UNIVERSIDADE ESTADUAL PAULISTA
"JÚLIO DE MESQUITA FILHO"
Instituto de Ciência e Tecnologia
Câmpus de Sorocaba

LUIZ EDUARDO ACCIARI

DESENVOLVIMENTO DE SISTEMA EMBARCADO DE MONITORAMENTO REMOTO DE DISPOSITIVO PARA LIBERAÇÃO DE FLUIDOS AQUECIDOS

Sorocaba
2024

LUIZ EDUARDO ACCIARI

DESENVOLVIMENTO DE SISTEMA EMBARCADO DE MONITORAMENTO REMOTO DE DISPOSITIVO PARA LIBERAÇÃO DE FLUIDOS AQUECIDOS

Projeto Final de Curso apresentado ao Instituto de Ciência e Tecnologia de Sorocaba, Universidade Estadual Paulista (UNESP), como parte dos requisitos para obtenção do grau de Bacharel em Engenharia de Controle e Automação.

Orientador: Prof. Dr. José Roberto Ribeiro Bortoleto.

Coorientador: Prof. Dr. Everson Martins.

Sorocaba

2024

A171d

Acciari, Luiz Eduardo

Desenvolvimento de sistema embarcado de monitoramento remoto de dispositivo para liberação de fluidos aquecidos / Luiz Eduardo Acciari. -- Sorocaba, 2024

71 p. : il., tabs.

Trabalho de conclusão de curso (Bacharelado - Engenharia de Controle e Automação) - Universidade Estadual Paulista (UNESP), Instituto de Ciência e Tecnologia, Sorocaba

Orientador: José Roberto Ribeiro Bortoleto

Coorientador: Everson Martins

1. Automação. 2. Controle eletrônico. 3. Sistemas embarcados (Computadores). 4. Internet das coisas. 5. Controladores programáveis. I. Título.

Sistema de geração automática de fichas catalográficas da Unesp. Biblioteca da Universidade Estadual Paulista (UNESP), Instituto de Ciência e Tecnologia, Sorocaba. Dados fornecidos pelo autor(a).

Essa ficha não pode ser modificada.

LUIZ EDUARDO ACCIARI

DESENVOLVIMENTO DE SISTEMA EMBARCADO DE MONITORAMENTO REMOTO DE DISPOSITIVO PARA LIBERAÇÃO DE FLUIDOS AQUECIDOS

ESTE TRABALHO DE GRADUAÇÃO FOI JULGADO ADEQUADO COMO PARTE DO REQUISITO PARA A OBTENÇÃO DO GRAU DE BACHAREL EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO

Prof. Dr. Everson Martins
Coordenador

BANCA EXAMINADORA:

Prof. Dr. JOSÉ ROBERTO RIBEIRO BORTOLETO
Orientador/UNESP – Campus de Sorocaba

Prof. Dr. EDUARDO VERRI LIBERADO
UNESP – Campus de Sorocaba

Prof. Dr. IVANDO SEVERINO DINIZ
UNESP – Campus de Sorocaba

Agosto de 2024

Dedico este trabalho a Deus, meus pais, avós, tias e amigos, por todo o apoio durante o decorrer da graduação e elaboração do presente trabalho.

AGRADECIMENTOS

Agradeço a Deus por me auxiliar a trilhar os caminhos durante minha graduação.

Agradeço aos meus pais pelo apoio no desenvolvimento do projeto.

Agradeço a minha mãe Márcia Gagliardi, por me auxiliar e me apoiar em minhas escolhas, em minha carreira e no meu desenvolvimento como profissional, estudante e como pessoa.

Agradeço também meus avós por me ajudarem a formar a pessoa que sou hoje e me incentivarem e correr atrás de coisas que gosto.

Agradeço ao fomento do Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) pelas bolsas PIBITI – processos 1812 e 3810. Cujos trabalhos serviram de base para o desenvolvimento do presente trabalho.

Agradeço também ao meu pai, Jorge Luiz, por me incentivar na área da Eletrônica quando eu era criança, me ensinando desde a soldar, analisar continuidade de circuitos e a mexer no multímetro.

Agradeço à minha tia, Joicilene Acciari, por ceder espaço em sua casa para implementar o protótipo.

Agradeço a minha tia, Dra. Heloisa Andréa Acciari, por me auxiliar na revisão do texto deste trabalho, bem como por me ensinar a estudar com seu método “pão na chapa”.

Agradeço especialmente o apoio incondicional de meu amigo Raoni Santos, por me auxiliar na montagem e desenvolvimento do protótipo em sua última versão.

Também agradeço meu amigo, veterano e companheiro de curso, Leonardo Parra Melatti, por seu incondicional apoio na formulação do presente trabalho.

Bem como, também agradeço a todos amigos que me auxiliaram e motivaram a desenvolver o sistema apresentado.

Agradeço também ao CNPq pelas bolsas nas quais desenvolvi projeto de iniciação científica que foi o ponto inicial do desenvolvimento deste trabalho, com suas respectivas melhorias e evoluções.

Agradeço aos professores Dr. José Roberto Ribeiro Bortoleto e Dr. Everson Martins pela orientação no desenvolvimento do presente projeto final de curso, aos quais auxiliaram a traçar melhores estratégias desde o início.

Agradeço também aos meus companheiros de trabalho na OTG pelo apoio em desenvolvimento de projetos Pablo, André, André Luiz, Tiago, Macedo e Cerrado. Bem como,

aos colegas da Going2: Vithor, Alexandro e Leonardo Uemura por auxiliar no meu desenvolvimento para a parte de programação e SCRUM.

Também, agradeço aos professores Marcello Belloti, Eduardo Verri Liberado e Eduardo Paciência Godoy por auxílios de ensinos de tópicos relacionados à pesquisa, bem como aos amigos Felipe Bueno e Lucas Mainente e pelo apoio.

“Na natureza nada se cria, nada se perde, tudo se transforma.”

Antoine Lavoisier.

RESUMO

O presente trabalho descreve o desenvolvimento de projeto final de curso para Engenharia de controle e Automação. Deste modo foram aplicados conceitos de automação, como arquitetura de supervisórios, Redes MODBUS, Protocolo MQTT, Redes TCP/IP, RS-485 e dashboards; bem como também conceitos de controle PID para o desenvolvimento do controlador com Arduino. Integrando conhecimentos a nível interdisciplinar de modo que permitiu a implementação de um protótipo de monitoramento remoto e controle de temperatura para fluidos em baixos fluxos. O protótipo implementado englobou tanto a montagem física de circuitos, gabinetes e soldas eletrônicas, tanto como a programação de placas ESP32 e Arduino de modo a construir um sistema embarcado com conectividade e integração à Internet das Coisas (IoT). Englobando também o desenvolvimento de Dashboard e de conceitos de encapsulamento de dados JSON para a comunicação entre dashboard remoto e o sistema físico. Com isso foram implementados circuitos eletrônicos projetados para maior resistência a EMI e menor latência de comunicação com a internet (WWW), utilizando comunicação TCP/IP cabeada no padrão Ethernet, para testes em redes locais. Portanto, foram obtidos dados de estudo de eficiência do controle de aquecimento, testes de latência e arquitetura para cenários com necessidade de redundância. Com isso, busca-se integrar e desenvolver um sistema robusto e aplicável a partir da síntese dos conhecimentos necessários ao Engenheiro de Controle e Automação.

Palavras-chave: Automação. Controle eletrônico. Sistemas embarcados (Computadores). Internet das coisas. Controladores programáveis.

ABSTRACT

This work describes the development of a final course project for Control and Automation Engineering. In this way, automation concepts were applied, such as supervisory architecture, MODBUS Networks, MQTT Protocol, TCP/IP Networks, RS-485 and dashboards; as well as PID control concepts for controller development with Arduino. Integrating knowledge at an interdisciplinary level in a way that allowed the implementation of a remote monitoring and temperature control prototype for fluids at low flows. The implemented prototype included both the physical assembly of circuits, cabinets and electronic soldering, as well as the programming of ESP32 and Arduino boards to build an embedded system with connectivity and integration to the Internet of Things (IoT). Also encompassing the development of Dashboard and JSON data encapsulation concepts for communication between remote dashboard and the physical system. With this, electronic circuits designed for greater resistance to EMI and lower latency of communication with the internet (WWW) were implemented, using TCP/IP communication wired in the Ethernet standard, for tests on local networks. Therefore, data was obtained from the heating control efficiency study, latency and architecture tests for scenarios requiring redundancy. With this, we seek to integrate and develop a robust and applicable system based on the synthesis of the knowledge necessary for the Control and Automation Engineer.

Keywords: Automation. Electronic control. Embedded systems (Computers). Internet of things. Programmable controllers.

Title in English: Development of embedded system for remote monitoring of a device for release of heated fluids.

LISTA DE FIGURAS

Figura 1 – Diagrama do TRIAC como dois SCRs ligados em antiparalelo 14
Figura 2 – Aplicação do TRIAC em circuito *dimmer* 14
Figura 3 – Módulo ESP-WROOM-32 15
Figura 4 – Placa de desenvolvimento ESP 32 DEVKIT V1 15
Figura 5 – Placa de desenvolvimento Arduino Nano 16
Figura 6 – Tela do Node-RED com fluxos de exemplo 17
Figura 7 – Diagrama de controle PID em malha fechada 18
Figura 8 – Arquitetura geral do sistema de monitoramento e controle 20
Figura 9 – Tela de inicialização após o comando “node-red” 22
Figura 10 – Painel de programação em fluxos do Node-RED 23
Figura 11 – Configurações de cada display de valores. 23
Figura 12 – Layout do dashboard implementado sem recebimento de dados 25
Figura 13 – Esquema elétrico do circuito de monitoramento com ESP32 27
Figura 14 – Placa com circuito de monitoramento baseado em ESP32 28
Figura 15 – Placa de monitoramento montada em caixa com conexões 29
Figura 16 – Configuração do microcontrolador no Arduino IDE 35
Figura 17 – Diagrama de controle PID 35
Figura 18 – Esquema elétrico do circuito 36
Figura 19 – Placa com circuito de controle baseado em Arduino Nano 37
Figura 20 – Placa de controle montada em caixa com conexões 38
Figura 21 – Configuração “Arduino Nano” no Arduino IDE 39
Figura 22 – Configuração de *bootloader* no Arduino IDE 39
Figura 23 – Esquema do circuito de atuação eletrônica 44
Figura 24 – Placa com circuito de atuação eletrônica 45
Figura 25 – Protótipo do sistema geral implementado 50
Figura 26 – Dashboard com valores recebidos do ESP32 50

LISTA DE QUADROS

Quadro 1 – Código de leitura do sensor DHT22 24
Quadro 2 – Código de leitura do sensor DHT22 30
Quadro 3 – Código de leitura e conversão dos pulsos do sensor de fluxo YF-S201 30
Quadro 4 – Código com interruptor contador de pulsos do sensor de fluxo YF-S201 30
Quadro 5 – Código responsável pela leitura dos dados no protocolo MODBUS 31
Quadro 6 – Código de detecção de erros do MODBUS 32
Quadro 7 – Código de encapsulamento JSON e publicação MQTT 32
Quadro 8 – Código de encapsulamento JSON e subscrição MQTT 33
Quadro 9 – Código de exibição dos dados no display LCD 34
Quadro 10 – Código do controle PID aplicado ao aquecimento 40
Quadro 11 – Código do envio de dados MODBUS 41
Quadro 12 – Código do interruptor detector de pontos de zero 41
Quadro 13 – Código dos vetores interruptores de botões do *setpoint* 42

LISTA DE TABELAS

Tabela 1 – Estudo de estabilidade de temperatura 49

SUMÁRIO

1 INTRODUÇÃO 12

2 OBJETIVOS 13

3 FUNDAMENTAÇÃO TEÓRICA 14

3.1. Triodo de corrente alternada (TRIAC) 14

3.2. Microcontrolador ESP32 15

3.3. Microcontrolador Arduino Nano 16

3.4. Node-RED 17

3.5. Controle PID 18

4 DESENVOLVIMENTO 19

4.1. Arquitetura do sistema 20

4.2. Supervisório 21

4.2.1. IMPLEMENTAÇÃO COM O NODE-RED 21

4.3. Monitoramento 25

4.4. Controle 35

4.5. Acionamento elétrico 43

5 RESULTADOS E DISCUSSÕES 46

5.1. Projeto eletrônico dos circuitos envolvidos 46

5.2. Interfaces de IHM 46

5.3. Algoritmos implementados 47

5.4. Implementação da comunicação via MQTT 47

5.5. Estudo de eficiência do controlador PID 48

5.6. Protótipo geral do sistema 49

6 CONCLUSÃO 51

REFERÊNCIAS 52

GLOSSÁRIO 56

ANEXO A – PROGRAMA *DATALOGGER* SUPERVISÓRIO COM ESP32 57

ANEXO B – PROGRAMA CONTROLADOR PID COM ARDUINO NANO 66

1 INTRODUÇÃO

O monitoramento de dados biométricos se apresenta como uma alternativa de ferramentas de auxílio à saúde: tanto em dispositivos *wearables* (vestíveis), quanto em dispositivos hospitalares. De modo com que se obtém o acompanhamento em tempo real utilizando Internet das Coisas para transmissão, recepção e monitoramento de dados com auxílio de sistemas embarcados.

Dentre tal cenário, a liberação controlada de fluidos tem-se mostrado uma abordagem importante no que se tange à engenharia aplicada em diversas áreas, destacando-se, em específico, a aplicação na área da saúde.

Por exemplo, dentro da área médica tem-se a necessidade de manejar fluidos, como soro e sangue, para recuperar pacientes e, em muitos casos, salvar vidas. Consequentemente, tais técnicas são largamente empregadas no tratamento de pacientes; como em transfusão de sangue, terapia ECMO, dentre outros (KHODELI et al., 2016). No contexto do manejo de soro e sangue, a inserção desses fluidos nos pacientes deve ser feita com temperatura controlada para evitar complicações posteriores na recuperação, haja vista que muitos processos bioquímicos são sensíveis a temperaturas baixas (CHEN et al., 2019).

Para isso, sistemas de controle de aquecimento em regime de baixa vazão são instrumentos que auxiliam os procedimentos que envolvem infusão sanguínea ou sorológica de aquecimento, principalmente devido à alta precisão e estabilidade necessárias (NI et al., 2020), que podem ser auxiliadas com o controle PID aplicado. Tal sistema, em conjunto com monitoramento de dados biométricos, pode ser uma alternativa interessante para melhorar os tratamentos aplicados.

A partir desse contexto, o estudo contínuo de sistemas de controle de fluxo e aquecimento de fluidos, bem como o monitoramento biométrico com IoT são ferramentas que podem melhorar a qualidade de tratamentos tanto para a área médica, quanto para aplicações em áreas industriais e agronômicas. Portanto, ao se desenvolver equipamentos com algoritmos de controle e monitoramento IoT, pode-se ter maior taxa de sucesso em diversos procedimentos. Tornando-os também mais viáveis do ponto de vista econômico, mais comuns para o bem-estar da sociedade.

2 OBJETIVOS

O presente projeto final de curso tem em seu objetivo principal implementar avanços no desenvolvimento do sistema de monitoramento e controle de aquecimento. Descrevendo-se os processos inerentes ao desenvolvimento sob a óptica da Engenharia de Controle e Automação, alinhando conhecimentos adquiridos durante a formação de forma multidisciplinar e integradora.

De forma específica, deseja-se obter um sistema embarcado com supervisório em dashboard com as variáveis obtidas, camadas de controle, estrutura SCADA e comunicação cabeada, com MODBUS e Ethernet, para maior robustez. Adicionalmente deseja-se aumentar a confiabilidade do sistema com isolamento entre o envio de dados de dashboard, controle e atuação, separando-se em três circuitos eletrônicos distintos e independentes.

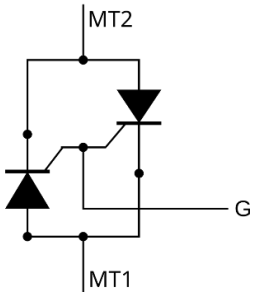
3 FUNDAMENTAÇÃO TEÓRICA

De modo a denotar os principais componentes para execução do projeto, são delimitados seus fundamentos teóricos, permitindo-se o melhor entendimento de suas respectivas funções exercidas no sistema geral implementado.

3.1. Triodo de corrente alternada (TRIAC)

O TRIAC, ou tríodo de corrente alternada, é um componente eletrônico composto por dois retificadores controlados por silício (SCRs) dispostos em antiparalelo, com acionamento realizado pelo terminal de porta, ou gatilho comum aos dois SCRs, que por sua vez, atuam como diodos com condução controlada pela porta. A figura 1 ilustra a combinação esquemática de dois SCRs, formando-se um TRIAC, com porta única (SEGUNDO; RODRIGUES, 2015).

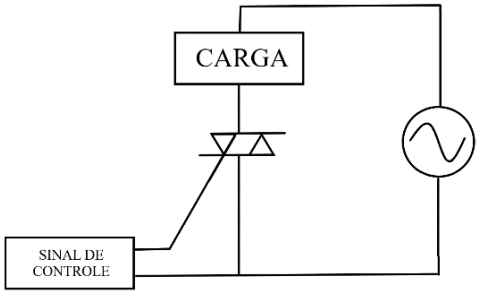
Figura 1 – Diagrama do TRIAC como dois SCRs ligados em antiparalelo.



