

🎓 Interrupções

Unidade 4 | Capítulo 4 - Aula síncrona (27/01/2024)

Prof. Wilton Lacerda Silva

Executores:



Coordenação:



Iniciativa:



Jornada até aqui...

Executores:



Coordenação:



Iniciativa:



Sumário

- Objetivos
- Revisão
- Polling
- Interrupções no RP2040
- Exemplos de Códigos
- Fenômeno do Bouncing
- Bonus
- Principais Pontos
- Conclusão

Pré-requisitos

- Familiaridade com a IDE do VS Code para desenvolvimento de software para microcontroladores, bem como manuseio do Kit de Estudo BitDogLab.
- Controlar os pinos de entrada e saída (GPIO).
- Acionamento de LEDs e leitura de botões.



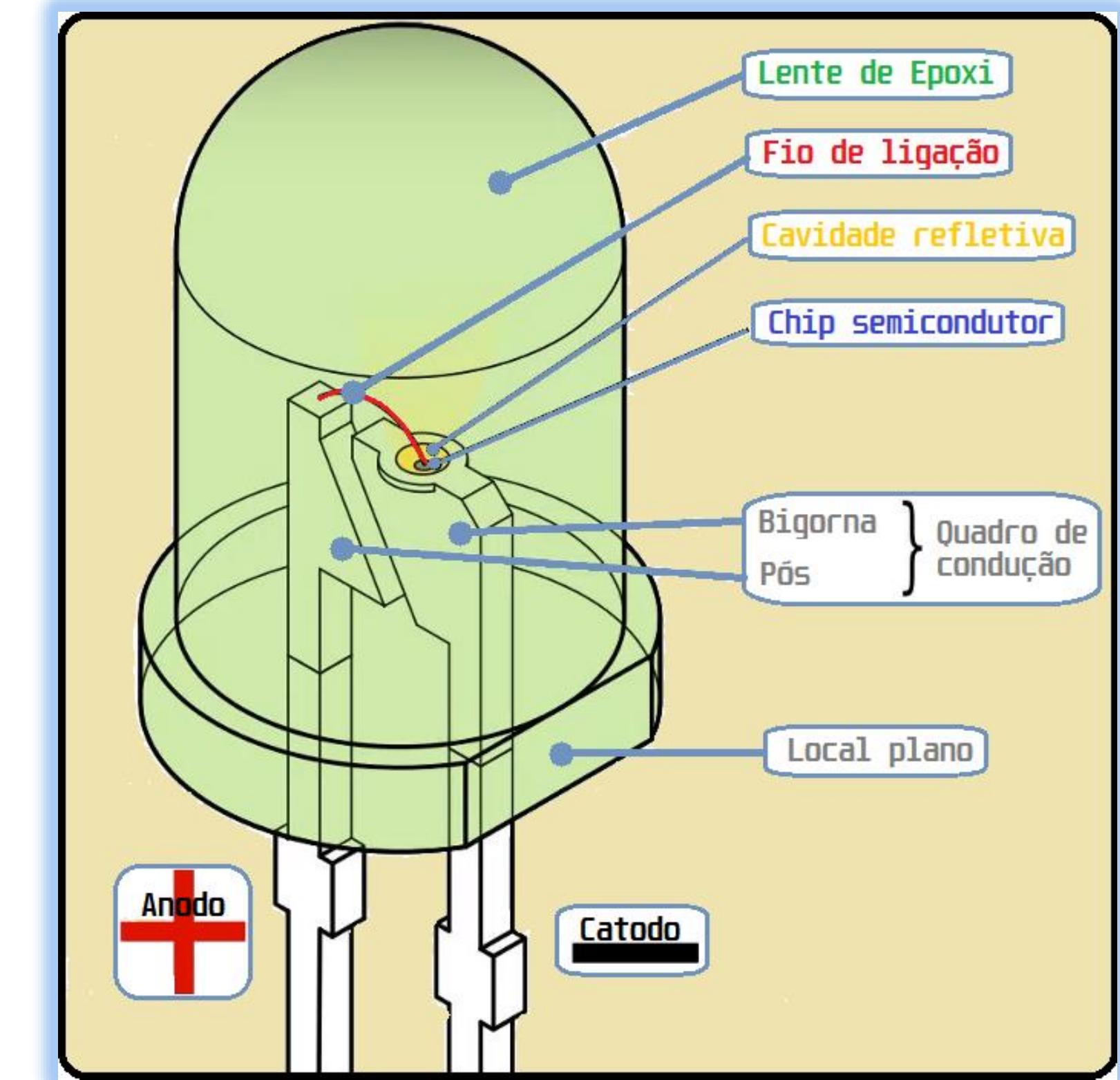
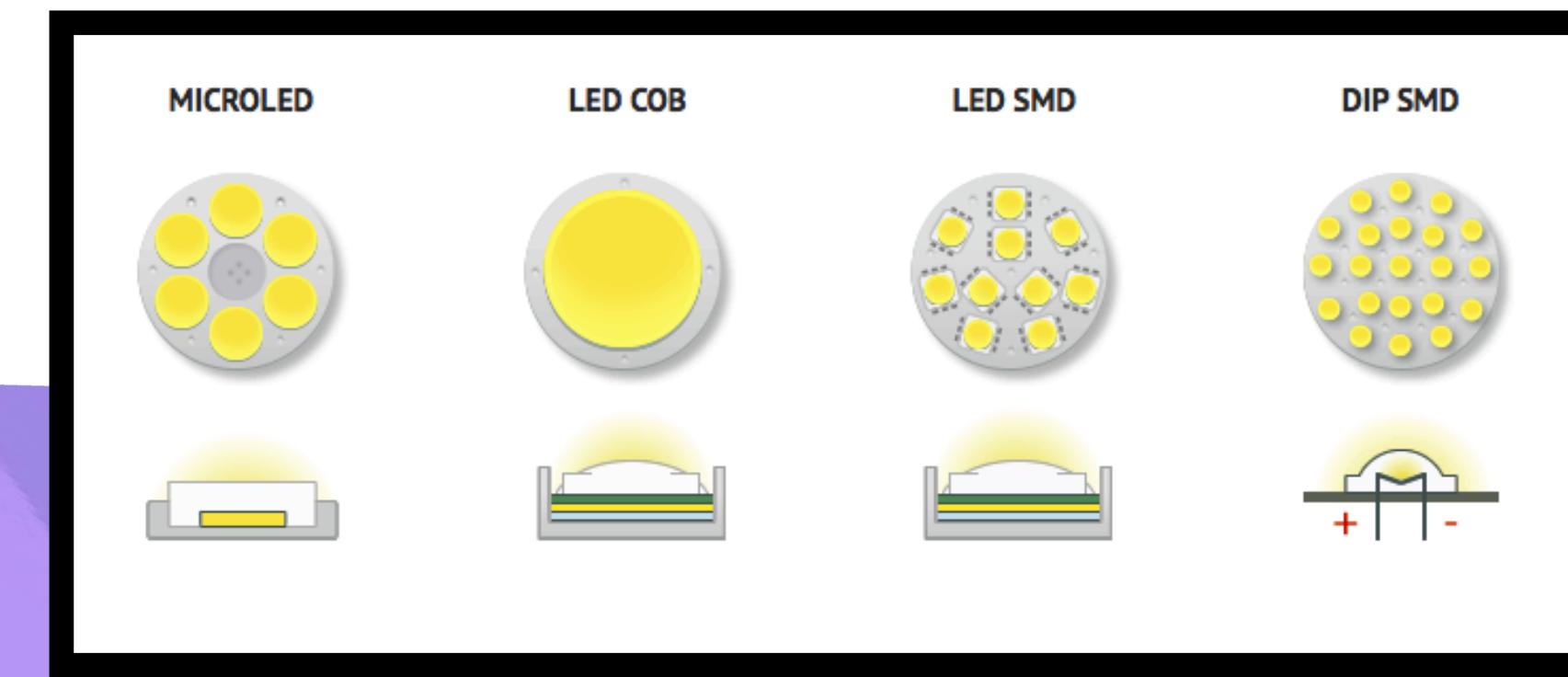
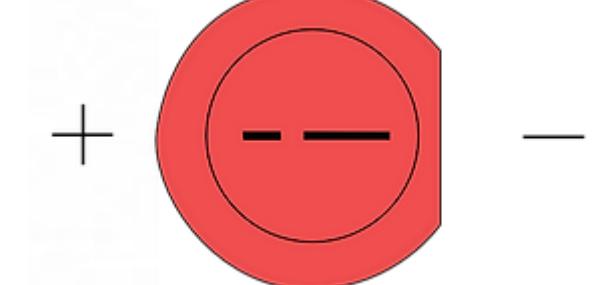
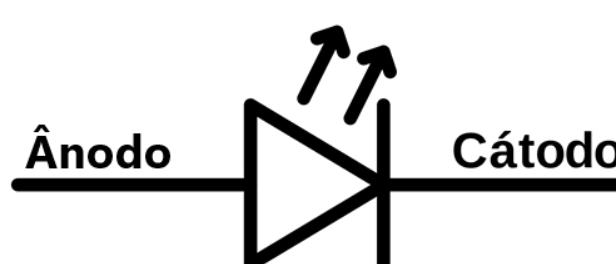
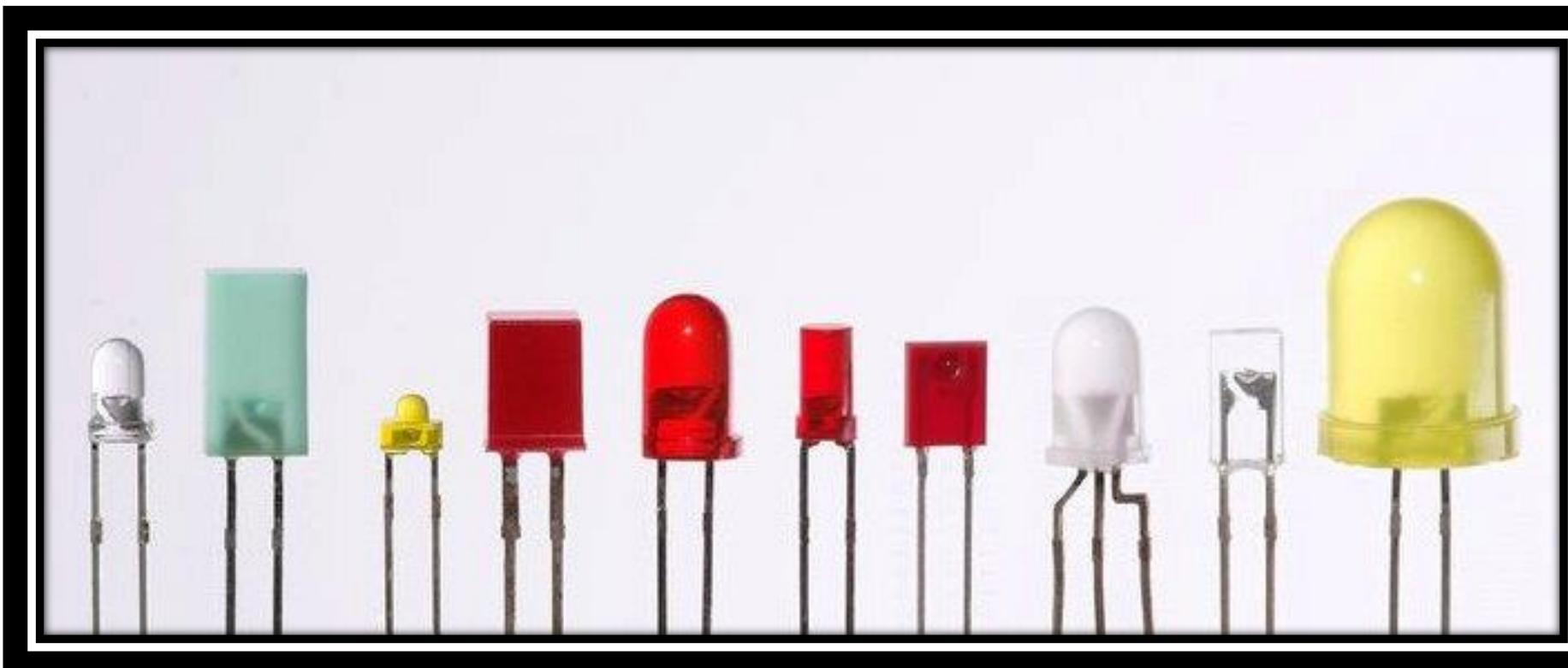
Objetivos

- Compreender o funcionamento do mecanismo de interrupções no RP2040.
- Configurar o módulo de interrupções e rotinas de tratamento de interrupções.
- Desenvolver código de utilização de interrupções e seus diferentes modos.



Revisão

LED é a sigla para Light Emitting Diode, ou seja, **Díodo Emissor de Luz**. É um componente eletrônico que converte energia elétrica diretamente em luz visível.

