Aluno do Embarcatech_37 no IFMA

Nome: Manoel Felipe Costa Furtado

Matrícula: 20251RSE.MTC0086

Atividade – Referente ao capítulo 02 da unidade 03.

Tema do Capítulo – Sistemas de Tempo Real: Conceitos Básicos.

Prazo dia 06/07/2025 as 23:59

Objetivo: Desenvolver uma aplicação embarcada na placa BitDogLab, utilizando FreeRTOS e Pico-SDK, com o objetivo de testar e demonstrar, de forma prática e didática, o funcionamento de multitarefas no controle de periféricos como LEDs, botões, joystick, microfone e buzzer, além de aplicar conceitos de leitura analógica, temporização e resposta a eventos em tempo real.

Enunciado: Aplicação Didática com FreeRTOS na BitDogLab: Multitarefa, Monitoramento e Autoteste de Periféricos

Desenvolva uma aplicação embarcada utilizando a placa BitDogLab, o FreeRTOS e a Pico-SDK, estruturada em três tarefas concorrentes, com foco em testes de hardware, monitora mento de sensores e sinalização visual/sonora. A aplicação deve ser iniciada automaticamente ao ligar a placa e funcionar conforme a descrição abaixo:

Descrição do Sistema:

Tarefa 1: Self-Test

- Testar LEDs RGB: acender e apagar sequencialmente.
- Testar Buzzer: gerar som simples.
- Testar botões A, B e joystick SW.
- Testar Joystick analógico (ADC0 e ADC1).
- Testar Microfone (ADC2).
- Imprimir resultados na porta USB com pequenas pausas para visualização.
- Deletar a si própria após concluir (vTaskDelete(NULL)).

Tarefa 2: Alive Task

- Piscar o LED vermelho (GPIO 13) indicando o funcionamento.
- Ciclo de 1000ms (500ms ligado e 500ms desligado).

Tarefa 3: Monitor de Joystick e Alarme

- Ler e imprimir as tensões dos eixos X (ADC1) e Y (ADC0) do joystick.

- Imprimir leituras a cada 50ms na porta USB.
- Se qualquer eixo exceder 3.00V, ativar o buzzer (GPIO 21).
- Desligar o buzzer quando os valores voltarem ao normal.

Instruções:

Após a inicialização da aplicação, ficam em funcionamento somente a Tarefa 2 e Tarefa 3, pois a Tarefa 1 se auto deleta após a inicialização. Todavia se a placa for reinicializada todo o processo deve se repetir novamente.

Periférico	Função	GPIO / ADC
LED Vermelho (Alive)	Piscar Alive	13
LED RGB Verde	Self-Test	11
LED RGB Azul	Self-Test	12
Buzzer	Alarme e Self-Test	21
Botão A	Self-Test	5
Botão B	Self-Test	6
Joystick SW	Self-Test	22
Joystick VRy	Leitura Analógica	ADC0 (26)
Joystick VRx	Leitura Analógica	ADC1 (27)
Microfone	Leitura Analógica	ADC2 (28)

Instruções de Implementação:

1) Faça o projeto no VSCode e execute na placa BitdogLab.
2) Adicione pequenos delays entre os testes no Self-Test para garantir tempo de leitura via USB.
3) Utilize a função sleep_ms() antes de iniciar o scheduler para garantir estabilização da USB CDC.

