Liga Acadêmica de Sistemas Embarcados
Centro de Informática · UFPE

FORMAÇÃO EM ESP32 · MÓDULO 3

Estação de **temperatura e umidade** com ESP-IDF

O mesmo projeto do módulo 2, agora no framework oficial da Espressif. Programa em C, tarefas do FreeRTOS, o driver I2C novo e um DHT22 lido por temporização, tudo simulado no Wokwi e preparado para a bancada.

Lucas Rayan Guerra
Ciência Embarcada · ESPDocs linkedin.com/in/lucasrguerra

Material de apoio
Formação presencial · uso livre para fins educacionais

Apresentação

O que muda em relação ao módulo 2, o que você precisa e como usar esta apostila.

No módulo 2 montamos uma estação que mede temperatura e umidade com um **DHT22** e mostra o resultado num **LCD 16×2**, usando o framework Arduino. Neste módulo o **circuito é exatamente o mesmo**, e o que muda é a camada de software: o programa passa a ser escrito em C com o **ESP-IDF**, o framework oficial da Espressif, o mesmo em que a própria empresa e a maioria dos produtos comerciais com ESP32 são desenvolvidos.

O Arduino escondia muita coisa: bibliotecas prontas falavam com o sensor e com o display, e bastava chamar `lcd.print()`. Aqui essas bibliotecas não existem, e por isso você vai ver, linha por linha, **como o I2C é configurado, como o display é inicializado em 4 bits e como o protocolo de um fio do DHT22 é lido por temporização**. É mais trabalho, e é também o melhor jeito de entender o que acontece dentro do chip.

O que você vai levar deste módulo

- Entender as camadas do ESP-IDF e a diferença entre `app_main()` e `setup()`/`loop()`.
- Criar, configurar, compilar, gravar e monitorar um projeto com `idf.py`.
- Ler a estrutura de um projeto: CMake, componentes, Kconfig e `sdkconfig`.
- Usar o driver I2C novo (`i2c_master`), tarefas do FreeRTOS, `esp_err_t` e o sistema de logs.
- Simular um firmware ESP-IDF no Wokwi e saber o que muda na bancada.

Pré-requisitos

Ter feito o módulo 2 ajuda bastante: a teoria do **DHT22** e do **barramento I2C** está lá e aqui só é recapitulada. Conhecer o básico de C (funções, ponteiros, `struct`) é suficiente. Para simular você precisa de um navegador e do ESP-IDF instalado para gerar o firmware, ou usar o `.bin` pronto.

Os arquivos deste módulo

Tudo o que aparece nesta apostila foi compilado e está na mesma pasta da formação:

ARQUIVO	CONTEÚDO
idf/	Fontes do projeto da estação: `main/estacao.c`, drivers, Kconfig, `wokwi.toml` e `diagram.json`
estacao-esp32s3.bin	Firmware já compilado, para simular sem instalar o ESP-IDF
idf-scanner-i2c/	Projeto de diagnóstico que lista os endereços do barramento I2C
modulo-3-idf.zip	Os dois projetos acima num só pacote
Apostila em PDF	Esta apostila, pronta para imprimir
Slides em PDF	Os slides da formação, em 16:9

Ao longo do texto você encontra quatro tipos de caixa:

DICA

Atalho, boa prática ou recomendação que economiza tempo.

ATENÇÃO

Ponto que costuma causar problema se passar despercebido.

ERRO COMUM

Armadilha concreta, com o sintoma que ela produz e como escapar.

CURIOSIDADE

Contexto ou detalhe técnico que ajuda a fixar o conceito.

Sumário

Nove capítulos, do Arduino ao ESP-IDF e da tela à bancada.

1 O projeto 5

2 Do Arduino ao ESP-IDF 7

3 Instalando e usando o ESP-IDF 9

4 Anatomia do projeto 11

5 O circuito 14

6 O programa 15

7 Simulando no Wokwi 25

8 Da simulação para a bancada 27

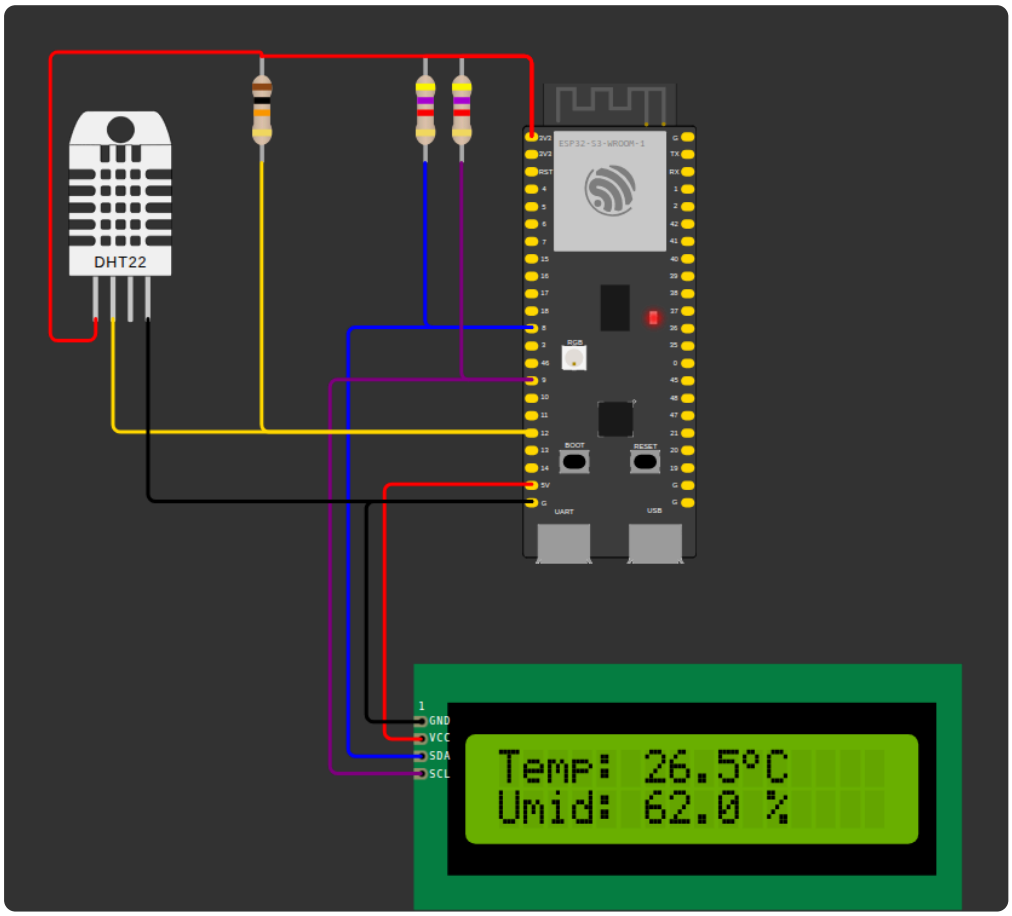
9 Problemas comuns e desafios 29

★ Avalie esta formação 31

1 O projeto

A mesma estação do módulo 2, com outra fundação.

