Projeto Final: Sistemas Digitais 2
Controle e monitoramento de uma estufa com *FreeRTOS*

Lucas Givaldo dos Santos Torres
Departamento de Eletrônica e Sistemas
Universidade Federal de Pernambuco
Recife, Pernambuco
lucas.givaldo@ufpe.br

***Abstract*—Relatório do Projeto Final da Disciplina Sistemas Digitais 2 (ME587), 2024.1, do Curso de Engenharia Mecânica do Centro de Tecnologia de Geociências da Universidade Federal de Pernambuco.**

***Professor*— José Rodrigues de Oliveira Neto.**

I. Introdução

Este projeto foi feito sobre a estrutura e dispositivos utilizados em uma monografia no Departamento de Eletrônica e Sistemas da UFPE (DES) [1], visando reestruturação de código por meio de um sistema operacional em tempo real, FreeRTOS [2].

A estrutura em questão é uma estufa, Figura 1, feita para agir como uma estufa normal, mas possuindo maior controle das condições que permitem a manutenção da vida em seu interior, além de permitir constante monitoramento remoto dos sensores que auxiliam no controle de suas operações.

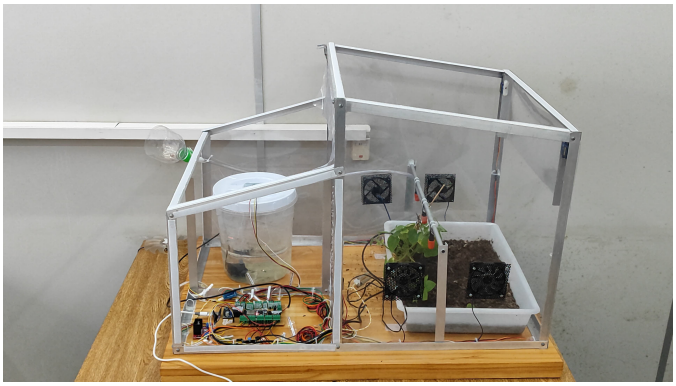

Fig. 1. Estufa montada, vista completa.

A reestruturação permite maior liberdade de desenvolvimento, compartimentando responsabilidades e simplificando a manutenção do código fonte. Combinando isso à alta capacidade de processamento do microcontrolador utilizado, a ESP32 [3], torna o funcionamento dos sensores mais assíncrono e eficiente. Mais sobre esses ganhos pode ser explicado uma vez que o sistema for devidamente apresentado.

O intuito final é integrar a estrutura a uma rede de nodos sensoreados pelo grupo de pesquisa de processamento digital de sinais no DES, junto ao projeto da rede *LORAWAN* [4], que utilizará uma estrutura de *hardware* similar à apresentada na estufa.

II. Componentes principais

Apesar do sistema funcionar plenamente em outros microcontroladores programáveis com a linguagem Arduíno [5], a ESP32 foi escolhida por sua acessibilidade, sua capacidade de armazenamento e por possuir wi-fi embutida, simplificando parte do projeto. Para o funcionamento automático da estufa, no projeto inicial foram definidos sensores, monitorando grandezas importantes do ambiente, e, em sincronia com esses sensores, atuadores, com intuito de manter as condições ideais para o crescimento de vida na estrutura. Os sensores escolhidos foram:

- DHT11 [6]: Umidade e temperatura no ar;
- DS18B20 [7]: Temperatura no solo;
- BH1750 [8]: Luminosidade, interna e externa;
- HC-SR04 [9]: Sensor ultrassônico, para verificar a quantidade de água no reservatório;
- Sensor de umidade no solo S12 [10]: Sonda resistente a corrosão e comparador HD-38.

Os sensores foram escolhidos visando preço e acessibilidade, todos estão entre os mais acessíveis no mercado de eletrônicos no ano em que este projeto foi escrito, 2024. Em especial, o sensor de umidade no solo possui uma opção mais barata e acessível, que usa o LM393 [11] como comparador, e foi descartado por sua sonda ser facilmente corroída com o tempo.

Havendo os parâmetros a serem monitorados, sob os mesmos critérios, foram selecionados atuadores para o controle das variáveis do ambiente:

- Bomba d'água;
- Fita LED RGB Endereçável [12]: 1 metro de fita, 60 diodos;
- 2 servo-motores [13]: Para controlar a "janela" da estufa e, consequentemente, a passagem de ar;

- 5 ventoinhas de 5v [14]: Uma para os componentes, 4 para controlar a passagem de ar na estufa.

Outros componentes fazem parte do sistema, mas serão melhor abordados quando a fabricação da placa estiver em questão.

III. Funcionamento do sistema

Tendo os componentes principais disponíveis, o sistema foi desenvolvido em uma *protoboard*, uma plataforma que facilita a conexão de componentes sem necessidade de solda.

Levando em consideração as especificações de cada componente, foi possível definir suas respectivas funções para aquisição de dados e, similarmente, funções de controle para os atuadores os atuadores, reutilizando, em boa parte, o código do material original.

A. Preparação do ambiente de trabalho

Para começar a editar o projeto, a linguagem *Arduino* tem suporte para diversas plataformas e ambientes de desenvolvimento, mas o utilizado foi a *Arduino IDE 2*, que possui melhor compatibilidade com a linguagem por ter sido criada com esse intuito. A instalação é feita similar mente em sistemas operacionais *Unix* e *Windows*, sendo apenas necessário executar o instalador e ajustar as dependências de acordo com as instruções fornecidas. Mais sobre a instalação do *Arduino IDE* pode ser encontrado na documentação da linguagem [15].

Após a instalação do ambiente de desenvolvimento integrado (IDE), foi realizada a instalação das bibliotecas referentes a cada um dos componentes utilizados e a instalação da plataforma ESP32. Para a instalação da ESP32, Figura 2 há uma página dedicada na documentação da *Espressif* [16], ensinando como adicionar um repositório com placas da empresa ao gerenciador do ambiente de desenvolvimento.

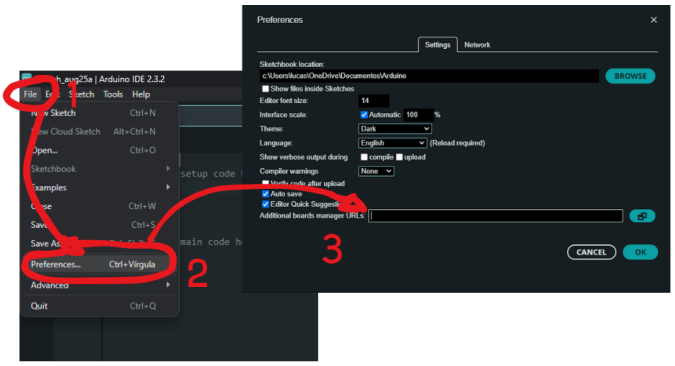

Fig. 2. Adicionando as placas *Espressif*

Após encontrar o tópico *Aditional board managers*, deve ser inserido o link *https://espressif.github.io/arduino-esp32/package_esp32_index.json*, adicionando as placas da *Espressif* ao gerenciador do *Arduino*. Agora, no gerenciador, Figura 3, devem ser instaladas as placas.

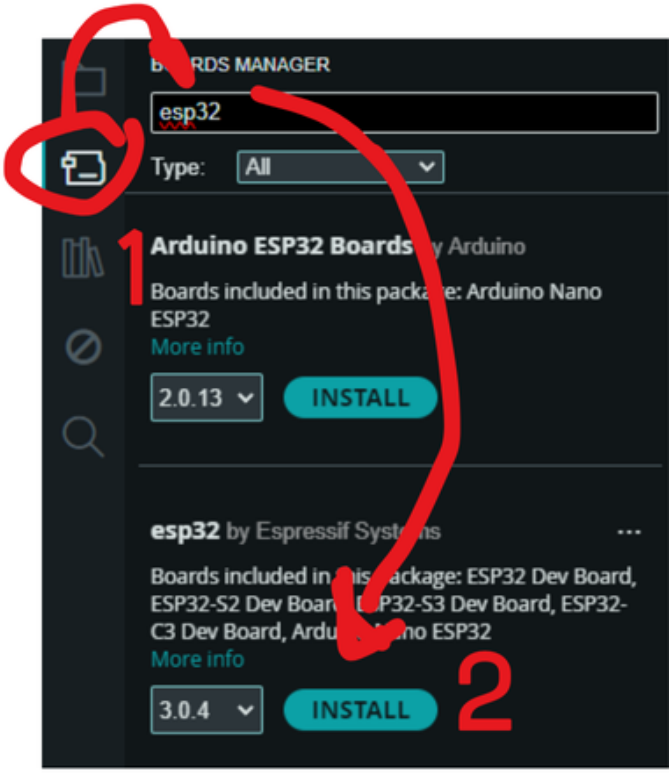

Fig. 3. Instalando as placas *Espressif*.