Fonte: TUNGSTEN; NASRETDINOV, 2022.

Deste modo, pode-se controlar a condução ou não condução de tensão alternada em alimentação AC, permitindo-se o controle de tensão eficaz de acordo com o ângulo de disparo, consequentemente controlando-se a potência sobre a carga do circuito, descrito por *dimmer*.

Figura 2 – Aplicação do TRIAC em circuito *dimmer*.



Fonte: Autoria própria.

3.2. Microcontrolador ESP32

ESP32 é uma série de microcontroladores do tipo sistema em um chip, ou system on a chip (SoC). Foi lançado pela Espressif em 2016, com funcionalidades que incluem Wi-Fi, Bluetooth e pode ser aplicado em grande quantidade de sistemas (ESPRESSIF, 2024).

O ESP32 é apresentado em formato de módulo, como ilustrado na figura 3, com solda em superfície de placa (SMD).

Figura 3 – Módulo ESP-WROOM-32.

Fonte: MOUSER, 2024.

Porém, para auxiliar o desenvolvimento de circuitos, testes e prototipagens, são utilizadas as placas de desenvolvimento baseadas em ESP32, por exemplo a DEVKIT V1, ilustrada na figura 4.

Figura 4 – Placa de desenvolvimento ESP 32 DEVKIT V1.

Fonte: Autoria própria.

Uma característica em que o ES32 se destaca em relação aos outros microcontroladores do mercado é o ponto de sua CPU ser constituída por processador Xtensa com dois núcleos de 32 bits atuando em 240 MHz cada. Ademais, apresenta modo ULP (ultra low power), ou “ultrabaixa potência”, em que ativa um coprocessador de baixo consumo para aplicações que exigem menor consumo de energia, por exemplo, quando necessita-se de alimentação por baterias.

3.3. Microcontrolador Arduino Nano

O microcontrolador Arduino nano, trata-se de uma plataforma de desenvolvimento criada pela Arduino, utilizando-se como base o microcontrolador ATMega328P, fabricado pela ATMEL. Seu primeiro lançamento foi em 2008, junto ao Arduino Uno. A proposta do Arduino Nano é de apresentar as mesmas funcionalidades do Arduino Uno, mas com menores dimensões (ARDUINO, 2024).

De modo a permitir a melhor prototipagem de circuitos, tais placas são implementadas em módulos montados e soldados em placas prontas para uso, com terminais laterais para uso em matriz de contatos e placas perfuradas com furos padrão. A placa de desenvolvimento do Arduino Nano é ilustrada na figura 5.

Figura 5 – Placa de desenvolvimento Arduino Nano.

Fonte: ARDUINO, 2024.

A CPU do microcontrolador ATMega328P é de núcleo único e 8 bits, com capacidade menor em relação ao ESP32. Porém, sua arquitetura AVR torna-o versátil para aplicações de controle, bem como apresenta ADCs (conversores analógico digitais) com comportamento melhor em relação ao ESP32 (LUCIDARME, 2023).

3.4. Node-RED

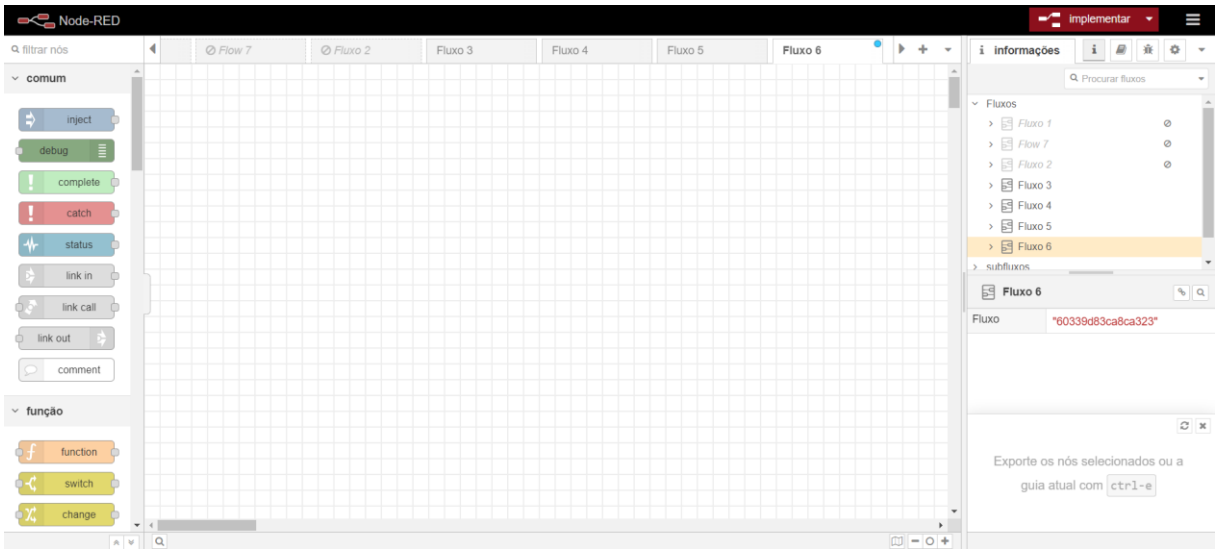
O Node-RED é uma ferramenta de programação para integração com hardwares, utilizando-se APIs para realizar a compilação de programação em blocos para Node.js. Desse modo, permite-se com que se crie páginas da web com layouts personalizáveis, dashboards, estruturas de compilação de dados, comunicação MQTT e integração com topologias de servidores para rodar de forma local ou em nuvem.

Ademais, apresenta um editor baseado em navegador, permitindo-se a implementação de dashboards sem a necessidade de instalação de uma API própria. Bem como, em casos que o node-RED roda em nuvem, não é necessária instalação local de componentes (NODE-RED, 2024).

Com tal plataforma são criados fluxos de programação pelo usuário, sem a necessidade de se atentar a detalhes de programação na linguagem Node.js, permitindo-se o foco no desenvolvimento da aplicação para iniciantes e permitindo maior acessibilidade a entusiastas para elaborarem seus próprios dashboards com informações oriundas de hardwares, por exemplo de *dataloggers* que enviam dados via MQTT para monitoramento.

A figura 6 ilustra a tela do ambiente de programação de fluxos via Node-RED.

Figura 6 – Tela do Node-RED com fluxos de exemplo.



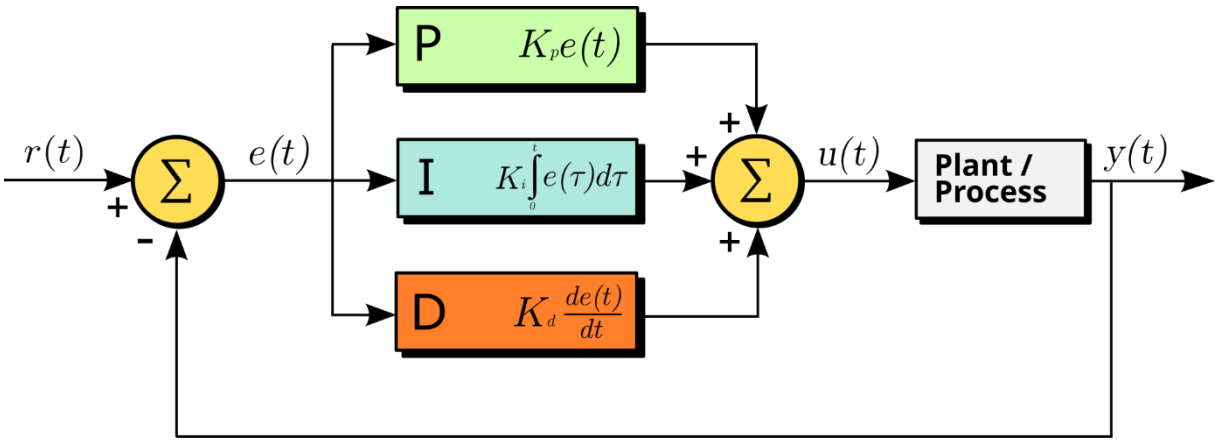
Fonte: Autoria própria.

3.5. Controle PID

Controle proporcional integral derivativo trata-se de uma técnica de controle de processos que utiliza um sistema em malha fechada com realimentação para determinar o erro entre o valor desejado e o valor obtido (OGATA, 2014)

A partir do erro obtido, são adicionados processamento sobre os valores usando constantes proporcional (Kp) multiplicada pelo erro, integral (Ki) com a integral do erro e derivativa (Kd) com a derivada do erro, como ilustra a figura 7.

Figura 7 – Diagrama de controle PID em malha fechada.



Fonte: URQUIZO, 2016.

Com a somatória dos resultados, obtém-se um sinal controlado pela técnica PID ao qual a variável final pode ser alcançada em função de um determinado intervalo de tempo, de modo com que o controle pode ser abstraído por uma função transferência, denominada u(t), como ilustra a figura 7.

Deste, modo, pode-se utilizar o controle PID para realizar uma metodologia de controle de aquecimento, de modo com que a malha de realimentação seja obtida por meio de um medidor de temperatura, bem como o valor desejado pode ser dado pelo *setpoint* de temperatura desejada.

Com isso o controle PID implementado de forma analógica ou digital em um circuito controlador, permite o controle do aquecimento conforme o sistema é realimentado com sua temperatura obtida.

4 DESENVOLVIMENTO

Foram utilizados microcontroladores ESP32 e Arduino Nano para o desenvolvimento do protótipo com sistema de monitoramento. De modo com que a seção de comunicação MQTT e monitoramento dos dados é realizada pelo ESP32. Enquanto o algoritmo de controle de aquecimento é executado no Arduino Nano de forma independente de conexão com a internet.

O desenvolvimento do projeto foi sintetizado em dez passos:

a) Definição da arquitetura do sistema, com camada de controle usando Arduino Nano e camada de monitoramento usando ESP32;
b) Calibração do código para aplicação do controle PID com circuito TRIAC de modo mais preciso;
c) Confecção do circuito em placas de circuito perfuradas, design de interface homem máquina utilizando-se display LCD I2C;
d) Implementação da conectividade MQTT por meio de Ethernet;
e) Programação de código para envio em padrão de um encapsulamento JSON com os dados dos sensores;
f) Implementação de circuito e comunicação MODBUS entre ESP32 e Arduino Nano;
g) Implementação da comunicação para acesso remoto de dados via MQTT;
h) Implementação e testes de sensoriamento e comunicação com dashboard na plataforma Node-RED;
i) Ajustes de código para escalabilidade com mais controladores PID baseados em Arduino Nano no mesmo barramento MODBUS de monitoramento
j) Montagem em caixas plásticas, ajustes de gabinete e de protótipo.

Tais passos foram desenvolvidos e implementados ao longo da vigência de 6 meses do desenvolvimento do projeto, utilizando-se de sistemas e equipamentos disponíveis para a sua elaboração, bem como a escolha de materiais, montagem de circuitos, e solda eletrônica realizadas para o correto funcionamento do protótipo.

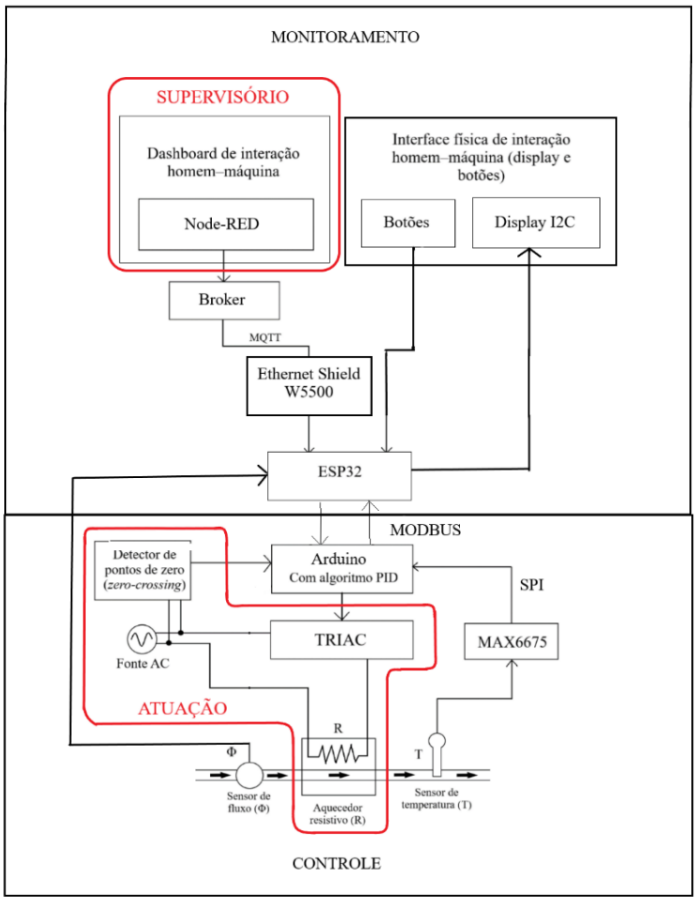
4.1. Arquitetura do sistema

O sistema desenvolvido pode ser classificado em quatro camadas: supervisório, monitoramento, controle e atuação. Com isso, se abstrai o caminho de dados em hardware e software de forma semelhante à utilizada em sistemas de arquitetura SCADA, com terminal mestre e remoto.

Tal topologia é utilizada para o monitoramento e replicação de informações da camada de controle para a camada de monitoramento, de modo a isolar a dependência de comunicação com a rede mundial de computadores (WWW) para as ações críticas de controle de temperatura. Bem como também permite a escalabilidade de mais controladores ao sistema de monitoramento.

A arquitetura do sistema implementado em protótipo é ilustrada na figura 8. Englobando desde a eletrônica atuadora sobre o aquecimento até o dashboard de supervisão de dados.

Figura 8 – Arquitetura geral do sistema de monitoramento e controle.



Fonte: Autoria própria.

4.2. Supervisório

Conforme a arquitetura delimitada na figura 1. A apresentação de dados em software é feita utilizando-se a ferramenta de Dashboard do Node-RED, que obtém os dados por meio do protocolo MQTT. Atuando como camada de supervisório do sistema.

A comunicação MQTT é feita com um broker gratuito de acesso público da HiveMQ, tal broker tem a função de orquestrar os dados vindos de clientes, que são os dispositivos que publicam e recebem os dados por meio de tal protocolo, usando a rede mundial de computadores (WWW) para a comunicação entre os clientes e o broker MQTT.

Neste caso, um cliente é o microcontrolador ESP32 atuando como transmissor de dados, enquanto o Node-RED é um cliente que atua como receptor e tratador de tais dados.

No dashboard do Node-RED, bem como no broker, pode-se utilizar o sistema com vários clientes, permitindo-se a expansão do sistema com vários ESP32 de monitoramento.

Para envio consistente das mensagens, é utilizado um encapsulamento JSON com os dados dos sensores de modo a ser legível para humanos e para máquinas, bem como interpretado pelo Node-RED.

4.2.1. IMPLEMENTAÇÃO COM O NODE-RED

De modo a ter um sistema mais robusto e independente de aplicações externas, o Node-RED foi instalado no computador com sistema operacional Windows utilizando-se sua documentação para instalação do programa (NODE-RED, 2024).

Para o Node-RED executar normalmente no ambiente local, deve ser instalado na máquina o Node.js, que executa códigos em Javascript sem precisar de um navegador. Tal configuração permite o Node-RED rodar em Javascript no computador local.