A estação continua com três peças. O **DHT22** mede temperatura e umidade relativa do ar, o **ESP32-S3** pede uma medição a cada dois segundos e o **LCD 16×2** mostra os números. Nada muda na tela nem na fiação; o que muda é quem faz o trabalho dentro do programa.



O circuito do módulo 2 e deste módulo é o mesmo. O display mostra a leitura do DHT22 simulado: 26,5 °C e 62 % de umidade.

O que muda em relação ao módulo 2

ASPECTO	MÓDULO 2 (ARDUINO)	MÓDULO 3 (ESP-IDF)
Linguagem	C++ com a API do Arduino	**C com as APIs da Espressif**
Ponto de entrada	`setup()` e `loop()`	`app_main()` **e tarefas**
DHT22	Biblioteca da Adafruit	**Driver próprio de ~70 linhas**
LCD	Biblioteca LiquidCrystal I2C	**Driver próprio sobre o I2C do IDF**
Pausa de 2 s	`delay(2000)`	`vTaskDelay()` **do FreeRTOS**
Configuração	`#define` no código	`menuconfig` **e Kconfig**
Compilação	IDE ou Wokwi online	`idf.py`**, no computador**
Simulação	Wokwi compila o código	**Wokwi carrega o firmware pronto**

Lista de material

No Wokwi as peças já existem. Para montar em bancada você vai precisar de:

ITEM	QTD.	OBSERVAÇÃO
ESP32-S3-DevKitC-1	1	Qualquer variante de memória serve
Sensor DHT22 (AM2302)	1	O componente branco de 4 pinos
Display LCD 16×2 com módulo I2C	1	A plaquinha PCF8574 vem soldada atrás
Conversor de nível I2C (BSS138)	1	Explicado no capítulo 8
Resistor de 10 kΩ	1	Pull-up do DHT22 (marrom, preto, laranja)
Resistor de 4,7 kΩ	2	Pull-ups do I2C (amarelo, violeta, vermelho)
Protoboard, jumpers e cabo USB de dados	—	Cabo só de carga não grava a placa

2 Do Arduino ao ESP-IDF

Camadas, tarefas e o preço da liberdade.

O **ESP-IDF** (*Espressif IoT Development Framework*) é o conjunto de software que a Espressif mantém para o ESP32: drivers de todos os periféricos, pilha de Wi-Fi e Bluetooth, sistema de arquivos, segurança e o sistema operacional de tempo real **FreeRTOS**. Quando você usa o Arduino no ESP32, o que existe por baixo é o próprio ESP-IDF: o núcleo Arduino é uma camada de conveniência construída sobre ele.



Os dois caminhos chegam ao mesmo hardware passando pelo ESP-IDF. No Arduino, uma camada a mais traduz as chamadas simples; no módulo 3 você fala com o IDF diretamente.

app_main() no lugar de setup() e loop()

No Arduino, o sistema chama `setup()` uma vez e depois `loop()` sem parar. No ESP-IDF existe só a função `app_main()`, executada uma vez por uma tarefa do FreeRTOS criada pelo sistema. Quando ela termina, **o programa não acaba**: as outras tarefas continuam rodando. O padrão deste módulo é fazer a inicialização em `app_main()` e criar uma **tarefa** com um laço infinito, o equivalente do `loop()`.

FreeRTOS em um parágrafo

Uma **tarefa** é uma função com laço próprio, com sua pilha e sua prioridade; o FreeRTOS alterna entre as tarefas e entre os dois núcleos do ESP32-S3. A pausa de uma tarefa não trava o chip: `vTaskDelay()` coloca a tarefa para dormir e libera o processador para as demais, ao contrário de um `delay()` de laço ocupado. O tempo é medido em *ticks*, e a macro `pdMS_TO_TICKS(2000)` converte 2000 milissegundos para essa unidade.

O que se ganha e o que custa

GANHA	CUSTA
Acesso a todos os recursos do chip, sem camada no meio	Mais código e mais conceitos para começar
Multitarefa de verdade, com prioridades	Cuidado com pilha, concorrência e seções críticas
Erros explícitos (`esp_err_t`) e logs por etiqueta	Cada chamada precisa ter seu retorno tratado
É o caminho do produto: segurança, OTA, certificações	Sem o acervo de bibliotecas prontas do Arduino
Configuração central (Kconfig) e compilação reprodutível	Instalação e compilação mais pesadas

> **ARDUINO E ESP-IDF NÃO SÃO RIVAIS**
>
> É possível usar o Arduino como componente dentro de um projeto ESP-IDF e misturar as duas formas. Aprender o IDF não joga fora o que você sabe: explica por que o Arduino se comporta como se comporta.

3 Instalando e usando o ESP-IDF

Três caminhos de instalação e seis comandos para o dia a dia.

O ESP-IDF precisa do compilador cruzado, do CMake, do Python e de várias ferramentas. A Espressif oferece três formas de ter tudo isso pronto. Use a que combinar com o seu computador; este módulo pede o ESP-IDF 5.3 ou mais novo, em que o driver I2C `i2c_master` usado aqui está disponível.

CAMINHO	QUANDO USAR
Extensão do VS Code	O mais simples. A extensão **ESP-IDF** da Espressif baixa e configura tudo pelo assistente de instalação e põe botões de compilar, gravar e monitorar na barra inferior.
Instalador offline (Windows)	Instala o ESP-IDF e abre um terminal já configurado (*ESP-IDF PowerShell*). Bom para quem trabalha só no terminal.
Linha de comando (Linux e macOS)	Clonar o repositório `esp-idf`, rodar `./install.sh esp32s3` e carregar o ambiente com `. ./export.sh` a cada terminal novo.
Docker	A imagem `espressif/idf` traz o ambiente pronto, sem instalar nada no sistema. Ideal para computador emprestado ou para integração contínua.