Havendo instalado as placas, por último deve ser conectado ao computador o microcontrolador para que o ambiente possa identificar a porta e a placa possa ser selecionada. A seleção da porta pode ser feita posteriormente, não havendo diferença no passo a passo descrito na Figura 4.

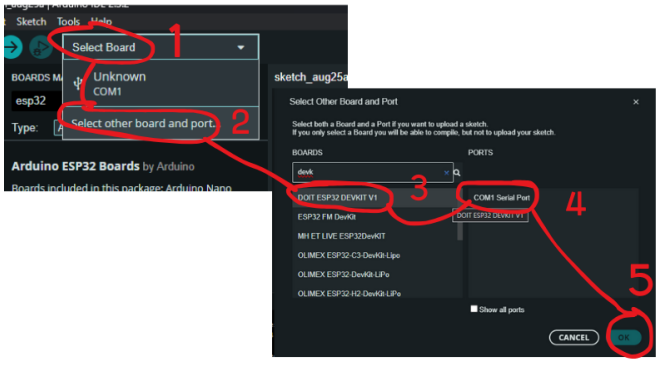

Fig. 4. Selecionando placa *DOIT ESP32 DEVKIT V1* e sua porta.

Quanto às bibliotecas, todas estão disponíveis pelo gerenciador de bibliotecas da IDE e as versões utilizadas foram:

- Adafruit NeoPixel (ver: 1.12.2, externa. **Não atualizar**): Controle da fita de LEDs;
- ESP32Servo (ver: 3.0.5): Controle dos servo-motores;

- DallasTemperature(DS18B20) (ver: 3.9.0): Sensoriamento da temperatura do solo;
- DHT sensor library(DHT11) (ver: 1.4.6): Sensoriamento da umidade e temperatura do ar;
- BH1750 (ver: 1.3.0): Luxímetros;
- NTPClient (ver: 3.1.0, externa. **Não atualizar**): Operações com o tempo atual.

A instalação de todas as bibliotecas é direta, Figura 5, não necessitando de pesquisa em fontes externas. Para buscar versões anteriores de cada biblioteca, o gerenciador oferece uma seleção ao lado do botão de instalação.

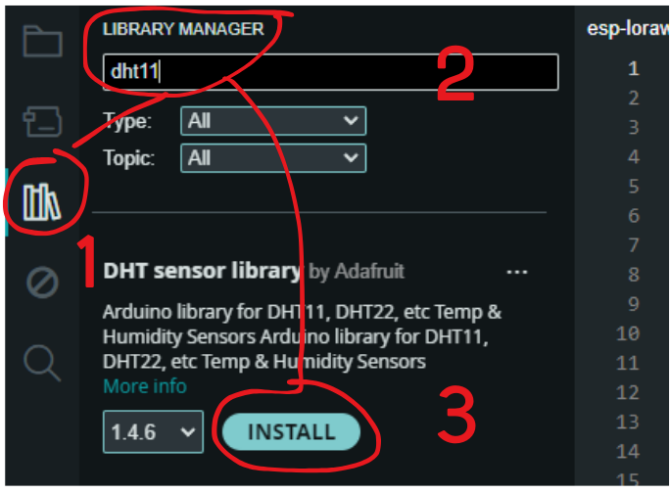

Fig. 5. Instalando bibliotecas extras de maneira convencional.

Apesar de ser encontrado no gerenciador, a versão do NTPClient utilizada é uma bifurcação da versão encontrada no gerenciador que permite ao código acesso, separadamente, a algumas partes da data [17]. O mesmo se aplica para a biblioteca *Adafruit NeoPixel*, que possui um erro na alocação de memória [18] impossibilitando seu uso em tarefas com a Esp32. Na documentação do *Arduino* é possível encontrar uma página ensinando o usuário a adicionar bibliotecas externas com um arquivo compactado, Figura 6, por isso, no repositório do projeto segue a versão utilizada das bibliotecas em questão.

Após a preparação completa do sistema, o primeiro passo para o desenvolvimento é a inclusão das bibliotecas utilizadas, Figura 7. A o ambiente de desenvolvimento resume a inclusão de bibliotecas utilizadas no sistema, então a "Arduino.h" e a "freeRTOS.h", esta em específico para desenvolvimento com Esp32, já estão inclusas.

B. Estrutura

1) Criação de tarefas: Quanto à estrutura do código, na inicialização são feitas as operações de comunicação rotineiras, são definidas as variáveis gerais do sistema, bem como funções de execução única e as tarefas. Na documentação do *FreeRTOS* [2] é possível encontrar uma página descrevendo os primeiros passos com o sistema

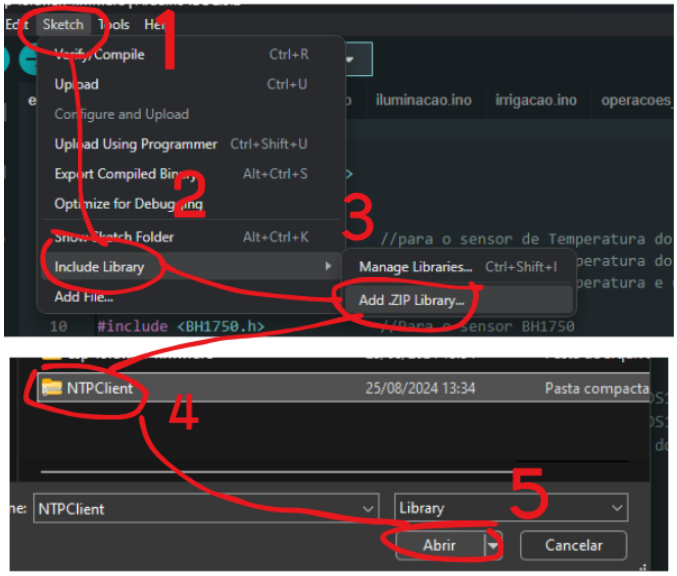

Fig. 6. Instalando bibliotecas externas.

```
//Bibliotecas
#include <WiFi.h>
#include <WiFiClientSecure.h>
#include <HTTPClient.h>
#include <ArduinoJson.h>
#include <OneWire.h>              //para o sensor de Temperatura do Solo (DS18B20)
#include <DallasTemperature.h>    //Para o sensor de Temperatura do Solo (DS18B20)
#include <DHT.h>                  //Para o sensor de temperatura e umidade do Ar (DHT11)
#include <Wire.h>                 //Para o sensor BH1750
#include <BH1750.h>               //Para o sensor BH1750
#include <ESP32Servo.h>           //Para o servo motor
#include <Adafruit_NeoPixel.h>    //Para o LED enderecavel WS2812B
#include <NTPClient.h>            //Para socilitar hora do servido NTP
#include <EEPROM.h>
```

Fig. 7. Inclusão das bibliotecas instaladas.

operacional. As tarefas, *Tasks*, são objetos do *FreeRTOS* que permitem a execução paralela de comandos, sendo nesse caso utilizadas por aumentarem o encapsulamento e facilitarem a organização de responsabilidades. Dentro de uma tarefa podem ser declaradas variáveis específicas, funções de execução única e um ciclo de repetição indeterminada, "*loop infinito*", Figura 8, sendo estruturas similares à macroestrutura do Arduíno comum, *"setup-loop"*.

```
void testeTask(void* parametro);

static TaskHandle_t testeHandler = NULL;

void setup() {

  xTaskCreatePinnedToCore(testeTask, "Teste", 2000, NULL, tskIDLE_PRIORITY, &testeHandler, 0);

void testeTask(void* parametro){
  // Zona sem repetições

  while(true){ // Ciclo infinito

  }
}
```

Fig. 8. Configuração e estrutura de uma tarefa ordinária.