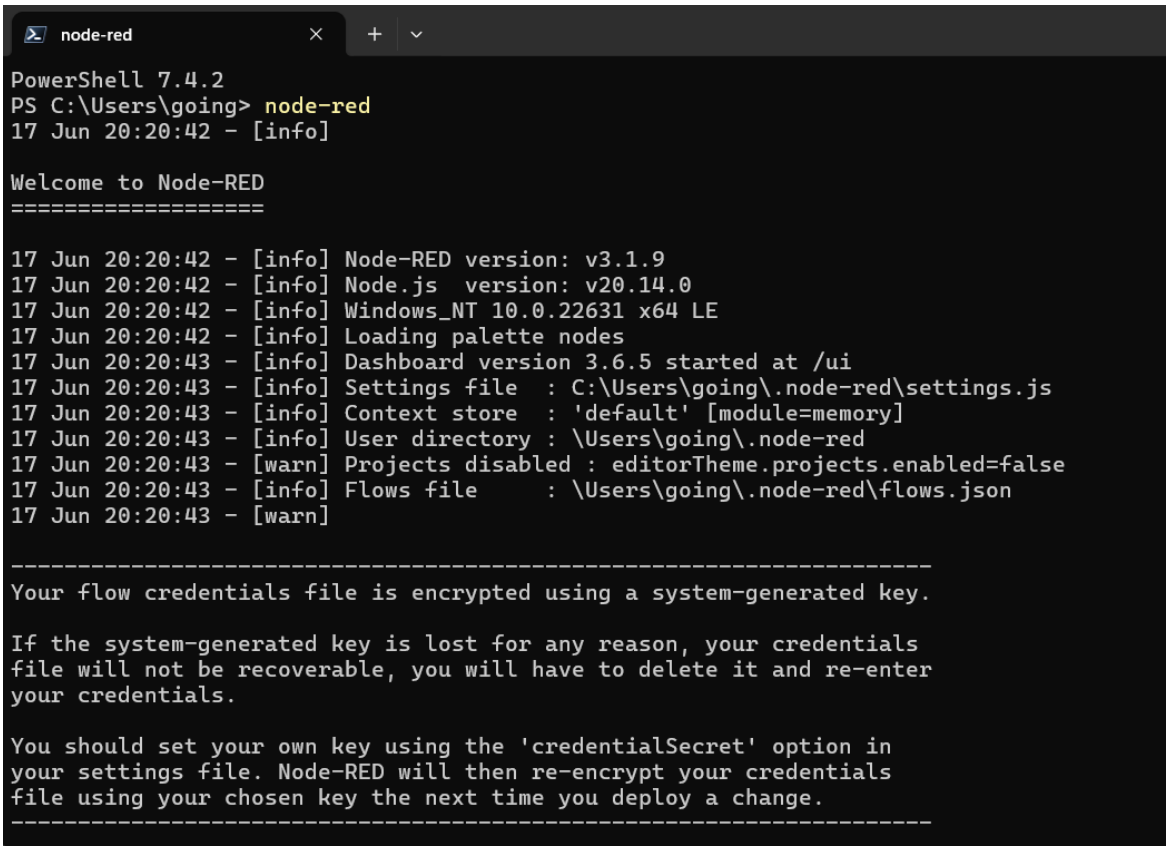
Abstraindo-se os conceitos de desenvolvimento, pode-se ter que o Node-RED atua como uma ferramenta de desenvolvimento que atua tanto para gerar dashboards de *frontend* (programação de interface visual), bem como para construir toda infraestrutura de comunicação e conversões necessárias em *backend* (programação de funcionalidades) para o correto funcionamento do sistema.

Após a instalação de Node.js, Node-RED e seus complementos, pode-se executar o programa. Para isso, foi necessário direcionar a linha de comando para o diretório em que o

programa está instalado e, posteriormente, escrever na linha de comando do Windows Powershell 7 o comando “node-red”. Com isso o computador roda o programa Node-RED atuando como um servidor local conectado à WWW.

A figura 9 ilustra a tela de inicialização após o comando “node-red”. A partir desta tela, tem-se um endereço IP local gerado para acesso à tela de programação gráfica do Node-RED.

Figura 9 – Tela de inicialização após o comando “node-red”.

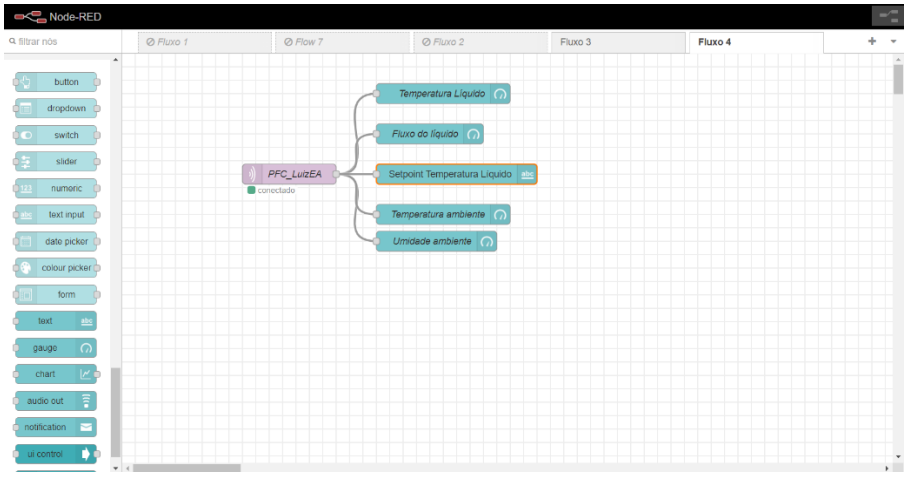


Fonte: Autoria própria.

Ao acessar-se a página do servidor local do Node-RED, disponível no endereço IP mostrado no prompt de comando, temos acesso à tela de programação visual. Nesta tela, foi implementado o programa para transporte de dados do JSON, com os blocos de dashboard e de MQTT. Onde os blocos de dados de JSON são recebidos via MQTT no tópico que o dashboard assina.

Após a programação na linguagem gráfica de programação do Node-RED, obteve-se o painel de programação em fluxos ilustrado na figura 10. Onde pode-se observar o bloco de subscrição MQTT em rosa e os blocos de dashboard em azul.

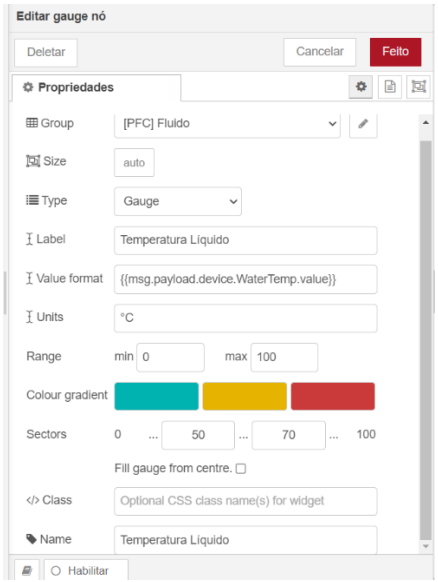
Figura 10 – Painel de programação em fluxos do Node-RED.



Fonte: Autoria própria.

Com os dados desse tópico recebidos, são utilizados blocos do dashboard para interpretar cada dado do JSON utilizando-se a configuração ilustrada na figura 11.

Figura 11 – Configurações de cada display de valores.



Fonte: Autoria própria.

Ao observar-se o campo *value format* (formato do valor) na figura 11, tem se o endereço em JSON ao qual o bloco segue para obter o valor da variável definida em sua configuração. No caso do bloco observado na figura 11, trata-se da temperatura do líquido, que é delimitada pelo endereço "{{msg.payload.device.WaterTemp.value}}", que é definido no programa de monitoramento do ESP32.

Com o endereçamento correto na configuração, os dados são separados a partir da estrutura JSON denotada no quadro 1.

Quadro 1 – Código de leitura do sensor DHT22.

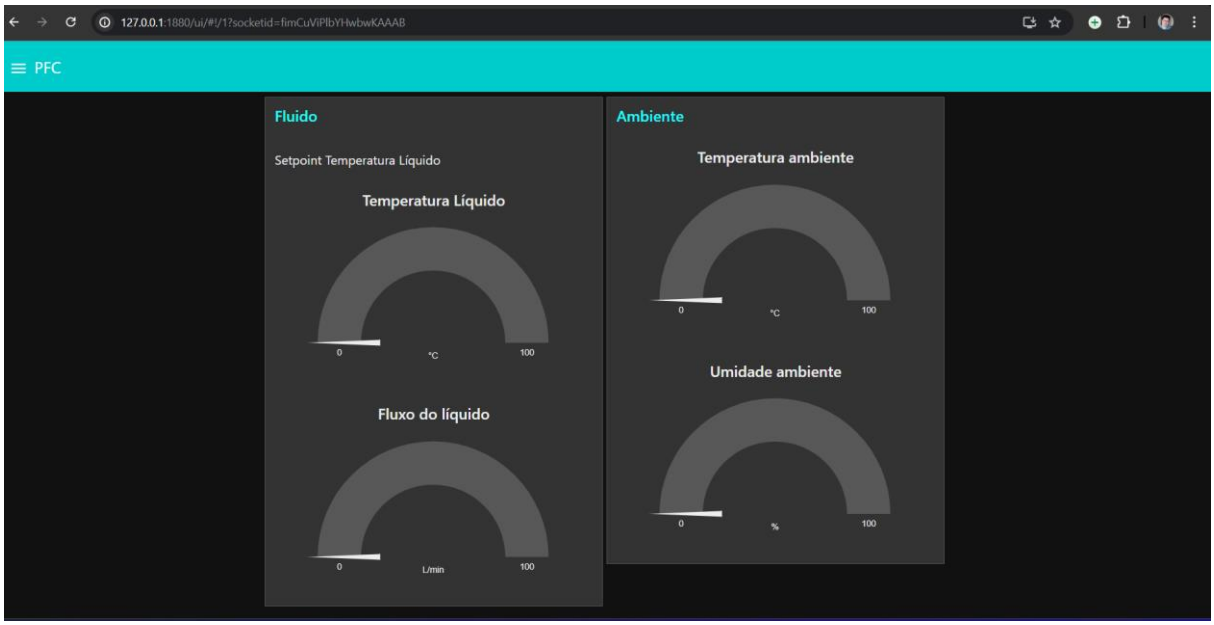
```
{
  "device":{
    "ID":"DEV01",
    "WaterTemp":{
      "type":"Fluid_temp",
      "value":35.1,
      "unit":"°C"
    },
    "WaterFlow":{
      "type":"Fluid_flow",
      "value":5.6,
      "unit":"L/min"
    },
    "WaterSetpoint":{
      "type":"Fluid_setpoint",
      "value":40.0,
      "unit":"°C"
    },
    "AmbientTemp":{
      "type":"Ambient_temp",
      "value":23.1,
      "unit":"°C"
    },
    "AmbientHum":{
      "type":"Ambient_hum",
      "value":51.2,
      "unit":"%"
    }
  }
}
```

Fonte: Autoria própria.

Tal estrutura JSON é implementada pelo trecho de publicação inserido no programa do microcontrolador ESP32, indicado no quadro. Deste modo, atuando como saída de publicação do ESP32 via MQTT e interpretado pelo Node-RED.

A figura 12 ilustra o dashboard implementado pelo Node-RED de acordo com o programa do fluxo da figura 10, para o caso de o ESP32 desconectado, mostrando os dados em vazio.

Figura 12 – Layout do dashboard implementado sem recebimento de dados.



Fonte: Autoria própria.

4.3. Monitoramento

Já para a camada de monitoramento utiliza o ESP32 conectado a um módulo Shield Ethernet W5500 para comunicação com a rede mundial de computadores (WWW) e, consequentemente, com o broker, que publica e subscreve as mensagens segundo a estrutura JSON definida no dispositivo.

Foi escolhida a conexão por rede cabeada por tal metodologia garantir maior estabilidade de conexão e menor interferência no espectro de radiofrequência (RF). Garantindo-se maior confiabilidade no envio de dados essenciais para a aplicação.

O protótipo implementado utilizou o microcontrolador ESP32 para a camada de monitoramento, devido à sua maior capacidade de processamento, memória RAM e memória flash. Bem como melhor interfaceamento com redes de internet e uma grande variedade de bibliotecas disponíveis para uso, auxiliando-se na implementação do programa em seu respectivo firmware.

No hardware de monitoramento implementado com o ESP32, também foi adicionado um sensor de temperatura e umidade DHT22 para leitura de temperatura e umidade ambientes a fim de se monitorar a diferença entre a temperatura ambiente e a do fluido a ser aquecido.

Para comunicação com a interface de controle é utilizado o protocolo MODBUS RTU, pois permite a comunicação do monitor com vários dispositivos clientes em um mesmo barramento (MODICON, 1996).

O protocolo MODBUS RTU, que, por sua vez é aplicado sobre os sinais seriais recebidos pelo barramento RS-485 implementado no circuito com auxílio do CI MAX485, da Maxim Integrated (MAXIM INTEGRATED, 2014).

Pode-se traçar um paralelo entre a arquitetura de redes industriais de comunicação e os dispositivos do sistema.

As camadas *FieldBus* e *DeviceBus*, do modelo OSI de redes aplicado ao MODBUS (GODOY, 2021) podem ser correlacionadas de modo que, o ESP32 atua na camada *FieldBus* do modelo OSI, enquanto o Arduino Nano se situa na camada *DeviceBus*.

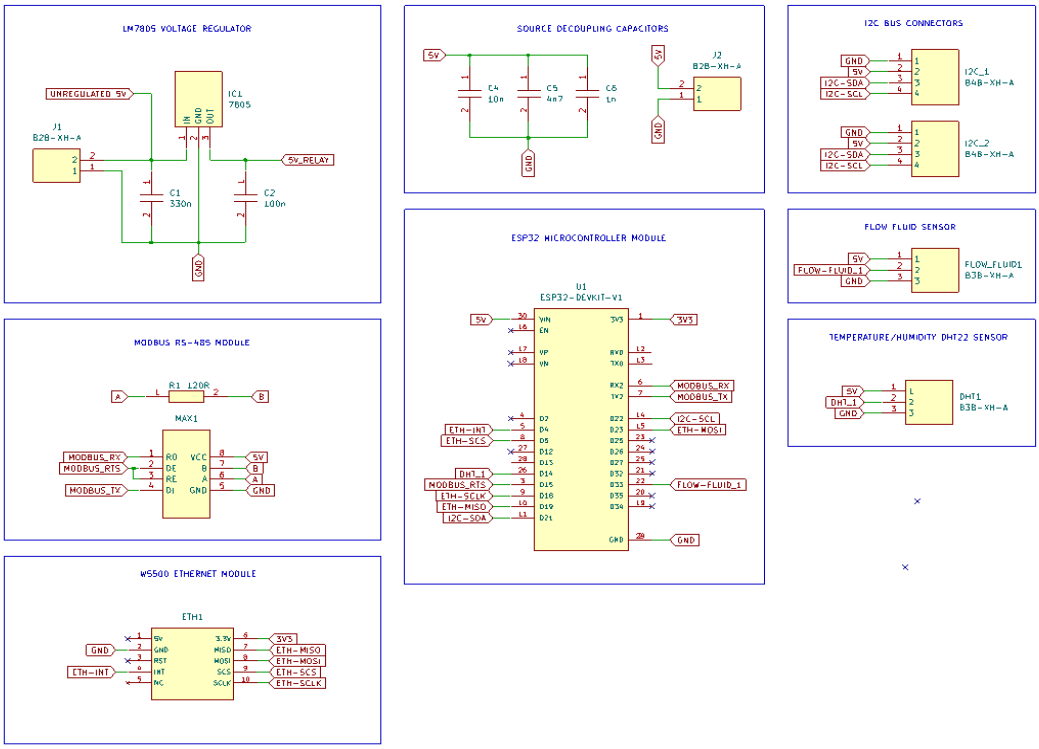
Ademais, com o protocolo MODBUS RTU, pode-se escalar a comunicação até 32 dispositivos simultaneamente, por meio da interface padrão EIA/TIA-485 (WEG, 2013, p. 7).

Com isso, o ESP32 (ESPRESSIF, 2023) atua como líder e o Arduino Nano (ARDUINO, 2024) como agente da comunicação. O Arduino Nano utilizado tem sua plataforma baseada no microcontrolador Atmel ATmega328P com aplicações versáteis para automação (ATMEL, 2015).

Ademais, tem-se independência entre o sistema de controle e o de monitoramento, de modo que, se o ESP32 apresentar falhas, tais falhas não afetam a rotina de controle do Arduino Nano, e vice-versa. Logo, a confiabilidade do sistema é maior em relação às topologias baseadas em microcontrolador único.

A figura 13 ilustra o esquema do circuito de monitoramento, implementado no software KiCad e usando como base os módulos prontos de ESP32.

Figura 13 – Esquema elétrico do circuito de monitoramento com ESP32.



Fonte: Autoria própria.

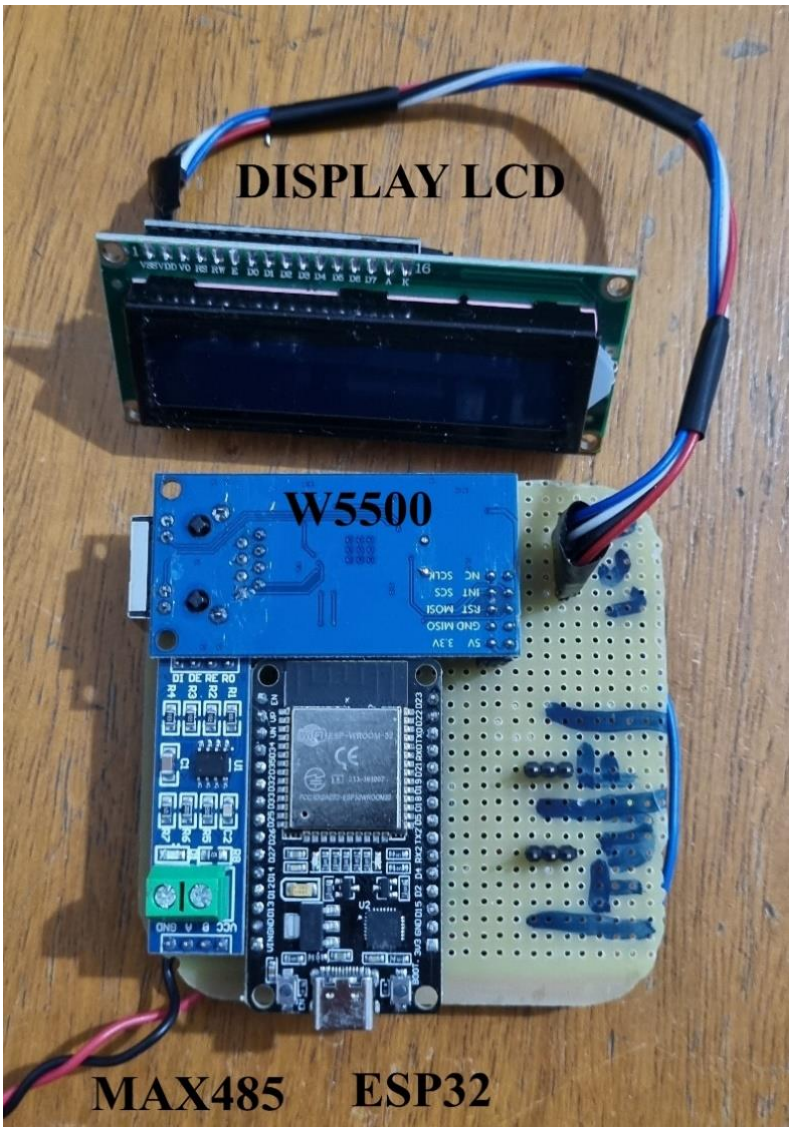
No esquema elétrico da figura 6, pode-se destacar o ESP32 com rótulos de conexão para seus respectivos periféricos, permitindo-se a comunicação de dados entre si com protocolos I2C, SPI e *Onewire*.

