Aluno do Embarcatech_37 no IFMA

Nome: Manoel Felipe Costa Furtado

Matrícula: 20251RSE.MTC0086

Atividade – Referente ao capítulo 02 da unidade 02

Tema do Capítulo - IEEE 802.11, LoRaWAN, SigFox, 4G, 5G

Prazo dia 08/06/2025 as 23:59

Enunciado: A atividade propõe o desenvolvimento de um servidor HTTP embarcado no Raspberry Pi Pico W, capaz de atuar como Access Point, fornecer uma página HTML para controle de LED e leitura da temperatura interna.

Desenvolver uma aplicação embarcada utilizando o Raspberry Pi Pico W capaz de:

- Criar uma rede Wi-Fi local (Access Point);
- Executar um servidor HTTP embarcado;
- Controlar um LED, conectado no GPIO13, por meio do navegador web;
- Exibir na interface web a temperatura interna do microcontrolador RP2040;
- Exibir mensagens de depuração no terminal USB.

Componentes esperados:

- LED físico ligado ao GPIO 13;
- Uso do ADC canal 4 para leitura da temperatura;
- Página HTML dinâmica com `snprintf()`;
- Servidor TCP rodando na porta 80;
- Respostas HTTP com `tcp_write()`;
- Debug no terminal via `printf()` usando `pico_stdio_usb`.

- Nome do arquivo principal: “Atividade_02.c” → Na pasta src.
- Vídeo mostrando o seu funcionamento
  Link: https://youtu.be/vkvBspjkuT0
- GitHub:
  https://github.com/ManoelFelipe/Embarcatech_37/tree/main/Unidade_02/Cap_02/Atividade_02

Organização do projeto: Código foi modularizado.
Na Pasta src:

- Atividade_02.c (arquivo principal com main())
- wifi_ap.h / wifi_ap.c (gerenciamento da rede WiFi AP)
- temperature.h / temperature.c (leitura de temperatura)
- led_control.h / led_control.c (controle do LED)
- web_server.h / web_server.c (servidor HTTP)
- debug.h / debug.c (funções de depuração)

Na Pasta libs:

- Servidor DHCP: dhcpserver/dhcpserver.c e dhcpserver.h
- Servidor DNS: dnsserver/dnsserver.c e dnsserver.h
- lwipopts.h - Arquivo de configuração da pilha de rede LwIP para o Raspberry Pi Pico W.

Benefícios desta abordagem:

- Main enxuta: A função principal fica clara e fácil de entender.
- Modularidade: Cada funcionalidade está separada em seu próprio módulo.
- Reusabilidade: Os módulos podem ser usados em outros projetos.
- Manutenibilidade: Fácil de fazer alterações em áreas específicas.
- Testabilidade: Cada módulo pode ser testado isoladamente.

Como entender o projeto:

1. Comece pelo main() em Atividade_02.c para ver a ordem em que tudo acontece.
2. Vá abrindo os headers .h para saber o que cada módulo oferece.
3. Leia os .c se quiser entender os detalhes internos.
4. Todos os arquivos estão comentados com Doxygen em português para facilitar.

