPorts, poderá executar:

```
Unset
sudo port install cmake ninja dfu-util
```

- Se preferir, consulte as páginas iniciais do CMake e Ninja para downloads de instalação do macOS.

Obtendo o ESP-IDF

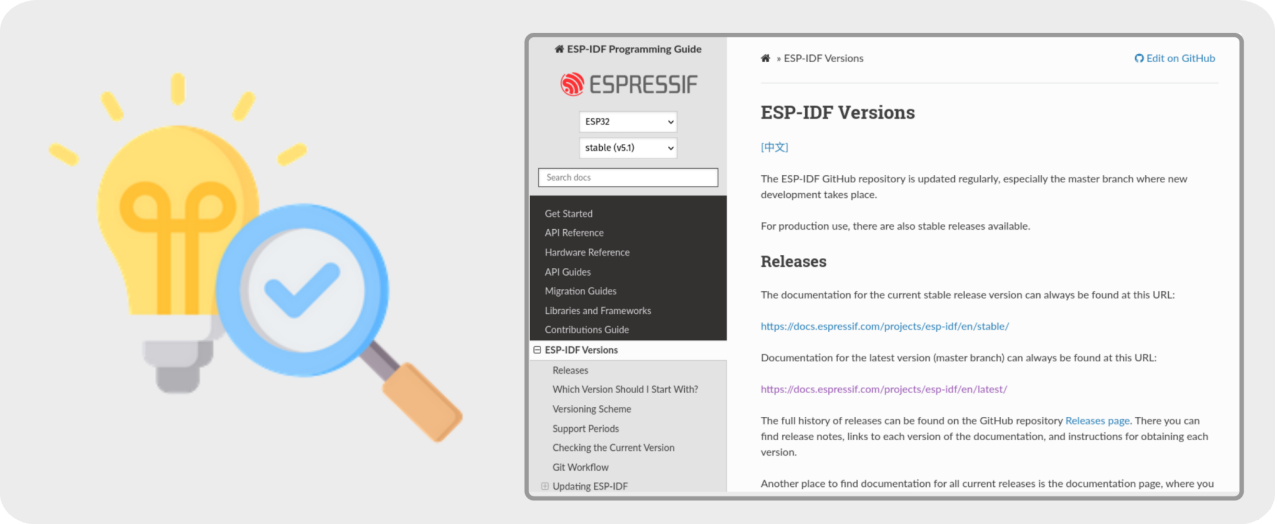
Para obter o ESP-IDF do Github, navegue até o diretório de instalação e clone o repositório com git clone, seguindo as instruções abaixo específicas para o seu sistema operacional.

Abra o Terminal e execute os seguintes comandos:

```
Unset
mkdir -p ~/esp
cd ~/esp
git clone -b v5.1 --recursive https://github.com/espressif/esp-idf.git
```

O ESP-IDF será baixado em`~/esp/esp-idf`.

Consulte as [versões do ESP-IDF] para obter informações sobre qual versão do ESP-IDF usar em uma determinada situação. Mude a versão caso necessário.



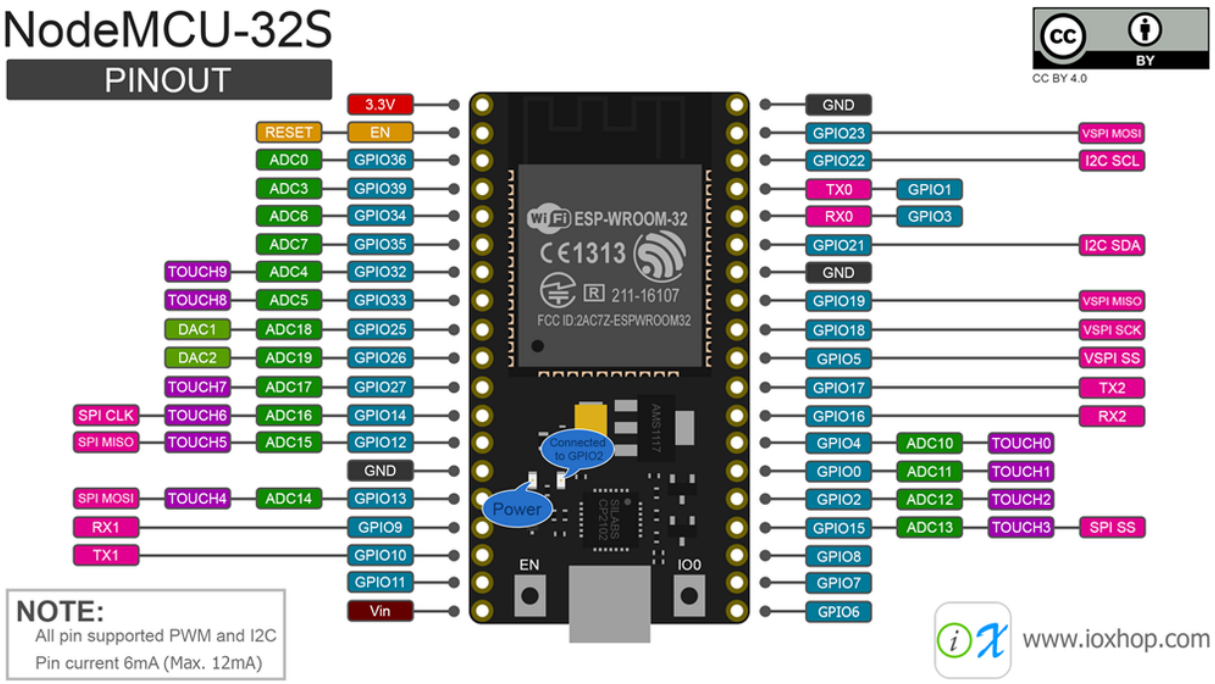
Explorando o ESP-IDF

Agora que temos o ambiente de desenvolvimento configurado, é hora de começar a explorar as capacidades do ESP32 com o ESP-IDF. Neste capítulo, você aprenderá os conceitos fundamentais e alguns dos recursos mais importantes do framework para desenvolver um projeto com o ESP-IDF.

Exemplo “Hello World”

Vamos começar com o tradicional “Hello World”. Criaremos um projeto simples para piscar um LED conectado à placa com ESP32. Isso nos permitirá verificar se tudo está configurado corretamente e iniciar nossa jornada no desenvolvimento com o ESP-IDF.

Para esse exemplo usarei uma placa NodeMCU-32S que possui um LED ligado ao pino IO2:

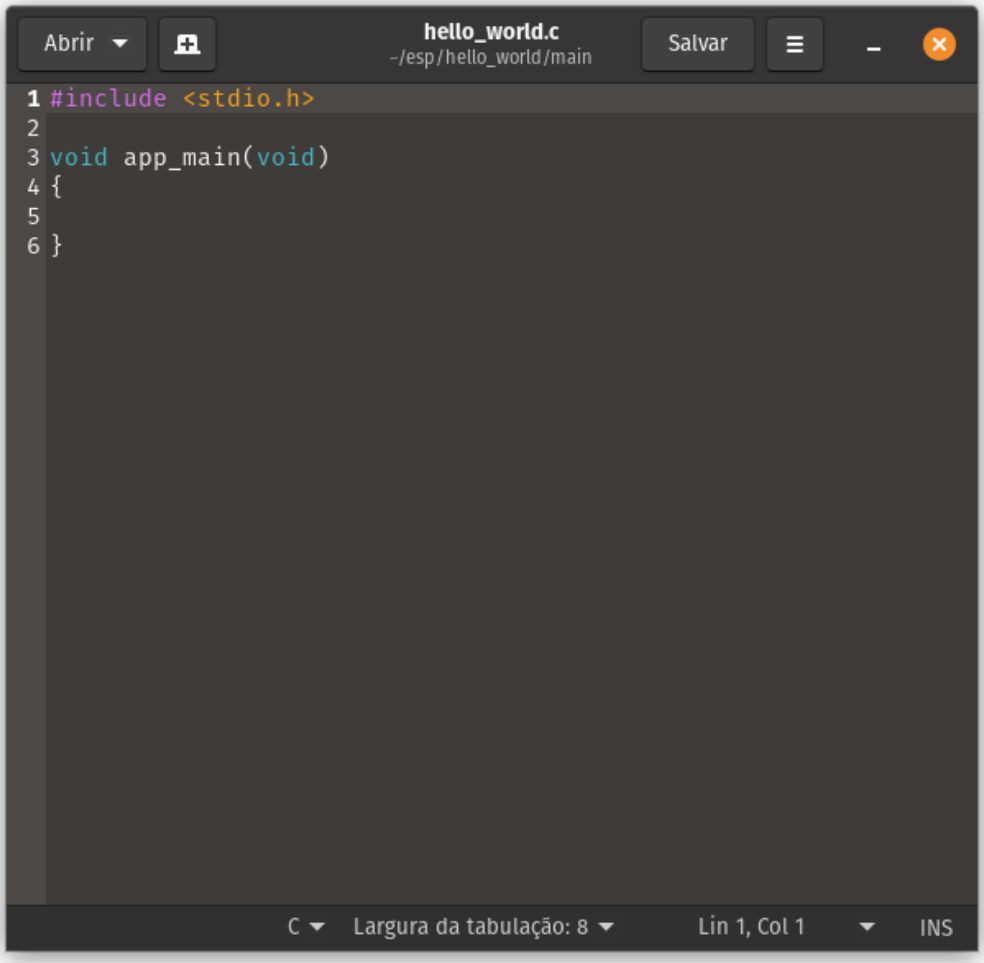


O primeiro passo é abrir o terminal ou prompt de comando e navegar para o diretório do projeto que criamos usando o comando:

```
Unset
idf.py create-project hello_world
```

Será criada toda a estrutura do projeto.

No diretório `main`, você verá um arquivo chamado `hello_world.c`. Este é o ponto de entrada do seu programa. Abra esse arquivo em um editor de texto ou IDE de sua preferência para editar o código.



Digite o código abaixo para o nosso “Hello World”:

```C/C++
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

void app_main(void)
{
  // Configuração do pino GPIO para o LED (número do pino pode variar)
  gpio_set_direction(GPIO_NUM_2, GPIO_MODE_OUTPUT);

  printf("Hello World! \n");

  while (1)
  {
    // Liga o LED
    gpio_set_level(GPIO_NUM_2, 1);
    vTaskDelay(1000 / portTICK_PERIOD_MS); // Aguarda 1 segundo
    printf("LED - ON \n");

    // Desliga o LED
    gpio_set_level(GPIO_NUM_2, 0);
    vTaskDelay(1000 / portTICK_PERIOD_MS); // Aguarda 1 segundo
    printf("LED - OFF \n");
  }
}
```

O código acima é um exemplo simples de como utilizar o ESP-IDF para controlar um LED conectado ao pino GPIO 2 (você pode precisar adaptar o número do pino para o seu hardware específico). O programa liga e desliga o LED em um intervalo de 1 segundo.