O módulo W5500 de Ethernet utiliza comunicação serial por meio do protocolo Serial Peripheral Interface (SPI), ou interface serial para periféricos. Permitindo-se a comunicação com vários dispositivos no barramento SPI do ESP32 por meio do acionamento do terminal Chip Select (CS), ou seletor de chip.

Já o módulo display LCD utiliza um expansor de portas I2C para economizar portas GPIO do ESP32. Tal protocolo permite-se com que se tenha vários periféricos conectados ao mesmo barramento, senso que cada periférico atua como um *slave* com endereço de registrador próprio em uma estrutura *master–slave* com o ESP32.

A figura 14 ilustra o circuito de monitoramento montado na placa com ESP32. Com os componentes na ordem de cima para baixo, esquerda para direita: display LCD I2C, módulo Shield Ethernet W5500, módulo MAX485 para RS-485 e módulo ESP32.

Figura 14 – Placa com circuito de monitoramento baseado em ESP32.

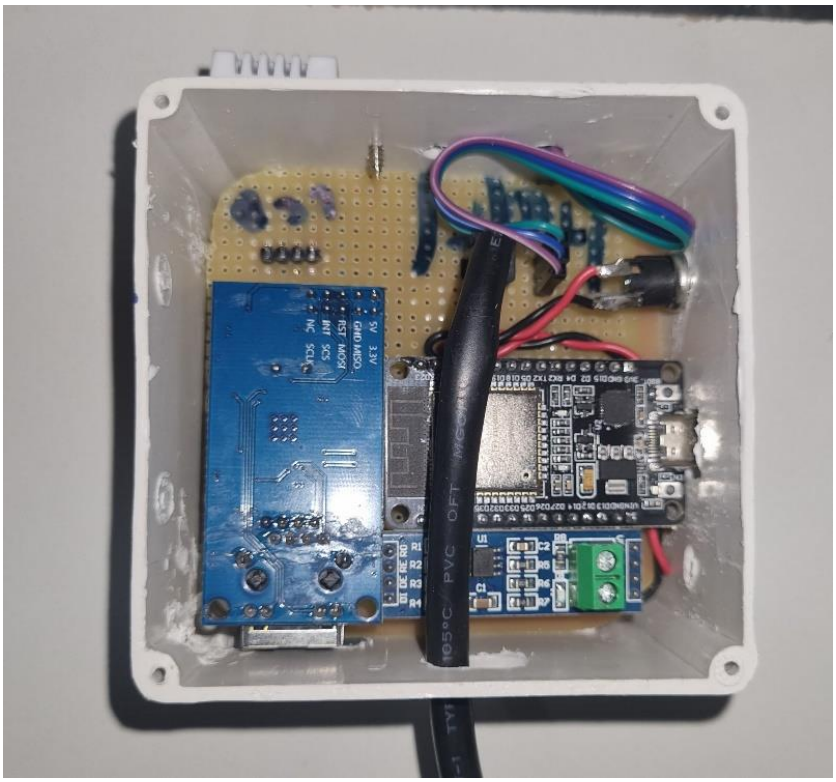


Fonte: Autoria própria.

A presente placa ilustrada na figura 14 foi implementada com a técnica de solda *Point Throught Hole* (PTH), ou terminal através do buraco. Permitindo-se a fixação correta dos componentes eletrônicos nas ilhas de solda da placa padrão de prototipagem, bem como permite a elaboração de circuitos com cabos na parte inferior, de modo com que a solda permita uma maior resistência mecânica para o conjunto.

Já a figura 15 ilustra o circuito de monitoramento montado com gabinete, montado na placa com ESP32, sensor de temperatura e ambiente DHT22, conexão para o sensor YF-S201, cabeamento de MODBUS e cabeamento de alimentação.

Figura 15 – Placa de monitoramento montada em caixa com conexões.



Fonte: Autoria própria.

Na figura 15, nota-se o sensor DHT22 fixado externamente para leitura de temperatura e umidade externas, a fim de se ter parâmetros de diferença de temperatura entre fluido e ambiente, bem como monitorar as condições ambientais.

Ademais, o cabo de revestimento preto presente na figura 15 trata-se da conexão com o sensor de fluxo YF-S201, ao qual é conectado por conector JST em barra pinada de 2,54 mm soldada na placa, para melhor confiabilidade do sinal.

Junto ao circuito, foi desenvolvido também o programa do anexo A, em linguagem de programação C++, no framework Arduino, para o ESP32 atuar no monitoramento de sensores, leitura MODBUS e envio de estruturas JSON com os dados encapsulados via MQTT para o Broker público HiveMQ na WWW.

O quadro 2 denota o trecho de código responsável pelo monitoramento do sensor de temperatura e umidade ambientais por meio do sensor conjugado de temperatura e umidade DHT22.

Quadro 2 – Código de leitura do sensor DHT22.

```
external_hum = dht1.readHumidity();
external_temp = dht1.readTemperature();
```

Fonte: Autoria própria.

O quadro 3 delimita o trecho de código responsável pelo monitoramento do sensor de fluxo por efeito Hall YF-S201.

Quadro 3 – Código de leitura e conversão dos pulsos do sensor de fluxo YF-S201.

```
pulse1Sec = pulseCount;
pulseCount = 0;

water_flux = ((1000.0 / (millis() - previousMillis)) * pulse1Sec) / calibrationFactor;
previousMillis = millis();

flowMilliLitres = (flowRate / 60) * 1000;
totalMilliLitres += flowMilliLitres;
```

Fonte: Autoria própria.

Como trata-se de um sinal cujo valor de fluxo influencia analogamente na frequência do pulso gerado, o ESP32 realiza a contagem de tais pulsos usando um interruptor de microcontrolador, mostrado na função de código do quadro 4. Com isso, a cada vez que é detectado um pulso, o programa para, lê o pulso, registra de retorna ao programa.

Quadro 4 – Código com interruptor contador de pulsos do sensor de fluxo YF-S201.

```
void IRAM_ATTR pulseCounter()
{
  pulseCount++;
}
```

Fonte: Autoria própria.

O quadro 5 mostra o trecho do programa ao qual é responsável pela leitura de dados MODBUS, atuando como cliente (ou mestre) sobre os dispositivos servidores no barramento. No caso, os dados são coletados do Arduino que atua como dispositivo servidor de dados de *setpoint* e temperatura do fluido. Tal leitura é realizada com auxílio da biblioteca *modbus-8266* (SARMENTO, A.; EMELIANOV, A, 2024).

Quadro 5 – Código responsável pela leitura dos dados no protocolo MODBUS.

```
//Leitura MODBUS master para CDU (SLAVE 2):
uint16_t res[REG_COUNT]; // Vetor dos registradores lidos, cujo número é determinado pelo contador de variáveis.
if (!mb.slave()) {    // Checa se não há conflito de linha.
  mb.readIreg(SLAVE_ID, FIRST_REG, res, REG_COUNT, cb); // Especifica requisição de leitura do registrador de entrada (Input Register) para o slave de ID determinado no primeiro campo da função.;
  while(mb.slave()) { // Checa se linha está livre.
    mb.task(); //Executa comandos de requisição do modbus master para o modbus slave.
    delay(10); //Atraso para melhor integridade do sinal.
  }
  water_temp = (res[0])/100; // Guarda temperatura água (saída), recebido no valor do registrador de entrada 0x001.
  setpoint_temp = (res[1])/100; // Guarda setpoint de temperatura, recebido no valor do registrador de entrada 0x002.

  Serial.print("\n");
  Serial.print("\nTemperatura da água (saída): ");
  Serial.print(water_temp);
  Serial.print(" °C");
  Serial.print("\nSetpoint de temperatura: ");
  Serial.print(setpoint_temp);
  Serial.print(" °C");
  Serial.print("\n");
  Serial.print("\n");
}
```

Fonte: Autoria própria.

Com o código do quadro 5, o ESP32 solicita os dados de temperatura do fluido e temperatura desejada (*setpoint*) programada no microcontrolador por meio de registradores de valores inteiros.

Para contornar os valores inteiros, são adicionados divisores por cem nos valores obtidos das variáveis, de modo a permitir a comunicação de valores fracionados com decimais por meio de valores inteiros. Obtendo-se um condicionamento para os valores recebidos por meio do protocolo MODBUS RTU. Com isso, tem se valores com 4 algarismos significativos em suas respectivas mantissas.

O quadro 6 delimita a função de detecção de erros para a leitura de MODBUS, permitindo-se com que os dados sejam lidos usando o protocolo MODBUS RTU, que, por sua vez é aplicado sobre os sinais seriais recebidos pelo barramento RS-485.

Quadro 6 – Código de detecção de erros do MODBUS.

```
bool cb(Modbus::ResultCode event, uint16_t transactionId, void* data) { //Função de detecção de erros.
  if (event != Modbus::EX_SUCCESS) {
    Serial.print("Request result: 0x");
    Serial.print(event, HEX);
  }
  return true;
}
```

Fonte: Autoria própria.

O quadro 7 delimita o trecho de código que realiza o encapsulamento e publicação de dados nas estruturas JSON, com auxílio da biblioteca *ArduinoJSON* (BLANCHON, 2024). Tais estruturas JSON com os dados são publicadas via MQTT para o Broker público HiveMQ na WWW, com auxílio da biblioteca AsyncMQTT_ESP32 (ROGER; HOANG, 2022).

Quadro 7 – Código de encapsulamento JSON e publicação MQTT.

```
// JSON for Publish:
JsonDocument docPublisher;
docPublisher["device"]["ID"] = "DEV01";
docPublisher["device"]["WaterTemp"]["type"] = "Water_temp";
docPublisher["device"]["WaterTemp"]["value"] = w_temp;
docPublisher["device"]["WaterTemp"]["unit"] = "°C";
docPublisher["device"]["WaterFlow"]["type"] = "Water_flow";
docPublisher["device"]["WaterFlow"]["value"] = w_flux;
docPublisher["device"]["WaterFlow"]["unit"] = "L/min";
docPublisher["device"]["WaterSetpoint"]["type"] = "Water_setpoint";
docPublisher["device"]["WaterSetpoint"]["value"] = setpoint_temp;
docPublisher["device"]["WaterSetpoint"]["unit"] = "°C";
docPublisher["device"]["AmbientTemp"]["type"] = "Ambient_temp";
docPublisher["device"]["AmbientTemp"]["value"] = ext_temp;
docPublisher["device"]["AmbientTemp"]["unit"] = "°C";
docPublisher["device"]["AmbientHum"]["type"] = "Ambient_hum";
docPublisher["device"]["AmbientHum"]["value"] = ext_hum;
docPublisher["device"]["AmbientHum"]["unit"] = "%";

String payloadPublisher;
serializeJson(docPublisher, payloadPublisher);
uint16_t packetIdPublisher = mqttClient.publish(MQTT_PUB_TOPIC, 0, true, payloadPublisher.c_str());
Serial.printf("Published Temperature to %s with packetId: %i\n", MQTT_PUB_TOPIC, packetIdPublisher);
```

Fonte: Autoria própria.

O quadro 8 mostra o trecho de código que realiza a subscrição (leitura) de dados recebidos via MQTT nas estruturas JSON, utilizando-se as bibliotecas *ArduinoJSON* e *AsyncMQTT_ESP32*.

Quadro 8 – Código de encapsulamento JSON e subscrição MQTT.

```
// JSON Setpoint Subscriber
JsonDocument docSubscriber;
docSubscriber["config"]["ID"] = "DEV01";
docSubscriber["config"]["sensor_3"]["type"] = "Fluid_setpoint";
setpoint_temp = docSubscriber["config"]["sensor_3"]["value"];
docSubscriber["config"]["sensor_3"]["unit"] = "°C";

String payloadSubscriber;
serializeJson(docSubscriber, payloadSubscriber);
uint16_t packetIdSubscriber = mqttClient.publish(MQTT_SUB_TOPIC, 0, true, payloadSubscriber.c_str());
Serial.printf("Published Temperature to %s with packetId: %i\n", MQTT_SUB_TOPIC, packetIdSubscriber);
```

Fonte: Autoria própria.

Deste modo, com as funcionalidades de publicação e subscrição do protocolo MQTT, permite-se com que os dados sejam lidos e escritos em qualquer localização com acesso aberto à internet. Podendo-se conectar a outros clientes de MQTT para fazer a leitura dos dados. Como é o caso do supervisório do dashboard, que atua como cliente do broker e assina o tópico onde são enviados os dados encapsulados por JSON.

Tal arquitetura permite, com um broker dimensionado, a escalabilidade do sistema, com coleta de dados de mais dispositivos IoT, permitindo-se uma leitura múltipla de dados, com seus respectivos dashboards maiores para tais casos.

Com objetivo de permitir uma Interface Homem–Máquina (IHM) para monitorar alguns dados importantes e demonstrar a transmissão de dados via Modbus RTU, foi utilizado um Display LCD com 16X2 caracteres. Para evitar uso de muitos pinos do ESP32, foi utilizado o conjunto do display LCD com o Expansor de Entradas e Saídas PCF8574, de 8 bits, comunicando com o ESP32 pelo barramento I2C.

O protocolo I2C permite vários dispositivos no mesmo barramento, economizando portas do microcontrolador, somente necessitando de que seja direcionado cada um em seu respectivo endereço hexadecimal.

Com isso, para o correto funcionamento do visor, foi utilizada uma biblioteca própria implementada para uso do conjunto formado pelo display LCD com o expansor PCF8574, denominada *LiquidCrystal_I2C.h* (BRABANDER, F.; SCHWARTZ, M., 2024).

O quadro 9 mostra o trecho de código que realiza a exibição dos dados no display LCD, utilizando-se a biblioteca *LiquidCrystal_I2C.h*.

Quadro 9 – Código de exibição dos dados no display LCD.

```
// Set cursor to first column, first row
lcd.setCursor(0, 0);
lcd.print("Temp: ");
lcd.print(water_temp,3);
lcd.print(" C");

lcd.setCursor(0, 1);
lcd.print("Fluxo: ");
lcd.print(water_flux,3);
lcd.print(" L/min");
```

Fonte: Autoria própria.

Deste modo, com o código do quadro 9, são mostrados no display LCD valores de temperatura, em [°C], obtidos pelo MODBUS; bem como valores de taxa de fluxo, em [L/min], permitindo-se a exposição de informações relevantes ao processo de monitoramento no display de IHM (interação homem–máquina) do circuito de monitoramento.

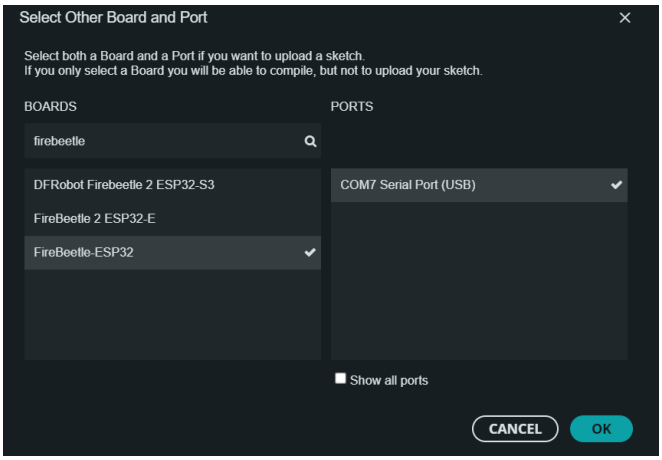
Em suma, tem-se o sistema fechado de monitoramento e comunicação com os dispositivos e envio de dados para o supervisório, usando protocolo MQTT e JSON para leitura simples dos dados.

Com isso, foi realizada a gravação do código fonte do anexo A no ESP32 presente na placa do sistema de monitoramento.

Tal gravação foi realizada no ambiente de programação do Arduino, o Arduino IDE. EM que foi configurado para compilar o código original em C++ do Arduino para as instruções do ESP32. Para isso é necessária a instalação da plataforma ESP32 no compilador do Arduino IDE, realizada por meio do gerenciador de placas do Arduino IDE, em suas configurações (SANTOS; SANTOS, 2016). Tal configuração é disponibilizada em código aberto pela própria fabricante do ESP32, a Espressif.