- Código: src/Atividade_02_.c

```
/**
 * @file    Atividade_02.c
 * @brief   Aplicação principal que cria um ponto-de-acesso Wi-Fi, publica uma
 * página HTTP para controlar um LED e exibir a temperatura interna do RP2040.
 *
 * @details Este projeto demonstra uma arquitetura de software modular para sistemas
 * embarcados usando o Raspberry Pi Pico W. As funcionalidades de hardware,
 * rede e depuração são encapsuladas em módulos distintos para manter a
 * função main() limpa e focada na lógica da aplicação.
 *
```

```c
 * @author  Manoel Furtado
 * @date    08 de Junho de 2025
 * @copyright 2025 Manoel Furtado (MIT License) (veja LICENSE.md)
 */

#include <stdio.h>           // Inclui a biblioteca padrão de Entrada/Saída (para usar printf).
#include "pico/stdlib.h"     // Inclui a biblioteca principal do SDK do Pico (funções de hardware).
#include "wifi_ap.h"         // Inclui o módulo que gerencia o ponto de acesso Wi-Fi.
#include "led_control.h"     // Inclui o módulo para controlar o estado do LED.
#include "temperature.h"     // Inclui o módulo para ler o sensor de temperatura interno.
#include "web_server.h"      // Inclui o módulo do servidor HTTP que serve a página web.
#include "debug.h"           // Inclui o módulo com funções de ajuda para depuração.

/* ====== Configurações gerais da aplicação ============================== */

/** @brief Define o pino GPIO ao qual o LED está conectado. */
#define APP_LED_GPIO          13

/** @brief Define o número de amostras a serem lidas do ADC para calcular a média da temperatura.
 * @details Um número maior de amostras resulta em uma leitura mais estável, mas leva mais tempo. */
#define APP_TEMP_SAMPLES      64

/** @brief Define o intervalo, em milissegundos, para a exibição de mensagens de depuração no console. */
#define APP_DEBUG_PERIOD_MS   1000

/** @brief Define o SSID (nome da rede) para o ponto de acesso Wi-Fi que será criado. */
#define APP_WIFI_SSID         "picow_test"

/** @brief Define a senha para o ponto de acesso Wi-Fi. */
#define APP_WIFI_PASSWORD     "password"

/** @brief Define a porta na qual o servidor HTTP irá escutar por conexões. A porta 80 é a padrão para HTTP. */
#define APP_HTTP_PORT         80

/* ====== main() ========================================================= */

/**
 * @brief Ponto de entrada principal da aplicação.
 * @details A função realiza três etapas principais:
 * 1. Inicializa todos os subsistemas (LED, temperatura, Wi-Fi, servidor web).
 * 2. Entra em um loop infinito para processar eventos de rede e comandos do usuário.
 * 3. Realiza uma finalização ordenada quando solicitado.
 * @return int Retorna 0 em caso de sucesso e 1 em caso de falha na inicialização.
 */
int main(void)
{
    // Inicializa a E/S padrão (para que `printf` funcione, geralmente via USB-serial).
    stdio_init_all();
```

```c
printf("\n=== Atividade 02 - aplicação modular ===\n");

/* 1. Inicialização dos módulos ----------------------------------- */

// Inicializa o módulo de controle do LED no GPIO definido, com o estado inicial desligado.
led_init(APP_LED_GPIO, /*initial_state=*/false);

// Inicializa o módulo de temperatura, especificando o número de amostras para a média.
temperature_init(/*num_samples=*/APP_TEMP_SAMPLES);

// Inicializa o modo Access Point (AP) do Wi-Fi. Se falhar, encerra a aplicação.
if (!wifi_ap_init(APP_WIFI_SSID, APP_WIFI_PASSWORD))  {
    return 1;
}

// Inicia o servidor web na porta configurada. Se falhar, encerra a aplicação.
if (!web_server_start(APP_HTTP_PORT)) {
    return 1;
}

// Configura um temporizador para a próxima mensagem de depuração periódica.
absolute_time_t next_dbg = make_timeout_time_ms(APP_DEBUG_PERIOD_MS);

/* 2. Loop principal ---------------------------------------------- */

// O loop continua enquanto não houver uma solicitação de desligamento.
while (!wifi_ap_must_shutdown()) {

    // Funções de "polling" que devem ser chamadas repetidamente para que a pilha de
    // rede (lwIP) e o driver do Wi-Fi (cyw43) processem eventos pendentes.
    // Essencial para o funcionamento da rede em modo não-bloqueante.
    wifi_ap_poll();
    web_server_poll();

    // Bloco para exibir mensagens de depuração em intervalos regulares.
    // `absolute_time_diff_us` verifica se o tempo `next_dbg` já foi alcançado.
    if (absolute_time_diff_us(get_absolute_time(), next_dbg) < 0) {
        debug_status("PERIODIC");
        // Agenda a próxima exibição para o futuro, mantendo o intervalo preciso.
        next_dbg = delayed_by_ms(next_dbg, APP_DEBUG_PERIOD_MS);
    }

    // Verifica se o caractere 'q' ou 'Q' foi digitado no console serial.
    // `getchar_timeout_us(0)` é uma forma não-bloqueante de ler um caractere.
    int c = getchar_timeout_us(0);
    if (c == 'q' || c == 'Q') {
        // Se 'q' foi pressionado, solicita o desligamento do ponto de acesso,
        // o que fará com que o loop principal termine.
```

```
            wifi_ap_request_shutdown();
        }

        // Uma pequena pausa para evitar que o loop consuma 100% da CPU.
        // Isso permite que o processador economize energia e lide com outras tarefas de baixa prioridade.
        sleep_ms(10);
    }

    /* 3. Finalização ordenada ----------------------------------------- */

    // Para o servidor web, deixando de aceitar novas conexões.
    web_server_stop();
    // Desativa a interface Wi-Fi e libera os recursos.
    wifi_ap_deinit();

    printf("Encerrado.\n");
    return 0;
}
```