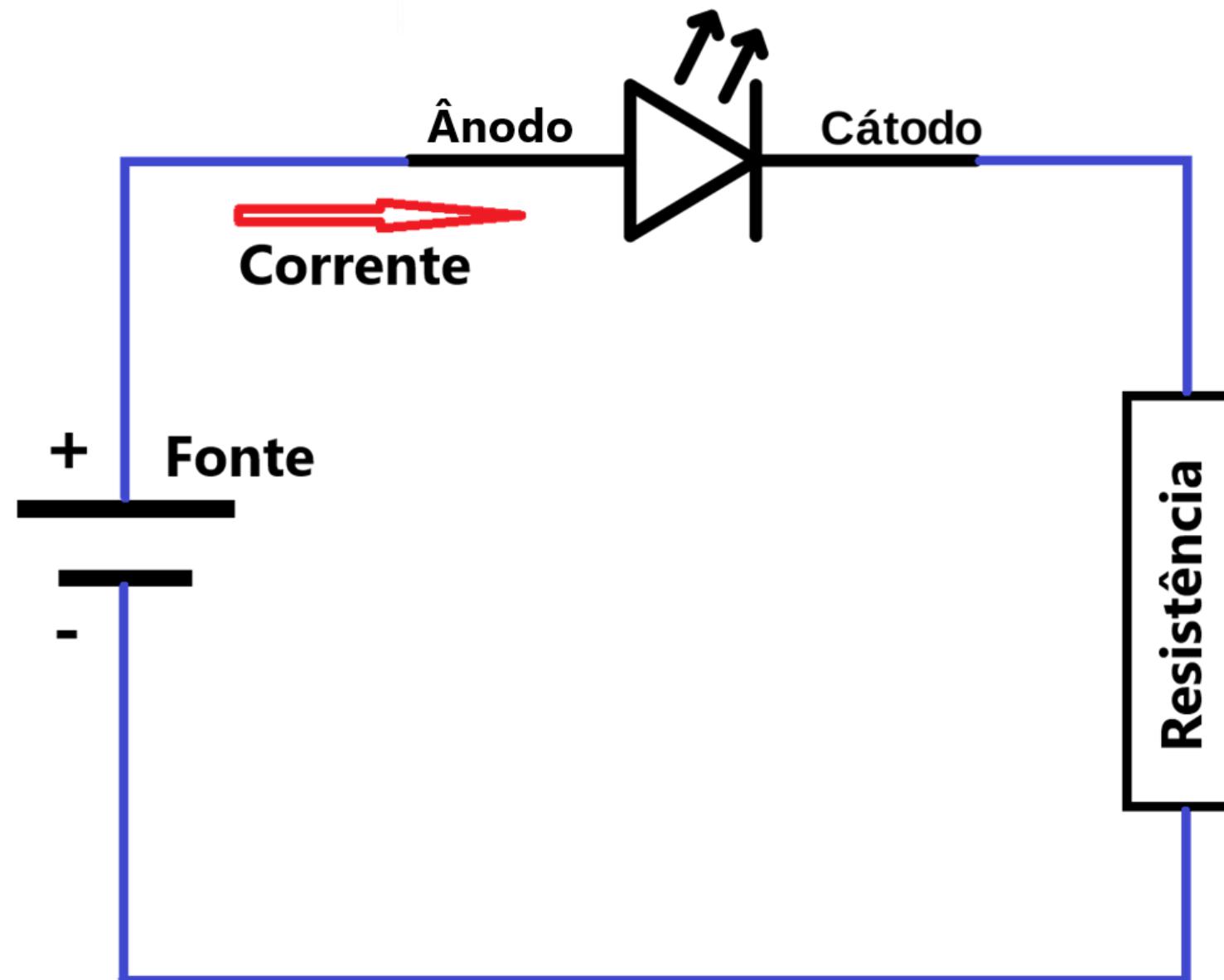


<https://www.hardwarecentral.net/single-post/2018/10/03/%C3%B3ptica-led>

Revisão

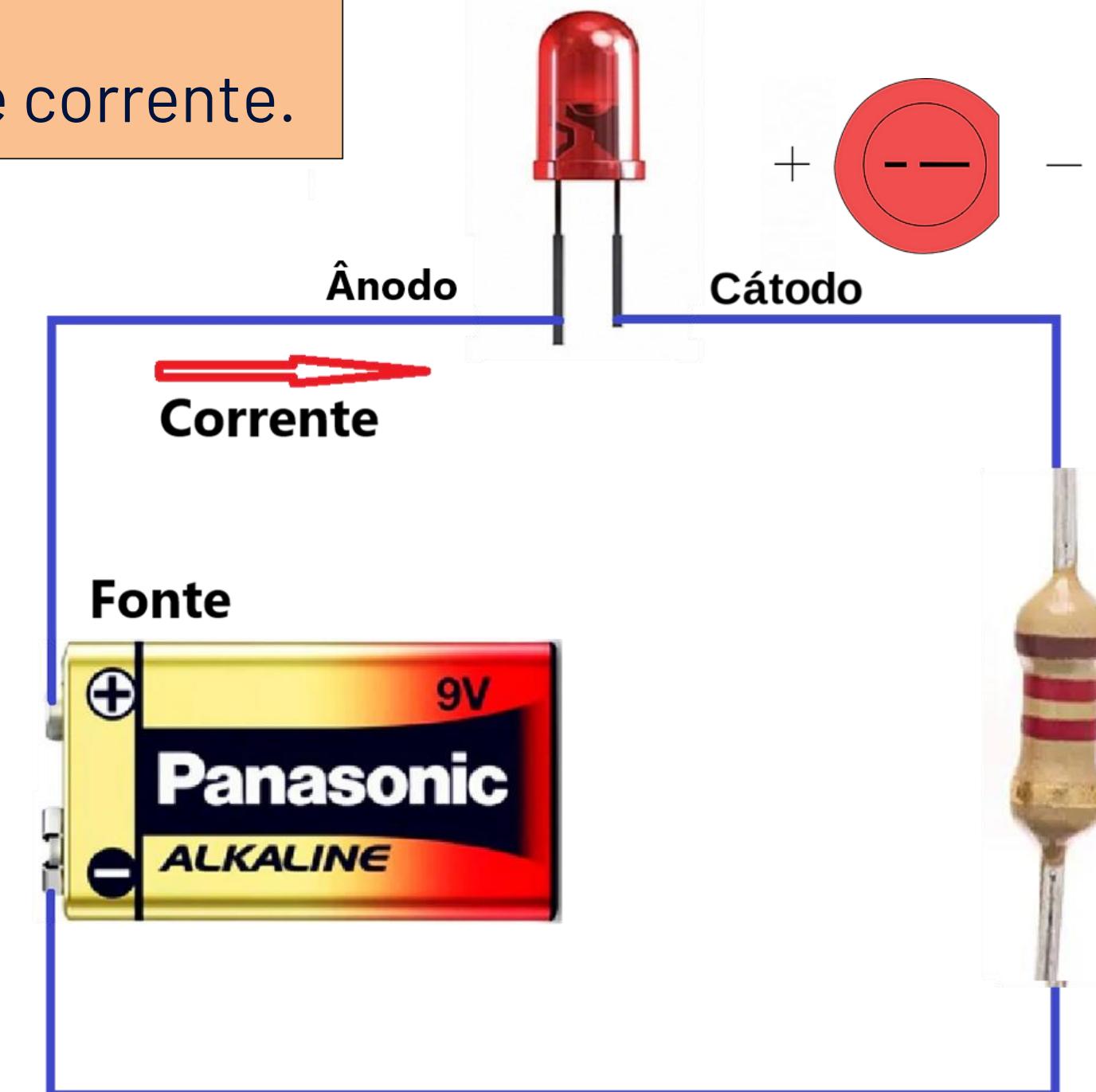
Ligação de um LED:

Atenção com a polarização e com o resistor limitador de corrente.



$$I = \frac{V_{ss} - V_d}{R}$$

$$I = \frac{9 - 1,8}{1200} = 6mA$$

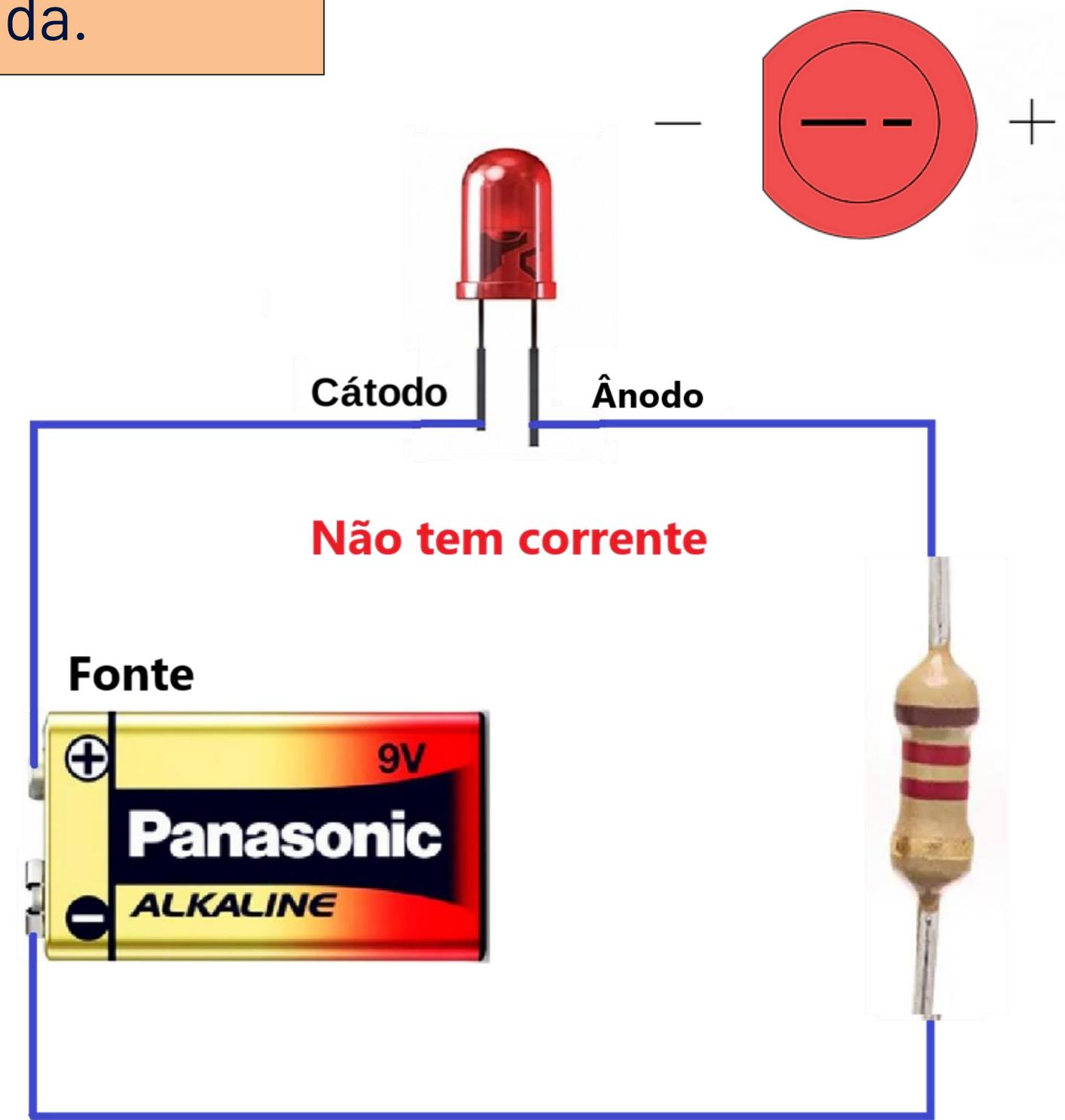
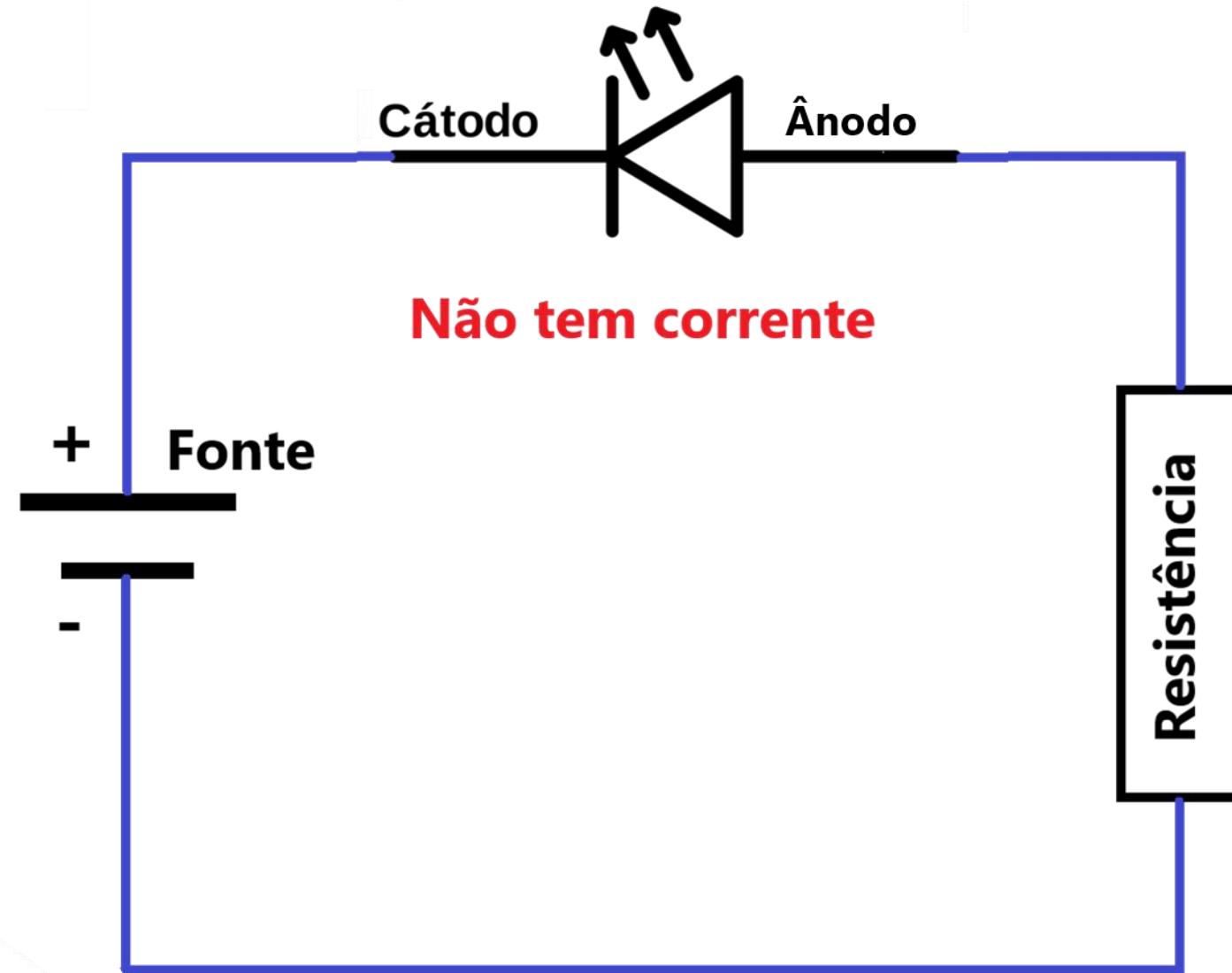


$$I = \frac{3,3 - 1,8}{1200} = 1,25mA$$

Revisão

Ligação de um LED:

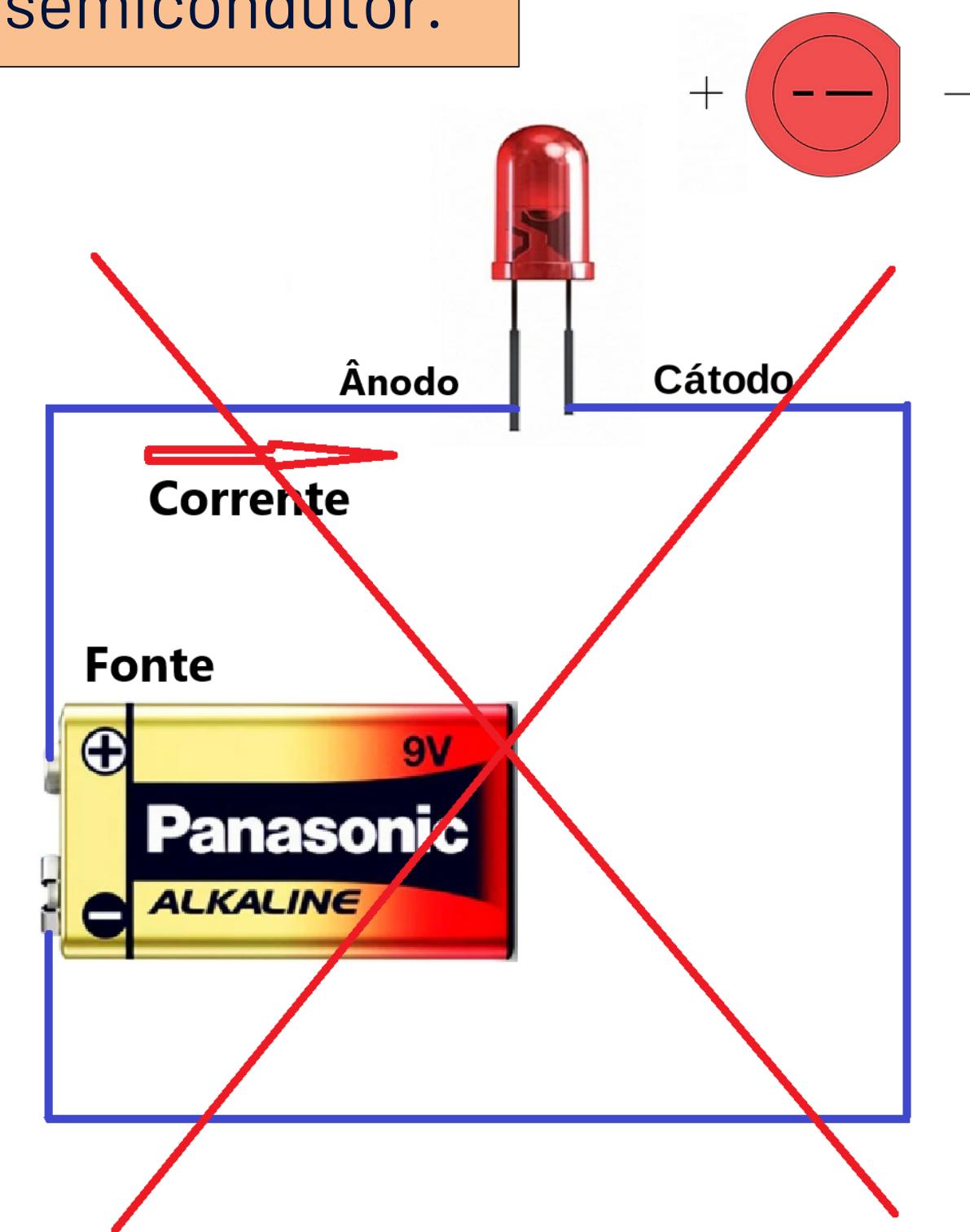
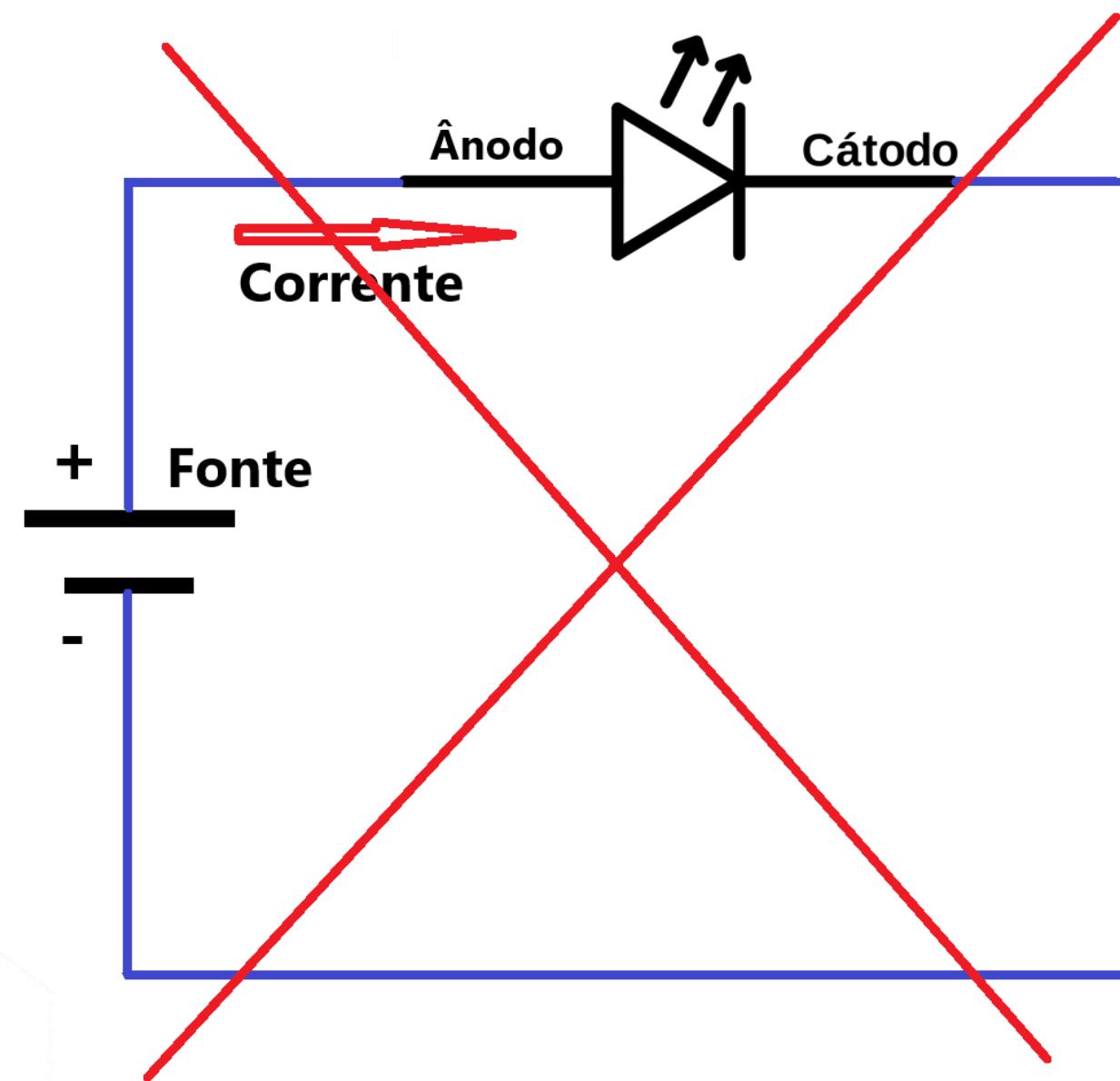
Deste modo não funciona. Entretanto, não danificará nada.