Compilando e gravando o Projeto

Antes de compilar é importante certificar-se que o target está configurado adequadamente:

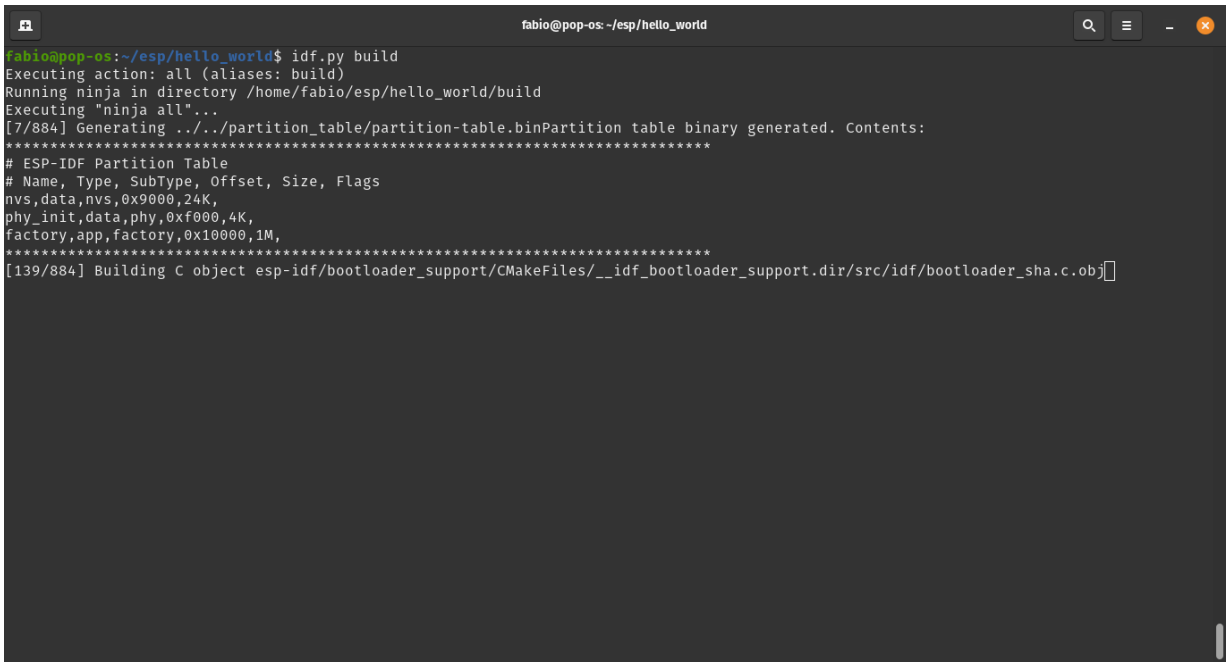
```
Unset
idf.py set-target esp32
```

Troque o target para a versão do ESP32 que estiver usando. Opções disponíveis atualmente:

```
Unset
esp32  esp32s2 esp32c3 esp32s3 esp32c2 esp32c6 esp32h2
```

Para compilar o projeto use o comando:

```
Unset
idf.py build
```



ℹ️ Aguarde a compilação. Note que a primeira compilação demora um pouco.

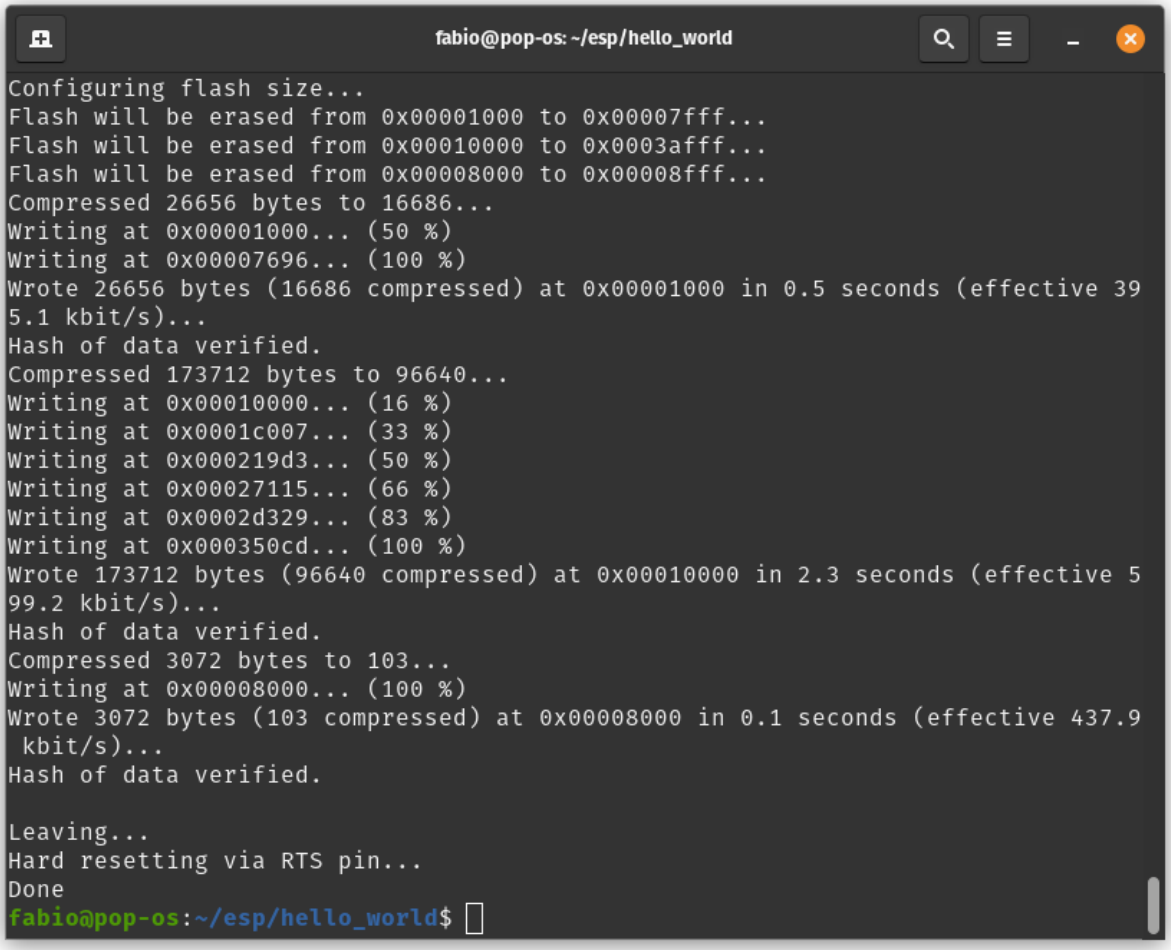
Após a compilação vamos gravar o binário na placa usando o comando flash:

```
Unset
idf.py -p (porta_serial) flash
```

Onde `(porta_serial)` é a porta serial à qual o ESP32 está conectado (por exemplo, "/dev/ttyUSB0" no Linux ou "COM3" no Windows).

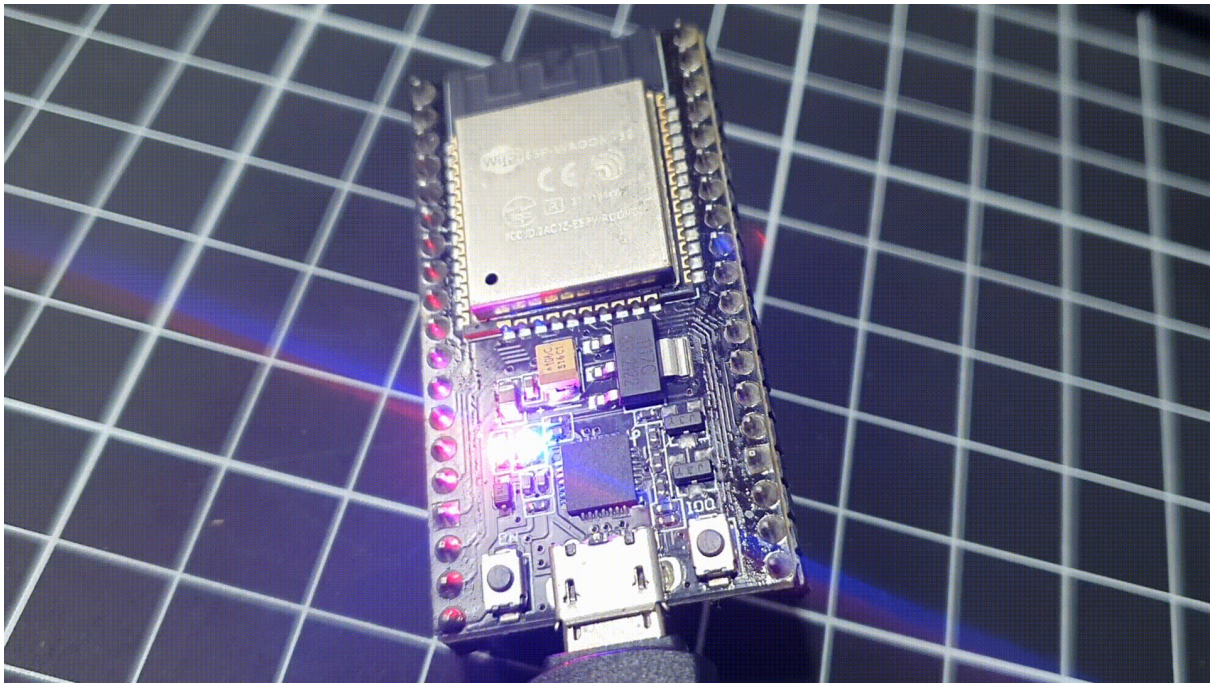
No meu caso, ficou:

```
Unset
idf.py -p /dev/ttyUSB0 flash
```



Funcionamento

Após a conclusão do flash, você deve observar o LED conectado ao ESP32 piscando alternadamente a cada segundo, indicando que o programa “Hello World” está em execução.



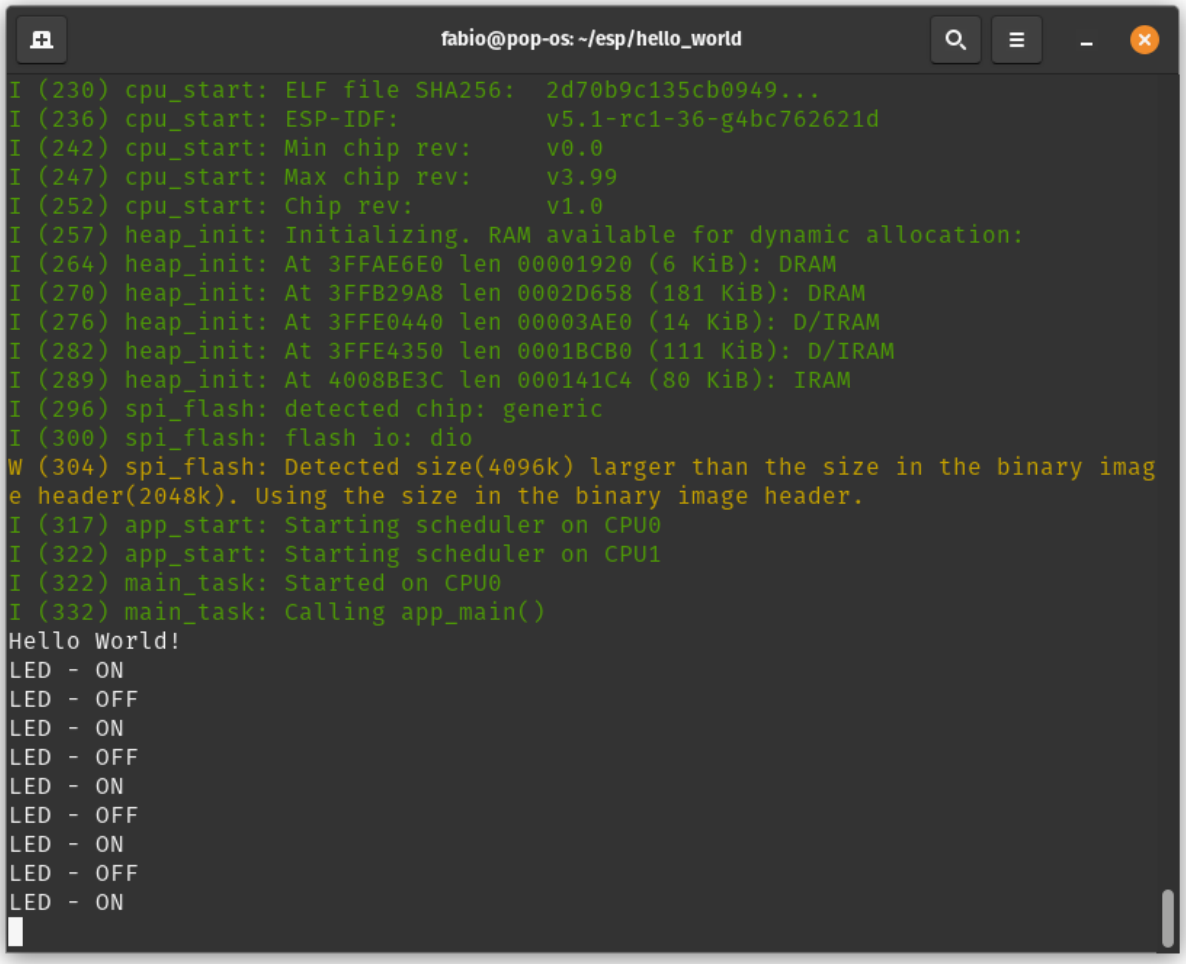
Também podemos verificar a mensagem enviada no monitor serial. Para isso execute:

```
Unset
idf.py -p (porta_serial) monitor
```

Onde `(porta_serial)` é a porta serial à qual o ESP32 está conectado (por exemplo, “/dev/ttyUSB0” no Linux ou “COM3” no Windows).