Links das simulações

- Nome do arquivo principal: “main.c”.
- Link do Projeto completo com a pasta build:
- Link do Vídeo: https://youtu.be/JhrmZqsJAOY
- GitHub: https://github.com/ManoelFelipe/Embarcatech_37/tree/main/Unidade_03/Cap_02/Atividade_02

Como entender o projeto:

1. Comece pela função “int main()” em main.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: main.c

```
/**
 * @file main.c
 * @author Manoel Felipe Costa Furtado
 * @date 2025-06-30
 * @version 1.1 - Versão extensivamente comentada
 * @copyright 2025 Manoel Furtado (MIT License)
 *
 * @brief Aplicação didática para a placa BitDogLab utilizando FreeRTOS.
 *
 * Este projeto demonstra uma arquitetura de sistema embarcado multi-fase, utilizando
 * o sistema operacional de tempo real FreeRTOS no microcontrolador RP2040.
 * A aplicação é dividida em três tarefas principais com um esquema de escalonamento híbrido:
 *
 * 1.   Fase de Inicialização (Self-Test):
 * - Uma tarefa de auto-teste (`self_test_task`) é executada de forma exclusiva e sequencial
 *   na inicialização do sistema.
 * - Ela verifica os principais periféricos da placa (LEDs, buzzer, botões, joystick, microfone).
 * - Esta fase garante que o hardware está funcional antes de iniciar a operação normal.
 * - Ao final, ela habilita as outras tarefas e se auto-deleta.
 *
 * 2.   Fase de Operação Normal (RMS):
 * - Após o self-test, duas tarefas concorrentes são executadas sob uma política
 *   de escalonamento preemptivo por prioridades fixas (Rate-Monotonic Scheduling - RMS).
 * - alive_task: Pisca um LED Vermelho para sinalizar que o sistema está operante.
```

```
 * - joystick_monitor_task: Lê o joystick em alta frequência, imprime os valores e aciona
 *   um alarme caso os limites de tensão sejam excedidos.
 *
 * Este exemplo é um excelente ponto de partida para entender conceitos-chave de sistemas de
 * tempo real, como gerenciamento de tarefas, prioridades, sincronização (suspend/resume)
 * e políticas de escalonamento.
 */

//=============================================================================
// Inclusão de Bibliotecas (Headers)
//=============================================================================
#include <stdio.h>               ///< Biblioteca padrão de Entrada/Saída para funções como printf.
#include "pico/stdlib.h"         ///< Biblioteca principal do SDK do Raspberry Pi Pico, essencial para a maioria das
funções.
#include "hardware/gpio.h"       ///< Biblioteca para controle dos pinos de I/O de propósito geral (GPIO).
#include "hardware/adc.h"        ///< Biblioteca para controle do Conversor Analógico-Digital (ADC).
#include "FreeRTOS.h"            ///< Header principal do FreeRTOS, define os tipos e funções do kernel.
#include "task.h"                ///< Header da API de gerenciamento de tarefas do FreeRTOS (criar, deletar,
suspender, etc.).

//=============================================================================
// Definições de Hardware e Constantes
//=============================================================================

/** @name Definições dos Pinos dos LEDs */
///@{
const uint LED_ALIVE_PIN = 13;          ///< Pino do LED vermelho "Alive", indica que o sistema está funcionando.
const uint LED_RGB_GREEN_PIN = 11;      ///< Pino do componente Verde do LED RGB.
const uint LED_RGB_BLUE_PIN = 12;       ///< Pino do componente Azul do LED RGB.
///@}

/** @name Definições dos Pinos dos Atuadores */
///@{