Quanto aos parâmetros da função *xTaskCreatePinnedToCore*, são referentes, em ordem, a:

1) *pxTaskCode*: Função que representa a tarefa e define seu comportamento;
2) *pcName*: Nome dado à tarefa;
3) *usStackDepth*: Quantidade de bytes alocados para a tarefa;
4) *pvParameters*: Parâmetros passados à função em *pxTaskCode*;
5) *uxPriority*: Prioridade de execução da tarefa, crescente conforme a prioridade. Nesse projeto, todas as tarefas foram definidas para a prioridade *tskIDLE_PRIORITY*, palavra chave do sistema que define prioridade 0;
6) *pvCreatedTask*: Objeto *handler* da tarefa. Responsável por controlar seu comportamento geral, mas não será utilizado;
7) *xCoreID*: Identificação do núcleo utilizado pela ESP32. Nesse projeto, todas as tarefas estão fixas ao primeiro núcleo (0);

Em toda criação de tarefa devem ser escolhidas a prioridade e a quantidade de memória alocada. A quantidade de memória alocada foi definida com a função *uxTaskGetStackHighWaterMark(&handlerDaTarefa)* que retorna a quantidade de memória restante na tarefa. O valor retornado não é precisamente o valor de bytes livres, visto que um espaço de memória não escrito dentro da tarefa não é contado até que o mesmo seja utilizado, logo, por meio de avaliações consecutivas, foi posto a todas as tarefas, cerca de 500 bytes de folga com relação ao valor retornado, permitindo o funcionamento ininterrupto.

2) Mutex: Para reduzir as falhas durante as comunicações entre tarefas foi criado um *mutex* [2], Figura 9, uma estrutura de sistemas em tempo real que permite alterações em zonas críticas por diferentes tarefas de maneira segura. Sendo assim, após qualquer medição, o *mutex* é sinalizado pela tarefa, os dados da variável global são alterados, depois o *mutex* é "devolvido".

Antes do setup()

```
40  // Mutex de dados
41  static SemaphoreHandle_t mutex;
```

Dentro do setup()

```
87    // Criação de mutex
88    mutex = xSemaphoreCreateMutex();
```

Fig. 9. Configuração do *Mutex*.

A variável global criada é uma estrutura, *DadosSalvos*, Figura 10, que armazena os dados obtidos em todas as tarefas. Seu único acesso em código é por meio das funções *xSemaphoreTake(mutex, portMAX_DELAY);* e *xSemaphoreGive(mutex, portMAX_DELAY);*. Ambas sinalizam o uso da variável e o argumento *portMAX_DELAY* permite à espera pelo direito ao *mutex* durar quanto for necessário.

```
// Dados bloqueados pelo mutex
struct DadosSalvos {
  uint8_t umidadeDoAr;
  uint8_t umidadeDoSolo;
  float temperaturaDoAr;
  float temperaturaDoSolo;
  uint16_t luminosidadeExterna;
  uint16_t luminosidadeInterna;
};

static DadosSalvos pacote = { 0, 0, 0, 0, 0, 0 };
```

Fig. 10. Criação da estrutura do pacote de dados.

3) Estrutura de todas as Tarefas: Todas as tarefas seguem a mesma estrutura, facilitando o desenvolvimento e adição de novas funcionalidades ao sistema. Cada tarefa tem um arquivo com sufixo ".ino" referente ao seu código fonte, Figura 11. Quando um projeto *Arduino* é criado, o ambiente de desenvolvimento o armazena numa pasta com o seu nome, sem o sufixo, e todos os demais arquivos ".ino" nessa pasta são compilados em ordem alfabética, exceto o arquivo com nome igual à pasta, que é executado primeiro por ser o principal.

eertos.ino | comunicacao.ino | iluminacao.ino | irrigacao.ino | ventilacao.ino | tarefaTeste.ino

```
// Configuração
TipoDeSensor objetoSensor;
TipoDeAtuador objetoAtuador;

// Sensor
TipoDeDado obtemTeste() {

return dado;
}

// Atuador
void controleTeste(TipoDeComando comando) {

return;
}

//Tarefa
void testeTask(void* parametros) {

  while (true) {
  }
}
```

Fig. 11. Estrutura de uma tarefa

Os arquivos possuem, no topo, as funções controle e sensoriamento do sistema e, abaixo, a função de tarefa, Figura 12. Nela estão divididas as responsabilidades em "*obter*(parâmetro a ser obtido)", "*controla*(Atuador a ser controlado)" e "(nome do sistema)Task". Configurações sobre o sistema podem ser realizadas tanto no topo do arquivo quanto imediatamente antes do ciclo infinito da função da tarefa. Essa escolha influencia na alocação de memória. Declarar uma variável fora da tarefa e atribuir

seus valores dentro da mesma, evita o uso da memória destinada ao objeto *Task*.

```
//Tarefa
void testeTask(void* parametros) {
  // Configuração
  objetoAtuador = Atuador();
  objetoSensor = Sensor();

  // Variável
  TipoDeValor valorMedido = objetoSensor.medir();

  TickType_t temporizador = xTaskGetTickCount();
  while (true) {
    // Medição
    valorMedido = objetoSensor.medir();

    /*
     * lógica (Comparação, requisito)
     */
    controleTeste(resultadoDaLógica);


    // Armazena dados na variável global
    xSemaphoreTake(mutex, portMAX_DELAY);
    pacote.valorSensor = valorMedido;
    xSemaphoreGive(mutex);

    vTaskDelayUntil(&temporizador, periodoEntreExecução_MS/portTICK_PERIOD_MS);
  }
```

Fig. 12. Estrutura da função *callback*.

Além disso, nas definições do arquivo principal, há uma variável booleana, TESTE, que define se o sistema deve executar enviando escrevendo a consequência de seus comandos por serial. Por exemplo: Com a variável TESTE ativada, a função *obterIluminacaoExterna* vai escrever "Nível de Iluminância no Exterior da Estufa: (**ilumincacao**) lx" e, em seguida, *controleLed* vai escrever "Ligando (ou Desligando) Led na Luz Branca".

C. Comunicação

O arquivo que contém a lógica responsável pela comunicação, *comunicacao.ino*, possui 3 funções, *erase_client_id*, *primeiro_acesso*, *envio_task*. Além disso, no topo, estão definidos os endereços para aquisição de identificador, troca de dados e a chave de acesso, Figura 13.

```
1   const char* iDADDR = "h  // Requisição de ID
2   const char* dadosADDR =  // Troca de dados
3   const char* pass = "U2F  // Chave de acesso

 8 > void erase_client_id() {…
17   }

23 > void primeiro_acesso() {…
62   }

72 > void envioTask(void* parametro) {…
130 }
```

Fig. 13. Estrutura do arquivo de comunicação.

O processo de envio de informação ao banco de dados, parte necessária para a análise remota, precisa passar por uma etapa de segurança antes de tornar a comunicação periódica. Uma vez concluída a verificação, o dispositivo tem acesso permitido ao tópico de envio do servidor.

Em suma, a comunicação segue:

1) Requisição de identificador (ID);
2) Armazenamento do ID;
3) Envio permitido.

Com o uso de uma chave aleatória fornecida pelo servidor, o código executa na inicialização uma verificação no endereço de memória *EEPROM* [19], no *setup*, onde a identificação do dispositivo deveria estar, comparando-a ao valor nulo, Figura 14. Vale ressaltar que o microcontrolador ESP32 não possui esse tipo de memória mas a extensão que permite sua programação com a linguagem Arduíno aloca a memória resistente, que não é apagada após a reinicialização do sistema, em memória *flash*, de maneira prática adicionando ao desenvolvimento o comando *"EEPROM.commit()"* no fim das instruções de escrita.

```
66   EEPROM.begin(EEPROM_SIZE);
67
81   //Avalia identificação da placa
82   EEPROM.get(EEPROM_ADDR, clientId);
83   while (clientId == 65535) {
84     primeiro_acesso();
85     EEPROM.get(EEPROM_ADDR, clientId);
86   }
87
```

Fig. 14. Código preparando a comunicação na função *setup*.

Caso nenhum identificador seja encontrado o aplicativo executa a função *primeiro_acesso*, que envia uma requisição com a chave esperando um novo ID para escrever na memória resistente e prosseguir com o funcionamento normal, Figura 15. Durante a comunicação são utilizadas as bibliotecas *HTTPClient* e *WiFiClient* para permitir a criação e envio de mensagens.

Uma vez ajustados os parâmetros para a comunicação, periodicamente, atualmente um período de 20 minutos, o sistema coleta os dados do *mutex*, os associa a um período no tempo, *timestamp*, e ao ID, enviando imediatamente. Caso esse ID seja negado, é apagado da memória e o microcontrolador é reiniciado por meio da função*erase_client_id*. Esse comportamento é definido pela tarefa *envioTask*, Figura 16, que segue à ordem de operação:

1) Define o temporizador e entra no *vTaskDelayUntil*;
2) Atualiza o objeto da biblioteca NTPClient, atualizando assim o tempo;
3) Aciona o *mutex*, cria parte da mensagem referente aos dados;
4) Formata a data conforme o padrão requisitado;
5) Une identificação, dados e tempo, nesta ordem;
6) Efetua o envio, comparando o resultado da requisição para garantir que a identificação está correta.