O AMBIENTE PRECISA ESTAR CARREGADO

Se o terminal responder que `idf.py` não foi encontrado, o ambiente do IDF não foi carregado naquela janela. No Linux e no macOS rode `. ./export.sh` dentro da pasta do ESP-IDF; no Windows abra o terminal *ESP-IDF* pelo menu Iniciar; no VS Code use o terminal que a extensão abre.

O ciclo de trabalho com idf.py

Dentro da pasta do projeto, estes são os comandos que você vai repetir o tempo todo:

COMANDO	O QUE FAZ
`idf.py set-target esp32s3`	Escolhe o chip e gera o `sdkconfig`. Faça uma vez por projeto (ou ao trocar de chip).
`idf.py menuconfig`	Abre o menu de configuração em texto: pinos, endereço do LCD e centenas de opções do sistema.
`idf.py build`	Compila e gera o firmware na pasta `build/`.
`idf.py -p PORTA flash`	Grava o firmware na placa pela porta serial (`/dev/ttyUSB0`, `/dev/ttyACM0` ou `COMx`).
`idf.py -p PORTA monitor`	Mostra a saída do programa. Para sair, use **Ctrl+]**.
`idf.py -p PORTA flash monitor`	Grava e já abre o monitor, o comando mais usado.
`idf.py fullclean`	Apaga a pasta `build/`; resolve estados estranhos de compilação.

PRIMEIRO PROJETO: OS EXEMPLOS JÁ EXISTEM

O ESP-IDF traz dezenas de exemplos em `examples/`. Antes de escrever do zero, copie `examples/get-started/hello_world`, rode o ciclo completo e só depois troque o código. Se o `hello_world` funciona, a instalação está certa.

A PORTA SERIAL NÃO APARECE OU DÁ "PERMISSÃO NEGADA"

No Linux, seu usuário precisa estar no grupo `dialout` (ou `uucp`, conforme a distribuição): `sudo usermod -aG dialout $USER`, e depois sair e entrar na sessão. No Windows e no macOS, troque o cabo por um de dados e confira se o sistema reconhece a placa. Na DevKitC-1 há duas portas USB; a rotulada **UART** é a mais confiável para gravar.

4 Anatomia do projeto

CMake, o componente main e a configuração em Kconfig.

Um projeto ESP-IDF é uma pasta com uma estrutura fixa. Cada pedaço do código vive num **componente**, e o único que você precisa criar é o `main`:

```
idf/
├── CMakeLists.txt          # o projeto: nome e inclusão do sistema de build
├── sdkconfig.defaults      # valores iniciais da configuração
├── wokwi.toml              # aponta o Wokwi para o firmware compilado
├── diagram.json            # o circuito do Wokwi
└── main/
    ├── CMakeLists.txt      # registra o componente main
    ├── Kconfig.projbuild   # opções que aparecem no menuconfig
    ├── estacao.c           # app_main() e a tarefa da estação
    ├── dht22.c / dht22.h   # driver do sensor
    └── lcd1602.c / .h      # driver do display
```

CMakeLists.txt da raiz

Define o nome do projeto e importa o sistema de build do IDF. São só três linhas, e quase nunca mudam:

```
cmake_minimum_required(VERSION 3.16)
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
project(estacao)
```

CMakeLists.txt do componente main

Lista os arquivos-fonte, as pastas de cabeçalhos e, em `REQUIRES`, os componentes do IDF de que o código depende. Cada driver que o programa usa precisa aparecer aqui:

```
idf_component_register(SRCS "estacao.c" "lcd1602.c" "dht22.c"
                       INCLUDE_DIRS "."
                       REQUIRES esp_driver_i2c esp_driver_gpio esp_timer esp_rom)
```

COMPONENTE	PARA QUE O PROJETO O USA
esp_driver_i2c	Driver I2C novo (`i2c_master.h`) do LCD
esp_driver_gpio	Leitura e escrita do pino do DHT22
esp_timer	Relógio em microssegundos para medir os pulsos
esp_rom	Atrasos curtos em microssegundos (`esp_rom_delay_us`)

Kconfig.projbuild e sdkconfig

No Arduino, o pino do sensor era um `#define`. No IDF, valores que dependem da montagem viram **opções de configuração**: o arquivo `Kconfig.projbuild` as declara, o `idf.py menuconfig` deixa você editá-las e o resultado vai para o `sdkconfig`, que o build converte em macros `CONFIG_...` disponíveis no código.

```
menu "Estação de temperatura e umidade"

    config PINO_DHT
        int "GPIO do DATA do DHT22"
        default 12
        range 0 48

    config PINO_SDA
        int "GPIO do SDA (I2C)"
        default 8
        range 0 48

    config PINO_SCL
        int "GPIO do SCL (I2C)"
        default 9
        range 0 48

    config LCD_ENDERECO
        hex "Endereço I2C do LCD (PCF8574)"
        default 0x27
        range 0x03 0x77
        help
            0x27 para o PCF8574 e 0x3F para o PCF8574A.
            Se o LCD não responder, rode o projeto idf-scanner-i2c.

endmenu
```

Assim, `CONFIG_PINO_DHT`, `CONFIG_PINO_SDA`, `CONFIG_PINO_SCL` e `CONFIG_LCD_ENDERECO` existem no programa sem nenhum `#define`. Trocar o display de endereço é abrir o menu, não recompilar à mão. A faixa `range` impede valores impossíveis, como um GPIO 99.

sdkconfig.defaults

Ajustes que valem para quem clonar o projeto, sem precisar versionar o `sdkconfig` inteiro:

```
# Alvo e ajustes do projeto (o idf.py set-target esp32s3 grava o alvo no sdkconfig)
CONFIG_IDF_TARGET="esp32s3"
CONFIG_ESPTOOLPY_FLASHSIZE_4MB=y
# Console pela UART0 (o Wokwi liga o Serial Monitor nela)
CONFIG_ESP_CONSOLE_UART_DEFAULT=y
```

NÃO EDITE O SDKCONFIG À MÃO

O `sdkconfig` é gerado e tem milhares de linhas. Mude opções pelo `menuconfig` e guarde no `sdkconfig.defaults` só o que o projeto exige. Se um valor do `sdkconfig.defaults` não surtir efeito, apague o `sdkconfig` e rode `set-target` de novo: os defaults só são lidos quando o arquivo não existe.

5 O circuito

Nove ligações e três resistores, como no módulo 2.

A teoria do **DHT22** (como ele mede e como um bit atravessa o fio) e do **I2C** (SDA, SCL, endereços e dreno aberto) está nos capítulos 2 e 3 da apostila do módulo 2. Aqui fica só a recapitulação do que importa para a fiação; o circuito montado está na figura do capítulo 1.