const uint BUZZER_PIN = 21;             ///< Pino do buzzer para emissão de alertas sonoros.
///@}

/** @name Definições dos Pinos de Entrada (Botões) */
///@{
const uint BTN_A_PIN = 5;               ///< Pino do Botão A.
const uint BTN_B_PIN = 6;               ///< Pino do Botão B.
const uint JOYSTICK_SW_PIN = 22;        ///< Pino do botão de clique (switch) do joystick.
///@}

/** @name Definições dos Pinos Analógicos (ADC) */
///@{
const uint JOYSTICK_Y_PIN = 26;         ///< Pino do eixo Y do joystick, conectado ao canal 0 do ADC (ADC0).
const uint JOYSTICK_X_PIN = 27;         ///< Pino do eixo X do joystick, conectado ao canal 1 do ADC (ADC1).
const uint MICROPHONE_PIN = 28;         ///< Pino do microfone, conectado ao canal 2 do ADC (ADC2).
```

```
///@}

/**
 * @brief Fator de conversão para o ADC.
 * O ADC do RP2040 tem 12 bits, o que significa 2^12 = 4096 níveis de digitalização (de 0 a 4095).
 * A tensão de referência é 3.3V. A fórmula para converter o valor lido pelo ADC em tensão é:
 * Voltagem = (ValorLido / 4095) * 3.3V.
 */
const float ADC_CONVERSION_FACTOR = 3.3f / (1 << 12);

//=============================================================================
// Variáveis Globais (Handles das Tarefas)
//=============================================================================

/**
 * @brief Handles (identificadores) para as tarefas.
 * Um TaskHandle_t é um tipo de "ponteiro" ou "identificador" que o FreeRTOS usa para
 * se referir a uma tarefa específica. Precisamos deles para controlar tarefas
 * a partir de outra, como neste caso, onde a `self_test_task` precisa
 * "acordar" (retomar) as outras duas tarefas.
 */
TaskHandle_t alive_task_handle = NULL;
TaskHandle_t joystick_monitor_task_handle = NULL;

//=============================================================================
// Definições de Prioridade das Tarefas
//=============================================================================

/**
 * @brief Definição das prioridades para o escalonador.
 *
 * A estratégia de escalonamento é híbrida:
 * 1.  Self-Test (FIFO): A self_test_task tem a maior prioridade de todas para garantir que
 *     seja executada primeiro e por completo, sem preempção das outras.
 *
 * 2.  Operação Normal (RMS): As tarefas alive_task e joystick_monitor_task seguem a
 *     política Rate-Monotonic Scheduling (RMS). A regra do RMS é que tarefas com
 *     períodos menores (maior frequência de execução) devem ter prioridades mais altas.
 * - joystick_monitor_task: Período = 50ms.
 * - alive_task: Período = 1000ms.
 * Portanto, a tarefa do joystick tem prioridade maior que a tarefa do LED "alive".
 *
 * tskIDLE_PRIORITY é a prioridade mais baixa (0), usada pela tarefa ociosa (Idle Task).
 * Usamos ela como base para definir as outras prioridades de forma relativa e portável.
 */
#define TASK_ALIVE_PRIORITY      (tskIDLE_PRIORITY + 1)   ///< Prioridade mais baixa para a tarefa menos crítica.
#define TASK_JOYSTICK_PRIORITY   (tskIDLE_PRIORITY + 2)   ///< Prioridade intermediária para a tarefa de alta
frequência.
```

```c
#define TASK_SELFTEST_PRIORITY   (tskIDLE_PRIORITY + 3)   ///< Prioridade mais alta para garantir a execução na inicialização.

//===========================================================================
// Implementação das Tarefas
//===========================================================================

/**
 * @brief Tarefa 1: Realiza um auto-teste sequencial dos periféricos da placa.
 *