D. Iluminação

O sistema de iluminação verifica a cada hora, por meio de luxímetros BH1750 as iluminações interna e externa, comparando a iluminação externa com um valor mínimo

```
void primeiro_acesso() {
  Serial.println("Primeiro Acesso");
  uint16_t payload;
  delay(2000);
  // Envio da mensagem
  WiFiClient wifiClient;
  HTTPClient httpClient;

  httpClient.begin(wifiClient, iDADDR);
  httpClient.addHeader("Content-Type", "application/json");
  httpClient.addHeader("x-api-key", pass);

  Serial.println("getting");

  int httpResponseCode = httpClient.GET();  // quebra
  Serial.println("getou");
  if (httpResponseCode == 200) {
    payload = httpClient.getString().toInt();
    EEPROM.put(EEPROM_ADDR, payload);  // debug
    if (EEPROM.commit()) {
      Serial.println("Success!");
    } else {
      Serial.println("Falha");
      return;
    }
  } else {
    Serial.println("Falha na aquisicao do id");
  }
```

Fig. 15. Código do primeiro acesso.

```
void envioTask(void* parametro) {

  TickType_t temporizador = xTaskGetTickCount();  // temporizador

  while (true) {

    vTaskDelayUntil(&temporizador, periEnvio / portTICK_PERIOD_MS);

    char dadoVetorizado[150];
    xSemaphoreTake(mutex, portMAX_DELAY);
    sprintf(dadoVetorizado, "\"%s\":\"%i\",\"%s\":\"%i\"
    xSemaphoreGive(mutex);

  //formatação de tempo
  char dataFormatada[11];
  char horaFormatada[9];

  const char* dataCrua = timeClient.getFormattedDate().c_str();
  strncpy(dataFormatada, dataCrua, 10);
  strncpy(horaFormatada, dataCrua + 11, 8);
  dataFormatada[10] = '\0';
  horaFormatada[8] = '\0';  // Definindo caracter nulo no vetor

  Serial.println("Criando mensagem");

  // Criação da mensagem
  char mensagem[200];
  sprintf(mensagem, "{\"id_placa\":\"%i\",%s,\"data\":\"%s %s\"}", c
```

Fig. 16. Código do primeiro acesso.

e, caso esse valor seja maior que o medido, a fita de LEDs é apagada. As luzes se mantém acesas somente durante o dia para não interferir com o ciclo circadiano das plantas [20]. Uma vez aferidas as luminosidades, a tarefa espera a oportunidade de alterar os valores do *mutex*.

As funções *obterIluminação(...)* utilizam um objeto da classe *BH1750* e seu método *readLightLevel*, que retorna um ponto flutuante que representa o valor em lux da iluminância no sensor em questão. A diferença entre os sensores é definida antes do ciclo infinito da tarefa, onde, para cada um, é chamado o método *begin* com os parâmetros na configuração: *(BH1750::CONTINUOUS_HIGH_RES_MODE, (endereço))*. O valor do endereço é 0x23 para o externo, quando o pino ADR está ligado à referência, e 0x5C para o interno, quando o pino ADR está ligado ao VCC. Isso é parte da comunicação I2C que permite a distinção entre sensores no mesmo barramento.

As operações com LEDs são feitas um a um, em seguida executando o método *show* para aplicar na fita as alterações, Figura 17. A fita é posta em branco com 60% de intensidade por poder exigir muita corrente em seu funcionamento e ser, nessa configuração, o atuador que passa mais tempo ligado.

```
if (estado) {
  //Liga LED no branco
  for (int posicaoLED = 0; posicaoLED < quantidadeLeds; posicaoLED++) {
    leds.setPixelColor(posicaoLED, leds.Color(255, 255, 255));  // Branco
  }
  leds.show();

  if (TESTE) {
    Serial.println("Ligando Led na Luz Branca");
  }
  return;
}
```

Fig. 17. Ativação da fita. A desativação é feita substituindo *leds.Color()* por 0x000000.

E. Irrigação

O sistema de irrigação verifica a cada hora, por meio da sonda a umidade do solo comparando com um valor mínimo e, caso esse valor seja maior que o medido, a bomba d'água é acionada. Na função de acionamento da bomba é verificado, antes do acionamento, o nível da água, impedindo que a mesma continue a operar sem água no reservatório. Após isso, o código entra em *loop* irrigando por 8 segundos, esperando 2, realizando as medições novamente e repetindo até as medições estarem aceitáveis. Ao sair do *loop*, a tarefa espera a oportunidade de alterar o valor do *mutex*.

A função *obterUmidadeSolo* realiza a leitura do pino de entrada do comparador e usa a função *map* para mapear a tensão obtida entre os valores de umidade máxima e mínima definidos no começo do arquivo para uma escala de porcentagem.

A função *obterNivelReservatorio*, Figura 18, usa o sensor de ultrassom para medir o tempo em que uma onda emitida reflete na superfície da água e retorna ao captador. O sinal no pino *TRIGGER* é posto em nível alto por aproximadamente 10 micro segundos, depois em nível baixo, em seguida, por meio da função *pulseIn*, é medido o tempo em que o pino *ECHO* fica em nível alto. O tempo aferido é multiplicado então por 0,034 (velocidade do som em centímetros por micro segundos) e dividido por 2. Em seguida, utilizando as dimensões do reservatório definidas previamente no arquivo, é definido proporcionalmente em código quanta água resta no recipiente.

O controle da bomba, realizado por *controleBombaDagua*, é realizado por uma função que aplica a lógica invertida necessária para a ativação do seu circuito de chaveamento. Sendo assim, para acionar, ou desativar, a bomba, *digitalWrite(bombaDagua_PIN, LOW (ou HIGH)*.

```
long pulse;                 //Variável que armazena o tempo de duração do echo
uint8_t distanciaAtualAgua;  //Variável que armazena o valor da distância, em centímetros, entre o sensor e o nivel de agua atual
int nivelAguaReservatorio;   //Nivel de agua presente no reservatorio em porcentagem

//Aciona o trigger do módulo ultrassônico gerando o pulso de trigger
digitalWrite(triggerUltrassonRx_PIN, HIGH);  //Pulso de trigger em nivel alto
delayMicroseconds(10);                       //duração de 10 micro segundos
digitalWrite(triggerUltrassonRx_PIN, LOW);   //Pulso de trigger em nivel baixo
pulse = pulseIn(echoUltrassonTx_PIN, HIGH);  //Mede o tempo em que o pino de echo fica em nível alto

distanciaAtualAgua = (pulse * 0.034) / 2;  //Valor da distância em centímetros (utilizando velocidade do som em m/micro segundos)
distanciaAtualAgua = distanciaAtualAgua;
if (TESTE) {…
}

if (distanciaAtualAgua > distanciaReservatorioVazio) {
  distanciaAtualAgua = distanciaReservatorioVazio;  //para se caso medir a mais que a distancia maxima do reservatorio (erro)
}

nivelAguaReservatorio = 100 * (1 - ((distanciaAtualAgua - distanciaReservatorioCheio) / (distanciaReservatorioVazio - distanciaReservatorioCheio)));

if (TESTE) {…
}
```

Fig. 18. Passos descritos no texto, como implementados.

F. Ventilação

O sistema de ventilação verifica a cada 30 minutos, por meio do DHT11 a umidade e temperatura do ar, além da temperatura do solo com o DS18B20, comparando a temperatura do ar com um valor máximo e, caso esse valor seja menor que o medido, as 4 ventoinhas são ligadas, assim como os 2 servomotores que abrem a escotilha no topo da estufa. A quinta ventoinha se mantém ligada o tempo inteiro, tendo sido posta para aumentar a passagem de ar na área de controle do sistema, onde o microcontrolador se encontra. Uma vez aferidas as temperaturas e a umidade, a tarefa espera a oportunidade de alterar os valores do *mutex*.

Para o controle do teto, são definidos objetos do tipo *Servo* antes do ciclo infinito na tarefa *ventilacaoTask*. Na função *controleDoTeto*, os pinos definidos são atribuídos aos objetos. Um dos servos é posto a 180° e outro a 0, na configuração atual essa ordem para servo direito e servo esquerdo é o comando de fechar, o inverso, seria o comando de abrir. Essa implementação pode mudar conforme a montagem dos servo motores, mas, estando um frente ao outro, essa configuração persiste.

As ventoinhas são controladas diretamente por um pino digital acionando uma ponte H. Em código, a função *controleVentoinha* define seu pino com HIGH para ligadas e LOW para desligadas.

Para aferir a temperatura do solo, na função *obterTemperaturaSolo*, o objeto da *DallasTemperature* é utilizado com o método *requestTemperatures*, atualizando seus valores. Após isso, a temperatura em celcius pode ser obtida com o método *getTempCByIndex*.