No meu caso, ficou:

```
Unset
idf.py -p /dev/ttyUSB0 monitor
```



menuconfig: Personalizando suas Configurações do Projeto

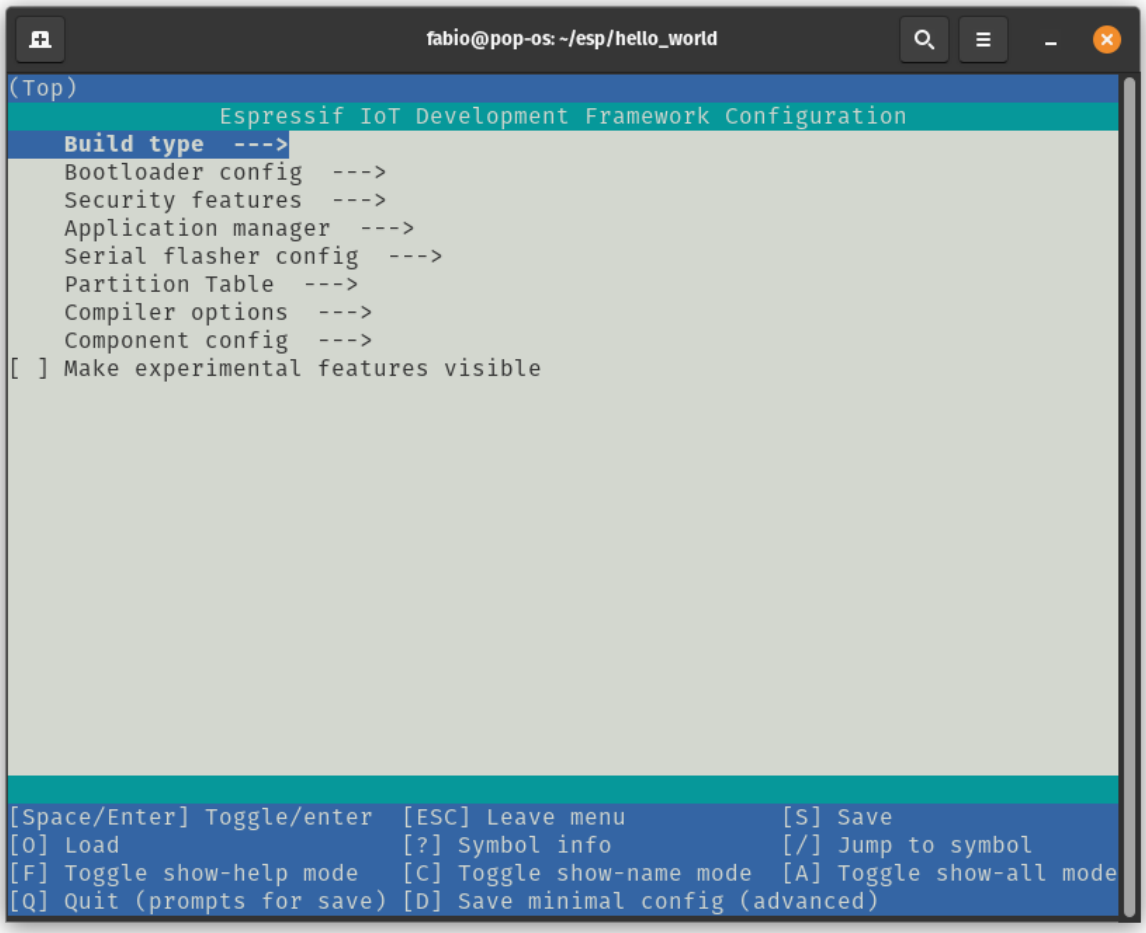
Uma das características poderosas do ESP-IDF é a capacidade de personalizar as configurações do projeto sem a necessidade de editar diretamente o código-fonte. O ESP-IDF fornece uma ferramenta de linha de comando chamada `menuconfig`, que oferece uma interface gráfica simples para configurar várias opções do projeto.

Acessando o menuconfig

Para acessar o menuconfig, certifique-se de estar no diretório raiz do seu projeto no terminal ou prompt de comando e execute o seguinte comando:

```
Unset
idf.py menuconfig
```

Isso abrirá uma janela de configuração interativa, onde você poderá navegar pelas opções e personalizar as configurações do seu projeto.



No menuconfig podemos configurar variáveis específicas do projeto, por exemplo, nome e senha da rede Wi-Fi, a velocidade do processador, etc. Nesse momento, recomendo que navegue para ver as opções disponíveis, mas não faça nenhuma alteração.

Trabalhando com GPIO

Os GPIO (General Purpose Input/Output) permitem interagir com o mundo externo para acionar ou ler dispositivos com sinais digitais (0 ou 1). Por exemplo, podemos ligar/desligar: LEDs, relés, motores, etc. Ou fazer a leitura do estado de botões, sensor de presença e muito mais. Neste capítulo, você aprenderá como configurar os GPIOs e utilizá-los.

Configuração do Pino GPIO

O ESP32 possui uma matriz de pinos GPIO que podem ser configurados como entradas ou saídas. Como saídas, você pode controlar dispositivos externos, como LEDs ou relés. Como entradas, você pode ler o estado de sensores, botões e outros dispositivos.

Para configurar um pino GPIO, você deve seguir os seguintes passos:

1 - **Definir a direção do pino GPIO:** Escolha se o pino será uma entrada ou saída. Use a função `gpio_set_direction()` para definir a direção do pino. Por exemplo:

```C/C++
// Exemplo: Configuração do pino GPIO 25 como saída
gpio_set_direction(GPIO_NUM_25, GPIO_MODE_OUTPUT);
```

```C/C++
// Exemplo: Configuração do pino GPIO 16 como entrada
gpio_set_direction(GPIO_NUM_16, GPIO_MODE_INPUT);
```

Controle de Saída (Output)

Após configurar um pino GPIO como saída, você pode controlá-lo para acionar componentes externos, como LEDs ou relés. Use as funções `gpio_set_level()` para definir o nível do pino como alto (1) ou baixo (0). Por exemplo:

```
C/C++
#include "driver/gpio.h"

// Configuração do pino GPIO 25 como saída
gpio_set_direction(GPIO_NUM_25, GPIO_MODE_OUTPUT);

// Ligar o LED conectado ao pino GPIO 25
gpio_set_level(GPIO_NUM_25, 1);

// Aguardar 1 segundo
vTaskDelay(1000 / portTICK_PERIOD_MS);

// Desligar o LED
gpio_set_level(GPIO_NUM_25, 0);
```

Leitura de Entrada (Input)

Quando um pino GPIO é configurado como entrada, você pode ler seu estado para obter informações de sensores, botões e outros dispositivos. Use a função `gpio_get_level()` para ler o estado do pino (1 para nível alto e 0 para nível baixo). Por exemplo:

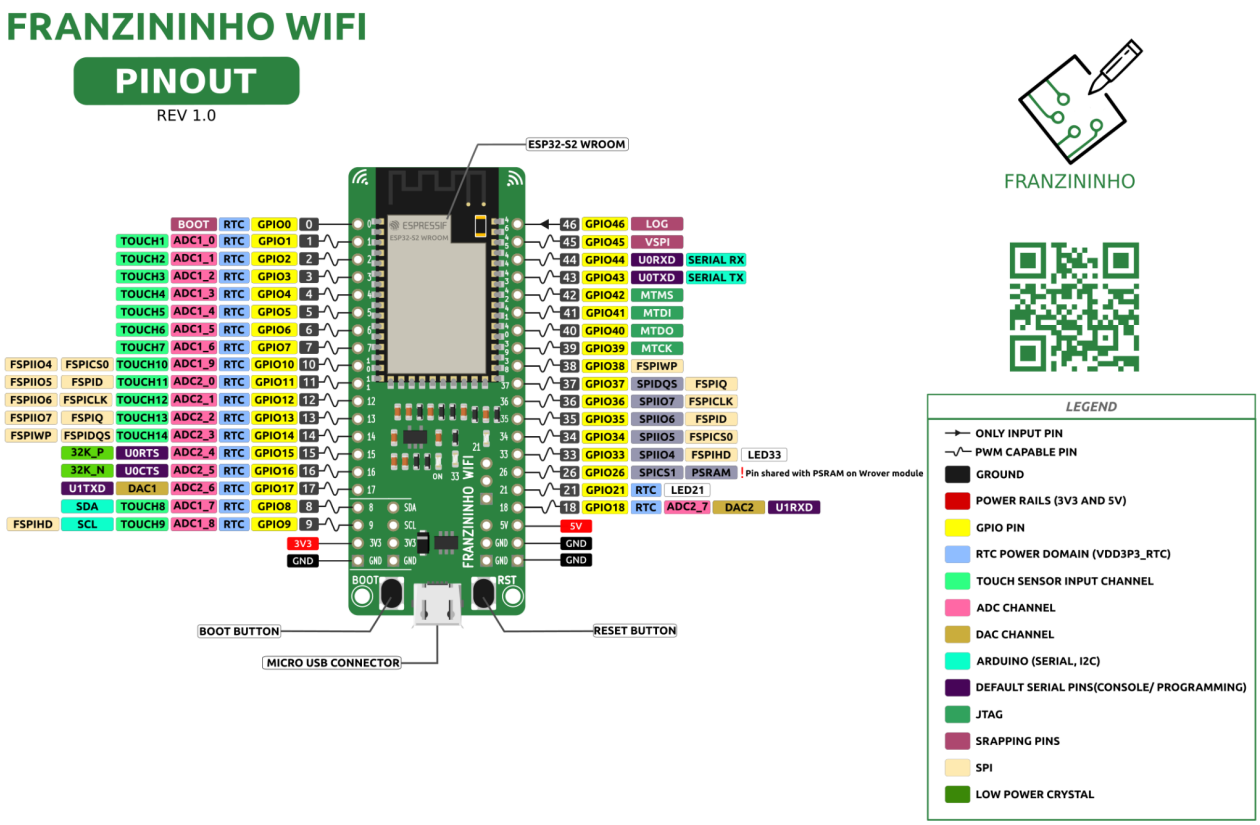
```
C/C++
#include "driver/gpio.h"

// Configuração do pino GPIO 4 como entrada
gpio_set_direction(GPIO_NUM_4, GPIO_MODE_INPUT);

// Ler o estado do pino GPIO 4
int estado_pino = gpio_get_level(GPIO_NUM_4);
```

```
// Verificar o estado
if (estado_pino == 1) {
  printf("O botão está pressionado!\n");
} else {
  printf("O botão não está pressionado.\n");
}
```