DE	PARA	COR E FUNÇÃO
DHT22 VCC	3V3	Vermelho · alimentação
DHT22 DATA	GPIO12	Amarelo · dados do sensor
DHT22 GND	GND	Preto · terra
Resistor 10 kΩ	GPIO12 ↔ 3V3	Pull-up do DHT22
LCD VCC	5V	Vermelho · o display precisa de 5 V
LCD GND	GND	Preto · terra
LCD SDA	GPIO8	Azul · dados do I2C
LCD SCL	GPIO9	Roxo · relógio do I2C
2 resistores de 4,7 kΩ	GPIO8 e GPIO9 ↔ 3V3	Pull-ups do I2C

No IDF os pinos não têm padrão embutido: GPIO8 e GPIO9 são os valores do Kconfig.

PULL-UPS: OS EXTERNOS CONTINUAM OBRIGATÓRIOS

O driver I2C do IDF consegue ligar os pull-ups internos do ESP32 (`enable_internal_pullup`), e o código deste módulo liga. Mas eles valem cerca de 45 kΩ, fracos demais para o I2C a 100 kHz com fios de verdade. Os **4,7 kΩ para o 3V3** seguem no circuito, e o de **10 kΩ no DATA** do DHT22 também. Os pinos do ESP32-S3 não toleram 5 V, por isso o pull-up vai para o 3V3 e nunca para o 5V.

6 O programa

Uma tarefa, dois drivers e cada chamada com seu resultado checado.

O programa tem três arquivos de código: `estacao.c`, com a lógica, e dois drivers escritos para este módulo, `lcd1602.c` e `dht22.c`. Os trechos abaixo são copiados dos arquivos da pasta idf/main/, que foram compilados e simulados. Começamos pelo topo, `estacao.c`, e descemos até os drivers.

1. estacao.c: a configuração

```c
// Formação em ESP32 - LASER - Módulo 3
// Estação de temperatura e umidade com ESP-IDF: DHT22 + LCD 16x2 I2C + ESP32-S3
#include <stdio.h>
#include "esp_log.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "sdkconfig.h"
#include "dht22.h"
#include "lcd1602.h"

static const char *TAG = "estacao";

// Símbolo de grau: 8 linhas de 5 pixels (mesmo desenho do módulo 2)
static const uint8_t GRAU[8] = { 0b01100, 0b10010, 0b10010, 0b01100, 0, 0, 0, 0 };

static lcd1602_t lcd;
```

Os cabeçalhos trazem os logs (`esp_log.h`), o FreeRTOS e os dois drivers. `sdkconfig.h` define as macros `CONFIG_...` do Kconfig, e `TAG` é a etiqueta que identifica este arquivo nas mensagens do monitor. O vetor `GRAU` é o mesmo desenho do módulo 2, linha a linha em binário, agora `const` e guardado na memória de programa.

2. estacao.c: escrever uma linha

```c
// Escreve uma linha de 16 colunas (o resto fica em branco)
static void linha(int n, const char *texto)
{
    char buf[17];
    snprintf(buf, sizeof(buf), "%-16s", texto);
    lcd1602_cursor(&lcd, 0, n);
    lcd1602_print(&lcd, buf);
}
```

O formato `%-16s` alinha o texto à esquerda e **completa com espaços até 16 colunas**. É a solução limpa para o problema das sobras de caracteres do módulo 2: toda linha escrita apaga a anterior, sem `clear()` e sem a tela piscar.

3. estacao.c: a tarefa

```c
// Tarefa: lê o sensor e atualiza o LCD a cada 2 s
static void tarefa_estacao(void *arg)
{
    while (true) {
        float temperatura, umidade;
        esp_err_t err = dht22_ler(CONFIG_PINO_DHT, &temperatura, &umidade);

        if (err ≠ ESP_OK) {
            ESP_LOGE(TAG, "Falha na leitura do DHT22: %s", esp_err_to_name(err));
            linha(0, "Falha no sensor");
            linha(1, "");
        } else {
            char buf[17];
            // \010 (código 8) desenha o slot 0 da CGRAM: o símbolo de grau
            snprintf(buf, sizeof(buf), "Temp: %.1f\010C", temperatura);
            linha(0, buf);
            snprintf(buf, sizeof(buf), "Umid: %.1f %%", umidade);
            linha(1, buf);
            ESP_LOGI(TAG, "Temperatura: %.1f C | Umidade: %.1f %%", temperatura, umidade);
        }
        vTaskDelay(pdMS_TO_TICKS(2000));   // o DHT22 aceita 1 leitura a cada 2 s
    }
}
```

Esta função é o `loop()` do módulo 2. O laço `while (true)` nunca termina, e `vTaskDelay()` no fim dorme por dois segundos sem travar o resto do sistema. O sensor devolve um `esp_err_t`: se não for `ESP_OK`, o display mostra "Falha no sensor" e o erro vai para o log. Se der certo, temperatura e umidade são formatadas com `snprintf`, uma casa decimal, e escritas nas duas linhas. A sequência `\010` é o **caractere de código 8 em octal**: o HD44780 espelha as posições 0 a 7 da memória de caracteres também em 8 a 15, e escrever 8 imprime o grau guardado no slot 0.

4. estacao.c: app_main()

```
void app_main(void)
{
    esp_err_t err = lcd1602_init(&lcd, CONFIG_PINO_SDA, CONFIG_PINO_SCL, CONFIG_LCD_ENDERECO);
    if (err ≠ ESP_OK) {
        ESP_LOGE(TAG, "LCD não respondeu (%s). Confira SDA, SCL, alimentação e o endereço
0x%02X",
                 esp_err_to_name(err), CONFIG_LCD_ENDERECO);
    }
    ESP_ERROR_CHECK_WITHOUT_ABORT(lcd1602_create_char(&lcd, 0, GRAU));
    linha(0, "LASER  ESP-IDF");
    linha(1, "Iniciando...");
    vTaskDelay(pdMS_TO_TICKS(2000));       // o DHT22 precisa de ~2 s para a primeira leitura
    lcd1602_clear(&lcd);

    xTaskCreate(tarefa_estacao, "estacao", 4096, NULL, 5, NULL);
}
```

Chamada uma vez, inicializa o LCD, registra o símbolo de grau, mostra a abertura por dois segundos (a primeira leitura do DHT22 precisa desse tempo) e cria a tarefa. `xTaskCreate()` recebe a função, um nome, o tamanho da pilha em bytes (4096), um argumento, a prioridade (5) e um ponteiro opcional para guardar o identificador. Se o LCD não responder, o programa **avisa no log e segue**, em vez de travar: assim o sensor continua a ser lido e impresso no monitor.

