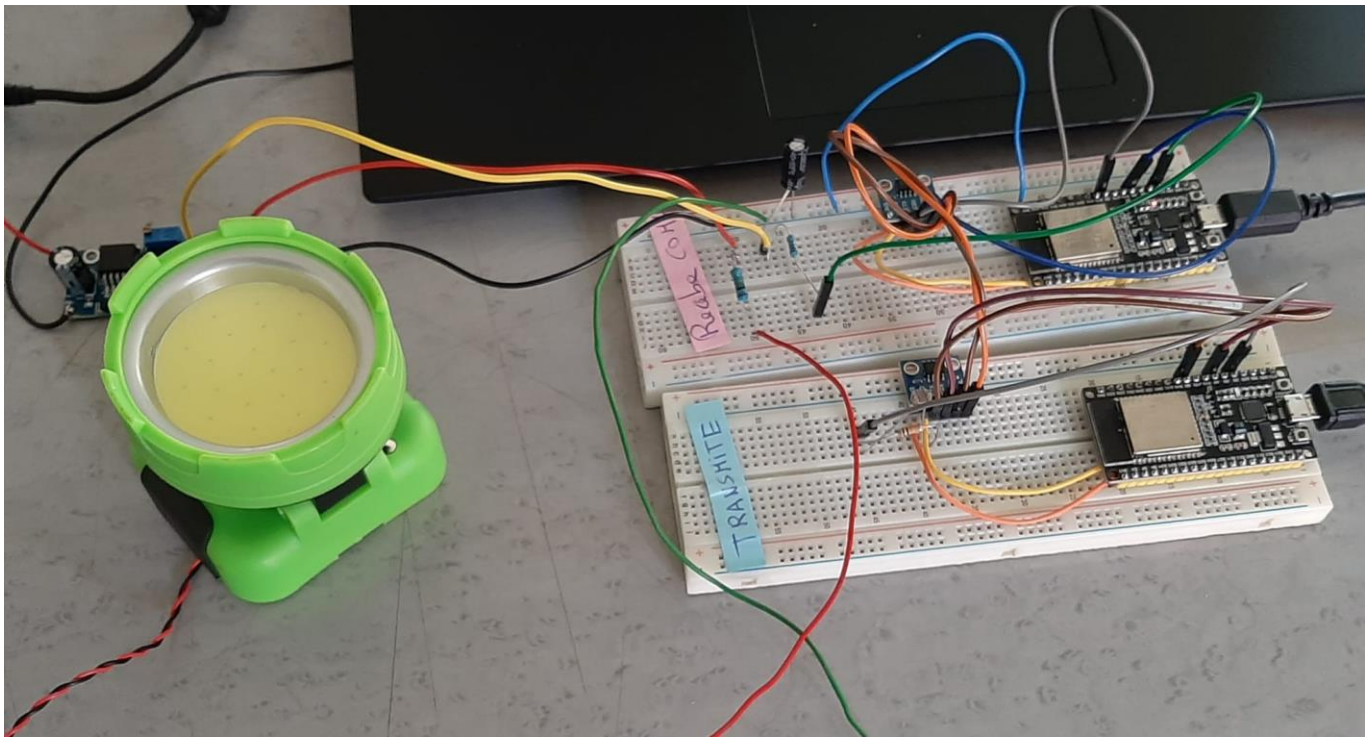
ipt Instituto Politécnico de Tomar

estt.ipt
Escola Superior
de Tecnologia de Tomar
Instituto Politécnico de Tomar

Mestrado em Engenharia Eletrotécnica



Iluminação Inteligente e Leitura de Dados em Tempo Real

Projeto Final SDC

Ano letivo 2024-25

20969, David Rito 22840, João Protásio 23154, Rúben Laranjo

E: aluno20969@ipt.pt **E:** aluno22840@ipt.pt **E:** aluno23154@ipt.pt

CONTEÚDO

Iluminação Inteligente 3

Objetivo 3

Introdução 3

Plataformas e Ferramentas 4

Esquema Elétrico 7

Fluxograma 8

ESP32 SLAVE (Transmissor) 9

Descrição 9

Código 9

ESP32 MASTER (Recetor) 10

Descrição 10

Código 10

MQTT (Subscriber) e Amazenamento de Dados na InfluxDB 13

Descrição 13

Código 13

InfluxDB & Grafana 15

Criação de Gráficos e Configuração Dashboard em InfluxDB & Grafana 16

Conclusão 18

Anexos 19

Iluminação Inteligente

Objetivo

Projetar e desenvolver um Sistema Distribuído de Controlo, de forma a cobrir, o mais amplamente possível, os assuntos focados nesta unidade curricular. Este relatório pretende fazer um enquadramento ao tema abordado onde demonstra todo o desenvolvimento, assim como as ferramentas e plataformas usadas, os dispositivos utilizados, esquemáticos, software e protocolos de comunicação.

Introdução

Todos os meses, as pessoas procuram formas de poupar na fatura da luz. Com sistemas "inteligentes", esse objetivo é mais fácil de alcançar. Um desses sistemas pode ser aplicado à iluminação.

A iluminação "inteligente" nada mais é do que uma nova forma de ativação da luz, em que não é necessário realizar uma ativação através de dispositivos tradicionais, como um interruptor. O seu funcionamento, bem como a regulação da luminosidade artificial, resulta de um sistema de ativação automático, realizado por meio de sensores que detetam a intensidade luminosa. A intensidade luminosa do ponto de luz é regulada automaticamente em função da luminosidade ambiente, sendo o fator determinante para controlo do mesmo.

Estes sistemas podem ser configurados e controlados através de sistemas *wireless*, tais como *smartphones* ou *smart speakers* ou através de sistemas cablados ligados a consolas ou a computadores. Permitem definir o nível de conforto em função da atividade realizada na divisão. Tomemos uma sala de aula como exemplo, com recurso a um simples comando, a iluminação pode ser ajustada para permitir uma boa visualização de retroprojeção ou do que se escreve no quadro.

Como se facilmente se constata, o tema do nosso grupo recaiu num sistema "inteligente" de controlo de iluminação. A figura 1 exemplifica a estrutura utilizada para o projeto, repartido por 3 partes, comunicação remota entre dois ESP32, a gestão de dados pela máquina virtual e armazenamento em base de dados por InfluxDB e monitorizado no Grafana.

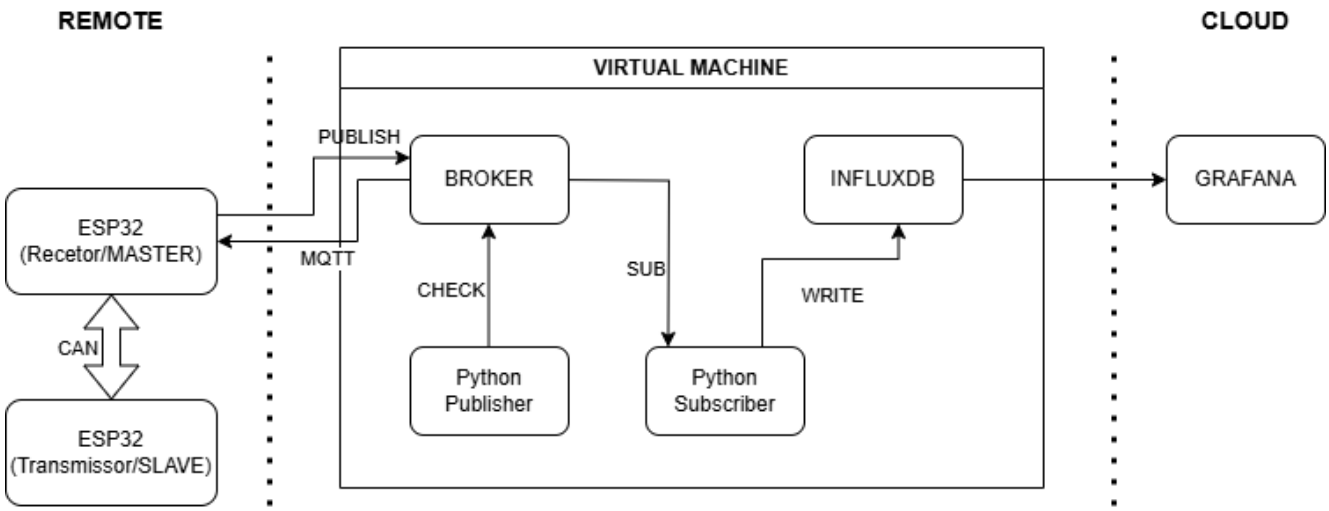


Figura 1 – Estrutura do Projeto

Plataformas e Ferramentas

Este capítulo enumera e apresenta uma breve introdução de todas as ferramentas utilizadas para este projeto. Todos os códigos apresentados são específicos para cada plataforma, visto que cada um tem o seu propósito no projeto final e que cumpre os requisitos do mesmo.

- *transmitter e receiver* (.cpp)

  Ambos os códigos são construídos em C++, e localizados em cada ESP32, SLAVE e MASTER, respetivamente. Os ambientes utilizados para este projeto foram o Arduino IDE e VSCODE. O Arduino oferece simplicidade na implementação de certas bibliotecas e funções, o VSCODE oferece mais versatilidade na configuração de programas, em que é possível atribuir scripts para diferentes portas COM, permitindo assim um tempo rápido de teste com 2 placas ESP32 conectadas ao mesmo computador e integração de FREERTOS pelo PlatformIO. Ambos os ambientes foram utilizados de acordo com as necessidades.

