Aluno do Embarcatech_37 no IFMA

Nome: Manoel Felipe Costa Furtado

Matrícula: 20251RSE.MTC0086

Atividade 08 – Referente ao capítulo 8 da unidade 01 – ETHERNET\WIFI

Prazo dia 25/05/2025 as 23:59

Enunciado: Simulador Portátil de Alarme para Treinamentos de Brigadas e Evacuação

Este trabalho propõe o desenvolvimento de uma ferramenta portátil, baseada no microcontrolador Raspberry Pi Pico W, destinada a simulações de emergência e treinamentos de evacuação. O sistema opera em modo Access Point, permitindo que o instrutor ative remotamente o modo "ALARME" por meio de um dispositivo móvel conectado à rede Wi-Fi local. Uma vez acionado, o sistema emite sinais visuais e sonoros — com LED piscante, buzzer ativo — e exibe, no display OLED, a mensagem "EVACUAR", simulando uma situação real de emergência, sem depender de infraestrutura de rede externa.

- Nome do arquivo principal: "Atividade_08.c" → Na pasta src.
- Na pasta lib tem os arquivos sobre o display OLED.
- Vídeo mostrando o seu funcionamento
  Link: https://youtu.be/A4gO1GWlYl0
- GitHub:https://github.com/ManoelFelipe/Embarcatech_37/tree/main/Unidade_01/Cap_08/Atividade_08
- Drive: Atividade_08.zip

Como o código é muito grande eu fiz na estrutura de módulos para ajudar no entendimento:

- Atividade_08.c: Conterá a função main() principal, a inicialização básica do sistema e o loop principal do programa. Ele chamará funções dos outros módulos.
- app_config.h: Um arquivo de cabeçalho para centralizar todas as definições e constantes do projeto (pinos GPIO, credenciais Wi-Fi, mensagens, etc.).
- oled_display.h / oled_display.c: Módulo responsável por toda a interação com o display OLED SSD1306, incluindo inicialização e atualização de mensagens.

- alarm_control.h / alarm_control.c: Módulo para gerenciar a lógica do alarme, o estado dos LEDs (verde, vermelho, azul de status do AP) e do buzzer.
- network_manager.h / network_manager.c: Módulo para gerenciar a configuração do Wi-Fi em modo Access Point, os servidores DHCP e DNS, e toda a lógica do servidor TCP/HTTP para as requisições web.
- O CMakeLists.txt está todo organizado para atender a estrutura abaixo.

```
|-- src/
|   |-- Atividade_08.c  (onde está a função main)
|   |-- app_config.h
|   |-- oled_display.h
|   |-- oled_display.c
|   |-- alarm_control.h
|   |-- alarm_control.c
|   |-- network_manager.h
|   |-- network_manager.c
|-- lib/
|   |-- ssd1306/
|   |   |-- ssd1306.h
|   |   |-- ssd1306_i2c.h
|   |   |-- ssd1306_i2c.c
|   |   |-- ssd1306_font.h
|   |--lwipopts.h
|-- dhcpserver/
|   |-- dhcpserver.h
|   |-- dhcpserver.c
|-- dnsserver/
|   |-- dnsserver.h
|   |-- dnsserver.c
|-- CMakeLists.txt
|-- pico_sdk_import.cmake
```

- Código: Atividade_08.c

```c
/**
 * @file Atividade_08.c
 * @brief Simulador Portátil de Alarme para Treinamentos de Brigadas e Evacuação (Versão Modularizada)
 *
 * @objectives
 * - Configurar o Raspberry Pi Pico W como um ponto de acesso (Access Point) Wi-Fi.
 * - Iniciar servidores DHCP e DNS locais para permitir a conexão de dispositivos clientes.
 * - Criar um servidor HTTP embarcado que disponibiliza uma página HTML de controle.
 * - Permitir o controle remoto de um sistema de alarme (LEDs, Buzzer, Display OLED).
 * - Indicar estado do alarme via LED vermelho piscante e buzzer intermitente.
 * - Exibir mensagens de status ("EVACUAR", "Sistema em repouso") em um display OLED SSD1306.
 * - Usar LED verde para indicar sistema em repouso e LED azul para status do Access Point.
 * - Finalização controlada do modo Access Point via tecla 'd' no terminal serial.
 *
 * @pinout
 * Definido em app_config.h
 *
 * @author  Manoel Furtado
 * @date    25 maio 2025
 * @copyright 2025 Manoel Furtado (MIT License) (veja LICENSE.md)
 */

#include "pico/stdlib.h"     // Para stdio_init_all, etc.
#include "pico/cyw43_arch.h" // Para cyw43_arch_poll, etc.
#include <stdio.h>           // Para printf
#include <stdlib.h>          // Para calloc, free
```

```c
#include <assert.h>          // Para assert

#include "app_config.h"      // Configurações do projeto
#include "oled_display.h"    // Gerenciamento do display OLED
#include "alarm_control.h"   // Lógica do alarme e controle de LEDs/Buzzer
#include "network_manager.h" // Gerenciamento de rede (Wi-Fi AP, DHCP, DNS, HTTP Server)

// --- Variável Global para Estado do Servidor/Aplicação ---
// A estrutura TCP_SERVER_T é definida em network_manager.h
// Usamos um ponteiro para ela, alocado dinamicamente no main.
static TCP_SERVER_T *g_server_state = NULL;

/**
 * @brief Callback chamado quando um caractere está disponível na entrada serial (stdio).
 * Usado para detectar a tecla 'd' para desabilitar o Access Point e encerrar a aplicação.
 * @param param Ponteiro para a estrutura de estado do servidor TCP (TCP_SERVER_T).
 */
void key_pressed_func(void *param) {
    if (!param) {
        return;
    }
    TCP_SERVER_T *state = (TCP_SERVER_T*)param;
    int key = getchar_timeout_us(0); // Lê um caractere sem bloquear

    // Se a tecla 'd' ou 'D' for pressionada e o estado de complete ainda não foi setado
    if ((key == 'd' || key == 'D') && state && !state->complete) {
        printf("\nTecla 'd' pressionada. Preparando para encerrar...\n");
        // Apenas sinaliza para o loop principal que o programa deve terminar.