5. O driver I2C novo

Até o ESP-IDF 5.1 o I2C usava `driver/i2c.h`, com comandos montados à mão. O driver **i2c_master** separa as coisas em dois níveis: o *barramento* (os dois fios e a velocidade) e os *dispositivos* nele. Primeiro o cabeçalho do nosso driver de LCD, que esconde isso do resto do programa:

```
#pragma once
#include "driver/i2c_master.h"
#include "esp_err.h"

typedef struct {
    i2c_master_dev_handle_t dev;    // LCD no barramento I2C
} lcd1602_t;

// Cria o barramento I2C, liga o LCD e deixa o display pronto (backlight ligado)
esp_err_t lcd1602_init(lcd1602_t *lcd, int sda, int scl, uint8_t endereco);
esp_err_t lcd1602_clear(lcd1602_t *lcd);
esp_err_t lcd1602_cursor(lcd1602_t *lcd, int coluna, int linha);
esp_err_t lcd1602_print(lcd1602_t *lcd, const char *texto);
esp_err_t lcd1602_write_char(lcd1602_t *lcd, uint8_t caractere);
// Grava um desenho de 5x8 pixels em um dos 8 slots da CGRAM (0 a 7)
esp_err_t lcd1602_create_char(lcd1602_t *lcd, uint8_t slot, const uint8_t padrao[8]);
```

E a parte da inicialização que cria o barramento e registra o display:

```
esp_err_t lcd1602_init(lcd1602_t *lcd, int sda, int scl, uint8_t endereco)
{
    // Barramento I2C novo (driver/i2c_master.h)
    i2c_master_bus_config_t bus_cfg = {
        .i2c_port = I2C_NUM_0,
        .sda_io_num = sda,
        .scl_io_num = scl,
        .clk_source = I2C_CLK_SRC_DEFAULT,
        .glitch_ignore_cnt = 7,
        // Pull-up interno é fraco (~45 kΩ): NÃO substitui os 4,7 kΩ externos
        .flags.enable_internal_pullup = true,
    };
    i2c_master_bus_handle_t bus;
    ESP_ERROR_CHECK(i2c_new_master_bus(&bus_cfg, &bus));

    // O LCD é um dispositivo nesse barramento
    i2c_device_config_t dev_cfg = {
        .dev_addr_length = I2C_ADDR_BIT_LEN_7,
        .device_address = endereco,
        .scl_speed_hz = 100000,
    };
    ESP_ERROR_CHECK(i2c_master_bus_add_device(bus, &dev_cfg, &lcd->dev));
```

`i2c_new_master_bus()` configura a porta (`I2C_NUM_0`), os pinos, a fonte de relógio padrão e um filtro de ruído de 7 ciclos (`glitch_ignore_cnt`). `i2c_master_bus_add_device()` cria o *dispositivo*: endereço de 7 bits e velocidade de 100 kHz. O resultado, `lcd->dev`, é o identificador usado em toda transmissão. `ESP_ERROR_CHECK` aborta o programa com uma mensagem se a criação falhar: aqui é apropriado, porque sem barramento não há o que fazer.

6. O LCD por trás do PCF8574

A plaquinha PCF8574 é um expansor com 8 saídas controladas pelo I2C. Elas estão ligadas ao display assim: três bits de controle e os quatro bits de dados altos do HD44780, que por isso é usado em **modo de 4 bits**.

BIT	SINAL	FUNÇÃO
P0	RS	0 = comando, 1 = caractere
P1	RW	Sempre 0: só escrevemos no display
P2	E	*Enable*: o display lê o dado quando E desce
P3	Backlight	Liga a luz de fundo
P4 a P7	D4 a D7	Os quatro bits de dados

```c
// Driver do LCD 16x2 com a plaquinha I2C (PCF8574), em modo de 4 bits.
#include "lcd1602.h"
#include "esp_rom_sys.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

// Bits do PCF8574: P0=RS, P1=RW, P2=E, P3=backlight, P4..P7=D4..D7
#define BIT_RS  0x01
#define BIT_E   0x04
#define BIT_BL  0x08

#define TIMEOUT_MS 100
```

```c
// Envia 4 bits: coloca o dado, sobe E e desce E (o LCD lê na descida)
static esp_err_t enviar_nibble(lcd1602_t *lcd, uint8_t nibble, uint8_t rs)
{
    uint8_t base = (nibble << 4) | BIT_BL | rs;
    uint8_t seq[2] = { base | BIT_E, base };
    esp_err_t err = i2c_master_transmit(lcd->dev, seq, sizeof(seq), TIMEOUT_MS);
    esp_rom_delay_us(50);
    return err;
}
```

```c
// Um byte = dois nibbles (primeiro o mais significativo)
static esp_err_t enviar(lcd1602_t *lcd, uint8_t valor, uint8_t rs)
{
    esp_err_t err = enviar_nibble(lcd, valor >> 4, rs);
    return err != ESP_OK ? err : enviar_nibble(lcd, valor & 0x0F, rs);
}

static esp_err_t comando(lcd1602_t *lcd, uint8_t cmd) { return enviar(lcd, cmd, 0); }
```

Um byte de dados ou comando vai em **dois pedaços de 4 bits** (*nibbles*), o mais significativo primeiro. Cada nibble é montado no byte do PCF8574 e enviado duas vezes numa única transação I2C: uma com `E` em 1 e outra com `E` em 0. A descida de `E` faz o display capturar o dado. O `i2c_master_transmit()` recebe o dispositivo, os bytes, a quantidade e um limite de espera em milissegundos.

```
    // Sequência de partida do HD44780: força o modo de 4 bits
    vTaskDelay(pdMS_TO_TICKS(50));
    esp_err_t err = enviar_nibble(lcd, 0x03, 0);
    if (err ≠ ESP_OK) {
        return err;   // LCD não respondeu: fiação ou endereço errados
    }
    vTaskDelay(pdMS_TO_TICKS(5));
    enviar_nibble(lcd, 0x03, 0);
    esp_rom_delay_us(150);
    enviar_nibble(lcd, 0x03, 0);
    enviar_nibble(lcd, 0x02, 0);          // passa para 4 bits

    comando(lcd, 0x28);                   // 4 bits, 2 linhas, fonte 5x8
    comando(lcd, 0x0C);                   // display ligado, sem cursor
    comando(lcd, 0x06);                   // cursor anda para a direita
    return lcd1602_clear(lcd);
}
```

Depois de ligado, o HD44780 está em modo de 8 bits e precisa de uma sequência fixa do fabricante para passar para 4: três envios de `0x3` e um de `0x2`, com pausas. A primeira transmissão é também o teste de presença: se o display não responder (`err` diferente de `ESP_OK`), a função desiste e devolve o erro. Os comandos finais configuram duas linhas, ligam o display sem cursor e fazem o cursor andar para a direita.