A figura 16 ilustra a configuração do tipo de microcontrolador conectado à porta serial (COM) do notebook, no programa Arduino IDE. Para corretas compilação e gravação de firmware. No caso, o ESP32 é compilado com o tipo FireBeetle-ESP32, por tal configuração ser mais versátil e abrangente às diversas arquiteturas presentes de ESP32, proporcionando gravação e compilação corretas para a placa.

Figura 16 – Configuração do microcontrolador no Arduino IDE.

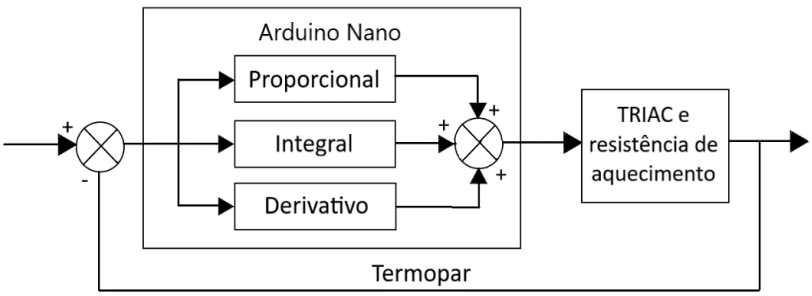


Fonte: Autoria própria.

4.4. Controle

Para o controle foi utilizado o algoritmo PID baseado no programa e circuito de acionamento com TRIAC proposto por Andrei Gabriel (GABRIEL, 2023). Seu funcionamento é exemplificado no diagrama da figura 17.

Figura 17 – Diagrama de controle PID.

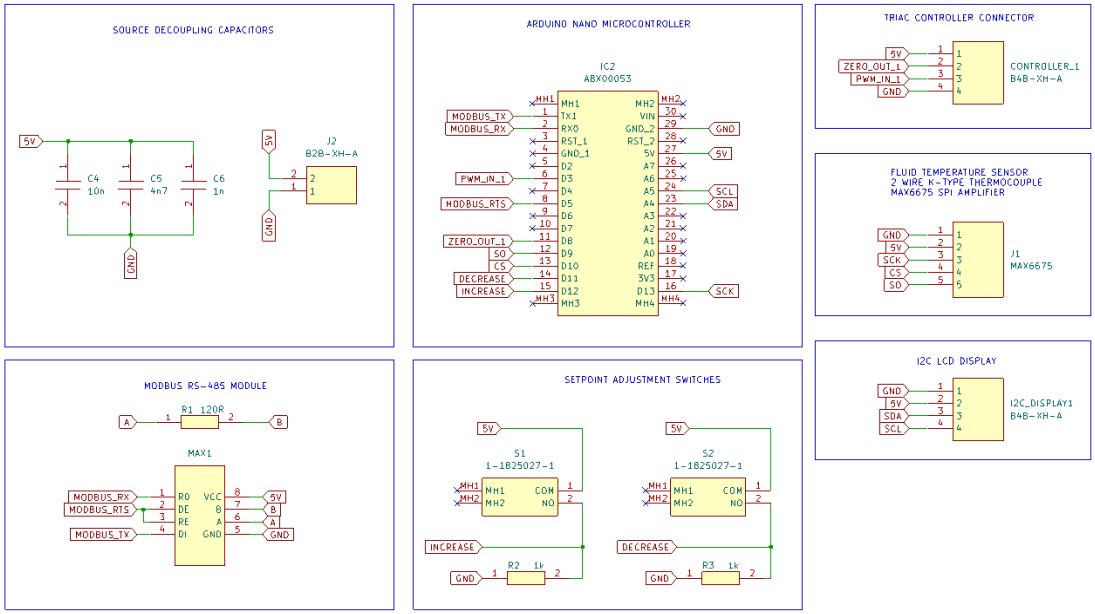


Fonte: Autoria própria.

Também, para o controle, foi utilizado o desenvolvimento presente nos trabalhos anteriores de iniciação científica pelo autor (ACCIARI; BORTOLETO; MARTINS, 2022) e (ACCIARI; BORTOLETO; MARTINS, 2023), que serviram como base para o presente trabalho.

Com a topologia de controle estabelecida, foi implementado o circuito elétrico do sistema, ilustrado na figura 18, contemplando as conexões para o módulo de acionamento e MODBUS, módulo transceptor MAX485 de RS-485, módulo leitor de termopar RTD MAX6675 e o microcontrolador Arduino Nano.

Figura 18 – Esquema elétrico do circuito.



Fonte: Autoria própria.

Destaca-se no esquema elétrico da figura 18 o microcontrolador Arduino Nano, cujos pinos de GPIOs são conectados aos periféricos de leitura de temperatura, transceptor RS-485, botões de incremento e decremento, display LCD I2C e capacitores de desacoplamento.

Para correta leitura da temperatura, foi utilizado o módulo MAX6675 condicionamento de sinal de temperatura obtido a partir do termopar tipo K. Deste modo, o periférico envia as leituras realizadas por meio do protocolo SPI para o Arduino Nano.

De modo semelhante, também foi adicionado o módulo MAX485, que se trata de um transceptor *half-duplex* de RS-485, utilizado para a interface MOSBUS RTU, tal interface é

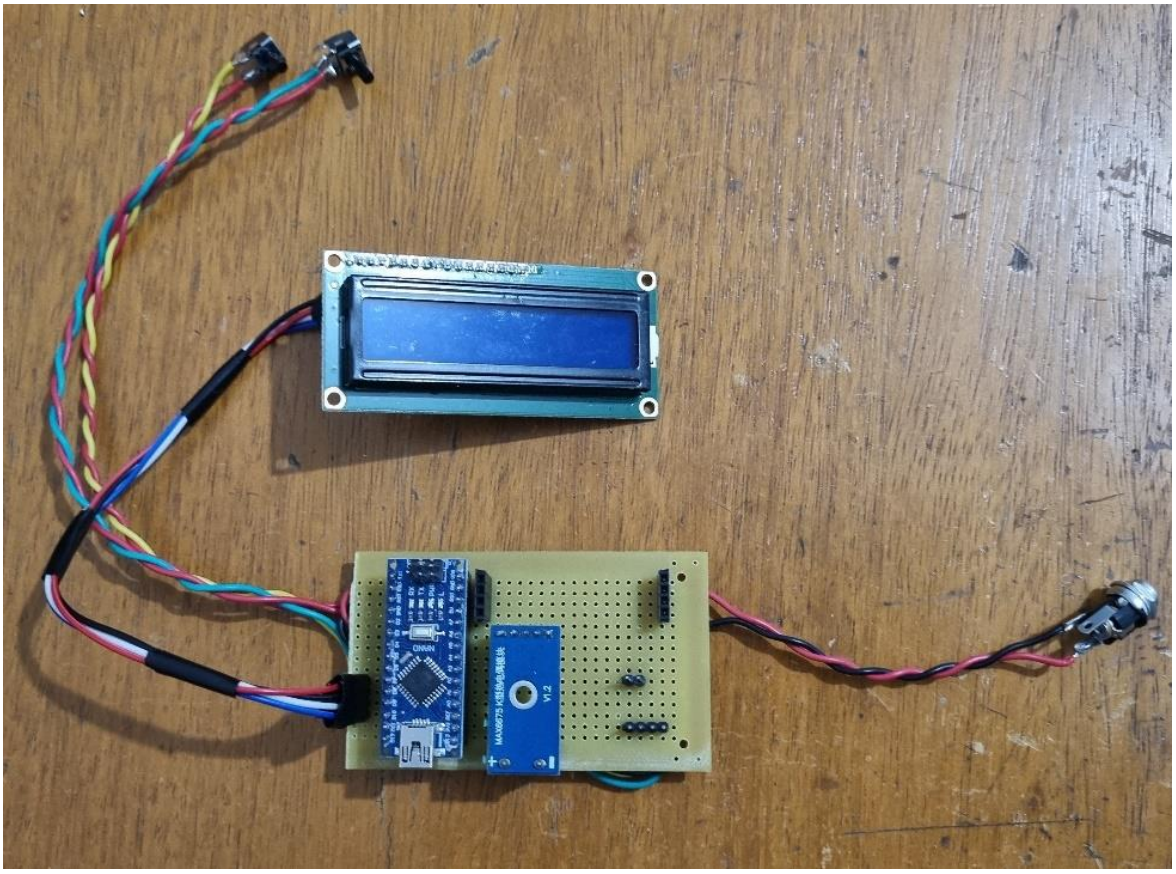
conectada com os terminais de comunicação serial TXD e RXD do Arduino, permitindo-se o envio de dados via MODBUS RTU para o circuito de monitoramento com ESP32.

Após a implementação em esquema, o circuito foi montado em placas perfuradas padrão. Bem como as placas foram cortadas para caber nos gabinetes definidos, permitindo-se a fixação por pressão.

A figura 19 ilustra o circuito de controle montado na placa com o Arduino Nano. Com os componentes na ordem de cima para baixo, esquerda para direita: display LCD I2C, Arduino Nano, módulo MAX6675, ainda sem o módulo MAX485 para RS-485.

Bem como foram adicionadas chaves tácteis para seleção de *setpoint* de temperatura desejada.

Figura 19 – Placa com circuito de controle baseado em Arduino Nano.

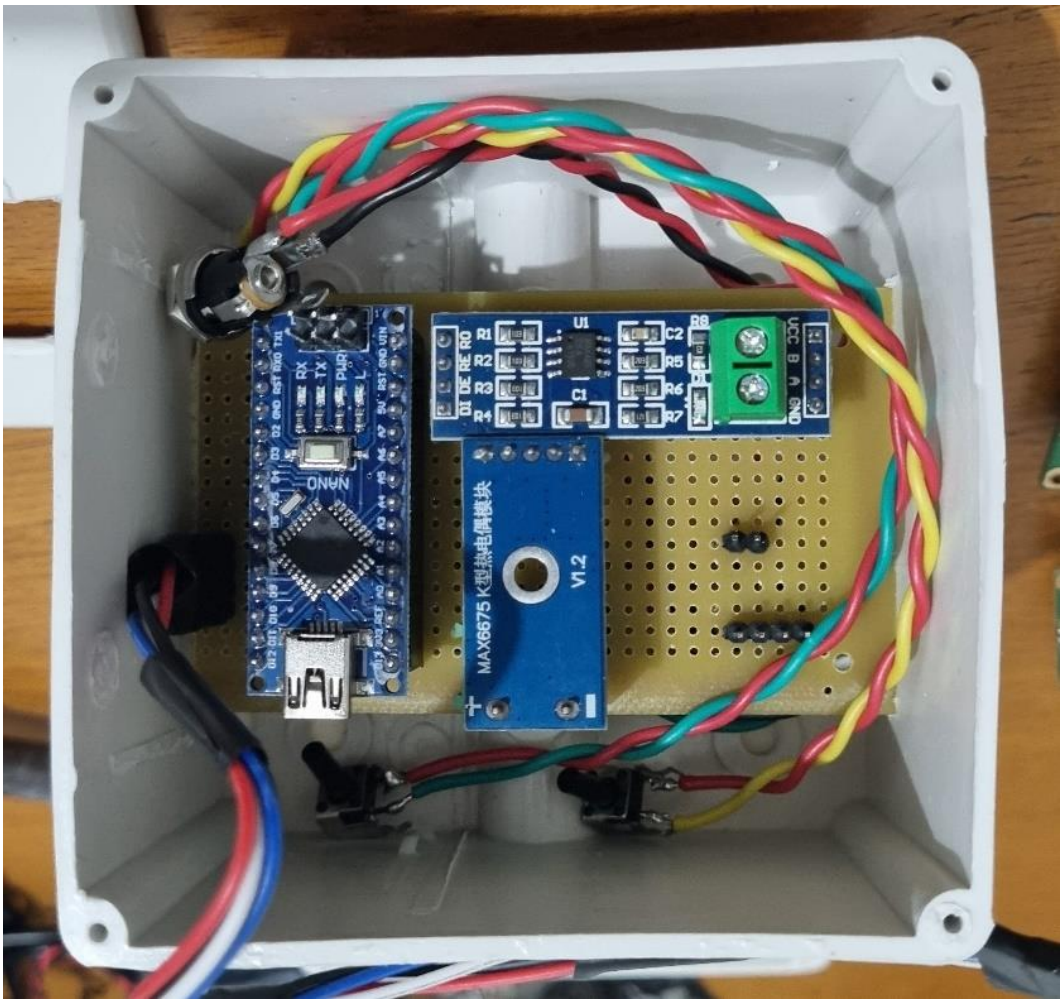


Fonte: Autoria própria.

Deste modo, a figura 19 ilustra a implementação do circuito em placa padrão perfurada, usando técnica de montagem por solda PTH, com o circuito implementado por cabos na parte inferior da placa.

Já a figura 20 ilustra o circuito de controle montado com gabinete, com o microcontrolador Arduino Nano, com o módulo MAX485 para RS-485, cabeamento de alimentação e chaves para seleção de *setpoint*.

Figura 20 – Placa de controle montada em caixa com conexões.



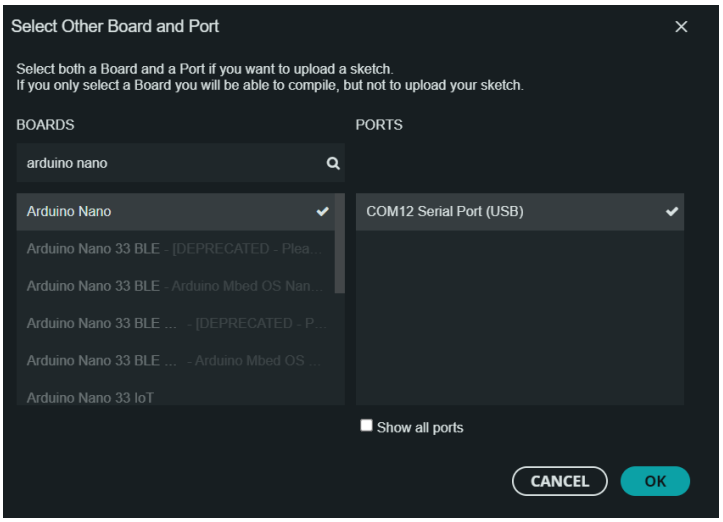
Fonte: Autoria própria.

A montagem ilustrada na figura 20 permitiu maior resistência mecânica ao circuito, bem como permitiu a implementação de IHM com botões e display LCD I2C fixados na parte frontal do gabinete de plástico.

Bem como, a montagem no gabinete auxiliou na proteção do circuito contra intempéries relacionadas à respingos de água do circuito hidráulico, permitindo-se maior confiabilidade de funcionamento do circuito nos testes implementados com fluxo de água.

Para a correta gravação do programa no Arduino Nano, foi necessário atribuir o tipo de placa “Arduino Nano” na tela de configuração do software Arduino IDE, conforme ilustra a figura 21.

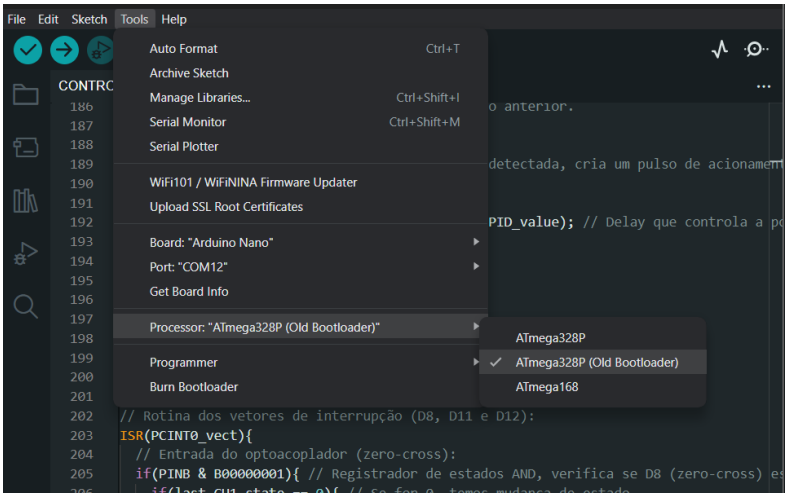
Figura 21 – Configuração “Arduino Nano” no Arduino IDE.



Fonte: Autoria própria.

Adicionalmente, foi necessário também adequar o Arduino IDE para a opção de *bootloader* antigo do microcontrolador ATmega328P, presente no Arduino Nano. Isso se deve à compatibilidade do microcontrolador utilizado somente com o tal *bootloader* antigo. A figura 22 ilustra a sequência de comandos necessários para a modificação do *bootloader* para a versão antiga “ATmega328P (Old Bootloader)”.

Figura 22 – Configuração de *bootloader* no Arduino IDE.



Fonte: Autoria própria.

Junto ao circuito de controle, foi desenvolvido também o programa do anexo B, em linguagem C++, no framework Arduino, para o Arduino Nano atuar no monitoramento de sensores, controle de aquecimento e envio de dados por MODBUS.

O quadro 10 denota o trecho de código responsável pelo monitoramento do sensor de temperatura PT100 e pelo controle PID aplicado ao aquecimento no Arduino Nano.