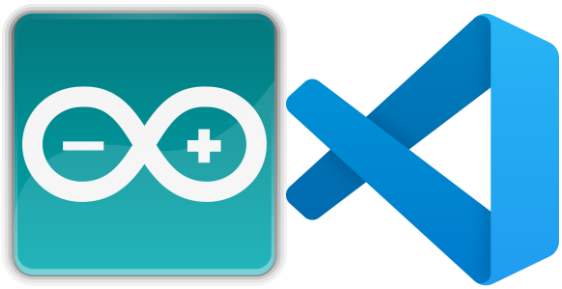


Figura 2 – Editores de Código

- Main_subscribe (.py)

  O *script* em *Python* é executado através do VSCODE instalado no sistema operativo LINUX. A ativação do broker (*Mosquitto Client*) é realizada pela máquina virtual Ubuntu e conjunto com o *script* ao qual subscreve aos tópicos no *broker*. O estabelecimento com o cliente MQTT necessita cumprir alguns requisitos: o IP associado aos ESP32 devem estar na mesma *network* do servidor MQTT e confirmar que a rede na máquina virtual se encontra interligada(*bridged*). Caso o MQTTX assuma os mesmos tópicos que o *script* executado, será então subscrito e a receção de mensagens é feita.

Figura 3 – Máquina Virtual Ubuntu & MQTTX

- Base de Dados

Através do mesmo *script* executado na máquina virtual, todos os dados recebidos pelo MQTTX são imediatamente enviados para base de dados. A InfluxDB é uma base de dados de séries cronológicas (TSDB) desenvolvida pela empresa InfluxData. É utilizada para o armazenamento e recuperação de dados de séries temporais em domínios como a monitorização de operações, métricas de aplicações, dados de sensores da IoT e análises em tempo real. Esta base de dados revela-se essencial para monitorização de todos os dados obtidos com registo temporal e uma melhor monitorização em um grande número de amostras para futuros estudos.

Figura 4 – InfluxDB

Todos os dados recebidos podem ser monitorizados, selecionando os respetivos campos e visualizados num gráfico como mostra no exemplo. Esta é a primeira visualização gráfica em que observamos os dados, posteriormente no relatório irá ser mostrado em *dashboard*, uma visualização mais interativa e de fácil observação dos diversos valores que são emitidos para a base de dados.

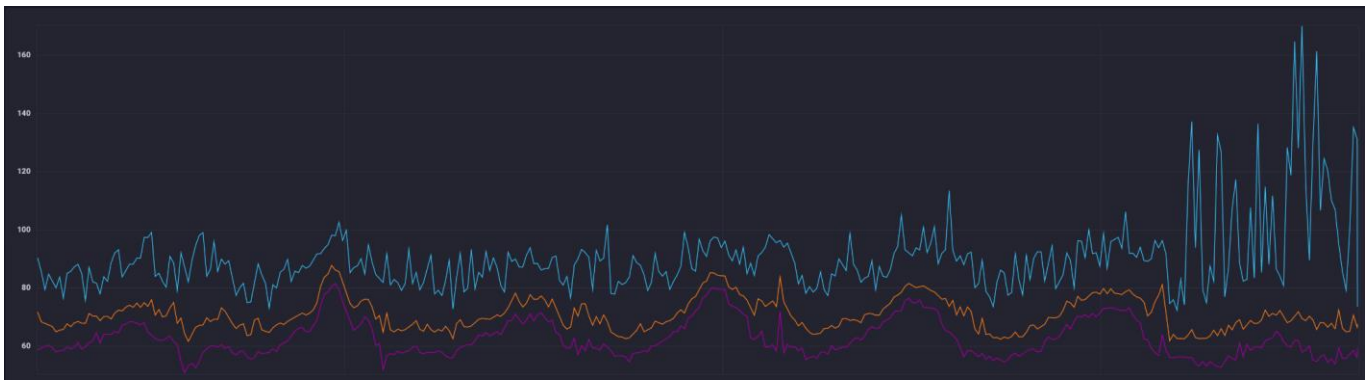


Figura 5 – Visualização de Dados em InfluxDB

- *CLOUD*

A monitorização pode ser realizada apenas pelo InfluxDB, em que já temos uma ótima visualização e monitorização dos dados, mas o objetivo será tornar a monitorização mais acessível e com possibilidade de armazenar em *Cloud* e para contornar este desafio, foi optado por a supervisão dos dados em Grafana Cloud. Uma forma de integrar em formato WEB e que fornece integrações de observabilidade de dados, que permite criar um sistema híbrido para que seja possível monitorizar diferentes amostras em nuvem e abrindo a hipótese de emparelhamento com ferramentas de open-source.

Figura 6 – Grafana

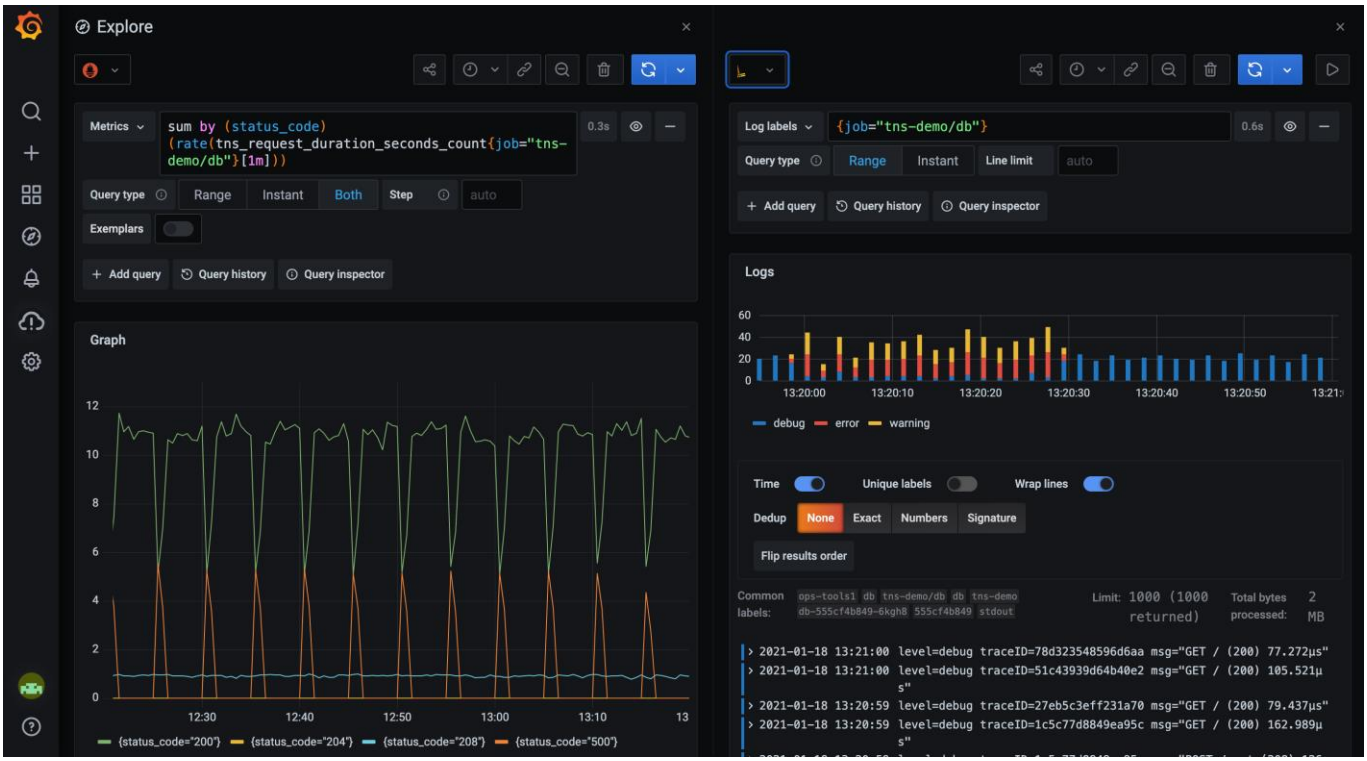


Figura 7 – Editor do Grafana (Exemplo)

Esquema Elétrico

O objetivo principal desta configuração é transmitir dados de sensores, especificamente medições de intensidade luminosa de um LDR (*Light Dependent Resistor*), do Slave para o Master através da rede de bus CAN e ajuste da intensidade de luz aplicada ao LED de acordo com a luminosidade ambiente.

O Slave ESP32 lê a intensidade da luz do sensor LDR e converte-a num valor digital. Estes dados do sensor são transmitidos através do barramento CAN utilizando o transcetor SN65HVD230. O ESP32 mestre recebe os dados, processa-os e pode ainda transmiti-los através de outros protocolos de comunicação, neste caso o MQTT, para monitorização remota em InfluxDB ou Grafana.

O esquema demonstra um sistema de comunicação por barramento CAN utilizando dois microcontroladores ESP32, permitindo a transmissão de dados de sensores em tempo real. A utilização de transcetores(*transceivers*) SN65HVD230 garante uma comunicação fiável pelo protocolo de comunicação CAN, enquanto o regulador de tensão LM2596 fornece energia estável a todo sistema.