7. Posição, texto e caractere de grau

```
esp_err_t lcd1602_cursor(lcd1602_t *lcd, int coluna, int linha)
{
    static const uint8_t inicio_linha[] = { 0x00, 0x40 };
    return comando(lcd, 0x80 | (inicio_linha[linha] + coluna));
}
```

```
esp_err_t lcd1602_print(lcd1602_t *lcd, const char *texto)
{
    esp_err_t err = ESP_OK;
    for (; *texto && err == ESP_OK; texto++) {
        err = lcd1602_write_char(lcd, (uint8_t)*texto);
    }
    return err;
}
```

```c
esp_err_t lcd1602_create_char(lcd1602_t *lcd, uint8_t slot, const uint8_t padrao[8])
{
    esp_err_t err = comando(lcd, 0x40 | ((slot & 7) << 3));   // endereço na CGRAM
    for (int i = 0; i < 8 && err == ESP_OK; i++) {
        err = enviar(lcd, padrao[i], BIT_RS);
    }
    return err;
}
```

O cursor usa o comando `0x80` somado ao endereço de memória do display: a linha 1 começa em `0x00` e a linha 2 em `0x40`. `print` envia um caractere por vez, parando no primeiro erro. `create_char` manda o comando `0x40` com o número do slot para apontar para a **CGRAM**, a memória dos caracteres personalizados, e em seguida as 8 linhas do desenho como dados.

8. O DHT22 por temporização

O protocolo do DHT22, descrito no módulo 2, não é I2C nem SPI: o valor de cada bit está na **duração** de um pulso. Sem biblioteca, o driver mede esses tempos com o relógio em microssegundos do ESP32. A função auxiliar espera a linha chegar a um nível e diz quanto tempo levou:

```c
// Driver do DHT22 por bit-bang: o protocolo é medido em microssegundos.
#include "dht22.h"
#include "driver/gpio.h"
#include "esp_rom_sys.h"
#include "esp_timer.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"

// Espera o pino chegar a `nivel`; devolve quantos us levou (-1 se estourou o limite)
static int esperar_nivel(int pino, int nivel, int limite_us)
{
    int64_t inicio = esp_timer_get_time();
    while (gpio_get_level(pino) != nivel) {
        if (esp_timer_get_time() - inicio > limite_us) {
            return -1;
        }
    }
    return (int)(esp_timer_get_time() - inicio);
}
```

E a leitura em si, em três partes. Primeiro o pedido ao sensor:

```
esp_err_t dht22_ler(int pino, float *temperatura, float *umidade)
{
    uint8_t dado[5] = {0};
    esp_err_t erro = ESP_OK;

    // Open-drain: o ESP só puxa para 0; o resistor de 10 k externo leva a 1
    gpio_set_direction(pino, GPIO_MODE_INPUT_OUTPUT_OD);
    gpio_set_level(pino, 1);

    // Pedido de leitura: linha em 0 por ~2 ms, depois solta
    gpio_set_level(pino, 0);
    esp_rom_delay_us(2000);
```

O pino é configurado como **entrada e saída em dreno aberto** (`GPIO_MODE_INPUT_OUTPUT_OD`): o ESP32 só sabe puxar a linha para 0, e quem a leva a 1 é o resistor de 10 kΩ, como no sensor. A linha cai por cerca de 2 ms, o pedido de leitura, e depois é solta.

```
    // Os bits duram ~50 us: sem interrupções até o fim da leitura (~4 ms)
    static portMUX_TYPE mux = portMUX_INITIALIZER_UNLOCKED;
    portENTER_CRITICAL(&mux);

    gpio_set_level(pino, 1);
    // Resposta do sensor: 80 us em 0, 80 us em 1
    if (esperar_nivel(pino, 0, 100) < 0 ||
        esperar_nivel(pino, 1, 100) < 0 ||
        esperar_nivel(pino, 0, 100) < 0) {
        erro = ESP_ERR_TIMEOUT;
    }

    // 40 bits: 50 us em 0, depois o tempo em 1 decide o bit (~27 us = 0, ~70 us = 1)
    for (int i = 0; i < 40 && erro == ESP_OK; i++) {
        if (esperar_nivel(pino, 1, 100) < 0) { erro = ESP_ERR_TIMEOUT; break; }
        int alto = esperar_nivel(pino, 0, 120);
        if (alto < 0) { erro = ESP_ERR_TIMEOUT; break; }
        dado[i / 8] ⟸ 1;
        if (alto > 40) {
            dado[i / 8] ⊨ 1;
        }
    }
    portEXIT_CRITICAL(&mux);
```

Os pulsos duram dezenas de microssegundos, e qualquer interrupção no meio estragaria a medida. Por isso a leitura inteira, cerca de 4 ms, fica numa **seção crítica** (`portENTER_CRITICAL`), que bloqueia as interrupções do núcleo. Primeiro a resposta do sensor, de 80 µs em 0 e 80 µs em 1; depois os 40 bits. O tempo em nível alto decide o bit: acima de 40 µs é **1**, abaixo é **0**. Cada espera tem um limite, e qualquer estouro vira `ESP_ERR_TIMEOUT`.

```
    if (erro ≠ ESP_OK) {
        return erro;
    }
    // O último byte é a soma dos quatro primeiros
    if ((uint8_t)(dado[0] + dado[1] + dado[2] + dado[3]) ≠ dado[4]) {
        return ESP_ERR_INVALID_CRC;
    }

    // Valores em décimos; o bit mais alto da temperatura é o sinal
    *umidade = ((dado[0] << 8) | dado[1]) / 10.0f;
    float t = (((dado[2] & 0x7F) << 8) | dado[3]) / 10.0f;
    *temperatura = (dado[2] & 0x80) ? -t : t;
    return ESP_OK;
}
```

Fora da seção crítica, o resultado é validado: o quinto byte é a soma dos quatro primeiros, e se não bater os dados vieram corrompidos (`ESP_ERR_INVALID_CRC`). Umidade e temperatura chegam em **décimos** em 16 bits; o bit mais alto da temperatura é o sinal, por isso temperaturas negativas funcionam. O cabeçalho `dht22.h` resume o contrato:

```
#pragma once
#include "esp_err.h"

// Lê o DHT22 no GPIO indicado. Devolve ESP_OK, ESP_ERR_TIMEOUT
// (sensor não responde) ou ESP_ERR_INVALID_CRC (dados corrompidos).
esp_err_t dht22_ler(int pino, float *temperatura, float *umidade);
```

> **SEÇÃO CRÍTICA É CURTA OU É PROBLEMA**
>
> Com as interrupções bloqueadas, o Wi-Fi e os outros periféricos daquele núcleo param. 4 ms a cada 2 s é aceitável; centenas de milissegundos seriam um defeito. Quando o projeto crescer com Wi-Fi, o caminho é o periférico RMT, que mede pulsos por hardware e dispensa a seção crítica.