Revisão

Ligação de um LED:

Não ligue LEDs **sem resistores** limitadores de corrente. A corrente será extremamente alta e **danificará** o dispositivo semicondutor.

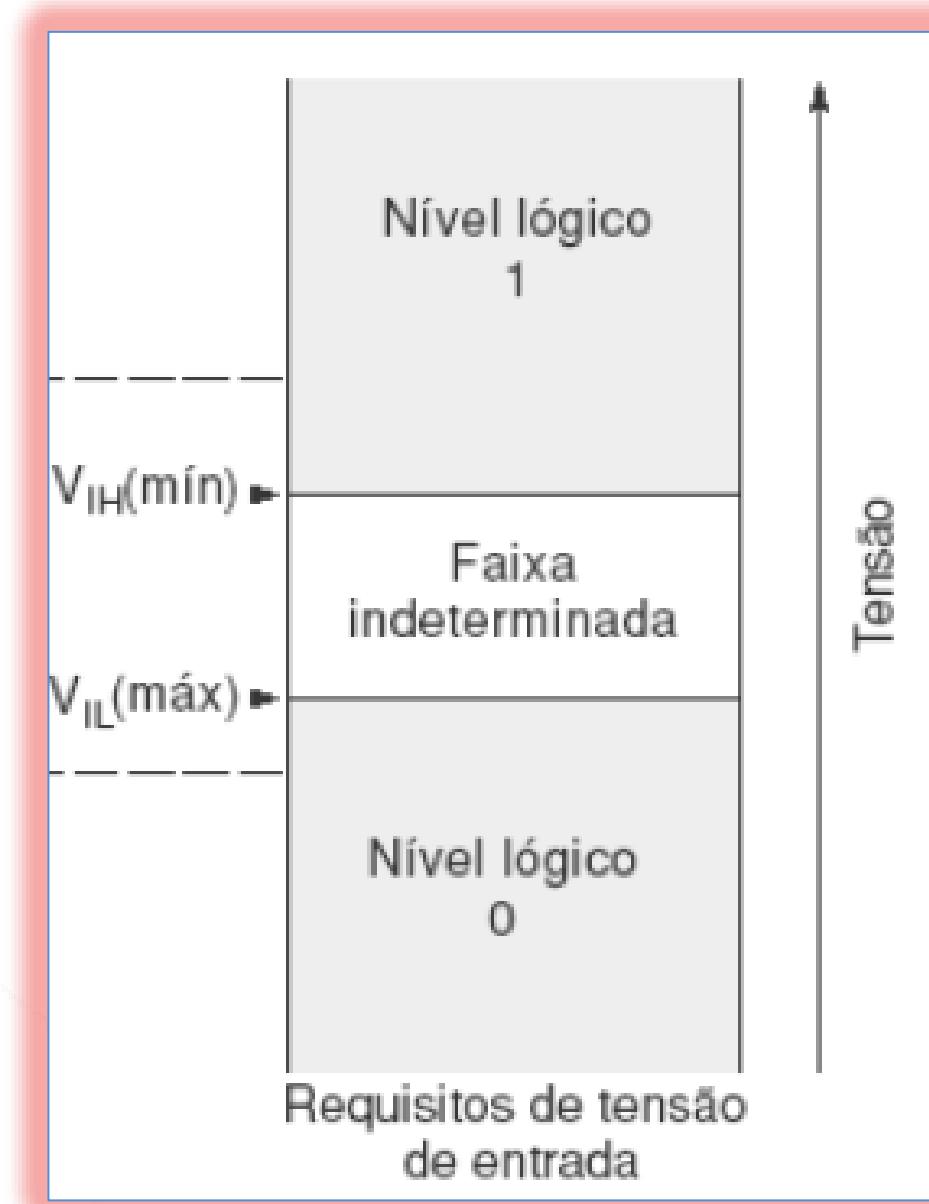


Revisão

Níveis de tensão relacionados às tomadas de decisão

O Raspberry Pi Pico (RP2), baseado no microcontrolador RP2040, opera com níveis lógicos de **3,3 V**. As faixas de tensão típicas para reconhecer os níveis lógicos são:

- **0 lógico (LOW)**: De aproximadamente **-0,3 V** a **0,8 V**.
- **1 lógico (HIGH)**: De aproximadamente **2,0 V** a **3,3+0,3 V**.



5.5.3.4. IO Electrical Characteristics

Parameter	Symbol	Minimum	Maximum	Units
Pin Input Leakage Current	I_{IN}		1	μA
Input Voltage High @ $\text{IOVDD}=1.8\text{V}$	V_{IH}	$0.65 * \text{IOVDD}$	$\text{IOVDD} + 0.3$	V
Input Voltage High @ $\text{IOVDD}=2.5\text{V}$	V_{IH}	1.7	$\text{IOVDD} + 0.3$	V
Input Voltage High @ $\text{IOVDD}=3.3\text{V}$	V_{IH}	2	$\text{IOVDD} + 0.3$	V
Input Voltage Low @ $\text{IOVDD}=1.8\text{V}$	V_{IL}	-0.3	$0.35 * \text{IOVDD}$	V
Input Voltage Low @ $\text{IOVDD}=2.5\text{V}$	V_{IL}	-0.3	0.7	V
Input Voltage Low @ $\text{IOVDD}=3.3\text{V}$	V_{IL}	-0.3	0.8	V

Revisão

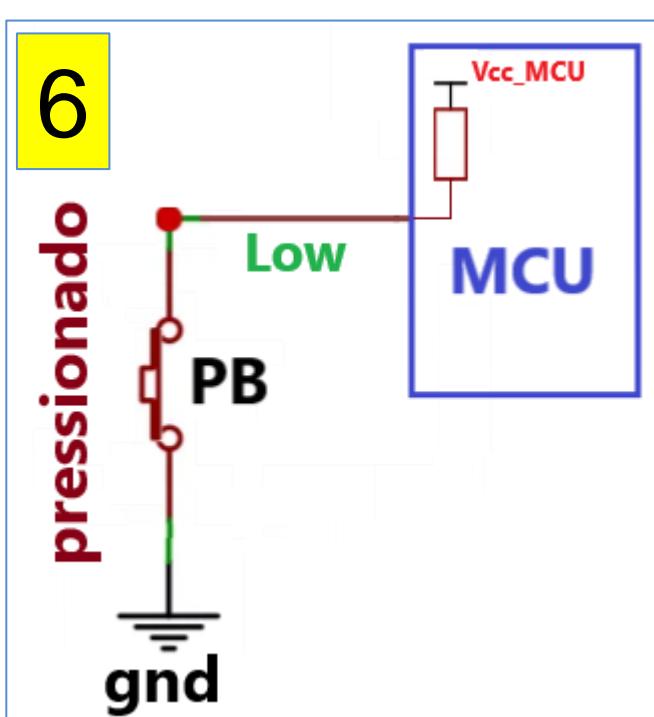
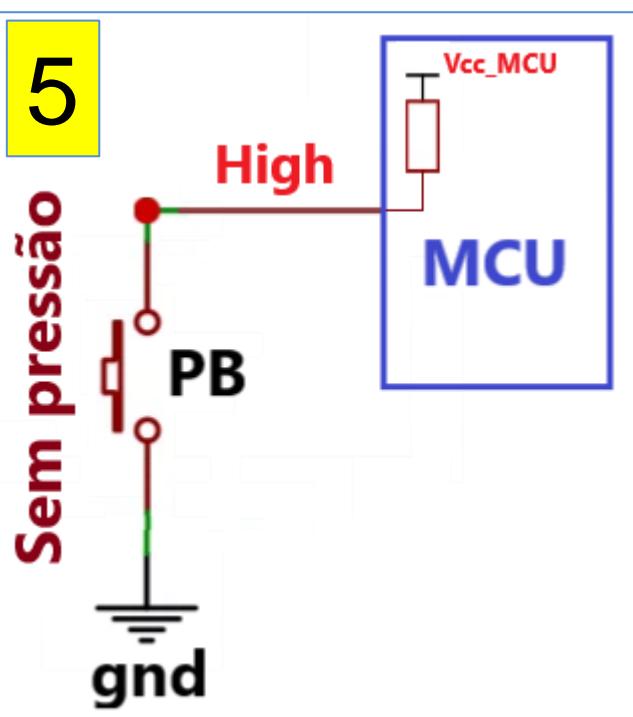
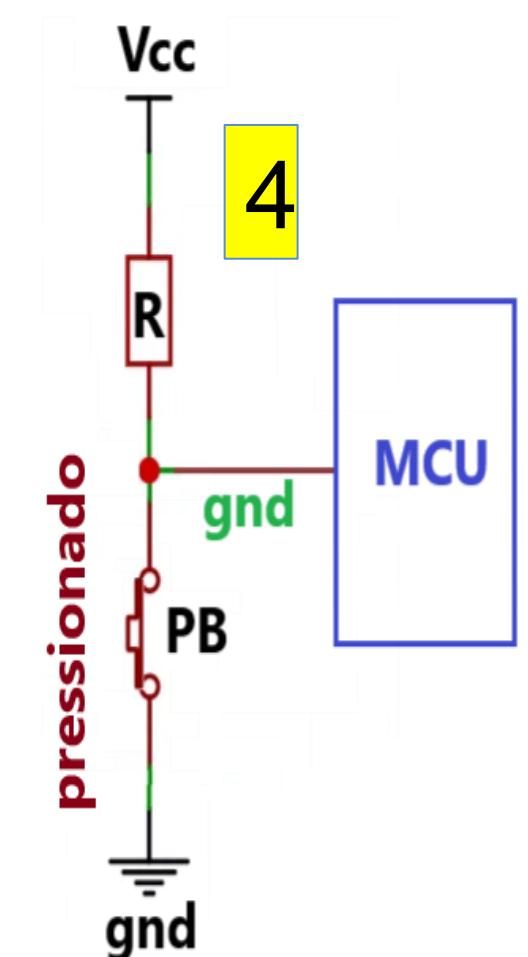
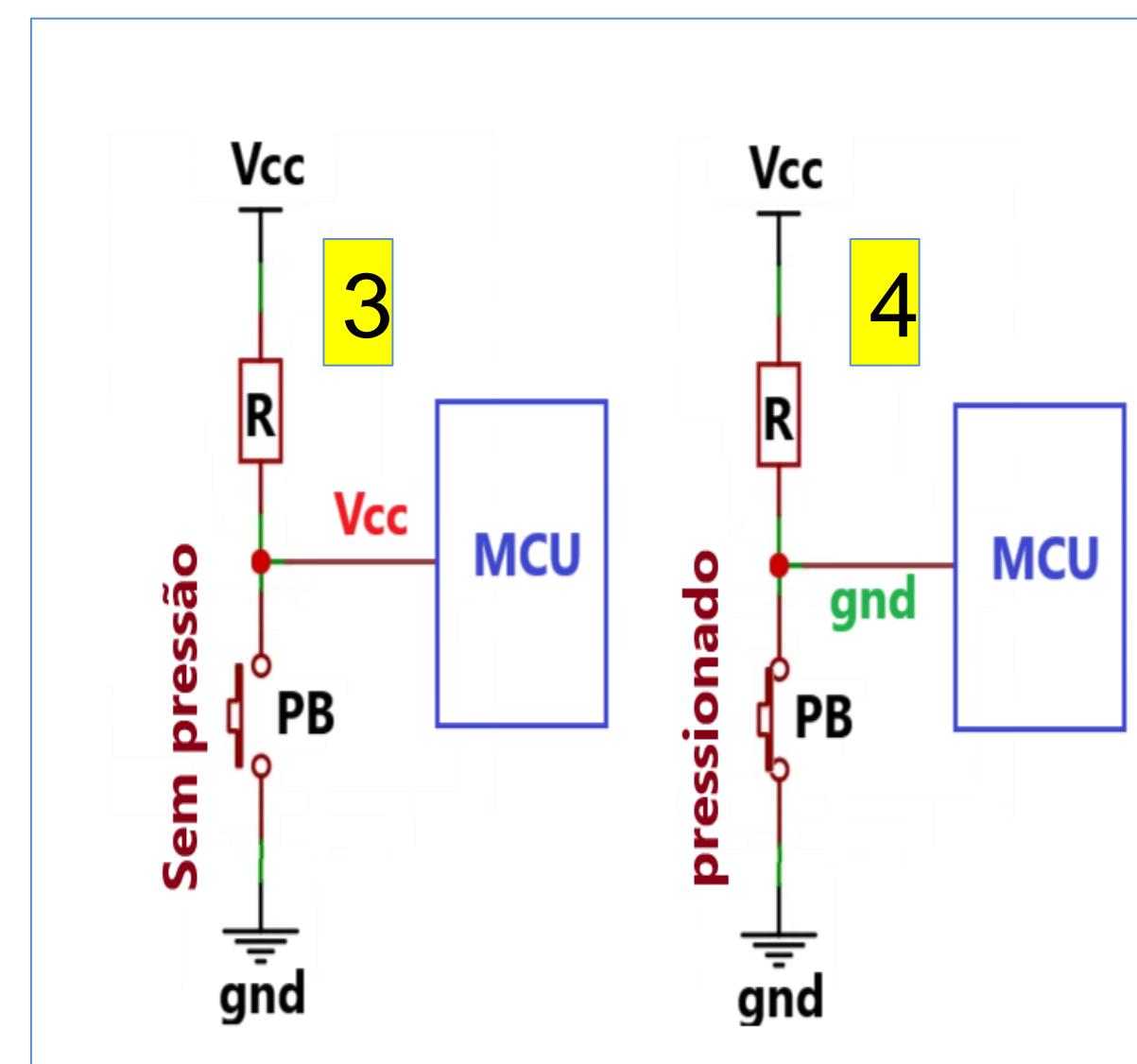
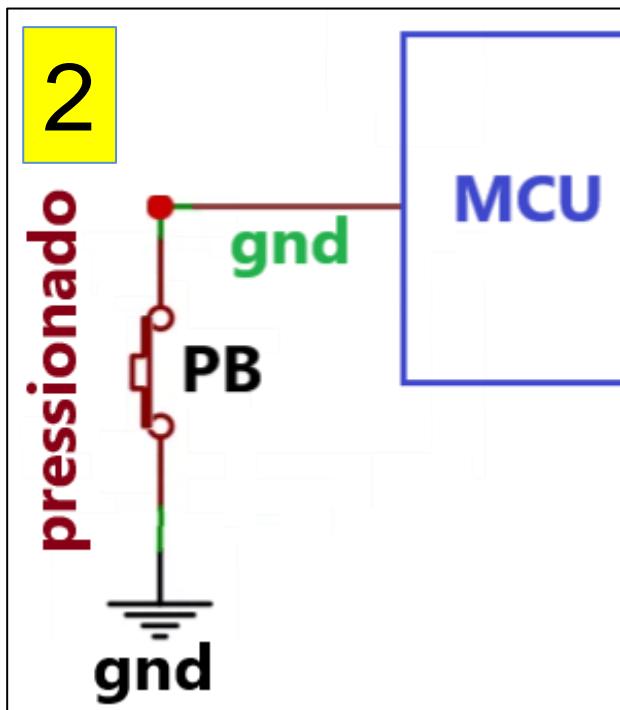
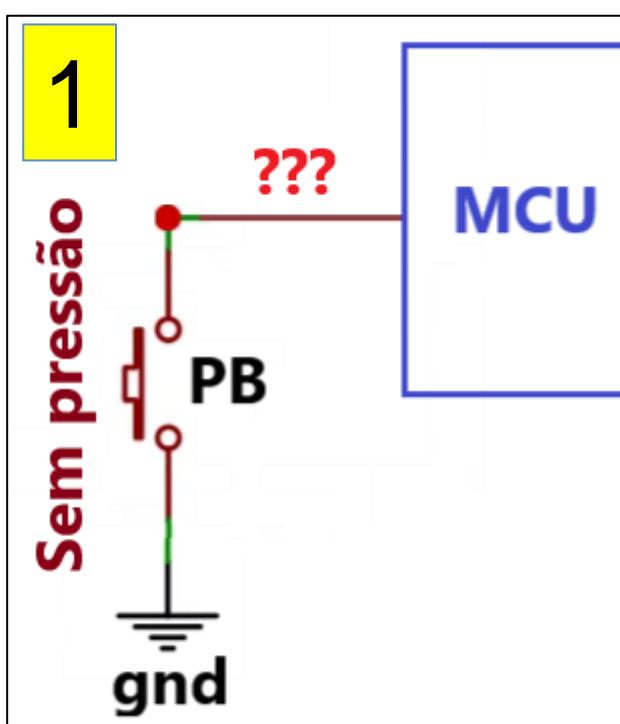
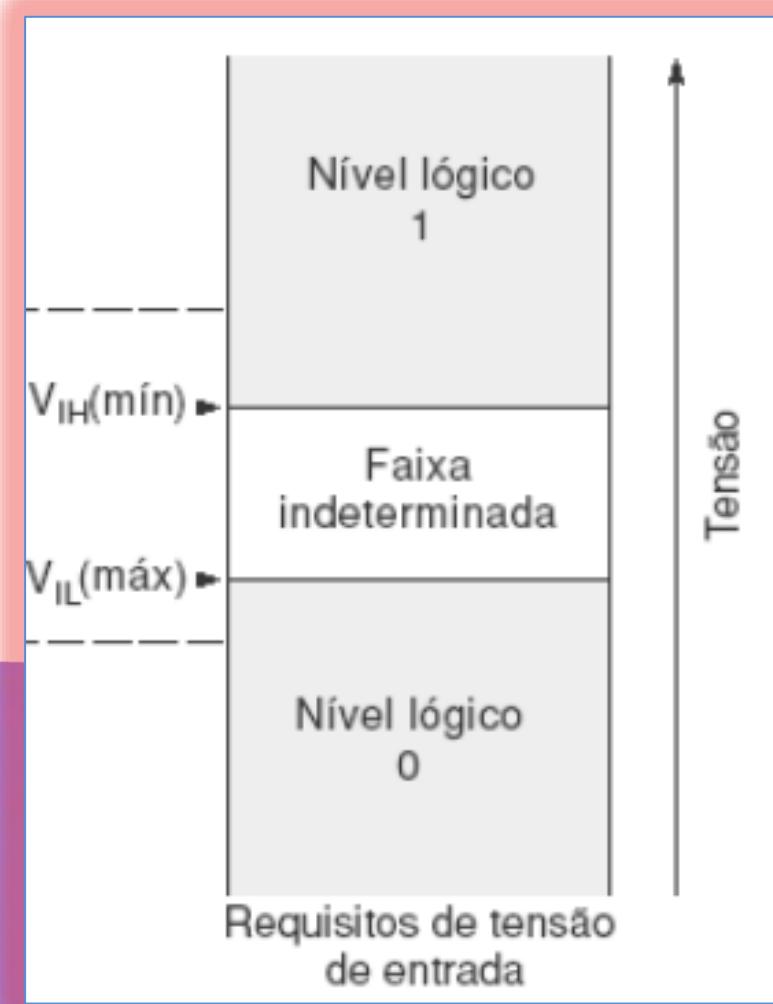
Resistores **pull-up** e **pull-down** são componentes eletrônicos que definem o estado de um pino de um circuito quando não há sinal ativo. Eles são usados para garantir que o sinal digital seja bem definido.