As grandezas obtidas pelo DHT11, são diretamente obtidas com os métodos *readTemperature* e *readHumidity*, não havendo necessidade de um método a parte para seu tratamento.

IV. Estrutura Física

Durante o projeto foi reutilizada, totalmente, a estrutura do antigo sistema [1], isto é, a estrutura se trata de uma estufa com formado de casa em miniatura com sua fundação em madeira e cantos em alumínio. As paredes dessa estrutura são de plástico, tornando apenas a madeira e a terra elementos de peso considerável.

Dentro da estrutura existem 3 ambientes, Figura **??**, em um se concentram as conexões e a central eletrônica, Figura 19, onde se encontram os terminais, ao lado o reservatório de água com a bomba.

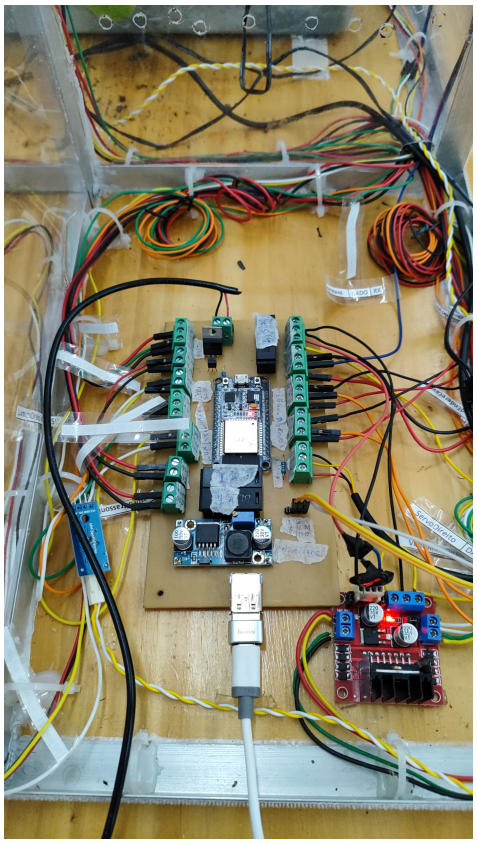

Fig. 19. Ambiente do sistema

A planta escolhida para o teste foi um feijão verde, por ser de simples obtenção manutenção.

V. Estrutura Eletrônica

Uma vez validado, o sistema foi traduzido em um projeto com o *software* Kicad, Figura 20, programa que permite a esquematização de sistemas elétricos e o desenho de placas de circuito impresso (PCIs), levando em consideração os parâmetros de funcionamento ideal descritos em suas respectivas folhas de dados. Mais sobre o funcionamento do programa pode ser encontrado e sua instalação pode ser encontrado nos guias disponíveis em sua página principal [21].

Para a visualização de todos os componentes, algumas bibliotecas desenvolvidas pela comunidade precisaram ser agregadas ao projeto. Todas podem ser substituídas por componentes genéricos com a disposição correta de pinos, apesar disso, o arquivo de projeto do programa já salva uma

cópia das bibliotecas utilizadas no esquemático, garantindo a reprodutibilidade do mesmo posteriormente.

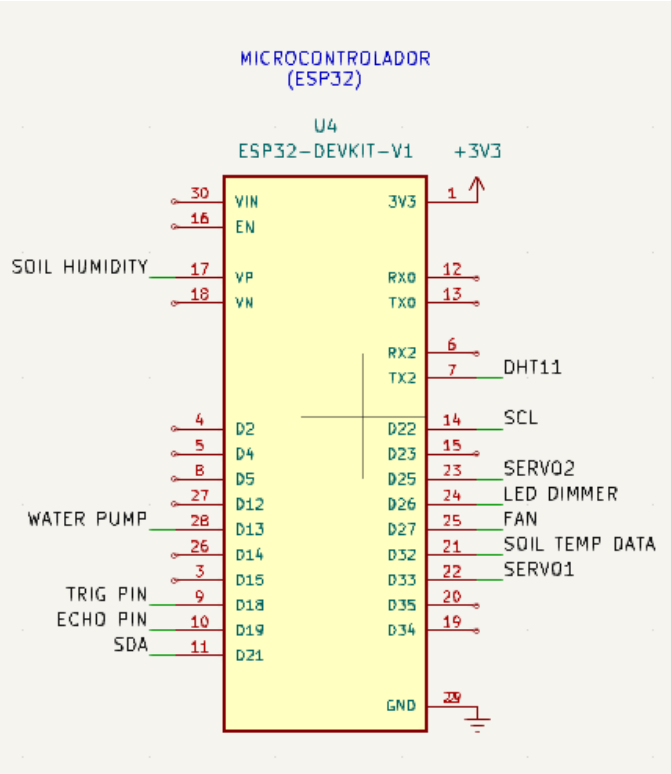

Fig. 20. Diagrama do microcontrolador no *Kicad*.

A. Alimentação

O sistema é alimentado por uma porta USB tipo A, fêmea, por disponibilidade, com intuito de utilizar um adaptador para USB tipo C e assim permitir o uso de carregadores de 5V, que além de possuir alta disponibilidade no mercado estão sendo fabricados com potências mais altas para suprir a demanda de *smartphones* mais novos. Para os sensores e o microcontrolador, esse potencial passa por um regulador *stepdown* [22] que reduz o potêncial para 3,3V, Figura 21.

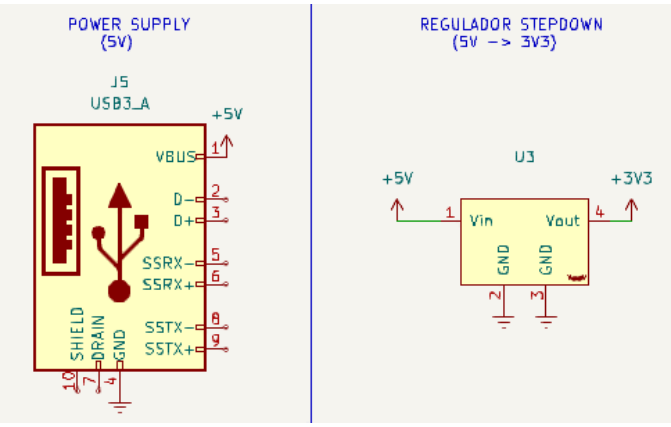

Fig. 21. Alimentação. Entrada USB e regulador, respectivamente.

B. Sensor de Luminosidade

Os luxímetros selecionados se comunicam por meio de barramento I2C [23], permitindo o uso de apenas duas portas dedicadas ambos, SDA e SCL, sendo diferenciados apenas seus endereços, conforme a Figura 22. O endereço (pino ADDR) do luxímetro interno foi ligado ao valor lógico máximo e o pino do sensor externo foi ligado ao valor 0, no plano de referência. Essa diferença, graças ao protocolo, pode ser notada em código, o que permite o simultâneo e discriminado.

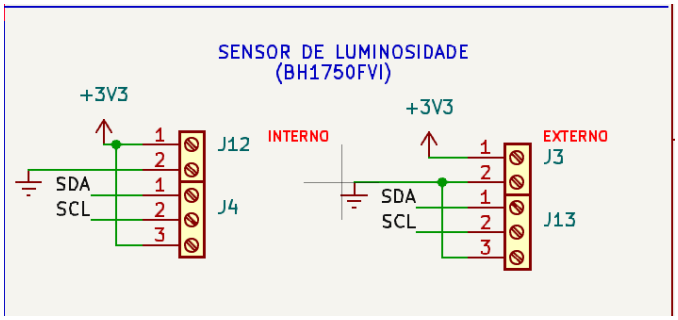

Fig. 22. Circuito dos luxímetros.

C. Sensor de Umidade no Solo

O comparador utilizado pela sonda que afere a umidade do solo, Figura 23 retorna os valores por meio de sinal analógico. Essa configuração permite o uso tanto do aparato usado no projeto, HD-38, quando do, mais simples de adquirir, LM393 com a sonda vulnerável a corrosão.

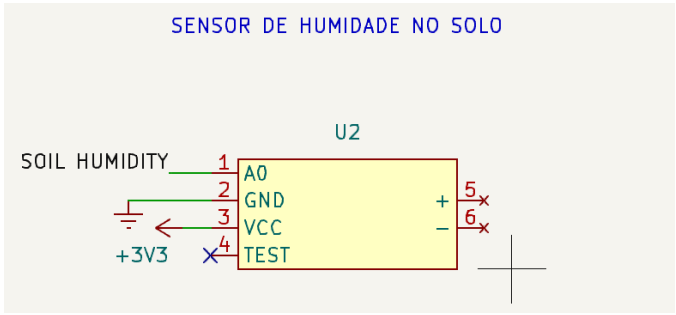

Fig. 23. Comparador de umidade no solo.