 * Esta tarefa é executada uma única vez na inicialização do sistema. Ela tem a
 * prioridade mais alta e as outras tarefas principais são criadas em estado suspenso,
 * garantindo que esta rotina seja concluída sem interrupções. Ao final, ela
 * retoma (acorda) as outras tarefas e se auto-deleta da memória.
 *
 * @param pvParameters Ponteiro para parâmetros da tarefa (não utilizado neste caso).
 */
void self_test_task(void *pvParameters) {
    printf("--- Iniciando Self-Test (Execucao Exclusiva) ---\n");
    printf("As outras tarefas estao suspensas e aguardando...\n\n");
    vTaskDelay(pdMS_TO_TICKS(2000)); // Pausa para permitir a leitura da mensagem inicial.

    // 1. Teste dos LEDs RGB
    printf("1. Testando LEDs RGB...\n");
    gpio_put(LED_RGB_GREEN_PIN, 1);
    printf("   - LED Verde ON\n");
    vTaskDelay(pdMS_TO_TICKS(1000));
    gpio_put(LED_RGB_GREEN_PIN, 0);

    gpio_put(LED_RGB_BLUE_PIN, 1);
    printf("   - LED Azul ON\n");
    vTaskDelay(pdMS_TO_TICKS(1000));
    gpio_put(LED_RGB_BLUE_PIN, 0);

    gpio_put(LED_ALIVE_PIN, 1);
    printf("   - LED Vermelho ON\n");
    vTaskDelay(pdMS_TO_TICKS(1000));
    gpio_put(LED_ALIVE_PIN, 0);

    printf("   - Teste de LEDs concluido.\n\n");
    vTaskDelay(pdMS_TO_TICKS(1000));

    // 2. Teste do Buzzer
    printf("2. Testando Buzzer...\n");
    gpio_put(BUZZER_PIN, 1);
    vTaskDelay(pdMS_TO_TICKS(500));
    gpio_put(BUZZER_PIN, 0);
    printf("   - Teste de Buzzer concluido.\n\n");
```

```
    vTaskDelay(pdMS_TO_TICKS(1000));

    // 3. Teste dos Botões (lê o estado atual)
    printf("3. Lendo estado dos botoes (0 = Pressionado)...\n");
    printf("   - Botao A: %d\n", gpio_get(BTN_A_PIN));
    vTaskDelay(pdMS_TO_TICKS(500));
    printf("   - Botao B: %d\n", gpio_get(BTN_B_PIN));
    vTaskDelay(pdMS_TO_TICKS(500));
    printf("   - Joystick SW: %d\n", gpio_get(JOYSTICK_SW_PIN));

    printf("   - Teste de botoes concluido.\n\n");
    vTaskDelay(pdMS_TO_TICKS(1000));

    // 4. Teste dos Periféricos Analógicos
    printf("4. Testando perifericos analogicos...\n");
    adc_select_input(0); // Seleciona o canal 0 do ADC (Joystick Y)
    printf("   - Joystick Y (ADC0): Tensao = %.2f V\n", adc_read() * ADC_CONVERSION_FACTOR);
    vTaskDelay(pdMS_TO_TICKS(500));

    adc_select_input(1); // Seleciona o canal 1 do ADC (Joystick X)
    printf("   - Joystick X (ADC1): Tensao = %.2f V\n", adc_read() * ADC_CONVERSION_FACTOR);
    vTaskDelay(pdMS_TO_TICKS(500));

    adc_select_input(2); // Seleciona o canal 2 do ADC (Microfone)
    printf("   - Microfone (ADC2): Tensao = %.2f V\n", adc_read() * ADC_CONVERSION_FACTOR);
    printf("   - Teste de analogicos concluido.\n\n");
    vTaskDelay(pdMS_TO_TICKS(1000));

    // Conclusão da Fase 1 e início da Fase 2
    printf("--- Self-Test Concluido ---\n");
    printf("Retomando tarefas em background (Alive e Joystick)...\n\n");

    /// AÇÃO CRÍTICA: Retoma (acorda) as outras tarefas que estavam suspensas.
    /// A partir deste ponto, o escalonador começará a gerenciá-las.
    vTaskResume(alive_task_handle);
    vTaskResume(joystick_monitor_task_handle);

    /// AÇÃO CRÍTICA: Deleta a tarefa de self-test.
    /// A tarefa libera todos os recursos que estava utilizando (pilha, TCB)
    /// e é removida do gerenciamento do escalonador.
    vTaskDelete(NULL); // O parâmetro NULL significa "delete a si mesma".
}

/**
 * @brief Tarefa 2: Pisca o LED "Alive" para indicar que o sistema está em operação.
 *