Quadro 10 – Código do controle PID aplicado ao aquecimento.

```
water_temperature = thermocouple.readCelsius(); // Temperatura em °C.
PID_error = setpoint - water_temperature; // Calcula o erro PID.
// Constante integral -> Afeta erros acima de 30 °C:
if(PID_error > 30){
  PID_i = 0;
}
PID_p = kp * PID_error; // Calcula atuação P (proporcional).
PID_i = PID_i + (ki * PID_error); // Calcula atuação I (integral).
timePrev = Time; // Tempo anterior é armazenado antes da leitura de tempo atual.
Time = millis(); // Leitura atual de tempo.
elapsedTime = (Time - timePrev) / 1000; // Tempo decorrido.
PID_d = kd*((PID_error - previous_error)/elapsedTime); // Calcula atuação D (derivativa).
PID_value = PID_p + PID_i + PID_d; // Calcula o valor PID total.

// Definição do espectro de delay entre 0 e 7400 µs:
if(PID_value < 0){
  PID_value = 0;
  }
if(PID_value > 7400){
  PID_value = 7400;
  }
```

Fonte: Autoria própria.

De outro modo, o quadro 11 delimita o trecho de código responsável pelo envio de dados MODBUS pelo Arduino Nano atuando como dispositivo servidor ou *slave* da rede MODBUS RTU do sistema de monitoramento.

Quadro 11 – Código do envio de dados MODBUS.

```
   // Envio de dados MODBUS:
   mb.slave(SLAVE_ID); // Cria um slave modbus com ID = SLAVE_ID.
   // Temperatura fluido:
   mb.addIreg(1); //Adiciona ao slave um registro de endereço 1.
   mb.Ireg(1,water_temperature*100); //Atribui ao registro 1 do slave o valor de temperatura do fluido.
   // Setpoint temperatura fluido:
   mb.addIreg(2); //Adiciona ao slave um registro de endereço 2.
   mb.Ireg(2,setpoint*100); //Atribui ao registro 1 do slave o valor de setpoint de temperatura do fluido.
   mb.task(); //Necessário para rodar nossa tarefa de slave.
   yield(); //Necessário para rodar nossa tarefa de slave.
```

Fonte: Autoria própria.

Tal código do quadro 11 usa a biblioteca modbus-esp8266 (EMELIANOV; BARBOSA, 2024), que apresentou melhor performance e comunicação de acordo com o padrão do protocolo MODBUS RTU, com uso do barramento RS-485.

Do mesmo modo que expresso no quadro 6 para o programa de monitoramento, no programa de controle aplicado ao Arduino Nano, foi implementado o mesmo código de detecção de erros para a interface MODBUS RTU.

Ademais, o quadro 12 mostra o trecho do programa responsável pela detecção e acionamento do interruptor de detecção de pontos de zero (zero-crossing) para o correto acionamento do TRIAC de acordo com a frequência e fase da forma de onda da alimentação elétrica do aquecedor e, consequentemente, permite o controle de temperatura do fluido.

Quadro 12 – Código do interruptor detector de pontos de zero.

```
  // Se a interrupção do sinal de zero-cross é detectada, cria um pulso de acionamento de 100 µs.
  if (zero_cross_detected)
    {
     delayMicroseconds(maximum_firing_delay - PID_value); // Delay que controla a potência AC de saída.
     digitalWrite(firing_pin,HIGH);
     delayMicroseconds(100);
     digitalWrite(firing_pin,LOW);
     zero_cross_detected = false;
    }
```

Fonte: Autoria própria.

O quadro 13 delimita a função de vetores interruptores responsáveis pelo ajuste de *setpoint* do programa.

Quadro 13 – Código dos vetores interruptores de botões do *setpoint*.

```
// Rotina dos vetores de interrupção (D8, D11 e D12):
ISR(PCINT0_vect){
  // Entrada do optoacoplador (zero-cross):
  if(PINB & B00000001){ // Registrador de estados AND, verifica se D8 (zero-cross) está em nível alto.
    if(last_CH1_state == 0){ // Se for 0, temos mudança de estado.
      zero_cross_detected = true; // Mudança de estado detectada, utiliza-se os pontos de subida e descida do sinal zero-cross.
    }
  }
  else if(last_CH1_state == 1){ // Se zero-cross estiver em nível 0 o último estado era 1, então há mudança de estado.
    zero_cross_detected = true; // Mudança de estado detectada, utiliza-se os pontos de subida e descida do sinal zero-cross.
    last_CH1_state = 0; // Armazena o estado atual como último estado para o próximo loop.
    }
    // Entrada do botão increase:
    if(PINB & B00001000){ // Registrador de estados AND, verifica se D11 (increase) está em nível alto.
      if (!pressed_1)
      {
        setpoint = setpoint + 5; // Aumenta a temperatura em passos de 5 °C.
        delay(20);
        pressed_1 = true;
      }
    }
    else if (pressed_1)
    {
      pressed_1 = false;
    }

    // Entrada do botão decrease:
    if(PINB & B00010000){ // Registrador de estados AND, verifica se D12 (decrease) está em nível alto.
      if (!pressed_2)
      {
        setpoint = setpoint - 5; // Diminui a temperatura em passos de 5 °C.
        delay(20);
        pressed_2 = true;
      }
    }
    else if (pressed_2)
    {
      pressed_2 = false;
    }
}
```

Fonte: Autoria própria.

O uso de interruptor para a execução dessa rotina, permite com que somente se o botão for pressionado, a função será executada, obtendo-se um código mais rápido e com menor uso de processamento. Com isso o processamento é mais bem utilizado para o controle PID.

Com isso, foi possível obter um programa com algoritmo de PID ajustável às diversas condições de temperatura que podem ser atingidas pela resistência e fluxo de fluido.

4.5. Acionamento elétrico

O acionamento elétrico foi realizado com o uso da variação de potência na carga resistiva de aquecimento utilizando-se a metodologia de modulação de ângulo de disparo (POMILIO, 2014) sobre o TRIAC para controlar a tensão eficaz sobre a carga resistiva (BRAGA, 2024), que consequentemente, varia a potência dissipada em forma de calor pela resistência por meio do efeito Joule (HALLIDAY; RESNICK; WALKER, 2016).

Como tal topologia utiliza tensões AC de 220 V, o circuito necessita de sincronismo com a frequência da rede elétrica para controlar o ângulo de disparo. Tal controle é feito com a alimentação de um sinal detector de pontos de zero.

Tais sinais de pontos de zero e de acionamento do TRIAC são condicionados e isolados por meio de fotoacopladores, de modo a isolar as portas do microcontrolador Arduino Nano (ATmega328P) das tensões elétricas AC da ordem de 220 V.

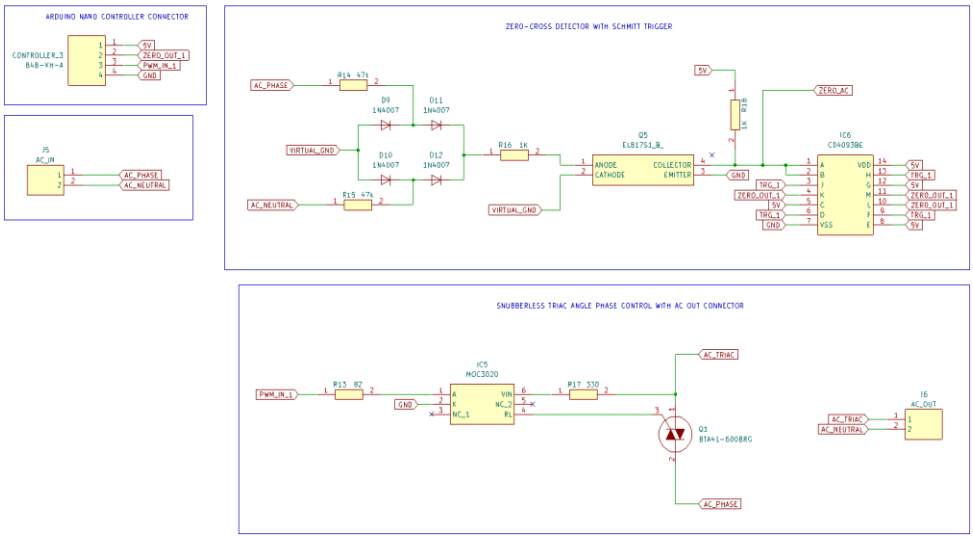
Com isso, acrescentou-se camadas de confiabilidade sobre os trabalhos anteriores PIBITI 2021 (ACCIARI; BORTOLETO; MARTINS, 2022) e PIBITI 2022 (ACCIARI; BORTOLETO; MARTINS, 2023) desenvolvidos pelo autor, que foram bases para as melhorias denotadas no presente trabalho.

Tais camadas de confiabilidade podem ser descritas como: adição de sistema MODBUS para supervisão do sistema, separação do dispositivo controlador em relação ao dispositivo de monitoramento e isolamento de sinais AC oriundos da rede elétrica.

Para correto dimensionamento em função da corrente utilizada pelo aquecedor resistivo de 6800 W, foi utilizado o TRIAC BTA41, da ST Microelectronics (STMICROELECTRONICS, 2023), que opera na faixa de corrente desejada, com dimensionamento dobrado, para suporte a picos, harmônicas e flutuações de tensão e corrente sobre o sistema. Permitindo-o atuar dentro da faixa de corrente de 0 a 25 A sem superaquecimento dos condutores.

A figura 23 ilustra o esquema elétrico do circuito de atuação eletrônica com circuitos integrados de TRIAC, fotoacoplador e disparador Schmitt.

Figura 23 – Esquema do circuito de atuação eletrônica.



Fonte: Autoria própria.

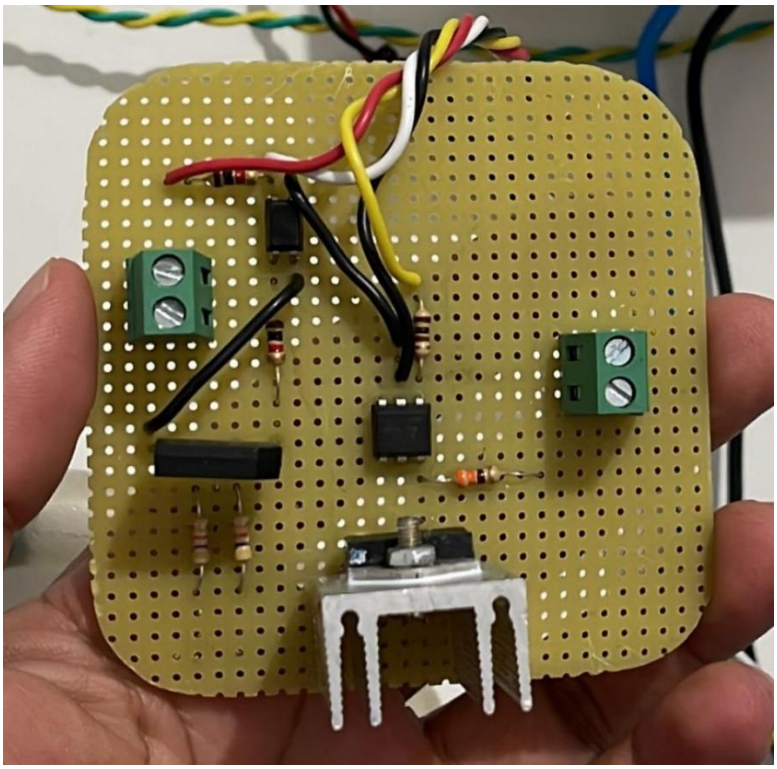
O TRIAC tem como função controlar o ângulo de disparo da senoide da tensão de alimentação AC sobre a resistência. Com seu controle, a tensão eficaz obtida sobre a resistência varia de acordo com o ângulo de disparo, permitindo-se controle de potência dissipada pelo efeito Joule e, consequentemente, aumentando e diminuindo o aquecimento do fluido.

O fotoacoplador tem a função de isolar galvanicamente a tensão de alimentação (na ordem de 220 V) da tensão de controle (na ordem de 5 V). Onde o sinal de 5 V aciona com luz a tensão de alimentação de 220 V. Protegendo o circuito de controle, o usuário, reparador e impedindo danos causados pela fuga de alta tensão sobre os microcontroladores do sistema.

O disparador Schmitt (em inglês, *Schmitt trigger*) gera onda quadrada para o detector de pontos de zero (zero-crossing), de modo a o sinal obtido ser mais inteligível para portas digitais interruptoras do microcontrolador.

A figura 24 ilustra a placa com circuito de atuação eletrônica. O circuito da figura 23 foi implementado sobre placa padrão furada com dissipador de calor sobre o TRIAC para evitar superaquecimento do componente ou placa.

Figura 24 – Placa com circuito de atuação eletrônica.



Fonte: Autoria própria.

Foi utilizada a técnica de montagem de componentes PTH, do inglês *point trought hole*, ou pino através de buraco (https://www.arossat.com.br/montagem-componentes-pth), permitindo-se a solda nas ilhas com a correta fixação e conexão entre os componentes eletrônicos utilizados para a montagem implementada na placa.

A placa também foi cortada com as bordas em filetes côncavos para correta fixação da placa no gabinete. A escolha de um gabinete plástico foi de encontro aos intuitos de proteger o usuário do contato com altas tensões na ordem de 220 V, bem como de melhorar a resistência mecânica do circuito.

Com tal montagem aplicada a gabinetes, foi implementado o protótipo geral do sistema, discutido dentre os resultados obtidos na elaboração do presente trabalho, com indicação de posições, montagem e funcionalidade de cada setor na figura 25.

5 RESULTADOS E DISCUSSÕES

Os presentes resultados foram obtidos por meio da modelagem e prototipagem de circuitos eletrônicos, testes de comunicação MODBUS entre sistema supervisório e sistema de controle, testes de envio e recebimento de dados MQTT, prototipagem de circuitos eletrônicos para supervisório, controle e acionamento, dimensionamento de circuito elétrico e sistema hidráulico.

Com isso, foram elaborados testes de eficiência de aquecimento em cenários com e sem distúrbio de temperatura.

Por fim, foram realizados testes com dashboard e responsividade do sistema, de acordo com os parâmetros de latência de rede, intervalo de processamento do programa e responsividade do dashboard com Node-RED.

Em suma, sintetiza-se o protótipo implementado como um sistema embarcado de supervisório que monitora um sistema de controle aplicado no controle de aquecimento.

5.1. Projeto eletrônico dos circuitos envolvidos

A plataforma KiCad foi utilizada e se mostrou como uma ferramenta adequada para circuitos complexos com funcionalidades e automações que permitiram o correto dimensionamento dos circuitos de todos os sistemas envolvidos no protótipo implementado, considerando suas respectivas interligações ilustradas nas figuras 13, 18 e 23.

5.2. Interfaces de IHM

Tanto no hardware de monitoramento, quanto no hardware de controle, foram implementadas plataformas de interação humana (IHMs). Tais plataformas são compostas por displays LCD alfanuméricos de 16 caracteres/linha com 2 linhas (16×2 caracteres), resultando na resolução máxima de 32 caracteres na tela a cada vez. Tal resolução se mostrou boa para a supervisão de valores de temperatura.

Para economia de portas do microcontrolador, o uso do expansor de 8 bits usando protocolo I2C foi aplicado. Como o protocolo I2C tem velocidade máxima de 100 kbits/s, os dados não foram afetados no *display* ao se utilizar tal metodologia, haja vista que para 32 caracteres são necessários somente 256 bits e endereços de posicionamento de *display*.

Deste modo, no supervisório é possível ler os dados de fluxo do fluido e de temperatura obtida do controlador via MODBUS. Bem como, no display do controlador, foi possível analisar a diferença entre o *setpoint* e o valor de mensura em tempo real.

No caso do controlador, foram implementados botões tácteis usados para ajuste de *setpoint* de temperatura com passos de 5 °C a cada toque, para fácil ajuste. Tais botões com interruptores no programa apresentaram funcionamento correto e fluido, sem a necessidade de utilização de circuito eliminador de *debouncing* digital oriundo de transientes.

5.3. Algoritmos implementados

A separação de sistema de monitoramento e de controle permitiu a independência de processos e aumentou a confiabilidade de suas respectivas atuações.

Com isso, percebeu-se que em aplicações de controle, o Arduino Nano apresentou respostas concretas. Isso se deve pelo fato de o programa utilizar interruptores próprios da arquitetura AVR, que é a arquitetura utilizada pelo ATMEGA328P, por sua vez, utilizado no Arduino Nano.

Bem como o uso de ESP32 para comunicação Ethernet, MQTT, estruturador JSON, *datalogger* e cliente (master) MODBUS, se mostrou com ótima performance, haja vista a capacidade de o ESP32 ter dois núcleos de processamento, permitindo a execução simultânea de processamento interno do hardware dedicado.

Deste modo, percebeu-se que ambos os sistemas foram dimensionados para tarefas aos quais performam melhor em cada tarefa, com suas respectivas particularidades de arquiteturas internas usadas em prol de melhor performance do sistema geral.

Tal escolha refletiu em topologia de controle e monitoramento mais resistente e confiável. Haja vista que a topologia implementada permite redundâncias em um cenário de controle com camada de supervisão.

