

DevTitans

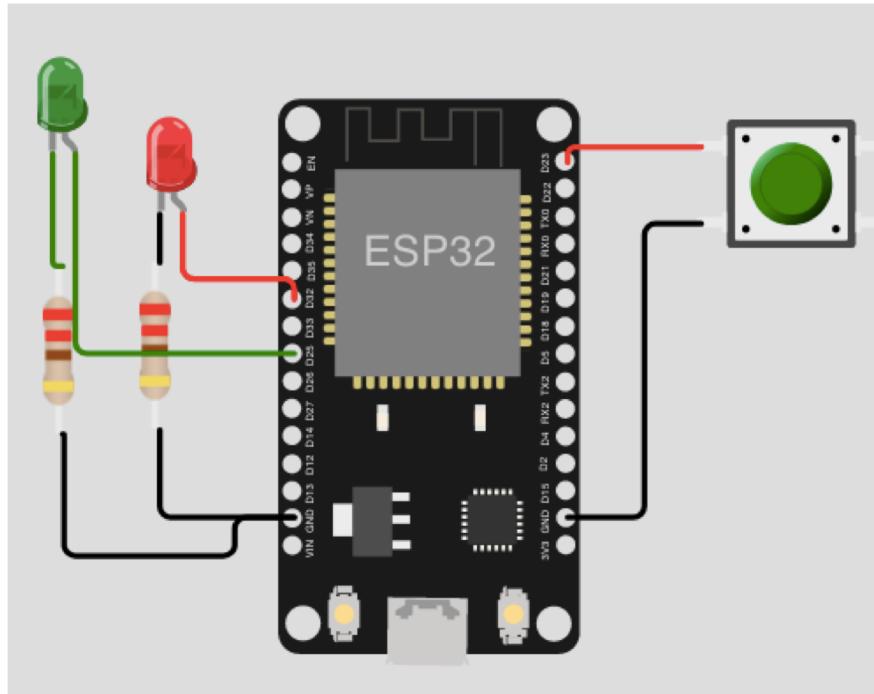
Interrupção (motivação, externa e timer)

Qual o problema com o `delay()`?

- A função `delay()` faz a execução do código parar por completo até que o tempo estabelecido seja executado
- Se o `delay()` for de 2 segundos então o código vai ficar “parado” por 2 segundos; somente após esse tempo o código volta a ser executado
- É o que em SO chamamos de “**espera ocupada**”
- Exemplo: o circuito a seguir tem um LED (verde) que alterna automaticamente entre ligado e desligado, e outro LED (vermelho) que alterna mediante o pressionar de um botão de pressão

Círcito

Considere o círcito abaixo



<https://wokwi.com/projects/334280987229815380>

Codificação

```
#define pinButton 23
#define pinLedVermelho 32
#define pinLedVerde 25

boolean pressLedVermelho = false;

void setup() {
    pinMode(pinLedVermelho, OUTPUT);
    pinMode(pinLedVerde, OUTPUT);
    pinMode(pinButton, INPUT_PULLUP);
}
```

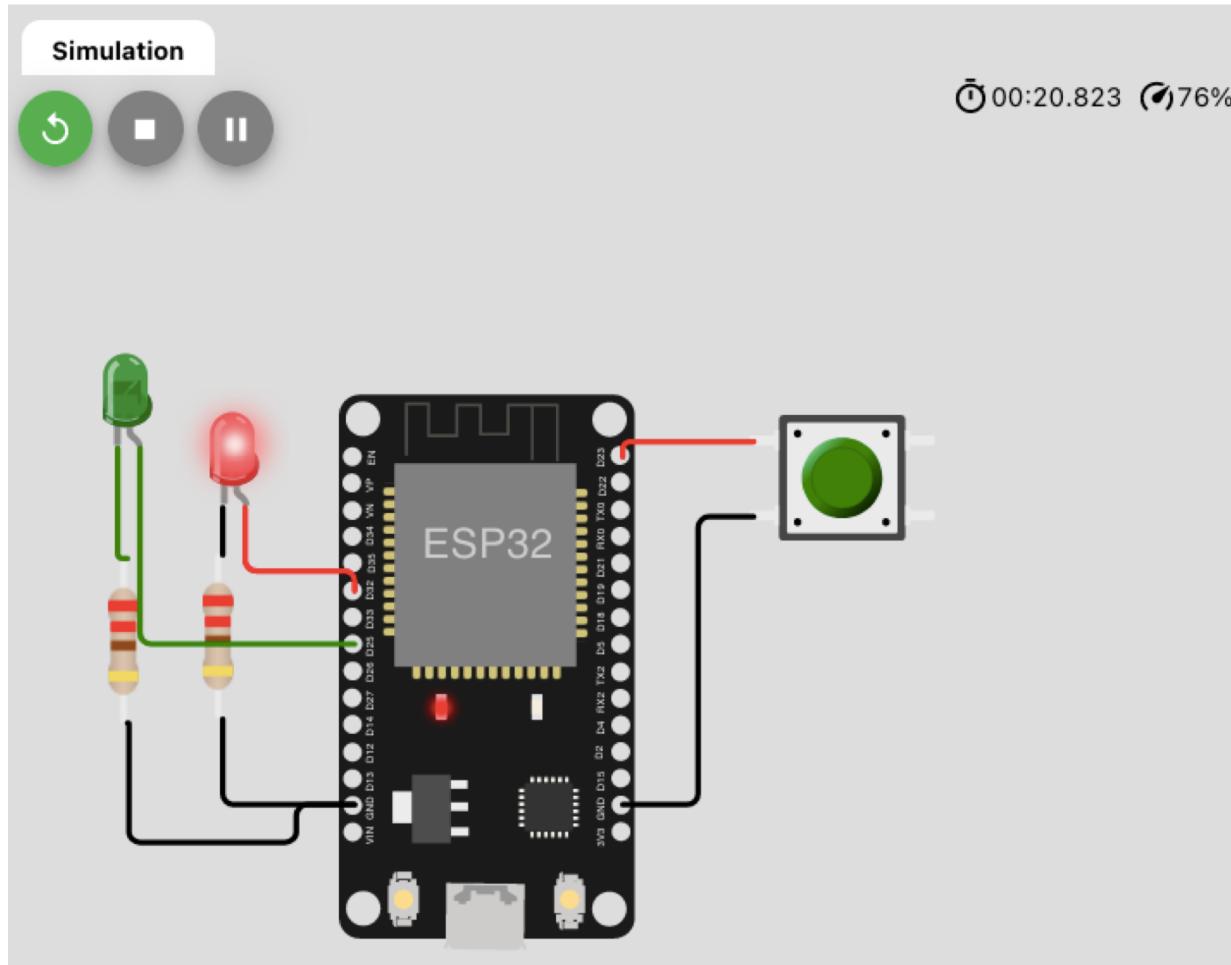
Codificação

```
void loop(){
    digitalWrite(pinLedVerde, HIGH) ;
    delay (200) ;
    digitalWrite(pinLedVerde, LOW) ;
    delay (200) ;

    if (digitalRead(pinButton))
        pressLedVermelho = true;

    if (digitalRead(pinButton)==LOW && pressLedVermelho) {
        digitalWrite(pinLedVermelho,!digitalRead(pinLedVermelho)) ;
        pressLedVermelho = false;
    }
}
```

Simulação



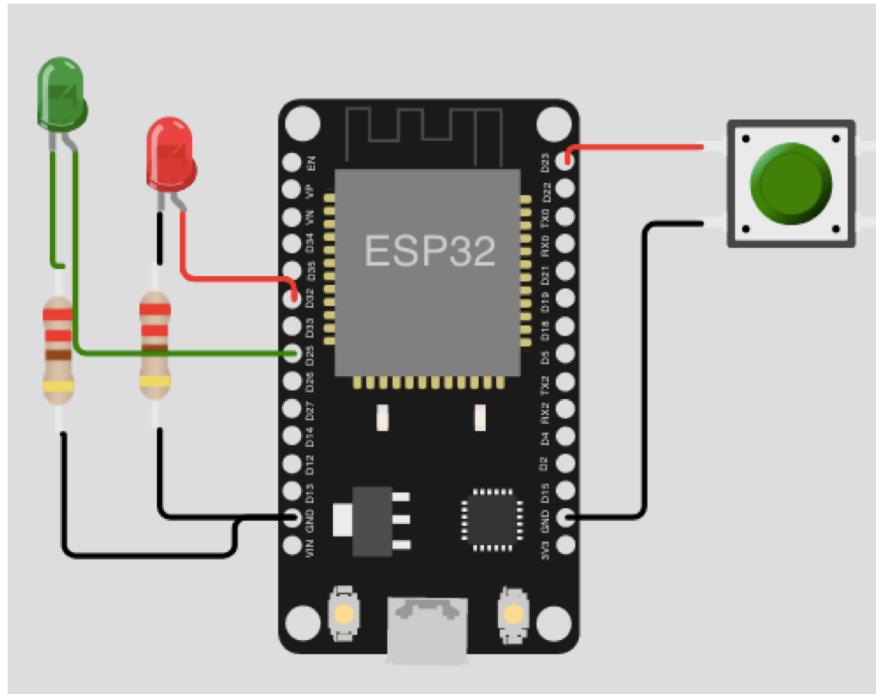
Apertar o botão
nem sempre será
atendido,
exatamente por
causa do delay()

Qual o problema com o millis?