D. Sensor de Temperatura do Solo

O sensor de temperatura a prova d'água utilizado, foi ligado conforme sua ficha técnica com um resistor de *pull up*, por segurança, Figura 24.

E. Sensor de Umidade e Temperatura do Ar

O DHT11 possui 4 pinos, sendo um deles não conectado. Na Figura 25, o componentes possui apenas 3 saídas, isso deixa a responsabilidade da seleção correta dos pinos para a montagem. De acordo com a folha de dados [6], a disposição correta é a da Figura 26.

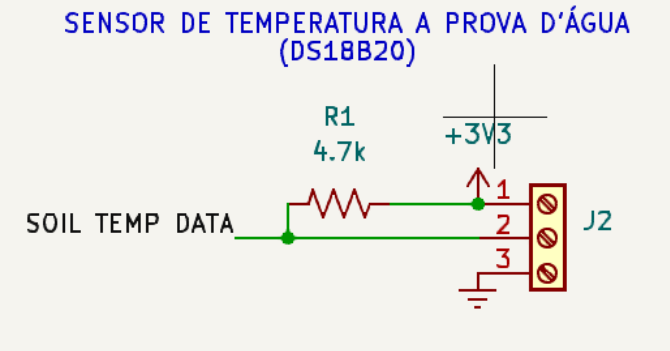

Fig. 24. Temperatura do solo

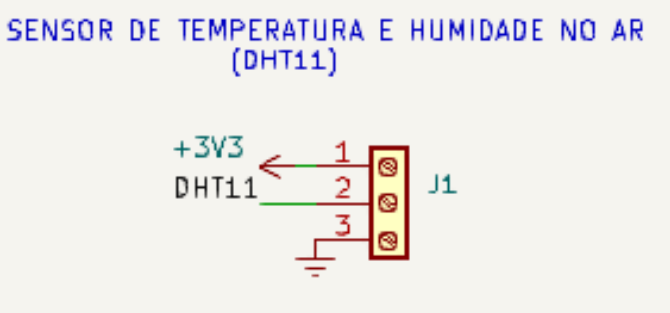

Fig. 25. Sensor de umidade e temperatura do ar.

F. Sensor de Nível

O sensor ultrassônico utilizado para medir o nível utiliza dois pinos, além da alimentação. Conforme a Figura 27, um para o pino *TRIGGER* e outro para o pino *ECHO*

G. Bomba D'água

Conforme a Figura 28, para o acionamento da bomba, foram utilizados dois transistores, um transistor bipolar de junção (TBJ) [25] e um *mosfet* canal N [26], ligados de maneira a fazer uma chave de lógica inversa, isto é, em código, a saída que controla a ativação do dispositivo está sempre fornecendo 3,3V e igualando seu potencial a 0 quando a ativação é necessária. Os resistores foram

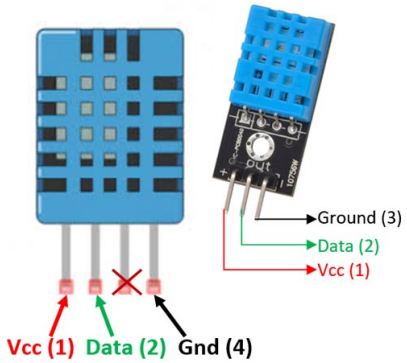

Fig. 26. Disposição dos pinos do DHT11. Fonte: Components101 [24].

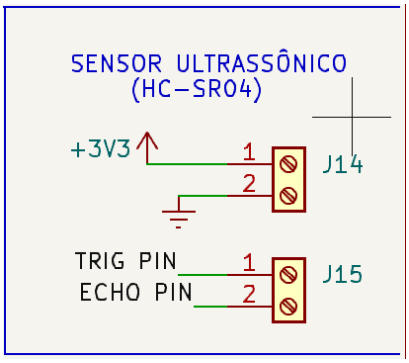

Fig. 27. Sensor ultrassônico.

selecionados de forma a reduzir a corrente transitando o microcontrolador, respeitando os limites do ESP32 [3].

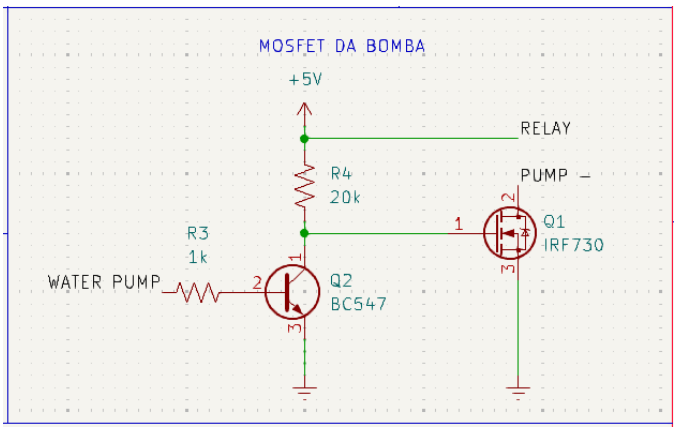

Fig. 28. Sistema de acionamento da bomba d'água.

H. Fita de LEDs

Para a iluminação, os LEDs selecionados necessitam de 3 fios, a alimentação e o *LED DIMMER*, Figura 29, que controla a intensidade e a cor dos diodos individualmente.

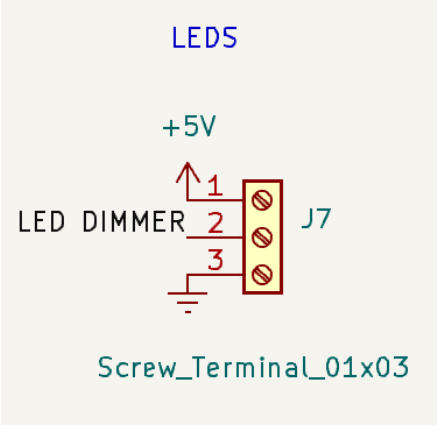

Fig. 29. Saída para a fita de LEDs.

I. Ventoinhas

As ventoinhas são controladas por uma ponte H [27], Figura 30, dispositivo que permite o controle de um motor de corrente contínua, com a disposição conforme a Figura 31.

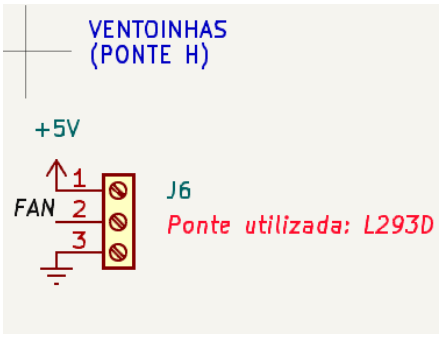

Fig. 30. Saída para a ponte H

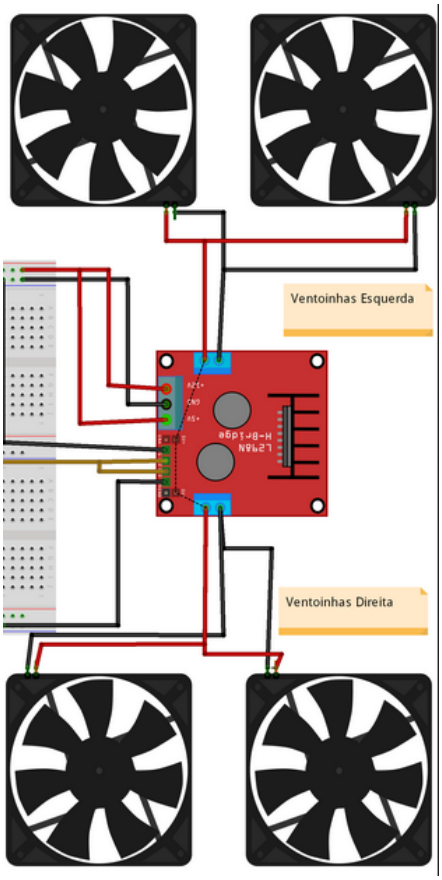

Fig. 31. Ligação da ponte H. Os fios pretos se referem à rede GND, potencial 0, e os fios vermelhos ao potencial de 5V. O fio marrom é ligado ao pino que controla o acionamento. Fonte: TCC de Felime Mororo [1].

J. Fusíveis

Como medida de segurança foram ligados porta-fusíveis aos servo-motores e à bomba d'água32, em respeito a possíveis picos de corrente. Os valores dos fusíveis foram tomados como os valores máximos de corrente vistos nas folhas de dados, sendo um fusível de 1A para cada servo-motor e um de 1,5A para a bomba.