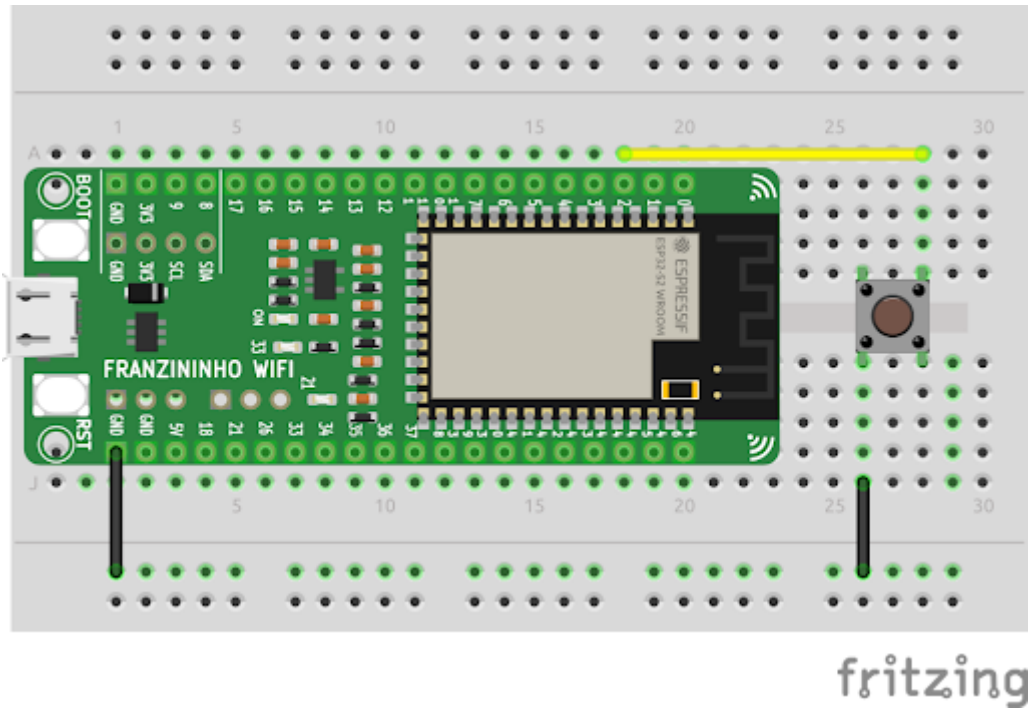
Exemplo: Leitura de botão e acionamento de um LED

Criaremos um exemplo simples que fará a leitura de um botão e inverterá o estado de um LED quando for pressionado. Esse exemplo nos auxiliará a entendermos a configuração dos GPIOs com Entrada e Saída, e também como fazer a leitura de um pino configurado como entrada e como mudar o estado de um pino configurado como saída.

Para esse exemplo usarei uma placa [Franzininho WiFi]:



Circuito:



Crie um novo projeto:

```
Unset
idf.py create-project gpio
```

Para esse exemplo usarei uma Franzininho WiFi. Ela possui o ESP32-S2. Dessa forma é necessário ajustar o target:

```
Unset
idf.py set-target esp32s2
```

Digite o seguinte código no `gpio.c`:

```
C/C++
// Inclusão de arquivos de cabeçalho
#include <stdio.h>
#include <freertos/FreeRTOS.h>
```

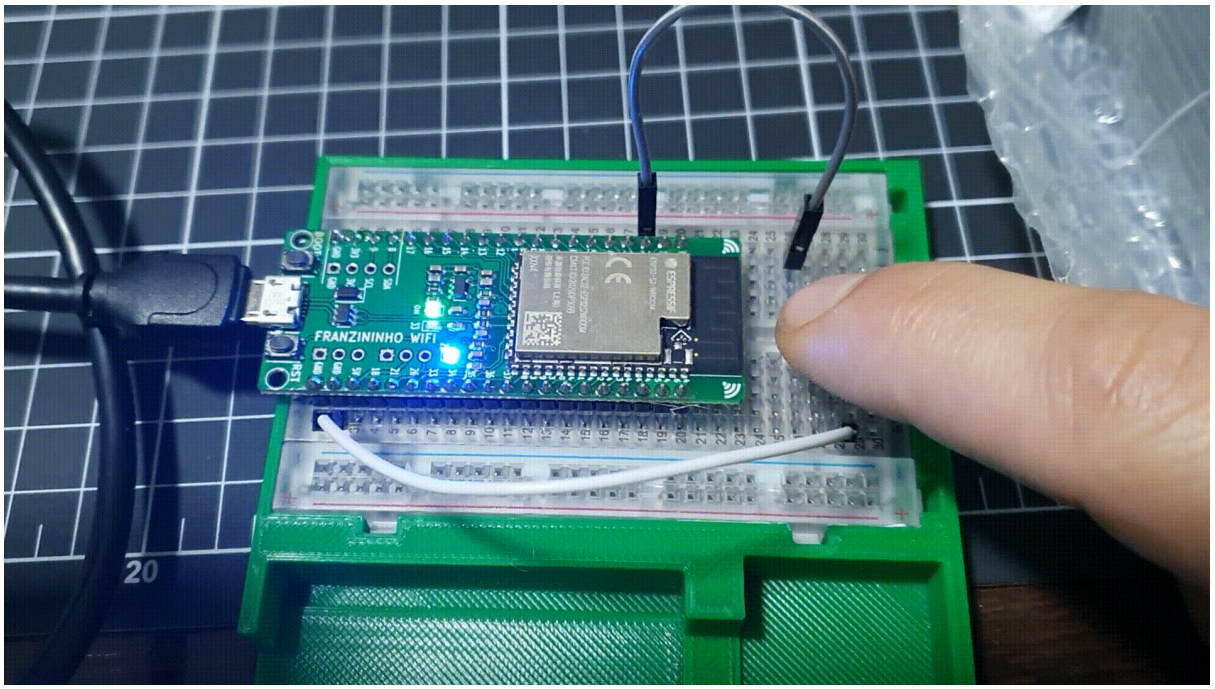
```c
#include <freertos/task.h>
#include <driver/gpio.h>

// Definições de Pinos
#define BUTTON_PIN GPIO_NUM_2
#define LED_PIN GPIO_NUM_21

void app_main() {
  gpio_set_direction(LED_PIN, GPIO_MODE_OUTPUT);   // Configura o pino do LED como saída
  gpio_set_direction(BUTTON_PIN, GPIO_MODE_INPUT);  // Configura o pino do botão como entrada
  gpio_set_pull_mode(BUTTON_PIN, GPIO_PULLUP_ONLY); // Habilita o resistor de pull-up no pino do botão

  int button_state = 1; // Inicializa o estado do botão como solto
  bool i = 0;     // Variável para alternar o estado do LED

  while (1) {
    int new_state = gpio_get_level(BUTTON_PIN);   // Lê o estado atual do botão

    if (new_state != button_state) {       // Verifica se o estado do botão mudou
      button_state = new_state;        // Atualiza o estado do botão

      if (button_state == 0) {        // Se o botão estiver pressionado
        printf("BOTÃO PRESSIONADO\n");     // Imprime uma mensagem indicando que o botão foi pressionado
        gpio_set_level(LED_PIN, i^=1);     // Inverte o estado do LED
      } else {
        printf("BOTÃO SOLTO\n");       // Imprime uma mensagem indicando que o botão foi solto
      }
    }
```

```
    vTaskDelay(pdMS_TO_TICKS(10)); // Atraso de debounce de 10 milissegundos
  }
}
```

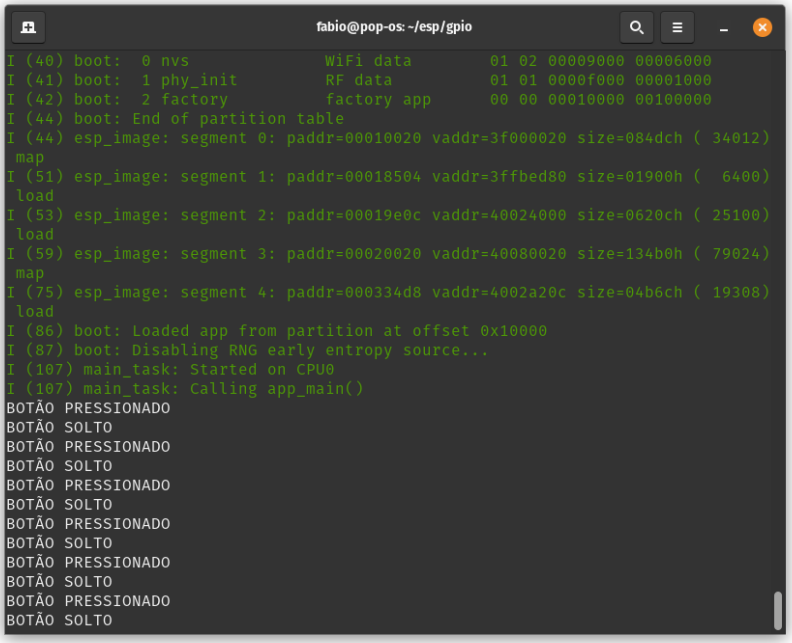
Compile o projeto, grave na placa e abra o monitor serial.

```
Unset
idf.py -p /dev/ttyACM0 build flash monitor
```

Pressione e solte o botão e verifique se o estado do LED mudará.



No monitor serial será exibido a mensagem:



Considerações Finais

Trabalhar com GPIOs é essencial para controlar dispositivos e interagir com o ambiente externo em projetos com o ESP32. As possibilidades são vastas, desde acender LEDs até capturar dados de sensores. Com o conhecimento adquirido neste capítulo, você está preparado para criar projetos mais complexos e interativos, aproveitando ao máximo a versatilidade dos GPIOs do ESP32.

PWM (Pulse Width Modulation)

O PWM (Pulse Width Modulation) é uma técnica utilizada para controlar a intensidade de um sinal digital por meio da variação do ciclo de trabalho, ou seja, a proporção entre o tempo em que o sinal está em nível alto e o tempo em que está em nível baixo. Essa técnica é especialmente útil para controlar dispositivos como LEDs, motores, servos e outros componentes que requerem uma variação contínua de energia.

Configuração do PWM no ESP32

O ESP32 possui módulos PWM embutidos que permitem gerar sinais PWM em diversos pinos GPIO. Para começar a utilizar o PWM, você precisa configurar um pino GPIO para funcionar como saída PWM. A biblioteca `driver/ledc.h` do ESP-IDF é responsável pelo controle do PWM no ESP32.

