embarca tech | Residência Tecnológica em Sistemas Embarcados

Clock e Interruptores

Unidade 4 | Capítulo 5

Pedro Henrique Almeida Miranda

Executores: Instituto Federal Piauí; Instituto Federal Rio Grande do Norte; Instituto Federal Maranhão; Instituto Federal Ceará; cepedi; Instituto Hardware BR

Coordenação: Softex

Iniciativa: Ministério da Ciência, Tecnologia e Inovação; Governo Federal Brasil União e Reconstrução

Sumário

Unidade 4

Capítulo 5

1. Boas-vindas e Introdução

Olá, estudante do EmbarcaTech!
Seja bem-vindo a mais uma etapa do nosso curso de capacitação. Nesta unidade, vamos explorar práticas avançadas sobre clocks e temporizadores. O foco desta aula tem como objetivo proporcionar uma compreensão detalhada de como os clocks e temporizadores funcionam no Raspberry Pi Pico W. Exploraremos os clocks e temporizadores no RP2040, abordando suas características, exemplos práticos, e como utilizá-los em diferentes projetos de sistemas embarcados.

Com base nos conhecimentos adquiridos anteriormente sobre interrupções e GPIOs, estamos prontos para dar mais um passo em direção ao desenvolvimento de sistemas embarcados. Nesta aula, você terá a oportunidade de aplicar a programação em C para realizar projetos práticos e desafiadores. Vamos aprender a controlar o tempo de forma mais eficiente e precisa. Aproveite este material para aprofundar seus conhecimentos e experimente configurar os clocks e temporizadores no seu Raspberry Pi Pico W. Vamos explorar juntos o potencial desse incrível microcontrolador!

Vamos lá?

2. Objetivos desta Unidade

Ao final desta aula, espera-se que você tenha desenvolvido a capacidade de utilizar técnicas de temporização usando temporizadores (hardware ou software) em sistemas embarcados com o Raspberry Pi Pico [2] ou com a plataforma educacional BitDogLab[1], configurando e controlando de forma eficiente o tempo e, as portas de entrada e saída (GPIO) para interagir com diversos dispositivos eletrônicos.

Essas práticas permitirão que você adquira habilidades fundamentais para a resolução de problemas recorrentes em sistemas embarcados, como a sincronização de tarefas e a otimização dos recursos do microcontrolador. Você estará apto a aplicar essas técnicas em projetos práticos, utilizando tanto o Raspberry Pi Pico quanto a BitDogLab, expandindo sua capacidade de criar soluções inovadoras e eficazes para os desafios reais no desenvolvimento de sistemas embarcados.

3. Clock

O clock de um microcontrolador é como o seu coração, ditando o ritmo de todas as suas operações. No caso do Raspberry Pi Pico W, o microcontrolador RP2040 possui uma arquitetura de clock bastante flexível e complexa, permitindo a configuração e o ajuste de múltiplos clocks para atender diferentes necessidades.

O clock define a velocidade com que o microcontrolador executa as instruções. Ele é essencial para sincronizar todas as operações internas e garantir que os componentes funcionem em harmonia. No RP2040, o clock principal é gerado por um oscilador de cristal de 12 MHz, que é multiplicado internamente para gerar clocks mais rápidos, como 125 MHz ou até mais, dependendo da aplicação.

Essa frequência base pode ser alterada por meio de circuitos chamados PLLs (Phase-Locked Loops), que permitem aumentar ou diminuir a frequência para atender às necessidades específicas de cada aplicação. Isso permite que o microcontrolador opere de maneira eficiente, balanceando consumo de energia e desempenho.
O RP2040 possui vários subsistemas de clock que fornecem frequências independentes para diferentes periféricos, como GPIO, UART, SPI, I2C,

PWM e ADC. Cada um desses clocks pode ser configurado de forma independente, permitindo que cada periférico opere na frequência ideal para sua função. O RP2040 possui um sistema de clocks flexível e preciso, composto por vários componentes:

Clock Principal (Sys Clock)	O RP2040 pode operar com um clock principal de até 133 MHz. Este é o clock que é utilizado para a maioria das operações do processador e do sistema.
Clock do Núcleo (Core Clock)	O RP2040 possui dois núcleos ARM Cortex-M0+ que compartilham um clock principal. O clock de cada núcleo é derivado do clock principal, e ambos os núcleos operam na mesma frequência.
Clock de Periféricos	O RP2040 também tem clocks separados para periféricos, como o ADC, UARTs e SPI. Esses clocks podem ser ajustados independentemente do clock principal para otimizar o desempenho e a eficiência.
Clock de Sistema (Sys Tick)	O sistema também inclui um temporizador de sistema, conhecido como SysTick, que pode ser usado para gerar interrupções periódicas.

Para configurar os clocks no Raspberry Pi Pico W, utilizamos o SDK (Software Development Kit) que oferece funções específicas para essa tarefa. Vamos abordar os passos básicos para configuração do clock, utilizando o PLL e ajustando a frequência dos diferentes clocks.

SAIBA MAIS

Dica de Leitura: Para uma visão mais detalhada sobre os fundamentos do clock confira o documento datasheet. Esta é a documentação oficial do microcontrolador RP2040, que descreve em detalhes a arquitetura, incluindo os sistemas de clock, temporizadores, periféricos e muito mais. É uma referência essencial para qualquer pessoa que esteja desenvolvendo com o RP2040.

3.1. PLL

O PLL (Phase-Locked Loop) é um componente crucial que permite ao RP2040 aumentar a frequência do clock base de 12 MHz para valores maiores, como 125 MHz ou até mais. Ele faz isso multiplicando a frequência base de acordo com um fator de multiplicação definido pelo usuário.

• Clique aqui para ver exemplo 01 – Configurando o PLL para 125 MHz (página 12)

Neste exemplo, o PLL é configurado para multiplicar a frequência base de 12 MHz para 125 MHz. Isso é feito definindo os parâmetros do pll_init() e aplicando-os ao clock principal com clock_configure().

• SAIBA MAIS

Sabia mais em: Hardware APIs - Raspberry Pi Documentation

4. Temporizadores

Os temporizadores são elementos essenciais em sistemas embarcados, permitindo a execução de tarefas em intervalos específicos sem a necessidade de intervenção constante do processador. O RP2040 possui temporizadores que podem ser usados para controlar eventos periódicos de forma eficiente.
O RP2040 oferece uma variedade de temporizadores que podem ser usados para diferentes propósitos:

Temporizadores de Hardware (Timer Hardware)	O RP2040 tem vários temporizadores de hardware que podem ser usados para gerar interrupções temporizadas ou medir o tempo com precisão. Esses temporizadores são configuráveis e oferecem uma ampla gama de possibilidades para gerar eventos baseados em tempo.
Temporizador de Sistema (SysTick)	Este temporizador é útil para gerar interrupções regulares. É frequentemente usado para implementar sistemas de tempo real ou para criar intervalos regulares para execução de tarefas.