- Tem outra alternativa ao uso de **delay()**?
- Sim, a função **millis()** funciona melhor que o **delay()**
- Entretanto, a função **millis()**, ainda assim, requer a verificação contínua se a condição é verdadeira
- E mais, dependendo do caso, esta solução pode não ser tão precisa e existem situações em que a precisão seja um requisito importante
- Consideremos o mesmo exemplo anterior, trocando o **delay()** por **millis()**

Círcito

O círcito é o mesmo



Anodo (+) do LED Verm (Pino D32)
Catodo (-) do LED (Term. Resistor1)
Anodo (+) do LED Verde (Pino D25)
Catodo (-) do LED (Term. Resistor2)
Term. Resistor1 (Pino GND2)
Term. Resistor2 (Pino GND2)
Contato1 (D23)
Contato2 (GND1)

<https://wokwi.com/projects/334315971241050708>

Codificação

```
#define pinButton 23
#define pinLedVermelho 32
#define pinLedVerde 25

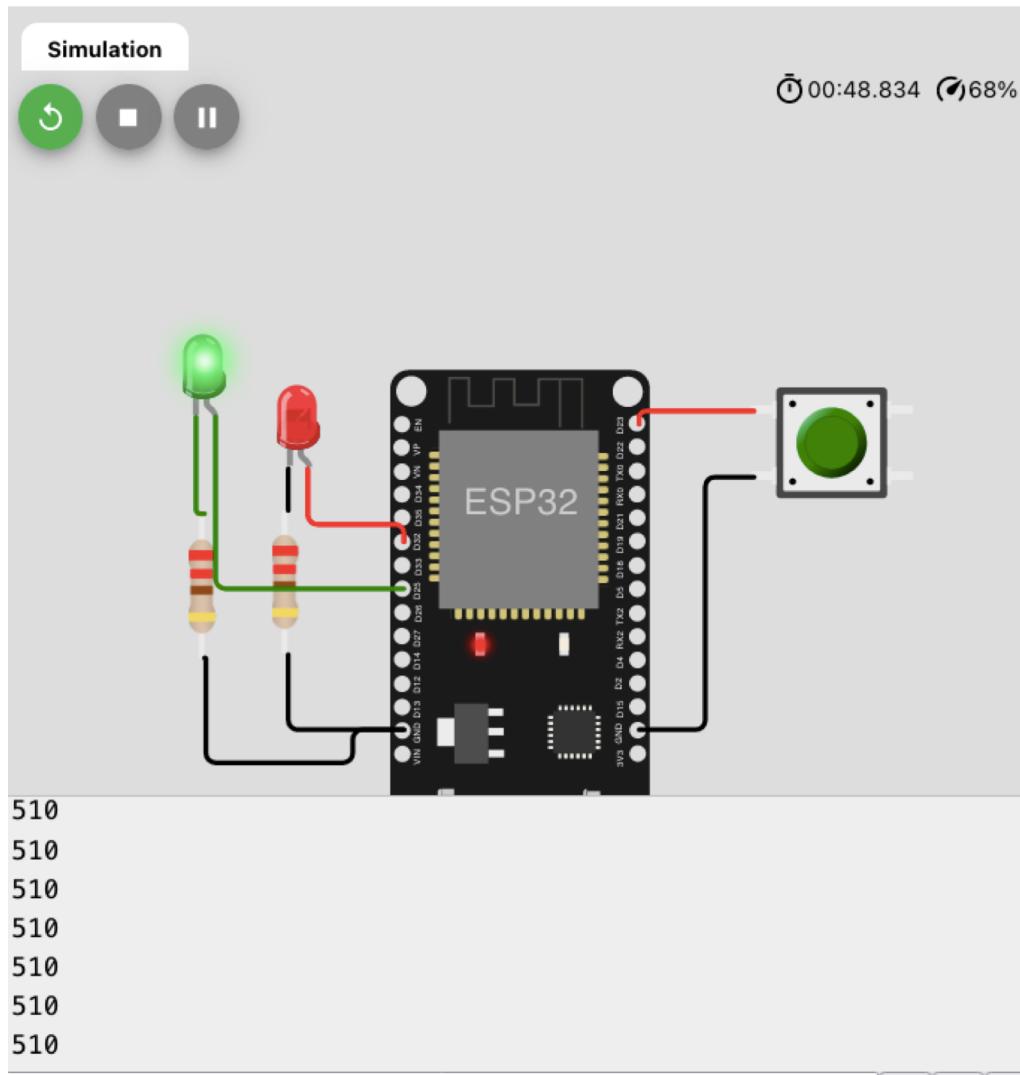
boolean pressLedVermelho = false;
unsigned long IntervalLedVerde = 500;
unsigned long previousMillisVerde = 0;

void setup() {
    Serial.begin(9600);
    pinMode(pinLedVermelho, OUTPUT);
    pinMode(pinLedVerde, OUTPUT);
    pinMode(pinButton, INPUT_PULLUP);
}
```

Codificação

```
void loop() {
    unsigned long currentMillisVerde = millis();
    if (currentMillisVerde - previousMillisVerde
        > IntervalLedVerde) {
        Serial.println(currentMillisVerde - previousMillisVerde);
        previousMillisVerde = currentMillisVerde;
        digitalWrite(pinLedVerde, !digitalRead(pinLedVerde));
    }
    delay(15); // simulando uma instrução
    if (digitalRead(pinButton)) pressLedVermelho = true;
    if (digitalRead(pinButton)==LOW && pressLedVermelho) {
        digitalWrite(pinLedVermelho, !digitalRead(pinLedVermelho));
        pressLedVermelho = false;
    }
    delay(15); // simulando outra instrução
}
```

Simulação



A frequência de 500ms
(LED Verde) não é
atingida

Se o **delay()**, que
representa instruções
entre as duas
verificações, for maior, a
precisão ficará cada vez
mais distante

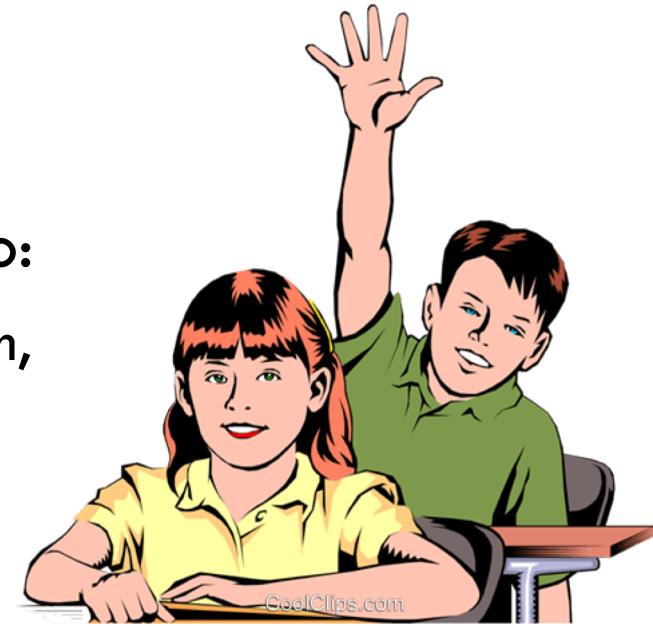
Tem solução?

- Resposta: **Interrupções**

Interrupção Externa

Por que utilizar leitura de GPIOs via interrupção?

- Imagine a seguinte situação: você é um professor que deseja saber se algum aluno tem alguma dúvida sobre o conteúdo
- Você tem duas alternativas para isso:
 1. Perguntar a todos os alunos, um a um, quem tem dúvida; ou
 2. Pergunta a todos de uma vez se há alguma dúvida e, caso alguém levante a mão, você pergunta a essa pessoa qual a dúvida e a esclarece



Por que utilizar leitura de GPIOs via interrupção?