Figuras 1 e 2: Sem Pull-up.

Note que o pino está em um estado lógico indefinido.

Figuras 3 e 4: Resistores de Pull-up externos.

Figuras 5 e 6: Resistores de Pull-up internos ao micro controlador (MCU) ativados pela configuração do software.



Polling e Interrupções

Imagine que você está lendo um livro enquanto aguarda a entrega de um pacote. Como não possui campainha, o entregador combinou de deixar o pacote discretamente na porta, sem fazer barulho. (Ou seja, você não será interrompido.) Nesse cenário, de tempos em tempos, você interromperia sua leitura para verificar se o pacote já chegou. No contexto do MCU, essa verificação periódica é conhecida como **polling**.



O termo polling pode ser visto, no contexto de microcontroladores, como:

- Consulta contínua e periódica.
- Varredura.
- Sondagem.

Polling e Interrupções

Comparação

- Polling: O MCU verifica repetidamente se um evento ocorreu, consumindo recursos do processador mesmo quando nada acontece.
- Interrupções: O MCU espera passivamente até receber um sinal de interrupção, o que é mais eficiente, pois libera o processador para executar outras tarefas enquanto aguarda eventos.

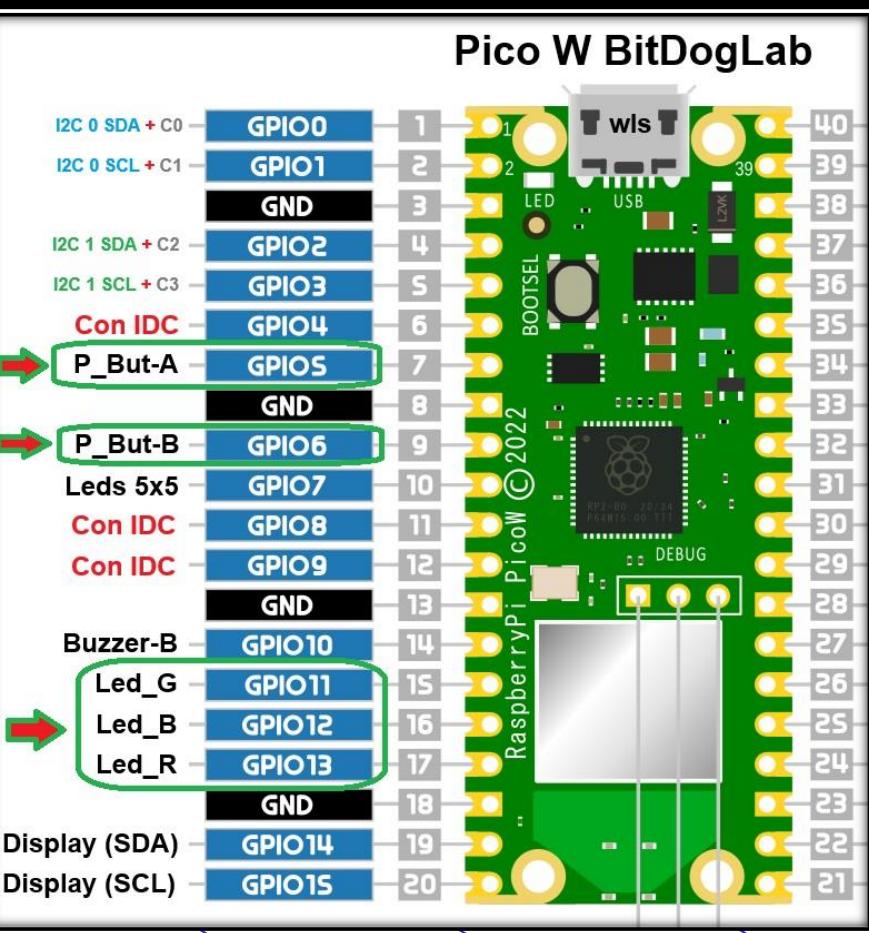
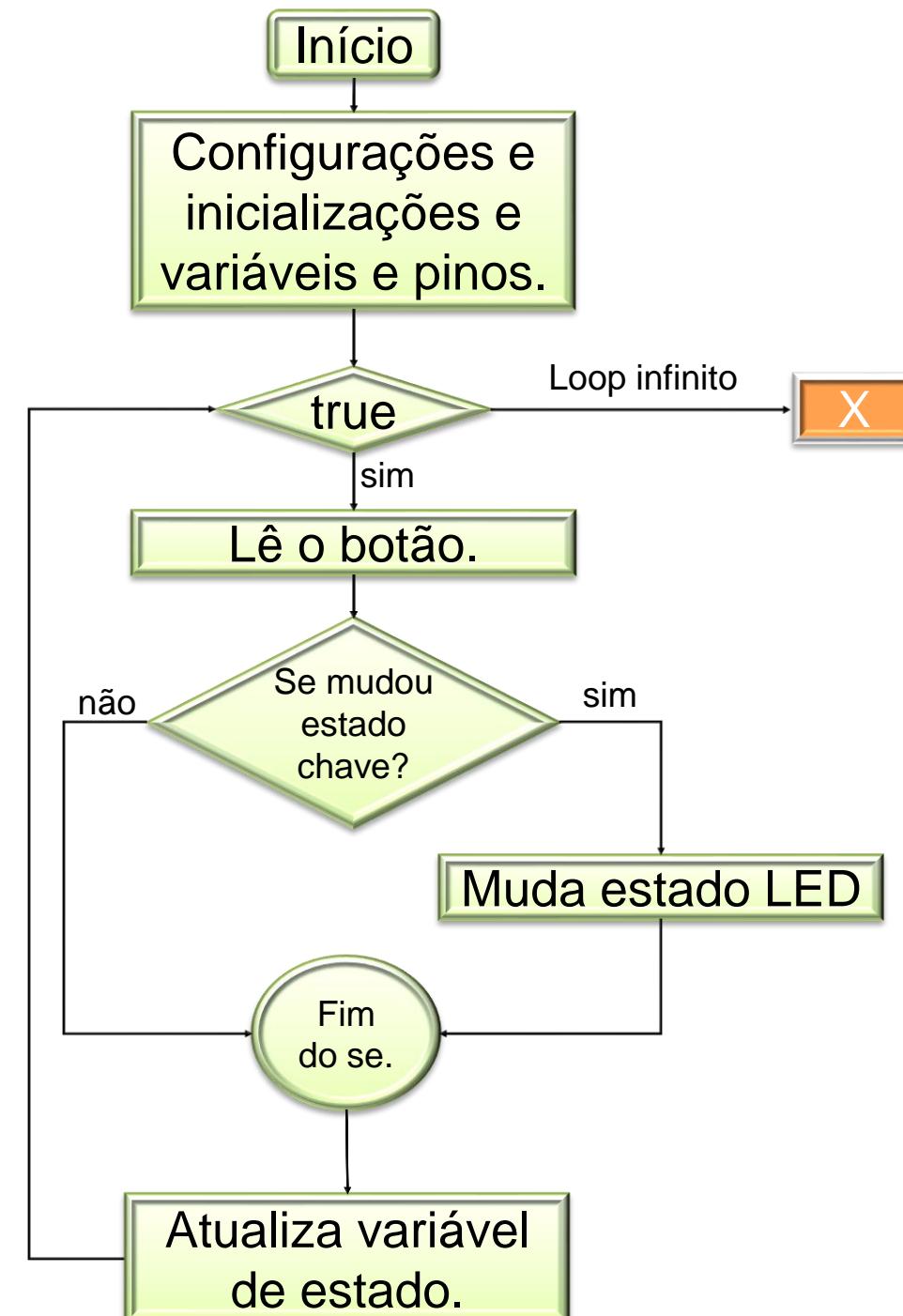
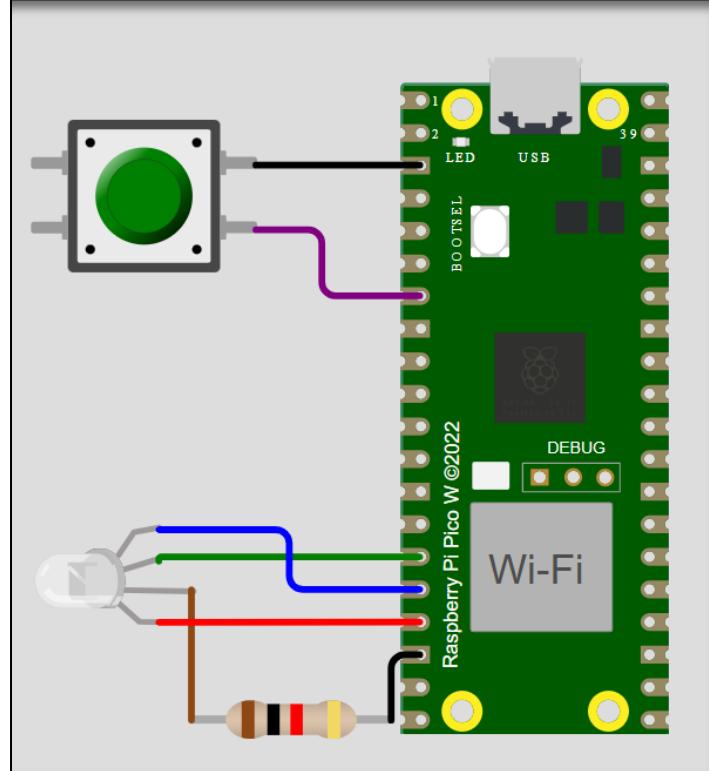


Exemplo prático de polling:

Um programa verifica constantemente se um botão foi pressionado.

Exemplo de polling no RP2040

Neste exemplo o usuário pressionará um botão e o LED VERDE mudará o seu estado.