        // A desinicialização dos módulos ocorrerá no main, após o loop.
        state->complete = true;
    }
}

/**
 * @brief Função principal da aplicação.
 *
 * Inicializa todos os módulos, entra no loop principal para processar
 * eventos de rede e lógica do alarme, e depois desinicializa os módulos
 * ao encerrar.
 *
 * @return 0 se sucesso, 1 em caso de erro na inicialização.
 */
int main() {
    // Inicializa stdio para comunicação serial (printf, getchar)
    stdio_init_all();
    printf("Simulador Portatil de Alarme - Iniciando (Versão Modularizada)...\n");

    // Aloca memória para o estado do servidor/aplicação.
```

```
    // Este estado é compartilhado e modificado pela função key_pressed_func.
    g_server_state = calloc(1, sizeof(TCP_SERVER_T));
    if (!g_server_state) {
        DEBUG_printf("Falha ao alocar estado do servidor TCP.\n");
        return 1;
    }
    g_server_state->complete = false; // Garante estado inicial

    // Inicializa o módulo de controle do alarme (LEDs, Buzzer)
    alarm_control_init();

    // Inicializa o módulo do display OLED
    oled_display_init();

    // Atualiza o display OLED com o status inicial do alarme (que é inativo)
    oled_display_update_status(alarm_control_is_active());

    // Inicializa o gerenciador de rede (Wi-Fi AP, DHCP, DNS, Servidor HTTP)
    if (!network_manager_init(g_server_state)) {
        DEBUG_printf("Falha ao inicializar o gerenciador de rede.\n");
        // Se a rede falhar, desliga saídas de alarme e limpa OLED antes de sair.
        alarm_control_shutdown_outputs();
        oled_display_clear(); // Ou uma mensagem de erro no OLED
        free(g_server_state);
        g_server_state = NULL;
        return 1;
    }

    // Configura callback para ser notificado quando um caractere é pressionado no terminal serial
    stdio_set_chars_available_callback(key_pressed_func, g_server_state);
    printf("Pressione 'd' no terminal serial para desabilitar o Access Point e encerrar.\n");


    // Loop principal do programa
    while(g_server_state && !g_server_state->complete) {
        // Se estiver usando PICO_CYW43_ARCH_POLL, é necessário chamar cyw43_arch_poll()
        // periodicamente para processar eventos Wi-Fi e LwIP.
        #if PICO_CYW43_ARCH_POLL
        cyw43_arch_poll();
        #endif

        // Processa a lógica do alarme (piscar LEDs/Buzzer se ativo)
        alarm_control_process();

        // O PICO_CYW43_ARCH_POLL define se o driver Wi-Fi é baseado em polling ou interrupções.
        // Se baseado em polling, cyw43_arch_wait_for_work_until() pode ser usado para
        // economizar energia esperando por trabalho ou por um timeout.
        #if PICO_CYW43_ARCH_POLL
        // Espera por trabalho do Wi-Fi/LwIP ou até 10ms.
```

```
        // Um timeout menor permite que a lógica do alarme (piscar) seja mais responsiva
        // se não houver muita atividade de rede.
        cyw43_arch_wait_for_work_until(make_timeout_time_ms(10));
        #else
        // Se não estiver usando polling, o trabalho Wi-Fi/LwIP ocorre em background (interrupções).
        // Um pequeno sleep_ms pode ser usado para ceder tempo a outras tarefas, se houver.
        sleep_ms(10);
        #endif
    }

    // --- Encerramento da Aplicação ---
    printf("Encerrando aplicação...\n");

    // Desabilita o callback do stdio para evitar chamadas durante o shutdown
    stdio_set_chars_available_callback(NULL, NULL);

    // Desliga as saídas do alarme (LEDs, buzzer)
    alarm_control_shutdown_outputs();

    // Mostra mensagem de AP desligado no OLED
    oled_display_show_ap_disabled();

    // Desinicializa o gerenciador de rede (fecha servidor TCP, desabilita AP, DHCP, DNS)
    // A função key_pressed_func já chama cyw43_arch_disable_ap_mode(),
    // então network_manager_deinit pode focar em fechar TCP e desiniciar DHCP/DNS.
    // A ordem de cyw43_arch_disable_ap_mode() e cyw43_arch_deinit() é importante.
    // Primeiro desabilitamos o modo AP (se ainda não foi feito pela key_pressed_func).
    // Esta lógica está agora encapsulada em network_manager_deinit.
    // A key_pressed_func APENAS seta state->complete.
    // O AP é desabilitado em network_manager_deinit.

    // A desabilitação do AP e o fechamento dos servidores de rede precisam do LwIP rodando.
    // O cyw43_arch_deinit final desliga tudo.
    // A key_pressed_func apenas sinaliza. O main faz o shutdown ordenado.
    if (g_server_state) { // Verifica se o estado ainda é válido
         // Primeiro, desabilitar o AP explicitamente antes de fechar os servidores LwIP
        printf("Desabilitando modo Access Point...\n");
        cyw43_arch_lwip_begin();
        cyw43_arch_disable_ap_mode();
        cyw43_arch_lwip_end();
        alarm_control_set_ap_led(false); // Garante que o LED do AP está desligado

        network_manager_deinit(g_server_state); // Fecha TCP, desliga DHCP/DNS
    }


    // Desinicializa a arquitetura CYW43 (Wi-Fi) como último passo da rede.
    cyw43_arch_deinit();
    printf("Arquitetura CYW43 desinicializada.\n");
```

```
    // Libera a memória alocada para o estado do servidor
    if (g_server_state) {
        free(g_server_state);
        g_server_state = NULL;
    }

    printf("Simulador de Alarme encerrado.\n");
    return 0;
}
```

```c
/**
 * @file app_config.h
 * @brief Arquivo de configuração central para definições e constantes do projeto.