- Na maioria dos casos, a forma 2 é a mais eficiente, pois leva muito menos tempo e é algo mais focado (*interação somente com quem tem dúvida*)
- Nos sistemas embarcados, esse dilema também existe se trocarmos o **professor** pela CPU (processador) e os **alunos** por GPIOs
- A forma 1 (menos eficiente) se chama leitura **polling**
- A forma 2 (mais eficiente) se chama leitura via **interrupção**

Interrupção Externa

- Uma **interrupção externa** é um evento que faz com que o processador **pare** a execução do programa e **desvie** a execução para um bloco de código chamado **rotina de interrupção (Interrupt Service Routine)**
- Ao terminar o tratamento de interrupção o controle **retorna ao programa interrompido**, exatamente na mesma instrução em que estava quando ocorreu a interrupção

Interrupção

- Vantagem: pode **melhorar a performance**, pois você realiza certas operações sem estar constantemente verificando se algo aconteceu (**polling**)
- O **delay()** nem sempre é o ideal porque **prende** o processador (não faz **nada** - “**espera ocupada**”)
- A interrupção tem sempre **dois** pontos chaves:
 - **Condição de interrupção**, que é a condição que vai indicar o início da interrupção
 - **Função a ser executada**, código que faz o tratamento da interrupção

Como configurar uma interrupção externa

- A função **attachInterrupt(pin, ISR, mode)** configura a interrupção para determinado pino, qual a função que responderá por ela (ISR), e qual “modo” que acionará a interrupção (*veremos em seguida*)
- A função **detachInterrupt(pin)** desativa a interrupção do pino e não tem retorno nenhum

Interrupt Service Routine (ISR)

- Os ISRs são tipos especiais de funções que possuem algumas limitações exclusivas:
 - Um ISR não pode ter nenhum parâmetro e não deve retornar nada
 - Um ISR deve ser o mais **curto e rápido** possível
 - Se seu programa utiliza múltiplos ISRs, apenas um pode ser executado por vez
 - **delay()** e **millis()** utilizam interrupções para contar, portanto, não incrementará dentro de um ISR

Interrupção

- **Variáveis globais (flags)** devem ser usadas para passar dados entre uma ISR e o programa principal
- Para garantir que as variáveis compartilhadas são **atualizadas corretamente**, declare-as usando o modificador **volatile**
- A diretiva **volatile** faz com que o compilador carregue a variável **da RAM** e não de um **Registrador** (pode ser impreciso devido às rotinas de serviço de interrupção)

Modo

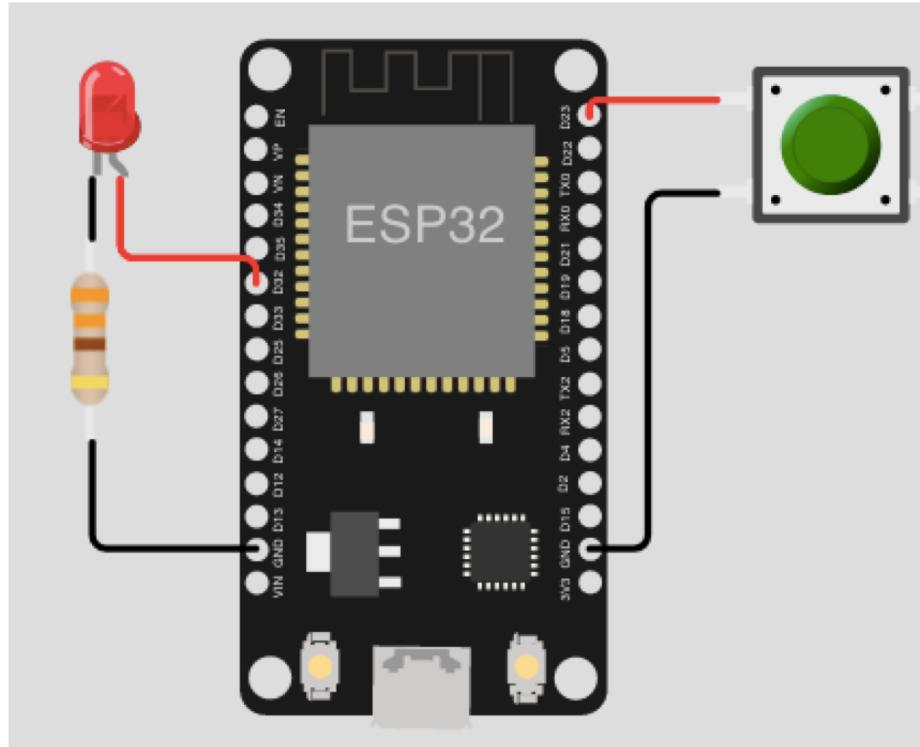
- Há diversos modos (**condições**) que definem quando a interrupção deve ser ativada
 - ▣ **LOW** : aciona a interrupção sempre que o pino estiver baixo
 - ▣ **HIGH** : aciona a interrupção sempre que o pino estiver alto
 - ▣ **CHANGE**: aciona a interrupção sempre que o pino muda de estado
 - ▣ **RISING**: aciona a interrupção quando o pino vai de baixo para alto (**LOW > HIGH**)
 - ▣ **FALLING**: para acionar a interrupção quando o pino vai de alto para baixo (**HIGH > LOW**)

Interrupção Externa 1

Ligar / Desligar um LED através do
pressionamento de um botão (interrupção)

Círculo

Construa o círculo abaixo



Anodo (+) do LED (Pino D32)

Catodo (-) do LED (Term. Resistor)

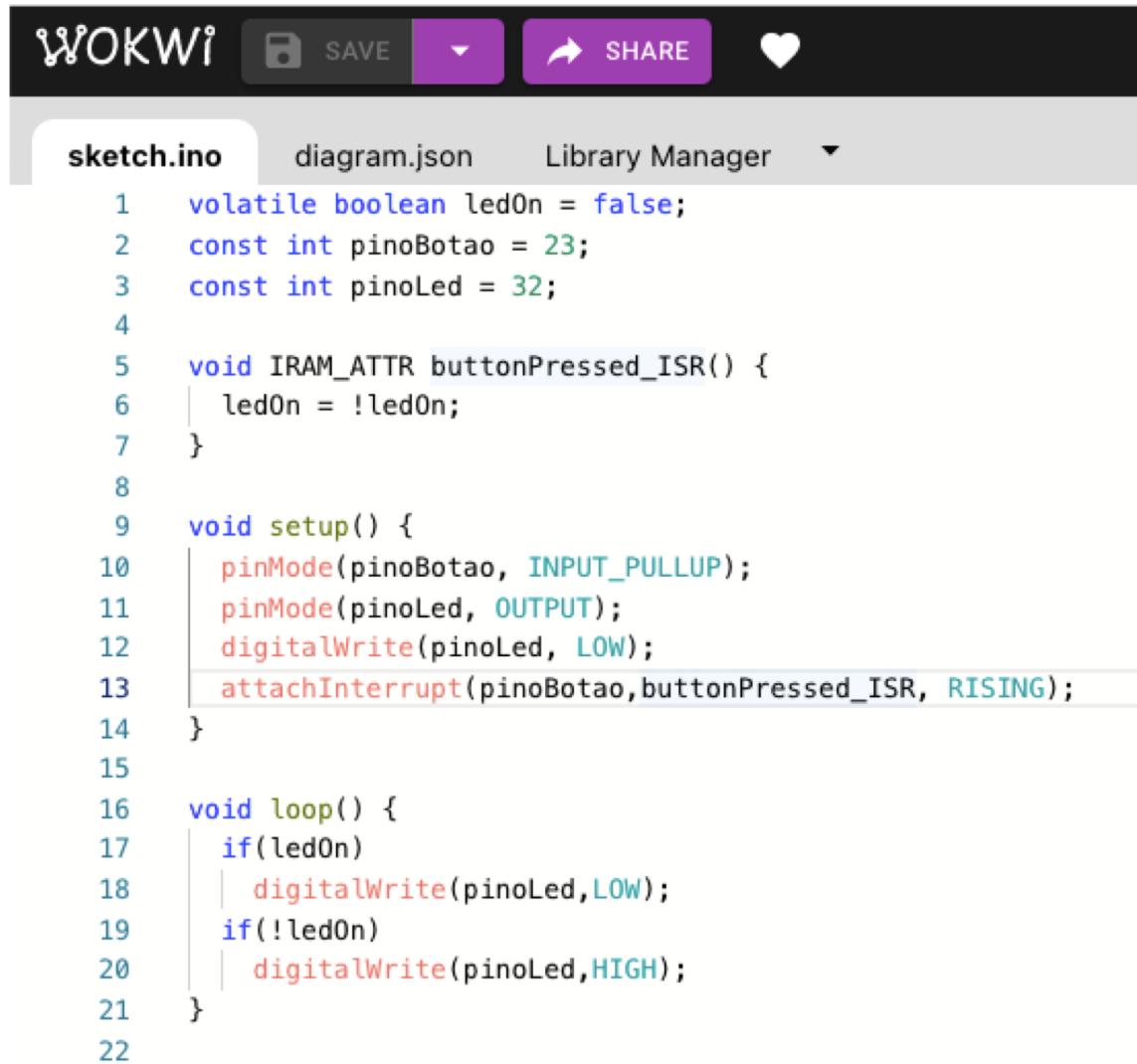
Term. Resistor (Pino GND2)