- Código: src/led_control.c

```
/**
 * @file    led_control.c
 * @brief   Implementação do módulo de controle de LED.
 * @details Contém a lógica para manipular um pino GPIO como uma saída digital
 * para acender ou apagar um LED.
 */

#include "led_control.h"
#include "pico/stdlib.h"

/**
 * @brief Variável estática para armazenar o número do pino GPIO do LED.
 * @details Sendo `static`, esta variável só é visível dentro deste arquivo.
 * O valor 25 é o padrão para o LED on-board do Pico (não o Pico W).
 * A função led_init() irá atualizar este valor.
 */
static int led_gpio = 25;

/**
 * @brief Inicializa o pino GPIO para controlar o LED.
 * @param gpio O número do pino GPIO ao qual o LED está conectado.
 * @param initial_state O estado inicial do LED (true para ligado, false para desligado).
 */
void led_init(int gpio, bool initial_state)
{
    // Armazena o pino GPIO fornecido para uso por outras funções do módulo.
    led_gpio = gpio;
```

```
    // Inicializa o pino GPIO.
    gpio_init(led_gpio);
    // Configura a direção do pino como saída (OUTPUT).
    gpio_set_dir(led_gpio, GPIO_OUT);
    // Define o estado inicial do LED.
    led_set(initial_state);
}

/**
 * @brief Define o estado do LED.
 * @param on `true` para ligar o LED, `false` para desligá-lo.
 */
void led_set(bool on) {
    // `gpio_put` define o nível lógico do pino. `on` (true) corresponde ao nível alto (3.3V).
    gpio_put(led_gpio, on);
}

/**
 * @brief Obtém o estado atual do LED.
 * @return `true` se o LED estiver ligado, `false` caso contrário.
 */
bool led_get(void) {
    // `gpio_get` lê o último valor que foi escrito no pino de saída.
    return gpio_get(led_gpio);
}
```

- Código: src/led_control.h

```
/**
 * @file    led_control.h
 * @brief   Interface pública para o módulo de controle de um LED.
 * @details Este header declara as funções para inicializar, ligar/desligar
 * e obter o estado de um LED conectado a um pino GPIO.
 */

#ifndef LED_CONTROL_H
#define LED_CONTROL_H

#include <stdbool.h> // Para usar o tipo `bool` (true/false)

/**
 * @brief Inicializa o pino GPIO para controlar o LED.
 * @param gpio O número do pino GPIO ao qual o LED está conectado.
 * @param initial_state O estado inicial do LED (true para ligado, false para desligado).
 */
void  led_init(int gpio, bool initial_state);
```

```c
/**
 * @brief Define o estado do LED.
 * @param on `true` para ligar o LED, `false` para desligá-lo.
 */
void  led_set (bool on);

/**
 * @brief Obtém o estado atual do LED.
 * @return `true` se o LED estiver ligado, `false` caso contrário.
 */
bool  led_get (void);

#endif // LED_CONTROL_H
```

- Código: src/temperature.c

```c
/**
 * @file    temperature.c
 * @brief   Implementação do módulo de leitura do sensor de temperatura.
 * @details Este módulo utiliza o ADC do RP2040 para ler a voltagem do sensor
 * de temperatura interno e a converte para graus Celsius usando uma
 * fórmula fornecida no datasheet do microcontrolador.
 */

#include "temperature.h"
#include "pico/stdlib.h"
#include "hardware/adc.h"

/// @brief Armazena o número de amostras a serem lidas para o cálculo da média.
static int   samples = 1;
/// @brief Um valor de offset que pode ser usado para calibrar o sensor, se necessário.
static float user_offset = 0.0f;

/**
 * @brief Inicializa o hardware necessário para a leitura da temperatura.
 * @param num_samples O número de leituras do ADC para calcular a média.
 */
void temperature_init(int num_samples)
{
    // Garante que o número de amostras seja pelo menos 1.
    samples = (num_samples > 0) ? num_samples : 1;

    // Inicializa o subsistema ADC.
    adc_init();
    // Habilita o sensor de temperatura, que está multiplexado com os pinos do ADC.
    adc_set_temp_sensor_enabled(true);
    // Seleciona o canal 4 do ADC, que é o canal conectado ao sensor de temperatura.
    adc_select_input(4);
```

```c
}

/**
 * @brief Lê a temperatura atual do sensor interno.
 * @return A temperatura medida em graus Celsius (°C).
 */
float temperature_read_c(void)
{
    // Constantes para a conversão, baseadas no datasheet do RP2040.
    const float VREF = 3.3f;        // Tensão de referência do ADC é 3.3V.
    const float conv = VREF / (1 << 12); // Fator de conversão de valor ADC (12-bit) para voltagem.

    // Acumula a soma das leituras do ADC.
    uint32_t sum = 0;
    for (int i = 0; i < samples; ++i) {
        sum += adc_read();
        sleep_us(5); // Pequena pausa entre leituras.
    }

    // Calcula a voltagem média.
    float voltage = (sum / (float)samples) * conv;

    // Fórmula de conversão de voltagem para temperatura em Celsius, conforme datasheet do RP2040.
    // A fórmula é: T = 27 - (V_be - 0.706) / 0.001721
    // Onde 27°C é a temperatura de referência, e 0.706V é a voltagem medida a 27°C.
    return 27.0f - (voltage - 0.706f) / 0.001721f + user_offset;
}
```