5.4. Implementação da comunicação via MQTT

Em testes anteriores com topologia Wi-Fi, obteve-se valores de latência abaixo de 100 ms. Já com a comunicação por rede cabeada, a latência apresentou valores abaixo de 30 ms. O que indica que a implementação de topologia com rede cabeada foi melhorado.

Ademais, o broker HiveMQ apresentou uma latência adicional de 20 ms, totalizando-se 50 ms.

Deste modo, o fator controlador de tempo de atualização de dados preponderante foi o intervalo de envio de dados do ESP32.

De modo a se adequar às limitações impostas pelo broker HiveMQ em sua versão gratuita, a taxa de atualização escolhida foi de 3 segundos. Permitindo um monitoramento adequado das variáveis de temperatura. Haja vista que os processos de transferência de calor envolvidos no presente trabalho não apresentam variações bruscas de temperatura.

Cabe ressaltar que o uso do protocolo JSON permitiu o envio de vários dados em um único bloco e, com isso, contornou a limitação de quantidades de mensagem por segundo do broker em versão gratuita utilizado. Deste, modo, trata-se de um sistema mais eficiente e arquitetura mais robusta em relação ao sistema anterior, que apresentava dados MQTT isolados.

5.5. Estudo de eficiência e estabilidade do controlador PID

O controle PID foi projetado utilizando-se ajuste manual por meio do ajuste das constantes Kp, Ki e Kd no algoritmo de controle, utilizando-se potenciômetros de 100 kΩ nas entradas analógicas A1, A2 e A3 do Arduino Nano. Deste modo, com cada potenciômetro atuando sobre cada constante do controle, foi analisado qual apresentou resposta com menor tempo de resposta.

De tal modo, foi utilizada também a auto sintonia do MATLAB para o sistema com constantes semelhantes ao simulado. Porém, não apresentou resultados de constantes semelhante ao obtido com o ensaio experimental com potenciômetros.

De modo experimental, foram obtidas as constantes: proporcional Kp = 203 [], integral Ki= 7,2 [] e derivativa Kd = 1,04 [], que produziram resultados com menor *overshoot* de temperatura e oscilação amortecida, obtendo-se os resultados experimentais da tabela 1.

Deste modo, pode-se traçar uma relação entre o ângulo de ataque e a temperatura, de modo que em 0°, o aquecimento foi de aproximadamente 9 °C a cada segundo, enquanto, a 90° obteve-se um aumento de temperatura de aproximadamente 4 °C a cada segundo.

Com isso, tem-se que, conforme o aumento do ângulo de disparo, a taxa de aquecimento diminui.

Para definir-se a estabilidade do sistema, foi realizado um teste em que se mediu o tempo de aquecimento para se obter uma temperatura final semelhante à temperatura do *setpoint*. Em um caso foi utilizada água à temperatura ambiente (aproximadamente 20 °C), enquanto, em outro caso foi utilizado água à 10 ºC para se utilizar de um distúrbio no processo de controle PID.

Os dois casos foram medidos com *setpoints* de temperatura de 32 °C e 38 °C, para se avaliar o comportamento de tempo de aquecimento para cada uma das temperaturas com um fluxo de aproximadamente 14 mL/s. A tabela 1 indica os valores obtidos para os ensaios de estabilidade do sistema em função da adição de distúrbio.

Tabela 1 – Estudo de estabilidade de temperatura.

***Setpoint* (°C)**	**32,0**	**38,0**	**32,0**	**38,0**
Distúrbio (°C)	—	—	10	10
Termopar (°C)	31,8	37,9	31,9	38,1
Tempo (s)	2,4	2,6	7,4	7,9

Fonte: Autoria própria.

Com o estudo de estabilidade elaborado, obteve-se que o algoritmo MQTT permitiu o transporte de dados via internet das coisas de modo rápido e com ausência de erros, apresentando latência de 20 ms. A temperatura apresentou estabilidade e erro de 0,3 °C.

5.6. Protótipo geral do sistema

O sistema desenvolvido pode ser classificado em quatro camadas: supervisório, monitoramento, controle e atuação. Com isso, se abstrai o caminho de dados em hardware e software de forma semelhante à utilizada em sistemas de arquitetura SCADA, com terminal mestre e remoto.

O protótipo desenvolvido englobou o dashboard de interação homem-máquina em página do Node-RED, circuitos de monitoramento, controle e atuação eletrônica, sensoriamento de fluxo, temperatura do fluido, temperatura externa e umidade externa dispostos em gabinetes de modo a ficarem protegidos de intempéries oriundas do sistema hidráulico.

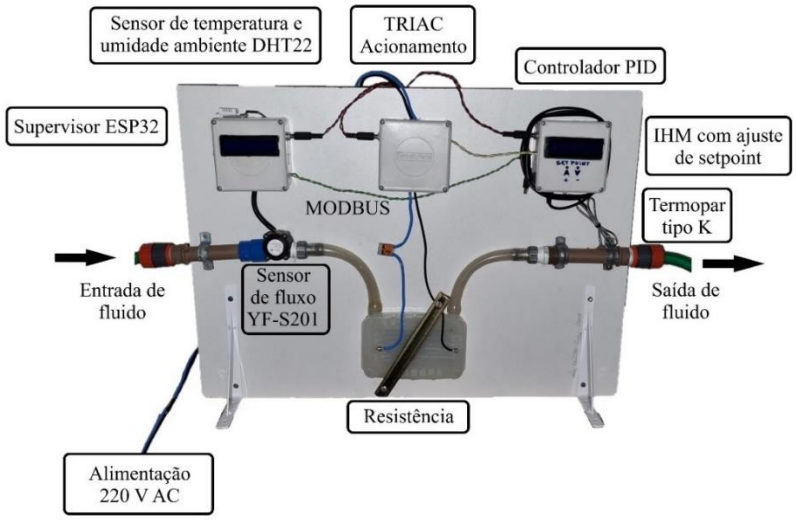
Para conectividade, o padrão ethernet se mostrou melhor do que o Wi-Fi oferecendo maior confiabilidade de conexão, bem como menor latência no tráfego de pacotes do ESP32 com a WEB. Bem como os sinais dos sensores e as interferências do sistema de atuação eletrônica não foram significativas para afetar o funcionamento do circuito.

Os três módulos de circuitos (monitoramento, controle e atuação) foram interligados por cabeamento de par trançado para comunicações de dados, que se mostraram como eficientes na diminuição de EMI (interferências eletromagnéticas). Bem como foram fixados em uma chapa de madeira de 40 cm x 60 cm para facilitar a instalação, manutenção e demonstração dos componentes do sistema.

Tal montagem apresentou versatilidade, por ser fixa e aplicável em diversos cenários sem a necessidade de ajustes de nível da ordem de centímetros para correto funcionamento do sistema.

A figura 25 ilustra o protótipo implementado do sistema geral implementado, com indicações de cada dispositivo localizado em sua respectiva região dentro do conjunto do protótipo.

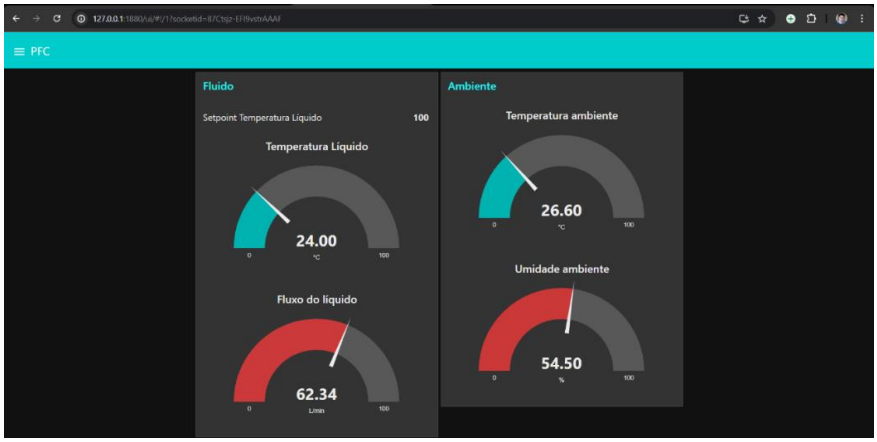
Figura 25 – Protótipo do sistema geral implementado.



Fonte: Autoria própria.

A figura 26 ilustra o dashboard implementado no Node-RED com os valores recebidos do sistema de monitoramento, com informações publicadas em MQTT pelo ESP32, com auxílio de suas respectivas estruturas JSON.

Figura 26 – Dashboard com valores recebidos do ESP32.



Fonte: Autoria própria.

6 CONCLUSÃO

Foi possível obter um programa com algoritmo de PID ajustável às diversas condições de temperatura que podem ser atingidas pelos parâmetros da resistência utilizada e da quantidade de fluido a ser aquecida.

Deste modo, o sistema possibilita flexibilidade para o correto dimensionamento de cada aplicação desejada para o aquecimento. Tornando-o versátil e aplicável em diversos cenários, com diversos tipos de resistências e sistemas de diversas magnitudes.

Em suma, o sistema apresentou resultados de acordo com o dimensionamento realizado no protótipo.

Ademais, com a implementação do sistema de comunicação MODBUS, o gerenciamento do sistema de monitoramento e supervisório pode ser escalado para diversos controladores, bem como ser conectado a outros dispositivos de sensoriamento, com as devidas adaptações para a tender aos requisitos do protocolo MODBUS.

A escalabilidade do sistema de monitoramento permite a implementação de sistemas de redundância de controle, em topologias que podem utilizar dois controladores em paralelo para aplicações críticas, tais como manejo e aquecimento de fluidos na área da saúde, bem como em outros sistemas que necessitem de redundância e acompanhamento contínuo de dados.

REFERÊNCIAS

ACCIARI, Luiz Eduardo; BORTOLETO, José Roberto Ribeiro; MARTINS, Everson. **Controle de aquecimento para escoamento em baixo fluxo**. XXXIII Congresso de Iniciação Científica da Unesp, 2022. Disponível em: https://even3.blob.core.windows.net/anais/458224.pdf . Acesso em 25 de julho de 2024.

ACCIARI, Luiz Eduardo; BORTOLETO, José Roberto Ribeiro; MARTINS, Everson. **Sistema de monitoramento remoto de dispositivo para liberação de fluidos aquecidos**. XXXIV Congresso de Iniciação Científica da Unesp, 2023. Disponível em: https://even3.blob.core.windows.net/anais/584113.pdf . Acesso em 25 de julho de 2024.

ALTUS. **O que é um datalogger e por que você deveria utilizá-lo em sua aplicação?** Altus Sistemas de Automação S. A. 2017. Disponível em: https://www.altus.com.br/post/397/o-que-e-um-datalogger-e-por-que-voce-deveria-utiliza-lo-em-sua-aplicacao. Acesso em 6 de agosto de 2024.

ARDUINO. **Arduino Nano: Product Reference Manual**. Arduino S.r.l., 2024. Disponível em: https://docs.arduino.cc/resources/datasheets/A000005-datasheet.pdf. Acesso em 25 de julho de 2024.

ATMEL. **ATmega328P: 8-bit AVR Microcontroller with 32K Bytes In-System Programmable Flash**. Atmel Corporation, Inc., 2015. Disponível em: https://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-7810-Automotive-Microcontrollers-ATmega328P_Datasheet.pdf. Acesso em 25 de julho de 2024.

BLANCHON, Benoit. **ArduinoJson**. Arduino, 2024. Disponível em https://www.arduino.cc/reference/en/libraries/arduinojson/. Acesso em 5 de agosto de 2024.

BRABANDER, Frank de; SCHWARTZ, Marco. **LiquidCrystal I2C**. Arduino, 2024. Disponível em https://www.arduino.cc/reference/en/libraries/liquidcrystal-i2c/. Acesso em 5 de agosto de 2024.

BRAGA, Newton C. **Como Funciona o Triac (ART1794)**. 2024. Disponível em: https://www.newtoncbraga.com.br/index.php/como-funciona/16284-como-funciona-o-triac-art1794.html. Acesso em 5 de agosto de 2024.

CHEN et al. Warming strategies for preventing hypothermia and shivering during cesarean section: A systematic review with network meta-analysis of randomized clinical trials. **International Journal of Surgery**, n. 71, p. 21-28, set. 2019. Disponível em https://www.sciencedirect.com/science/article/pii/S174391911930233X?via%3Dihub. Acesso em 6 de junho de 2024.

EMELIANOV, Alexander; BARBOSA, André Sarmento. Modbus Library for Arduino. 2024. Disponível em: https://github.com/emelianov/modbus-esp8266. Acesso em 25 de julho de 2024.

ESPRESSIF. **ESP32 Series: Datasheet version 4.6**. Espressif Systems, 2024. Disponível em: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf. Acesso em 25 de julho de 2024.

ESPRESSIF. **ESP32-WROOM-32 Datasheet: Version 3.4**. Espressif Systems, 2023. Disponível em: https://www.espressif.com/sites/default/files/documentation/esp32-wroom-32_datasheet_en.pdf. Acesso em 25 de julho de 2024.

GABRIEL, Andrei. **220 V AC heater PID temperature control**. Electronoobs, 2023. Disponível em: https://electronoobs.com/eng_arduino_tut39.php. Acesso em 22 de maio de 2024.

GODOY, Eduardo Paciência. **Aula 8: redes MODBUS**. Notas de aula da disciplina Redes Industriais de Comunicação. UNESP. 2021.

HALLIDAY, David; RESNICK, Robert; WALKER, Jearl. **Fundamentos de física, volume 3: eletromagnetismo**. 10. ed. Rio de Janeiro: LTC, 2016. 408p.

JO, Marcelo. **Bootloader em microcontroladores STM32F0**. Embarcados. 2015. Disponível em: https://embarcados.com.br/bootloader-stm32f0/. Acesso em 6 de agosto de 2024.

KHODELI et al. Practical and Theoretical Considerations for ECMO System Development. In FIRSTENBERG, Michael S. **Extracorporeal Membrane Oxygenation – Advances in Therapy**. 1ª ed. 2016. IntechOpen, DOI: 10.5772/64267. Disponível em https://www.intechopen.com/books/extracorporeal-membrane-oxygenation-advances-in-therapy/practical-and-theoretical-considerations-for-ecmo-system-development. Acesso em 6 de junho de 2024.

LUCIDARME, Philippe. **Linearity of the ESP32 ADC**. Lulu's Blog. 2023. Disponível em https://lucidar.me/en/esp32/linearity-of-the-esp32-adc/. Acesso em 05 de agosto de 2024.

MAXIM INTEGRATED. **MAX481/MAX483/MAX485/MAX487–MAX491/MAX1487: Low-Power, Slew-Rate-Limited RS-485/RS-422 Transceivers**. Maxim Integrated Products, Inc., 2014. Disponível em https://www.analog.com/media/en/technical-documentation/data-sheets/MAX1487-MAX491.pdf. Acesso em 25 de julho de 2024.

MODICON. **Modbus Protocol Reference Guide: PI–MBUS–300 Rev. J**. Modicon, Inc., 1996. Disponível em: https://www.modbus.org/docs/PI_MBUS_300.pdf. Acesso em 25 de julho de 2024.

MOUSER. **Espressif Systems ESP32-WROOM-32 MCU Modules**. Mouser Electronics. Disponível em: https://br.mouser.com/new/espressif/espressif-esp32-wroom-32-modules/. Acesso em 5 de agosto de 2024.

NI et al. Effects of combined warmed preoperative forced-air and warmed perioperative intravenous fluids on maternal temperature during cesarean section: a prospective, randomized, controlled clinical trial. **BMC Anesthesiol**, n. 20, 48. 2020. Disponível em https://doi.org/10.1186/s12871-020-00970-7. Acesso em 6 de junho de 2024.

NODE-RED. **Running Node-RED locally**. OpenJS Foundation, 2024. Disponível em: https://nodered.org/docs/getting-started/local. Acesso em 25 de julho de 2024.

OGATA, Katsuhiko. Capítulo 8: Controladores PID e controladores PID modificados. In: OGATA, Katsuhiko. **Engenharia de controle moder**no. 5. ed. São Paulo: Pearson Prentice Hall, 2014. 809 p.

POMILIO, J. A. Capítulo 2: Técnicas de modulação de potência. In: POMILIO, J. A. Eletrônica de Potência. UNICAMP. 2014. Disponível em: https://www.dsce.fee.unicamp.br/~antenor/pdffiles/eltpot/cap2.pdf. Acesso em 5 de agosto de 2024.

ROGER, Marvin; HOANG, Khoi. **AsyncMQTT_ESP32 Library**. Github. 2022. Disponível em: https://github.com/khoih-prog/AsyncMQTT_ESP32. Acesso em 5 de agosto de 2024.

SANTOS, Rui; SANTOS, Sara. **Installing the ESP32 Board in Arduino IDE (Windows, Mac OS X, Linux)**. Random Nerd Tutorials, 2016. Disponível em: https://randomnerdtutorials.com/installing-the-esp32-board-in-arduino-ide-windows-instructions/. Acesso em 5 de agosto de 2024.

SARMENTO, André; EMELIANOV, Alexander. **Modbus Library for Arduino**. Github. 2024. Disponível em: https://github.com/emelianov/modbus-esp8266. Acesso em 5 de agosto de 2024.

