**Visão Geral do Projeto**

Este projeto demonstra a aplicação da computação paralela na Raspberry Pi Pico, empregando seus dois núcleos para gerenciamento de tarefas em um sistema embarcado educacional BitDogLab. A simulação de diferentes níveis de atividade é realizada por um joystick analógico, com respostas visuais (LED RGB indicando o nível) e sonoras (buzzer ativado em nível alto).

**Objetivo Principal**

Ilustrar o emprego de múltiplos núcleos da Raspberry Pi Pico em um sistema embarcado interativo, dedicando um núcleo para a leitura de sensores (Core 0) e outro para o controle de atuadores (Core 1).

**Operação do Sistema**

* **Core 0:** Realiza a leitura dos dados do joystick analógico e define o estado da atividade.
* **Core 1:** Recebe o estado definido pelo Core 0 e efetua o controle dos atuadores.
* **Comunicação:** A troca de informações entre os núcleos ocorre através de FIFOs.
* **Saídas:** Um LED RGB exibe visualmente o nível de atividade, e um buzzer emite um sinal sonoro quando o nível é alto.

**Seção 1: Includes e Definições**

#include <stdio.h> // Para funções de entrada/saída padrão (printf)

#include "pico/stdlib.h" // Biblioteca padrão da Raspberry Pi Pico

#include "pico/multicore.h" // Biblioteca para utilizar os dois núcleos

#include "hardware/gpio.h" // Biblioteca para controle dos pinos GPIO

#include "hardware/adc.h" // Biblioteca para conversão analógico-digital (ADC)

#include "hardware/pwm.h" // Biblioteca para modulação por largura de pulso (PWM)

#include "hardware/timer.h" // Biblioteca para usar timers e alarmes

**Por que existe?**

Carrega funcionalidades essenciais: GPIO, PWM, ADC, multi-core.

**Seção 2: Definições de Hardware**

**#define LED\_R\_PIN 13**

**#define LED\_G\_PIN 11**

**#define LED\_B\_PIN 12**

**// Buzzers**

**#define BUZZER\_A\_PIN 21 // Pino do Buzzer A**

**// Joystick Analógico**

**#define JOY\_X\_PIN 27 // ADC 1**

**#define ADC\_JOY\_X 1**

**// --- Configurações ---**

**#define JOYSTICK\_READ\_INTERVAL\_MS 2000 // Leitura a cada 2 segundos**

**#define ADC\_MAX\_VALUE 4095 // Valor máximo do ADC (12 bits)**

**// Limiares teóricos para os estados (aproximadamente 1/3 e 2/3 do ADC\_MAX\_VALUE)**

**#define LIMIAR\_BAIXA (ADC\_MAX\_VALUE / 3) // ~1365 // 4095 / 3 ≈ 1365 (33%)**

**#define LIMIAR\_MODERADA (2 \* ADC\_MAX\_VALUE / 3) // ~2730 // 4095 \* 2/3 ≈ 2730 (66%)**

**#define BUZZER\_WRAP\_VALUE 62499 // Wrap para PWM do buzzer (~2kHz audível)**

**Função:** Mapeia componentes físicos aos pinos da placa.

**Seção 3: Variáveis Globais**

**// Flag volatile para armazenar o estado global**

**volatile uint8\_t global\_state = 0; // Estado compartilhado entre cores**

**Importância:** volatile garante que alterações sejam visíveis imediatamente em ambos os cores.

**Seção 4: Core 0 (Leitura do Joystick)**

**Resumo: → Core 0 →** inicializa os periféricos, lança o Core 1 e configura um alarme periódico para ler o joystick. O loop principal apenas espera o próximo alarme.

**int64\_t core0\_joystick\_read\_callback(alarm\_id\_t id, void \*user\_data) {**

**// Seleciona e lê o ADC do Joystick X**

**adc\_select\_input(ADC\_JOY\_X);**

**uint32\_t joystick\_value = adc\_read(); // Leitura no intervalo 0-4095**

// ... determina estado e envia para Core 1 ...