- temperature.h

```c
/**
 * @file    temperature.h
 * @brief   Interface pública para o módulo de leitura do sensor de temperatura interno.
 * @details Declara as funções para inicializar o conversor analógico-digital (ADC)
 * e ler a temperatura em graus Celsius.
 */

#ifndef TEMPERATURE_H
#define TEMPERATURE_H

/**
 * @brief Inicializa o hardware necessário para a leitura da temperatura.
 * @details Configura o ADC e habilita o sensor de temperatura interno do RP2040.
 * @param num_samples O número de leituras do ADC a serem feitas para calcular uma média.
 * Isso ajuda a reduzir o ruído e obter um valor mais estável.
 */
void   temperature_init(int num_samples);
```

```c
/**
 * @brief Lê a temperatura atual do sensor interno.
 * @return A temperatura medida em graus Celsius (°C).
 */
float  temperature_read_c(void);

#endif // TEMPERATURE_H
```

- debug.c

```c
/**
 * @file    debug.c
 * @brief   Implementação das funções de depuração.
 * @details Centraliza a lógica de formatação de mensagens de status para
 * facilitar a depuração da aplicação.
 */

#include "debug.h"
#include "led_control.h"
#include "temperature.h"
#include <stdio.h>

/**
 * @brief Imprime uma linha de status formatada no console.
 * @param tag Uma string para identificar o contexto da mensagem de depuração.
 */
void debug_status(const char *tag)
{
    // `printf` para formatar e enviar a string para a saída padrão (serial-USB).
    // `led_get() ? "ON" : "OFF"` é um operador ternário que escolhe a string "ON" ou "OFF"
    // com base no retorno da função `led_get()`.
    printf("[%s] LED=%s | Temp=%.2f °C\n",
           tag,
           led_get() ? "ON" : "OFF",
           temperature_read_c());
}
```

- debug.h

```c
/**
 * @file    debug.h
 * @brief   Interface pública para funções de depuração.
 * @details Declara funções úteis para imprimir informações de status no console.
 */

#ifndef DEBUG_H
#define DEBUG_H
```

```c
/**
 * @brief Imprime uma linha de status formatada no console.
 * @details Exibe o estado atual do LED e a temperatura, prefixados por uma tag.
 * @param tag Uma string para identificar o contexto da mensagem de depuração (ex: "HTTP", "PERIODIC").
 */
void debug_status(const char *tag);

#endif // DEBUG_H
```