Temporizadores de Pulse Width Modulation (PWM)	O RP2040 possui blocos de PWM que permitem gerar sinais PWM com precisão. Isso é útil para controlar servomecanismos, motores e outros dispositivos baseados em PWM. Esse assunto será visto no capítulo 7.
Temporizadores de Contagem (Counters)	O RP2040 pode ser configurado para contar eventos ou pulsos, o que é útil para aplicações que requerem a medição precisa de frequência ou eventos.

4.1. Utilizando Temporizadores no código

Utilizaremos a plataforma **BitDogLab** e **Wokwi** para ilustrar exemplos práticos, oferecendo um ambiente de aprendizado integrado e interativo. A BitDogLab, equipada com diversos periféricos como displays, botões e LEDs, será o suporte ideal para experimentarmos as funcionalidades de temporização e controlarmos o comportamento dos dispositivos em tempo real. Vamos ver alguns exemplos básicos utilizando temporizadores:

• Clique aqui para ver exemplo 02 Temporizador de Sistema (página 14)

• Clique aqui para ver exemplo 03 Temporizador de Hardware (página 16)

Comparativo entre os dois exemplos:
Os dois programas apresentados estão realizando a mesma tarefa básica: imprimir uma mensagem a cada segundo, mas usam abordagens diferentes para controlar o tempo. Vou detalhar as diferenças entre as duas implementações:

Primeiro Programa (Cap5_Ex2): delayed_by_us() e time_reached()

O primeiro código usa funções relacionadas ao tempo absoluto e verificações periódicas do tempo (time_reached()). Ele define um tempo futuro (next_wake_time) e continuamente verifica se esse tempo foi alcançado. Quando o tempo é alcançado, uma mensagem é impressa, e o tempo futuro é ajustado para o próximo intervalo.

Vantagem:

- **Simplicidade:** Essa abordagem é simples, pois você controla o tempo diretamente no loop principal.
- **Controle Flexível:** Como o loop está verificando continuamente o tempo, é fácil ajustar o comportamento ou adicionar mais lógica ao programa a qualquer momento.

Desvantagem:

- **Uso de CPU:** Embora o programa tenha uma pequena pausa (sleep_ms(1)), ele ainda está executando o loop principal continuamente, o que pode resultar em maior uso de CPU, mesmo quando o programa não está fazendo algo significativo.
- **Bloqueante para outras tarefas:** Se você quiser executar várias tarefas no mesmo loop, precisa gerenciar manualmente as condições de tempo e as execuções.

Segundo Programa (Cap5_Ex3): add_repeating_timer_ms()

O segundo código usa um temporizador de hardware repetitivo para acionar uma função (repeating_timer_callback()) a cada segundo. O temporizador é configurado para acionar automaticamente a função de callback, que imprime a mensagem.

Vantagem:

- **Menor Uso de CPU:** O temporizador está sendo gerenciado por hardware, então o loop principal não precisa fazer verificações contínuas de tempo. Isso libera o processador para outras tarefas.
- **Desacoplamento do Loop Principal:** A lógica de temporização está separada do loop principal. Isso significa que o loop pode ser usado para outras tarefas sem precisar lidar diretamente com o tempo.
- **Interrupções:** Usar temporizadores permite responder a eventos (neste caso, o tempo) de forma mais eficiente, pois o temporizador gera uma interrupção quando o tempo é atingido.

Desvantagem:

- **Complexidade Adicional:** O uso de temporizadores pode ser mais difícil de entender para iniciantes, uma vez que envolve o uso de callbacks e gerenciamento de interrupções.

Após a compreensão, configuração e aplicação de dois modos de configuração dos temporizadores, podemos utilizar ambos os códigos apresentados para controlar um LED, vejamos a seguir:

- Clique aqui para ver exemplo 04
  Temporizador de sistema para alternar um LED (página 18)

- Clique aqui para ver exemplo 05
  Temporizador de hardware para alternar um LED (página 21)

Para mudar um pouco a lógica de utilização dos temporizadores, iremos utilizá-los para controlar o tempo em que o LED ficará ligado. Assim, vamos implementar um código que acenda um LED por alguns segundos quando um botão é pressionado e utilizar o temporizador para desligar o LED automaticamente após esse período, vejamos a seguir:

- Clique aqui para ver exemplo 06
  Controle temporizado de LED com botão e desligamento automático utilizando o temporizador de sistema (página 24)

- Clique aqui para ver exemplo 07
  Controle temporizado de LED com botão e desligamento automático utilizando o temporizador de hardware (página 27)

- Clique aqui para ver exemplo 08
  Controle de LED com duplo clique e temporização automática (página 31)

O programa controla um LED e um botão no Raspberry Pi Pico. O código monitora o botão a cada 100 ms e acende o LED após duas pressões consecutivas. O LED permanece aceso por 4 segundos e depois é desligado automaticamente.

5. Aplicações de Clocks e Temporizadores

Compreender e configurar corretamente os clocks e temporizadores é essencial para o desenvolvimento de aplicações embarcadas eficientes. Vamos agora explorar algumas aplicações práticas onde esses conhecimentos podem ser aplicados.

- Controle de Motores com PWM: Configurar o clock do módulo PWM para gerar sinais precisos que controlam a velocidade e direção de motores em robôs ou drones.
- Amostragem de Sensores com Temporizadores: Usar temporizadores de hardware para garantir a leitura periódica de sensores de temperatura ou umidade, sem sobrecarregar o processador.
- Economia de Energia: Ajustar dinamicamente os clocks para reduzir o consumo de energia durante períodos de inatividade, mantendo apenas os clocks necessários ativos.

SINTESE

Revisando, aprendemos que o clock e o temporizador são essenciais para gerenciar e integrar diferentes dispositivos em sistemas embarcados. Exploramos as principais configurações e aplicações dos clocks e dos temporizadores no RP2040 da Raspberry Pi Pico W. Através de exemplos práticos, vimos como configurar e programar os temporizadores, de forma a otimizar tempo e tornar o código mais eficiente. Ao compreender suas características e aplicações, você estará mais preparado para desenvolver projetos robustos e escaláveis.

Não se esqueça de revisar o material e testar os exemplos práticos apresentados neste e-book. Experimente configurar os temporizadores em seus próprios projetos e explore as diferentes combinações de dispositivos. Para consolidar o conhecimento adquirido, acesse a plataforma de aprendizagem e complete a atividade prática disponível. Mãos à obra!