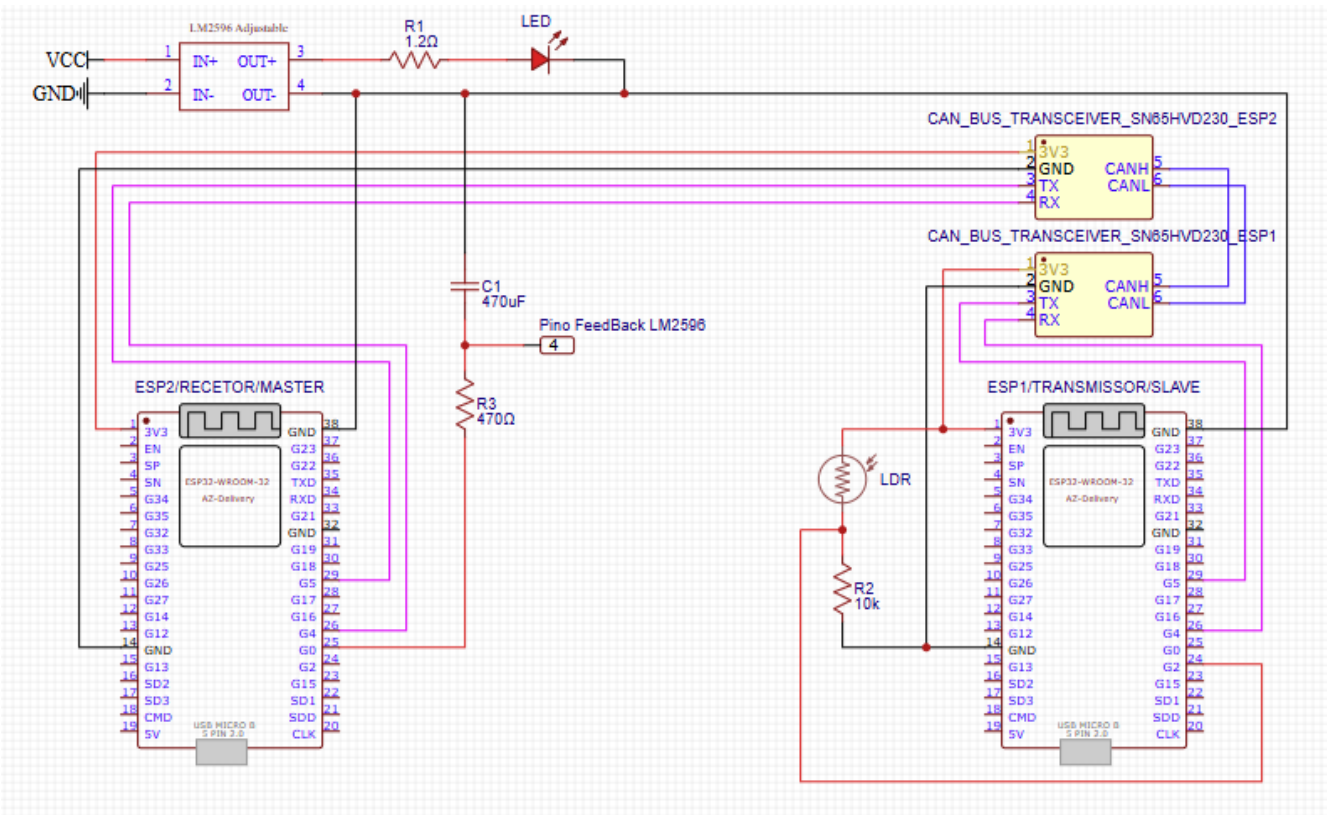


Figura 8 – Esquema Elétrico

Fluxograma

Neste fluxograma é mostrado como é idealizado e projetado o trabalho. Ambos os ESP32 estão repartidos por tarefas distintas que os carateriza como o SLAVE e MASTER. São designados desta forma, pois o SLAVE está responsável pela leitura do LDR, e enviar esses dados por CAN. Inicialmente a ideia seria conectar todos os periféricos ao SLAVE, para reduzir o número de processos no MASTER, sendo estes o sensor e a lâmpada LED. De forma a pôr à prova o uso de *tasks* com prioridades diferentes, decidimos estudar o comportamento do MASTER ao qual possui tarefas mais complexas. Entre estas tarefas, o MASTER é responsável por ler o valor LDR proveniente do SLAVE, e de seguida, recalcular o PWM a ser injetado à lâmpada, mapear as variáveis enviadas para o *broker* e uma tarefa extra de enviar para o SLAVE o valor atual do PWM.

Sumarizando:

1) $SLAVE \begin{cases} Lê\ o\ valor\ de\ PWM\ proveniente\ do\ MASTER\ (CAN) \\ Lê\ o\ valor\ do\ LDR \\ Envia\ o\ valor\ do\ LDR\ para\ o\ MASTER\ (CAN) \end{cases}$

2) $MASTER \begin{cases} Lê\ o\ valor\ de\ LDR\ proveniente\ do\ SLAVE\ (CAN) \\ Mapea\ as\ variáveis\ a\ publicar \\ Publica\ o\ valor\ do\ LDR,,brilho\ e\ PWM\ para\ MQTT \\ Envia\ o\ valor\ do\ PWM\ para\ o\ SLAVE\ (CAN) \\ Ajusta\ o\ valor\ de\ PWM\ do\ LED \end{cases}$

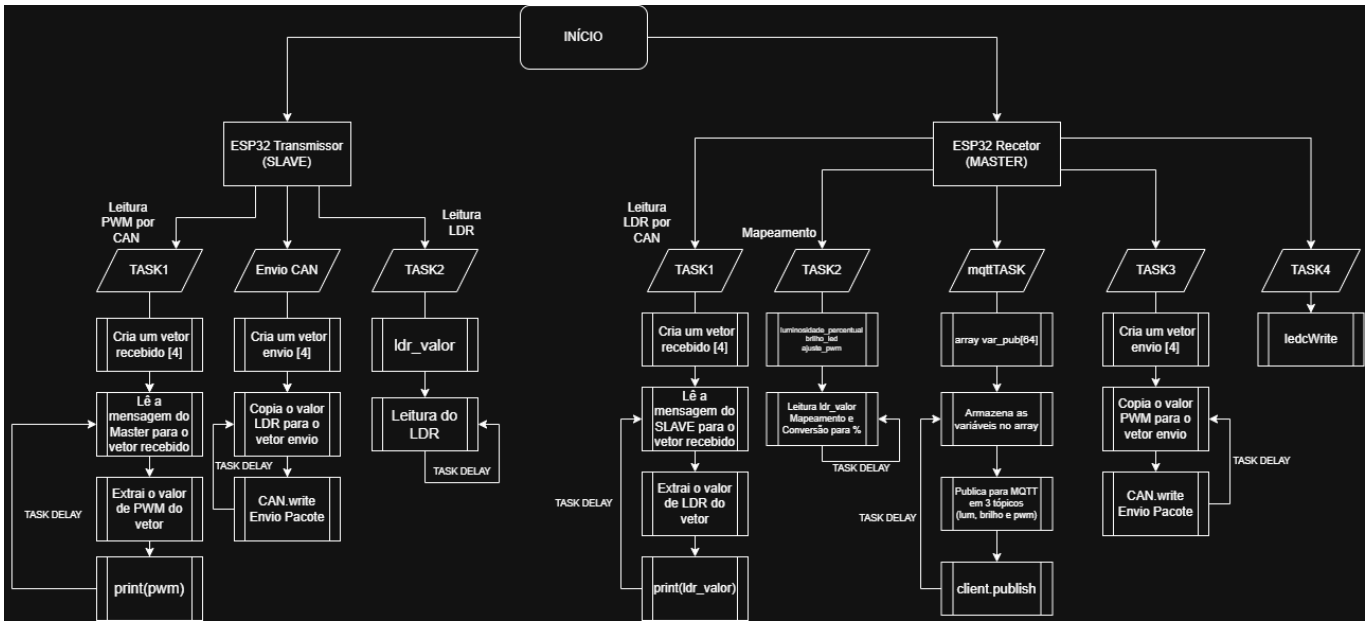


Figura 9 – Fluxograma

No próximo capítulo, é descrito todo o código utilizado, desde o código para cada controlador e a *script* em *Python* manuseada pela máquina virtual. O código é colocado na totalidade em anexo e de seguida explicado apenas excertos do código que se revelem essenciais para a compreensão do mesmo.

ESP32 SLAVE (Transmissor)

Descrição: Este projeto implementa um sistema baseado no ESP32 utilizando o FreeRTOS para *multitasking*. O ESP32 lê dados de um sensor LDR, processa-os e transmite através de um barramento CAN, enquanto recebe e apresenta dados de mensagens CAN recebidas. Em seguida, são demonstrados alguns excertos de código relevantes para compreensão do mesmo.

