Aluno do Embarcatech_37 no IFMA

Nome: Manoel Felipe Costa Furtado

Matrícula: 20251RSE.MTC0086

Atividade – Referente ao capítulo 03 da unidade 03

Tema do Capítulo – FreeRTOS - Programação Multithread

Prazo dia 14/07/2025 as 23:59

Enunciado: Sistema de Aquisição usando FreeRTOS no Raspberry Pi Pico W.

Desenvolva um sistema multitarefa no Raspberry Pi Pico W, utilizando o FreeRTOS e o SDK do Pico. O sistema deve realizar a leitura de um joystick analógico (eixos X e Y), detectar o pressionamento de um botão e gerar feedback sonoro por meio de um buzzer. O projeto deve utilizar filas, mutexes e semáforo contador para comunicação e sincronização entre tarefas. A aplicação deverá ser composta por quatro tarefas, conforme descrito a seguir:

Tarefa 1: Leitura dos Eixos (VRx/VRy)

- Lê os valores dos pinos analógicos a cada 100ms.
- Envia os dados para a fila.

Tarefa 2: Leitura do Botão (SW)

- Verifica o estado do botão a cada 50ms.
- Em caso de clique, envia evento para a fila.

Tarefa 3: Processamento dos Dados

- Recebe dados da fila e exibe no terminal (protegido por mutex).
- Aciona o buzzer apenas se houver movimento significativo ou botão pressionado.

Tarefa 4: Controle do Buzzer (Implícita na Tarefa 3)

- Buzzer ligado por 100ms somente se houver evento válido.
- Controlado por semáforo contador com máximo de 2 acessos simultâneos.

Instruções:

- Utilize o FreeRTOS com a biblioteca Pico-SDK.
- utilize as 4 tarefas.
- Implemente uma fila (queue) para comunicação entre as tarefas.

- Use um mutex para proteger o acesso ao terminal serial (printf).
- Use um semáforo contador para controlar o acesso ao buzzer.
- GPIO26: Entrada analógica – VRy (eixo vertical do joystick)
- GPIO27: Entrada analógica – VRx (eixo horizontal do joystick)
- GPIO22: Entrada digital – Botão SW do joystick (pull-up interno ativado)
- GPIO21: Saída digital – Buzzer passivo

- Nome do arquivo principal: “main.c”.
- Link do Código completo:
- GitHub: https://github.com/ManoelFelipe/Embarcatech_37/tree/main/Unidade_03/Cap_03/Atividade_03

Aproveitando para exercitar eu implementei duas características a mais no trabalho para aproveitar o projeto integrador. O SMP e Debounce mais aprimorado.

Checklist de Confirmação do Projeto

Requisito e Implementação no Código

- 4 Tarefas distintas: joystick_task, button_task, processing_task e buzzer_task foram criadas e estão funcionando.
- Tarefa 1: Ler Joystick a 100ms: A joystick_task usa vTaskDelay(pdMS_TO_TICKS(100)) e envia os dados para a fila.
- Tarefa 2: Ler Botão a 50ms: A button_task tem um loop principal com vTaskDelay(pdMS_TO_TICKS(50)) e envia o evento.
- Lógica de Debounce: A button_task possui a lógica aprimorada que confirma a pressão e aguarda o botão ser solto.
- Tarefa 3: Processar Dados: A processing_task recebe da fila e aciona o buzzer para movimento ou clique.
- Tarefa 4: Controle do Buzzer: A buzzer_task fica bloqueada no semáforo e toca o buzzer por 100ms quando liberada.
- Fila (Queue): event_queue é usada para comunicação segura entre os núcleos.
- Mutex: usb_mutex protege todas as chamadas printf nos dois núcleos, evitando conflitos na serial.
- Semáforo Contador: buzzer_sem é criado com contagem máxima de 2 (xSemaphoreCreateCounting(2, 0)).

- Pinagem (GPIOs): Todos os pinos (VRX, VRY, SW, BUZZER) estão definidos corretamente.
- Distribuição SMP: vTaskCoreAffinitySet é usado para fixar as tarefas de I/O no Core 0 e as de processamento no Core 1.
- Tratamento de Erros: A criação de todos os objetos e tarefas do FreeRTOS é verificada, e o critical_error_handler é chamado em caso de falha.