Aqui estão os passos para configurar o PWM no ESP32:

1. **Incluir a biblioteca do PWM:** No seu arquivo de código-fonte, inclua a biblioteca `driver/ledc.h`:

```
C/C++
#include "driver/ledc.h"
```

2. **Configurar um Canal PWM:** O ESP32 possui vários canais PWM disponíveis. Cada canal está associado a um conjunto de configurações, como frequência e resolução. Utilize a função `ledc_timer_config_t` para configurar o canal PWM desejado:

```
C/C++
// Configuração do canal PWM
ledc_timer_config_t timer_conf = {  //ledc timer struct

   .speed_mode   = LEDC_LOW_SPEED_MODE,  //timer mode
   .timer_num    = LEDC_TIMER_0,     //timer number
   .freq_hz      = 1000,         //frequency in Hz
```

```
  .duty_resolution = LEDC_TIMER_12_BIT,  //duty resolution
  .clk_cfg    = LEDC_AUTO_CLK    //clock source
};
ledc_timer_config(&timer_conf);    //apply the configuration
```

3. **Configurar o Pino GPIO para PWM:** Depois de configurar o canal PWM, é necessário associá-lo a um pino GPIO específico usando a função `ledc_channel_config_t`:

```
C/C++
// Configuração do pino GPIO para PWM
ledc_channel_config_t channel_conf = {    //ledc channel struct
    .speed_mode = LEDC_LOW_SPEED_MODE,   //speed mode
    .channel  = LEDC_CHANNEL_0,     //channel number
    .timer_sel = LEDC_TIMER_0,      //select timer
    .intr_type = LEDC_INTR_DISABLE,    //interrupt disabled
    .gpio_num  = GPIO_NUM_7,      //GPIO number
    .duty   = 0,          //duty cycle
};
ledc_channel_config(&channel_conf);    //apply the configuration
```

Controle de Intensidade com PWM

Uma vez configurado o PWM, você pode controlar a intensidade do dispositivo conectado ao pino GPIO. Use a função `ledc_set_duty()` para definir o ciclo de trabalho (duty cycle) do PWM:

```
C/C++
// Definir o ciclo de trabalho para 50% (meia intensidade)
ledc_set_duty(LEDC_HIGH_SPEED_MODE, LEDC_CHANNEL_0, 512);

// Atualizar o PWM
ledc_update_duty(LEDC_HIGH_SPEED_MODE, LEDC_CHANNEL_0);
```

O valor do ciclo de trabalho varia de 0 (0% de intensidade) a 1023 (100% de intensidade). Com o uso do PWM, você pode controlar a intensidade de dispositivos como LEDs ou a velocidade de motores e servos, criando efeitos interessantes em seus projetos com o ESP32.

Transição Suave com `ledc_set_fade_time_and_start`

Além do controle básico de intensidade, o ESP-IDF também oferece uma função chamada `ledc_set_fade_time_and_start()`, que permite criar transições suaves entre diferentes níveis de intensidade. Isso é especialmente útil para criar efeitos de fade-in e fade-out suaves em LEDs, por exemplo.

Aqui está um exemplo de como usar essa função:

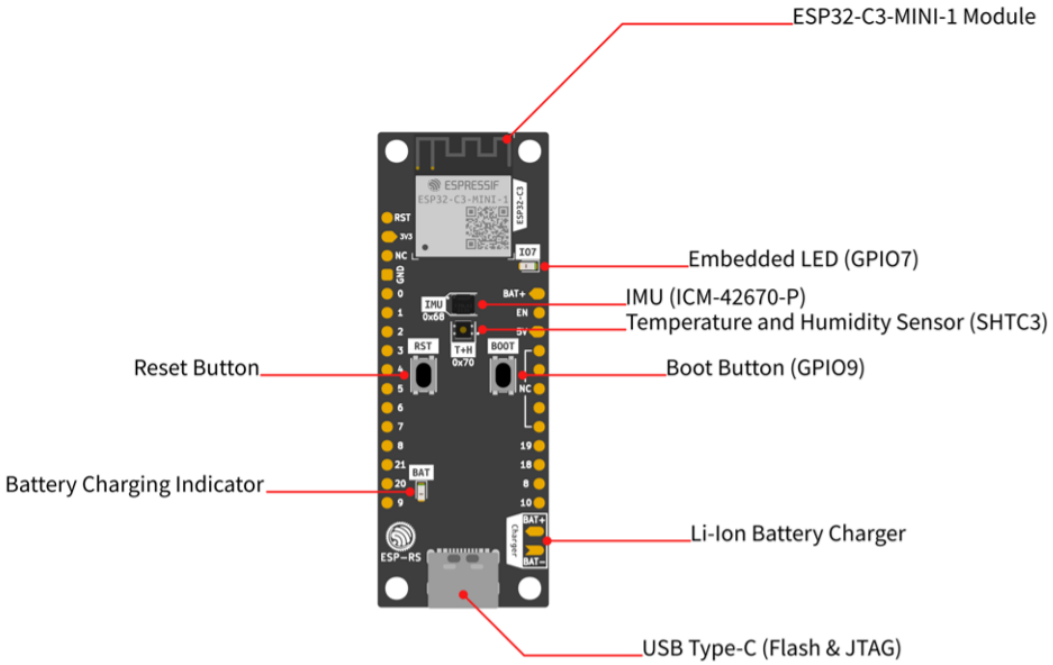
```
C/C++
// Definir o tempo de transição para 2 segundos (2000 ms)
ledc_set_fade_time_and_start(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0,4096, 1000,
2000);
```

Neste exemplo, o canal PWM 0 transitará suavemente de seu valor atual para 1023 (100% de intensidade) ao longo de um período de 2 segundos. Você pode ajustar o valor de destino e o tempo de transição de acordo com suas necessidades.

Exemplo: Efeito Fade em um LED usando PWM

Neste exemplo, você aprenderá como criar um efeito de fade suave em um LED usando a Modulação por Largura de Pulso (PWM) no ESP32. O efeito de fade suave é obtido gradualmente alterando o brilho do LED, criando um efeito visual agradável.

Para esse exemplo usarei a placa ESP-RS que possui o ESP32-C3 e um LED ligado ao pino IO7. Você pode usar a placa de sua preferência.



⚠️ **Se você for usar outra placa, conecte um LED ao pino GPIO7 (ou outro pino de sua escolha). Certifique-se de ter um resistor de limitação de corrente em série com o LED para evitar danos.**

Crie um novo projeto:

```
Unset
idf.py create-project fade_led
```

Para esse exemplo usarei uma placa com ESP32-C3. Dessa forma é necessário ajustar o target:

```
Unset
idf.py set-target esp32c3
```

Digite o seguinte código no `fade_led.c`:

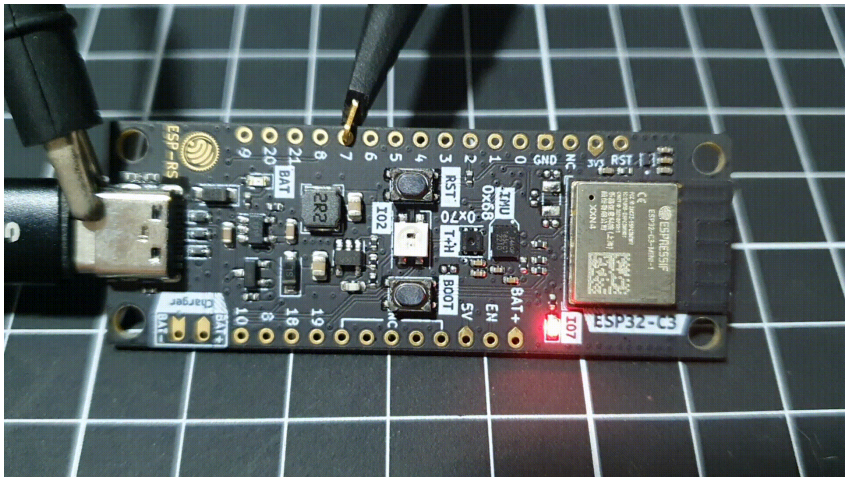
```C/C++
#include <stdio.h>
//freeRTOS includes
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
//ESP32 includes
#include "driver/ledc.h"


void app_main(void) // Main
{
 ledc_timer_config_t timer_conf = {  //ledc timer struct

   .speed_mode   = LEDC_LOW_SPEED_MODE, //timer mode
   .timer_num    = LEDC_TIMER_0,    //timer number
   .freq_hz    = 1000,       //frequency in Hz
   .duty_resolution = LEDC_TIMER_12_BIT,  //duty resolution
   .clk_cfg    = LEDC_AUTO_CLK    //clock source
 };
 ledc_timer_config(&timer_conf);    //apply the configuration


 ledc_channel_config_t channel_conf ={    //ledc channel struct
    .speed_mode = LEDC_LOW_SPEED_MODE,   //speed mode
    .channel  = LEDC_CHANNEL_0,     //channel number
    .timer_sel = LEDC_TIMER_0,     //select timer
    .intr_type = LEDC_INTR_DISABLE,    //interrupt disabled
    .gpio_num  = GPIO_NUM_7,      //GPIO number
    .duty   = 0,          //duty cycle
 };
 ledc_channel_config(&channel_conf);    //apply the configuration

 //iniatilize fade service
```

```
 ledc_fade_func_install(0);       //install the fade function

 while(1)
 {  //fade routine
    ledc_set_fade_time_and_start(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0,4096, 1000,
LEDC_FADE_WAIT_DONE); //set fade time and start fade
    vTaskDelay(1000 / portTICK_PERIOD_MS);                          //delay 1 second

ledc_set_fade_time_and_start(LEDC_LOW_SPEED_MODE,LEDC_CHANNEL_0,0,1000,LEDC_FADE_
WAIT_DONE);   //set fade time and start fade
    vTaskDelay(1000 / portTICK_PERIOD_MS);                          //delay 1 second
 }

}
```

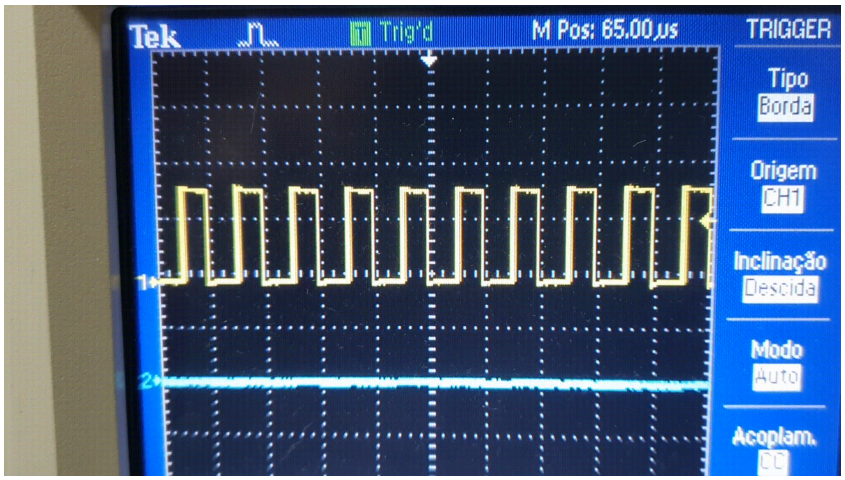
Compile o projeto e grave na placa:

```
Unset
idf.py -p /dev/ttyACM0 build flash
```

Após a gravação, o LED começará a exibir o efeito de fade suave.



Se você tiver acesso a um osciloscópio, poderá ver a variação do sinal PWM:



Considerações Finais

Trabalhar com PWM é essencial em aplicações de sistemas embarcados. Podemos controlar como LEDs, motores, servos e outros componentes que requerem uma variação de tensão.

Com o conhecimento adquirido neste capítulo, você está preparado para usar o PWM em projetos com ESP32.