 * Esta é uma tarefa de baixa prioridade e baixa frequência que serve como um
 * "heartbeat", um sinal visual de que o processador e o escalonador do FreeRTOS
```

```
 * estão funcionando corretamente.
 *
 * @param pvParameters Ponteiro para parâmetros da tarefa (não utilizado).
 */
void alive_task(void *pvParameters) {
    // Loop infinito é o padrão para tarefas persistentes no FreeRTOS.
    while (1) {
        gpio_put(LED_ALIVE_PIN, 1); // Liga o LED vermelho.

        // Pausa a tarefa por 500 milissegundos.
        // A macro pdMS_TO_TICKS converte milissegundos para "ticks" do sistema,
        // que é a unidade de tempo do escalonador do FreeRTOS.
        vTaskDelay(pdMS_TO_TICKS(500));

        gpio_put(LED_ALIVE_PIN, 0); // Desliga o LED.
        vTaskDelay(pdMS_TO_TICKS(500)); // Pausa por mais 500ms, completando o ciclo de 1s.
    }
}

/**
 * @brief Tarefa 3: Monitora o joystick e aciona um alarme sonoro.
 *
 * Esta tarefa é de alta frequência (executa a cada 50ms). Ela lê as tensões
 * dos eixos X e Y do joystick, imprime na serial e, se qualquer uma delas
 * ultrapassar 3.0V, ativa o buzzer.
 *
 * @param pvParameters Ponteiro para parâmetros da tarefa (não utilizado).
 */
void joystick_monitor_task(void *pvParameters) {
    const float ALARM_THRESHOLD_V = 3.00f; ///< Limite de tensão para acionar o alarme.
    bool alarm_active = false; ///< Variável para controlar o estado do alarme e evitar msgs repetidas.

    while (1) {
        // Leitura e conversão do eixo Y do joystick
        adc_select_input(0);
        float y_volt = adc_read() * ADC_CONVERSION_FACTOR;

        // Leitura e conversão do eixo X do joystick
        adc_select_input(1);
        float x_volt = adc_read() * ADC_CONVERSION_FACTOR;

        // Imprime os valores formatados no monitor serial.
        printf("Joystick -> Y: %.2f V, X: %.2f V\n", y_volt, x_volt);

        // Lógica para ativar/desativar o alarme
        if (y_volt > ALARM_THRESHOLD_V || x_volt > ALARM_THRESHOLD_V) {
            // Imprime a mensagem de alerta apenas na primeira vez que o limite é excedido.
            if (!alarm_active) {
```

```
                printf("!!! ALARME: Tensao do Joystick acima de %.2fV !!!\n", ALARM_THRESHOLD_V);
                alarm_active = true;
            }
            gpio_put(BUZZER_PIN, 1); // Ativa o buzzer.
        } else {
            // Imprime a mensagem de normalização apenas na primeira vez que o valor volta ao normal.
            if (alarm_active) {
                printf("--- Alarme Desativado: Tensao normalizada ---\n");
                alarm_active = false;
            }
            gpio_put(BUZZER_PIN, 0); // Desativa o buzzer.
        }

        // Pausa a tarefa por 50ms, garantindo sua periodicidade.
        vTaskDelay(pdMS_TO_TICKS(50));
    }
}

//=====================================================================================
// Funções de Configuração
//=====================================================================================

/**
 * @brief Inicializa todo o hardware periférico utilizado na aplicação.
 *
 * Esta função configura os pinos GPIO como saída (LEDs, Buzzer) ou entrada
 * (Botões), e inicializa o hardware do ADC para as leituras analógicas.
 * É chamada uma única vez antes de iniciar o escalonador do FreeRTOS.
 *
 * @note A boa prática de agrupar toda a configuração de hardware em uma
 * única função mantém o `main` mais limpo e organizado.
 */
void setup_hardware() {