7. Conclusão

Os clocks e temporizadores são **recursos fundamentais** em sistemas embarcados, e o Raspberry Pi Pico W oferece uma flexibilidade incrível para configurá-los. Ao compreender como ajustar e utilizar esses recursos, você pode desenvolver aplicações mais eficientes, precisas e econômicas. Continue explorando e experimentando as possibilidades

que o controle de tempo e frequência oferecem, e você verá como esses conceitos podem transformar seus projetos! Aproveite este material para aprofundar seus conhecimentos e experimente configurar os clocks e temporizadores no seu Raspberry Pi Pico W. Sua participação e dedicação são essenciais para o sucesso em projetos de sistemas embarcados. Continue explorando, praticando e se desafiando a criar soluções inovadoras. Até a próxima!

Referências

[1] BITDOGLAB. BitDogLab. Disponível em: https://github.com/BitDogLab/BitDogLab. Acesso em: 26 set. 2024.
[2] RASPBERRY PI FOUNDATION. RP2040 Datasheet. Disponível em: https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf. Acesso em: 25 set. 2024.
[3] GAY, Warren. Debouncing. In: GAY, Warren. Exploring the Raspberry Pi 2 with C++. 1. ed. New York: Apress, 2015. p. 105-112.
[4] CIRCUIT BASICS. How to set up a keypad on an Arduino. Disponível em: https://www.circuitbasics.com/how-to-set-up-a-keypad-on-an-arduino/. Acesso em: 26 set. 2024.

Exemplo 1

Resumo do exemplo

- O código inicializa o sistema de clocks do microcontrolador RP2040 e configura o PLL do sistema para gerar um clock de 125 MHz.
- A partir dessa configuração, o clk_sys do microcontrolador passa a operar a 125 MHz.
- O código está preparado para adicionar mais funcionalidades dentro do loop principal infinito.

Link do programa comentado:
https://wokwi.com/projects/409672647153452033

A seguir, vamos analisar detalhadamente cada parte do código:

Inclusão de Bibliotecas

- **pico/stdlib.h:** Inclui a biblioteca padrão da plataforma Raspberry Pi Pico. Fornece funções básicas de entrada/saída, controle de GPIO, temporização, etc.
- **hardware/clocks.h:** Inclui as definições e funções relacionadas ao gerenciamento dos clocks internos do microcontrolador RP2040.
- **hardware/pll.h:** Fornece funções e definições para controlar e configurar os Phase-Locked Loops (PLLs), que são usados para gerar frequências de clock mais altas a partir de uma frequência de referência.

Função Principal main()

A função principal inicializa o hardware e configura o sistema de clock. Vamos detalhar cada etapa:

- **stdio_init_all();**: Inicializa todas as interfaces de comunicação padrão, como UART e USB, para permitir o uso de funções de E/S padrão, como printf. Isso é útil para depuração e comunicação serial.
- **clocks_init();**: Inicializa o sistema de gerenciamento de clocks do microcontrolador. Esta função configura os clocks básicos, como o relógio de referência e os divisores de frequência padrão.
  - » pll_init(pll_sys, 1, 1500 * MHZ, 6 * MHZ, 1);: Configura o PLL do sistema (pll_sys) para gerar um clock de 125 MHz. Aqui está o detalhamento

dos parâmetros:
  - » pll_sys: O PLL do sistema que será configurado.
  - » 1: O valor de pós-divisor. No RP2040, isso normalmente define o número de fases de saída (usualmente 1 para uma fase única).
  - » 1500 * MHZ: A frequência alvo do PLL (1500 MHz).
  - » 6 * MHZ: A frequência de referência de entrada do PLL, geralmente derivada do cristal externo de 12 MHz.
  - » 1: O pré-divisor de entrada (divisor da frequência de referência). Aqui, é 1, o que significa que a frequência de referência de entrada do PLL será 6 MHz (12 MHz/2).

O PLL usa a fórmula: Frequência de saída= (Frequência de referência ×Multiplicador) / (Divisor)

Para este exemplo: Frequência de saída=(6×250) / (1)=1500 MHz
E depois, o clock é dividido para obter 125 MHz.

- **clock_configure(clk_sys, ...):** Configura o clock do sistema (clk_sys) para usar o PLL configurado como fonte de clock. Vamos detalhar os parâmetros:
  - » clk_sys: O clock do sistema a ser configurado.
  - » CLOCKS_CLK_SYS_CTRL_SRC_VALUE_CLKSRC_CLK_SYS_AUX: Configura o clock de origem para um clock auxiliar.
  - » CLOCKS_CLK_SYS_CTRL_AUXSRC_VALUE_CLKSRC_PLL_SYS: Define o PLL do sistema (pll_sys) como fonte auxiliar de clock.
  - » 125 * MHZ: A frequência desejada de 125 MHz para o clk_sys.
  - » 125 * MHZ: A frequência de origem, que neste caso também é de 125 MHz.

Esta configuração garante que o clk_sys esteja rodando exatamente a 125 MHz, usando o pll_sys como fonte.

- **printf("CLK_SYS está operando a 125 MHz\n");**: Imprime uma mensagem indicando que o clock do sistema (clk_sys) está operando a 125 MHz. Essa mensagem pode ser vista na saída serial.
- **while (true) {...}:** Um loop infinito que mantém o programa rodando. Aqui você pode adicionar outras funcionalidades ou tarefas que o microcontrolador deve executar continuamente.

Exemplo 2

Resumo do exemplo

- O programa inicializa a comunicação serial.
- Define um intervalo de 1 segundo.
- Calcula o próximo momento (next_wake_time) em que deve executar uma ação.
- Entra em um loop infinito onde verifica continuamente se o tempo atual atingiu o tempo definido.
- Quando o tempo é atingido, imprime uma mensagem indicando que 1 segundo se passou.
- Atualiza o tempo para o próximo ciclo de 1 segundo.
- Adiciona uma pausa de 1 milissegundo em cada iteração do loop para reduzir o uso da CPU.

Link do programa comentado:
https://wokwi.com/projects/409690557858797569

A seguir, vamos analisar detalhadamente cada parte do código:

Inclusão de Bibliotecas

- #include "pico/stdlib.h": Inclui a biblioteca padrão para funções básicas de entrada e saída, controle de GPIO, temporização e comunicação serial.
- #include "pico/time.h": Inclui a biblioteca para gerenciamento de tempo e temporizações. Essa biblioteca fornece funções para trabalhar com tempos absolutos e relativos, bem como atrasos precisos.