Como entender o projeto:

1. Comece pelo 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
 * @copyright 2025 Manoel Furtado (MIT License) (veja LICENSE.md)
 * @version 1.0
 * @date 2025-07-06
 * @brief Sistema multitarefa com FreeRTOS e SMP no Pico W para leitura de joystick e controle de buzzer.
 *
 * @details
 * Este projeto implementa um sistema com 4 tarefas no Raspberry Pi Pico W usando FreeRTOS,
 * com uma distribuição de tarefas entre os dois núcleos do RP2040 (Symmetric Multiprocessing - SMP).
 * O sistema lê um joystick analógico e um botão, processa os dados e aciona um buzzer.
 * * **Arquitetura SMP:**
 * - **Núcleo 0 (Core 0):** Dedicado às tarefas de entrada (I/O).
 * - `joystick_task`: Lê os eixos X/Y do joystick.
 * - `button_task`: Lê o botão do joystick.
 * - **Núcleo 1 (Core 1):** Dedicado às tarefas de processamento e atuação.
 * - `processing_task`: Processa os eventos da fila.
 * - `buzzer_task`: Controla o buzzer.
 * * **Mecanismos de Sincronização:**
 * - **Fila (Queue):** Uma fila unificada (`event_queue`) é usada para enviar eventos das tarefas de entrada (Núcleo 0) para a tarefa de processamento (Núcleo 1).
 * - **Mutex:** O acesso ao terminal serial (printf) é protegido por um mutex para evitar mensagens corrompidas, já que tarefas em diferentes núcleos poderiam tentar escrever ao mesmo tempo.
 * - **Semáforo Contador:** Gerencia o acesso ao buzzer, permitindo enfileirar até 2 solicitações de som.
 *
 * @note Para compilar este projeto, as seguintes configurações devem estar ativas no arquivo FreeRTOSConfig.h:
 * @code
```

```
 * #define configNUMBER_OF_CORES 2
 * #define configUSE_CORE_AFFINITY 1
 * Defina no FreeRTOSConfig.h , caso não tenha:
 * #define configUSE_PASSIVE_IDLE_HOOK 0
 * @endcode
 */

// --- Inclusão das bibliotecas essenciais ---
#include <stdio.h>        // Biblioteca padrão de entrada/saída
#include "pico/stdlib.h"  // Funções de I/O e temporização do Raspberry Pi Pico
#include "hardware/adc.h" // Controle do ADC interno (para leitura analógica)
#include "FreeRTOS.h"     // Núcleo do FreeRTOS
#include "task.h"         // Funções de criação e controle de tarefas
#include "queue.h"        // Funções de criação e manipulação de filas
#include "semphr.h"       // Biblioteca para semáforos e mutexes

// --- Definições de pinos GPIO ---
#define VRY_PIN 26          // ADC0 para o eixo Y do joystick
#define VRX_PIN 27          // ADC1 para o eixo X do joystick
#define JOYSTICK_SW_PIN 22  // Pino digital para o botão do joystick
#define BUZZER_PIN 21       // Pino digital para o buzzer passivo
#define ERROR_LED_PIN 13    // Pino para o LED de sinalização de erro crítico

// --- Definições para a fila de eventos ---

/** @brief Enumeração para identificar o tipo de evento na fila. */
typedef enum {
    JOYSTICK_EVENT,
    BUTTON_EVENT
} event_type_t;

/** @brief Estrutura de dados para os itens da fila de eventos. */
typedef struct {
    event_type_t type;    // Tipo do evento (joystick ou botão)
    uint16_t data[2];     // Dados (usado para coordenadas do joystick)
} queue_event_t;


// --- Handles globais para objetos FreeRTOS ---
QueueHandle_t event_queue;
SemaphoreHandle_t usb_mutex;
SemaphoreHandle_t buzzer_sem;

// --- Handles para as tarefas (necessários para definir a afinidade de núcleo) ---
TaskHandle_t joystick_task_handle;
TaskHandle_t button_task_handle;
TaskHandle_t processing_task_handle;
TaskHandle_t buzzer_task_handle;
```

```c
// --- Protótipo da função de tratamento de erro ---
void critical_error_handler();