Contato1 (D23)

Contato2 (GND1)

<https://wokwi.com/projects/334202666333766226>

Codificação



The image shows the WOKWi IDE interface. At the top, there are buttons for 'SAVE' and 'SHARE'. Below the header, there are tabs for 'sketch.ino', 'diagram.json', and 'Library Manager'. The main area contains the following Arduino sketch code:

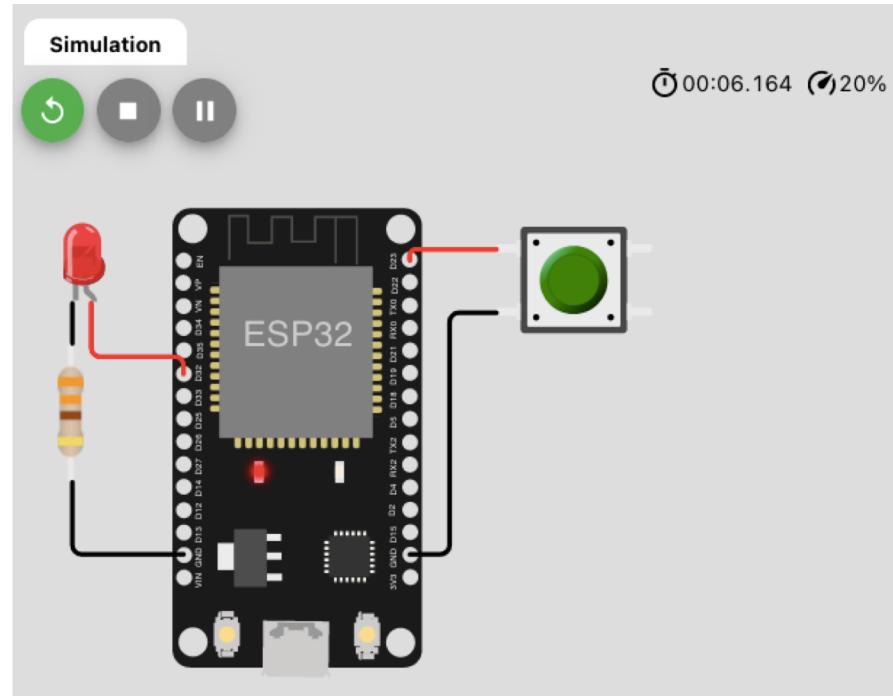
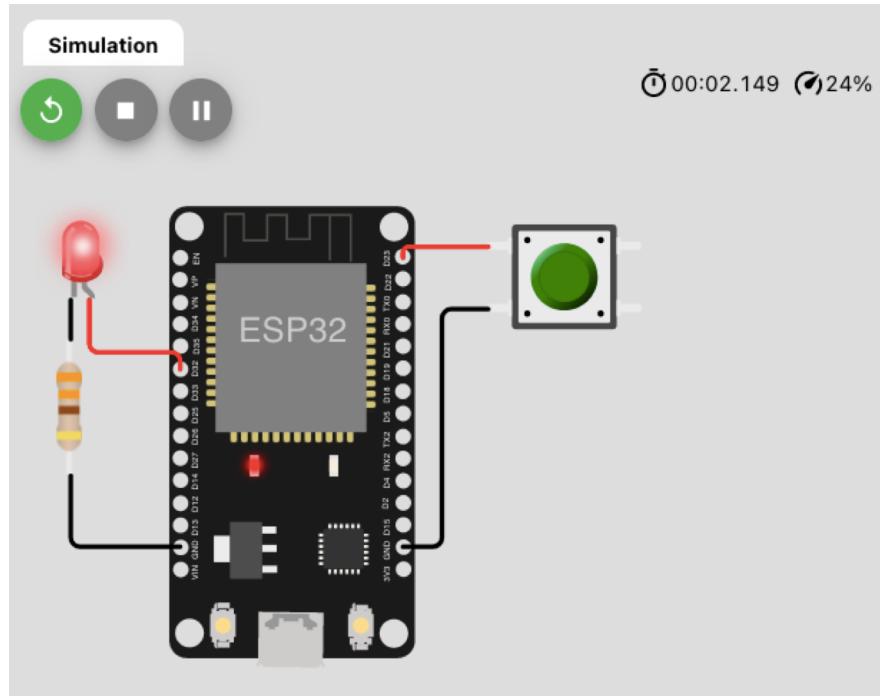
```
1 volatile boolean ledOn = false;
2 const int pinoBotao = 23;
3 const int pinoLed = 32;
4
5 void IRAM_ATTR buttonPressed_ISR() {
6     ledOn = !ledOn;
7 }
8
9 void setup() {
10    pinMode(pinoBotao, INPUT_PULLUP);
11    pinMode(pinoLed, OUTPUT);
12    digitalWrite(pinoLed, LOW);
13    attachInterrupt(pinoBotao,buttonPressed_ISR, RISING);
14 }
15
16 void loop() {
17     if(ledOn)
18         digitalWrite(pinoLed,LOW);
19     if(!ledOn)
20         digitalWrite(pinoLed,HIGH);
21 }
22
```

Ao definir o **ISR** como **IRAM_ATTR** indica que o trecho de código ficará na seção de instruções da **RAM** e não na Flash, ganhando uma maior **velocidade**

Codificação

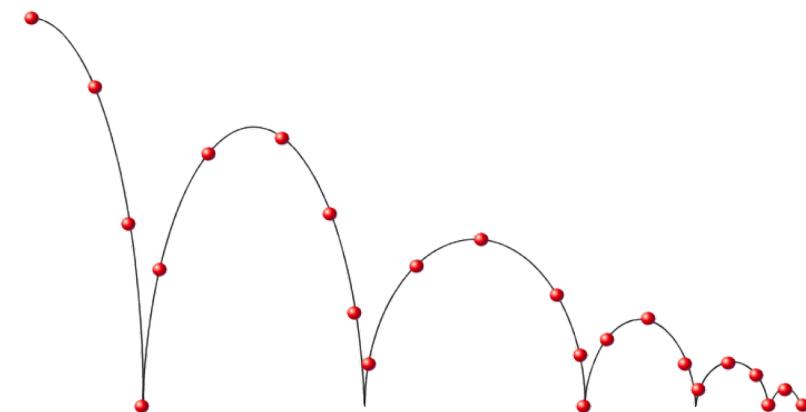
```
volatile boolean ledOn = false;
const int pinoBotao = 23;
const int pinoLed = 32;
void IRAM_ATTR buttonPressed_ISR() {
    ledOn = !ledOn;
}
void setup() {
    pinMode(pinoBotao, INPUT_PULLUP);
    pinMode(pinoLed, OUTPUT);
    digitalWrite(pinoLed, LOW);
    attachInterrupt(pinoBotao, buttonPressed_ISR, RISING);
}
void loop() {
    if(ledOn) digitalWrite(pinoLed, LOW);
    if(!ledOn) digitalWrite(pinoLed, HIGH);
}
```

Simulação



Problema

- Todas as chaves mecânicas possuem um comportamento chamado “**bouncing**”, que é quando o microcontrolador interpreta múltiplos toques quando na verdade o botão só foi pressionado uma única vez. Isto é causado pela **mola** que existe no interior dos botões. Esse efeito é visto como se o **botão estivesse sendo ligado e desligado várias vezes**