- web_server.c

```c
/**
 * @file    web_server.c
 * @brief   Implementação de um servidor HTTP simples usando a pilha lwIP.
 * @details Este módulo gerencia conexões TCP, analisa requisições HTTP GET básicas,
 * e serve uma página HTML dinâmica que permite ao usuário interagir com
 * o dispositivo (controlar LED e ver temperatura).
 */

#include "web_server.h"
#include "led_control.h"
#include "temperature.h"
#include "debug.h"
#include "lwip/tcp.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

/* ---------- página HTML ------------------------------------------------*/

/**
 * @brief Gera dinamicamente o conteúdo da página HTML.
 * @param buf Ponteiro para o buffer onde a página HTML será escrita.
 * @param len Tamanho máximo do buffer.
 * @return O número de caracteres escritos no buffer (excluindo o terminador nulo).
 */
static int make_page(char *buf, size_t len)
{
    // Obtém o estado atual do LED e da temperatura.
    bool  on  = led_get();
    float t   = temperature_read_c();

    // Define strings que mudarão com base no estado do LED.
    const char *state_cls  = on ? "state-on"  : "state-off";    // Classe CSS para o status.
    const char *button_txt = on ? "Desligar LED" : "Ligar LED"; // Texto do botão.
    const char *param      = on ? "off" : "on";                 // Parâmetro no link do botão.

    // `snprintf` é usado para construir a string HTML de forma segura, evitando buffer overflow.
```

```
    // A página contém CSS embutido para estilização e placeholders (%s, %.2f) que são
    // substituídos pelos valores dinâmicos.
    return snprintf(buf, len,
      "<!DOCTYPE html><html><head><meta charset=\"utf-8\">"
      "<meta name=\"viewport\" content=\"width=device-width,initial-scale=1\">"
      "<title>Pico W</title>"
      "<style>"
        "body{font-family:sans-serif;text-align:center;margin-top:40px;background:#f2f2f2;}"
        "span.temp{color:#0D47A1;font-weight:bold;}"
        ".state-on{background:#8BC34A;color:#000;}"
        ".state-off{background:#EF5350;color:#000;}"
        "button{padding:14px 24px;font-size:18px;border:0;border-radius:10px;cursor:pointer;}"
      "</style></head><body>"
      "<h1>Manoel_Atividade 02_Und. 02</h1>"
      "<h2>Servidor HTTP Pico W</h2>"
      "<p>Temperatura interna: <span class=\"temp\">%.2f °C</span></p>"
      "<p>Status do LED: <span class=\"%s\">%s</span></p>"
      "<p><a href=\"/?led=%s\"><button class=\"%s\">%s</button></a></p>"
      "</body></html>",
      t, state_cls, on ? "ON" : "OFF", param, state_cls, button_txt);
}

/* ---------- estado do cliente ------------------------------------------*/

/**
 * @struct client_t
 * @brief  Estrutura para armazenar o estado de uma conexão de cliente individual.
 */
typedef struct {
    struct tcp_pcb *pcb;    ///< Ponteiro para o Bloco de Controle de Protocolo (PCB) do lwIP.
    char            hdr[128]; ///< Buffer para os cabeçalhos HTTP.
    char            body[1024];///< Buffer para o corpo da página HTML.
    int             hdr_len, body_len, sent; ///< Comprimentos e contagem de bytes enviados.
} client_t;

/* ---------- forward declarations ---------------------------------------*/
// Declarações antecipadas das funções de callback para que possam ser usadas antes de suas definições.
static err_t  on_accept(void *arg, struct tcp_pcb *new_pcb, err_t err);
static err_t  on_recv  (void *arg, struct tcp_pcb *pcb, struct pbuf *p, err_t err);
static err_t  on_sent  (void *arg, struct tcp_pcb *pcb, u16_t len);
static err_t  on_poll  (void *arg, struct tcp_pcb *pcb);
static void   close_cli(struct tcp_pcb *pcb, client_t *st);

/* ---------- servidor ---------------------------------------------------*/

/// @brief PCB global para o servidor de escuta (listening).
static struct tcp_pcb *srv_pcb = NULL;
```

```c
/**
 * @brief Inicia o servidor HTTP.
 * @param port Porta TCP para escutar.
 * @return `true` em sucesso, `false` em falha.
 */
bool web_server_start(int port)
{
    // Cria um novo PCB para TCP em qualquer tipo de endereço IP (IPv4 ou IPv6).
    srv_pcb = tcp_new_ip_type(IPADDR_TYPE_ANY);
    if (!srv_pcb) return false;

    // Associa (bind) o PCB a qualquer endereço IP local e à porta especificada.
    if (tcp_bind(srv_pcb, IP_ANY_TYPE, port) != ERR_OK) {
        tcp_close(srv_pcb);
        return false;
    }

    // Coloca o servidor no estado de escuta (LISTEN), com um backlog de 4 conexões pendentes.
    srv_pcb = tcp_listen_with_backlog(srv_pcb, 4);

    // Registra a função `on_accept` para ser chamada quando um novo cliente se conectar.
    // Este é o coração do modelo de programação assíncrono do lwIP.
    tcp_accept(srv_pcb, on_accept);