 *
 * @author  Manoel Furtado
 * @date    25 maio 2025
 */

#ifndef APP_CONFIG_H
#define APP_CONFIG_H

// Definições do Wi-Fi Access Point
#define WIFI_SSID "PICO_ALARME_AP"      /**< Nome da rede Wi-Fi (SSID) a ser criada. */
#define WIFI_PASSWORD "picoalarme123"   /**< Senha da rede Wi-Fi. */
#define TCP_PORT 80                     /**< Porta TCP para o servidor HTTP. */

// Definições dos Pinos GPIO
#define LED_GREEN_GPIO 11   /**< Pino GPIO para o LED Verde (sistema em repouso). */
#define LED_BLUE_GPIO 12    /**< Pino GPIO para o LED Azul (status do Access Point). */
#define LED_RED_GPIO 13     /**< Pino GPIO para o LED Vermelho (alarme ativo). */
#define BUZZER_GPIO 10      /**< Pino GPIO para o Buzzer. */

// Definições do Display OLED I2C
#define I2C_SDA_PIN 14      /**< Pino GPIO para o SDA da comunicação I2C com o OLED. */
#define I2C_SCL_PIN 15      /**< Pino GPIO para o SCL da comunicação I2C com o OLED. */
#define OLED_I2C_CLOCK 400000 /**< Velocidade do clock I2C para o OLED (400 kHz). */

// Configurações do Alarme
#define ALARM_BLINK_INTERVAL_MS 500 /**< Intervalo em milissegundos para piscar o LED vermelho e o buzzer. */

// Mensagens para o Display OLED
#define MSG_EVACUAR "EVACUAR"           /**< Mensagem para estado de alarme ativo. */
#define MSG_REPOUSO_L1 "Sistema em"     /**< Linha 1 para estado de sistema em repouso. */
#define MSG_REPOUSO_L2 "repouso"        /**< Linha 2 para estado de sistema em repouso. */
#define MSG_AP_OFF "AP Desativado"      /**< Mensagem quando o Access Point é desativado. */

// Outras definições do servidor HTTP
#define DEBUG_printf printf /**< Define a função de impressão para debug. Pode ser redirecionada se necessário. */
#define POLL_TIME_S 5     /**< Tempo de poll para conexões TCP (define o intervalo do callback tcp_poll). */

#endif // APP_CONFIG_H
```

```c
/**
 * @file alarm_control.c
 * @brief Implementação do controle do sistema de alarme (LEDs e Buzzer).
 *
 * @author  Manoel Furtado
 * @date    25 maio 2025
```

```c
 */

#include <stdio.h>
#include "alarm_control.h"
#include "app_config.h"
#include "oled_display.h"    // Para atualizar o display quando o estado do alarme muda
#include "pico/stdlib.h"     // Para time_us_64()
#include "hardware/gpio.h"

// Variáveis estáticas para encapsular o estado do alarme dentro deste módulo.
static volatile bool s_alarm_active = false;            /**< Estado atual do alarme (ativo/inativo). */
static bool s_alarm_output_toggle_state = false;    /**< Estado de toggle para piscar LED/Buzzer. */
static uint64_t s_last_toggle_time_us = 0;          /**< Timestamp do último toggle do LED/Buzzer. */

/**
 * @brief Inicializa os GPIOs para os LEDs e o buzzer.
 */
void alarm_control_init(void) {
    // LED Verde (sistema em repouso)
    gpio_init(LED_GREEN_GPIO);
    gpio_set_dir(LED_GREEN_GPIO, GPIO_OUT);
    gpio_put(LED_GREEN_GPIO, true); // Ligado inicialmente

    // LED Azul (status do AP)
    gpio_init(LED_BLUE_GPIO);
    gpio_set_dir(LED_BLUE_GPIO, GPIO_OUT);
    gpio_put(LED_BLUE_GPIO, false); // Desligado inicialmente

    // LED Vermelho (alarme ativo)
    gpio_init(LED_RED_GPIO);
    gpio_set_dir(LED_RED_GPIO, GPIO_OUT);
    gpio_put(LED_RED_GPIO, false); // Desligado inicialmente

    // Buzzer
    gpio_init(BUZZER_GPIO);
    gpio_set_dir(BUZZER_GPIO, GPIO_OUT);
    gpio_put(BUZZER_GPIO, false); // Desligado inicialmente

    printf("GPIOs para LEDs e Buzzer inicializados.\n");
}

/**
 * @brief Define o estado do alarme (ativo ou inativo).
 */
void alarm_control_set_active(bool active) {
    // Muda o estado apenas se for diferente do atual para evitar processamento desnecessário
    if (s_alarm_active != active) {
        s_alarm_active = active;
```

```
        oled_display_update_status(s_alarm_active); // Atualiza o display OLED

        if (s_alarm_active) {
            printf("Alarme ATIVADO.\n");
            gpio_put(LED_GREEN_GPIO, false); // Apaga LED verde
            // LED vermelho e buzzer serão controlados por alarm_control_process()
            s_last_toggle_time_us = time_us_64(); // Reseta o timer de blink para iniciar imediatamente
            s_alarm_output_toggle_state = false; // Garante que o primeiro blink acenda
        } else {
            printf("Alarme DESATIVADO.\n");
            gpio_put(LED_RED_GPIO, false);    // Apaga LED vermelho
            gpio_put(BUZZER_GPIO, false);  // Desliga buzzer
            gpio_put(LED_GREEN_GPIO, true);   // Acende LED verde
            s_alarm_output_toggle_state = false; // Reseta estado do toggle
        }
    }
}

/**
 * @brief Verifica se o alarme está atualmente ativo.
 */
bool alarm_control_is_active(void) {
    return s_alarm_active;
}

/**
 * @brief Processa a lógica de temporização do alarme.