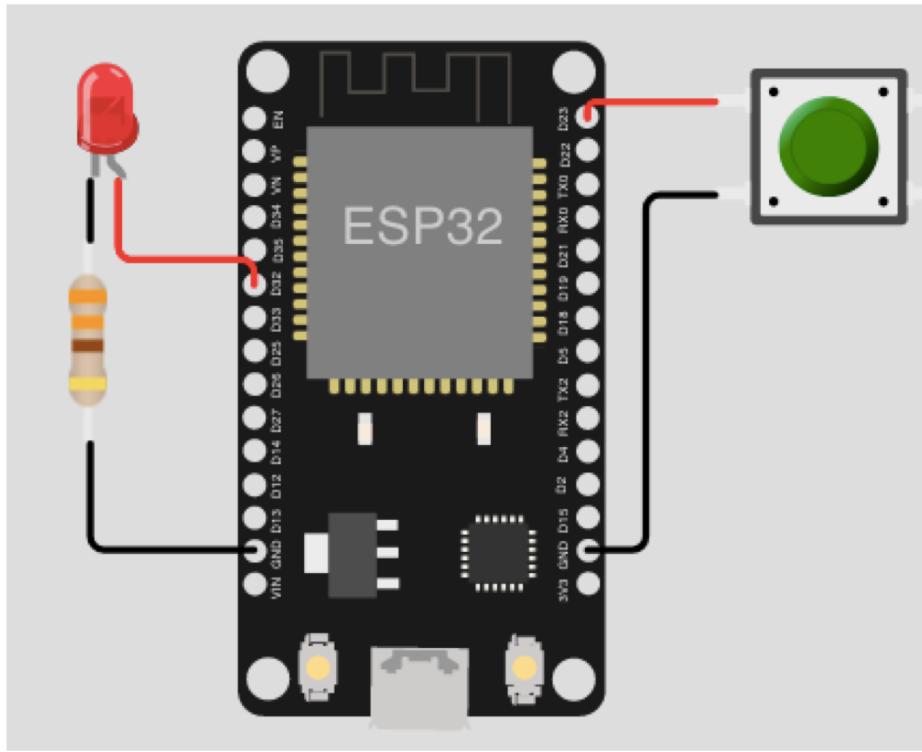


Interrupção Externa (*debouncing*)

Ligar / Desligar um LED através do
pressionamento de um botão (interrupção)

Círcito

O círcito é o mesmo



Anodo (+) do LED (Pino D32)
Catodo (-) do LED (Term. Resistor)
Term. Resistor (Pino GND2)
Contato1 (D23)
Contato2 (GND1)

<https://wokwi.com/projects/334203305466004051>

Codificação

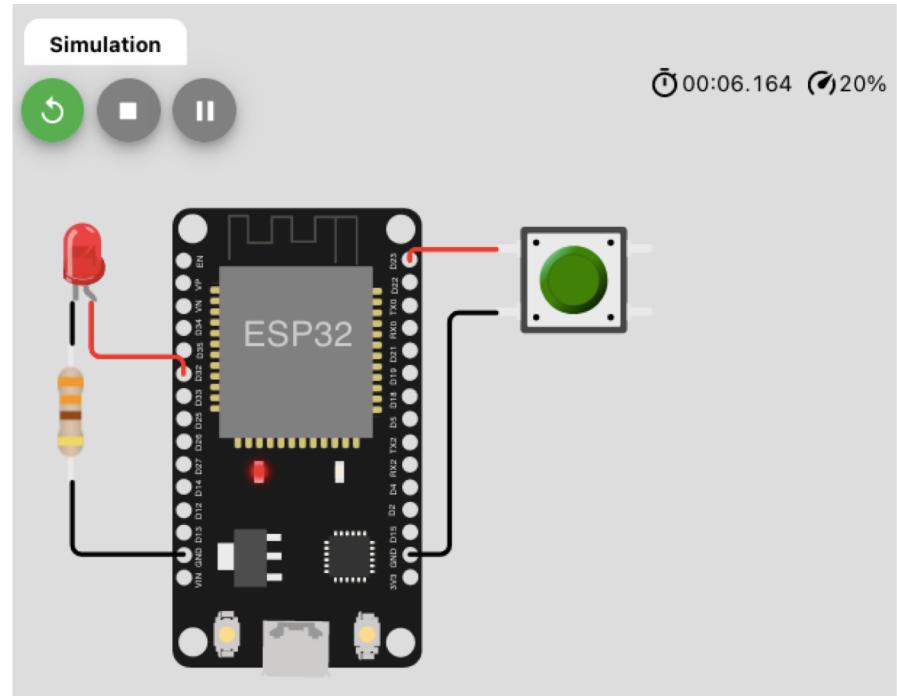
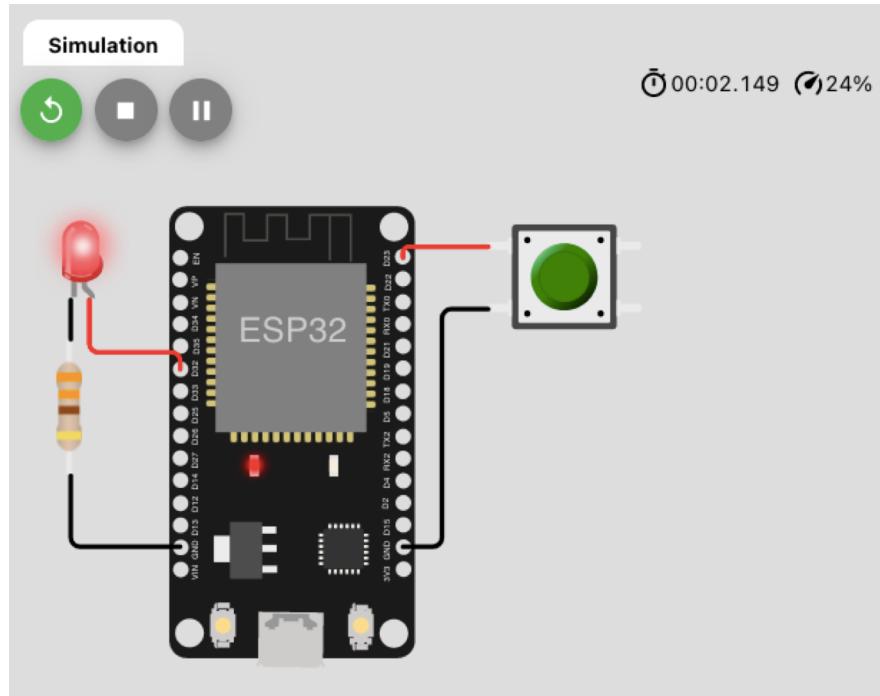
```
volatile boolean ledOn = false;
const int pinoBotao = 23;
const int pinoLed = 32;
const int TEMPO_DEBOUNCE = 100;
unsigned long ultimo_acionamento = 0;

void IRAM_ATTR buttonPressed_ISR() {
    if ( (xTaskGetTickCount() - ultimo_acionamento)
        >= TEMPO_DEBOUNCE ) {
        ledOn = !ledOn;
        ultimo_acionamento = xTaskGetTickCount();
    }
}
```

Codificação

```
void setup() {  
    pinMode(pinoBotao, INPUT_PULLUP);  
    pinMode(pinoLed, OUTPUT);  
    digitalWrite(pinoLed, LOW);  
    attachInterrupt(pinoBotao, buttonPressed_ISR, RISING);  
}  
  
void loop() {  
    if(ledOn)  digitalWrite(pinoLed,LOW);  
    if(!ledOn) digitalWrite(pinoLed,HIGH);  
}
```

Simulação



Agora funciona bem!