Função Principal main()

A função principal inicializa o sistema e configura o temporizador para imprimir uma mensagem a cada segundo.

- **stdio_init_all();**: Inicializa todas as interfaces de comunicação padrão, como UART e USB, permitindo o uso de funções como printf para enviar dados através da saída serial. É essencial para depuração e comunicação com o mundo externo.
- **uint32_t interval=1000;**: Define o intervalo de tempo em milissegundos para a temporização. Neste caso, o intervalo é de 1000 milissegundos, o que corresponde a 1 segundo.

- **absolute_time_t next_wake_time:** Declara uma variável do tipo absolute_time_t, que representa um ponto específico no tempo, como um timestamp.
- **get_absolute_time():** Retorna o tempo absoluto atual do sistema.
- **delayed_by_us(get_absolute_time(), interval * 1000);**: Calcula um tempo absoluto futuro, adicionando o intervalo especificado (em microsegundos) ao tempo atual. Como o intervalo está em milissegundos, é multiplicado por 1000 para convertê-lo para microsegundos. Esse valor é armazenado em next_wake_time como o próximo momento em que a ação deve ocorrer.
- **while (true) {...}**: Um loop infinito que mantém o programa rodando continuamente. Todo o código dentro desse loop será repetido indefinidamente.
- **time_reached(next_wake_time)**: Verifica se o tempo absoluto atual já passou ou alcançou o valor definido em next_wake_time. Se sim, retorna true, indicando que o intervalo de tempo foi atingido.
- **printf("1 segundo passou\n");**: Imprime a mensagem "1 segundo passou" na saída serial, indicando que o intervalo de 1 segundo foi atingido.
- **next_wake_time = delayed_by_us(next_wake_time, interval * 1000);**: Atualiza next_wake_time adicionando o intervalo de 1 segundo ao tempo anterior next_wake_time. Isso define o próximo momento em que a mensagem será impressa, garantindo que o ciclo se repita a cada segundo.
- **sleep_ms(1);**: Faz o programa "dormir" por 1 milissegundo. Essa pausa é utilizada para reduzir o uso da CPU, evitando que o microcontrolador execute o loop incessantemente sem pausa. Isso ajuda a economizar energia e a reduzir o desgaste da CPU.

Exemplo 3

Resumo do exemplo

- O código inicializa a comunicação serial e configura um temporizador repetitivo que chama a função repeating_timer_callback a cada 1 segundo.
- Na função repeating_timer_callback, a mensagem "1 segundo passou" é impressa na saída serial.
- O loop principal (while (true)) continua executando, mas faz uma pausa de 1 segundo em cada iteração para reduzir o uso da CPU, permitindo a execução de outras tarefas no futuro, se necessário.
- O temporizador continua funcionando independentemente do loop principal, acionando o callback a cada 1 segundo.

Link do programa comentado:
https://wokwi.com/projects/409690905220586497

A seguir, vamos analisar detalhadamente cada parte do código:

Inclusão de Bibliotecas

- **#include "pico/stdlib.h"**: Inclui a biblioteca padrão do Raspberry Pi Pico para funções básicas como GPIO, temporização, controle de entradas e saídas, e comunicação serial.
- **#include "hardware/timer.h"**: Inclui a biblioteca para controle e uso de temporizadores de hardware. Fornece funções para configurar e manipular temporizadores, inclusive timers repetitivos.
- **#include <stdio.h>**: Inclui a biblioteca padrão de entrada e saída do C, permitindo o uso de funções como printf para exibir mensagens.

Função de Callback do Temporizador

- **bool repeating_timer_callback(struct repeating_timer *t)**: Esta função é um "callback", ou seja, uma função que será chamada automaticamente sempre que o temporizador configurado disparar.
  - » **Parâmetro struct repeating_timer *t**: Um ponteiro para a estrutura de dados do temporizador que chamou a função. Ele pode ser usado para acessar informações sobre o temporizador.
  - » **printf("1 segundo passou\n");**: Imprime a mensagem "1 segundo passou" na saída serial a cada vez que a função é chamada, indicando que 1 segundo se passou.

» **return true;**: Retorna true para manter o temporizador repetindo. Se retornasse false, o temporizador pararia de chamar esta função.

Função Principal main()

- **stdio_init_all();**: Inicializa todas as interfaces de entrada e saída padrão (por exemplo, USB ou UART), permitindo o uso de funções como printf para comunicação serial.
- **struct repeating_timer timer;**: Declara uma variável timer do tipo struct repeating_timer, que será usada para armazenar informações sobre o temporizador configurado.
- **add_repeating_timer_ms(1000, repeating_timer_callback, NULL, &timer);**: Configura um temporizador que chamará a função repeating_timer_callback a cada 1000 milissegundos (1 segundo).
  - » **1000**: O intervalo em milissegundos entre cada chamada do callback.
  - » **repeating_timer_callback**: O ponteiro para a função que será chamada a cada intervalo.
  - » **NULL**: Um valor de dado de usuário opcional, que pode ser passado para o callback. Neste caso, NULL indica que não há dados adicionais.
  - » **&timer**: Um ponteiro para a estrutura do temporizador que será preenchida com informações sobre o temporizador configurado. Essa estrutura pode ser usada para gerenciar o temporizador mais tarde.
- **while (true) {...}**: Loop infinito que mantém o programa em execução contínua. A função principal do programa termina apenas se a execução for interrompida manualmente.
- **sleep_ms(1000);**: Faz o programa "dormir" por 1000 milissegundos (1 segundo). Essa pausa é usada para reduzir o uso da CPU enquanto espera. No entanto, isso não afeta o temporizador configurado, pois ele opera em um sistema de interrupção independente do loop principal.

Exemplo 4

Resumo do exemplo

- **Inicialização:** O pino 12 é configurado como saída para controlar um LED. A comunicação serial é inicializada para possíveis mensagens de depuração.
- **Definição de Tempo:** Um intervalo de 1 segundo é definido para alternar o estado do LED.
- **Loop Infinito:** O programa verifica continuamente se o tempo de 1 segundo se passou. Quando o tempo é alcançado, o estado do LED é alternado (ligado/desligado).
- **Pausa no Loop:** Uma pequena pausa de 1 milissegundo é introduzida para evitar o uso excessivo da CPU.

Link do programa comentado:
https://wokwi.com/projects/409692282956192769

A seguir, vamos analisar detalhadamente cada parte do código:

Inclusão de Bibliotecas

- **pico/stdlib.h:** Esta biblioteca fornece funções essenciais para o controle de GPIO (entrada e saída digital), temporização, controle de pinos e comunicação serial no microcontrolador RP2040. Ela facilita a configuração e manipulação de hardware básico.
- **pico/time.h:** Fornece funções para trabalhar com tempos absolutos e relativos, atrasos e temporizadores. É utilizada aqui para medir intervalos de tempo e calcular eventos futuros.