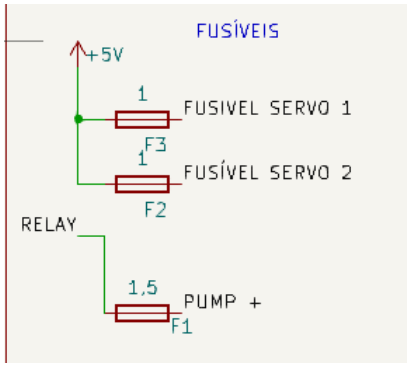

Fig. 32. Fusíveis.

K. Desenho da PCI

Após a conclusão do diagrama elétrico do circuito, sobraram alguns componentes auxiliares à placa para serem definidos. Os terminais que permitem o uso de fios foram selecionados conforme a disponibilidade e preço, sendo selecionados modelos da *Phoenix Contacts* de duas e três entradas, por serem os mais comuns no mercado de prototipação. Graças à pequena diferença e à natureza de baixa precisão da manufatura da placa, tanto os modelos de 5mm [28] [29] de distância entre pinos, quanto os modelos de 5,08mm [30] [31] são aceitáveis. Além deles, foram utilizados pinos terminais de 2,54mm de distância pino a pino, Figura 33, no lugar do comparador do sensor de umidade por sua portabilidade. Alguns comparadores vêm com diferentes disposições de pinos, o uso desses terminais não somente evita o descarte da placa no caso de aquisição de um módulo diferente, como permite o uso de um aparato diferente, como o já mencionado LM393.

Fig. 33. Terminais de 6 pinos. Fonte: Wikipedia [32]

Após a organização do diagrama, o programa permite atualizar a área de roteamento de placas de circuito impresso com base no desenho do diagrama. Essa ferramenta distribui as *footprints*, desenhos planos dos componentes eletrônicos que interagem com o programa, de todos os componentes e indica onde as linhas de rede, artifício do programa que representa as ligações entre dois ou mais componentes, devem estar ligadas.

A zona azul da Figura 34 é o plano de aterramento. Essa configuração de projeto permite que uma das faces condutoras da placa possua um potencial próprio, nesse caso, 0, isso simplifica o roteamento. Maior parte dos componentes precisam de acesso à referencia, logo, o programa identifica a presença do plano referencial e torna os furos, comumente chamados de *pads*, vazados, igualando o potencial deles à referência sem a necessidade de passagem da mesma rota por todo o perímetro.

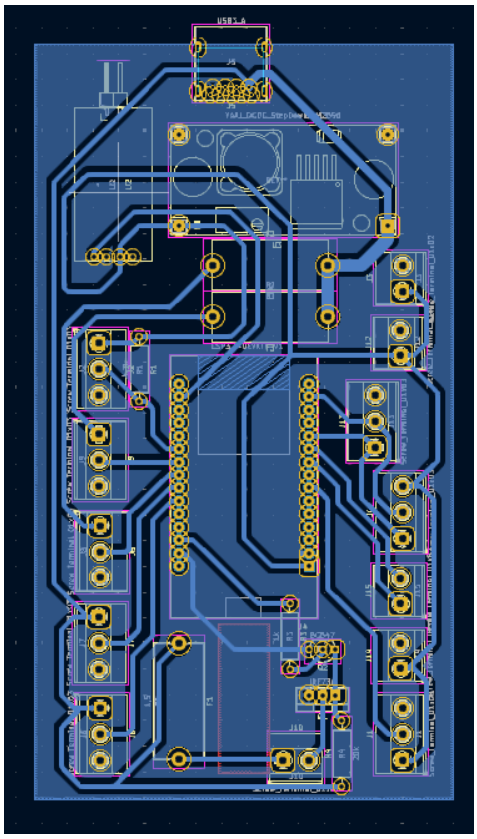

Fig. 34. Projeto da placa de circuito impresso do sistema.

O roteamento foi todo feito na camada "de baixo", isto é, a camada traseira da placa. O aplicativo considera que os componentes estão frente ao desenvolvedor, o que faz necessário, para a manufatura escolhida, que o sistema seja espelhado ao projetista. Mais sobre isso na seção sobre manufatura.

Os sensores foram dispostos, então, de um lado e os atuadores do outro, de maneira que facilitasse o caminho para as rotas de alimentação e fosse reduzido o número de *jumpers*, rotas criadas por cima de outras, ou em outra camada de condução, que encarecem e tornam o sistema mais susceptível a falhas.

VI. Manufatura

Após a conclusão do desenho da placa, é possível utilizar seu projeto para a manufatura da PCI. Dos processos disponíveis durante a prototipação, usinagem com fresadora controlada por computador (CNC), e corrosão por percloreto de ferro, a corrosão foi escolhida por critério de logística.

O processo se trata do uso de um solvente, percloreto de ferro, para corroer o cobre de uma placa [33]. Os materiais mais simples de encontra para a placa base são fibra de vidro e fenolite, sendo a primeira mais resistente a altas temperaturas e possuindo superfície de condução mais uniforme. Ainda assim, por nenhum desses parâmetros serem impeditivos, fenolite foi escolhida, dada sua disponibilidade e custo.

A. Impressão

Tendo o projeto concluído, deve-se considerar sua disposição. Se o projeto foi feito na camada de baixo, a camada de cima terá a configuração da Figura 35 e a de baixo a da Figura 36, o que é precisamente o resultado almejado na PCI final. Caso o projeto tenha sido feito na camada da frente, as rotas vão estar dispostas na camada frente ao desenvolvedor, o que não é ideal, mas, como se trata de um projeto de uma face só, é corrigível.

No editor, há a opção de imprimir o projeto, nenhuma configuração extra é necessária, apenas selecionar a camada em que o projeto foi feito, nesse caso, a camada *B.Cu* de *Bottom copper*, ou *F.Cu* de *Front copper*, caso tenha sido feito na camada da frente. Nesse caso, também, é importante marcar a opção *imprimir espelhado*, para resolver a disposição das rotas.

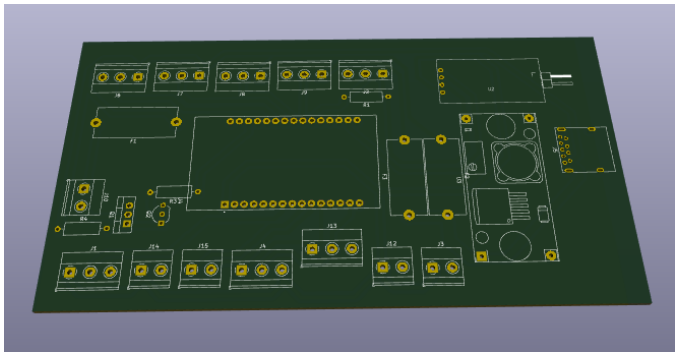

Fig. 35. Simulação da camada de cima.

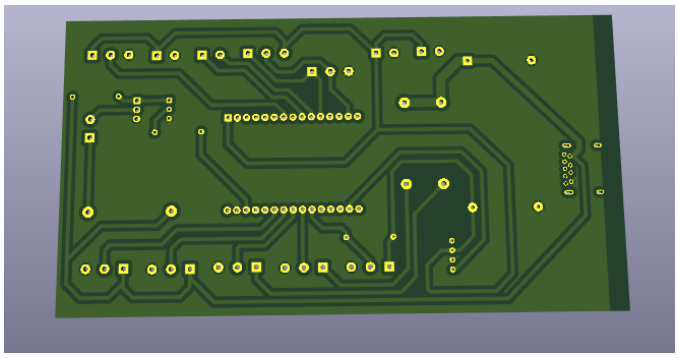

Fig. 36. Simulação da camada de baixo.

Durante o processo, o solvente irá corroer todo o cobre com que entrar em contato. A impressão feita serve como proteção para as zonas de cobre que são necessárias. O percloreto de ferro não corrói cobre coberto por tinta, propriedade necessária para a prototipação.

A impressão do projeto deve ser feita sobre papel reflexivo, papel foto foi utilizado nesse procedimento, com impressora a laser, por permitir a transferência de tinta por termo-indução. Em seguida, essa impressão deve ser selada por fita, foi utilizada fita crepe devido ao calor que seria submetida, frente à superfície condutora da placa limpa. A limpeza da placa é importante. Impressões digitais e manchas podem afetar o resultado final, então é recomendado o uso de lã de aço e álcool isopropílico para o processo de preparação.