    printf("[HTTP] Escutando na porta %d\n", port);
    return true;
}

/**
 * @brief Para o servidor HTTP.
 */
void web_server_stop(void)
{
    if (srv_pcb) {
        tcp_close(srv_pcb);
        srv_pcb = NULL;
    }
}

/**
 * @brief Função de poll para o servidor.
 */
void web_server_poll(void) { /* Vazio, pois tudo é gerenciado por callbacks. */ }

/* ---------- callbacks ----------------------------------------------------*/

/**
 * @brief Callback chamado quando uma nova conexão TCP é aceita.
 * @param arg Argumento opcional (não usado aqui).
```

```
 * @param pcb O PCB da nova conexão.
 * @param err Código de erro.
 * @return ERR_OK em sucesso.
 */
static err_t on_accept(void *arg, struct tcp_pcb *pcb, err_t err)
{
    if (err != ERR_OK || !pcb) return ERR_VAL;

    // Aloca memória para a estrutura de estado do novo cliente.
    client_t *st = calloc(1, sizeof(client_t));
    if (!st) return ERR_MEM;

    // Registra os callbacks para esta conexão específica.
    tcp_arg (pcb, st);       // Associa o estado `st` a esta conexão.
    tcp_recv(pcb, on_recv);  // Função a ser chamada quando dados forem recebidos.
    tcp_sent(pcb, on_sent);  // Função a ser chamada quando dados forem enviados com sucesso.
    tcp_poll(pcb, on_poll, 10); // Função a ser chamada periodicamente.
    printf("[HTTP] Cliente %s conectado\n", ipaddr_ntoa(&pcb->remote_ip));
    return ERR_OK;
}

/**
 * @brief Callback chamado quando dados são recebidos de um cliente.
 * @param arg Estado do cliente (`client_t`).
 * @param pcb PCB da conexão.
 * @param p  Buffer (`pbuf`) com os dados recebidos. Se NULL, o cliente fechou a conexão.
 * @param err Código de erro.
 * @return ERR_OK em sucesso.
 */
static err_t on_recv(void *arg, struct tcp_pcb *pcb, struct pbuf *p, err_t err)
{
    client_t *st = arg;
    // Se p for NULL, o cliente fechou a conexão.
    if (!p) {
        close_cli(pcb, st);
        return ERR_OK;
    }

    /* Análise super simples da requisição GET */
    char req[64] = {0};
    pbuf_copy_partial(p, req, sizeof req - 1, 0); // Copia o início da requisição para um buffer local.
    pbuf_free(p); // Libera o buffer de recepção.

    // Procura pelo parâmetro "?led=" na requisição.
    char *q = strstr(req, "?led=");
    if (q) {
        q += 5; // Avança o ponteiro para depois de "?led=".
        if (!strncmp(q, "on", 2)) {
```

```
            led_set(true); // Se o valor for "on", liga o LED.
        } else if (!strncmp(q, "off", 3)) {
            led_set(false); // Se for "off", desliga o LED.
        }
    }

    // Prepara a resposta HTTP.
    st->body_len = make_page(st->body, sizeof st->body);
    st->hdr_len  = snprintf(st->hdr, sizeof st->hdr,
        "HTTP/1.1 200 OK\r\nContent-Length: %d\r\nContent-Type: text/html\r\nConnection: close\r\n\r\n",
        st->body_len);

    // Envia o cabeçalho e o corpo da resposta.
    st->sent = 0;
    tcp_write(pcb, st->hdr,  st->hdr_len, 0);
    tcp_write(pcb, st->body, st->body_len, TCP_WRITE_FLAG_COPY);

    // Imprime uma mensagem de depuração indicando que uma requisição foi tratada.
    debug_status("HTTP");
    return ERR_OK;
}

/**
 * @brief Callback chamado quando o envio de dados é confirmado pelo cliente (ACK).
 * @param arg Estado do cliente.
 * @param pcb PCB da conexão.
 * @param len Número de bytes confirmados.
 * @return ERR_OK em sucesso.
 */
static err_t on_sent(void *arg, struct tcp_pcb *pcb, u16_t len)
{
    client_t *st = arg;
    st->sent += len;
    // Se todos os dados (cabeçalho + corpo) foram enviados, fecha a conexão.
    if (st->sent >= st->hdr_len + st->body_len) {
        close_cli(pcb, st);
    }
    return ERR_OK;
}

/**
 * @brief Callback de poll, chamado periodicamente. Usado como um timeout.
 * @details Se esta função for chamada, significa que a conexão está ociosa.
 * Neste caso, optamos por fechá-la para liberar recursos.
 * @param arg Estado do cliente.
 * @param pcb PCB da conexão.
 * @return ERR_OK.
 */
```

```c
static err_t on_poll(void *arg, struct tcp_pcb *pcb) {
    close_cli(pcb, arg);
    return ERR_OK;
}

/**
 * @brief Fecha uma conexão de cliente e libera os recursos associados.
 * @param pcb PCB da conexão a ser fechada.
 * @param st Estado do cliente a ser liberado.
 */
static void close_cli(struct tcp_pcb *pcb, client_t *st)
{
    if (pcb) {
        // Desregistra todos os callbacks para evitar chamadas futuras em um PCB inválido.
        tcp_arg (pcb, NULL);
        tcp_err(pcb, NULL);
        tcp_recv(pcb, NULL);
        tcp_sent(pcb, NULL);
        tcp_poll(pcb, NULL, 0);
        // Fecha a conexão TCP.
        tcp_close(pcb);
    }
    // Libera a memória da estrutura de estado do cliente.
    free(st);
}
```