9. Erros e logs

Quase toda função do IDF devolve um `esp_err_t`: `ESP_OK` (zero) para sucesso, ou um código como `ESP_ERR_TIMEOUT`. Três formas de tratar o retorno aparecem neste programa:

FORMA	QUANDO USAR
`ESP_ERROR_CHECK(x)`	Erro irrecuperável: imprime o arquivo e a linha e reinicia a placa.
`ESP_ERROR_CHECK_WITHOUT_ABORT(x)`	Erro que deve ser avisado, mas não impede o resto de funcionar.
`if (err ≠ ESP_OK) { ... }`	Erro esperado, que o programa trata, como a falha do sensor.

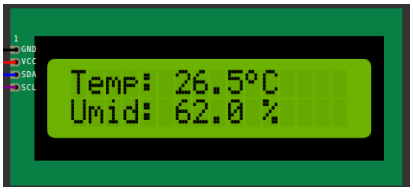
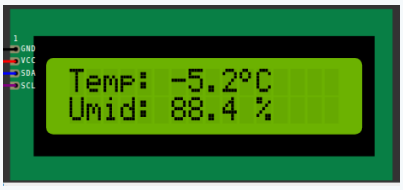
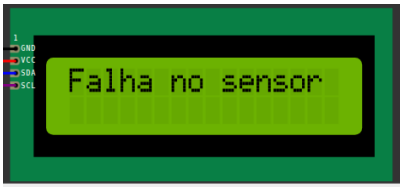
Para mensagens use `ESP_LOGI` (informação), `ESP_LOGW` (aviso) e `ESP_LOGE` (erro), sempre com a etiqueta `TAG`. O monitor mostra cada nível com uma cor e o tempo desde a partida, e `esp_err_to_name()` traduz o código numérico para um texto legível:

```
I (412) estacao: Temperatura: 26.5 C | Umidade: 62.0 %
E (2415) estacao: Falha na leitura do DHT22: ESP_ERR_TIMEOUT
```

ESQUECER O REQUIRES NO CMAKE

Se o build reclamar que `driver/i2c_master.h` não existe, o componente `esp_driver_i2c` não está listado em `REQUIRES` no `CMakeLists.txt` do `main`. É a primeira falha de quem usa um periférico novo no IDF.

Os três testes feitos no Wokwi

SITUAÇÃO	O QUE O DISPLAY MOSTROU
Sala, 26,5 °C e 62 %	
Frio, −5,2 °C e 88,4 %	
Fio de dados desconectado	

No terceiro teste o monitor registra `ESP_ERR_TIMEOUT` e o display avisa a falha.

7 Simulando no Wokwi

O Wokwi não compila ESP-IDF: ele roda o firmware que você gera.

No módulo 2 o Wokwi online compilava o `sketch.ino`. Para o ESP-IDF isso não existe: o Wokwi **simula um firmware já compilado**. O caminho é compilar no seu computador com `idf.py build` e entregar o resultado ao simulador. Há duas formas, e a segunda dispensa o ESP-IDF se você usar o `.bin` do pacote.

Opção A: no navegador, com o .bin

1 Tenha o binário único

Gere com `idf.py merge-bin -o estacao-esp32s3.bin` depois do build, ou use o `estacao-esp32s3.bin` da pasta do módulo.

2 Crie um projeto ESP32-S3

Acesse **wokwi.com/projects/new/esp32-s3** e substitua o conteúdo da aba *diagram.json* pelo arquivo de mesmo nome da pasta `idf/`.

3 Envie o firmware

Clique no editor de código, pressione **F1** e escolha **Upload Firmware and Start Simulation...**. Selecione o `.bin` e a simulação começa.

Opção B: no VS Code, com a extensão Wokwi

1 Compile

Rode `idf.py build` e abra a pasta `idf/` no VS Code. Ela precisa ter o `wokwi.toml` e o `diagram.json` na raiz.

2 Inicie o simulador

Pressione **F1** e escolha **Wokwi: Start Simulator**. A cada novo `idf.py build` o Wokwi recarrega o firmware.

O arquivo `wokwi.toml` diz ao simulador onde estão o firmware e o ELF, que traz os símbolos para depuração:

```
[wokwi]
version = 1
firmware = 'build/flasher_args.json'
elf = 'build/estacao.elf'
```

MUDE O CLIMA COM UM CLIQUE

Com a simulação rodando, clique no DHT22: aparecem os controles deslizantes de temperatura e umidade. Mova-os e veja o display acompanhar na leitura seguinte, e o monitor mostrar as linhas `estacao: Temperatura: ...` a cada 2 s.

O WOKWI ABRE MAS A TELA FICA VAZIA

Confira três coisas: se o `diagram.json` é o do módulo 3 (com pull-ups e ligações nos GPIO8, GPIO9 e GPIO12), se o firmware foi compilado para `esp32s3` (o `set-target` certo) e se você enviou o `.bin` único, não os arquivos soltos da pasta `build/`.

8 Da simulação para a bancada

O que o Wokwi perdoa e o hardware de verdade não.

O simulador modela os componentes como peças perfeitas. Na bancada, dois detalhes fazem diferença, e valem exatamente como no módulo 2.