Função main()

A função main() é o ponto de entrada do programa. Vamos analisar cada bloco dentro dela:

Inicialização da Comunicação Serial

- **Função stdio_init_all():** Inicializa todas as interfaces de comunicação padrão do microcontrolador, como USB e UART. Isso permite o uso de funções como printf para enviar dados para um terminal serial. É essencial para depuração, pois podemos imprimir informações no terminal.

Configuração do Pino de Saída

- **const uint LED_PIN = 12;:** Define o número do pino (12) como uma constante chamada LED_PIN. Isso facilita a leitura e a modificação do código, pois você pode mudar o número do pino em um único lugar se necessário.
- **gpio_init(LED_PIN);:** Inicializa o pino especificado (LED_PIN) como um pino GPIO. Isso prepara o pino para ser configurado como entrada ou saída.
- **gpio_set_dir(LED_PIN, GPIO_OUT);:** Configura o pino 12 como uma saída digital, permitindo que o código controle se o pino está em nível alto (3,3 V) ou baixo (0 V). Nesse caso, será usado para controlar um LED.

Definição do Intervalo de Tempo

- **uint32_t interval = 1000;:** Define uma variável interval do tipo uint32_t (um inteiro não assinado de 32 bits) com o valor 1000. Esse valor representa um intervalo de tempo de 1000 milissegundos, ou seja, 1 segundo. Este será o período de tempo entre cada alternância do LED (ligado/desligado).

Cálculo do Próximo Tempo para Alternar o LED

- **absolute_time_t next_wake_time:** Declara uma variável next_wake_time do tipo absolute_time_t, que representa um ponto específico no tempo.
- **get_absolute_time():** Retorna o tempo absoluto atual do sistema como um valor do tipo absolute_time_t.
- **delayed_by_us(...):** Calcula um tempo absoluto futuro, adicionando o valor especificado ao tempo atual. Como interval está em milissegundos e delayed_by_us() espera um valor em microsegundos, multiplicamos interval por 1000 (1 milissegundo = 1000 microsegundos).
- **next_wake_time:** Representa o próximo momento em que o LED deve alternar seu estado.

Estado Inicial do LED

- **bool led_on = false;:** Declara uma variável booleana led_on e a inicializa como false. Isso indica que o LED começa desligado. A variável será usada para armazenar e alternar o estado atual do LED (ligado/desligado).

Loop Infinito para Alternância do LED

- **while (true):** Inicia um loop infinito, que mantém o programa em execução contínua. O microcontrolador não para a execução, a menos que seja reiniciado ou desligado.
- **if (time_reached(next_wake_time)):** Verifica se o tempo atual alcançou ou ultrapassou next_wake_time. Se true, significa que 1 segundo se passou desde a última alternância do LED.
- **led_on = !led_on;:** Alterna o valor de led_on entre true (ligado) e false (desligado). Isso inverte o estado do LED.
- **gpio_put(LED_PIN, led_on);:** Define o estado do pino 12 (LED_PIN) com base no valor de led_on. Se led_on for true, o pino recebe nível alto (LED ligado); se for false, recebe nível baixo (LED desligado).
- **next_wake_time = delayed_by_us(next_wake_time, interval * 1000);:** Calcula o próximo momento em que o LED deve alternar de estado, adicionando 1 segundo ao next_wake_time atual.
- **sleep_ms(1);:** Introduz uma pequena pausa de 1 milissegundo no loop. Isso ajuda a reduzir o uso da CPU, evitando que o loop seja executado muito rapidamente e consuma recursos desnecessários. É uma prática comum em loops infinitos para evitar o uso excessivo da CPU.
- **return 0;:** É uma boa prática colocar um retorno no final da função main(), indicando que o programa terminou com sucesso. No entanto, devido ao loop infinito, esse código nunca será alcançado.

Exemplo 5

Resumo do exemplo

- **Inicialização:** O pino 12 é configurado como saída e um temporizador repetitivo é configurado para chamar a função repeating_timer_callback a cada 1 segundo.
- **Alternância do LED:** A cada segundo, a função repeating_timer_callback é chamada, alternando o estado do LED entre ligado e desligado e imprimindo uma mensagem de depuração na saída serial.
- **Loop Principal:** O loop principal (while (true)) não realiza nenhuma tarefa específica, mas pode ser utilizado para adicionar outras funcionalidades ao microcontrolador.

Link do programa comentado:
https://wokwi.com/projects/409325289916859393

A seguir, vamos analisar detalhadamente cada parte do código:

Inclusão de Bibliotecas

- **pico/stdlib.h:** Inclui funções essenciais para controle de GPIO (entrada/saída digital), temporização e comunicação serial. É necessária para configurar o LED e controlar seu estado.
- **hardware/timer.h:** Fornece funções e tipos de dados para trabalhar com temporizadores de hardware, incluindo a criação de temporizadores repetitivos que disparam funções em intervalos definidos.
- **stdio.h:** Fornece funções de entrada e saída padrão, como printf, para imprimir mensagens na interface serial. Isso é útil para depuração.

Definição do Pino do LED

- **const uint LED_PIN = 12;:** Define uma constante LED_PIN do tipo uint (inteiro sem sinal) com o valor 12. Essa constante representa o número do pino GPIO (pino 12) ao qual o LED está conectado. O uso de uma constante facilita a alteração do pino caso seja necessário, além de melhorar a legibilidade do código.

Função de Callback do Temporizador

- **Função repeating_timer_callback(struct repeating_timer *t):** Esta função é chamada automaticamente pelo temporizador a cada intervalo definido (1 segundo, neste caso). A função altera o estado do

LED e imprime uma mensagem na saída serial.

- » **static bool led_on = false;:** Declara uma variável booleana led_on como estática. A palavra-chave static faz com que a variável preserve seu valor entre chamadas sucessivas da função. Inicialmente, led_on é definida como false (LED desligado).
- » **led_on = !led_on;:** Alterna o valor de led_on entre true e false. Se led_on for false, a expressão !led_on será true, e vice-versa. Isso inverte o estado atual do LED.
- » **gpio_put(LED_PIN, led_on);:** Define o estado do pino LED_PIN (pino 12) com base no valor de led_on. Se led_on for true, o pino será colocado em nível alto (3,3V), ligando o LED. Se led_on for false, o pino será colocado em nível baixo (0V), desligando o LED.
- » **printf("LED %s\n", led_on ? "ligado" : "desligado");:** Imprime uma mensagem indicando o estado atual do LED. A expressão led_on ? "ligado" : "desligado" é um operador ternário que escolhe a string "ligado" se led_on for true e "desligado" se led_on for false. Isso fornece feedback visual na saída serial, útil para depuração.
- » **return true;:** Retorna true para continuar repetindo a chamada da função repeating_timer_callback. Se false fosse retornado, o temporizador pararia de chamar esta função.