Código

```
#include <CAN.h>           // Biblioteca CAN

#define TX_GPIO_NUM   5  // GPIO 5 como pino CAN TX
#define RX_GPIO_NUM   4  // GPIO 3 como pino CAN RX

#define ldr_PIN 2       // GPIO 2 como entrada do LDR

float ldr_valor;  // Variavel que armazena as leituras do LDR
float pwm;        // Variavel que armazena o valor PWM

// Declaracao das Tarefas
TaskHandle_t envioHandle;
TaskHandle_t Task1;
TaskHandle_t Task2;

// ========================
// TASK: ENVIO CAN
// ========================
void envio(void *pvParameters){
  while(1){  // Ciclo Infinito

    // Preparacao da Mensagem Desejada
    uint8_t envio[4];  // Criacao do vetor 4 bytes
    memcpy(envio, &ldr_valor, sizeof(ldr_valor)); // Copia o valor do LDR para o vetor

    CAN.beginPacket(0x12); // Identificacao do pacote de envio ID 0x12
    CAN.write(envio,4);    // Envio dos 4 bytes de dados
    CAN.endPacket();       // Fim da Transmissao CAN

    vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
  }
}
```

A biblioteca CAN.h está incluída para gerir a comunicação pelo protocolo CAN. São atribuídos os pinos digitais (4 e 5) para RX e TX, respetivamente. As variáveis utilizadas são a o valor do LDR proveniente da entrada ADC (pino 2) e pwm que armazena o valor PWM emitido pelo MASTER. Com recurso aos cabeçalhos das tarefas, podemos então efetuar multitarefa. A primeira tarefa cria um *array* de 4 bytes, e com recurso a "memcpy", copiamos os blocos de dados da variável ldr_valor, onde posteriormente é realizado o envio do pacote para o barramento CAN com o seu ID pré-definido.

Figura 10 – Envio CAN

```
// ======================
// TASK 1: RECECAO CAN
// ======================
void task1(void *pvParameters) {
  while (1) {  // Ciclo Infinito

    int packetSize = CAN.parsePacket();  // Verificacao da rececao de um pacote CAN

    if (packetSize) {  // Caso receba o pacote
      int endereco_mensagem = CAN.packetId(); // Obtem o ID do pacote

      if (endereco_mensagem == 0x10) {  // Caso o ID seja 0x10
        uint8_t recebido[4];  // Criacao do array que armazena os bytes recebidos

        for (int i = 0; i < 4; i++) {
          recebido[i] = CAN.read();  // Leitura dos 4 bytes provenientes do barramento CAN
        }

        memcpy(&pwm, recebido, sizeof(pwm)); // Converte os bytes para float

        Serial.print(pwm);  // Imprime o valor PWM
        Serial.println();
      }
    }

    vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
  }
}
```

A tarefa de receção serve para testar a comunicação entre os dois microcontroladores, foi então criada uma tarefa que lê o valor de pwm proveniente do MASTER. Com recurso ao *parsePacket*, confirmamos a presença de pacotes no barramento CAN. À semelhança da tarefa anterior, é criado um *array* de 4 *bytes*, e os 4 *bytes* lidos do barramento são armazenados na variável pwm.

Figura 11 – Receção CAN

```
// ====================
// TASK 2: Leitura LDR
// ====================
void task2(void *pvParameters) {
  while (1) {  // Ciclo Infinito

    ldr_valor = analogRead(ldr_PIN); // Leitura valor analógico do LDR

    vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
  }
}
```

```
// ====================
// SETUP FUNCTION
// ====================
void setup() {
  Serial.begin(115200);  // Inicializacao Comunicacao Serie 115200 baud rate
  while (!Serial);       // Espera até o barramento Série estiver pronto

  // Configuracao barramento CAN pinos RX e TX
  CAN.setPins(RX_GPIO_NUM, TX_GPIO_NUM);

  // Inicializacao da comunicacao CAN a 500 kbps
  if (CAN.begin(500E3)) {
    Serial.println("Iniciou Comunicação CAN com SUCESSO");
  } else {
    Serial.println("Ocorreu algum erro ao iniciar a comunicação CAN");
    while (1);  // Cancela a comunicacao CAN caso ocorra alguma falha
  }
  // Atribuicao das tarefas FreeRTOS e definidas para o 1º Nucleo
  xTaskCreatePinnedToCore(envio, "envio", 2058, NULL, 1, &envioHandle, 1); // Envio CAN
  xTaskCreatePinnedToCore(task1, "Task1", 4096, NULL, 1, &Task1, 1); // Rececao CAN
  xTaskCreatePinnedToCore(task2, "Task1", 4096, NULL, 1, &Task2, 1); // Leitura LDR
}
```

A tarefa de leitura de LDR recorre à leitura analógica do conversor Analógico-Digital, e armazenado na variável ldr_valor. De notar que todas as tarefas, têm um atraso de 100ms antes de poderem ser executadas novamente. O *setup* inicializa pela habitual configuração da comunicação série para possibilidade de observar os valores no monitor. A configuração CAN com recurso à biblioteca, onde é configurado os pinos digitais declarados inicialmente para RX e TX e estabelecida a comunicação entre os dois microcontroladores.
Por fim, são criadas as tarefas, sendo estas atribuídas ao primeiro *core* (núcleo 1) do ESP32 e com a mesma prioridade.

Figura 12 – Leitura LDR e Setup Config SLAVE

ESP32 MASTER (Recetor)

Descrição: Este projeto implementa um sistema baseado no ESP32 que funciona como um recetor de barramento CAN e um editor MQTT, facilitando a comunicação entre dispositivos. O sistema integra WiFi, MQTT, comunicação de barramento CAN e controlo PWM, aproveitando a execução de tarefas em paralelo para uma execução eficiente de todos as tarefas desejadas. Em seguida, são demonstrados alguns excertos de código relevantes para compreensão do mesmo.

Código

```
//
//    ESP1 SLAVE RECETOR
//
#include <CAN.h> // Biblioteca CAN
#include <WiFi.h> // Biblioteca WiFi
#include <PubSubClient.h> // // Biblioteca MQTT Client


#define TX_GPIO_NUM   5 // GPIO 5 como pino CAN TX
#define RX_GPIO_NUM   4 // GPIO 3 como pino CAN RX


const int pwmChannel = 0; // canal 0 PWM
const int freq = 5000;  // Frequencia PWM 5kHz
const int resolution = 8;   // Resolucao 8 bits PWM (0<->255)
const int pwmPin = 15;  // Saida PWM Pino 15


float ldr_valor;
float ajuste_pwm;
float luminusidade_percentual;
float brilho_led;


// Configuracao WiFi e MQTT
const char* ssid = "iPhone";           // WiFi SSID
const char* password = "laranjo56";    // WiFi Password
const char* mqtt_server = "172.20.10.5"; // MQTT Broker IP

// Conexao WIFI<->MQTT
WiFiClient espClient;
PubSubClient client(espClient);
```

Este projeto tem a inclusão das bibliotecas CAN para comunicar com o parceiro, a biblioteca Wi-Fi, de forma a conectar o MASTER à mesma *network* que o servidor MQTT e PubSubClient para estabelecer comunicação com o cliente. De igual modo, os pinos RX e TX são atribuídos, e neste caso, como queremos publicar diversos dados, temos as variáveis de ldr, pwm, luminosidade e brilho. O pwm é configurado a 5kHz com uma resolução de 8 bits e atribuído ao pino 15. É realizada a conexão ao Wi-Fi de modo que esteja na mesma *network* que o *broker*.

Figura 13 – Declarações

```
// ========================
// TASK: PUBLICACAO NO MQTT
// ========================
void mqttTask(void* parameter) {
  client.setServer(mqtt_server, 1883); // Define o servidor MQTT e a porta padrão (1883)

  while (true) {
    if (!client.connected()) { // Verifica se há conexão com o broker MQTT
      reconnect(); // Se não estiver conectado, tenta reconectar, executa reconnect
    }

    client.loop(); // Mantém a conexão MQTT ativa e processa mensagens

    char var_pub[64]; // Array de caracteres para armazenar os dados a serem publicados

    // Publica a luminosidade percentual no tópico "sensor/dados/lum"
    snprintf(var_pub, sizeof(var_pub), "%.2f", luminusidade_percentual);
    client.publish("sensor/dados/lum", var_pub);

  // Publica o brilho do LED no tópico "sensor/dados/brilho"
    snprintf(var_pub, sizeof(var_pub), "%.2f", brilho_led);
    client.publish("sensor/dados/brilho", var_pub);

    // Publica o ajuste de PWM no tópico "sensor/dados/pwm"
    snprintf(var_pub, sizeof(var_pub), "%.2f", ajuste_pwm);
    client.publish("sensor/dados/pwm", var_pub);

    vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
  }
}
```

Este excerto representa a tarefa de publicação dos dados no *broker*, são atribuídos o servidor e a porta padrão. Através de outra função, é possível reconectar ao cliente em caso de falha e de seguida, é então criado um *array* de carateres que armazena os dados a visualizar (luminosidade, brilho e pwm). O armazenamento é possível através do "snprintf" que formata em *float* e armazena os dados das variáveis no *array var_pub*, que posteriormente é publicado no cliente de MQTTX com os tópicos lum, brilho e pwm.

