UFMG

UNIVERSIDADE FEDERAL DE MINAS GERAIS

PROJETOS DE SISTEMAS EMBUTIDOS

Laboratório 1 – Pisca LED

Vincent Pernarh

BELO HORIZONTE 2024/2

Professor : Diogenes Cecilio da Silva Junior

A. INTRODUÇÃO

Este relatório tem como objetivo apresentar os conceitos fundamentais do projeto de sistemas embarcados. Para isso, foram realizadas diversas atividades, incluindo a seleção e descrição do microcontrolador e de seus periféricos, visando a realização de um experimento de acionamento de um LED controlado por uma interrupção gerada por uma chave. Da Seção 2 à Seção 4, serão detalhadas as características do microcontrolador utilizado e sua configuração para alcançar o resultado esperado.

B. DESCRIÇÃO DO SDK E PERIFÉRICOS

Para o projeto, utilizaremos o SDK do “ESP32-WROOM”, que é uma plataforma poderosa e versátil para o desenvolvimento de sistemas embarcados. A seguir, apresento os detalhes técnicos:

a. Nome do SDK: ESP32-WROOM
b. Modelo: ESP32-WROOM-32
c. Microcontrolador: ESP32-D0WDQ6 (dual-core Xtensa® 32-bit LX6)

d. **Memórias**
- Flash: 4MB
- SRAM: 520KB

e. Interfaces de E/S:
- 34 pinos GPIO
- 12 pinos de entrada analógica (ADC)
- 2 pinos de saída analógica (DAC)
- Interfaces de comunicação: UART, SPI, I2C, PWM, I2S, CAN
- Conectividade Wi-Fi e Bluetooth integrados

f. Localização dos Sinais de Interface

Abaixo está a imagem com a localização de todos os sinais de interface disponíveis no ESP32-WROOM-32, facilitando a conexão e utilização dos periféricos durante o desenvolvimento : Figure 1.

g. Ambiente de Desenvolvimento de Software.
O ambiente de desenvolvimento utilizado será o **Arduino IDE**, que oferece suporte completo para a programação e compilação de projetos com o ESP32-WROOM. O Arduino IDE permite um processo de desenvolvimento simplificado, utilizando uma ampla gama de bibliotecas e exemplos para controle de periféricos e funcionalidades integradas do microcontrolador.

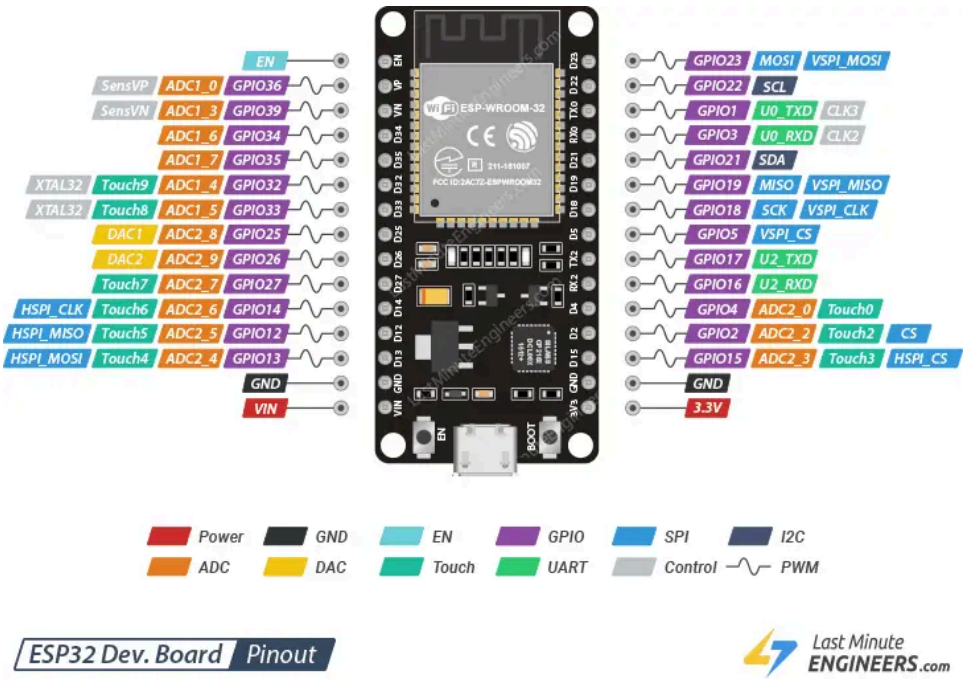

Figure 1: ESP32-WROOM - 32 PinOut

1. INTERFACE GPIO DE ENTRADA E SAÍDA.

Em um circuito elétrico, o resistor de pull-up é usado para garantir um estado conhecido para um sinal,geralmente quando utilizamos o resistor pull-up interno do ESP32-WROOM, evitamos a necessidade de adicionar um resistor externo ao circuito. A lógica por trás do uso do resistor interno é simples: o ESP32 possui resistores pull-up e pull-down configuráveis em seus pinos GPIO,no caso deste projeto, conectamos a GPIO 4, que podem ser habilitados via software.

Neste caso, ao configurar o pino de entrada para usar o resistor pull-up interno, garantimos que, quando a chave estiver fechada/ pressionada, o pino é conectado diretamente ao 3.3V, resultando em um nível alto (HIGH), permitindo que o microcontrolador detecte essa mudança de estado, vise o código abaixo.

```
pinMode(buttonPin, INPUT_PULLUP);
```

No ESP32, as interrupções são gerenciadas por vetores que apontam para os manipuladores de cada evento. O **RESET** tem a prioridade mais alta, seguido por outras interrupções configuradas, como as de GPIO. O ESP32 utiliza uma arquitetura mais avançada, baseada no Xtensa e no FreeRTOS, permitindo maior flexibilidade e controle sobre as prioridades das interrupções.

A configuração do pino de GPIO para interrupções inclui a inicialização do pino como entrada e a configuração da interrupção para detectar mudanças no sinal. O código faz uso da função *digitalPinToInterrupt()* para associar o pino digital 4 ao número da interrupção apropriado. A função de serviço de interrupção *switchInterrupt* é definida para atualizar variáveis globais quando a interrupção é acionada, permitindo ao sistema responder rapidamente a eventos sem a necessidade de sondagem contínua.

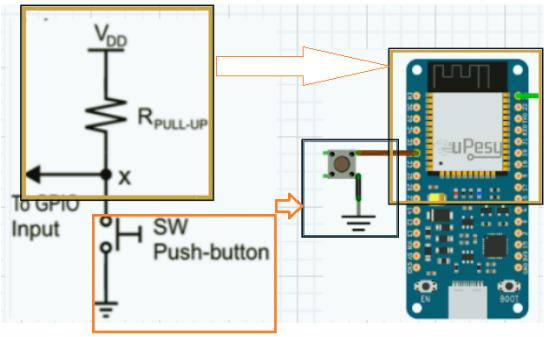

Figura 2 : Representação do resistor interno do ESP32-wroom (fonte : misto)

Além disso,o projeto consiste em um LED conectado a um pino GPIO configurado como saída no ESP32. O LED será ligado ao pino GPIO de modo que, quando o pino estiver em nível alto (HIGH), o LED acenderá, e quando estiver em nível baixo (LOW), o LED apagará.Neste projeto, o LED foi conectado ao GPIO 15, onde o ânodo (terminal positivo) do LED é conectado em série com um resistor de 330Ω, enquanto o cátodo (terminal negativo) é conectado ao pino de saída (GPIO) do ESP32.A função do resistor é para limitar a corrente, protegendo tanto o LED quanto o microcontrolador.

No código, o pino GPIO será configurado como saída utilizando o comando **`pinMode(pino, OUTPUT)`** no setup do Arduino IDE, onde o comando digitalWrite() é usado amarrando o pino e estado a ele assim o LED acende ou apaga.

No diagrama abaixo, é possível observar que, quando o comando **digitalWrite(pino, HIGH)** é executado, o pino GPIO recebe 3.3V do ESP32. A corrente passa pelo resistor, que limita o fluxo, permitindo que o LED conduza e acenda. Por outro lado, ao executar o comando **digitalWrite(pino, LOW)**, o pino GPIO é conectado diretamente ao GND, interrompendo a condução no LED e fazendo com que ele apague.

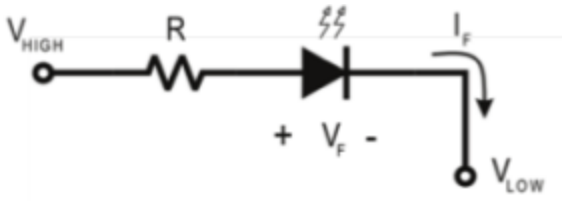

Figura 3 : Diagrama elétrico para conduzir um LED(Fonte : Slide do professor).

2. No código abaixo, a ISR do ESP32 foi configurada para detectar bordas de **subida (RISING)** quando uma interrupção ocorre. Para outras situações, a ISR pode ser configurada para bordas de **descida (FALLING)** ou mudanças de **estado (CHANGE).** A ISR verifica o estado do pino GPIO no momento da interrupção para determinar qual borda gerou o evento.

```
attachInterrupt(digitalPinToInterrupt(buttonPin), switchInterrupt, RISING);
```

A configuração de borda de subida foi escolhida devido ao uso do resistor pull-up interno do ESP32. Como o pull-up mantém o pino em nível lógico **HIGH** por padrão, o sinal é invertido, e a interrupção ocorre apenas quando o botão é **solto** (transição de **LOW** para **HIGH**). Isso facilita o controle e processamento do evento gerado pelo botão,veja figura 2.

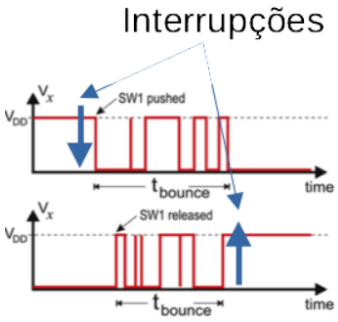

Figura 4 : Configuração de detecção de sinais.

3. **EXPERIMENTO.**

Para realização do projeto, foi empregado o diagrama abaixo :

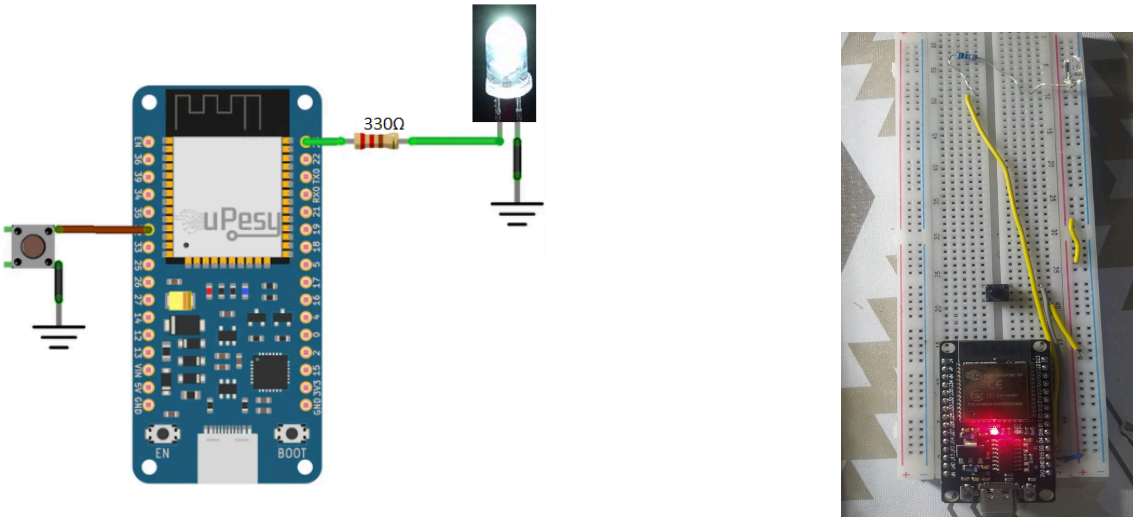

Figura 5 : Circuito elétrico esquemático e montagem real.

4. Requisitos do sistema :

a. Esperar a chave ser pressionada e liberada.
b. Acender o LED por 1 segundo.
c. Apague o LED por 2 segundos.
d. Finalmente, piscar rapidamente o LED por meio segundo indicando fim de ciclo, ou seja, acenda por 0,25 segundo, apague por 0,25 segundo, acenda por 0,25 segundo, apague o LED;
e. 5. Volte ao passo 1.

Baseado na funcionalidade do sistema, a MEF – Máquina de Estados Finita, foi elaborado sendo base para construção do código,veja figura 6 .

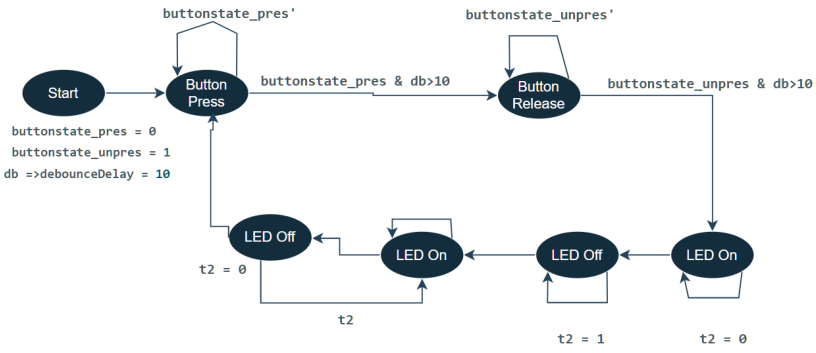

Figura 6 : MEF – Máquina de Estados Finita do sistema.

5. Descrição Funcional.

O código começa definindo os pinos para o botão e o LED bem como os estados e outras variáveis auxiliares, além de duas variáveis para rastrear o estado do botão pressionado e não pressionado. O tempo de debounce é configurado para 10 milissegundos que é implementado no ISR, o que ajuda a evitar leituras incorretas durante o acionamento do botão.

Na função `setup()`, o pino do botão é configurado como entrada com resistor pull-up, enquanto o pino do LED é configurado como saída, iniciando o LED apagado. A comunicação Serial é ativada para depuração. Além disso, a função `attachInterrupt()` é chamada para configurar uma interrupção no pino do botão, que vai disparar a função `switchInterrupt()` quando o botão for pressionado.

A função `loop()` é responsável por gerenciar o ciclo de estados da máquina de estados finitos (FSM). O programa começa aguardando o pressionamento do botão, o que aciona a transição para o estado `WAIT_PRESS`. Em seguida, aguarda a liberação do botão, respeitando o tempo de debounce. Após o botão ser pressionado e liberado, o LED é aceso por 1 segundo no estado `LED_SEQUENCE`. Depois disso, o LED é apagado por 2 segundos antes de iniciar uma sequência de piscadas rápidas no estado `PISCA_LED`, sinalizando o fim da operação.

A função `switchInterrupt()` é executada sempre que ocorre uma interrupção no pino do botão, garantindo que o estado da FSM seja atualizado para `WAIT_PRESS` após um evento de pressionamento do botão. Ela implementa a lógica de debounce para evitar que múltiplas leituras incorretas ocorram durante o acionamento do botão.

Finalmente, a função `func_delay()` é usada para implementar atrasos temporais, como o tempo de acionamento do LED e o tempo entre piscadas. Ela utiliza a função `millis()` para realizar o atraso e bloquear o processamento de outras tarefas ou interrupções.

6. CONCLUSÃO

Em conclusão, o projeto foi bem-sucedido, cumprindo todos os requisitos propostos. A aplicação dos conhecimentos fundamentais em circuitos eletrônicos foi crucial para o desenvolvimento, tornando o processo mais ágil e eficiente. A demonstração prática, através de um vídeo disponibilizado no YouTube, servirá como ferramenta visual para explicar o funcionamento do sistema, complementando a apresentação em sala de aula. Esse projeto não apenas consolidou os conceitos adquiridos, mas também proporcionou uma experiência valiosa de integração entre teoria e prática.

7. REFERENCIAS

a. https://lastminuteengineers.com/?__im-QrcQOQJp=15173755891247095350
b. https://www.upesy.com/blogs/tutorials/how-to-use-gpio-pins-of-esp32-with-arduino
c. https://lastminuteengineers.com/?__im-fcFKXFbU=8975801794328265667#google_vignette

8. Codígo

```
// States
enum State {WAIT_PRESS, WAIT_RELEASE, LED_SEQUENCE,
PISCA_LED,ON};
State currentState = ON;

// Define the pin numbers
const int buttonPin = 4;    // GPIO pin for button
const int ledPin = 15;      // GPIO pin for LED

// Debounce delay in milliseconds
const long debounceDelay = 10;
unsigned long lastDebounceTime = 0; // Último tempo de debounce

// Button states
bool buttonstate_pres = LOW;
bool buttonstate_unpres = HIGH;

void setup() {
  // Configure the button pin with an internal pull-up resistor
  pinMode(buttonPin, INPUT_PULLUP);

  // Configure the LED pin as output
  pinMode(ledPin, OUTPUT);

  // Initialize the LED to OFF
```

```
  digitalWrite(ledPin, LOW);
  attachInterrupt(digitalPinToInterrupt(buttonPin),
switchInterrupt, RISING);
  // Initialize serial for debugging
  Serial.begin(115200);
}

void loop() {
  switch (currentState) {
    case WAIT_PRESS:
      // Wait for the button to be pressed
      buttonstate_pres = HIGH;
      currentState = WAIT_RELEASE;  // Move to next state
      break;

    case WAIT_RELEASE:
      // Wait for the button to be released
      buttonstate_unpres = LOW;
      currentState = LED_SEQUENCE;  // Move to LED sequence

      break;

    case LED_SEQUENCE:
      // Turn LED ON for 1 second
      digitalWrite(ledPin, HIGH);
      Serial.println("Button pressed and released");
      func_delay(1000);

      // Turn LED OFF
      digitalWrite(ledPin, LOW);
      func_delay(2000);
      currentState =PISCA_LED;
      break;

    case PISCA_LED:
```

```
      // Turn LED ON for 250 ms (indicating the end of the
sequence)
      digitalWrite(ledPin, HIGH);
      func_delay(250);
      digitalWrite(ledPin, LOW);

      func_delay(250);
      digitalWrite(ledPin, HIGH);
      func_delay(250);
      digitalWrite(ledPin, LOW);

         // Return to initial state/ bloking state.
      currentState = ON;
      break;

    default:
      currentState = ON;
      break;
  }
}


//Implementing debounce function
void switchInterrupt() {
  unsigned long currentMillis = millis();


  if (currentMillis - lastDebounceTime >= debounceDelay ) {
   currentState = WAIT_PRESS;
  }


  lastDebounceTime = currentMillis;
}
```

```
//implementing delay function

void func_delay(unsigned long timer) {
  unsigned long startMillis = millis();  // Record the start
time
  while (millis() - startMillis < timer) {
    // Keep looping until the desired time has passed
  }
}
```