Função Principal main()

- **stdio_init_all();:** Inicializa a comunicação serial, permitindo o uso de printf para enviar mensagens pela interface serial (USB ou UART). Esta função é essencial para depuração, pois permite monitorar o comportamento do programa.
- **gpio_init(LED_PIN);:** Inicializa o pino 12 como um pino GPIO, preparando-o para ser configurado como entrada ou saída.
- **gpio_set_dir(LED_PIN, GPIO_OUT);:** Define o pino 12 como saída. Isso permite que o programa controle o estado do pino (ligado/desligado) para acender ou apagar o LED.
- **struct repeating_timer timer;:** Declara uma variável timer do tipo struct repeating_timer. Esta estrutura armazenará informações sobre o temporizador repetitivo que será configurado.
- **add_repeating_timer_ms(1000, repeating_timer_callback, NULL, &timer);:** Configura um temporizador que chama a função repeating_timer_callback a cada 1000 milissegundos (1 segundo).
  - » **1000:** O intervalo de tempo em milissegundos entre cada chamada da função de callback.

  - » **repeating_timer_callback:** O ponteiro para a função que será chamada a cada intervalo de tempo.
  - » **NULL:** Este parâmetro pode ser usado para passar dados adicionais para a função de callback, mas aqui não é necessário, então NULL é usado.
  - » **&timer:** Um ponteiro para a estrutura timer, que será preenchida com informações sobre o temporizador configurado. Essa estrutura pode ser usada para gerenciar ou parar o temporizador posteriormente.
- **while (true) {...}:** Um loop infinito que mantém o programa em execução contínua. Neste caso, o loop principal não faz nada (apenas mantém o programa rodando), pois o controle do LED é gerenciado pela função de callback do temporizador. Esse loop pode ser utilizado para adicionar outras tarefas que o microcontrolador deve executar.

Exemplo 6

Resumo do exemplo

- **Inicialização:** O pino do LED é configurado como saída e o pino do botão como entrada com resistor pull-up.
- **Loop Principal:** O loop verifica continuamente se o botão foi pressionado. Se o botão for pressionado e o LED estiver desligado, o LED é aceso.
- **Temporização:** Após 3 segundos, a função de callback é chamada, desligando o LED.
- **Uso de Alarmes:** A configuração do alarme permite executar ações temporizadas sem bloquear o loop principal.

Link do programa comentado:
https://wokwi.com/projects/409696442353327105

A seguir, vamos analisar detalhadamente cada parte do código:

Inclusão de Bibliotecas

- **pico/stdlib.h:** Esta biblioteca contém funções essenciais para operações básicas do microcontrolador, como controle de GPIO (entrada e saída digital), temporização, controle de pinos e comunicação serial.
- **pico/time.h:** Fornece funções e tipos de dados para trabalhar com temporizadores, temporizações e alarmes, permitindo programar eventos futuros que ocorram após um determinado intervalo de tempo.

Definição de Constantes

- **const uint LED_PIN = 11;:** Define uma constante LED_PIN para representar o pino 11 do GPIO. Isso facilita a leitura e a manutenção do código, pois qualquer referência ao pino 11 será feita usando LED_PIN.
- **const uint BUTTON_PIN = 5;:** Define uma constante BUTTON_PIN para representar o pino 5 do GPIO, que será utilizado para ler o estado do botão.

Definição de Variáveis Globais

- **bool led_on = false;:** Declara uma variável booleana para armazenar o estado atual do LED (ligado ou desligado). No entanto, esta variável não é utilizada diretamente no código atual.

- **bool led_active = false;:** Variável booleana que indica se o LED está atualmente aceso. Serve para evitar que o LED seja reativado enquanto já está aceso.
- **absolute_time_t turn_off_time;:** Declara uma variável do tipo absolute_time_t para armazenar o tempo em que o LED deve ser desligado. Embora seja declarada, ela não é utilizada no código.

Função de Callback para Desligamento do LED

- **int64_t turn_off_callback(alarm_id_t id, void *user_data):** Esta função é chamada automaticamente quando o alarme configurado expira (após 3 segundos). Vamos detalhar cada linha dentro da função:
  - » **gpio_put(LED_PIN, false);:** Define o pino LED_PIN (pino 11) como nível baixo (0V), desligando o LED.
  - » **led_active = false;:** Atualiza a variável led_active para false, indicando que o LED está desligado e permitindo que ele possa ser aceso novamente se o botão for pressionado.
  - » **return 0;:** Retorna 0 para indicar que o alarme não deve ser repetido. Se retornasse um valor diferente de 0, o alarme poderia ser configurado para repetir a chamada da função.

Função Principal main()

- **stdio_init_all();:** Inicializa a comunicação serial para permitir o uso de funções como printf para enviar mensagens para o terminal serial, útil para depuração. Embora essa função seja chamada, printf não é utilizado no código atual.

Configuração do LED

- **gpio_init(LED_PIN);:** Inicializa o pino GPIO 11 para que ele possa ser usado como entrada ou saída.
- **gpio_set_dir(LED_PIN, GPIO_OUT);:** Configura o pino 11 como saída. Isso significa que o programa pode controlar o estado desse pino (nível alto ou baixo) para ligar ou desligar o LED.

Configuração do Botão

- **gpio_init(BUTTON_PIN);:** Inicializa o pino GPIO 5 para uso.
- **gpio_set_dir(BUTTON_PIN, GPIO_IN);:** Configura o pino 5 como entrada, permitindo que o programa leia o estado do pino (nível alto ou baixo).

- **gpio_pull_up(BUTTON_PIN);:** Habilita o resistor de pull-up interno. Isso faz com que o pino seja lido como alto (3,3V) quando o botão não está pressionado, evitando flutuações indesejadas no estado do pino.