Figura 14 – Publicação MQTT

```
// ========================
// TASK1: RECECAO CAN
// ========================
void task1(void *pvParameters) {
  while (1) { // Ciclo Infinito
    int packetSize = CAN.parsePacket(); // Verificacao da rececao de um pacote CAN

    if (packetSize) { // Caso recebe o pacote
      int endereco_mensagem = CAN.packetId(); // Obtem o ID do pacote
      if (endereco_mensagem == 0x12) {  // Caso o ID seja 0x12
        uint8_t recebido[4]; // Criacao do array que armazena os bytes recebidos
        for (int i = 0; i < 4; i++) {
          recebido[i] = CAN.read(); // Leitura dos 4 bytes provenientes do barramento CAN
        }

        memcpy(&ldr_valor, recebido, sizeof(ldr_valor)); // Converte os bytes para float

        Serial.print(ldr_valor); // Imprime o valor LDR
        Serial.println();

      }
    }
    vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
  }
}
```

A tarefa de receção lê o valor de LDR proveniente do SLAVE. Com recurso ao *parsePacket*, confirmamos a presença de pacotes no barramento CAN. À semelhança da tarefa anterior, é criado um *array* de 4 *bytes*, e os 4 *bytes* lidos do barramento são armazenados na variável ldr_valor.

Figura 15 – Receção CAN

```
// TASK2: MAPEAMENTO
// ========================
void task2(void *pvParameters) {
while (1) { // Ciclo Infinito

    // Converte o valor do sensor LDR para porcentagem de luminosidade
    luminusidade_percentual = (ldr_valor / 4095)*100;

    // Converte o ajuste PWM para brilho do LED em porcentagem (invertido)
    brilho_led = map(ajuste_pwm,0,255,100,0);

    // Recalcula o ajuste de PWM baseado no valor do LDR
    ajuste_pwm = map(ldr_valor,0,4095,0,255);

    vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
  }
}

// ========================
// TASK3: ENVIO CAN
// ========================
void envio(void *pvParameters){
  while(1){ // Ciclo Infinito
  uint8_t envio[4]; // Cria um vetor de 4 posições para armazenar os dados a serem enviados
  memcpy(envio, &ajuste_pwm, sizeof(ajuste_pwm)); // Copia o valor de ajuste PWM para o vetor de envio

  CAN.beginPacket(0x10); // Identificacao do pacote de envio ID 0x12

  CAN.write(envio,4);  // Envio dos 4 bytes de dados

  CAN.endPacket(); // Fim da Transmissao CAN
```

O mapeamento converte as leituras do sensor LDR numa percentagem da luminosidade ambiente. O brilho do LED é ajustado inversamente ao nível de luz. Após obter o valor de LDR é através da leitura do LDR que se define um valor PWM para controlo automático da luminosidade. A tarefa de envio CAN cria um *array* de 4 *bytes*, e com recurso a "memcpy", copiamos os blocos de dados da variável ldr_valor, onde posteriormente é realizado o envio do pacote para o barramento CAN com o seu ID pré-definido.

Figura 16 – Mapeamento & Envio CAN

```
// ========================
// TASK4: PWM
// ========================
void task3(void *pvParameters) {
  while (1) { // Ciclo Infinito

    ledcWrite(pwmChannel, ajuste_pwm); // Ajusta o sinal PWM com o valor calculado
    Serial.print(ajuste_pwm); // Imprime o valor PWM
    delay(100); // Aguarda 100ms antes de atualizar novamente
}
}

void setup() {
  Serial.begin(115200); // Inicializacao Comunicacao Serie 115200 baud rate
  while (!Serial); // Espera até o barramento Série estiver pronto

  setup_wifi(); // Conecta ao WiFi antes de criar as tasks

  // Inicializacao da comunicacao CAN a 500 kbps
  CAN.setPins(RX_GPIO_NUM, TX_GPIO_NUM);
  if (CAN.begin(500E3)) {
    Serial.println("Iniciou Comunicação CAN com SUCESSO");
  } else {
    Serial.println("Ocorreu algum erro ao iniciar a comunicação CAN");
    while (1); // Cancela a comunicacao CAN caso ocorra alguma falha
  }

  pinMode(pwmPin, OUTPUT); // Define o pino PWM como saída
    ledcSetup(pwmChannel, freq, resolution); // Configura o canal PWM com frequência e resolução
    ledcAttachPin(pwmPin, pwmChannel); // Associa o pino ao canal PWM
```

```
  // Criando as tasks
  xTaskCreatePinnedToCore(task1, "Task1", 4096, NULL, 1, &Task1, 1); // Leitura LDR por CAN
  xTaskCreatePinnedToCore(task2, "Task2", 4096, NULL, 1, &Task2, 1); // Mapeamento
  xTaskCreatePinnedToCore(mqttTask, "MQTT Task", 6144, NULL, 1, &mqttTaskHandle, 1); // Publicacao dos topicos no broker
  xTaskCreatePinnedToCore(envio, "Task3", 4096, NULL, 1, &Task3, 1); // Envio CAN
  xTaskCreatePinnedToCore(task3, "Task4", 4096, NULL, 1, &Task4, 1); // Ajuste PWM
}
```

Por fim, temos a tarefa responsável pelo ajuste de PWM, que ajusta a intensidade de luz do LED em tempo real. A configuração é semelhante ao SLAVE, apenas com a inclusão do pino de saída para o LED, e todas as tarefas possuem a mesma prioridade atribuídas ao primeiro *core* do ESP32 MASTER.

Figura 17 – Setup Config MASTER

MQTT (Subscriber) e Amazenamento de Dados na InfluxDB

Descrição: Este código liga-se a um broker MQTT e a uma base de dados InfluxDB para armazenar os valores recebidos. Subscreve-se a três tópicos MQTT: "sensor/dados/lum", "sensor/dados/brilho" e "sensor/dados/pwm". Quando recebe uma mensagem, verifica a que tópico pertence, converte o valor recebido para inteiro e guardo-o na base de dados InfluxDB, associando-o a uma tag correspondente (lum, brilho ou pwm). A função subscribe (client) inscreve o cliente nos tópicos e define um callback para processar as mensagens recebidas. O programa permanece ativo através do loop MQTT, garantido que os dados continuam a ser recebidos e armazenados.

Código

```python
import influxdb_client, os, time
from influxdb_client import InfluxDBClient, Point, WritePrecision
from influxdb_client.client.write_api import SYNCHRONOUS

import random

from paho.mqtt import client as mqtt_client

token = "91J8rJEZQJoXqVnZGUxda3oaldNq1jKmFAPHHJCik6hXCdiqJlWT_IeP987r1h2d6L-WVxFC9npXD0ZlHI1mwg=="
org = "teste"
url = "http://localhost:8086"

write_client = influxdb_client.InfluxDBClient(url=url, token=token, org=org)

bucket="sdc"

write_api = write_client.write_api(write_options=SYNCHRONOUS)

broker = 'localhost'
port = 1883
topic = "sensor/dados/lum"
topic_2 = "sensor/dados/brilho"
topic_3 = "sensor/dados/pwm"
# Generate a Client ID with the subscribe prefix.
client_id = f'subscribe-{random.randint(0, 100)}'
# username = 'emqx'
# password = 'public'


```

Figura 18 - Código MQTT(Subscriber) – Armazenamento de Dados InfluxDB

Este código estabelece a ligação ao InfluxDB e ao broker MQTT para receber e armazenar dados. Começa por importar as bibliotecas necessárias e define as credenciais do InfluxDB, incluindo o token de autenticação, organização e URL do servidos. De seguida, cria um cliente InfluxDB e configura o bucket onde os dados serão armazenados. Para a comunicação MQTT, define o endereço do broker (localhost), a porta (1883) e os três tópicos específicos para receber dados. É gerado um client_id para identificar a conexão MQTT. O código prepara o cliente MQTT para a ligação ao broker e para subscrever-se aos tópicos definidos, garantindo que os valores recebidos sejam processados e armazenaos corretamente no InfluxDB.

```python
def connect_mqtt() -> mqtt_client:
    def on_connect(client, userdata, flags, rc):
        if rc == 0:
            print("Connected to MQTT Broker!")
        else:
            print("Failed to connect, return code %d\n", rc)

    client = mqtt_client.Client(mqtt_client.CallbackAPIVersion.VERSION1, client_id)
    # client.username_pw_set(username, password)
    client.on_connect = on_connect
    client.connect(broker, port)
    return client


```

Figura 19 - Código MQTT(Subscriber) – Armazenamento de Dados InfluxDB

É definida a função connect_mqtt(), que cria e configura um cliente MQTT para se conectar ao broker. A função on_connect() verifica o estado da ligação e imprime uma mensagem de sucesso ou erro. De seguida, é criado um cliente MQTT utilizando a versão 1 da API e é atribuído um client_id. O callback on_connect é associado ao cliente. Por fim, o cliente estabelece a conexão com o broker MQTT e é retornado para ser utilizado em outras funções.

```python
def subscribe(client: mqtt_client):
    def on_message(client, userdata, msg):
        if msg.topic == topic:  # Verifique se o tópico é igual ao "sensor/dados/lum"
            print(f"Received {msg.payload.decode()} from {msg.topic} topic")
            lum = int(float(msg.payload.decode()))
            print(lum)
            point = (
                Point("_a")
                .tag("teste", "lum")
                .field("deu", lum)
            )
            write_api.write(bucket=bucket, org="sdc", record=point)

        elif msg.topic == topic_2:  # Verifique se o tópico é igual ao "sensor/dados/brilho"
            print(f"Received {msg.payload.decode()} from {msg.topic} topic")
            brilho = int(float(msg.payload.decode()))
            print(brilho)
            point = (
                Point("_a")
                .tag("teste", "brilho")
                .field("deu", brilho)
            )
            write_api.write(bucket=bucket, org="sdc", record=point)

        elif msg.topic == topic_3:  # Verifique se o tópico é igual ao "sensor/dados/pwm"
            print(f"Received {msg.payload.decode()} from {msg.topic} topic")
            pwm = int(float(msg.payload.decode()))
            print(pwm)
            point = (
                Point("_a")
                .tag("teste", "pwm")
                .field("deu", pwm)
            )
            write_api.write(bucket=bucket, org="sdc", record=point)

    # Inscreve nos três tópicos
    client.subscribe(topic)
    client.subscribe(topic_2)
    client.subscribe(topic_3)

    client.on_message = on_message  # Chama a função de callback apenas uma vez
```