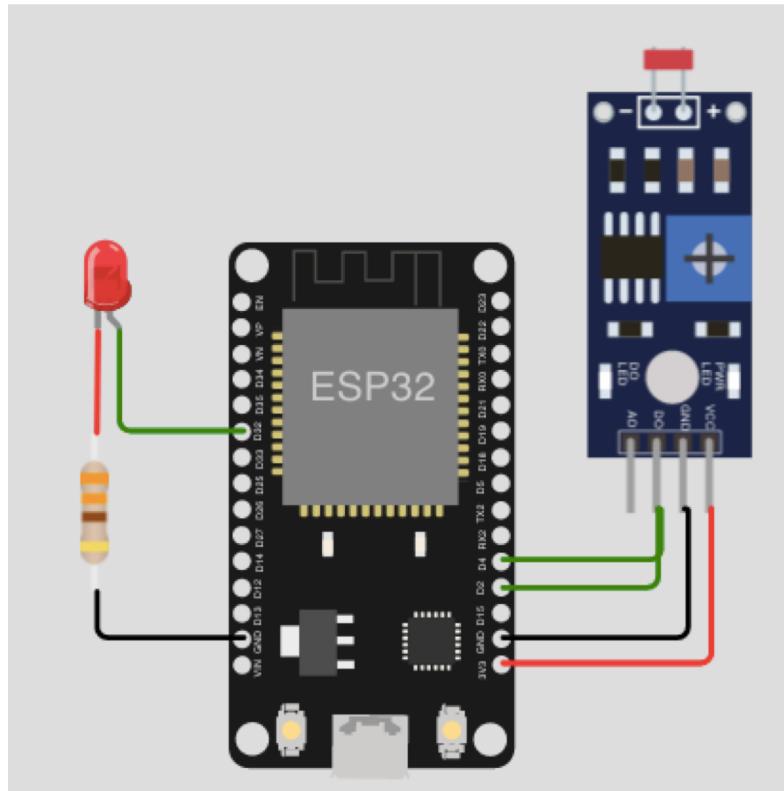
Interrupção 2

O LED é ligado caso a luminosidade esteja baixa

O LED é desligado caso a luminosidade esteja alta

O tratamento de ligar/desligar o LED é feito por interrupção

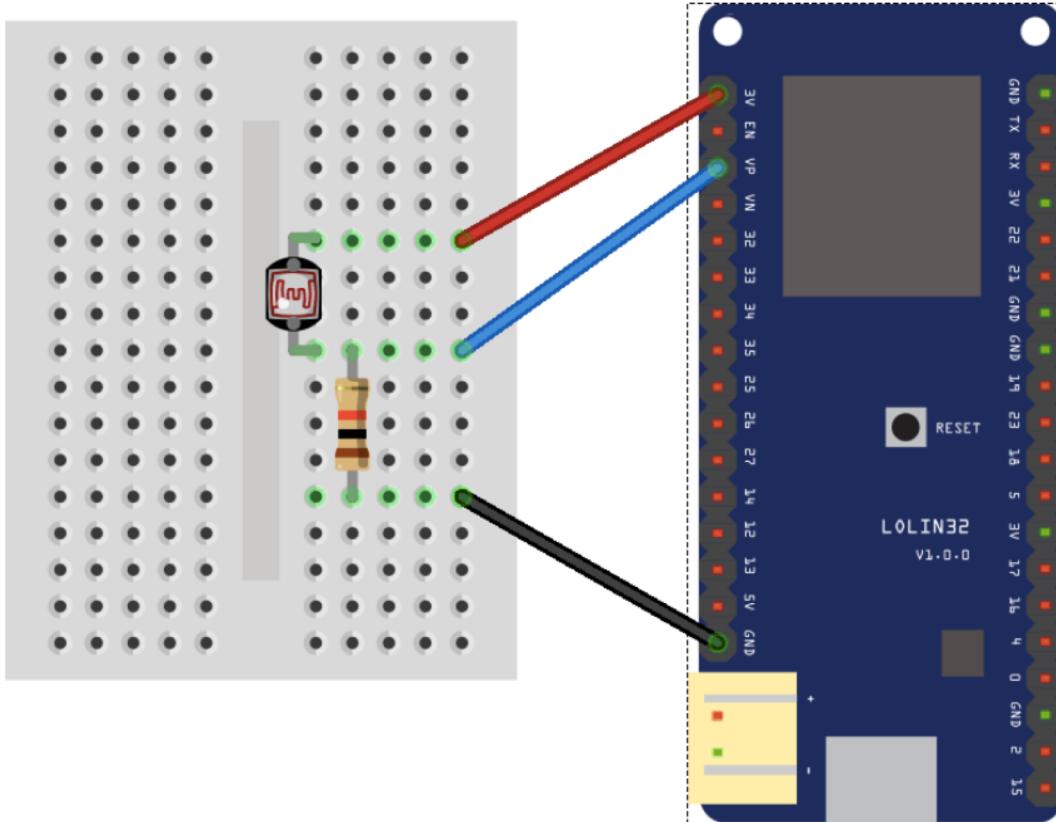
Circuito



Anodo (+) do LED Verm (Pino D32)
Catodo (-) do LED (Term. Resistor)
Term. Resistor (Pino GND2)
DO do LDR (D2 e D4)
GND do LDR (GND1)
VCC do LDR (3V3)

<https://wokwi.com/projects/334204435862389331>

Círcito



Se você tiver somente o sensor (não um módulo), e para garantir maior estabilidade do sinal, é importante incluir um resistor pull-down (ou pull-up)

Codificação

```
#define LED 32
#define pinoLDRLow 2
#define pinoLDRHigh 4

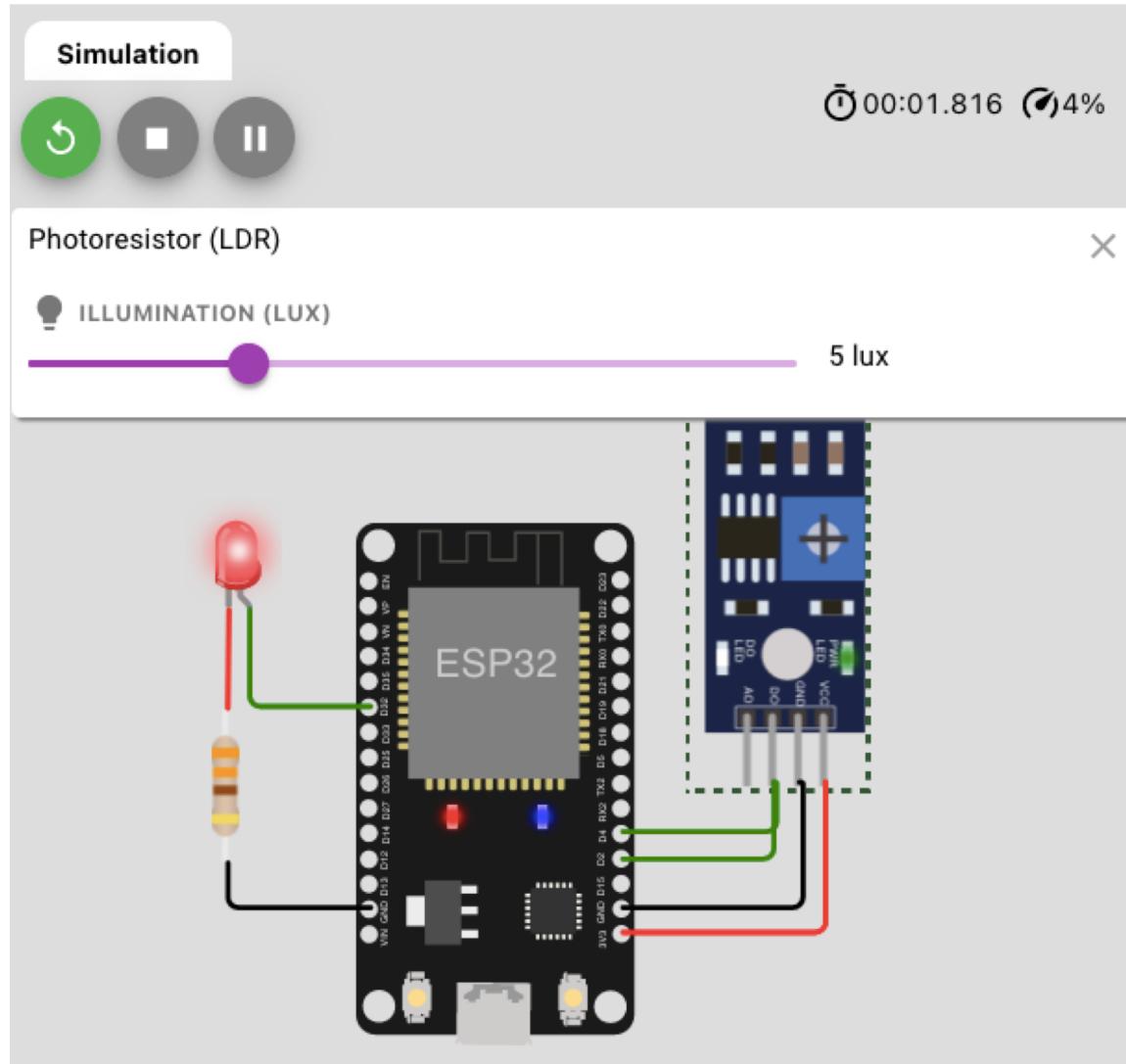
void IRAM_ATTR turnLEDON_ISR()
{
    digitalWrite(LED,HIGH);
}

void IRAM_ATTR turnLEDOFF_ISR()
{
    digitalWrite(LED,LOW);
}
```