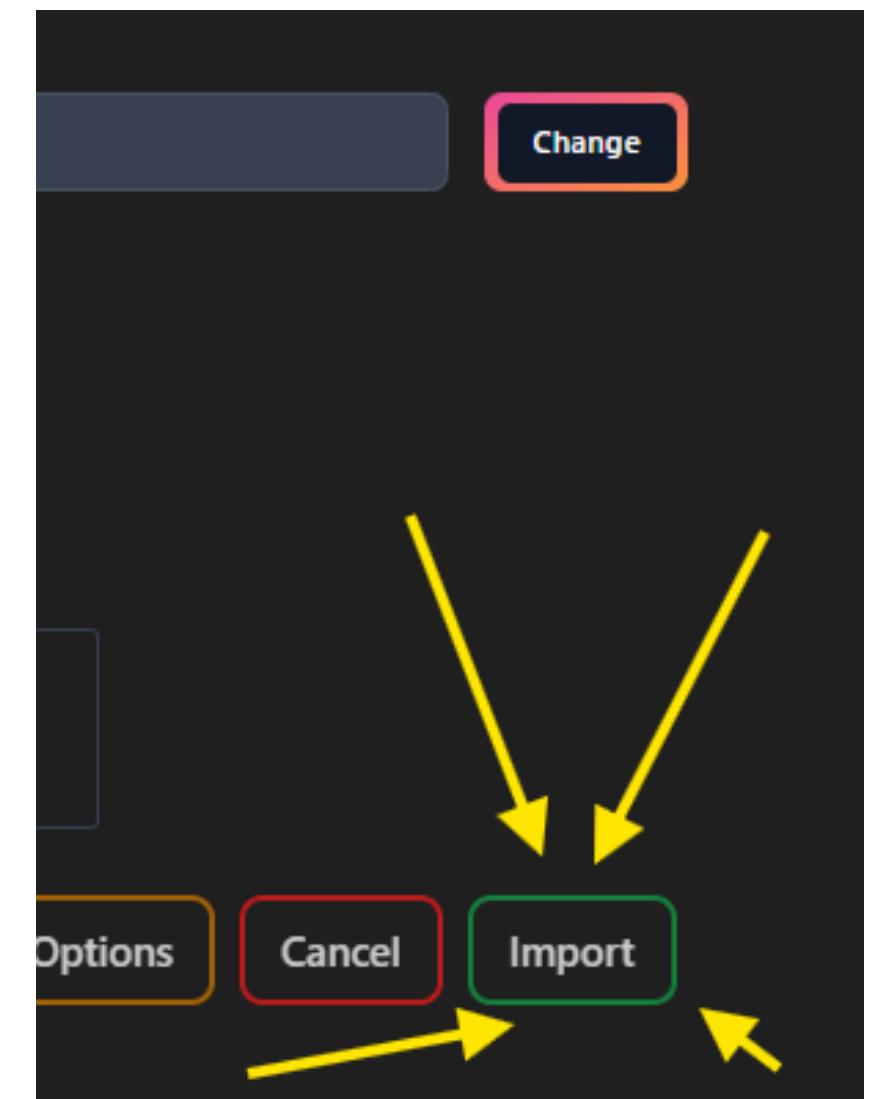
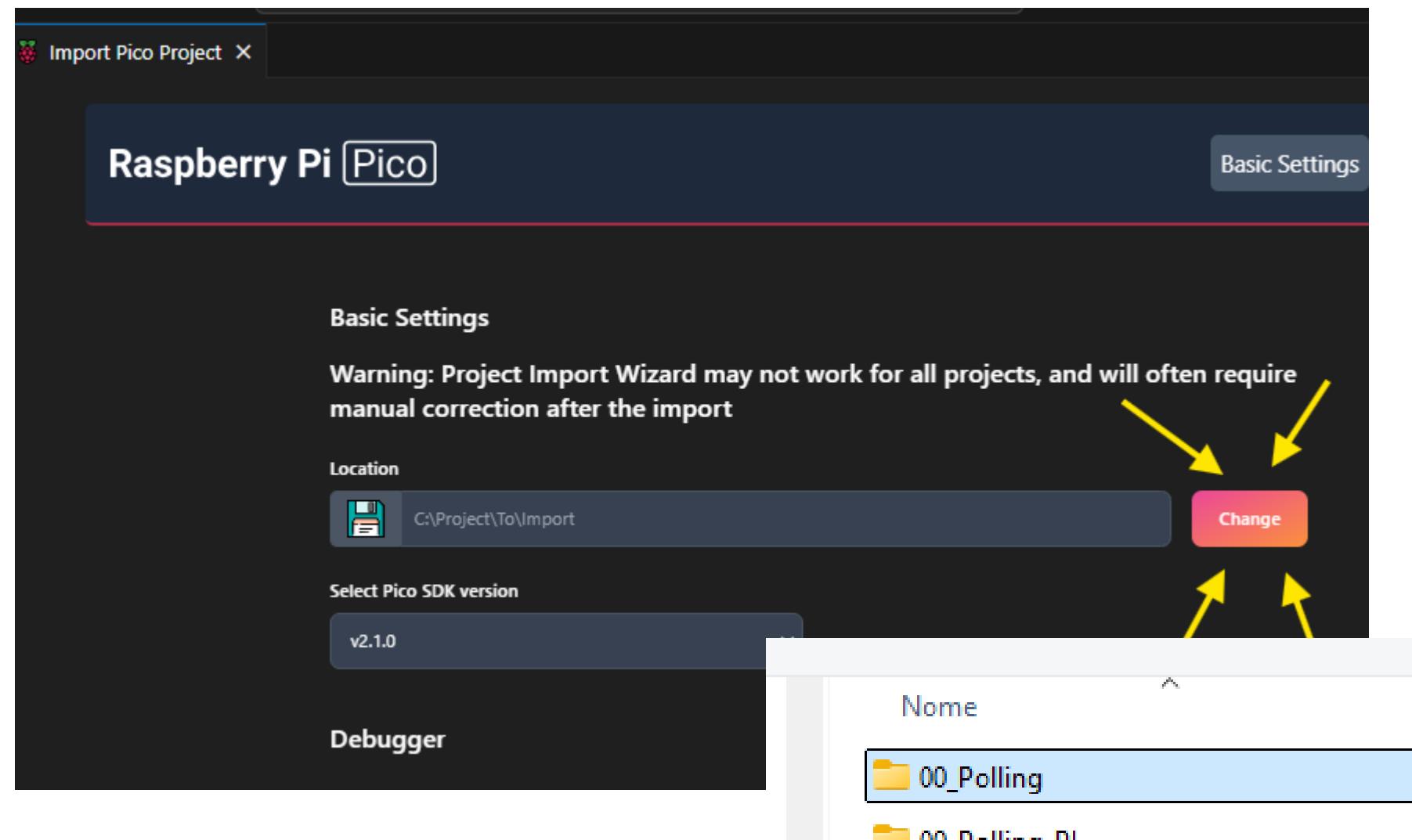
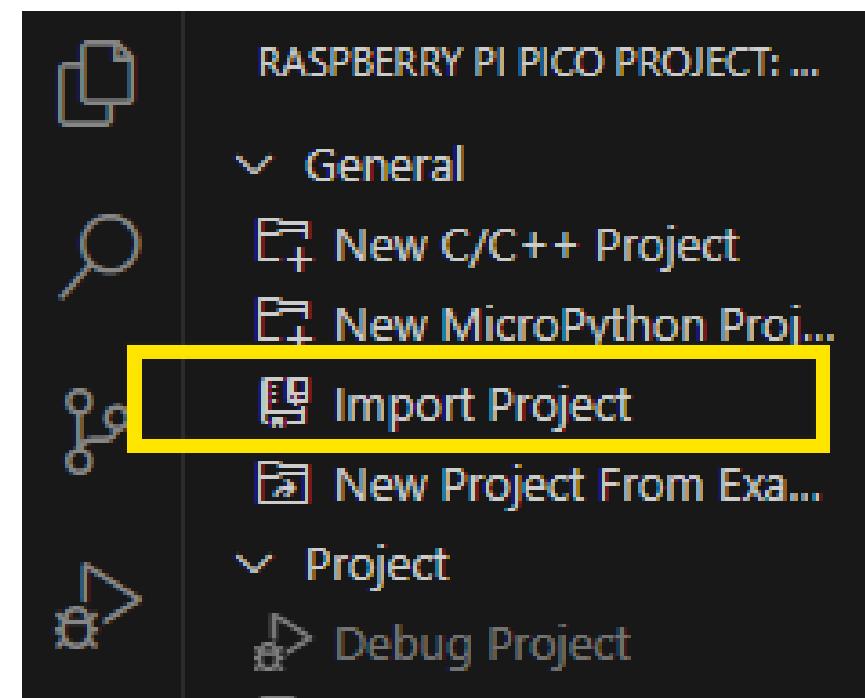


```
c SemInterrupt.c ...
1 #include "pico/stdlib.h"
2 // Configurações dos pinos
3 const uint led_pin = 11;      // Red=13, Blue=12, Green=11
4 const uint botao_pin = 5;     // Botão A = 5, Botão B = 6 , BotãoJoy = 22
5 int main()
6 {
7     // Inicializações
8     gpio_init(led_pin);          // Inicializa o pino do LED
9     gpio_set_dir(led_pin, GPIO_OUT); // Configura o pino como saída
10    gpio_put(led_pin, 0);         // Garante que o LED inicie apagado
11
12    gpio_init(botao_pin);        // Inicializa o botão
13    gpio_set_dir(botao_pin, GPIO_IN); // Configura o pino como entrada
14    gpio_pull_up(botao_pin);      // Habilita o pull-up interno
15    // Variáveis para o controle do estado do LED
16    bool led_estado = false;      // Estado inicial do LED (apagado)
17    bool ultimo_estado_botao = true; // Último estado do botão (inicialmente solto)
18    // Loop principal
19    while (true) {
20        // Lê o estado atual do botão
21        bool estado_atual_botao = gpio_get(botao_pin);
22        // Detecta uma transição do botão (pressionado e solto)
23        if (estado_atual_botao == false && ultimo_estado_botao == true) {
24            // Transição de solto para pressionado: alterna o estado do LED
25            led_estado = !led_estado;
26            gpio_put(led_pin, led_estado); // Atualiza o estado do LED
27        }
28        // Atualiza o estado anterior do botão
29        ultimo_estado_botao = estado_atual_botao;
30    }
31 }
```

Vá no: github.com/wiltonlacerda
git clone <https://github.com/wiltonlacerda/EmbarcaTechU4C4.git>

Exemplo de polling no RP2040

Na extensão do RP2 vá em “Import Project”, depois selecione a pasta e finalmente clicar em “Import”



Pollling e Interrupções

No contexto de um microcontrolador, uma interrupção é um sinal que interrompe o fluxo normal de execução do programa para que uma tarefa específica possa ser realizada imediatamente.

1. Evento de Interrupção: Pode ser um evento externo (como um botão sendo pressionado) ou interno (**como um temporizador**).
2. Reconhecimento de Interrupção: O microcontrolador reconhece o sinal de interrupção.
3. Execução da Rotina de Interrupção: O microcontrolador salva o estado atual de execução e executa uma rotina de serviço de interrupção (**ISR**) para lidar com o evento.
4. Retorno à Execução Normal: Após completar a **ISR**, o microcontrolador retorna ao ponto de parada da interrupção e retoma a execução normal do programa.



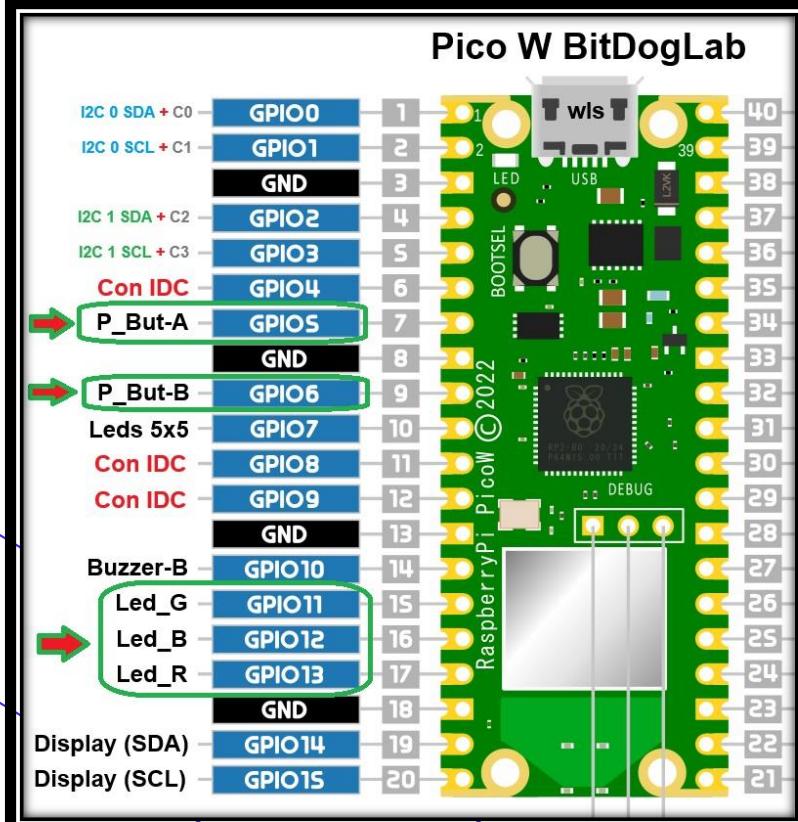
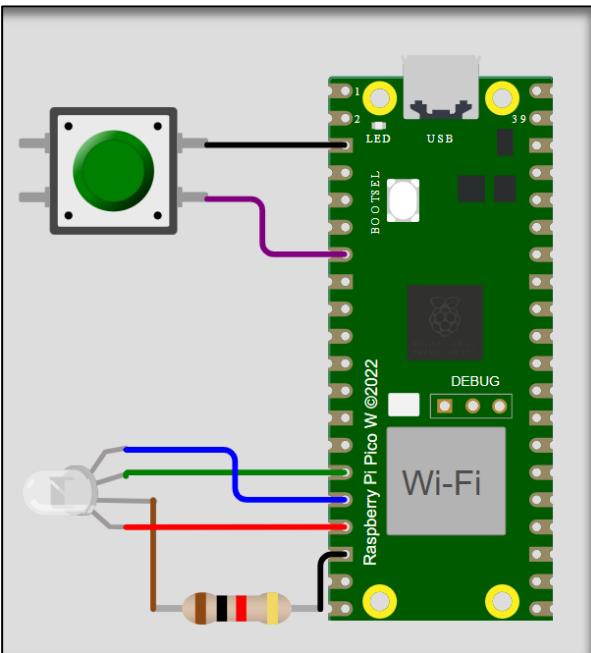
Comparação de uma interrupção de leitura de livro com o processo executado em um microcontrolador.

Interrupção em Casa	Interrupção no MCU
1) Você está lendo um livro.	O programa principal está em execução.
2) O entregador toca a campainha.	Um sinal de interrupção informa ao MCU que um evento ocorreu.
3) Você para de ler.	O MCU recebe o sinal de interrupção e suspende a execução do programa principal.
4) Marca a página em que estava.	O MCU salva o estado atual da execução do programa em seus registradores.
5) Recebe a entrega.	O MCU executa a rotina de interrupção correspondente ao evento recebido.
6) Retorna à página marcada.	O MCU restaura o estado salvo da execução do programa.
7) Retoma a leitura de onde parou.	O programa principal continua a partir de onde foi interrompido.

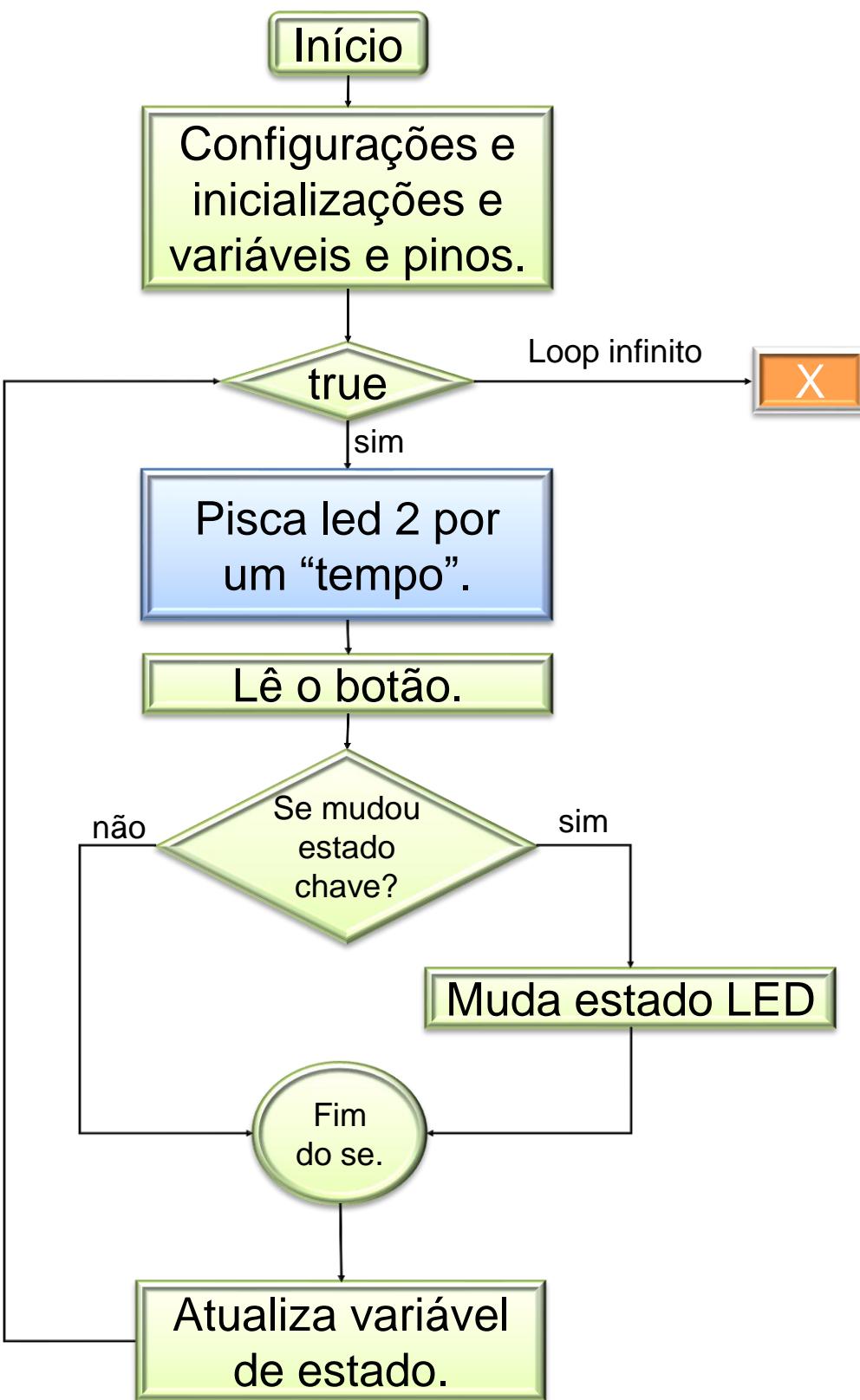


Sem uso de Interrupções no RP2040

Neste exemplo o LED GREEN piscará continuamente, e quando o usuário pressionar o botão A o LED RED, eventualmente, deve mudar o seu estado.