- web_server.h

```c
/**
 * @file    web_server.h
 * @brief   Interface pública para o módulo de servidor web.
 * @details Declara as funções para iniciar, parar e pollar o servidor HTTP.
 */

#ifndef WEB_SERVER_H
#define WEB_SERVER_H

#include <stdbool.h>

/**
 * @brief Inicia o servidor HTTP e começa a escutar por conexões.
 * @param port O número da porta TCP na qual o servidor irá operar.
 * @return `true` se o servidor foi iniciado com sucesso, `false` caso contrário.
 */
bool  web_server_start(int port);

/**
 * @brief Processa eventos de rede pendentes.
```

```
 * @details Esta função deve ser chamada periodicamente no loop principal da aplicação
 * para permitir que a pilha de rede processe pacotes recebidos e enviados.
 * Neste caso, com callbacks, ela pode estar vazia, mas é uma boa prática mantê-la.
 */
void  web_server_poll(void);

/**
 * @brief Para o servidor HTTP e libera os recursos associados.
 */
void  web_server_stop(void);

#endif // WEB_SERVER_H
```

- wifi_ap.c

```
/**
 * @file    wifi_ap.c
 * @brief   Implementação do gerenciador do Ponto de Acesso (AP) Wi-Fi.
 * @details Este módulo coordena a inicialização do hardware Wi-Fi (CYW43),
 * a configuração do modo AP, e a inicialização dos servidores DHCP e DNS
 * que são essenciais para uma rede funcional.
 */

#include "wifi_ap.h"
#include "pico/cyw43_arch.h" // Funções de arquitetura para o chip Wi-Fi CYW43
#include "dhcpserver.h"      // Interface do servidor DHCP
#include "dnsserver.h"       // Interface do servidor DNS
#include <stdio.h>

/**
 * @brief Estrutura estática para manter o estado dos serviços de rede.
 * @details Sendo `static`, esta estrutura é privada a este arquivo. Ela contém
 * as instâncias dos servidores DHCP e DNS, e um flag booleano para
 * controlar o desligamento (shutdown) da aplicação.
 */
static struct {
    dhcp_server_t dhcp;     ///< Instância do estado do servidor DHCP.
    dns_server_t  dns;      ///< Instância do estado do servidor DNS.
    bool          shutdown; ///< Flag que indica se o desligamento foi solicitado.
} net = {0};

/**
 * @brief Inicializa e configura o modo Access Point (AP).
 * @param ssid O nome da rede a ser criada.
 * @param password A senha da rede.
 * @return `true` em sucesso, `false` em falha.
 */
bool wifi_ap_init(const char *ssid, const char *password)
```

```
{
    // Inicializa o chip CYW43. Se falhar, é um erro crítico.
    if (cyw43_arch_init()) {
        puts("[WiFi] ERRO init");
        return false;
    }

    // `cyw43_arch_lwip_begin()` e `..._end()` funcionam como um lock/mutex.
    // Garantem que as operações na pilha de rede lwIP sejam atômicas e seguras
    // em relação a interrupções ou outros contextos de execução.
    cyw43_arch_lwip_begin();

    // Habilita o modo Access Point com o SSID, senha e tipo de autenticação especificados.
    cyw43_arch_enable_ap_mode(ssid, password, CYW43_AUTH_WPA2_AES_PSK);

    // Define os endereços de rede para o nosso Ponto de Acesso.
    // O Pico W terá o IP estático 192.168.4.1 e atuará como gateway.
    ip4_addr_t gw, mask;
    IP4_ADDR(&gw,   192, 168, 4, 1);
    IP4_ADDR(&mask, 255, 255, 255, 0);

    // Inicializa o servidor DHCP. Ele será responsável por distribuir IPs
    // na rede 192.168.4.0/24.
    dhcp_server_init(&net.dhcp, &gw, &mask);

    // Inicializa o servidor DNS. Ele responderá a todas as consultas DNS
    // com o endereço do gateway (nosso próprio IP).
    dns_server_init (&net.dns,  &gw);

    cyw43_arch_lwip_end(); // Libera o "lock" da pilha de rede.

    printf("[WiFi] AP \"%s\" ativo em %s\n", ssid, ipaddr_ntoa(&gw));
    return true;
}

/**
 * @brief Processa eventos de rede pendentes.
 */
void wifi_ap_poll(void)
{
// A diretiva #if PICO_CYW43_ARCH_POLL verifica se o modo de polling está
// habilitado no build. Se sim, a função cyw43_arch_poll() deve ser
// chamada periodicamente para que o driver Wi-Fi funcione corretamente.
#if PICO_CYW43_ARCH_POLL
    cyw43_arch_poll();
#endif
}
```

```c
/**
 * @brief Solicita o desligamento do Ponto de Acesso.
 */
void wifi_ap_request_shutdown(void) { net.shutdown = true; }

/**
 * @brief Verifica se o desligamento foi solicitado.
 * @return true se a flag de desligamento estiver ativa.
 */
bool wifi_ap_must_shutdown(void)    { return net.shutdown; }

/**
 * @brief Desinicializa todos os serviços de rede e o hardware Wi-Fi.
 */
void wifi_ap_deinit(void)
{
    cyw43_arch_lwip_begin(); // Adquire o "lock" para operações de rede.

    // Desliga os servidores em ordem: primeiro os de aplicação (DNS, DHCP),
    // depois o modo de rede do chip.
    dns_server_deinit (&net.dns);
    dhcp_server_deinit(&net.dhcp);
    cyw43_arch_disable_ap_mode();

    cyw43_arch_lwip_end(); // Libera o "lock".

    // Desinicializa completamente o chip CYW43.
    cyw43_arch_deinit();
}
```