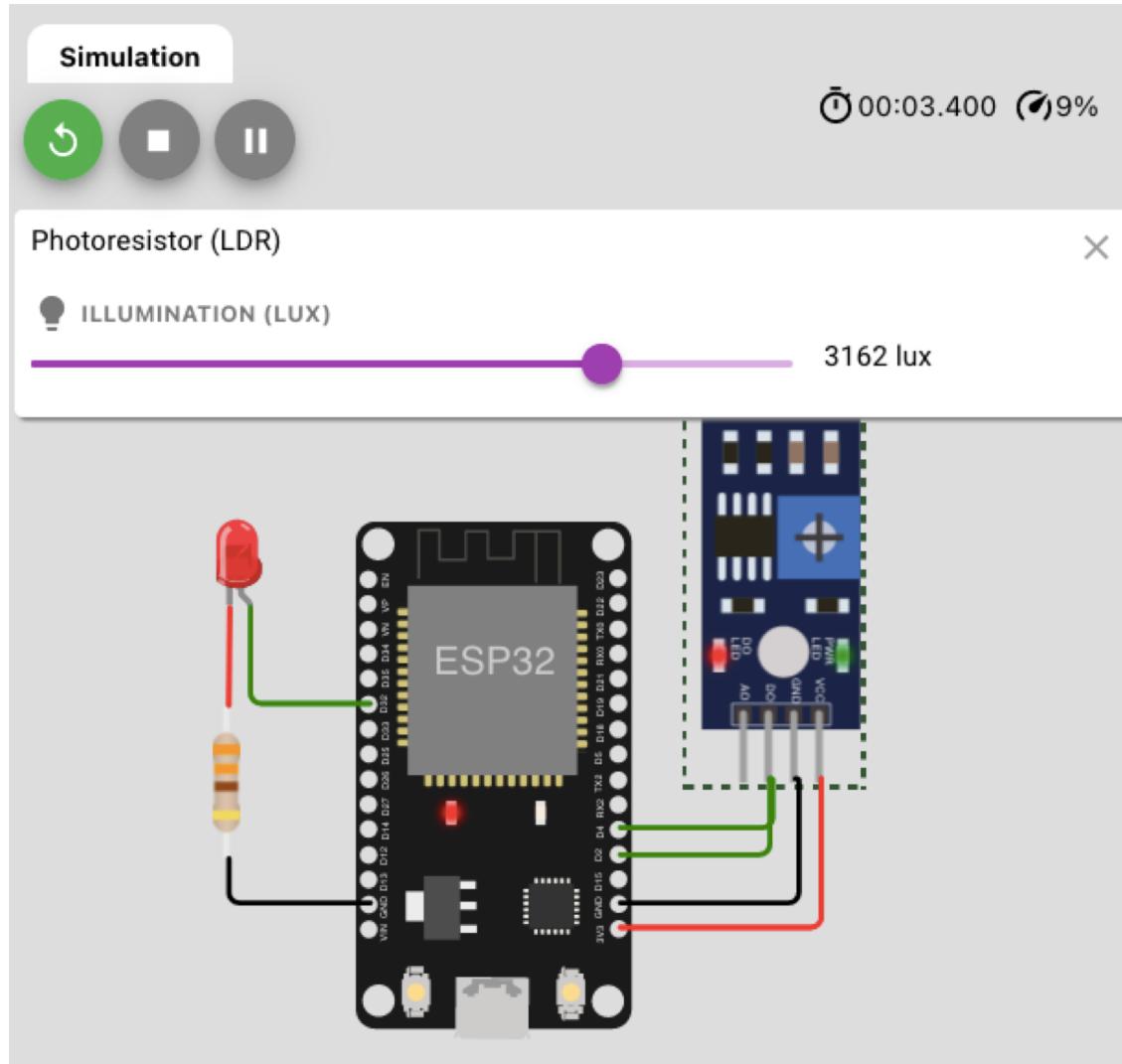
Codificação

```
void setup() {  
    pinMode(LED,OUTPUT);  
    pinMode(pinLDRLow,INPUT);  
    pinMode(pinLDRHigh,INPUT);  
  
    attachInterrupt(pinLDRLow, turnLEDon_ISR, RISING);  
    attachInterrupt(pinLDRHigh, turnLEDOff_ISR, FALLING);  
}  
  
void loop() {  
}
```

Simulação



Simulação



Interrupção do Timer

Introdução

- Um **temporizador** é essencialmente um **contador**
- Por exemplo, ao definir a frequência de contagem em **1 Hz**, você pode medir com precisão **1 segundo**
- Se definir a frequência em **100 Hz**, você pode medir com precisão **0,1s** ou **100ms**
- Se você quiser medir **0,5s** (500 ms), mas a frequência de **2 Hz** não está disponível, a ideia é contar 5 vezes a frequência de **100Hz** (que é igual a **5*100ms**)

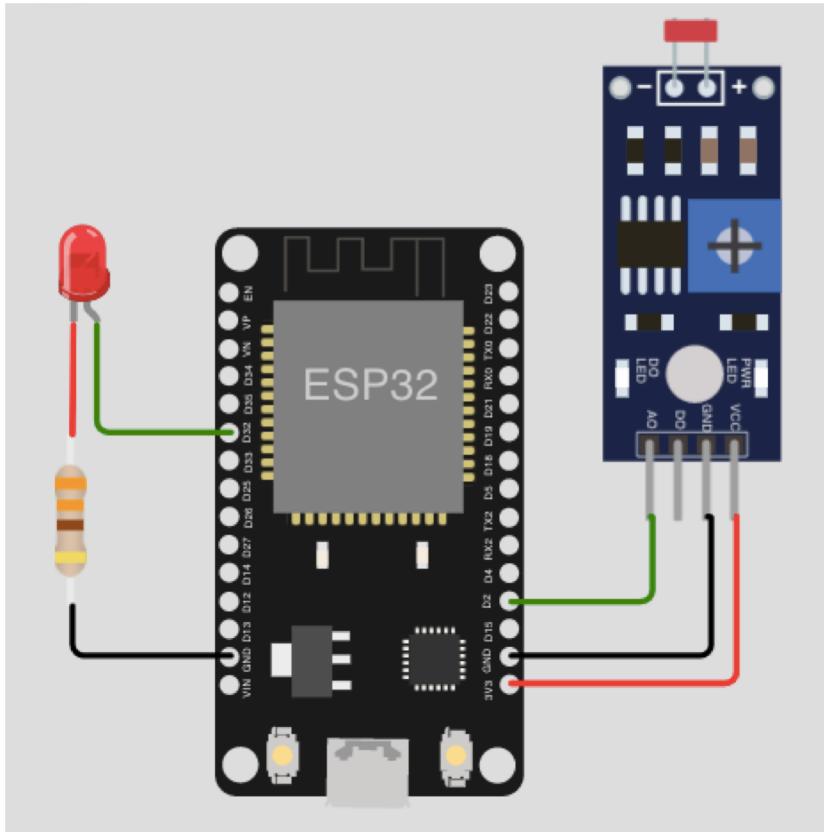
Introdução

- Os temporizadores funcionam exatamente dessa forma
- Em vez de 0 a 10, eles contam de 0 a $(2^n - 1)$, sendo n o número de bits do contador
- Assim, um contador de **8** bits contará de **0 a 255**, um contador de **16** bits contará de **0 a 65535**, um contador de **32** bits contará de **0 a 4294967295**, e assim por diante
- O ESP32 tem **quatro temporizadores de 64 bits**

Interrupção Timer LDR & LED

Ler o valor de um LDR via interrupção do timer a cada 1 segundo e alternar um LED

Circuito



Anodo (+) do LED (Pino D32)
Catodo (-) do LED (Term. Resistor)
Term. Resistor (Pino GND)

AO do LDR (D2 do ESP32)
GND do LDR (GND1 do ESP32)
VCC do LDR (3V3 do ESP32)

<https://wokwi.com/projects/335209941058978386>

Codificação

```
#define LDR_Pin 2
#define prescaler 40
#define intervalo 2000000 // 2s

hw_timer_t * timer = NULL;
volatile bool interrupted = false;
bool stateLED = true;
```

Codificação

```
void IRAM_ATTR onTimer() {  
    interrupted = true;  
}  
  
void setup() {  
    pinMode(LED, OUTPUT);  
    digitalWrite(LED, stateLED);  
    Serial.begin(115200);  
    timer = timerBegin(0, prescaler, true);  
    timerAttachInterrupt(timer, &onTimer, true);  
    timerAlarmWrite(timer, intervalo, true);  
    timerAlarmEnable(timer);  
}
```

Codificação

```
void loop() {  
    if(interrupted) {  
        Serial.print("LDR: ");  
        Serial.println(analogRead(LDR_Pin));  
        stateLED = !stateLED;  
        digitalWrite(LED, stateLED);  
        interrupted = false;  
    }  
}
```

Passo a passo do código

Constantes

```
#define LED 32
#define LDR_Pin 2
#define prescaler 40
#define intervalo 2000000 // 2s
```

LED é o pino onde está ligado o LED; **LDR_Pin** é o pino onde está ligado o LDR; assumimos um **prescaler** (divisor do clock) igual a 40; A frequência de contagem é **1MHz**; o **quanto_contar** é quantidade de vezes que a frequência de contagem deve ser incrementada para atingir o tempo esperado. Por exemplo, se esse valor for **1.000.000**, multiplicado pela frequência, vai dar 1s.