Projeto: 01_PollingBlink



```

c SemInterruptBlink.c > main()
1 #include "pico/stl.h"
2 const uint led_pin = 11;           // Red=13, Blue=12, Green=11
3 const uint led_pin_pisca = 13;    // LED para piscar
4 const uint botao_pin = 5;         // Botão A = 5, Botão B = 6 , BotãoJoy = 22
5 #define tempo 1000
6 int main()
7 {
8     gpio_init(led_pin);           // Inicializa o pino do LED
9     gpio_set_dir(led_pin, GPIO_OUT); // Configura o pino como saída
10    gpio_put(led_pin, 0);          // Garante que o LED inicie apagado
11    gpio_init(led_pin_pisca);      // Inicializa o pino do LED para piscar
12    gpio_set_dir(led_pin_pisca, GPIO_OUT); // Configura o pino como saída
13    gpio_put(led_pin_pisca, 0);    // Garante que o LED inicie apagado
14    gpio_init(botao_pin);         // Inicializa o botão
15    gpio_set_dir(botao_pin, GPIO_IN); // Configura o pino como entrada
16    gpio_pull_up(botao_pin);      // Habilita o pull-up interno
17    bool led_estado = false;       // Estado inicial do LED (apagado)
18    bool ultimo_estado_botao = true; // Último estado do botão (inicialmente solto)
19    while (true)
20    {
21        gpio_put(led_pin_pisca, 1); // Liga o LED
22        sleep_ms(tempo);          // Mantém ligado por "tempo" ms
23        gpio_put(led_pin_pisca, 0); // Desliga o LED
24        sleep_ms(tempo);          // Mantém desligado por "tempo" ms
25        bool estado_atual_botao = gpio_get(botao_pin);
26        if (estado_atual_botao == false && ultimo_estado_botao == true)
27        {
28            led_estado = !led_estado;
29            gpio_put(led_pin, led_estado); // Atualiza o estado do LED
30        }
31        ultimo_estado_botao = estado_atual_botao;
32    }
33 }

```

Interrupções no RP2040

Cada núcleo do RP2040 é equipado com um ARM **NVIC** (Nested Vectored Interrupt Controller), que conta com 32 entradas de interrupção.

Cada **NVIC** tem as mesmas interrupções roteadas para ele, com exceção das interrupções GPIO: há uma interrupção GPIO por banco, por núcleo. Elas são completamente independentes, então, por exemplo, o núcleo 0 pode ser interrompido pelo GPIO 0 no banco 0, e o núcleo 1 pelo GPIO 1 no mesmo banco.

No RP2040, apenas os 26 sinais IRQ inferiores são conectados no **NVIC**, e as IRQs 26 a 31 são vinculadas a zero (nunca disparando).

Interrupções no RP2040

Este microcontrolador é dotado de um módulo **ARM NVIC** para cada núcleo.
Das 32 interrupções 26 são utilizáveis.

IRQ	Interrupt Source								
0	TIMER_IRQ_0	6	XIP_IRQ	12	DMA_IRQ_1	18	SPI0_IRQ	24	I2C1_IRQ
1	TIMER_IRQ_1	7	PIO0_IRQ_0	13	IO_IRQ_BANK0	19	SPI1_IRQ	25	RTC_IRQ
2	TIMER_IRQ_2	8	PIO0_IRQ_1	14	IO_IRQ_QSPI	20	UART0_IRQ		
3	TIMER_IRQ_3	9	PIO1_IRQ_0	15	SIO_IRQ_PROC0	21	UART1_IRQ		
4	PWM_IRQ_WRAP	10	PIO1_IRQ_1	16	SIO_IRQ_PROC1	22	ADC_IRQ_FIFO		
5	USBCTRL_IRQ	11	DMA_IRQ_0	17	CLOCKS_IRQ	23	I2C0_IRQ		

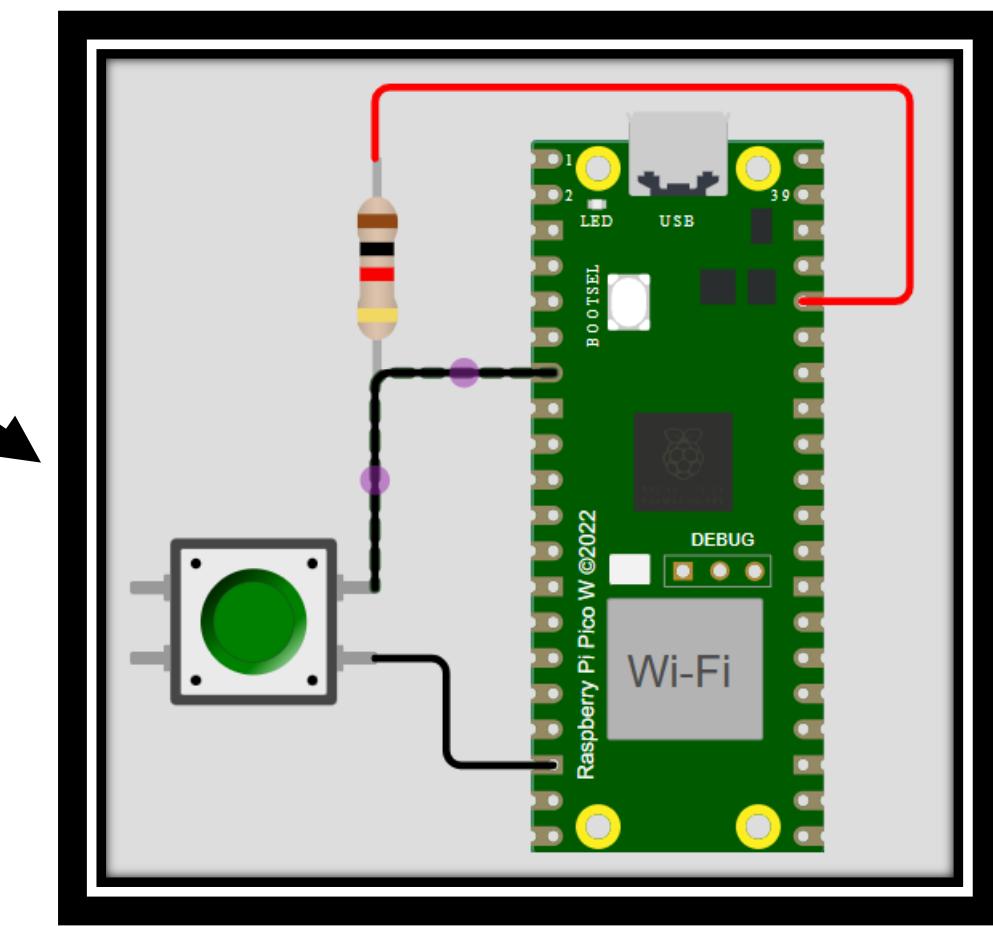
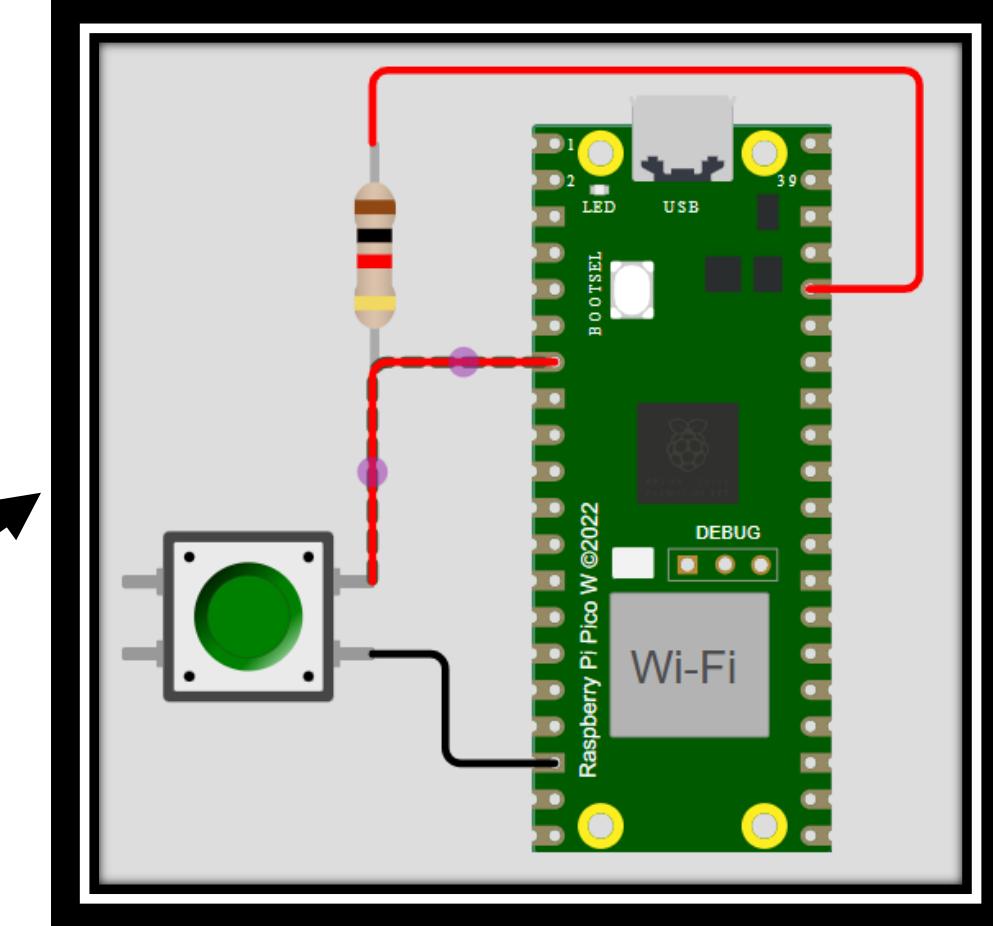
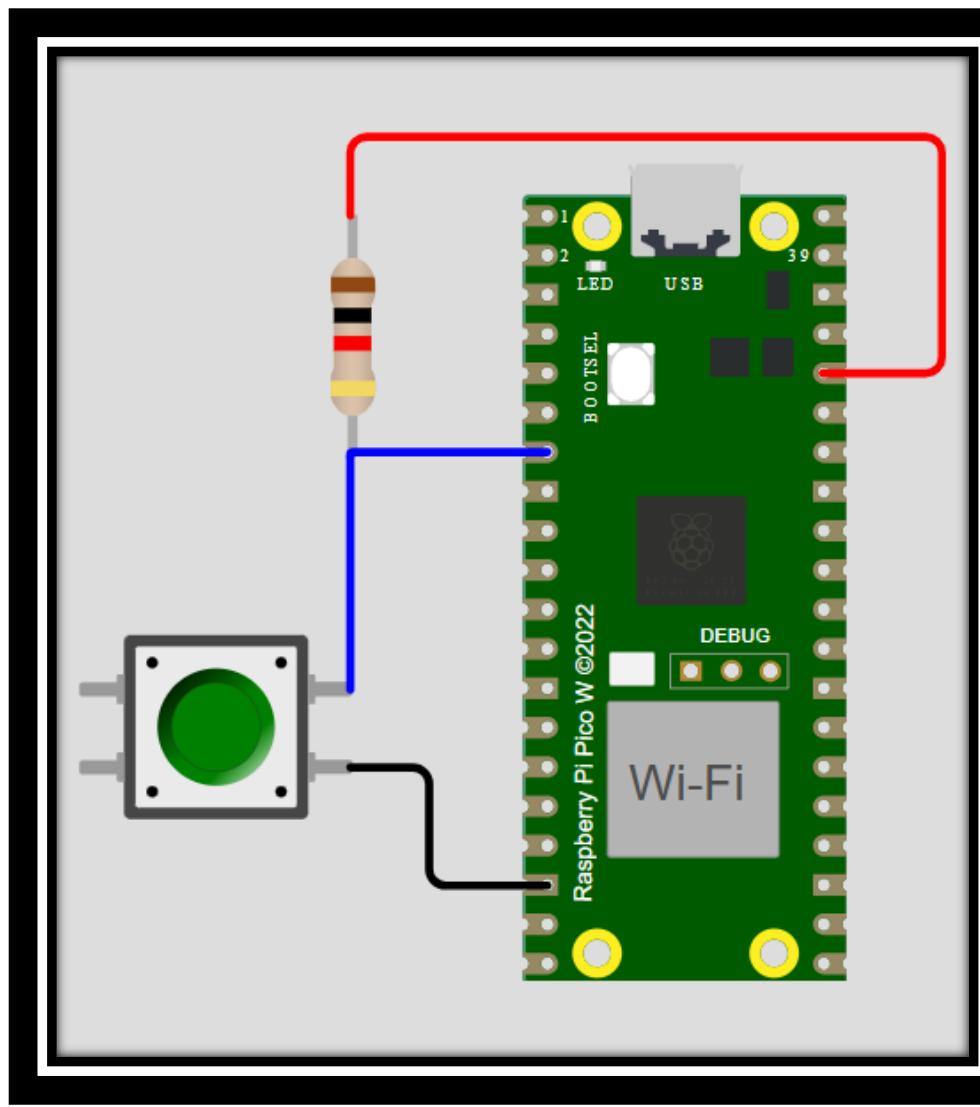
Interrupções do módulo GPIO

Cada pino do GPIO possui quatro interrupções:

- Níveis de prioridade
 - » GPIO_IRQ_LEVEL_LOW
 - » GPIO_IRQ_LEVEL_HIGH
 - » GPIO_IRQ_EDGE_FALL
 - » GPIO_IRQ_EDGE_RISE
 - » Normalmente, nós usamos as duas últimas que indicam
 - * Borda de Descida
 - * Borda de Subida
 - » Existe apenas uma interrupção que atende todos os eventos de GPIO
 - * De todos os pinos!

Somente para reforçar o aprendizado

Como se comporta o resistor de Pull-Up



1.I 1.r

2.I 2.r

1.I 1.r

2.I 2.r

Interrupções no RP2040

Raspberry Pi Pico-series C/C++ SDK

Chamada da função de interrupção

4.1.10.4. Functions

```
void gpio_set_function (uint gpio, gpio_function_t fn)
```

Select GPIO function.

```
void gpio_set_function_masked (uint32_t gpio_mask, gpio_function_t fn)
```

Select the function for multiple GPIOs.

```
void gpio_set_irq_callback (gpio_irq_callback_t callback)
```

Set the generic callback used for GPIO IRQ events for the current core.

```
void gpio_set_irq_enabled_with_callback (uint gpio, uint32_t event_mask, bool enabled, gpio_irq_callback_t callback)
```

Convenience function which performs multiple GPIO IRQ related initializations.

```
void gpio_set_dormant_irq_enabled (uint gpio, uint32_t event_mask, bool enabled)
```

Enable dormant wake up interrupt for specified GPIO and events.

```
static uint32_t gpio_get_irq_event_mask (uint gpio)
```

Return the current interrupt status (pending events) for the given GPIO.

```
void gpio_acknowledge_irq (uint gpio, uint32_t event_mask)
```

Acknowledge a GPIO interrupt for the specified events on the calling core.

4.1.10.6.3. gpio_irq_level

```
enum gpio_irq_level
```

An interrupt can be generated for every GPIO pin in 4 scenarios:

- Level High: the GPIO pin is a logical 1
- Level Low: the GPIO pin is a logical 0
- Edge High: the GPIO has transitioned from a logical 0 to a logical 1
- Edge Low: the GPIO has transitioned from a logical 1 to a logical 0

GPIO_IRQ_LEVEL_LOW	IRQ when the GPIO pin is a logical 1.
GPIO_IRQ_LEVEL_HIGH	IRQ when the GPIO pin is a logical 0.
GPIO_IRQ_EDGE_FALL	IRQ when the GPIO has transitioned from a logical 0 to a logical 1.
GPIO_IRQ_EDGE_RISE	IRQ when the GPIO has transitioned from a logical 1 to a logical 0.

```
gpio_set_irq_enabled_with_callback(button_0, GPIO_IRQ_EDGE_FALL, true, &gpio_irq_handler);
```

```
void gpio_irq_handler(uint gpio, uint32_t events)
{
    bool estado_atual = gpio_get(led_pin); // Obtém o estado atual
    gpio_put(led_pin, !estado_atual); // Alterna o estado
}
```

Interrupções no RP2040

Interrupções.

Uso do botão A e um LED RGB da placa BitDogLab para demonstração do funcionamento das interrupções.

O código a seguir faz com que o LED Blue, seja modificado a cada pressionada do botão.

```
1 //include "pico/stl.h"
2 //include "hardware/timer.h"
3 // Configurações dos pinos
4 const uint led_pin_red = 12;      //Red=13, Blue=12, Green=11
5 const uint button_0 = 5;          // Botão A = 5, Botão B = 6 , BotãoJoy = 22
6 // Prototipação da função de interrupção
7 static void gpio_irq_handler(uint gpio, uint32_t events);
8 int main()
9 {
10     // Inicializações
11     gpio_init(led_pin_red);        // Inicializa o pino do LED
12     gpio_set_dir(led_pin_red, GPIO_OUT); // Configura o pino como saída
13     gpio_init(button_0);           // Inicializa o botão
14     gpio_set_dir(button_0, GPIO_IN); // Configura o pino como entrada
15     gpio_pull_up(button_0);        // Habilita o pull-up interno
16     // Configuração da interrupção com callback
17     gpio_set_irq_enabled_with_callback(button_0, GPIO_IRQ_EDGE_FALL, true, &gpio_irq_handler);
18     // Loop principal
19     while (true)
20     {
21         // Não tem programa no loop.
22     }
23 }
24 // Função de interrupção "É possível observar o bounce do botão"
25 void gpio_irq_handler(uint gpio, uint32_t events)
26 {
27     bool estado_atual = gpio_get(led_pin_red); // Obtém o estado atual
28     gpio_put(led_pin_red, !estado_atual);       // Alterna o estado
29 }
```

Interrupções no RP2040

Interrupções.