 */
void alarm_control_process(void) {
    if (s_alarm_active) {
        uint64_t current_time_us = time_us_64();
        // Verifica se passou o intervalo para alternar o estado do LED/Buzzer
        if (current_time_us - s_last_toggle_time_us >= (ALARM_BLINK_INTERVAL_MS * 1000)) {
            s_alarm_output_toggle_state = !s_alarm_output_toggle_state; // Alterna o estado
            gpio_put(LED_RED_GPIO, s_alarm_output_toggle_state);      // Aplica ao LED vermelho
            gpio_put(BUZZER_GPIO, s_alarm_output_toggle_state);       // Aplica ao buzzer
            s_last_toggle_time_us = current_time_us;                  // Atualiza o timestamp
        }
    }
    // Se o alarme não estiver ativo, os LEDs e buzzer já foram definidos por alarm_control_set_active()
}

/**
 * @brief Controla o LED de status do Access Point (AP).
 */
void alarm_control_set_ap_led(bool on) {
    gpio_put(LED_BLUE_GPIO, on);
}
```

```c
/**
 * @brief Desliga todas as saídas controladas pelo sistema de alarme.
 */
void alarm_control_shutdown_outputs(void) {
    gpio_put(LED_RED_GPIO, false);
    gpio_put(BUZZER_GPIO, false);
    gpio_put(LED_GREEN_GPIO, false);
    // O LED azul (AP_LED) é controlado separadamente pela lógica de rede/main.
}
```

```c
/**
 * @file alarm_control.h
 * @brief Interface para controle do sistema de alarme (LEDs e Buzzer).
 *
 * Este módulo gerencia o estado do alarme, controla os LEDs de status
 * (verde, vermelho, azul) e o buzzer.
 *
 * @author  Manoel Furtado
 * @date    25 maio 2025
 */

#ifndef ALARM_CONTROL_H
#define ALARM_CONTROL_H

#include <stdbool.h> // Para o tipo bool

/**
 * @brief Inicializa os GPIOs para os LEDs e o buzzer.
 *
 * Configura os pinos GPIO como saídas e define seus estados iniciais.
 * O LED verde é ligado (sistema em repouso), os outros ficam desligados.
 */
void alarm_control_init(void);

/**
 * @brief Define o estado do alarme (ativo ou inativo).
 *
 * Atualiza o estado interno do alarme, controla os LEDs verde/vermelho
 * e solicita a atualização do display OLED.
 * @param active Verdadeiro para ativar o alarme, falso para desativar.
 */
void alarm_control_set_active(bool active);

/**
 * @brief Verifica se o alarme está atualmente ativo.
 *
 * @return Verdadeiro se o alarme estiver ativo, falso caso contrário.
```

```
 */
bool alarm_control_is_active(void);

/**
 * @brief Processa a lógica de temporização do alarme.
 *
 * Esta função deve ser chamada repetidamente no loop principal do programa.
 * Ela é responsável por piscar o LED vermelho e o buzzer se o alarme estiver ativo.
 */
void alarm_control_process(void);

/**
 * @brief Controla o LED de status do Access Point (AP).
 *
 * @param on Verdadeiro para ligar o LED azul, falso para desligar.
 */
void alarm_control_set_ap_led(bool on);

/**
 * @brief Desliga todas as saídas controladas pelo sistema de alarme.
 *
 * Usado durante o processo de encerramento do programa.
 */
void alarm_control_shutdown_outputs(void);

#endif // ALARM_CONTROL_H
```

```
/**
 * @file oled_display.c
 * @brief Implementação para controle do display OLED SSD1306.
 *
 * @author  Manoel Furtado
 * @date    25 maio 2025
 */

 #include <stdio.h>
#include "oled_display.h"
#include "app_config.h"
#include "ssd1306.h"         // Biblioteca principal do SSD1306
#include "hardware/i2c.h"
#include "pico/stdlib.h"     // Para memset, etc.
#include <string.h>          // Para strlen

// Buffer interno para o conteúdo do display OLED.
// O tamanho é definido pela biblioteca ssd1306 (ssd1306_buffer_length).
static uint8_t oled_buffer[ssd1306_buffer_length];

// Estrutura que define a área de renderização do OLED (tela inteira).
```

```c
// É definida pela biblioteca ssd1306.
static struct render_area display_area;

/**
 * @brief Inicializa o hardware I2C e o display OLED.
 */
void oled_display_init(void) {
    printf("Inicializando I2C para Display OLED...\n");
    // Usa diretamente i2c1, que corresponde aos pinos GP14 (SDA) e GP15 (SCL)
    // quando configurados para a função I2C.
    i2c_init(i2c1, OLED_I2C_CLOCK); // Inicializa I2C1 na velocidade definida
    gpio_set_function(I2C_SDA_PIN, GPIO_FUNC_I2C); // Configura pino SDA para função I2C
    gpio_set_function(I2C_SCL_PIN, GPIO_FUNC_I2C); // Configura pino SCL para função I2C
    // Pull-ups geralmente não são necessários nos pinos do Pico se o módulo OLED já os tiver.
    printf("Pinos I2C configurados (SDA: %d, SCL: %d para i2c1).\n", I2C_SDA_PIN, I2C_SCL_PIN);

    // Define a área de renderização para cobrir todo o display
    display_area.start_column = 0;
    display_area.end_column = ssd1306_width - 1; // ssd1306_width da lib
    display_area.start_page = 0;
    display_area.end_page = ssd1306_n_pages - 1; // ssd1306_n_pages da lib

    // Calcula o tamanho do buffer necessário para esta área.
    // Esta função é externa, fornecida pela biblioteca ssd1306.
    calculate_render_area_buffer_length(&display_area);

    // Inicializa o hardware do display OLED.
    // Esta função é externa, fornecida pela biblioteca ssd1306.
    ssd1306_init();
    printf("Display OLED SSD1306 inicializado.\n");
}

/**
 * @brief Atualiza o display OLED com o status atual do alarme.