- wifi_ap.h

```c
/**
 * @file    wifi_ap.h
 * @brief   Interface pública para o módulo de gerenciamento do Ponto de Acesso (AP) Wi-Fi.
 * @details Este header declara as funções para inicializar, gerenciar o ciclo de vida e
 * finalizar o modo Access Point do chip Wi-Fi. Ele serve como uma camada de
 * abstração de alto nível sobre as funções do SDK do Pico W e os servidores
 * de rede (DHCP e DNS).
 */

#ifndef WIFI_AP_H
#define WIFI_AP_H

#include <stdbool.h>

/**
 * @brief Inicializa o chip Wi-Fi e o configura em modo Access Point (AP).
```

```
 * @details Esta função inicializa o hardware CYW43, ativa o modo AP com o SSID e
 * senha fornecidos, e inicializa os servidores DHCP e DNS necessários para
 * que os clientes possam se conectar e obter um endereço IP.
 * @param ssid O nome da rede (SSID) a ser criada.
 * @param password A senha da rede Wi-Fi (WPA2-AES-PSK).
 * @return `true` se o ponto de acesso foi iniciado com sucesso, `false` caso contrário.
 */
bool  wifi_ap_init (const char *ssid, const char *password);

/**
 * @brief Processa eventos de rede pendentes do driver Wi-Fi.
 * @details Esta função deve ser chamada repetidamente no loop principal da aplicação,
 * especialmente quando se utiliza o modo de polling (`PICO_CYW43_ARCH_POLL`).
 * Ela permite que o driver Wi-Fi execute tarefas de baixo nível, como
 * o envio e recebimento de pacotes.
 */
void  wifi_ap_poll (void);

/**
 * @brief Desativa o modo AP e desinicializa o hardware Wi-Fi.
 * @details Realiza uma finalização ordenada, desligando os servidores DNS e DHCP
 * e, em seguida, desativando a interface Wi-Fi para economizar energia e
 * liberar recursos.
 */
void  wifi_ap_deinit(void);

/**
 * @brief Verifica se uma solicitação de desligamento foi feita.
 * @details Usado no loop principal (`while`) para determinar se a aplicação deve
 * continuar executando ou iniciar o processo de finalização.
 * @return `true` se o desligamento foi solicitado, `false` caso contrário.
 */
bool  wifi_ap_must_shutdown(void);

/**
 * @brief Sinaliza que a aplicação deve ser encerrada.
 * @details Esta função pode ser chamada a qualquer momento (por exemplo, a partir de
 * uma interrupção ou de um comando do usuário via serial) para solicitar
 * um desligamento limpo da rede.
 */
void  wifi_ap_request_shutdown(void);

#endif // WIFI_AP_H
```

- CMakeLists.txt

```
# Generated Cmake Pico project file
```

```
cmake_minimum_required(VERSION 3.13)

set(CMAKE_C_STANDARD 11)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_EXPORT_COMPILE_COMMANDS ON)

# Initialise pico_sdk from installed location
# (note this can come from environment, CMake cache etc)

# == DO NOT EDIT THE FOLLOWING LINES for the Raspberry Pi Pico VS Code Extension to work ==
if(WIN32)
set(USERHOME $ENV{USERPROFILE})
else()
set(USERHOME $ENV{HOME})
endif()
set(sdkVersion 2.1.1)
set(toolchainVersion 14_2_Rel1)
set(picotoolVersion 2.1.1)
set(picoVscode ${USERHOME}/.pico-sdk/cmake/pico-vscode.cmake)
if (EXISTS ${picoVscode})
include(${picoVscode})
endif()
# ====================================================================================
set(PICO_BOARD pico_w CACHE STRING "Board type")

# Pull in Raspberry Pi Pico SDK (must be before project)
include(pico_sdk_import.cmake)

project(Atividade_02 C CXX ASM)

# Initialise the Raspberry Pi Pico SDK
pico_sdk_init()

# Add executable. Default name is the project name, version 0.1

add_executable(Atividade_02
src/Atividade_02.c
src/temperature.c
src/web_server.c
src/wifi_ap.c
src/led_control.c
src/debug.c
libs/dhcpserver/dhcpserver.c
libs/dnsserver/dnsserver.c
)

pico_set_program_name(Atividade_02 "Atividade_02")
pico_set_program_version(Atividade_02 "0.1")
```

```
# Modify the below lines to enable/disable output over UART/USB
pico_enable_stdio_uart(Atividade_02 0)
pico_enable_stdio_usb(Atividade_02 01)

# Add the standard library to the build
target_link_libraries(Atividade_02
pico_stdlib
pico_cyw43_arch_lwip_threadsafe_background
pico_stdio_usb
hardware_gpio
hardware_adc
hardware_i2c
)

# Add the standard include files to the build
target_include_directories(Atividade_02 PRIVATE
${CMAKE_CURRENT_LIST_DIR}
#${CMAKE_CURRENT_LIST_DIR}/..
#${CMAKE_CURRENT_LIST_DIR}/src
${CMAKE_CURRENT_LIST_DIR}/libs
${CMAKE_CURRENT_LIST_DIR}/libs/dhcpserver
${CMAKE_CURRENT_LIST_DIR}/libs/dnsserver
)

pico_configure_ip4_address(Atividade_02 PRIVATE
CYW43_DEFAULT_IP_AP_ADDRESS 192.168.4.1
)

pico_add_extra_outputs(Atividade_02)
```