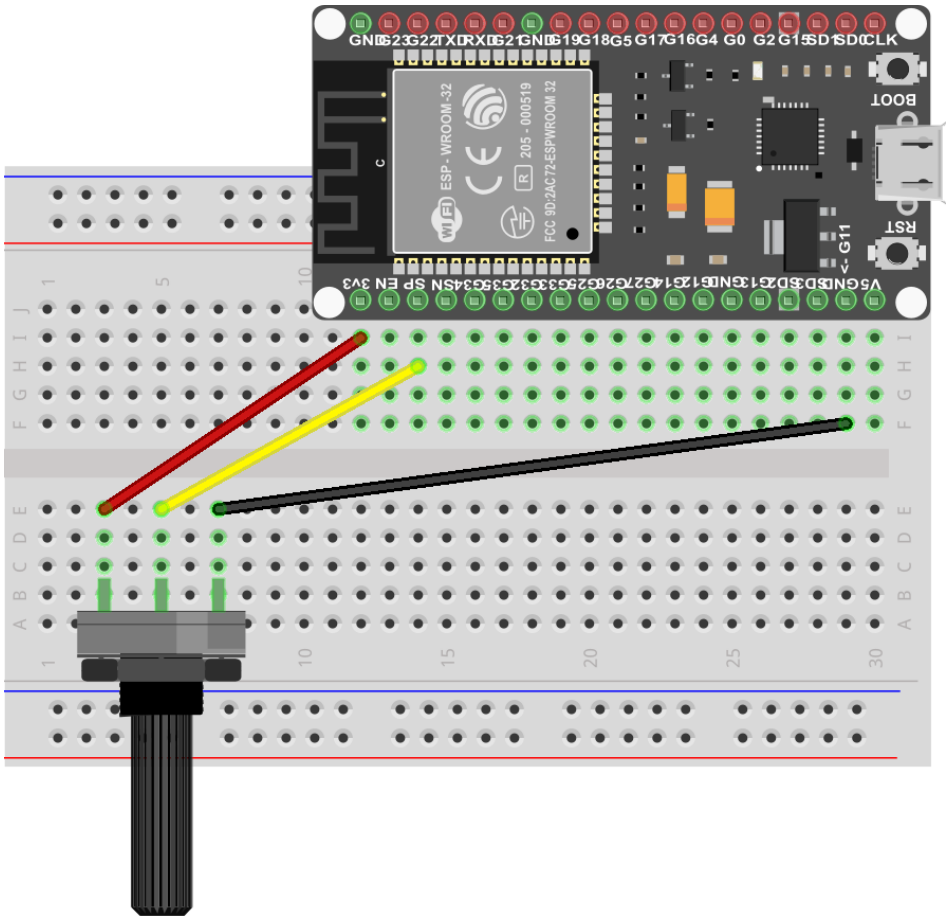
ADC (Analog to Digital Converter)

O ADC (Analog to Digital Converter) permite converter sinais analógicos em sinais digitais. Essa funcionalidade é especialmente útil quando você precisa medir valores analógicos, como leituras de sensores de temperatura, luz, umidade, entre outros.

Explorando o ADC no ESP32

O ESP32 possui vários canais ADC disponíveis, o que permite a leitura de múltiplas fontes de sinal analógico. Para utilizar o ADC, você precisa configurar o pino GPIO correto para funcionar como entrada analógica.

Para entendermos o funcionamento analisaremos um exemplo de leitura do canal 0 do ADC1 do ESP32. Para esse exemplo usaremos uma placa com ESP32 e um potenciômetro ligado ao canal 0(IO36).



Crie um novo projeto:

```
Unset
idf.py create-project adc_read
```

Digite o código abaixo no arquivo `adc_read.c`:

```
C/C++
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "soc/soc_caps.h"
#include "esp_log.h"
#include "esp_adc/adc_oneshot.h"
#include "esp_adc/adc_cali.h"
#include "esp_adc/adc_cali_scheme.h"

const static char *TAG = "EXAMPLE";

static int adc_raw[2][10];
static int voltage[2][10];
static bool example_adc_calibration_init(adc_unit_t unit, adc_channel_t channel,
adc_atten_t atten, adc_cali_handle_t *out_handle);
static void example_adc_calibration_deinit(adc_cali_handle_t handle);

void app_main(void)
{
  //-------------ADC1 Init---------------//
  adc_oneshot_unit_handle_t adc1_handle;
  adc_oneshot_unit_init_cfg_t init_config1 = {
    .unit_id = ADC_UNIT_1,
  };
```

```
  ESP_ERROR_CHECK(adc_oneshot_new_unit(&init_config1, &adc1_handle));

  //-------------ADC1 Config---------------//
  adc_oneshot_chan_cfg_t config = {
    .bitwidth = ADC_BITWIDTH_DEFAULT,
    .atten = ADC_ATTEN_DB_11,
  };
  ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle, ADC_CHANNEL_0,
&config));

  //-------------ADC1 Calibration Init---------------//
  adc_cali_handle_t adc1_cali_chan0_handle = NULL;
  bool do_calibration1_chan0 = example_adc_calibration_init(ADC_UNIT_1,
ADC_CHANNEL_0, ADC_ATTEN_DB_11, &adc1_cali_chan0_handle);

  while (1)
  {
    ESP_ERROR_CHECK(adc_oneshot_read(adc1_handle, ADC_CHANNEL_0, &adc_raw[0][0]));
    ESP_LOGI(TAG, "ADC%d Channel[%d] Raw Data: %d", ADC_UNIT_1 + 1, ADC_CHANNEL_0,
adc_raw[0][0]);
    if (do_calibration1_chan0)
    {
      ESP_ERROR_CHECK(adc_cali_raw_to_voltage(adc1_cali_chan0_handle,
adc_raw[0][0], &voltage[0][0]));
      ESP_LOGI(TAG, "ADC%d Channel[%d] Cali Voltage: %d mV", ADC_UNIT_1 + 1,
ADC_CHANNEL_0, voltage[0][0]);
    }

    vTaskDelay(pdMS_TO_TICKS(1000));
  }

  // Tear Down
  ESP_ERROR_CHECK(adc_oneshot_del_unit(adc1_handle));
  if (do_calibration1_chan0)
```

```
  {
    example_adc_calibration_deinit(adc1_cali_chan0_handle);
  }
}

/*---------------------------------------------------------------
    ADC Calibration
---------------------------------------------------------------*/
static bool example_adc_calibration_init(adc_unit_t unit, adc_channel_t channel,
adc_atten_t atten, adc_cali_handle_t *out_handle)
{
  adc_cali_handle_t handle = NULL;
  esp_err_t ret = ESP_FAIL;
  bool calibrated = false;

#if ADC_CALI_SCHEME_CURVE_FITTING_SUPPORTED
  if (!calibrated)
  {
    ESP_LOGI(TAG, "calibration scheme version is %s", "Curve Fitting");
    adc_cali_curve_fitting_config_t cali_config = {
      .unit_id = unit,
      .chan = channel,
      .atten = atten,
      .bitwidth = ADC_BITWIDTH_DEFAULT,
    };
    ret = adc_cali_create_scheme_curve_fitting(&cali_config, &handle);
    if (ret == ESP_OK)
    {
      calibrated = true;
    }
  }
#endif
```

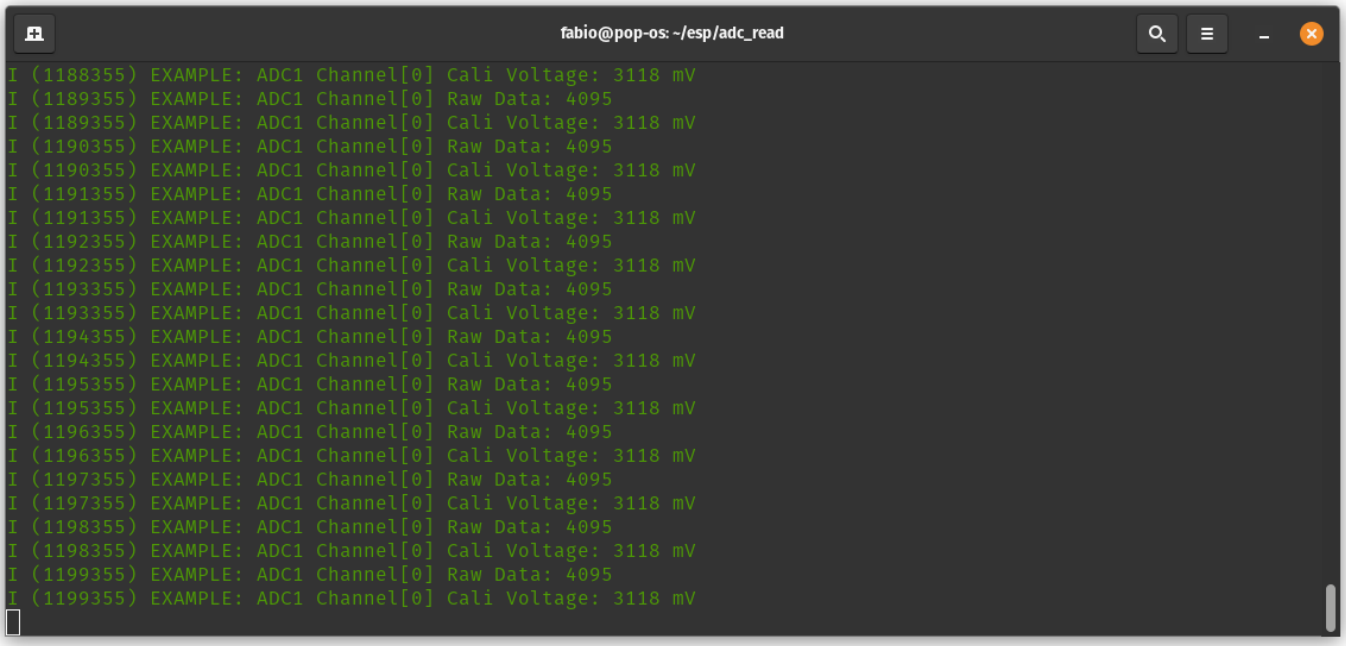
```
#if ADC_CALI_SCHEME_LINE_FITTING_SUPPORTED
  if (!calibrated)
  {
    ESP_LOGI(TAG, "calibration scheme version is %s", "Line Fitting");
    adc_cali_line_fitting_config_t cali_config = {
      .unit_id = unit,
      .atten = atten,
      .bitwidth = ADC_BITWIDTH_DEFAULT,
    };
    ret = adc_cali_create_scheme_line_fitting(&cali_config, &handle);
    if (ret == ESP_OK)
    {
      calibrated = true;
    }
  }
#endif

  *out_handle = handle;
  if (ret == ESP_OK)
  {
    ESP_LOGI(TAG, "Calibration Success");
  }
  else if (ret == ESP_ERR_NOT_SUPPORTED || !calibrated)
  {
    ESP_LOGW(TAG, "eFuse not burnt, skip software calibration");
  }
  else
  {
    ESP_LOGE(TAG, "Invalid arg or no memory");
  }

  return calibrated;
}
```

```
static void example_adc_calibration_deinit(adc_cali_handle_t handle)
{
#if ADC_CALI_SCHEME_CURVE_FITTING_SUPPORTED
  ESP_LOGI(TAG, "deregister %s calibration scheme", "Curve Fitting");
  ESP_ERROR_CHECK(adc_cali_delete_scheme_curve_fitting(handle));

#elif ADC_CALI_SCHEME_LINE_FITTING_SUPPORTED
  ESP_LOGI(TAG, "deregister %s calibration scheme", "Line Fitting");
  ESP_ERROR_CHECK(adc_cali_delete_scheme_line_fitting(handle));
#endif
}
```

Compile o código, grave na placa e abra o monitor serial:

```
Unset
idf.py -p /dev/ttyUSB0 build flash monitor
```

Varie o potenciômetro e veja os valores impressos no monitor serial:



Entendendo a configuração e uso do ADC

Inicialização e Configuração do ADC:

Para esse exemplo foi usado o ADC1 e o canal 0. Inicialmente é feita a configuração do ADC e canal. A configuração inclui a largura de bits do ADC e a atenuação do sinal (que controla a faixa de tensão que o ADC pode medir):

```
C/C++
//-------------ADC1 Init---------------//
  adc_oneshot_unit_handle_t adc1_handle;
  adc_oneshot_unit_init_cfg_t init_config1 = {
    .unit_id = ADC_UNIT_1,
  };
  ESP_ERROR_CHECK(adc_oneshot_new_unit(&init_config1, &adc1_handle));

  //-------------ADC1 Config---------------//
  adc_oneshot_chan_cfg_t config = {
    .bitwidth = ADC_BITWIDTH_DEFAULT,
    .atten = ADC_ATTEN_DB_11,
  };
  ESP_ERROR_CHECK(adc_oneshot_config_channel(adc1_handle, ADC_CHANNEL_0,
&config));
```

Calibração do ADC:

O ESP-IDF disponibiliza uma rotina para calibração do ADC com base na tensão de referência lida em fábrica e gravada no eFuses. Existem duas formas de calibração (que dependerá do modelo de ESP32 usado): ajuste de curva (`adc_cali_curve_fitting_config_t`) e ajuste linear (`adc_cali_line_fitting_config_t`). Se a calibração for bem-sucedida, a variável `do_calibration1_chan0` será definida como `true`.

```
C/C++
//-------------ADC1 Calibration Init---------------//
```

```
adc_cali_handle_t adc1_cali_chan0_handle = NULL;
bool do_calibration1_chan0 = example_adc_calibration_init(ADC_UNIT_1,
ADC_CHANNEL_0, ADC_ATTEN_DB_11, &adc1_cali_chan0_handle);
```

C/C++

```
/*---------------------------------------------------------------
    ADC Calibration
---------------------------------------------------------------*/
static bool example_adc_calibration_init(adc_unit_t unit, adc_channel_t channel,
adc_atten_t atten, adc_cali_handle_t *out_handle)
{
  adc_cali_handle_t handle = NULL;
  esp_err_t ret = ESP_FAIL;
  bool calibrated = false;

#if ADC_CALI_SCHEME_CURVE_FITTING_SUPPORTED
  if (!calibrated)
  {
    ESP_LOGI(TAG, "calibration scheme version is %s", "Curve Fitting");
    adc_cali_curve_fitting_config_t cali_config = {
      .unit_id = unit,
      .chan = channel,
      .atten = atten,
      .bitwidth = ADC_BITWIDTH_DEFAULT,
    };
    ret = adc_cali_create_scheme_curve_fitting(&cali_config, &handle);
    if (ret == ESP_OK)
    {
      calibrated = true;
    }
  }
#endif
```

```
#if ADC_CALI_SCHEME_LINE_FITTING_SUPPORTED
  if (!calibrated)
  {
    ESP_LOGI(TAG, "calibration scheme version is %s", "Line Fitting");
    adc_cali_line_fitting_config_t cali_config = {
      .unit_id = unit,
      .atten = atten,
      .bitwidth = ADC_BITWIDTH_DEFAULT,
    };
    ret = adc_cali_create_scheme_line_fitting(&cali_config, &handle);
    if (ret == ESP_OK)
    {
      calibrated = true;
    }
  }
#endif

  *out_handle = handle;
  if (ret == ESP_OK)
  {
    ESP_LOGI(TAG, "Calibration Success");
  }
  else if (ret == ESP_ERR_NOT_SUPPORTED || !calibrated)
  {
    ESP_LOGW(TAG, "eFuse not burnt, skip software calibration");
  }
  else
  {
    ESP_LOGE(TAG, "Invalid arg or no memory");
  }

  return calibrated;
```

```
}
```

Loop de Leitura do ADC e calculo da tensão com base na calibração:

No loop principal é feita a leitura do bruto presente no canal especificado para o ADC1. Se a calibração estiver ativada (`do_calibration1_chan0` é `true`), o valor bruto também é convertido para tensão calibrada. Os valores são enviados para o console serial:

```
C/C++
while (1)
  {
    ESP_ERROR_CHECK(adc_oneshot_read(adc1_handle, ADC_CHANNEL_0, &adc_raw[0][0]));
    ESP_LOGI(TAG, "ADC%d Channel[%d] Raw Data: %d", ADC_UNIT_1 + 1, ADC_CHANNEL_0,
adc_raw[0][0]);
    if (do_calibration1_chan0)
    {
      ESP_ERROR_CHECK(adc_cali_raw_to_voltage(adc1_cali_chan0_handle,
adc_raw[0][0], &voltage[0][0]));
      ESP_LOGI(TAG, "ADC%d Channel[%d] Cali Voltage: %d mV", ADC_UNIT_1 + 1,
ADC_CHANNEL_0, voltage[0][0]);
    }

    vTaskDelay(pdMS_TO_TICKS(1000));
  }
```

Em resumo, este exemplo demonstra como configurar e utilizar o ADC do ESP32 com o ESP-IDF para medir sinais analógicos e opcionalmente realizar a calibração do ADC para obter resultados mais precisos. A calibração é importante para minimizar erros e garantir que as leituras do ADC sejam confiáveis em diferentes cenários.

Recomendo que troque de canais e ADC, assim como as configurações disponíveis para verificar as possibilidades de uso do ADC do ESP32.

Considerações Finais

O ADC é uma funcionalidade importante para aquisição de dados analógicos no ESP32. Com o uso do ADC, você pode medir valores de sensores analógicos e processar esses dados digitalmente em seus projetos.

Lembre-se de que os sinais analógicos podem ser sensíveis a ruídos e interferências, então é importante projetar circuitos apropriados e realizar um bom tratamento de sinais para garantir leituras precisas e confiáveis.

WiFi no ESP32: Explorando o Potencial da Conectividade

O ESP32 oferece suporte a conectividade WiFi, permitindo que você se comunique com redes sem fio e acesse a internet. Neste tópico, vamos explorar o WiFi a partir de um exemplo prático disponível no repositório do ESP-IDF: o WiFi Scanner para detectar redes próximas.

Exemplo: WiFi Scanner

O WiFi Scanner é um exemplo que demonstra como usar o ESP32 para detectar redes WiFi disponíveis nas proximidades.

Vamos criar um projeto de WiFi Scanner copiando o exemplo Scan disponível no repositório do ESP-IDF instalado no computador.

No terminal, digite:

Linux e macOS:

```
Unset
cd ~/esp
cp -r $IDF_PATH/examples/wifi/scan .
```

Windows:

```
Unset
cd %userprofile%\esp
xcopy /e /i %IDF_PATH%\examples\wifi\scan
```

Compile o projeto, grave na placa e abra o terminal serial:

```
Unset
idf.py -p /dev/ttyUSB0 build flash monitor
```

Verifique no terminal se são exibidas as redes wifi próximas.

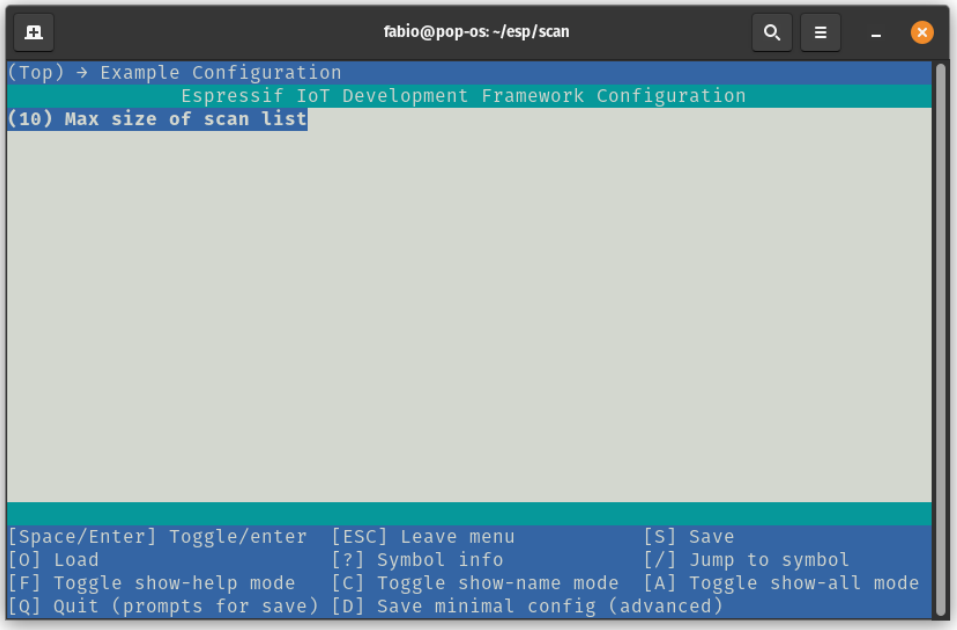
Entendendo o exemplo de WiFI Scanner:

Inclusão de Bibliotecas e Definições:

```
C/C++
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/event_groups.h"
#include "esp_wifi.h"
#include "esp_log.h"
#include "esp_event.h"
#include "nvs_flash.h"

#define DEFAULT_SCAN_LIST_SIZE CONFIG_EXAMPLE_SCAN_LIST_SIZE
```

O código começa incluindo várias bibliotecas necessárias para realizar as operações relacionadas ao Wi-Fi. Além disso, define-se uma macro `DEFAULT_SCAN_LIST_SIZE` que é usada para determinar o tamanho máximo da lista de redes Wi-Fi a serem escaneadas. Esse parâmetro pode ser configurado no menuconfig:

```
Unset
idf.py menuconfig
```