/**
 * @brief Tarefa 1: Lê os eixos X e Y do joystick. (Fixada no Núcleo 0)
 * @details A cada 100ms, esta tarefa lê os valores analógicos dos eixos do joystick,
 * monta um pacote de evento do tipo JOYSTICK_EVENT e o envia para a fila de eventos.
 * @param param Ponteiro para parâmetros da tarefa (não utilizado).
 */
void joystick_task(void *param) {
    adc_init();
    adc_gpio_init(VRX_PIN);
    adc_gpio_init(VRY_PIN);

    while (1) {
        adc_select_input(1); // Canal ADC para VRx
        uint16_t vrx = adc_read();

        adc_select_input(0); // Canal ADC para VRy
        uint16_t vry = adc_read();

        // --- Impressão de Debug (Protegida por Mutex) ---
        // Pega o mutex antes de usar o printf para evitar conflito com o Core 1
        if (xSemaphoreTake(usb_mutex, pdMS_TO_TICKS(100))) {
            printf("CORE 0: Joystick leu X=%d, Y=%d\n", vrx, vry);
            // Devolve o mutex assim que terminar
            xSemaphoreGive(usb_mutex);
        }

        queue_event_t event = {
            .type = JOYSTICK_EVENT,
            .data = {vrx, vry}
        };

        xQueueSend(event_queue, &event, 0);
        vTaskDelay(pdMS_TO_TICKS(100));
    }
}