Figura 20 - Código MQTT(Subscriber) – Armazenamento de Dados InfluxDB

É definida a função subscribe(client: mqtt_client), que permite ao cliente MQTT subscrever três tópicos: sensor/dados/lum, sensor/dados/brilho e sensor/dados/pwm. A função on_message() processa as mensagens recebidas e identifica a que tópico pertencem. Dependendo do tópico, o valor da mensagem é convertido para inteiro, impresso no terminal e armazenado na base de dados InfluxDB. Cada valor é guardado na base de dados como um ponto de medição (Point) com uma tag correspondente ao tipo de dado ("lum", "brilho" ou "pwm"). A escrita na base de dados é feira através da função write_api.write(). O cliente MQTT é então subscrito usando client.subscribe() e a função on_message() é definida como callback para processar as mensagens recebidas. Isso garante que sempre que uma nova mensagem for recebida, será automaticamente processada e enviada para a base de dados.

```python
def run():
    client = connect_mqtt()
    subscribe(client)
    client.loop_forever()


if __name__ == '__main__':
    run()
```

Figura 21 - Código MQTT(Subscriber) – Armazenamento de Dados InfluxDB

É definida a função run(), que inicia a comunicação MQTT. Em primeiro lugar é chamada a função connect_mqtt() para estabelecer a ligação com o broker e depois subscribe(client) para subscrever aos tópicos. A função client.çoop_forever() mantém o cliente em execução continua para processar mensagens recebidas. Por fim, a verificação if __name__ == '__main__': garante que run() seja executada apenas quando o script for executado diretamente.

InfluxDB & Grafana

Assim que o script em python esteja a publicar os dados no InfluxDB seremos capazes de visualizar os dados. Para visualizarmos os dados no InfluxDB, é necessário selecionar o bucket, o point, a tag e o field, "sdc", "_a", "teste", "deu", respetivamente. Após a seleção, são apresentadas as escolhas de dados a serem visualizados neste caso o "pwm", "lum" e "brilho".

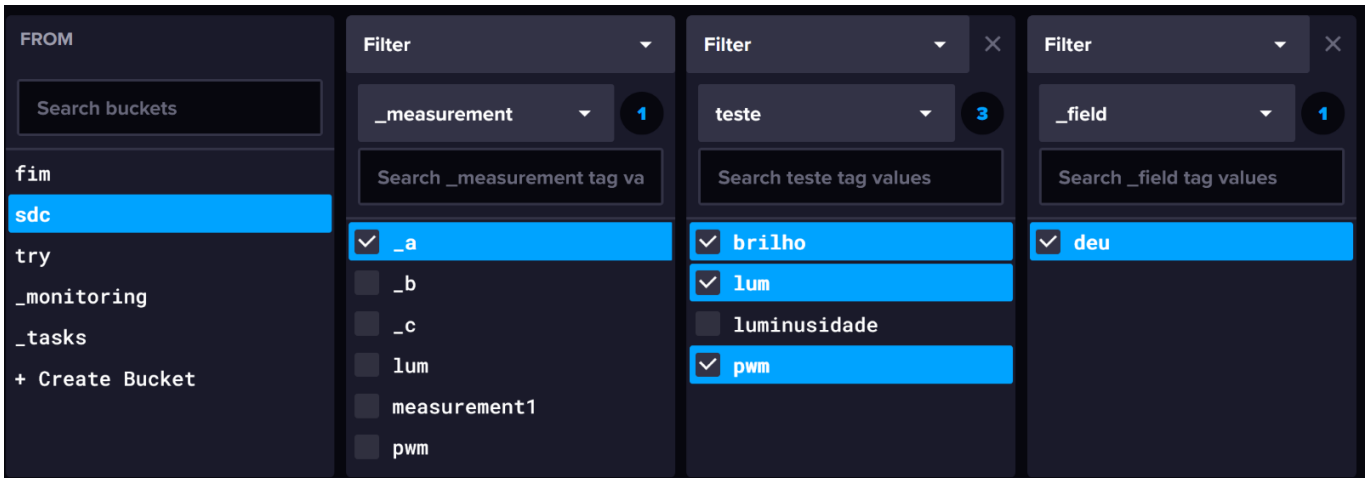


Figura 22– Seleção de Campos

Quando selecionado todos os dados desejados, basta pressionar o botão "Submit" e os dados são apresentados, idealmente de todos os eventos ocorridos há 1 minuto, sendo possível obter amostras num espaço de tempo maior caso se pretenda.

Figura 23 – Visualização de Dados

Criação de Gráficos e Configuração Dashboard em InfluxDB & Grafana

A criação de gráficos no InfluxDB são com recurso à dashboard, à semelhança como realizado anteriormente para visualizar os dados, teremos de selecionar os campos que desejamos para visualizar os dados pretendidos. Após a seleção veremos os dados em um gráfico de eixo Y (valor enviado), e eixo X (tempo). Para alterar e configurar o gráfico como pretendido no canto superior esquerdo existe a hipótese de alterar o tipo de gráfico a ser visualizado.

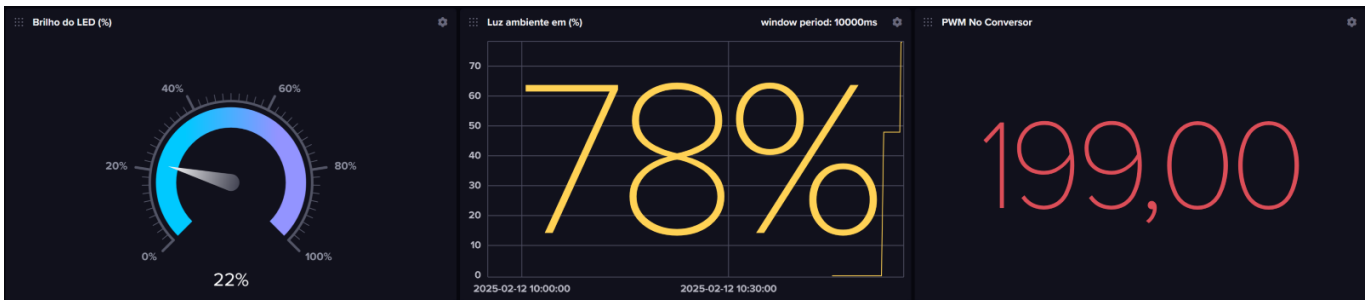


Figura 24 – Dashboard InfluxDB

Quando configurada a dashboard em Influx, podemos efetuar o mesmo processo para o Grafana. Primeiramente foi necessário instalar o Grafana no sistema operacional Linux. Quando devidamente logado no Grafana é necessário criar um data source. Quando criamos o datasource já seremos capazes de criar os gráficos no Grafana. Temos de criar uma dashboard, adicionar uma visualização e selecionar qual datasource pretendemos utilizar. Para utilizarmos os dados do datasource basta ir ao InfluxDB selecionar um dos gráficos criados anteriormente e copiar o código e importar para o Grafana como no exemplo abaixo.

```
from(bucket: "sdc")
  |> range(start: v.timeRangeStart, stop: v.timeRangeStop)
  |> filter(fn: (r) => r["_measurement"] == "_a")
  |> filter(fn: (r) => r["_field"] == "deu")
  |> filter(fn: (r) => r["teste"] == "brilho")
  |> aggregateWindow(every: v.windowPeriod, fn: mean, createEmpty: false)
  |> yield(name: "mean")
```

Figura 25 – Importação dos Gráficos InfluxDB <-> Grafana

Quando tudo estiver configurado, conseguiremos ver a dashboard no Grafana com os dados importados do InfluxDB.