B. Indução térmica

Após a preparação é feita a indução térmica. Esse processo pode ser feito com um soprador térmico, um ferro de passar ou, a maneira mais indicada, com uma prensa térmica, método utilizado nesta manufatura. Devido à pressão e à alta temperatura, a tinta da folha é transferida para a superfície condutora, prevenindo a corrosão das zonas desejadas pelo projetista. Durante essa etapa podem haver falhas, mas devem ser corrigidas com um marcador permanente, cuja tinta possui a mesma propriedade da tinta de impressão.

C. Corrosão

Uma vez que o desenho tenha sido posto na placa, a corrosão pode acontecer. O solvente, normalmente comprado em pó, deve ser preparado conforme as instruções da embalagem, e a placa deve ser deixada reagindo por um período que pode variar de 10 minutos a 1 hora, sendo necessário que, periodicamente, a placa seja agitada para verificar o andamento da corrosão. O resultado final pode ser visto na Figura 37. Essa disparidade no período pode acontecer devido ao tamanho da placa corroída ou da vida útil do percloreto. Após diversos usos, o solvente começa a acumular material desprendido de suas reações e perde sua eficiência, aumentando o tempo necessário para a conclusão do processo. Entretanto, o solvente é reutilizável e deve ser preservado conforme as instruções do fabricante.

D. Acabamento

Com a placa final em mãos, resta cortar as bordas, podendo se feito com um arco de serra ou uma retificadora, e fazer os furos. Os furos foram feitos com brocas de 1mm e 0,75mm. Os terminais de placa para fio possuem seção maior, que para os demais é grande o suficiente para interferir em rotas e furos vizinhos. Todo o processo foi feito em uma furadeira de bancada, conforme a Figura 38

Por fim, estando a placa furada, com água e lã de aço, foi removida a tinta e os restos de papel que sobraram da corrosão, o que deixou a placa com o aspecto da Figura 39. Restando apenas o processo de solda, que foi feito seguindo o mapeamento do projeto, já discutido.

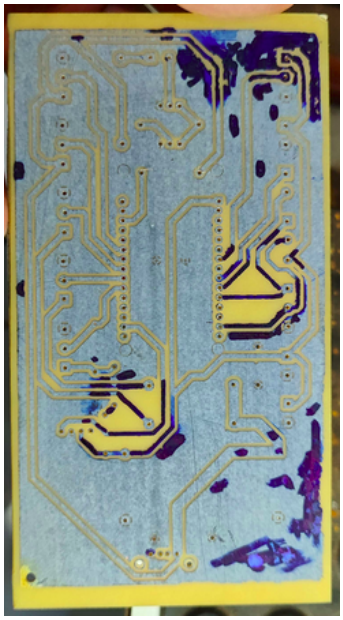

Fig. 37. PCI corroída.

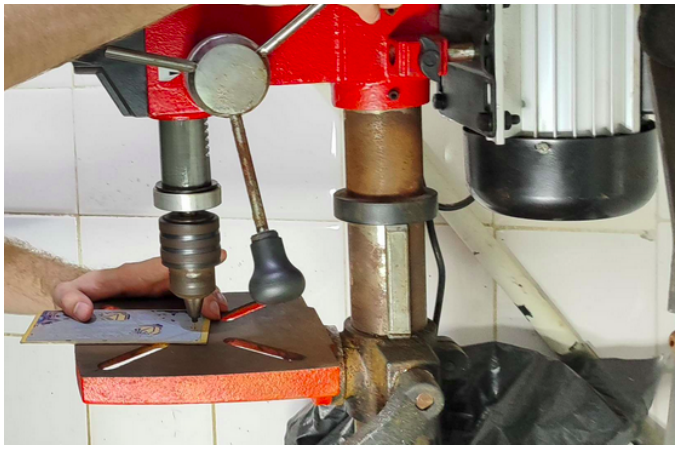

Fig. 38. Furadeira de bancada utilizada.

VII. Resultados

Após algumas semanas com o sistema funcionando de maneira autônoma, enquanto poucas correções e testes eram feitos, a planta cresceu, Figura 40. A estufa se provou um sistema embarcado com baixa manutenção de bom uso.

Quanto à estrutura, possíveis atualizações de código para reduzir o impacto do método que controla a iluminação e um sistema de iluminação mais eficiente resultariam em plantas com maior crescimento.

O sistema de comunicação também se provou eficaz e, sendo posto em par com uma interface própria, Figura 41, forneceu monitoramento de acordo com a necessidade do projeto.

Referências

[1] Felipe L. Mororó, "Desenvolvimento de um protÓtipo de estufa inteligente para a agricultura com tecnologia iot," 2023.

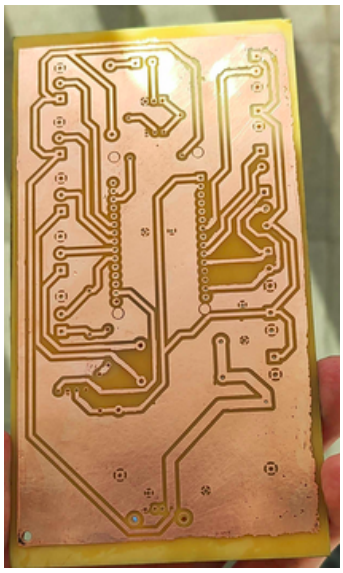

Fig. 39. Placa final.

Fig. 40. Ambiente do sistema

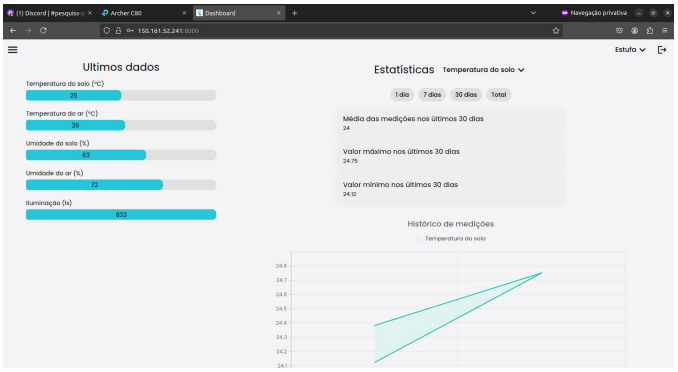

Fig. 41. *Front-end* do sistema

[2] Amazon Web Services, *Mastering the FreeRTOS™ Real Time Kernel*, 1.0 edition, 2024.
[3] Espressif Systems, *Technical reference manual*, 5.2 edition, 2024.
[4] RoboCore, "Lorawan - conceitos básicos," .
[5] Arduino, *Arduino Documentation*.
[6] Mouser Electronics, *DHT11 Humidity Temperature Sensor*.
[7] Analog Devices, *DSB1820 - Programmable Resolution 1-Wire Digital Thermometer*.
[8] Adafruit Industries, *Adafruit BH1750 Ambient Light Sensor*.
[9] Handson Technology, *HC-SR04 Ultrasonic Sensor Module User Guide*, 2.0 edition.
[10] *Soil Moisture Hygrometer Detection Sensor Module W/ Corrosion Resistance Probe DC 3.3-12V for Arduino*.
[11] ON Semiconductor, *Low Offset Voltage Dual Comparators*.
[12] Wolrdsemi, *WS2812B Intelligent control LED integrated light source*.
[13] Imperial college of London, *SERVO MOTOR SG90*.
[14] SparkFun, *Fan-spec*.
[15] Arduino, *Downloading and installing the Arduino IDE 2*.
[16] Espressif Systems, *arduino-esp32: Installing*.
[17] ," .
[18] ," .
[19] Espressif Systems, *Memory Types - ESP32*.
[20] Ms. Ricardo Augusto de Oliveira Rodrigues, ," .
[21] Kicad, *Kicad Documentation*, 2024.
[22] Texas Instruments Incorporated, *LM2596 SIMPLE SWITCHER® Power Converter 150-kHz- 3-A Step-Down Voltage Regulator*.
[23] Fábio Souza, *Comunicação I2C*.
[24] Components101, ," .
[25] FairChild Semiconductor Corporation, *BC546 / BC547 / BC548 / BC549 / BC550 - NPN Epitaxial Silicon Transistor*.
[26] VISHAY, *IRF730*.
[27] Texas Instruments Incorporated, *L293x Quadruple Half-H Drivers*.
[28] DigiKey, "Term blk 2pos side entry 5mm pcb," .
[29] DigiKey, "Term blk 3pos side entry 5mm pcb," .
[30] DigiKey, "Term blk 2p side ent 5.08mm pcb," .
[31] DigiKey, "Term blk 3p side ent 5.08mm pcb," .
[32] WikiPedia, "Pinheader," .
[33] Rosana Guse, *Como fazer uma placa de circuito impresso?*