/**
 * @brief Tarefa 2: Lê o botão do joystick. (Fixada no Núcleo 0)
 * @details A cada 50ms, esta tarefa verifica o estado do botão. Se pressionado,
 * envia um evento do tipo BUTTON_EVENT para a fila e aguarda um tempo
 * para debounce, evitando múltiplos acionamentos.
 * @param param Ponteiro para parâmetros da tarefa (não utilizado).
 */
void button_task(void *param) {
    gpio_init(JOYSTICK_SW_PIN);
```

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

    while (1) {

        // 1. Procura pela borda de descida (início do pressionamento)
        if (!gpio_get(JOYSTICK_SW_PIN)) {

            // 2. DEBOUNCE: Aguarda um curto período para filtrar a vibração mecânica.
            vTaskDelay(pdMS_TO_TICKS(50));

            // 3. CONFIRMAÇÃO: Verifica se o botão AINDA está pressionado após o delay.
            // Isso confirma que não foi apenas um ruído passageiro.
            if (!gpio_get(JOYSTICK_SW_PIN)) {

                // --- AÇÃO: Pressionamento confirmado ---
                // --- Impressão de Debug (Protegida por Mutex) ---
                if (xSemaphoreTake(usb_mutex, pdMS_TO_TICKS(100))) {
                    printf("CORE 0: Botao detectado!\n");
                    xSemaphoreGive(usb_mutex);
                }

                queue_event_t event = {
                    .type = BUTTON_EVENT,
                    .data = {0, 0} // Dados não são usados para este evento
                };
                xQueueSend(event_queue, &event, 0);

                // 4. AGUARDA SOLTAR: Loop que espera o botão ser fisicamente solto.
                // Isso é crucial para garantir que apenas UM evento seja enviado por
                // cada ato de pressionar, não importa quanto tempo o botão fique seguro.
                while(!gpio_get(JOYSTICK_SW_PIN)) {
                    vTaskDelay(pdMS_TO_TICKS(50)); // Espera em blocos de 50ms para não sobrecarregar a CPU
                }
                if (xSemaphoreTake(usb_mutex, pdMS_TO_TICKS(100))) {
                    printf("CORE 0: Botao Solto!\n");
                    xSemaphoreGive(usb_mutex);
                }

            }
        }
        // Pausa principal da tarefa para ceder tempo de CPU
        vTaskDelay(pdMS_TO_TICKS(50));
    }
}

/**
 * @brief Tarefa 4: Controla o acionamento do buzzer. (Fixada no Núcleo 1)
```

```
 * @details Esta tarefa fica bloqueada aguardando o semáforo do buzzer.
 * Quando o semáforo é liberado pela tarefa de processamento, esta tarefa
 * aciona o buzzer por 100ms e depois volta a esperar.
 * @param param Ponteiro para parâmetros da tarefa (não utilizado).
 */
void buzzer_task(void *param) {
    gpio_init(BUZZER_PIN);
    gpio_set_dir(BUZZER_PIN, GPIO_OUT);

    while (1) {
        // Bloqueia indefinidamente até que o semáforo esteja disponível.
        // xSemaphoreTake é uma chamada de bloqueio segura que cede a CPU.
        if (xSemaphoreTake(buzzer_sem, portMAX_DELAY) == pdTRUE) {
            gpio_put(BUZZER_PIN, 1);
            vTaskDelay(pdMS_TO_TICKS(100)); // Buzzer ligado por 100ms, conforme requisito
            gpio_put(BUZZER_PIN, 0);
        }
    }
}

/**
 * @brief Tarefa 3: Processa os dados recebidos da fila. (Fixada no Núcleo 1)
 * @details Esta tarefa aguarda por eventos na fila. Ao receber um evento,
 * ela o processa: imprime no terminal serial (protegido por mutex) e,
 * se for um evento de botão ou um movimento significativo do joystick,
 * libera o semáforo para acionar o buzzer.
 * @param param Ponteiro para parâmetros da tarefa (não utilizado).
 */
void processing_task(void *param) {
    queue_event_t received_event;
    // Define uma "zona morta" para o joystick para evitar acionamentos por ruído
    // Valores típicos para um ADC de 12-bit (0-4095) com centro em ~2048
    const int dead_zone_low = 1000;
    const int dead_zone_high = 3000;

    while (1) {
        // Bloqueia a tarefa até que um item seja recebido da fila.
        // portMAX_DELAY significa esperar para sempre, se necessário.
        if (xQueueReceive(event_queue, &received_event, portMAX_DELAY) == pdTRUE) {

            int trigger_buzzer = 0;

            // Tenta obter o mutex para usar a porta serial. Espera até 100ms.
            // Essencial em um sistema SMP para evitar que duas tarefas em núcleos diferentes
            // escrevam na serial ao mesmo tempo, corrompendo a saída.
            if (xSemaphoreTake(usb_mutex, pdMS_TO_TICKS(100))) {
                switch (received_event.type) {
                    case JOYSTICK_EVENT:
```

```
                        // A impressão da leitura foi movida para a tarefa no Core 0
                        // Esta tarefa agora apenas imprime o evento que processa
                        printf("CORE 1: Joystick - X: %d, Y: %d\n", received_event.data[0], received_event.data[1]);
                        if (received_event.data[0] < dead_zone_low || received_event.data[0] > dead_zone_high ||
                            received_event.data[1] < dead_zone_low || received_event.data[1] > dead_zone_high) {
                            trigger_buzzer = 1;
                        }
                        break;
                    case BUTTON_EVENT:
                        printf("CORE 1: Processando evento de BOTAO.\n");
                        trigger_buzzer = 1;
                        break;
                }
                // Libera o mutex assim que terminar de usar a serial.
                xSemaphoreGive(usb_mutex);
            }

            if (trigger_buzzer) {
                // Libera ("dá") o semáforo para a buzzer_task tocar o som.
                // Se o semáforo já estiver no seu valor máximo (2), esta chamada não fará nada
                // e não bloqueará a tarefa.
                xSemaphoreGive(buzzer_sem);
            }
        }
    }
}

/**
 * @brief Sinaliza um erro crítico de inicialização piscando um LED.
 * @details Esta função é chamada se a alocação de um recurso essencial do
 * FreeRTOS falhar. Ela entra em um loop infinito, impedindo a execução
 * do programa principal e alertando o desenvolvedor sobre a falha.
 */
void critical_error_handler() {
    // Inicializa o pino do LED de erro
    gpio_init(ERROR_LED_PIN);
    gpio_set_dir(ERROR_LED_PIN, GPIO_OUT);
    // Loop infinito para piscar o LED e sinalizar a falha
    while (true) {
        gpio_put(ERROR_LED_PIN, 1);
        sleep_ms(200);
        gpio_put(ERROR_LED_PIN, 0);
        sleep_ms(200);
    }
}



/**
```

```
 * @brief Função principal, ponto de entrada do programa.
 * @details Orquestra toda a inicialização do sistema:
 * 1. Inicializa a E/S padrão.
 * 2. Cria os objetos de sincronização do FreeRTOS (fila, mutex, semáforo), com verificação de erro.
 * 3. Cria as 4 tarefas de aplicação, com verificação de erro.
 * 4. Define a afinidade de núcleo para cada tarefa (SMP).
 * 5. Inicia o escalonador do FreeRTOS.
 * @return int Nunca retorna.
 */
int main() {
    // Inicializa as funções padrão do Pico, incluindo a comunicação USB para o printf
    stdio_init_all();
    // Pausa para dar tempo de conectar um terminal serial e ver as mensagens de inicialização
    sleep_ms(2000);

    printf("Iniciando sistema com FreeRTOS e SMP...\n");
    printf("Tarefas de entrada no Core 0 | Tarefas de processamento no Core 1\n");

    // --- Criação dos Objetos FreeRTOS ---
    // Cria uma fila para até 10 eventos. Cada evento tem o tamanho da struct queue_event_t.
    printf("Criando fila de eventos...");
    event_queue = xQueueCreate(10, sizeof(queue_event_t));
    if (event_queue == NULL) {
        printf("ERRO CRITICO: Falha ao criar a fila de eventos.\n");
        critical_error_handler();
    }

    // Cria um mutex para proteger o acesso concorrente à USB/printf.
    printf("Criando mutex da USB...");
    usb_mutex = xSemaphoreCreateMutex();
    if (usb_mutex == NULL) {
        printf("ERRO CRITICO: Falha ao criar o mutex da USB.\n");
        critical_error_handler();
    }

    // Cria um semáforo do tipo "contador".
    // Parâmetro 1 (2): Contagem máxima. Até 2 "gives" podem ser acumulados.
    // Parâmetro 2 (0): Contagem inicial. O semáforo começa vazio.
    printf("Criando semaforo do buzzer...");
    buzzer_sem = xSemaphoreCreateCounting(2, 0);
    if (buzzer_sem == NULL) {
        printf("ERRO CRITICO: Falha ao criar o semaforo do buzzer.\n");
        critical_error_handler();
    }

    // --- Criação das Tarefas ---
    // (Função, Nome, Tam. Pilha, Parâmetro, Prioridade, Handle para a tarefa)
```

```
    printf("Criando tarefas de aplicacao...\n");
    if (xTaskCreate(joystick_task, "JoystickTask", 256, NULL, 1, &joystick_task_handle) != pdPASS) {
        printf("ERRO CRITICO: Falha ao criar a joystick_task.\n");
        critical_error_handler();
    }

    if (xTaskCreate(button_task, "ButtonTask", 256, NULL, 1, &button_task_handle) != pdPASS) {
        printf("ERRO CRITICO: Falha ao criar a button_task.\n");
        critical_error_handler();
    }

    if (xTaskCreate(processing_task, "ProcessingTask", 512, NULL, 1, &processing_task_handle) != pdPASS) {
        printf("ERRO CRITICO: Falha ao criar a processing_task.\n");
        critical_error_handler();
    }

    if (xTaskCreate(buzzer_task, "BuzzerTask", 256, NULL, 2, &buzzer_task_handle) != pdPASS) {
        printf("ERRO CRITICO: Falha ao criar a buzzer_task.\n");
        critical_error_handler();
    }
    printf("Todas as tarefas foram criadas com sucesso.\n");

    // --- Configuração da Afinidade de Núcleo (SMP) ---
    // Fixa as tarefas de leitura de sensores para rodar exclusivamente no Núcleo 0.
    // A máscara (1 << 0) representa o Core 0.
    vTaskCoreAffinitySet(joystick_task_handle, (1 << 0));
    vTaskCoreAffinitySet(button_task_handle,  (1 << 0));

    // Fixa as tarefas de processamento e atuação para rodar exclusivamente no Núcleo 1.
    // A máscara (1 << 1) representa o Core 1.
    vTaskCoreAffinitySet(processing_task_handle, (1 << 1));
    vTaskCoreAffinitySet(buzzer_task_handle,     (1 << 1));

    printf("Afinidade de núcleo configurada. Iniciando escalonador...\n");

    // Inicia o escalonador do FreeRTOS. A partir daqui, o FreeRTOS toma o controle.
    printf("--- INICIALIZACAO CONCLUIDA ---\n\n");
    vTaskStartScheduler();

    // Este loop infinito é uma salvaguarda. O programa nunca deve chegar aqui,
    // a menos que haja um erro crítico na inicialização do FreeRTOS (ex: falta de memória).
    while (1);
}
```