1. Os pull-ups escondidos na plaquinha I2C

Quase toda plaquinha PCF8574 traz **resistores de pull-up ligados ao VCC dela**. Com o display alimentado em 5 V, esses resistores puxam SDA e SCL para 5 V, e somar os nossos 4,7 kΩ para 3V3 não resolve: a linha fica em torno de 4 V, ainda acima do que o ESP32-S3 suporta. Há duas saídas corretas:

SOLUÇÃO	COMO FAZER
Conversor de nível (recomendado)	Módulo bidirecional com BSS138. O lado **LV** vai no 3V3 e nos GPIO8/9; o lado **HV** vai no 5V e no SDA/SCL do display. O módulo já tem pull-ups dos dois lados, então os 4,7 kΩ externos saem do circuito.
Remover os pull-ups da plaquinha	Dessoldar os dois resistores ligados a SDA e SCL na plaquinha e manter os 4,7 kΩ para 3V3, como no Wokwi. O PCF8574 reconhece 3,3 V como nível alto mesmo alimentado em 5 V.

COMO SABER SE A SUA PLAQUINHA TEM PULL-UPS

Com tudo desligado, meça a resistência entre o pino **SDA** e o **VCC** da plaquinha. Algo entre 2 kΩ e 10 kΩ indica um pull-up. Sem continuidade, ela não tem, e os 4,7 kΩ para 3V3 bastam.

"MAS EU LIGUEI DIRETO E FUNCIONOU"

É comum funcionar: o resistor limita a corrente e o ESP32 tem diodos de proteção. Funcionar não é estar dentro da especificação, e a Espressif não garante os pinos acima de 3,6 V. Em projeto que vai ficar ligado, use o conversor.

2. Gravando na placa

Com o ambiente carregado, o ciclo é `idf.py set-target esp32s3`, `idf.py build` e `idf.py -p PORTA flash monitor`. Para a placa entrar em modo de gravação na DevKitC-1, em geral basta o comando; se ela não responder, segure o botão **BOOT**, aperte e solte **RESET** e solte o BOOT. O monitor deve mostrar uma linha com a temperatura a cada dois segundos.

3. Descubra o endereço do display

Se o display acende mas não mostra nada mesmo com o contraste ajustado, o endereço pode ser `0x3F` em vez de `0x27`. O projeto idf-scanner-i2c/ varre o barramento com `i2c_master_probe()`, que devolve `ESP_OK` quando alguém responde ao endereço, e imprime os que encontrar:

```c
// Scanner I2C: lista o endereço de cada dispositivo no barramento.
// Use antes da estação se o LCD nao mostrar nada.
#include "driver/i2c_master.h"
#include "esp_log.h"

static const char *TAG = "scanner";

void app_main(void)
{
    i2c_master_bus_config_t cfg = {
        .i2c_port = I2C_NUM_0,
        .sda_io_num = 8,
        .scl_io_num = 9,
        .clk_source = I2C_CLK_SRC_DEFAULT,
        .glitch_ignore_cnt = 7,
        .flags.enable_internal_pullup = true,   // não substitui os 4,7 kΩ externos
    };
    i2c_master_bus_handle_t bus;
    ESP_ERROR_CHECK(i2c_new_master_bus(&cfg, &bus));

    ESP_LOGI(TAG, "Procurando dispositivos I2C...");
    int encontrados = 0;
    for (uint8_t endereco = 1; endereco < 127; endereco++) {
        // i2c_master_probe devolve ESP_OK se alguém responder (ACK)
        if (i2c_master_probe(bus, endereco, 50) == ESP_OK) {
            ESP_LOGI(TAG, "Dispositivo em 0x%02X", endereco);
            encontrados++;
        }
    }
    if (encontrados == 0) {
        ESP_LOGW(TAG, "Nenhum dispositivo. Confira SDA, SCL e alimentação.");
    }
}
```

```
I (320) scanner: Procurando dispositivos I2C...
I (340) scanner: Dispositivo em 0x27
```

Achado o endereço, ajuste-o no `idf.py menuconfig`, em *Estação de temperatura e umidade*, opção *Endereço I2C do LCD*, e compile de novo.

9 Problemas comuns e desafios

O que fazer quando não funciona, e para onde ir quando funcionar.

SINTOMA	CAUSA PROVÁVEL
"Falha no sensor" o tempo todo	Fio de dados fora do GPIO12, falta do pull-up de 10 kΩ ou VCC e GND invertidos. O monitor mostra `ESP_ERR_TIMEOUT`.
`ESP_ERR_INVALID_CRC` **de vez em quando**	Fio longo ou ruído: a leitura chegou corrompida. Tente de novo na próxima volta.
"LCD não respondeu" no log	SDA e SCL trocados, display sem 5 V ou endereço errado: rode o scanner.
Display aceso e sem texto	Contraste desajustado no trimpot, ou endereço errado.
`idf.py: command not found`	O ambiente do ESP-IDF não foi carregado neste terminal.
Erro de compilação com `driver/i2c_master.h`	Falta `esp_driver_i2c` no `REQUIRES`, ou o IDF é antigo demais.
Pino ou endereço mudado e nada acontece	A opção foi alterada no `menuconfig`, mas falta recompilar com `idf.py build`.
Reinicia em loop com *Stack overflow*	A pilha da tarefa (4096 bytes) é pequena para o que foi acrescentado; aumente em `xTaskCreate`.

Desafios

1 Mínima e máxima

Guarde a menor e a maior temperatura desde que a placa ligou e alterne a segunda linha entre a umidade e esses dois valores.

2 Duas tarefas

Separe a leitura do sensor e a atualização do display em duas tarefas que trocam dados por uma fila (`xQueueCreate`, `xQueueSend`, `xQueueReceive`).

3 Alerta de calor

Acenda o LED RGB do GPIO38 da placa quando a temperatura passar de 30 °C, usando o driver `gpio` do IDF.

4 Mais uma opção no Kconfig

Crie a opção `INTERVALO_MS` no `Kconfig.projbuild` e use `CONFIG_INTERVALO_MS` no lugar do 2000 fixo, respeitando o mínimo de 2 s do DHT22.

PRÓXIMO PASSO

A estação já mede e mostra. O ESP-IDF traz Wi-Fi e um cliente MQTT prontos (`esp_wifi`, `mqtt_client`): publicar essas leituras num servidor é o caminho natural.

DÚVIDAS, CORREÇÕES E CONTATO

Correções e sugestões são bem-vindas e ajudam as próximas turmas. Fale direto com o autor por contato@lucasrguerra.dev.br ou pelo LinkedIn, em linkedin.com/in/lucasrguerra.

Avalie esta formação

Dois minutos que ajudam as próximas turmas.



Aponte a câmera do celular para o código

ou acesse https://qr.codes/YQN2q6

Sua opinião orienta o que manter, o que corrigir e o que incluir nos próximos módulos: o ritmo, o nível das explicações, o que ficou confuso e o que gostaria de ver depois. As respostas levam poucos minutos. Obrigado por participar.