Uso do botão A LEDs RGB da placa BitDogLab para demonstração do funcionamento das interrupções.

O código faz o LED verde piscar a cada "tempo" ms.

Quando o botão A é pressionado, o LED azul alterna de estado.

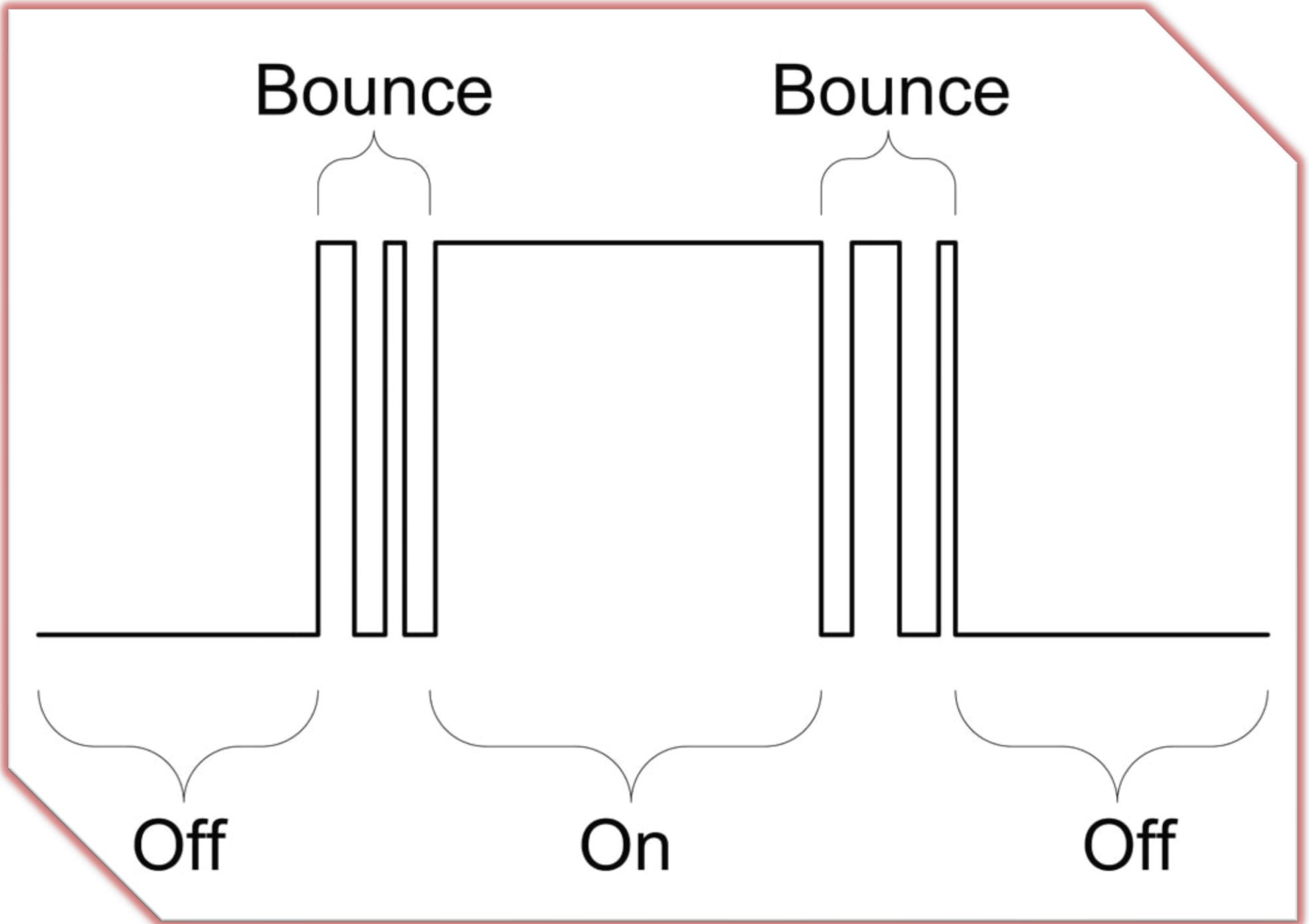
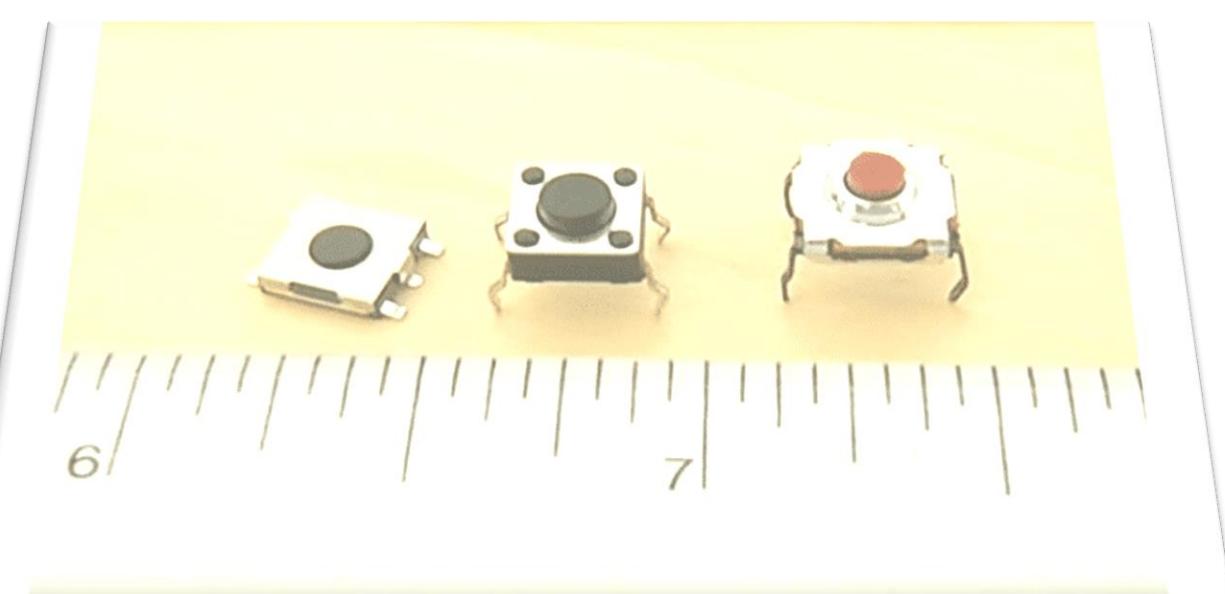
```
C Interrupt_Blink.c x
C Interrupt_Blink.c > gpio_irq_handler(uint, uint32_t)
1 #include "pico/stdlib.h"
2 const uint ledA_pin = 12;      // Blue => GPIO12
3 const uint ledB_pin = 11;      // Green => GPIO11
4 const uint button_0 = 5;       // Botão A = 5, Botão B = 6 , BotãoJoy = 22
5 #define tempo 2500
6 static void gpio_irq_handler(uint gpio, uint32_t events);
7 int main()
8 {
9     gpio_init(ledA_pin);          // Inicializa o pino do LED
10    gpio_set_dir(ledA_pin, GPIO_OUT); // Configura o pino como saída
11    gpio_init(ledB_pin);          // Inicializa o pino do LED
12    gpio_set_dir(ledB_pin, GPIO_OUT); // Configura o pino como saída
13    gpio_init(button_0);          // Inicializa o botão
14    gpio_set_dir(button_0, GPIO_IN); // Configura o pino como entrada
15    gpio_pull_up(button_0);        // Habilita o pull-up interno
16    gpio_set_irq_enabled_with_callback(button_0, GPIO_IRQ_EDGE_FALL, true, &gp
17
18    while (true)
19    {
20        gpio_put(ledB_pin, true);
21        sleep_ms(tempo);
22        gpio_put(ledB_pin, false);
23        sleep_ms(tempo);
24    }
25 }
26 // Função de interrupção "É possível observar o bounce do botão"
27 void gpio_irq_handler(uint gpio, uint32_t events)
28 {
29     bool estado_atual = gpio_get(ledA_pin); // Obtém o estado atual
30     gpio_put(ledA_pin, !estado_atual);      // Alterna o estado
31 }
```

Ln 31, Col 2 Spaces: 4 UTF-8 CRLF {} C Run Compile 25

Bouncing

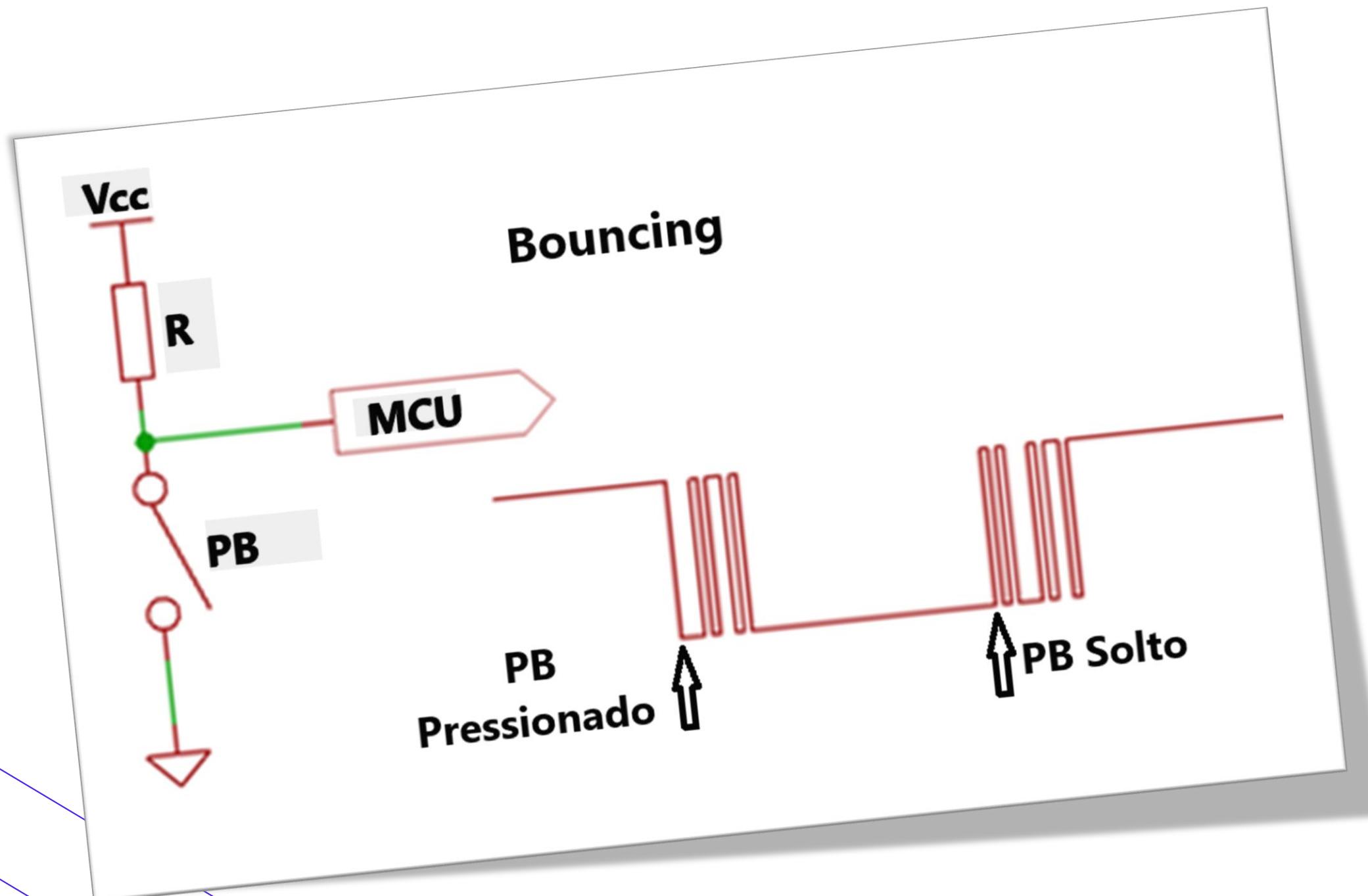
O que é Efeito Bouncing?

O Efeito Bouncing nada mais é que um fenômeno de **trepidação**. Ele é comum em Chaves, push buttons, reed-switch, entre outros. Ocorre quando os contatos internos não se conectam ou desconectam instantaneamente ao serem acionados. Isso resulta em múltiplos sinais de liga/desliga (picos de tensão) em um curto intervalo de tempo, em vez de uma transição limpa entre os estados.



<https://vhdlguru.blogspot.com/2017/09/pushbutton-debounce-circuit-in-vhdl.html>

Bouncing



Consequências do Bouncing

1. Ruído no sinal:

Um único pressionamento de botão pode ser lido como múltiplos pulsos pelo microcontrolador ou circuito.

2. Comportamento indesejado:

Em sistemas digitais, isso pode causar erros como: Acionamento repetido de funções.
Detectção incorreta de estados.

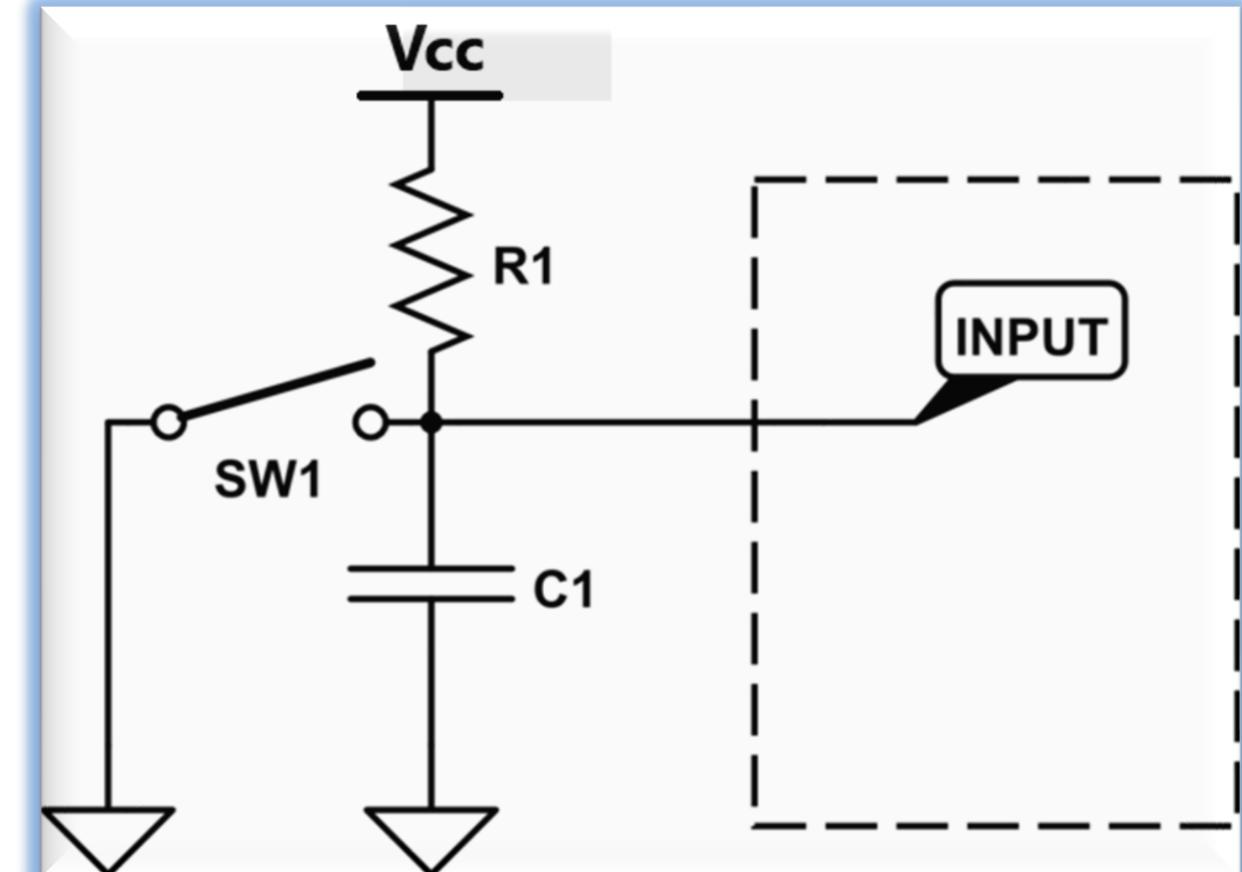
Como implementar o Debounce

1. Debounce por Hardware

- **Capacitor:** Conecte um capacitor em paralelo ao botão para filtrar picos rápidos (filtro RC).
- **Schmitt Trigger:** Use um circuito integrado para gerar transições de estado limpas.
- **Filtros RC e resistores pull-up/pull-down:** Atenuam as oscilações.