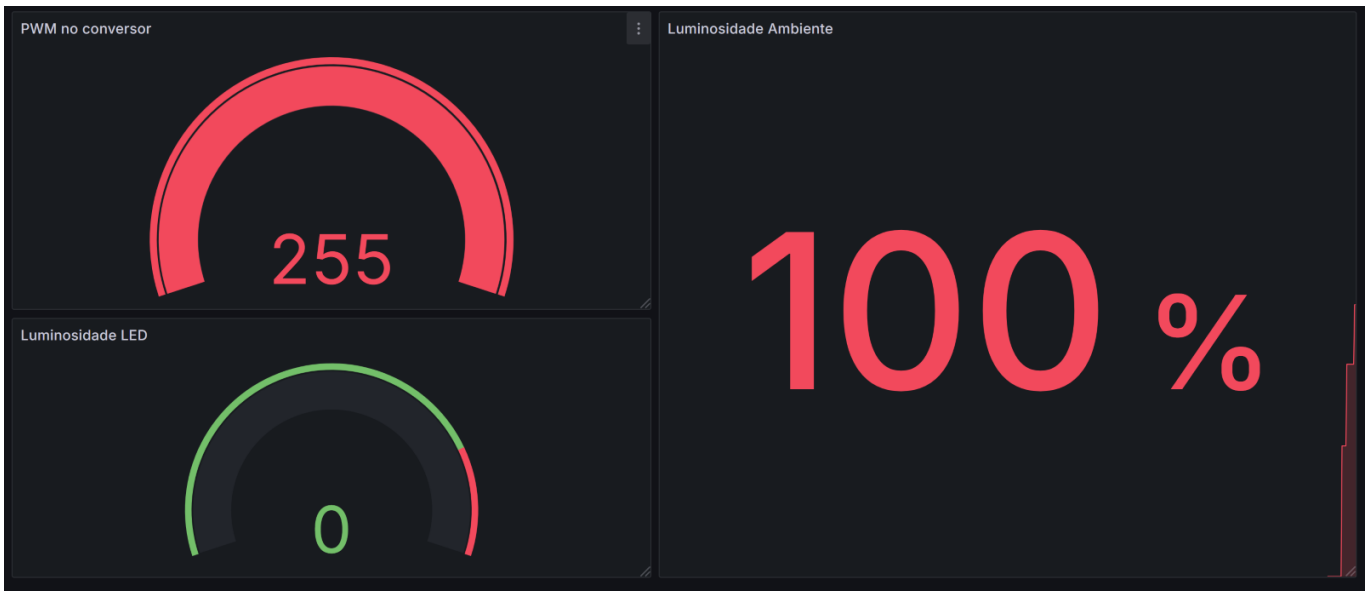


Figura 26 – Dashboard Grafana

Conclusão

Este projeto demonstrou a implementação de um sistema de iluminação inteligente baseado em ESP32, com comunicação CAN e integração com MQTT para monitorização e armazenamento de dados em InfluxDB e Grafana. A arquitetura desenvolvida permitiu uma comunicação eficiente entre os dispositivos, onde o ESP32 Slave realizou a leitura dos sensores e transmitiu os dados ao ESP32 Master, que processou e publicou as informações para um broker MQTT e ajusta a intensidade de luz emitida pelo LED em tempo real.

A utilização de FreeRTOS possibilitou uma gestão eficiente de múltiplas tarefas, garantindo uma resposta rápida e um funcionamento otimizado do sistema. A escolha do protocolo CAN assegurou uma comunicação robusta entre os microcontroladores, enquanto o MQTT permitiu a publicação e receção de dados em tempo real, viabilizando a supervisão remota através de ferramentas como o Grafana.

Além disso, a integração com a base de dados InfluxDB possibilitou o armazenamento e análise histórica dos dados obtidos, facilitando a interpretação das variações de luminosidade e os ajustes realizados pelo sistema. A visualização das métricas através do Grafana forneceu um painel interativo e acessível para monitorizar o comportamento do sistema em tempo real.

Em suma, o projeto atingiu com sucesso os seus objetivos, demonstrando a viabilidade de um sistema distribuído para controlo e supervisão de iluminação. O uso de tecnologias modernas e protocolos de comunicação adequados garantiu uma solução eficiente e escalável, com potencial de aplicação em ambientes reais que necessitem de automação e otimização energética.

Anexos

ESP1 SLAVE

```
//
//    ESP1 SLAVE TRANSMISSOR
//

#include <CAN.h>           // Biblioteca CAN

#define TX_GPIO_NUM   5  // GPIO 5 como pino CAN TX
#define RX_GPIO_NUM   4  // GPIO 3 como pino CAN RX

#define ldr_PIN 2       // GPIO 2 como entrada do LDR

float ldr_valor;  // Variavel que armazena as leituras do LDR
float pwm;        // Variavel que armazena o valor PWM

// Declaracao das Tarefas
TaskHandle_t envioHandle;
TaskHandle_t Task1;
TaskHandle_t Task2;

// =========================
// TASK: ENVIO CAN
// =========================
void envio(void *pvParameters){
  while(1){  // Ciclo Infinito

    // Preparacao da Mensagem Desejada
    uint8_t envio[4];  // Criacao do vetor 4 bytes
    memcpy(envio, &ldr_valor, sizeof(ldr_valor)); // Copia o valor do LDR para o vetor

    CAN.beginPacket(0x12); // Identificacao do pacote de envio ID 0x12
    CAN.write(envio,4);    // Envio dos 4 bytes de dados
    CAN.endPacket();       // Fim da Transmissao CAN

    vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
  }
}

// ======================
// TASK 1: RECECAO CAN
// ======================
void task1(void *pvParameters) {
  while (1) {  // Ciclo Infinito

    int packetSize = CAN.parsePacket();  // Verificacao da rececao de um pacote CAN

    if (packetSize) {  // Caso receba o pacote
      int endereco_mensagem = CAN.packetId(); // Obtem o ID do pacote

      if (endereco_mensagem == 0x10) {  // Caso o ID seja 0x10
        uint8_t recebido[4];  // Criacao do array que armazena os bytes recebidos

        for (int i = 0; i < 4; i++) {
          recebido[i] = CAN.read();  // Leitura dos 4 bytes provenientes do barramento CAN
        }

        memcpy(&pwm, recebido, sizeof(pwm)); // Converte os bytes para float

        Serial.print(pwm);  // Imprime o valor PWM
        Serial.println();
      }
    }

```

```
63.     vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
64.   }
65. }
66.
67. // =====================
68. // TASK 2: Leitura LDR
69. // =====================
70. void task2(void *pvParameters) {
71.   while (1) {  // Ciclo Infinito
72.
73.     ldr_valor = analogRead(ldr_PIN); // Leitura valor analógico do LDR
74.
75.     vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
76.   }
77. }
78.
79. // =====================
80. // SETUP FUNCTION
81. // =====================
82. void setup() {
83.   Serial.begin(115200);  // Inicializacao Comunicacao Serie 115200 baud rate
84.   while (!Serial);       // Espera até o barramento Série estiver pronto
85.
86.   // Configuracao barramento CAN pinos RX e TX
87.   CAN.setPins(RX_GPIO_NUM, TX_GPIO_NUM);
88.
89.   // Inicializacao da comunicacao CAN a 500 kbps
90.   if (CAN.begin(500E3)) {
91.     Serial.println("Iniciou Comunicação CAN com SUCESSO");
92.   } else {
93.     Serial.println("Ocorreu algum erro ao iniciar a comunicação CAN");
94.     while (1);  // Cancela a comunicacao CAN caso ocorra alguma falha
95.   }
96.   // Atribuicao das tarefas FreeRTOS e definidas para o 1º Nucleo
97.   xTaskCreatePinnedToCore(envio, "envio", 2058, NULL, 1, &envioHandle, 1); // Envio CAN
98.   xTaskCreatePinnedToCore(task1, "Task1", 4096, NULL, 1, &Task1, 1); // Rececao CAN
99.   xTaskCreatePinnedToCore(task2, "Task1", 4096, NULL, 1, &Task2, 1); // Leitura LDR
100. }
101.
102. // ===================
103. // LOOP
104. // ===================
105. void loop() {
106.   vTaskDelete(NULL); // nao se utiliza
107. }
108.
```