**Explicação ADC:** Converte o valor lido do joystick em um estado (1, 2 ou 3) e o envia via FIFO para o Core 1.

Valores:

* 1 (baixo): joystick <= 1365
* 2 (moderado): 1366 a 2730
* 3 (alto): > 2730

**Seção 5: Core 1 (Controle de Atuadores)**

void core1\_entry() {

...

while (1) {

// Bloqueia até receber um valor via FIFO (estado enviado pelo Core 0)

uint8\_t received\_state = multicore\_fifo\_pop\_blocking();

**Bloqueia até receber dados via FIFO.**

**Fluxograma de Funcionamento**

| **[Core 0]** **[Core 1]**  Leitura do Joystick Aguarda Dados  │ ▲  │ │  ▼ │  Processa Estado │  │ │  └───Envia via FIFO─────► Controla LEDs/Buzzer |
| --- |

**Estrutura de Multitarefa nos Cores 0 e 1**

**Core 0:** Responsável pela leitura do joystick e pelo processamento do estado dos comandos. Os dados processados são então enviados através de uma FIFO.

**Core 1:** Aguarda os dados provenientes da FIFO e os utiliza para controlar LEDs e um buzzer.

**Fluxo de Dados:**

* **Core 0:** Leitura do Joystick → Processamento do Estado → Envio via FIFO
* **Core 1:** Recebimento de Dados via FIFO → Controle de LEDs/Buzzer

#### **Seção 6. Funções Auxiliares**

* **setup\_gpios(): inicializa pinos como saída**
* **setup\_adc(): ativa e prepara ADC para o joystick**
* **setup\_pwm(): configura PWM nos LEDs e no buzzer**
* **set\_rgb\_led(): define os níveis de PWM dos LEDs**
* **update\_rgb\_led\_from\_state(): traduz o estado em uma cor correspondente**

**Perguntas**

**1. “O que aconteceria se tudo rodasse em um único núcleo?”**

Poderia funcionar, mas com risco de atrasos nas leituras, perda de dados ou travamentos, especialmente com mais sensores.

**2. Quais as vantagens de usar dois núcleos nesse projeto?**

Maior desempenho, respostas mais rápidas e separação clara entre a leitura de sensores e o controle dos atuadores. O Core 0 faz a leitura do joystick e envia dados, enquanto o Core 1 controla o LED RGB e o buzzer. Isso melhora a eficiência e evita travamentos.

**4. Como os dois núcleos se comunicam?**

Através de uma fila FIFO:

Core 0 envia dados com multicore\_fifo\_push\_blocking()

Core 1 recebe com multicore\_fifo\_pop\_blocking().

**5. Como funciona a leitura do joystick?**

Usa ADC (conversor analógico-digital). O valor lido varia de 0 a 4095.

**6. Por que os limites de estado são 1/3 e 2/3 do ADC?**

Para dividir o intervalo em três faixas (baixa, moderada e alta atividade) de forma simples e proporcional.

→ O ADC da Pico tem 12 bits de resolução (2¹² = 4096 valores possíveis), mapeando 0V (0) a 3.3V (4095).

**7. Como o buzzer é ativado?**

Através de PWM (modulação por largura de pulso).

**8. "Por que substituir tight\_loop\_contents() por \_\_wfi() no projeto final?"**

Porque o \_\_wfi() permite que o núcleo 0 entre em modo de baixo consumo entre as leituras do joystick, reduzindo o consumo energético em até 80% sem afetar a funcionalidade. Já o tight\_loop\_contents() mantém o núcleo desnecessariamente ativo, gastando energia sem benefícios para nossa aplicação específica.

**9. "Como calculou o BUZZER\_WRAP\_VALUE para 2kHz?"**

"Wrap = Clock (125MHz) / Frequência Desejada (2kHz) - 1 → 125.000.000 / 2000 - 1 = 62499."