2. Debounce por Software

- **Atraso fixo (delay):** Ignore leituras adicionais do botão por um intervalo de tempo após a primeira detecção.



Verificação do efeito bounce

Mesmo programa anterior. Só acrescentou
printf e “**a**”.

O código a seguir faz com que o Led azul seja
modificado a cada pressionada do botão. Já o LED verde
pisca a cada “**tempo**” ms.

Uma variável de verificação é incrementada a cada
interrupção.

Esta variável é exibida no **serial monitor** para verificar o
efeito Bounce.

Obs. Rodar no Wokwi também.

```
C Interrupt.c > ...
1 const uint ledA_pin = 12; // Blue => GPIO12
2 const uint ledB_pin = 11; // Green=> GPIO11
3 const uint button_0 = 5; // Botão A = 5, Botão B = 6 , BotãoJoy = 22
4 #define tempo 2500
5 static volatile uint a = 1;
6 static void gpio_irq_handler(uint gpio, uint32_t events);
7 int main()
8 {
9     stdio_init_all(); // Inicializa o terminal serial
10    gpio_init(ledA_pin); // Inicializa o pino do LED
11    gpio_set_dir(ledA_pin, GPIO_OUT); // Configura o pino como saída
12    gpio_init(ledB_pin); // Inicializa o pino do LED
13    gpio_set_dir(ledB_pin, GPIO_OUT); // Configura o pino como saída
14    gpio_init(button_0);
15    gpio_set_dir(button_0, GPIO_IN); // Configura o pino como entrada
16    gpio_pull_up(button_0); // Habilita o pull-up interno
17    gpio_set_irq_enabled_with_callback(button_0, GPIO_IRQ_EDGE_FALL, true, &gp
18    while (true)
19    {
20        gpio_put(ledB_pin, true);
21        sleep_ms(tempo);
22        gpio_put(ledB_pin, false);
23        sleep_ms(tempo);
24    }
25 }
26 void gpio_irq_handler(uint gpio, uint32_t events)
27 {
28     printf("Mudanca de Estado do Led. A = %d\n", a);
29     gpio_put(ledA_pin, !gpio_get(ledA_pin)); // Alterna o estado
30     a++; // incrementa a variavel de verificação
31 }
```

Debounce por Software

**Mesmo programa anterior, modificando a função
`gpio_irq_handler()`**

O código a seguir faz com que o Led azul seja modificado a cada pressionada do botão. Já o LED verde pisca a cada tempo ms.

Uma variável de verificação é incrementada a cada interrupção.

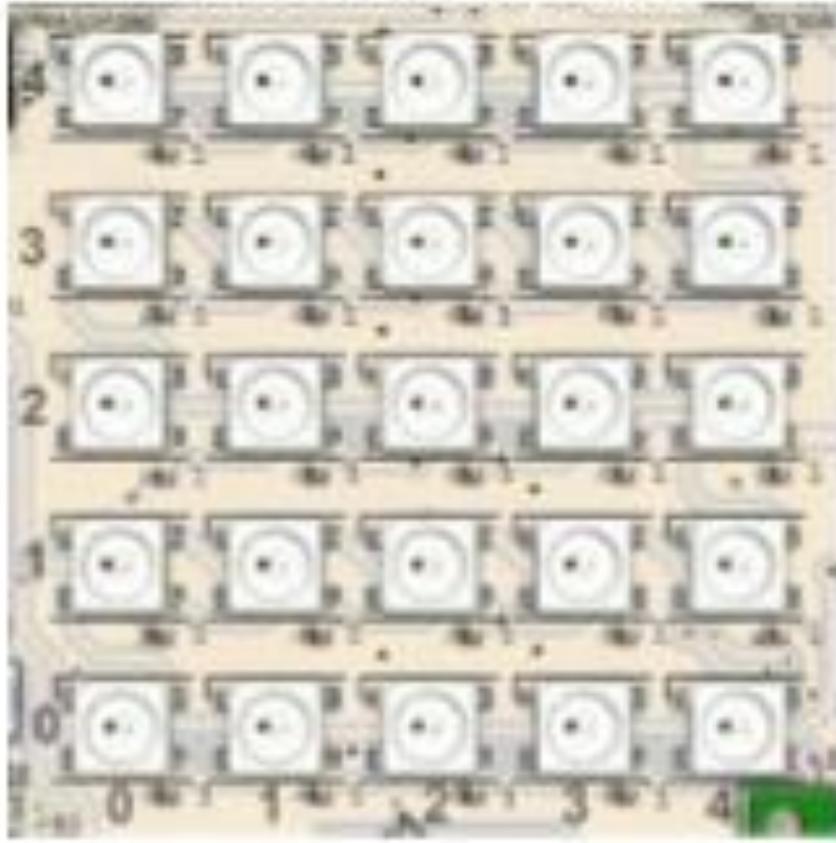
Esta variável é exibida no serial monitor para verificar o efeito Bounce.

Obs. Rodar no Wokwi também.

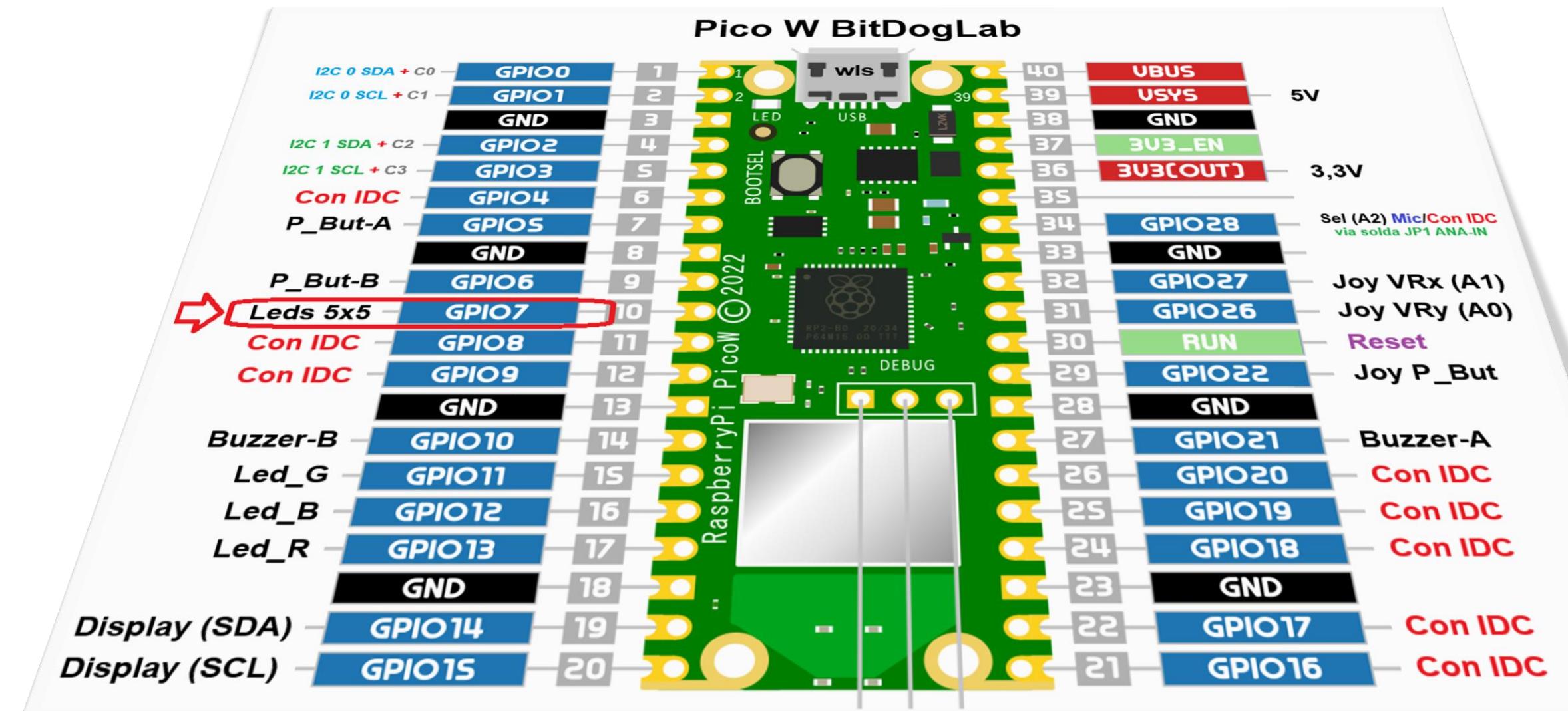
```
c Interrupt.c > main()
10 {
11     stdio_init_all();
12     gpio_init(ledA_pin);           // Inicializa o pino do LED
13     gpio_set_dir(ledA_pin, GPIO_OUT); // Configura o pino como saída
14     gpio_init(ledB_pin);           // Inicializa o pino do LED
15     gpio_set_dir(ledB_pin, GPIO_OUT); // Configura o pino como saída
16     gpio_init(button_0);
17     gpio_set_dir(button_0, GPIO_IN); // Configura o pino como entrada
18     gpio_pull_up(button_0);        // Habilita o pull-up interno
19     gpio_set_irq_enabled_with_callback(button_0, GPIO_IRQ_EDGE_FALL, true, &gp
20     while (true)
21     {
22         gpio_put(ledB_pin, true);
23         sleep_ms(tempo);
24         gpio_put(ledB_pin, false);
25         sleep_ms(tempo);
26     }
27 }
28 void gpio_irq_handler(uint gpio, uint32_t events)
29 {
30     uint32_t current_time = to_us_since_boot(get_absolute_time());
31     printf("A = %d\n", a);
32     // Verifica se passou tempo suficiente desde o último evento
33     if (current_time - last_time > 200000) // 50 ms de debouncing
34     {
35         last_time = current_time; // Atualiza o tempo do último evento
36         printf("Mudanca de Estado do Led. A = %d\n", a);
37         gpio_put(ledA_pin, !gpio_get(ledA_pin)); // Alterna o estado
38         a++;                                // incrementa a variavel de v
39     }
40 }
```

Matriz de Leds BitDogLab

4) Matriz de LEDs coloridos (LED-RGB 5x5 5050 WS2812)



[24, 23, 22, 21, 20],
[15, 16, 17, 18, 19],
[14, 13, 12, 11, 10],
[05, 06, 07, 08, 09],
[04, 03, 02, 01, 00]

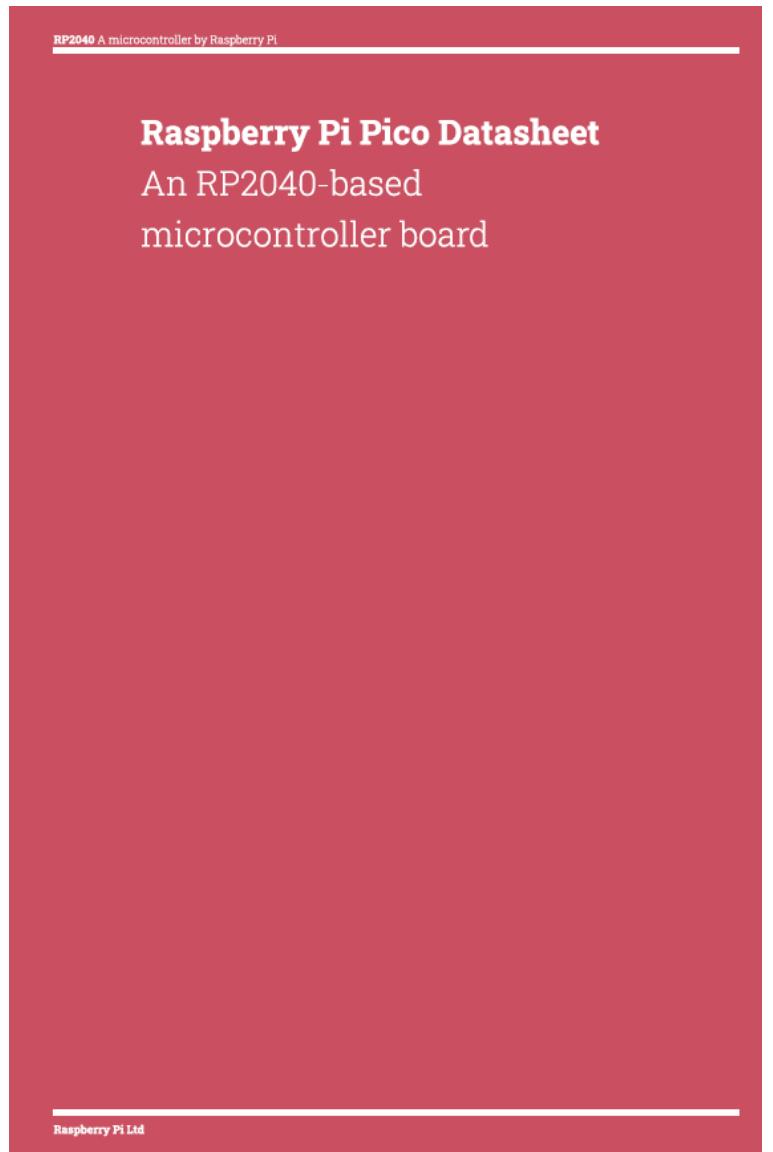
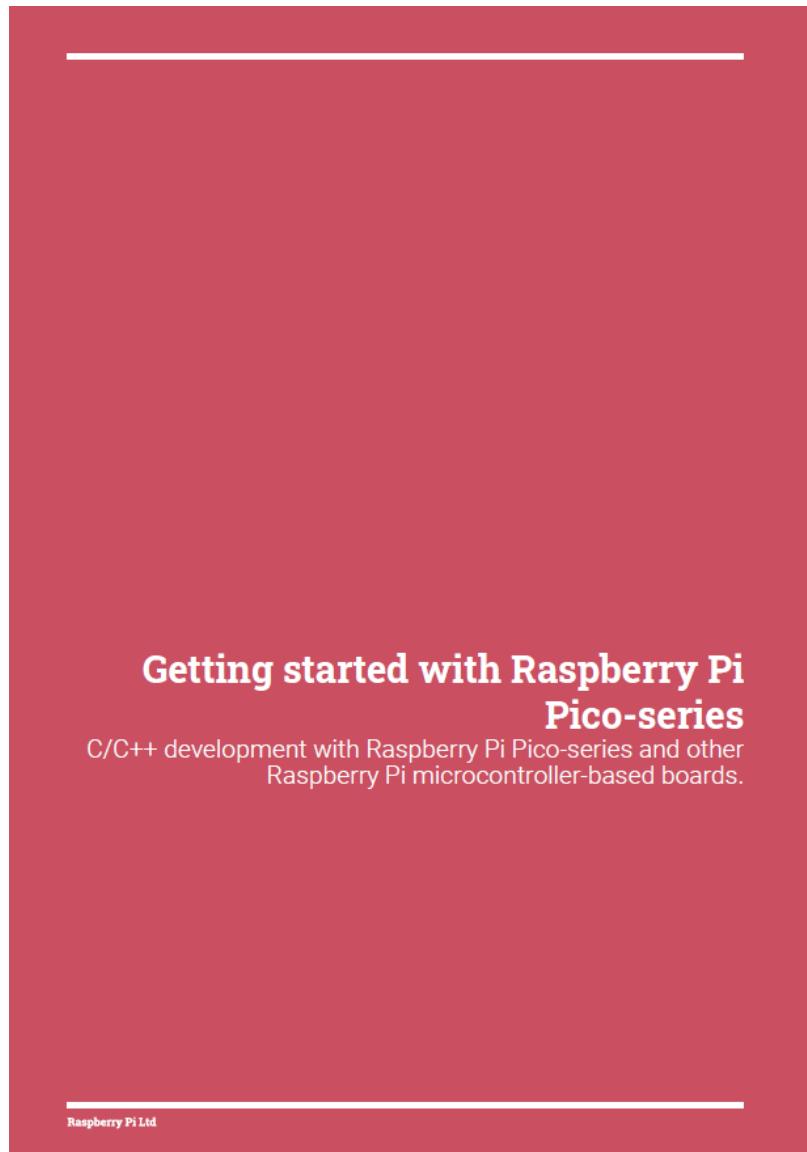


Interrupções

Finalizando

Interrupções

Referências



<https://github.com/BitDogLab/BitDogLab/tree/main/doc>

<https://www.raspberrypi.com/documentation/microcontrollers/pico-series.html>

Pontos Principais Abordados

- Revisão sobre componentes eletrônicos da BDL;
- Polling;
- Interrupções;
- Bouncing, consequências e sua mitigação;

Obrigado!

Executores:



Coordenação:



Iniciativa:

