UFMG

UNIVERSIDADE FEDERAL DE MINAS GERAIS

PROJETOS DE SISTEMAS EMBUTIDOS

Laboratório 2 – Pisca LED Com Temporizador e Interrupção

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 e a utilização de temporizadores para controlar o debounce da 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.

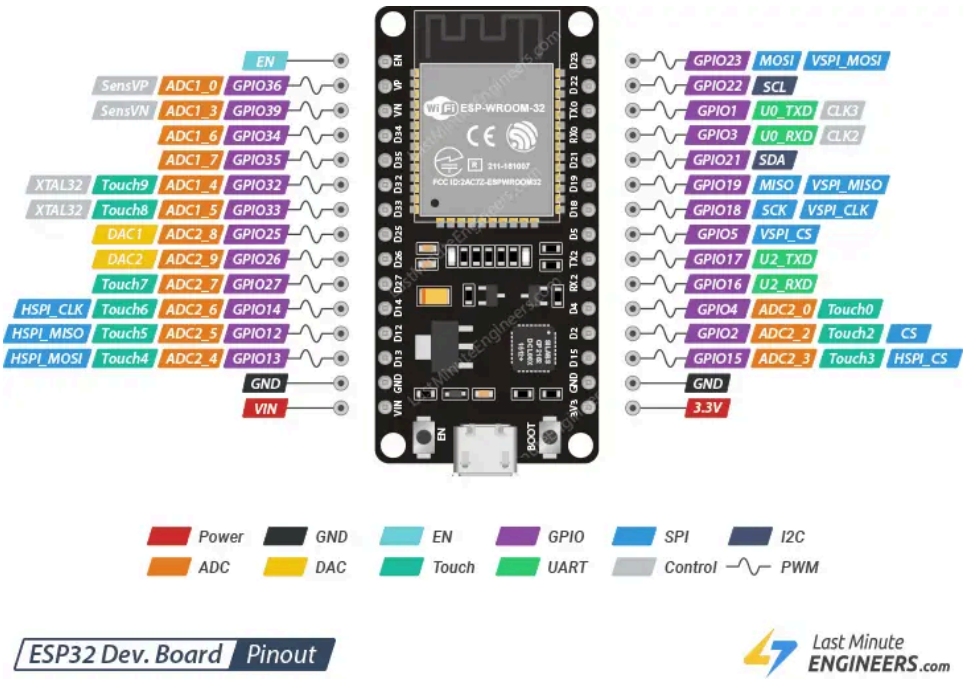

Figure 1: ESP32-WROOM - 32 PinOut

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.

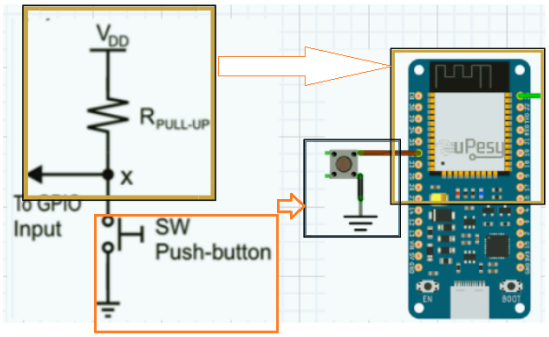

Figura 2 : Representação do resistor interno do ESP32-wroom (fonte : misto)

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.

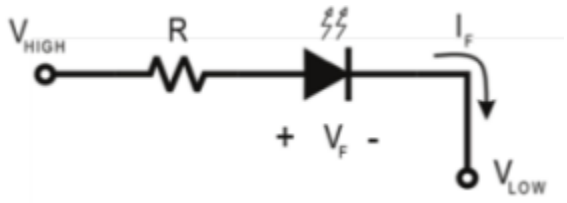

Figura 3 : Diagrama elétrico para conduzir um LED(Fonte : Slide do professor).

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

   attachInterrupt(digitalPinToInterrupt(buttonPin), switchInterrupt, Change);

   A configuração da interrupção é escolhida como **change**, pois queremos detectar quando ocorre uma mudança de estado nos botões e mapear estes estados para os passos seguintes, 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.

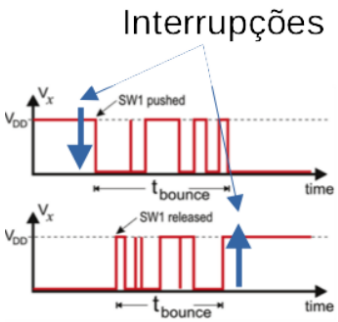

Figura 4 : Configuração de detecção de sinais.

3. **EXPERIMENTO.**

   Para realização do projeto, foi empregado o diagrama abaixo :

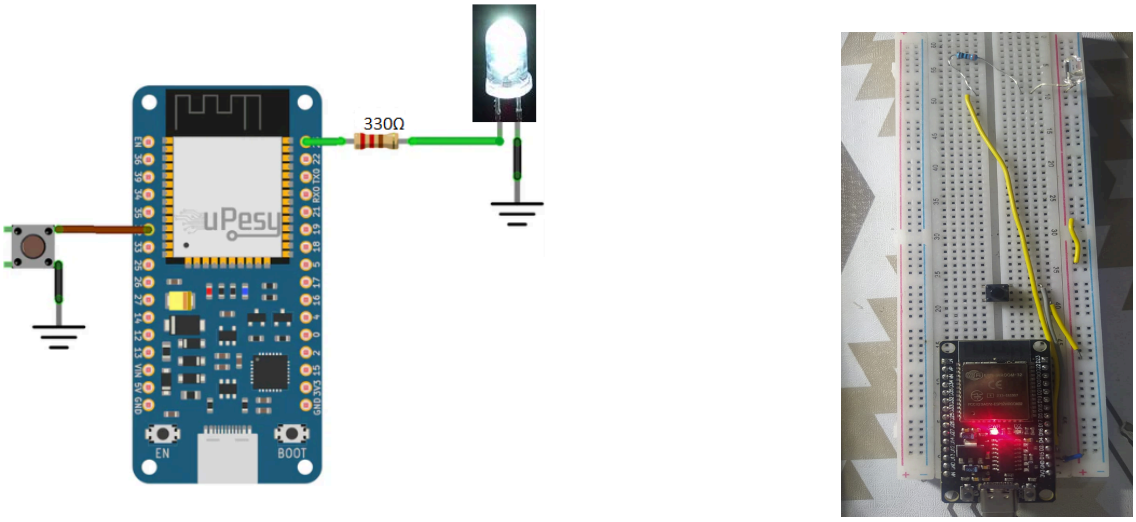

Figura 5 : Circuito elétrico esquemático e montagem real.

4. Requisitos do sistema :

1. Esperar a chave ser pressionada fazer o debounce.
2. Medir o tempo T até a chave ser liberada.
3. Fzer o debounce.
4. Se T<1 segundos,então pisca o LED uma vez(acender e apagar por 0,5 segundos)
5. Se 1s < T < 2s, então pisca o LED 2 vezes,
6. Se T >3s,então o LED 3 vezes.
7. Finalmente,pisca 2 vezes rapidamente o LED indicando fim de ciclo (acender por 0,25s,apagar por 0,25s)
8. 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 .

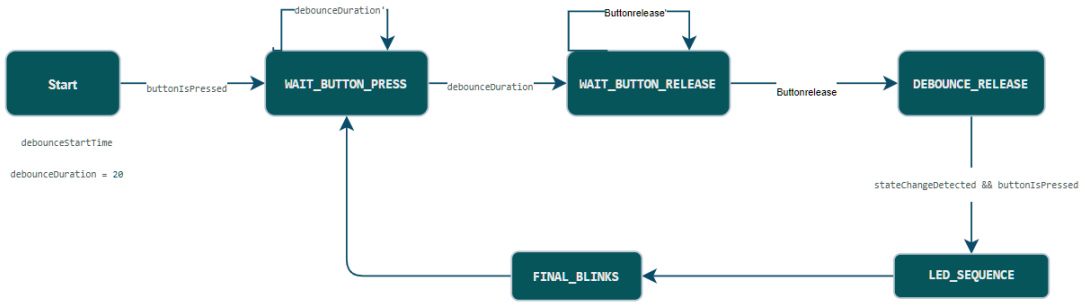

Figura 6 : MEF – Máquina de Estados Finita do sistema.

5. Descrição Funcional do Sistema.

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 de pressionado e liberado é configurado para 20 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 `handleButtonInterrupt()` 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 e interrupção para ser ativado, o que aciona a transição para o estado `WAIT_BUTTON_PRESS`, enquanto isso, o contador é disparado para computar os tempo em que o led pode ser acendida (T<1s = uma vez, 1s<T>2s = 2 vezes e por fim T>3s = 3 vezes).Em seguida, aguarda a liberação do botão, respeitando o tempo de debounce. Após o botão ser pressionado e liberado o temporizador pare de contar,e em seguida vai para o estado `DEBOUNCERELEASE` , neste estado, o conferencia do tempo foi feito e caso seja verdadeiro, o sistema vai para o estado `LED_SEQUENCE,` Este estado é crucial para o bem funcionamento do sistema, uma série de condições foram feitas para computar quantas vezes o led deve acender, sendo assim, um if e else foram estipulados e passado para uma função se chama `blinkLED(int times, int duration)` quem cuida de acender e desligar os LEDs de acordo com o número que deve acender e a duração do mesmo.

A função `handleButtonInterrupt()` é executada sempre que ocorre uma interrupção no pino do botão, garantindo que o estado da FSM seja atualizado para `WAIT_BUTTON_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 `millis()` é usada para implementar atrasos temporais e computar os temporizadores necessários para funcionamento do sistema, como o tempo de acionamento do LED e o tempo entre piscadas.

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 da prática, seria realizada em sala de aula junto ao professor da disciplina.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
d. https://www.makerhero.com/blog/uso-de-interrupcoes-externas-com-esp32/?srsltid=AfmBOopP60QVgv9yEjDnc1TA8CZvLKmW1_egBgHHLfq5Y7rhHiXBnlDs
e. https://medium.com/@madeadhika39/turn-on-led-on-esp32-with-push-button-8c8ee1b3652f

8. Codígo

```
const int ledGPIO = 21;
const int buttonGPIO = 15;

unsigned long debounceStartTime = 0;
```

```
const unsigned long debounceDuration = 20;

volatile bool stateChangeDetected = false;
volatile bool buttonCurrentState = HIGH;
volatile bool buttonIsPressed = false;

bool ledSequenceInProgress = false;
unsigned long pressStartTime = 0; // Time when the button was
pressed
unsigned long pressDuration = 0; // Duration of the button
press

enum States {
    WAIT_BUTTON_PRESS,
    WAIT_BUTTON_RELEASE,
    DEBOUNCE_RELEASE,
    LED_SEQUENCE,
    FINAL_BLINKS
};
int currentState = WAIT_BUTTON_PRESS;

void setup() {
    Serial.begin(115200);
    pinMode(ledGPIO, OUTPUT);
    pinMode(buttonGPIO, INPUT_PULLUP);
    attachInterrupt(digitalPinToInterrupt(buttonGPIO),
handleButtonInterrupt, CHANGE);
}

void loop() {
    unsigned long currentMillis = millis();

    switch (currentState) {
        case WAIT_BUTTON_PRESS:
            if (stateChangeDetected && buttonIsPressed) {
                stateChangeDetected = false;
                pressStartTime = currentMillis; // Record the
press start time
                currentState = WAIT_BUTTON_RELEASE;
            }
```

```
            break;

        case WAIT_BUTTON_RELEASE:
            if (stateChangeDetected && !buttonIsPressed) {
                stateChangeDetected = false;
                pressDuration = currentMillis -
pressStartTime; // Calculate press duration
                debounceStartTime = currentMillis;
                currentState = DEBOUNCE_RELEASE;
            }
            break;

        case DEBOUNCE_RELEASE:
            if ((currentMillis - debounceStartTime) >
debounceDuration) {
                currentState = LED_SEQUENCE;
                ledSequenceInProgress = true;
            }
            break;

        case LED_SEQUENCE:
            if (pressDuration < 1000) {
                blinkLED(1, 500); // Blink 1 time, 0.5 seconds
each
            } else if (pressDuration >= 1000 && pressDuration
< 2000) {
                blinkLED(2, 500); // Blink 2 times, 0.5
seconds each
            } else if (pressDuration >= 3000) {
                blinkLED(3, 500); // Blink 3 times, 0.5
seconds each
            }
            currentState = FINAL_BLINKS;
            break;

        case FINAL_BLINKS:
            blinkLED(2, 250); // Rapid blink 2 times (0.25s
each)
            ledSequenceInProgress = false;
```

```
            currentState = WAIT_BUTTON_PRESS; // Return to
waiting for button press
            break;
    }
}

void handleButtonInterrupt() {
    if (!ledSequenceInProgress) {
        buttonCurrentState = digitalRead(buttonGPIO);
        if ((millis() - debounceStartTime) > debounceDuration)
{ // Check debounce timing
            stateChangeDetected = true;
            buttonIsPressed = (buttonCurrentState == LOW);
        }
        debounceStartTime = millis();
    }
}

void blinkLED(int times, int duration) {
    for (int i = 0; i < times; i++) {
        digitalWrite(ledGPIO, HIGH);
        delay(duration / 2);
        digitalWrite(ledGPIO, LOW);
        delay(duration / 2);
    }
}
```