ESP2 MASTER

```
//
//    ESP2 Master
//
#include <CAN.h> // Biblioteca CAN
#include <WiFi.h> // Biblioteca WiFi
#include <PubSubClient.h> // // Biblioteca MQTT Client


#define TX_GPIO_NUM   5 // GPIO 5 como pino CAN TX
#define RX_GPIO_NUM   4 // GPIO 3 como pino CAN RX


const int pwmChannel = 0; // canal 0 PWM
const int freq = 5000;  // Frequencia PWM 5kHz
const int resolution = 8;   // Resolucao 8 bits PWM (0<->255)
const int pwmPin = 15;  // Saida PWM Pino 15


float ldr_valor;
float ajuste_pwm;
float luminusidade_percentual;
float brilho_led;


// Configuracao WiFi e MQTT
const char* ssid = "iPhone";           // WiFi SSID
const char* password = "laranjo56";    // WiFi Password
const char* mqtt_server = "172.20.10.5"; // MQTT Broker IP

// Conexao WIFI<->MQTT
WiFiClient espClient;
PubSubClient client(espClient);

// Declaracao das Tarefas
TaskHandle_t mqttTaskHandle;
TaskHandle_t Task1;
TaskHandle_t Task2;
TaskHandle_t Task3;
TaskHandle_t Task4;


void setup_wifi() { // conexao WiFi
  Serial.println();
  Serial.print("Connecting to WiFi...");
  WiFi.mode(WIFI_STA);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.print(".");
  }

  Serial.println("\nWiFi connected");
  Serial.println("IP address: ");
  Serial.println(WiFi.localIP());
}

void reconnect() { // Reconexao ao cliente MQTT caso ocorra falhas
  while (!client.connected()) {
    Serial.print("Attempting MQTT connection...");
    char clientId[24];
    snprintf(clientId, sizeof(clientId), "ESP32Master-%04X", random(0xffff));
    if (client.connect(clientId)) {
      Serial.println("connected");
    } else {
```

```
      Serial.print("failed, rc=");
      Serial.print(client.state());
      Serial.println(" retrying in 5 seconds");
      delay(3000);
    }
  }

}

// =========================
// TASK: PUBLICACAO NO MQTT
// =========================
void mqttTask(void* parameter) {
  client.setServer(mqtt_server, 1883); // Define o servidor MQTT e a porta padrão (1883)

  while (true) {
    if (!client.connected()) { // Verifica se há conexão com o broker MQTT
      reconnect(); // Se não estiver conectado, tenta reconectar, executa reconnect
    }

    client.loop(); // Mantém a conexão MQTT ativa e processa mensagens

    char var_pub[64]; // Array de caracteres para armazenar os dados a serem publicados

    // Publica a luminosidade percentual no tópico "sensor/dados/lum"
    snprintf(var_pub, sizeof(var_pub), "%.2f", luminusidade_percentual);
    client.publish("sensor/dados/lum", var_pub);

     // Publica o brilho do LED no tópico "sensor/dados/brilho"
    snprintf(var_pub, sizeof(var_pub), "%.2f", brilho_led);
    client.publish("sensor/dados/brilho", var_pub);

    // Publica o ajuste de PWM no tópico "sensor/dados/pwm"
    snprintf(var_pub, sizeof(var_pub), "%.2f", ajuste_pwm);
    client.publish("sensor/dados/pwm", var_pub);

    vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
  }
}

// =========================
// TASK1: RECECAO CAN
// =========================
void task1(void *pvParameters) {
  while (1) { // Ciclo Infinito
    int packetSize = CAN.parsePacket(); // Verificacao da rececao de um pacote CAN

    if (packetSize) { // Caso recebe o pacote
      int endereco_mensagem = CAN.packetId(); // Obtem o ID do pacote
      if (endereco_mensagem == 0x12) {  // Caso o ID seja 0x12
        uint8_t recebido[4]; // Criacao do array que armazena os bytes recebidos
        for (int i = 0; i < 4; i++) {
          recebido[i] = CAN.read(); // Leitura dos 4 bytes provenientes do barramento CAN
        }

        memcpy(&ldr_valor, recebido, sizeof(ldr_valor)); // Converte os bytes para float

        Serial.print(ldr_valor); // Imprime o valor LDR
        Serial.println();

      }
    }
    vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
  }
}

// =========================
// TASK2: MAPEAMENTO
```

```
// =========================
void task2(void *pvParameters) {
while (1) { // Ciclo Infinito

    // Converte o valor do sensor LDR para porcentagem de luminosidade
    luminusidade_percentual = (ldr_valor / 4095)*100;

    // Converte o ajuste PWM para brilho do LED em porcentagem (invertido)
    brilho_led = map(ajuste_pwm,0,255,100,0);

    // Recalcula o ajuste de PWM baseado no valor do LDR
    ajuste_pwm = map(ldr_valor,0,4095,0,255);

    vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
  }
}

// =========================
// TASK3: ENVIO CAN
// =========================
void envio(void *pvParameters){
  while(1){ // Ciclo Infinito
  uint8_t envio[4]; // Cria um vetor de 4 posições para armazenar os dados a serem enviados
  memcpy(envio, &ajuste_pwm, sizeof(ajuste_pwm)); // Copia o valor de ajuste PWM para o vetor de envio

  CAN.beginPacket(0x10); // Identificacao do pacote de envio ID 0x12

  CAN.write(envio,4);  // Envio dos 4 bytes de dados

  CAN.endPacket(); // Fim da Transmissao CAN

  vTaskDelay(100 / portTICK_PERIOD_MS); // Task Delay de 100ms
  }

}

// =========================
// TASK4: PWM
// =========================
void task3(void *pvParameters) {
  while (1) { // Ciclo Infinito

    ledcWrite(pwmChannel, ajuste_pwm); // Ajusta o sinal PWM com o valor calculado
    Serial.print(ajuste_pwm); // Imprime o valor PWM
    delay(100); // Aguarda 100ms antes de atualizar novamente
}
}

void setup() {
  Serial.begin(115200); // Inicializacao Comunicacao Serie 115200 baud rate
  while (!Serial); // Espera até o barramento Série estiver pronto

  setup_wifi(); // Conecta ao WiFi antes de criar as tasks

  // Inicializacao da comunicacao CAN a 500 kbps
  CAN.setPins(RX_GPIO_NUM, TX_GPIO_NUM);
  if (CAN.begin(500E3)) {
    Serial.println("Iniciou Comunicação CAN com SUCESSO");
  } else {
    Serial.println("Ocorreu algum erro ao iniciar a comunicação CAN");
    while (1); // Cancela a comunicacao CAN caso ocorra alguma falha
  }

  pinMode(pwmPin, OUTPUT); // Define o pino PWM como saída
    ledcSetup(pwmChannel, freq, resolution); // Configura o canal PWM com frequência e resolução
    ledcAttachPin(pwmPin, pwmChannel); // Associa o pino ao canal PWM

  // Criando as tasks
```

```
202.   xTaskCreatePinnedToCore(task1, "Task1", 4096, NULL, 1, &Task1, 1); // Leitura LDR por CAN
203.   xTaskCreatePinnedToCore(task2, "Task2", 4096, NULL, 1, &Task2, 1); // Mapeamento
204.   xTaskCreatePinnedToCore(mqttTask, "MQTT Task", 6144, NULL, 1, &mqttTaskHandle, 1); // Publicacao dos
topicos no broker
205.   xTaskCreatePinnedToCore(envio, "Task3", 4096, NULL, 1, &Task3, 1); // Envio CAN
206.   xTaskCreatePinnedToCore(task3, "Task4", 4096, NULL, 1, &Task4, 1); // Ajuste PWM
207. }
208.
209.
210. void loop() {
211.   vTaskDelete(NULL); // nao se utiliza
212. }
213.
```

Armazenamento de Dados InfluxDB

```
 1. import influxdb_client, os, time
 2. from influxdb_client import InfluxDBClient, Point, WritePrecision
 3. from influxdb_client.client.write_api import SYNCHRONOUS
 4.
 5. import random
 6.
 7. from paho.mqtt import client as mqtt_client
 8.
 9. token = "9lJ8rJEZQJoXqVnZGUxda3oaldNq1jKmFAPHHJCik6hXCdiqJlWT_IeP987r1h2d6L-WVxFC9npXD0ZlHI1mwg=="
10. org = "teste"
11. url = "http://localhost:8086"
12.
13. write_client = influxdb_client.InfluxDBClient(url=url, token=token, org=org)
14.
15. bucket="sdc"
16.
17. write_api = write_client.write_api(write_options=SYNCHRONOUS)
18.
19. broker = 'localhost'
20. port = 1883
21. topic = "sensor/dados/lum"
22. topic_2 = "sensor/dados/brilho"
23. topic_3 = "sensor/dados/pwm"
24. # Generate a Client ID with the subscribe prefix.
25. client_id = f'subscribe-{random.randint(0, 100)}'
26. # username = 'emqx'
27. # password = 'public'
28.
29.
30. def connect_mqtt() -> mqtt_client:
31.     def on_connect(client, userdata, flags, rc):
32.         if rc == 0:
33.             print("Connected to MQTT Broker!")
34.         else:
35.             print("Failed to connect, return code %d\n", rc)
36.
37.     client = mqtt_client.Client(mqtt_client.CallbackAPIVersion.VERSION1, client_id)
38.     # client.username_pw_set(username, password)
39.     client.on_connect = on_connect
40.     client.connect(broker, port)
41.     return client
42.
43.
44. def subscribe(client: mqtt_client):
45.     def on_message(client, userdata, msg):
46.         if msg.topic == topic:  # Verifique se o tópico é igual ao "sensor/dados/lum"
47.             print(f"Received {msg.payload.decode()} from {msg.topic} topic")
48.             lum = int(float(msg.payload.decode()))
49.             print(lum)
50.             point = (
```

```
                Point("_a")
                .tag("teste", "lum")
                .field("deu", lum)
            )
            write_api.write(bucket=bucket, org="sdc", record=point)

        elif msg.topic == topic_2:  # Verifique se o tópico é igual ao "sensor/dados/brilho"
            print(f"Received {msg.payload.decode()} from {msg.topic} topic")
            brilho = int(float(msg.payload.decode()))
            print(brilho)
            point = (
                Point("_a")
                .tag("teste", "brilho")
                .field("deu", brilho)
            )
            write_api.write(bucket=bucket, org="sdc", record=point)

        elif msg.topic == topic_3:  # Verifique se o tópico é igual ao "sensor/dados/pwm"
            print(f"Received {msg.payload.decode()} from {msg.topic} topic")
            pwm = int(float(msg.payload.decode()))
            print(pwm)
            point = (
                Point("_a")
                .tag("teste", "pwm")
                .field("deu", pwm)
            )
            write_api.write(bucket=bucket, org="sdc", record=point)

    # Inscreve nos três tópicos
    client.subscribe(topic)
    client.subscribe(topic_2)
    client.subscribe(topic_3)

    client.on_message = on_message  # Chama a função de callback apenas uma vez




def run():
    client = connect_mqtt()
    subscribe(client)
    client.loop_forever()


if _name_ == '_main_':
    run()

```