Loop Principal do Programa

- **while (true) {...}:** Um loop infinito que mantém o programa rodando continuamente. O microcontrolador verifica constantemente o estado do botão e realiza ações apropriadas.
- **Verificação do Botão Pressionado:**
  - » **if (gpio_get(BUTTON_PIN) == 0 && !led_active):** Verifica se o botão está pressionado (gpio_get(BUTTON_PIN) == 0, nível baixo) e se o LED não está ativo (!led_active). Isso evita que o LED seja reativado durante os 3 segundos em que já está aceso.
  - » **Debounce (sleep_ms(50);):** Adiciona um atraso de 50 ms para evitar leituras incorretas devido a oscilações rápidas quando o botão é pressionado.
  - » **Verificação Repetida:**
    - **if (gpio_get(BUTTON_PIN) == 0):** Verifica novamente o estado do botão após o debounce para garantir que ele está realmente pressionado.
- **Acionamento do LED:**
  - » **gpio_put(LED_PIN, true);:** Liga o LED configurando o pino 11 como nível alto.
  - » **led_active = true;:** Atualiza led_active para true, indicando que o LED está aceso e impedindo que o LED seja aceso novamente durante o intervalo de 3 segundos.
- **Configuração do Alarme para Desligar o LED:**
  - » **add_alarm_in_ms(3000, turn_off_callback, NULL, false);:** Agenda um alarme para chamar a função turn_off_callback após 3000 milissegundos (3 segundos). O LED será desligado quando esse alarme disparar.
- **Pausa no Loop:**
  - » **sleep_ms(10);:** Adiciona uma pequena pausa de 10 ms para reduzir o uso da CPU, evitando que o loop seja executado desnecessariamente rápido e consuma recursos do microcontrolador.

Exemplo 7

Resumo do exemplo

- **Inicialização:** O LED é configurado como saída e o botão como entrada com resistor pull-up interno.
- **Verificação do Botão:** O programa verifica continuamente se o botão foi pressionado.
- **Acendimento do LED:** Se o botão for pressionado e o LED estiver desligado, o LED é aceso por 2 segundos.
- **Desligamento do LED:** Um temporizador repetitivo verifica se o tempo de 2 segundos passou e, se sim, desliga o LED.
- **Controle de Reativação:** O LED só pode ser aceso novamente após ser desligado, evitando múltiplas ativações durante o intervalo.

Link do programa comentado:
https://wokwi.com/projects/409325572648671233

A seguir, vamos analisar detalhadamente cada parte do código:

Inclusão de Bibliotecas

- **#include"pico/stdlib.h":** Inclui a biblioteca padrão do Raspberry Pi Pico que fornece funções essenciais para manipulação de GPIO (entradas e saídas digitais), temporização, controle de pinos e comunicação serial.
- **#include "hardware/timer.h":** Inclui a biblioteca para utilização de temporizadores de hardware. Essa biblioteca permite a criação e configuração de temporizadores, além do uso de alarmes para executar funções em intervalos específicos.

Definição de Constantes

- **const uint LED_PIN = 12;:** Define uma constante LED_PIN que representa o pino GPIO 12. Esse pino será utilizado para controlar o LED, facilitando a leitura e manutenção do código, pois o número do pino é substituído por um nome significativo.
- **const uint BUTTON_PIN = 5;:** Define uma constante BUTTON_PIN para representar o pino GPIO 5, que será utilizado para ler o estado do botão. Isso também facilita a leitura e a modificação do código.

Variáveis de Controle

- **bool led_on = false;:** Variável booleana que indica se o LED está ligado

(true) ou desligado (false). Apesar de declarada, ela não é utilizada diretamente no controle do LED neste código específico.

- **absolute_time_t turn_off_time;:** Armazena o momento exato (tempo absoluto) em que o LED deve ser desligado. Este tempo é calculado quando o botão é pressionado, somando um intervalo de 2 segundos ao tempo atual.
- **bool led_active = false;:** Variável que indica se o LED está ativo (true). Serve para impedir que o LED seja reativado enquanto ainda está aceso, evitando múltiplas ativações desnecessárias durante o período de 2 segundos.

Função de Callback do Temporizador Repetitivo

- **bool repeating_timer_callback(struct repeating_timer *t):** Função de callback que é chamada repetidamente pelo temporizador a cada segundo. A função verifica se o LED deve ser desligado com base no tempo atual e no tempo de desligamento (turn_off_time).
  - » **if (led_active && absolute_time_diff_us(get_absolute_time(), turn_off_time) <= 0):**
    - **led_active:** Verifica se o LED está atualmente ativo (ligado).
    - **absolute_time_diff_us(get_absolute_time(), turn_off_time) <= 0:** Calcula a diferença entre o tempo atual (get_absolute_time()) e o tempo de desligamento (turn_off_time). Se a diferença for menor ou igual a zero, significa que o tempo para desligar o LED foi alcançado ou ultrapassado.
  - » **gpio_put(LED_PIN, false);:** Desliga o LED definindo o pino LED_PIN (12) como nível baixo (false).
  - » **led_active = false;:** Atualiza a variável led_active para indicar que o LED foi desligado, permitindo que o LED possa ser ligado novamente quando o botão for pressionado.
  - » **return true;:** Retorna true para garantir que o temporizador continue chamando esta função repetidamente.

Função main()

- **stdio_init_all();:** Inicializa a comunicação serial padrão, permitindo o uso de funções como printf para enviar mensagens para um terminal serial. Isso é útil para depuração, embora não seja utilizado diretamente neste código.

Configuração do LED e do Botão

- **gpio_init(LED_PIN);:** Inicializa o pino GPIO 12 para ser utilizado como uma entrada ou saída digital.
- **gpio_set_dir(LED_PIN, GPIO_OUT);:** Configura o pino 12 como saída, permitindo que o programa controle o estado do pino para ligar ou desligar o LED.
- **gpio_init(BUTTON_PIN);:** Inicializa o pino GPIO 5, que será utilizado para ler o estado do botão.
- **gpio_set_dir(BUTTON_PIN, GPIO_IN);:** Configura o pino 5 como entrada, permitindo que o programa leia o estado do botão (pressionado ou não).
- **gpio_pull_up(BUTTON_PIN);:** Ativa o resistor de pull-up interno no pino 5. Isso faz com que o pino seja lido como alto (3,3 V) quando o botão não está pressionado, evitando flutuações indesejadas de leitura.

Configuração do Temporizador

- **struct repeating_timer timer;:** Declara uma variável timer do tipo struct repeating_timer que armazenará informações sobre o temporizador configurado.
- **add_repeating_timer_ms(1000, repeating_timer_callback, NULL, &timer);:**
  - » Configura um temporizador repetitivo que chama a função repeating_timer_callback a cada 1000 milissegundos (1 segundo).
  - » **Parâmetros:**
    - **1000:** Intervalo de tempo em milissegundos entre cada chamada da função de callback.
    - **repeating_timer_callback:** Ponteiro para a função de callback que será chamada a cada intervalo.
    - **NULL:** Pode ser usado para passar dados adicionais para a função de callback, mas não é utilizado aqui.
    - **&timer:** Ponteiro para a estrutura que armazenará informações do temporizador.