 */
void oled_display_update_status(bool is_alarm_active) {
    // Limpa o buffer do OLED preenchendo com zeros (pixels apagados)
    memset(oled_buffer, 0, ssd1306_buffer_length);

    if (is_alarm_active) {
        // Se o alarme estiver ativo, exibe "EVACUAR" centralizado.
        // A altura do caractere é 8 pixels. O display tem geralmente 64 pixels de altura.
        // Posição Y para centralizar verticalmente um texto de 1 linha: (altura_display / 2) - (altura_fonte / 2)
        // Ex: (64 / 2) - (8 / 2) = 32 - 4 = 28
        ssd1306_draw_string(oled_buffer, (ssd1306_width - (strlen(MSG_EVACUAR) * 8)) / 2, 28, MSG_EVACUAR);
    } else {
        // Se o alarme estiver inativo, exibe "Sistema em repouso" em duas linhas, centralizado.
        // Posição Y para 2 linhas: L1 em ~20, L2 em ~36 (ajustar conforme gosto visual)
```

```
        ssd1306_draw_string(oled_buffer, (ssd1306_width - (strlen(MSG_REPOUSO_L1) * 8)) / 2, 20, MSG_REPOUSO_L1);
        ssd1306_draw_string(oled_buffer, (ssd1306_width - (strlen(MSG_REPOUSO_L2) * 8)) / 2, 36, MSG_REPOUSO_L2);
    }
    // Envia o conteúdo do buffer para ser renderizado no display OLED.
    // Esta função é externa, fornecida pela biblioteca ssd1306.
    render_on_display(oled_buffer, &display_area);
}

/**
 * @brief Exibe a mensagem "AP Desativado" no display OLED.
 */
void oled_display_show_ap_disabled(void) {
    memset(oled_buffer, 0, ssd1306_buffer_length);
    ssd1306_draw_string(oled_buffer, (ssd1306_width - (strlen(MSG_AP_OFF) * 8)) / 2, 28, MSG_AP_OFF);
    render_on_display(oled_buffer, &display_area);
}

/**
 * @brief Limpa o conteúdo do display OLED.
 */
void oled_display_clear(void) {
    memset(oled_buffer, 0, ssd1306_buffer_length);
    render_on_display(oled_buffer, &display_area);
}
```

```
/**
 * @file oled_display.h
 * @brief Interface para controle do display OLED SSD1306.
 *
 * Este módulo encapsula a inicialização e as funções de atualização
 * para o display OLED.
 *
 * @author  Manoel Furtado
 * @date    25 maio 2025
 */

#ifndef OLED_DISPLAY_H
#define OLED_DISPLAY_H

#include <stdbool.h> // Para o tipo bool

/**
 * @brief Inicializa o hardware I2C e o display OLED.
 *
 * Configura os pinos I2C, inicializa a comunicação I2C e envia os comandos
 * de inicialização para o display SSD1306.
 * Deve ser chamada uma vez no início do programa.
 */
```

```c
void oled_display_init(void);

/**
 * @brief Atualiza o display OLED com o status atual do alarme.
 *
 * Exibe "EVACUAR" se o alarme estiver ativo, ou "Sistema em repouso" caso contrário.
 * @param is_alarm_active Verdadeiro se o alarme estiver ativo, falso caso contrário.
 */
void oled_display_update_status(bool is_alarm_active);

/**
 * @brief Exibe a mensagem "AP Desativado" no display OLED.
 *
 * Usada quando o modo Access Point é finalizado pelo usuário.
 */
void oled_display_show_ap_disabled(void);

/**
 * @brief Limpa o conteúdo do display OLED.
 */
void oled_display_clear(void);

#endif // OLED_DISPLAY_H
```

```c
/**
 * @file network_manager.c
 * @brief Implementação do gerenciamento da rede Wi-Fi e servidor HTTP.
 *
 * @author  Manoel Furtado
 * @date    25 maio 2025
 */

#include "network_manager.h"
#include "app_config.h"
#include "alarm_control.h" // Para interagir com a lógica do alarme

#include <string.h>
#include <stdio.h>
#include <stdlib.h> // Para calloc/free

#include "pico/stdlib.h"
#include "pico/cyw43_arch.h"

#include "lwip/pbuf.h"
#include "lwip/tcp.h"

#include "dhcpserver.h"
#include "dnsserver.h"
```

```c
// --- Definições HTTP Internas ---
#define HTTP_GET "GET"
#define HTTP_RESPONSE_HEADERS "HTTP/1.1 %d OK\nContent-Length: %d\nContent-Type: text/html; charset=utf-8\nConnection: close\n\n"
#define ALARM_PAGE_BODY "<html><head><title>Controle de Alarme</title><meta name=\"viewport\" content=\"width=device-width, initial-scale=1.0\"></head>" \
                        "<body><style>body{font-family: Arial, sans-serif; text-align: center; margin-top: 50px;} " \
                        "h1{color: #333;} h2{color: #444; font-size: 1.2em; margin-top: 0px;} p{color: #555;} " \
                        ".button {display: inline-block; padding: 15px 25px; font-size: 20px; cursor: pointer; " \
                        "text-align: center; text-decoration: none; outline: none; color: #fff; " \
                        "border: none; border-radius: 15px; box-shadow: 0 9px #999;} " \
                        ".button-on {background-color: #4CAF50;} .button-on:hover {background-color: #3e8e41} " \
                        ".button-off {background-color: #f44336;} .button-off:hover {background-color: #da190b} " \
                        ".status {font-weight: bold; font-size: 22px;} " \
                        ".status-on {color: #f44336;} .status-off {color: #4CAF50;}</style>" \
                        "<h1>Simulador Portatil de Alarme</h1>" \
                        "<h2>Atividade 08 - Manoel</h2>" \
                        "<p>Estado do Alarme: <strong class=\"status status-%s\">%s</strong></p>" \
                        "<p><a href=\"/?alarm=%s\" class=\"button button-%s\">%s Alarme</a></p>" \
                        "</body></html>"
#define ALARM_PARAM_STR "alarm=%s"
#define HTTP_RESPONSE_REDIRECT_TO_ROOT "HTTP/1.1 302 Redirect\nLocation: http://%s/\n\n"


// Instâncias dos servidores DHCP e DNS, gerenciadas estaticamente por este módulo.
static dhcp_server_t s_dhcp_server;
static dns_server_t s_dns_server;

/**
 * @brief Estrutura de estado para uma conexão TCP individual com um cliente.