Funções de Impressão do modo de autenticação e criptografia das redes:

```C/C++
static void print_auth_mode(int authmode)
{
  switch (authmode) {
  case WIFI_AUTH_OPEN:
    ESP_LOGI(TAG, "Authmode \tWIFI_AUTH_OPEN");
    break;
  case WIFI_AUTH_OWE:
    ESP_LOGI(TAG, "Authmode \tWIFI_AUTH_OWE");
    break;
  case WIFI_AUTH_WEP:
    ESP_LOGI(TAG, "Authmode \tWIFI_AUTH_WEP");
    break;
  case WIFI_AUTH_WPA_PSK:
    ESP_LOGI(TAG, "Authmode \tWIFI_AUTH_WPA_PSK");
    break;
  case WIFI_AUTH_WPA2_PSK:
    ESP_LOGI(TAG, "Authmode \tWIFI_AUTH_WPA2_PSK");
    break;
  case WIFI_AUTH_WPA_WPA2_PSK:
    ESP_LOGI(TAG, "Authmode \tWIFI_AUTH_WPA_WPA2_PSK");
    break;
  case WIFI_AUTH_WPA2_ENTERPRISE:
    ESP_LOGI(TAG, "Authmode \tWIFI_AUTH_WPA2_ENTERPRISE");
    break;
  case WIFI_AUTH_WPA3_PSK:
    ESP_LOGI(TAG, "Authmode \tWIFI_AUTH_WPA3_PSK");
    break;
  case WIFI_AUTH_WPA2_WPA3_PSK:
    ESP_LOGI(TAG, "Authmode \tWIFI_AUTH_WPA2_WPA3_PSK");
    break;
  default:
```

```
    ESP_LOGI(TAG, "Authmode \tWIFI_AUTH_UNKNOWN");
    break;
  }
}

static void print_cipher_type(int pairwise_cipher, int group_cipher)
{
  switch (pairwise_cipher) {
  case WIFI_CIPHER_TYPE_NONE:
    ESP_LOGI(TAG, "Pairwise Cipher \tWIFI_CIPHER_TYPE_NONE");
    break;
  case WIFI_CIPHER_TYPE_WEP40:
    ESP_LOGI(TAG, "Pairwise Cipher \tWIFI_CIPHER_TYPE_WEP40");
    break;
  case WIFI_CIPHER_TYPE_WEP104:
    ESP_LOGI(TAG, "Pairwise Cipher \tWIFI_CIPHER_TYPE_WEP104");
    break;
  case WIFI_CIPHER_TYPE_TKIP:
    ESP_LOGI(TAG, "Pairwise Cipher \tWIFI_CIPHER_TYPE_TKIP");
    break;
  case WIFI_CIPHER_TYPE_CCMP:
    ESP_LOGI(TAG, "Pairwise Cipher \tWIFI_CIPHER_TYPE_CCMP");
    break;
  case WIFI_CIPHER_TYPE_TKIP_CCMP:
    ESP_LOGI(TAG, "Pairwise Cipher \tWIFI_CIPHER_TYPE_TKIP_CCMP");
    break;
  case WIFI_CIPHER_TYPE_AES_CMAC128:
    ESP_LOGI(TAG, "Pairwise Cipher \tWIFI_CIPHER_TYPE_AES_CMAC128");
    break;
  case WIFI_CIPHER_TYPE_SMS4:
    ESP_LOGI(TAG, "Pairwise Cipher \tWIFI_CIPHER_TYPE_SMS4");
    break;
  case WIFI_CIPHER_TYPE_GCMP:
```

```
  ESP_LOGI(TAG, "Pairwise Cipher \tWIFI_CIPHER_TYPE_GCMP");
  break;
case WIFI_CIPHER_TYPE_GCMP256:
  ESP_LOGI(TAG, "Pairwise Cipher \tWIFI_CIPHER_TYPE_GCMP256");
  break;
default:
  ESP_LOGI(TAG, "Pairwise Cipher \tWIFI_CIPHER_TYPE_UNKNOWN");
  break;
}

switch (group_cipher) {
case WIFI_CIPHER_TYPE_NONE:
  ESP_LOGI(TAG, "Group Cipher \tWIFI_CIPHER_TYPE_NONE");
  break;
case WIFI_CIPHER_TYPE_WEP40:
  ESP_LOGI(TAG, "Group Cipher \tWIFI_CIPHER_TYPE_WEP40");
  break;
case WIFI_CIPHER_TYPE_WEP104:
  ESP_LOGI(TAG, "Group Cipher \tWIFI_CIPHER_TYPE_WEP104");
  break;
case WIFI_CIPHER_TYPE_TKIP:
  ESP_LOGI(TAG, "Group Cipher \tWIFI_CIPHER_TYPE_TKIP");
  break;
case WIFI_CIPHER_TYPE_CCMP:
  ESP_LOGI(TAG, "Group Cipher \tWIFI_CIPHER_TYPE_CCMP");
  break;
case WIFI_CIPHER_TYPE_TKIP_CCMP:
  ESP_LOGI(TAG, "Group Cipher \tWIFI_CIPHER_TYPE_TKIP_CCMP");
  break;
case WIFI_CIPHER_TYPE_SMS4:
  ESP_LOGI(TAG, "Group Cipher \tWIFI_CIPHER_TYPE_SMS4");
  break;
case WIFI_CIPHER_TYPE_GCMP:
```

```
    ESP_LOGI(TAG, "Group Cipher \tWIFI_CIPHER_TYPE_GCMP");
    break;
  case WIFI_CIPHER_TYPE_GCMP256:
    ESP_LOGI(TAG, "Group Cipher \tWIFI_CIPHER_TYPE_GCMP256");
    break;
  default:
    ESP_LOGI(TAG, "Group Cipher \tWIFI_CIPHER_TYPE_UNKNOWN");
    break;
  }
}
```

O código define duas funções: `print_auth_mode` para imprimir informações sobre o modo de autenticação e `print_cipher_type` para imprimir informações sobre os tipos de criptografia.

Função `wifi_scan`:

```
C/C++
/* Initialize Wi-Fi as sta and set scan method */
static void wifi_scan(void)
{
  ESP_ERROR_CHECK(esp_netif_init());
  ESP_ERROR_CHECK(esp_event_loop_create_default());
  esp_netif_t *sta_netif = esp_netif_create_default_wifi_sta();
  assert(sta_netif);

  wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
  ESP_ERROR_CHECK(esp_wifi_init(&cfg));

  uint16_t number = DEFAULT_SCAN_LIST_SIZE;
  wifi_ap_record_t ap_info[DEFAULT_SCAN_LIST_SIZE];
  uint16_t ap_count = 0;
  memset(ap_info, 0, sizeof(ap_info));
```

```
  ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA));
  ESP_ERROR_CHECK(esp_wifi_start());
  esp_wifi_scan_start(NULL, true);
  ESP_ERROR_CHECK(esp_wifi_scan_get_ap_records(&number, ap_info));
  ESP_ERROR_CHECK(esp_wifi_scan_get_ap_num(&ap_count));
  ESP_LOGI(TAG, "Total APs scanned = %u", ap_count);
  for (int i = 0; (i < DEFAULT_SCAN_LIST_SIZE) && (i < ap_count); i++) {
    ESP_LOGI(TAG, "SSID \t\t%s", ap_info[i].ssid);
    ESP_LOGI(TAG, "RSSI \t\t%d", ap_info[i].rssi);
    print_auth_mode(ap_info[i].authmode);
    if (ap_info[i].authmode != WIFI_AUTH_WEP) {
      print_cipher_type(ap_info[i].pairwise_cipher, ap_info[i].group_cipher);
    }
    ESP_LOGI(TAG, "Channel \t\t%d\n", ap_info[i].primary);
  }
}
```

Esta função é responsável por executar o escaneamento das redes Wi-Fi. Ela realiza as seguintes etapas:

- Inicializa a interface de rede.
- Inicializa o Wi-Fi com a configuração padrão.
- Configura o modo de operação Wi-Fi como “STA” (Estação).
- Inicia a operação Wi-Fi.
- Inicia o escaneamento das redes Wi-Fi próximas.
- Obtém informações sobre as redes escaneadas, incluindo o número total de redes encontradas e informações detalhadas sobre cada uma delas.
- Imprime as informações das redes, incluindo o SSID (nome da rede), intensidade do sinal (RSSI), modo de autenticação, tipo de cifragem e canal primário.

Função `app_main`:

```
C/C++
void app_main(void)
{
```

```
  // Initialize NVS
  esp_err_t ret = nvs_flash_init();
  if (ret == ESP_ERR_NVS_NO_FREE_PAGES || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
    ESP_ERROR_CHECK(nvs_flash_erase());
    ret = nvs_flash_init();
  }
  ESP_ERROR_CHECK( ret );

  wifi_scan();
}
```

Esta função é o ponto de entrada principal do programa. Ela realiza as seguintes etapas:

- Inicializa o Non-Volatile Storage (NVS), que é usado para armazenamento persistente de dados na memória flash.
- Realiza o escaneamento das redes Wi-Fi chamando a função `wifi_scan`.

Em resumo, este código demonstra como usar a biblioteca ESP-IDF (Espressif IoT Development Framework) para realizar o escaneamento de redes Wi-Fi próximas em uma placa ESP32 e imprimir informações relevantes sobre essas redes, como o nome (SSID), intensidade do sinal (RSSI), modo de autenticação e tipo de cifragem.

Este código pode ser útil para coletar informações sobre redes Wi-Fi próximas, como parte de um projeto que envolve a seleção automática da melhor rede para conexão ou para fins de diagnóstico.

Considerações Finais

O WiFi é uma das principais funcionalidades do ESP32, permitindo que o dispositivo se conecte à internet e comunique-se com outros dispositivos através de redes sem fio.

Explore os demais exemplos disponíveis no repositório do ESP-IDF e crie suas aplicações sem fio.

Próximos passos

Parabéns por concluir este guia de iniciação ao ESP-IDF e ao mundo do ESP32! Agora que você tem uma base sólida no desenvolvimento com o ESP-IDF e explorou algumas das funcionalidades essenciais do ESP32, há muitas possibilidades para aprofundar seus conhecimentos e criar projetos mais avançados. Aqui estão alguns próximos passos que você pode seguir:

1. **Projetos com Sensores:** Utilize o conhecimento adquirido sobre GPIO, PWM e ADC para integrar sensores ao seu projeto. Por exemplo, você pode criar um projeto que monitora a temperatura ambiente usando um sensor de temperatura e exibe os dados em um display LCD.

2. **Comunicação sem Fio Avançada:** Explore ainda mais a conectividade sem fio do ESP32, utilizando Bluetooth para comunicação com outros dispositivos, como smartphones, ou criando redes mesh para conectar vários dispositivos ESP32.

3. **Controle Remoto e Automação Residencial:** Utilize o ESP32 para criar um sistema de automação residencial, onde você pode controlar luzes, eletrodomésticos e outros dispositivos remotamente através de um aplicativo ou interface web.

4. **Integração com a IoT:** Aproveite a capacidade do ESP32 para se conectar à internet e integre-o com plataformas de IoT como o MQTT para enviar e receber dados de outros dispositivos e serviços.

5. **Otimização e Economia de Energia:** Aprenda técnicas avançadas de otimização de código e economia de energia para maximizar o desempenho do seu projeto e prolongar a vida útil da bateria, se aplicável.

6. **Aprendizado Profundo (Deep Learning):** O ESP32 possui recursos para tarefas de aprendizado de máquina em dispositivos de borda. Explore o uso de bibliotecas como TensorFlow Lite para implementar modelos de aprendizado profundo em seu dispositivo.

7. **Contribuir para a Comunidade:** Compartilhe seu conhecimento e aprendizado com a comunidade. Participe de fóruns e grupos de desenvolvedores relacionados ao ESP32 e ESP-IDF, e contribua com projetos de código aberto.

Lembre-se de que a jornada de aprendizado é contínua, e o desenvolvimento com o ESP32 oferece uma infinidade de oportunidades para aprimorar suas habilidades e criar projetos incríveis. Continue explorando, experimentando e se divertindo com o ESP32 e o ESP-IDF!

Agora é hora de colocar em prática o que você aprendeu e embarcar em suas próprias jornadas de desenvolvimento com o ESP32. Divirta-se construindo projetos incríveis e aproveitando todo o potencial dessa plataforma versátil e poderosa!

Referências

Durante a elaboração deste guia de iniciação ao ESP-IDF e ao ESP32, foram utilizadas várias fontes de conhecimento para garantir a precisão e abrangência das informações apresentadas. Abaixo estão algumas das principais referências utilizadas:

1. **Documentação Oficial do ESP-IDF**: https://docs.espressif.com/projects/esp-idf/en/latest/
2. **Datasheets do ESP32**: https://www.espressif.com/pt/support/documents/technical-documents
3. **GitHub do ESP-IDF**: https://github.com/espressif/esp-idf
4. Portal Embarcados: Resultados da busca: ESp32 - Embarcados - Sua fonte de informações sobre Sistemas Embarcados
5. Livro: ESP32 Com IDF: O Guia Profissional - De José Morais
6. Documentação da Franzininho WiFi: Primeiros Passos com ESP-IDF | Franzininho

É importante lembrar que o desenvolvimento com o ESP32 é uma área em constante evolução, e novas informações, atualizações e recursos podem ser disponibilizados regularmente. Portanto, é sempre recomendado verificar as fontes oficiais e a comunidade para obter as informações mais atualizadas sobre o ESP-IDF e o ESP32.