Variáveis globais

```
hw_timer_t * timer = NULL;
```

Para configurar o timer, precisaremos de um ponteiro para uma variável do tipo `hw_timer_t`, que usaremos posteriormente na função de `setup`

Variáveis globais

```
volatile bool interrupted = false;
```

A variável “**interrupted**” será usada pela rotina de serviço de interrupção (ISR) para **sinalizar** que ocorreu uma interrupção

Como as ISRs não devem executar operações longas, a ISR **apenas sinaliza e adia** o tratamento para o loop principal

Como esta variável será **compartilhada**, ela precisa ser declarada com a palavra-chave “**volatile**”, o que evita que seja removida devido a otimizações do compilador

Rotina de serviço de interrupção (ISR)

```
void IRAM_ATTR onTimer() {  
    interrupted = true;  
}
```

A função **onTimer** será chamada na interrupção

Não fazemos muito nessa função, apenas definimos o valor da variável booleana como true

Como **onTimer** é uma função de interrupção, ela **não** pode receber nenhum argumento **nem** retornar nada

A parte '**IRAM_ATTR**' diz que esse código é colocado na RAM interna (a RAM é muito mais rápida que o Flash)

A função setup

```
pinMode(LED, OUTPUT) ;  
digitalWrite(LED, stateLED) ;  
Serial.begin(115200) ;
```

Iniciamos a função `setup` dizendo que o pino do LED é de saída, e já ativando o LED e, em seguida, abrindo uma conexão serial

A função setup

```
timer = timerBegin(0, prescaler, true);
```

A função **timerBegin** inicializa o timer. A variável **timer** do tipo **hw_timer_t** é agora atribuída um valor usando **timerBegin**

Esta função recebe o número do timer que queremos usar (de 0 a 3, pois temos 4 timers de hardware), o valor do **prescaler** (divisor de frequência) e um **flag** indicando se o contador deve contar para cima (*true*) ou para baixo (*false*)

A função `setup`

```
timer = timerBegin(0, prescaler, true);
```

Para este exemplo, usaremos o **primeiro** temporizador (timer 0) e passaremos **true** para o último parâmetro, para que o contador seja contado para cima

Com relação ao **prescaler**, o valor que deu certo no wokwi foi 40 (mas deveria ser 80)



Independente disso, a frequência de incremento do contador deu **1MHz**. Isto significa que o contador incrementará a cada 1 microsegundo

A função `setup`

```
timerAttachInterrupt(timer, &onTimer, true);
```

Associamos a função `onTimer` como uma interrupção do timer, usando a função `timerAttachInterrupt`

Enquanto os dois primeiros argumentos são autoexplicativos, o terceiro, `true`, indica que queremos que a interrupção seja do tipo borda (o que significa que ela deve ser acionada em uma borda de subida (*rising*) ou de descida (*falling*))

A função setup

```
timerAlarmWrite(timer, intervalo, true);
```

A função **timerAlarmWrite** configura o intervalo e modo de repetição. Ela especifica que um alarme deve ser acionado toda vez que a contagem do temporizador atingir **intervalo**

O terceiro argumento desta função, **true**, indica que o temporizador deve ser **recarregado** automaticamente. Isso significa que após gerar um alarme uma vez, ele deve recarregar automaticamente e começar a **contar novamente** de 0 a **intervalo**.

A função setup

```
timerAlarmEnable(timer);
```

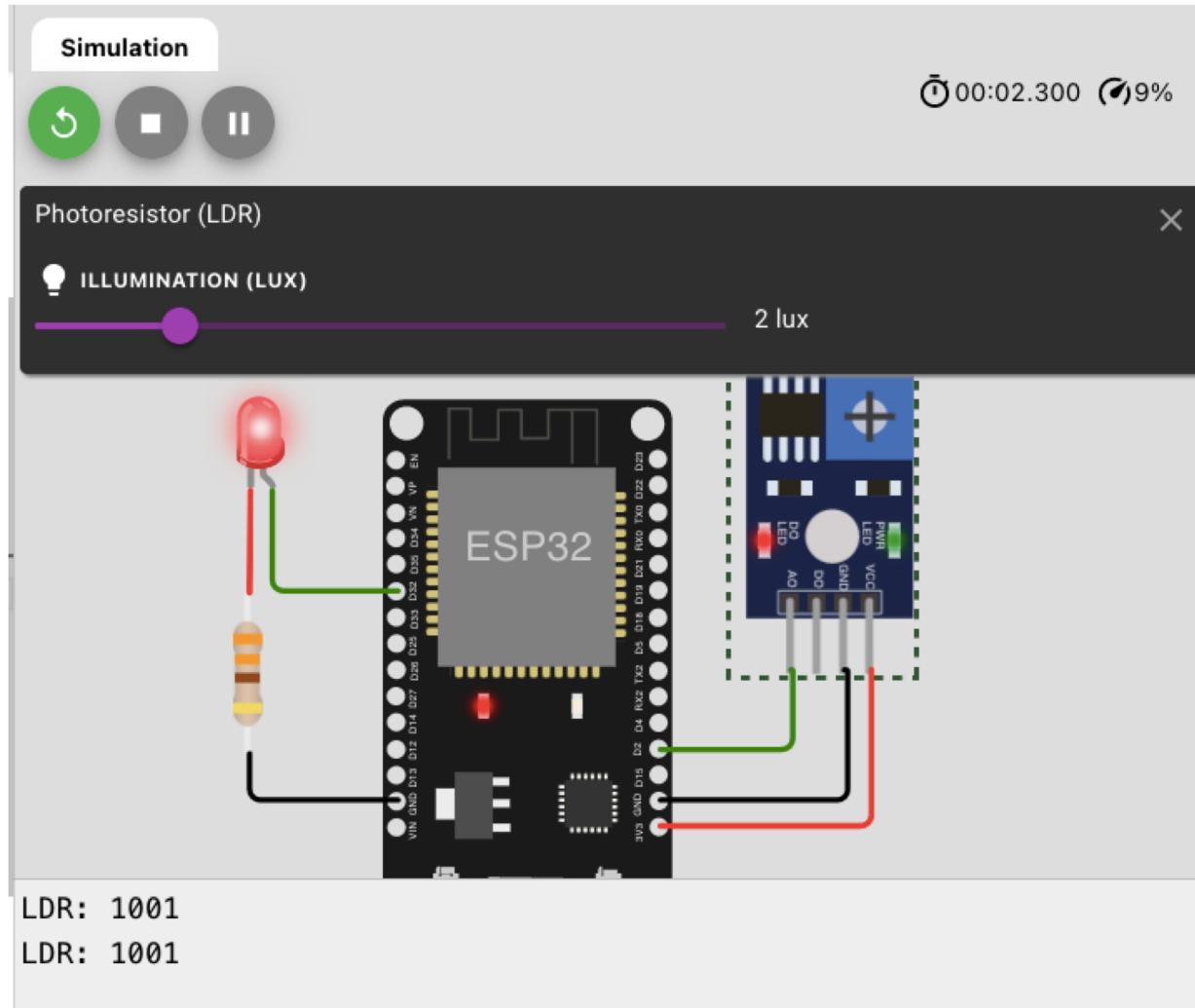
Por fim, habilitamos o Alarme usando
timerAlarmEnable

A função `loop()`

No `loop`, analisamos a variável "**interrupted**". Se for **true**, significa que o alarme foi acionado. Neste caso, devemos ler e imprimir o valor do sensor LDR, alterar o status do LED e, em seguida, o valor da variável compartilhada "**interrupted**" é tornada **false**

```
void loop() {  
    if(interrupted) {  
        Serial.print("LDR: ");  
        Serial.println(analogRead(LDR_Pin));  
        stateLED = !stateLED;  
        digitalWrite(LED, stateLED);  
        interrupted = false;  
    }  
}
```

Simulação



Simulação