 * Esta estrutura é específica para a implementação do servidor HTTP.
 */
typedef struct TCP_CONNECT_STATE_T_ {
    struct tcp_pcb *pcb;    /**< PCB para a conexão do cliente. */
    int sent_len;           /**< Bytes enviados ao cliente. */
    char headers[128];      /**< Buffer para os cabeçalhos HTTP. */
    char result[1500];      /**< Buffer para o corpo da página HTML. */
    int header_len;         /**< Comprimento dos cabeçalhos. */
    int result_len;         /**< Comprimento do corpo HTML. */
    ip_addr_t *gw;          /**< Ponteiro para o endereço IP do gateway (IP do Pico). */
} TCP_CONNECT_STATE_T;

// --- Protótipos de Funções Estáticas (Callbacks TCP e Lógica Interna) ---
static err_t tcp_close_client_connection(TCP_CONNECT_STATE_T *con_state, struct tcp_pcb *client_pcb, err_t close_err);
static err_t tcp_server_sent(void *arg, struct tcp_pcb *pcb, u16_t len);
static int http_generate_page_content(const char *params, char *result, size_t max_result_len);
static err_t tcp_server_recv(void *arg, struct tcp_pcb *pcb, struct pbuf *p, err_t err);
```

```c
static err_t tcp_server_poll(void *arg, struct tcp_pcb *pcb);
static void tcp_server_err(void *arg, err_t err);
static err_t tcp_server_accept(void *arg, struct tcp_pcb *client_pcb, err_t err);
static bool tcp_server_open_internal(TCP_SERVER_T *state); // Renomeada para evitar conflito de nome

/**
 * @brief Fecha a conexão com um cliente TCP.
 */
static err_t tcp_close_client_connection(TCP_CONNECT_STATE_T *con_state, struct tcp_pcb *client_pcb, err_t close_err)
{
    if (client_pcb) {
        assert(con_state && con_state->pcb == client_pcb);
        tcp_arg(client_pcb, NULL);
        tcp_poll(client_pcb, NULL, 0);
        tcp_sent(client_pcb, NULL);
        tcp_recv(client_pcb, NULL);
        tcp_err(client_pcb, NULL);
        err_t err = tcp_close(client_pcb);
        if (err != ERR_OK) {
            DEBUG_printf("Falha ao fechar TCP, erro %d. Abortando.\n", err);
            tcp_abort(client_pcb);
            close_err = ERR_ABRT;
        }
        if (con_state) {
            free(con_state);
        }
    }
    return close_err;
}

/**
 * @brief Callback: Dados enviados e confirmados (acked) pelo cliente.
 */
static err_t tcp_server_sent(void *arg, struct tcp_pcb *pcb, u16_t len) {
    TCP_CONNECT_STATE_T *con_state = (TCP_CONNECT_STATE_T*)arg;
    DEBUG_printf("TCP dados enviados (acked): %u bytes\n", len);
    con_state->sent_len += len;
    if (con_state->sent_len >= con_state->header_len + con_state->result_len) {
        DEBUG_printf("Todos os dados enviados e acked. Fechando conexão.\n");
        return tcp_close_client_connection(con_state, pcb, ERR_OK);
    }
    return ERR_OK;
}

/**
 * @brief Gera o conteúdo da página HTML de controle do alarme.
 */
static int http_generate_page_content(const char *params, char *result, size_t max_result_len) {
```

```
    char alarm_command_str[4]; // "on" ou "off"

    if (params) {
        if (sscanf(params, ALARM_PARAM_STR, alarm_command_str) == 1) {
            if (strcmp(alarm_command_str, "on") == 0) {
                // Chama a função do módulo de controle de alarme para ativar
                if (!alarm_control_is_active()) {
                     DEBUG_printf("Comando HTTP: LIGAR ALARME\n");
                    alarm_control_set_active(true);
                }
            } else if (strcmp(alarm_command_str, "off") == 0) {
                // Chama a função do módulo de controle de alarme para desativar
                if (alarm_control_is_active()) {
                    DEBUG_printf("Comando HTTP: DESLIGAR ALARME\n");
                    alarm_control_set_active(false);
                }
            }
        }
    }

    // Obtém o estado atual do alarme do módulo de controle
    bool is_active = alarm_control_is_active();
    const char *current_status_str = is_active ? "LIGADO" : "DESLIGADO";
    const char *status_class_suffix = is_active ? "on" : "off";
    const char *next_action_param = is_active ? "off" : "on";
    const char *button_text = is_active ? "Desligar" : "Ligar";
    const char *button_class_suffix = is_active ? "off" : "on";

    return snprintf(result, max_result_len, ALARM_PAGE_BODY,
                    status_class_suffix, current_status_str,
                    next_action_param, button_class_suffix, button_text);
}

/**
 * @brief Callback: Dados recebidos de um cliente TCP.
 */
err_t tcp_server_recv(void *arg, struct tcp_pcb *pcb, struct pbuf *p, err_t err) {
    TCP_CONNECT_STATE_T *con_state = (TCP_CONNECT_STATE_T*)arg;
    if (!p) {
        DEBUG_printf("Conexão fechada pelo cliente.\n");
        return tcp_close_client_connection(con_state, pcb, ERR_OK);
    }
    assert(con_state && con_state->pcb == pcb);

    if (p->tot_len > 0) {
        DEBUG_printf("TCP dados recebidos: %d bytes, erro: %d\n", p->tot_len, err);
        size_t copy_len = p->tot_len > sizeof(con_state->headers) - 1 ? sizeof(con_state->headers) - 1 : p->tot_len;
        pbuf_copy_partial(p, con_state->headers, copy_len, 0);
```

```
        con_state->headers[copy_len] = '\0';

        if (strncmp(HTTP_GET, con_state->headers, sizeof(HTTP_GET) - 1) == 0) {
            char *full_request_path_and_params = con_state->headers + sizeof(HTTP_GET) - 1;
            while(*full_request_path_and_params == ' ') full_request_path_and_params++;
            char *http_version_start = strchr(full_request_path_and_params, ' ');
            if (http_version_start) {
                *http_version_start = '\0';
            }
            char *params = strchr(full_request_path_and_params, '?');
            char *actual_path = full_request_path_and_params;
            if (params) {
                *params++ = '\0';
            }

            DEBUG_printf("Requisição HTTP: Caminho='%s', Parâmetros='%s'\n", actual_path, params ? params :
"Nenhum");

            if (strcmp(actual_path, "/") == 0) {
                con_state->result_len = http_generate_page_content(params, con_state->result, sizeof(con_state-
>result));
                if (con_state->result_len >= sizeof(con_state->result) - 1) {
                    DEBUG_printf("Buffer de resultado HTML muito pequeno: %d necessário, %u disponível.\n",
con_state->result_len, (unsigned int)sizeof(con_state->result));
                    pbuf_free(p);
                    return tcp_close_client_connection(con_state, pcb, ERR_CLSD);
                }
                con_state->header_len = snprintf(con_state->headers, sizeof(con_state->headers),
HTTP_RESPONSE_HEADERS, 200, con_state->result_len);
            } else {
                DEBUG_printf("Caminho '%s' não encontrado. Redirecionando para '/'.\n", actual_path);
                con_state->result_len = 0;
                con_state->header_len = snprintf(con_state->headers, sizeof(con_state->headers),
HTTP_RESPONSE_REDIRECT_TO_ROOT, ipaddr_ntoa(con_state->gw));
            }

            if (con_state->header_len >= sizeof(con_state->headers) - 1) {
                 DEBUG_printf("Buffer de cabeçalhos HTTP muito pequeno.\n");
                 pbuf_free(p);
                 return tcp_close_client_connection(con_state, pcb, ERR_CLSD);
            }

            con_state->sent_len = 0;
            err_t write_err = tcp_write(pcb, con_state->headers, con_state->header_len, 0);
            if (write_err != ERR_OK) {
                DEBUG_printf("Falha ao escrever cabeçalhos HTTP: %d\n", write_err);
                pbuf_free(p);
                return tcp_close_client_connection(con_state, pcb, write_err);
```

```
            }
            if (con_state->result_len > 0) {
                write_err = tcp_write(pcb, con_state->result, con_state->result_len, 0);
                if (write_err != ERR_OK) {
                    DEBUG_printf("Falha ao escrever corpo HTML: %d\n", write_err);
                    pbuf_free(p);
                    return tcp_close_client_connection(con_state, pcb, write_err);
                }
            }
        }
        tcp_recved(pcb, p->tot_len);
    }
    pbuf_free(p);
    return ERR_OK;
}

/**
 * @brief Callback: Polling periódico da conexão TCP.
 */
static err_t tcp_server_poll(void *arg, struct tcp_pcb *pcb) {
    TCP_CONNECT_STATE_T *con_state = (TCP_CONNECT_STATE_T*)arg;
    DEBUG_printf("TCP Poll callback. Fechando conexão inativa.\n");
    return tcp_close_client_connection(con_state, pcb, ERR_OK);
}

/**
 * @brief Callback: Erro na conexão TCP.
 */
static void tcp_server_err(void *arg, err_t err) {
    TCP_CONNECT_STATE_T *con_state = (TCP_CONNECT_STATE_T*)arg;
    if (err != ERR_ABRT) {
        DEBUG_printf("Erro na conexão TCP: %d\n", err);
        if (con_state) {
            tcp_close_client_connection(con_state, con_state->pcb, err);
        }
    }
}

/**
 * @brief Callback: Nova conexão TCP aceita.
 */
static err_t tcp_server_accept(void *arg, struct tcp_pcb *client_pcb, err_t err) {
    TCP_SERVER_T *state = (TCP_SERVER_T*)arg; // O 'arg' aqui é o estado global do servidor TCP
    if (err != ERR_OK || client_pcb == NULL) {
        DEBUG_printf("Falha ao aceitar conexão: %d\n", err);
        return ERR_VAL;
    }
    DEBUG_printf("Cliente conectado.\n");
```

```
    TCP_CONNECT_STATE_T *con_state = calloc(1, sizeof(TCP_CONNECT_STATE_T));
    if (!con_state) {
        DEBUG_printf("Falha ao alocar estado para conexão do cliente.\n");
        return ERR_MEM;
    }
    con_state->pcb = client_pcb;
    con_state->gw = &state->gw; // Passa o endereço do gateway do servidor principal

    tcp_arg(client_pcb, con_state);
    tcp_sent(client_pcb, tcp_server_sent);
    tcp_recv(client_pcb, tcp_server_recv);
    tcp_poll(client_pcb, tcp_server_poll, POLL_TIME_S * 2);
    tcp_err(client_pcb, tcp_server_err);

    return ERR_OK;
}

/**
 * @brief Abre o servidor TCP para escutar por conexões HTTP.
 * Função interna chamada por network_manager_init.
 */
static bool tcp_server_open_internal(TCP_SERVER_T *state) {
    DEBUG_printf("Iniciando servidor na porta %d\n", TCP_PORT);
    struct tcp_pcb *pcb = tcp_new_ip_type(IPADDR_TYPE_ANY);
    if (!pcb) {
        DEBUG_printf("Falha ao criar PCB TCP.\n");
        return false;
    }
    err_t err = tcp_bind(pcb, IP_ANY_TYPE, TCP_PORT);
    if (err != ERR_OK) {
        DEBUG_printf("Falha ao fazer bind na porta %d: %d\n", TCP_PORT, err);
        tcp_close(pcb);
        return false;
    }
    state->server_pcb = tcp_listen_with_backlog(pcb, 1);
    if (!state->server_pcb) {
        DEBUG_printf("Falha ao colocar servidor em modo LISTEN.\n");
        if (pcb) {
            tcp_close(pcb);
        }
        return false;
    }
    tcp_arg(state->server_pcb, state); // Passa o estado principal do servidor como argumento para accept
    tcp_accept(state->server_pcb, tcp_server_accept);
    return true;
}
```

```c
/**
 * @brief Inicializa o gerenciador de rede.
 */
bool network_manager_init(TCP_SERVER_T *state) {
    if (!state) {
        DEBUG_printf("Estado do servidor TCP nulo em network_manager_init.\n");
        return false;
    }

    if (cyw43_arch_init()) {
        DEBUG_printf("Falha ao inicializar cyw43_arch (Wi-Fi).\n");
        return false;
    }
    printf("CYW43 Arch inicializado.\n");

    cyw43_arch_lwip_begin(); // Protege acesso ao LwIP
    cyw43_arch_enable_ap_mode(WIFI_SSID, WIFI_PASSWORD, CYW43_AUTH_WPA2_AES_PSK);
    alarm_control_set_ap_led(true); // Liga LED Azul para indicar AP ativo

    ip4_addr_t mask;
    IP4_ADDR(&state->gw, 192, 168, 4, 1); // IP do Pico W (Gateway)
    IP4_ADDR(&mask, 255, 255, 255, 0);   // Máscara de sub-rede

    dhcp_server_init(&s_dhcp_server, &state->gw, &mask);
    printf("Servidor DHCP iniciado no IP %s\n", ipaddr_ntoa(&state->gw));

    dns_server_init(&s_dns_server, &state->gw);
    printf("Servidor DNS iniciado.\n");
    cyw43_arch_lwip_end();

    if (!tcp_server_open_internal(state)) {
        DEBUG_printf("Falha ao abrir servidor TCP.\n");
        // Desfaz inicializações parciais
        cyw43_arch_lwip_begin();
        dns_server_deinit(&s_dns_server);
        dhcp_server_deinit(&s_dhcp_server);
        cyw43_arch_disable_ap_mode();
        cyw43_arch_lwip_end();
        alarm_control_set_ap_led(false);
        cyw43_arch_deinit();
        return false;
    }
    printf("Servidor HTTP iniciado. Conecte-se a rede Wi-Fi '%s'.\n", WIFI_SSID);
    printf("Acesse http://%s no navegador.\n", ipaddr_ntoa(&state->gw));
    return true;
}

/**
```

```
 * @brief Desinicializa o gerenciador de rede.
 */
void network_manager_deinit(TCP_SERVER_T *state) {
    if (state && state->server_pcb) {
        tcp_arg(state->server_pcb, NULL);
        tcp_close(state->server_pcb);
        state->server_pcb = NULL;
    }
    // As conexões clientes ativas serão fechadas pelos seus timeouts ou callbacks de erro.

    cyw43_arch_lwip_begin();
    dns_server_deinit(&s_dns_server);
    dhcp_server_deinit(&s_dhcp_server);
    // A desabilitação do modo AP já foi feita na key_pressed_func,
    // mas podemos garantir aqui também se o state->complete for setado por outro motivo.
    // No entanto, a key_pressed_func já chama cyw43_arch_disable_ap_mode().
    // Se esta função for chamada após key_pressed_func, o AP já estará desabilitado.
    // cyw43_arch_disable_ap_mode(); // Pode ser redundante ou causar problema se chamado duas vezes sem reabilitar.
    cyw43_arch_lwip_end();

    alarm_control_set_ap_led(false);
    // cyw43_arch_deinit(); // A desinicialização final do CYW43 será feita no main.
    printf("Serviços de rede (DHCP, DNS, TCP) encerrados.\n");
}
```

```
/**
 * @file network_manager.h
 * @brief Interface para gerenciamento da rede Wi-Fi e servidor HTTP.
 *
 * Este módulo configura o Pico W como Access Point, gerencia os servidores
 * DHCP e DNS, e lida com as conexões e requisições do servidor HTTP.
 *
 * @author  Manoel Furtado
 * @date    25 maio 2025
 */

#ifndef NETWORK_MANAGER_H
#define NETWORK_MANAGER_H

#include "pico/cyw43_arch.h" // Para ip_addr_t e outras dependências de rede
#include <stdbool.h>

/**
 * @brief Estrutura de estado para o servidor TCP/AP.
 * Usada internamente pelo network_manager e potencialmente pela função main
 * para sinalizar o encerramento.
 */
typedef struct TCP_SERVER_T_ {
```

```c
    struct tcp_pcb *server_pcb; /**< Protocol Control Block (PCB) para o servidor TCP. */
    bool complete;              /**< Flag para indicar se o servidor/aplicação deve terminar. */
    ip_addr_t gw;               /**< Endereço IP do gateway (o próprio Pico W no modo AP). */
    // Adicione outros campos de estado global da rede aqui se necessário.
    // Instâncias dos servidores DHCP e DNS serão gerenciadas internamente no .c
} TCP_SERVER_T;



/**
 * @brief Inicializa o gerenciador de rede.
 *
 * Configura o Wi-Fi em modo Access Point, inicia os servidores DHCP e DNS,
 * e abre o servidor TCP para escutar por conexões HTTP.
 *
 * @param state Ponteiro para a estrutura de estado do servidor TCP, que será gerenciada.
 * @return Verdadeiro se a inicialização for bem-sucedida, falso caso contrário.
 */
bool network_manager_init(TCP_SERVER_T *state);

/**
 * @brief Desinicializa o gerenciador de rede.
 *
 * Fecha o servidor TCP, desabilita o modo AP, e desinicializa os servidores
 * DHCP, DNS e a arquitetura CYW43.
 *
 * @param state Ponteiro para a estrutura de estado do servidor TCP.
 */
void network_manager_deinit(TCP_SERVER_T *state);

#endif // NETWORK_MANAGER_H
```