Loop Principal do Programa

- **while (true) {...}:** Loop infinito que mantém o programa em execução contínua. Ele verifica constantemente o estado do botão e controla o LED conforme necessário.
- **Verificação do Estado do Botão:**
  - » **if (gpio_get(BUTTON_PIN) == 0 && !led_active):**

- **gpio_get(BUTTON_PIN) == 0:** Verifica se o botão está pressionado (nível baixo no pino).
- **!led_active:** Verifica se o LED não está ativo. Isso evita que o LED seja reativado enquanto ainda está aceso.

- **Debounce:**
  - » **sleep_ms(50);:** Aguarda 50 milissegundos para evitar leituras incorretas causadas pela oscilação do botão ao ser pressionado (debounce).
  - » **Verificação Repetida:**
    - **if (gpio_get(BUTTON_PIN) == 0):** Verifica novamente o estado do botão após o debounce para confirmar que ele está realmente pressionado.
- **Acendimento do LED:**
  - » **gpio_put(LED_PIN, true);:** Liga o LED definindo o pino 12 como nível alto.
  - » **led_active = true;:** Atualiza a variável led_active para indicar que o LED está aceso e impedir que o botão o acenda novamente enquanto ainda está no intervalo de 2 segundos.
  - » **Definição do Tempo de Desligamento:**
    - **turn_off_time = make_timeout_time_ms(2000);:** Define o tempo em que o LED deve ser desligado, 2 segundos a partir do momento atual.
- **Pausa no Loop:**
  - » **sleep_ms(10);:** Introduz uma pequena pausa de 10 ms para evitar que o loop seja executado muito rapidamente, reduzindo o uso da CPU e evitando leituras excessivas do botão.
- **return 0;:** O return no final da função main() é uma boa prática, embora nunca seja alcançado devido ao loop infinito.

Exemplo 8

Resumo do exemplo

- **Inicialização:** O LED é configurado como saída e o botão como entrada com resistor de pull-up.
- **Verificação do Botão:** O temporizador repetitivo verifica continuamente o estado do botão a cada 100 ms.
- **Debounce e Contagem de Pressões:** Se o botão for pressionado duas vezes em sequência com um intervalo adequado, o LED acende.
- **Temporização para Desligamento:** Um temporizador é configurado para desligar o LED após 4 segundos.
- **Controle Eficiente:** A função tight_loop_contents() mantém o loop principal otimizado, já que todas as tarefas são gerenciadas pelo temporizador.

Link do programa comentado:
https://wokwi.com/projects/409698053693899777

A seguir, vamos analisar detalhadamente cada parte do código:

Inclusão de Bibliotecas

- **pico/stdlib.h:** Inclui funções essenciais para manipulação de GPIO (entradas e saídas digitais), temporização e comunicação serial.
- **hardware/timer.h:** Fornece funcionalidades para utilizar temporizadores de hardware, permitindo a criação de temporizadores repetitivos e alarmes.
- **stdio.h:** Biblioteca padrão para funções de entrada e saída, como printf para exibir mensagens no terminal serial, útil para depuração.

Definição de Pinos

- Define os pinos GPIO que serão utilizados no código. LED_PIN representa o pino 13, que controla o LED, e BUTTON_PIN representa o pino 5, que lê o estado do botão.

Variáveis Globais Voláteis

- **volatile:** A palavra-chave volatile é usada para informar ao compilador que essas variáveis podem ser alteradas fora do fluxo de controle normal do programa, como em interrupções. Isso impede que o compilador otimize essas variáveis, garantindo que suas leituras e

escritas ocorram sempre que acessadas.
- **button_press_count:** Contador de quantas vezes o botão foi pressionado.
- **led_on:** Indica se o LED está atualmente ligado (true) ou desligado (false).
- **led_active:** Indica se o LED está em uso, prevenindo múltiplas ativações.

Função de Callback para Desligamento do LED
- Esta função é chamada pelo temporizador quando o alarme expira, ou seja, após 4 segundos da ativação do LED.
- **Desligamento do LED:** gpio_put(LED_PIN, false); desliga o LED.
- **led_active = false;:** Atualiza a variável led_active indicando que o LED foi desligado.
- **printf("LED desligado\n");:** Exibe uma mensagem de depuração na saída serial para indicar que o LED foi desligado.
- **return 0;:** Retorna 0 para indicar que o alarme não deve ser repetido.

Função de Callback do Temporizador Repetitivo
- **repeating_timer_callback(struct repeating_timer *t):** Esta função é chamada pelo temporizador a cada 100 ms para verificar o estado do botão e controlar o LED.
- **Debounce:**
  - » Verifica se o botão foi pressionado e espera 200 milissegundos (200000 microssegundos) para evitar leituras errôneas devido a oscilações mecânicas do botão.
- **Contagem de Pressões:**
  - » Se o botão foi pressionado duas vezes (button_press_count == 2), o LED é ligado e um temporizador é configurado para desligar o LED após 4 segundos.
- **add_alarm_in_ms(4000, turn_off_callback, NULL, false);:** Configura um alarme para chamar a função turn_off_callback em 4 segundos, desligando o LED.

Função Principal main()
- **stdio_init_all();:** Inicializa a comunicação serial para permitir o uso de printf para depuração.
- **Configuração do LED:**
  - » **gpio_init(LED_PIN);:** Inicializa o pino 13.

  - » **gpio_set_dir(LED_PIN, GPIO_OUT);:** Configura o pino 13 como saída.
  - » **gpio_put(LED_PIN, 0);:** Garante que o LED começa apagado (nível baixo).
- **Configuração do Botão:**
  - » **gpio_init(BUTTON_PIN);:** Inicializa o pino 5.
  - » **gpio_set_dir(BUTTON_PIN, GPIO_IN);:** Configura o pino 5 como entrada.
  - » **gpio_pull_up(BUTTON_PIN);:** Ativa o resistor pull-up interno, garantindo um valor de leitura alto (3,3 V) quando o botão não está pressionado.
- **Configuração do Temporizador:**
  - » **add_repeating_timer_ms(100, repeating_timer_callback, NULL, &timer);:** Configura um temporizador repetitivo que chama a função repeating_timer_callback a cada 100 ms para verificar o estado do botão e controlar o LED.
- **Loop Principal:**
  - » **while (true) {...}:** Mantém o programa em execução contínua. O tight_loop_contents() otimiza o loop para evitar consumo excessivo de CPU enquanto não há outras tarefas a serem executadas.


embarca
tech
Residência
Tecnológica
em Sistemas
Embarcados