    // Inicializa saídas digitais (LEDs e Buzzer)
    gpio_init(LED_ALIVE_PIN);
    gpio_set_dir(LED_ALIVE_PIN, GPIO_OUT);
    gpio_init(LED_RGB_GREEN_PIN);
    gpio_set_dir(LED_RGB_GREEN_PIN, GPIO_OUT);
    gpio_init(LED_RGB_BLUE_PIN);
    gpio_set_dir(LED_RGB_BLUE_PIN, GPIO_OUT);
    gpio_init(BUZZER_PIN);
    gpio_set_dir(BUZZER_PIN, GPIO_OUT);

    // Inicializa entradas digitais (Botões)
    // Habilitamos o resistor de pull-up interno para garantir que o pino
    // tenha um nível lógico ALTO (1) quando o botão não está pressionado.
    // Quando pressionado, o botão conecta o pino ao GND, resultando em um nível BAIXO (0).
    gpio_init(BTN_A_PIN);
```

```
    gpio_set_dir(BTN_A_PIN, GPIO_IN);
    gpio_pull_up(BTN_A_PIN);

    gpio_init(BTN_B_PIN);
    gpio_set_dir(BTN_B_PIN, GPIO_IN);
    gpio_pull_up(BTN_B_PIN);

    gpio_init(JOYSTICK_SW_PIN);
    gpio_set_dir(JOYSTICK_SW_PIN, GPIO_IN);
    gpio_pull_up(JOYSTICK_SW_PIN);

    // Inicializa o hardware do ADC e os pinos associados.
    adc_init();
    adc_gpio_init(JOYSTICK_Y_PIN);
    adc_gpio_init(JOYSTICK_X_PIN);
    adc_gpio_init(MICROPHONE_PIN);
}

//=============================================================================
// Função Principal (main)
//=============================================================================

/**
 * @brief Ponto de entrada do programa.
 *
 * A função `main` é responsável por:
 * 1. Inicializar o hardware básico (USB serial e periféricos).
 * 2. Criar todas as tarefas do FreeRTOS, definindo suas prioridades e estado inicial.
 * 3. Iniciar o escalonador do FreeRTOS, que assume o controle do processador.
 *
 * @return int - Teoricamente nunca retorna, pois o escalonador entra em um loop infinito.
 */
int main() {
    // Inicializa todo o hardware necessário.
    setup_hardware();

    // Inicializa a comunicação serial via USB e aguarda um tempo para
    // que o computador possa estabelecer a conexão. Essencial para não perder
    // as primeiras mensagens impressas.
    stdio_init_all();
    sleep_ms(1000);

    printf("====================================\n");
    printf("Iniciando Sistema FreeRTOS na BitDogLab\n");
    printf("====================================\n\n");

    /// Bloco de criação e configuração das tarefas, implementando a lógica de escalonamento.
```

```
    // 1. Cria as tarefas de operação normal, mas seus handles são salvos para
    //    que possam ser manipuladas antes do escalonador iniciar.
    xTaskCreate(alive_task, "AliveTask", 128, NULL, TASK_ALIVE_PRIORITY, &alive_task_handle);
    xTaskCreate(joystick_monitor_task, "JoystickMonitorTask", 256, NULL, TASK_JOYSTICK_PRIORITY,
&joystick_monitor_task_handle);

    // 2. Suspende as tarefas de operação normal. Elas não serão executadas pelo
    //    escalonador até que sejam explicitamente retomadas (resumed).
    vTaskSuspend(alive_task_handle);
    vTaskSuspend(joystick_monitor_task_handle);

    // 3. Cria a tarefa de self-test com a maior prioridade. Como as outras estão
    //    suspensas, esta será a única tarefa "Pronta" (Ready) quando o escalonador iniciar.
    xTaskCreate(self_test_task, "SelfTestTask", 256, NULL, TASK_SELFTEST_PRIORITY, NULL);

    /// Inicia o escalonador do FreeRTOS.
    /// A partir deste ponto, o FreeRTOS toma controle do fluxo de execução.
    /// Esta função nunca retorna.
    vTaskStartScheduler();

    // Este laço infinito é um safeguard e nunca deve ser alcançado.
    while (1) {
        // Laço vazio.
    }
}
```