SEGUNDO, Alan Kardek R.; RODRIGUES, Cristiano L. C. **Eletrônica de Potência e Acionamentos Elétricos**. Ouro Preto: Instituto Federal de Minas Gerais – Campus Ouro Preto, 2015. 130 p. Disponível em: https://www.ufsm.br/app/uploads/sites/413/2018/12/02_arte_eletronica_de_potencia.pdf. Acesso em 5 de agosto de 2024.

SOUZA, Fábio. **O que são sistemas embarcados?** Embarcados. 2022. Disponível em: https://embarcados.com.br/o-que-sao-sistemas-embarcados/#:~:text=Um%20sistema%20embarcado%20. Acesso em 6 de agosto de 2024.

STMICROELECTRONICS. STMicroelectronics. **BTA40, BTA41, BTB41 Datasheet: 800 V and 600 V, 40 A standard Triacs in TOP3 and RD91 package**. 2023. Disponível em https://www.st.com/en/thyristors-scr-and-ac-switches/bta41.html. Acesso em 5 de agosto de 2024.

TUNGSTEN, Yuri; NASRETDINOV, Vyacheslav. **Diagrama do TRIAC como dois SCRs ligados em antiparalelo**. Wikimedia Foundation, 2022. Disponível em https://commons.wikimedia.org/wiki/File:Esquema_interno_do_TRIAC.svg. Acesso em 5 de agosto de 2024.

URQUIZO, Arturo. **PID controller overview**. Wikimedia Foundation, 2011. Disponível em https://commons.wikimedia.org/wiki/File:PID_en.svg. Acesso em 5 de agosto de 2024.

WEG. **Modbus RTU PLC300: Manual do usuário**. WEG Equipamentos Elétricos S.A., 2013. Disponível em: https://static.weg.net/medias/downloadcenter/h14/h0d/WEG-plc300-comunicacao-modbus-rtu-10000850708-manual-portugues-br.pdf. Acesso em 25 de julho de 2024.

GLOSSÁRIO

IoT: Internet das Coisas, do inglês, *Internet of Things*.

Controle PID: Controle proporcional, integral e derivativo.

Sistemas embarcados: Sistemas computacionais compostos por hardware e software com intuito de executar uma tarefa específica dentro de um sistema geral maior, que pode englobar diversos sistemas embarcados (SOUZA, 2022).

Datalogger: Dispositivo que registra dados atrelados ao tempo em que tais dados foram medidos (ALTUS, 2017).

CI: Circuito Integrado. Circuito Eletrônico implementado em base de material semicondutor e encapsulado de forma integrada em um formato específico para diversas aplicações.

JST: Terminal sem solda japonês, do inglês, *Japan Solderless Terminal*. Tipo de conector de placa projetado para conexões de cabos em PCB sem a necessidade de solda. Muito utilizado em dispositivos eletrônicos modernos para a fácil manutenção, conexão de periféricos e interconexão de placas num dispositivo.

IHM: Interface Homem–Máquina.

Setpoint: Ponto desejado de uma variável em um sistema de controle

Bootloader: Primeiro programa a ser executado pelo microcontrolador, na rotina de inicialização, denominada boot (JO, 2015).

PCB: Placa de Circuito Impresso, do inglês, *Printed Circuit Board*.

ANEXO A – PROGRAMA *DATALOGGER* SUPERVISÓRIO COM ESP32

O anexo A ilustra o programa de monitoramento, comunicação MODBUS e comunicação MQTT dos dados encapsulados em JSON na íntegra, utilizando-se linguagem C++ aplicada ao framework Arduino IDE.

```
/*
Nome: Programa Supervisório PFC
Versão: 4.0
Discente: Luiz Eduardo Acciari.
RA: 182270629.
Orientador: Prof. Dr. José Roberto Ribeiro Bortoleto.
Co-orientador: Prof. Dr. Everson Martins.
Tipo: Projeto Final de Curso.
Graduação: Engenharia de Controle e Automação.

V1.0 - Criação de programa supervisório.
V2.0 - Configuração do MODBUS para dois envios de valores com registradores.
V3.0 - Adequação de payload JSON.
V4.0 - Mudança de variáveis e alinhamento de novo payload JSON, de acordo com Node-RED.

Códigos de referência:
[1] <https://RandomNerdTutorials.com/esp32-mqtt-publish-dht11-dht22-arduino/>. Primeiros testes utilizaram abordagens baseadas no programa do Rui Santos.
[2] <https://github.com/khoih-prog/AsyncMQTT_ESP32/blob/main/examples/ESP32_W5500/FullyFeatured_ESP32_W5500/FullyFeatured_ESP32_W5500.ino>. Exemplo do W5500 implementado para a biblioteca do Async MQTT.
[3] <https://github.com/emelianov/modbus-esp8266>. Biblioteca modbus-8266 do Emelianov com suporte para ESP32 e ESP8266, com Master e Slave, funcionamento com MAX485.
[4] <https://www.cytron.io/tutorial/interface-water-flow-sensor-using-esp32-board>
*/

#include "defines.h" //Arquivo de definições com bibliotecas, configurações e pinouts.

extern "C" {
  #include "freertos/FreeRTOS.h" //Biblioteca do Sistema Operacional FreeRTOS.
  #include "freertos/timers.h" //Biblioteca de timers do FreeRTOS.
}

#include <ModbusRTU.h> //Biblioteca modbus-esp8266 do Emelianov.
#include <AsyncMQTT_ESP32.h>  //Biblioteca de MQTT assíncrono com suporte para conexão SPI com o Ethernet Shield W5500.
#include <ArduinoJson.h> //Biblioteca para criação de payloads JSON.
#include <LiquidCrystal_I2C.h> //Biblioteca do display LCD I2C.
#include "DHT.h" //Biblioteca do sensor de temperatura e umidade DHT22.
```

```
//Configurações do Master:
#define SLAVE_ID 3 //Endereço do dispositivo.
#define FIRST_REG 1 //Valor do registrador da primeira variável. Ex: 0x001.
#define REG_COUNT 2 //Contador de variáveis de resistradores posteriores. Ex. 2 valores: 0x001, 0x002.

// Configuração do broker MQTT:
#define MQTT_HOST         "broker.hivemq.com"        // Broker address
#define MQTT_PORT         1883

// Configuração do tópico MQTT:
const char *MQTT_PUB_TOPIC  = "geral/controlador-1/dados";
const char *MQTT_SUB_TOPIC  = "geral/controlador-1/comandos";

// GPIO's do sensor de fluxo:
#define FLOW_SENSOR  33

// GPIO do sensor de temperatura e umidade DHT22:
#define DHT1PIN 14 //Externo

// Tipo do sensor de temperatura e umidade:
#define DHTTYPE DHT22   // DHT 22

// Set the LCD number of columns and rows
int lcdColumns = 16;
int lcdRows = 2;

float water_temp;
float external_hum;
float external_temp;
float water_flux;
float setpoint_temp;

String ext_hum;
String ext_temp;
String w_temp;
String w_flux;
String set_temp;

float calibrationFactor = 4.5;
volatile byte pulseCount;
byte pulse1Sec = 0;
float flowRate;
unsigned int flowMilliLitres;
unsigned long totalMilliLitres;

// Configurações do cliente MQTT assíncrono:
AsyncMqttClient mqttClient;
TimerHandle_t mqttReconnectTimer;

unsigned long previousMillis = 0;   //Guarda o último tempo em que os valores foram publicados.
const long interval = 5000;        //Intervalo para publicar as leituras dos sensores.
```

```
ModbusRTU mb; //Inicialização do Modbus.

unsigned long reconnectAttemptStart = 0;
const unsigned long reconnectAttemptInterval = 30 * 1000; // Intervalo de tentativa de reconexão de 30 segundos
const unsigned long maxReconnectTime = 15 * 60 * 1000; // 15 minutos para tentativas de reconexão
bool isReconnectTimerActive = false;

// Function Declarations
void IRAM_ATTR pulseCounter();
bool cb(Modbus::ResultCode event, uint16_t transactionId, void* data); //Função de detecção de erros do MODBUS.
void tryReconnect();
void connectToMqtt();
void ETH_event(WiFiEvent_t event);
void printSeparationLine();
void onMqttConnect(bool sessionPresent);
void onMqttDisconnect(AsyncMqttClientDisconnectReason reason);
void onMqttSubscribe(const uint16_t& packetId, const uint8_t& qos);
void onMqttUnsubscribe(const uint16_t& packetId);
void onMqttMessage(char* topic, char* payload, const AsyncMqttClientMessageProperties& properties, const size_t& len, const size_t& index, const size_t& total);
void onMqttPublish(const uint16_t& packetId);

LiquidCrystal_I2C lcd(0x27, lcdColumns, lcdRows); //Inicialização do display LCD I2C.

// Initialização dos sensores DHT22:
DHT dht1(DHT1PIN, DHTTYPE);

void setup() {
  Serial.begin(9600, SERIAL_8N1); //Baud rate do leitor monitor serial.
  Serial2.begin(9600, SERIAL_8N1); //Baud rate da interface RS-485.
  mb.begin(&Serial2,15); //MAX-485 interface half-duplex, necessita de pino de comutação de TX/RX especificado. RE/DE do módulo. Direciona a comunicação.
  mb.master(); //Inicializa o modo Master do Modbus.
  Serial.println();

  AMQTT_LOGWARN(F("Default SPI pinout:"));
  AMQTT_LOGWARN1(F("SPI_HOST:"), ETH_SPI_HOST);
  AMQTT_LOGWARN1(F("MOSI:"), MOSI_GPIO);
  AMQTT_LOGWARN1(F("MISO:"), MISO_GPIO);
  AMQTT_LOGWARN1(F("SCK:"),  SCK_GPIO);
  AMQTT_LOGWARN1(F("CS:"),   CS_GPIO);
  AMQTT_LOGWARN1(F("INT:"),  INT_GPIO);
  AMQTT_LOGWARN1(F("SPI Clock (MHz):"), SPI_CLOCK_MHZ);
  AMQTT_LOGWARN(F("========================"));

  // Configuração inicial, incluindo a configuração do timer de reconexão
  mqttReconnectTimer = xTimerCreate(
    "mqttReconnectTimer",
    pdMS_TO_TICKS(reconnectAttemptInterval),
    pdTRUE, // pdTRUE para auto-reload, permitindo repetição
```

```
    (void*)0,
    reinterpret_cast<TimerCallbackFunction_t>(tryReconnect)
  );


  WiFi.onEvent(ETH_event);
  ETH.begin( MISO_GPIO, MOSI_GPIO, SCK_GPIO, CS_GPIO, INT_GPIO,
SPI_CLOCK_MHZ, ETH_SPI_HOST );
  //ETH.config(myIP, myGW, mySN, myDNS); //Configuração do IP estático (caso necessite
de DHCP, comentar).

  mqttClient.onConnect(onMqttConnect);
  mqttClient.onDisconnect(onMqttDisconnect);
  mqttClient.onSubscribe(onMqttSubscribe);
  mqttClient.onUnsubscribe(onMqttUnsubscribe);
  mqttClient.onPublish(onMqttPublish);
  mqttClient.setServer(MQTT_HOST, MQTT_PORT);

  dht1.begin();

  pinMode(FLOW_SENSOR, INPUT_PULLUP);

  pulseCount = 0;
  flowRate = 0.0;
  flowMilliLitres = 0;
  totalMilliLitres = 0;
  previousMillis = 0;

  attachInterrupt(digitalPinToInterrupt(FLOW_SENSOR), pulseCounter, FALLING);

  // Inicialização do LCD:
  lcd.init();
  // Ligar luz traseira do LCD:
  lcd.backlight();
}

void loop() {
    unsigned long currentMillis = millis();
    if (currentMillis - previousMillis >= interval) {
        previousMillis = currentMillis;

        // Inicialização:
        external_hum = dht1.readHumidity();
        external_temp = dht1.readTemperature();

        // Conversão para Strings:
        String ext_hum = String(external_hum);
        String ext_temp = String(external_temp);
        String w_temp = String(water_temp);
        String w_flux = String(water_flux);
        //String set_temp = String(setpoint_temp);

        //Leitura MODBUS master para CDU (SLAVE 2):
        uint16_t res[REG_COUNT]; // Vetor dos registradores lidos, cujo número é determinado
```

```
pelo contador de variáveis.
      if (!mb.slave()) {    // Checa se não há conflito de linha.
        mb.readIreg(SLAVE_ID, FIRST_REG, res, REG_COUNT, cb); // Especifica
requisição de leitura do registrador de entrada (Input Register) para o slave de ID determinado
no primeiro campo da função.;
        while(mb.slave()) { // Checa se linha está livre.
          mb.task(); //Executa comandos de requisição do modbus master para o modbus slave.
          delay(10); //Atraso para melhor integridade do sinal.
        }
        water_temp = (res[0])/100; // Guarda temperatura água (saída), recebido no valor do
registrador de entrada 0x001.
        setpoint_temp = (res[1])/100; // Guarda setpoint de temperatura, recebido no valor do
registrador de entrada 0x002.

        Serial.print("\n");
        Serial.print("\nTemperatura da água (saída): ");
        Serial.print(water_temp);
        Serial.print(" °C");
        Serial.print("\nSetpoint de temperatura: ");
        Serial.print(setpoint_temp);
        Serial.print(" °C");
        Serial.print("\n");
        Serial.print("\n");
      }

      pulse1Sec = pulseCount;
      pulseCount = 0;

      water_flux = ((1000.0 / (millis() - previousMillis)) * pulse1Sec) / calibrationFactor;
      previousMillis = millis();

      flowMilliLitres = (flowRate / 60) * 1000;
      totalMilliLitres += flowMilliLitres;

      // Set cursor to first column, first row
      lcd.setCursor(0, 0);
      lcd.print("Temp: ");
      lcd.print(water_temp,3);
      lcd.print(" C");

      lcd.setCursor(0, 1);
      lcd.print("Fluxo: ");
      lcd.print(water_flux,3);
      lcd.print(" L/min");


      // JSON for Publish:
      JsonDocument docPublisher;
      docPublisher["device"]["ID"] = "DEV01";
      docPublisher["device"]["WaterTemp"]["type"] = "Water_temp";
      docPublisher["device"]["WaterTemp"]["value"] = w_temp;
      docPublisher["device"]["WaterTemp"]["unit"] = "°C";
      docPublisher["device"]["WaterFlow"]["type"] = "Water_flow";
      docPublisher["device"]["WaterFlow"]["value"] = w_flux;
```

```
      docPublisher["device"]["WaterFlow"]["unit"] = "L/min";
      docPublisher["device"]["WaterSetpoint"]["type"] = "Water_setpoint";
      docPublisher["device"]["WaterSetpoint"]["value"] = setpoint_temp;
      docPublisher["device"]["WaterSetpoint"]["unit"] = "°C";
      docPublisher["device"]["AmbientTemp"]["type"] = "Ambient_temp";
      docPublisher["device"]["AmbientTemp"]["value"] = ext_temp;
      docPublisher["device"]["AmbientTemp"]["unit"] = "°C";
      docPublisher["device"]["AmbientHum"]["type"] = "Ambient_hum";
      docPublisher["device"]["AmbientHum"]["value"] = ext_hum;
      docPublisher["device"]["AmbientHum"]["unit"] = "%";

      String payloadPublisher;
      serializeJson(docPublisher, payloadPublisher);
      uint16_t packetIdPublisher = mqttClient.publish(MQTT_PUB_TOPIC, 0, true,
payloadPublisher.c_str());
      Serial.printf("Published Temperature to %s with packetId: %i\n", MQTT_PUB_TOPIC,
packetIdPublisher);


      // JSON Setpoint Subscriber
      JsonDocument docSubscriber;
      docSubscriber["config"]["ID"] = "DEV01";
      docSubscriber["config"]["sensor_3"]["type"] = "Fluid_setpoint";
      setpoint_temp = docSubscriber["config"]["sensor_3"]["value"];
      docSubscriber["config"]["sensor_3"]["unit"] = "°C";

      String payloadSubscriber;
      serializeJson(docSubscriber, payloadSubscriber);
      uint16_t packetIdSubscriber = mqttClient.publish(MQTT_SUB_TOPIC, 0, true,
payloadSubscriber.c_str());
      Serial.printf("Published Temperature to %s with packetId: %i\n", MQTT_SUB_TOPIC,
packetIdSubscriber);
    }
}



void IRAM_ATTR pulseCounter()
{
  pulseCount++;
}

bool cb(Modbus::ResultCode event, uint16_t transactionId, void* data) { //Função de
detecção de erros.
  if (event != Modbus::EX_SUCCESS) {
    Serial.print("Request result: 0x");
    Serial.print(event, HEX);
  }
  return true;
}

void ETH_event(WiFiEvent_t event)
{
  switch (event)
```

```
  {
#if USING_CORE_ESP32_CORE_V200_PLUS

    case ARDUINO_EVENT_ETH_START:
      Serial.println("ETH starting");
      break;

    case ARDUINO_EVENT_ETH_CONNECTED:
      Serial.println("ETH connected");
      break;

    case ARDUINO_EVENT_ETH_GOT_IP:
      Serial.println("ETH got IP");
      Serial.print("IP address: ");
      Serial.println(ETH.localIP());
      connectToMqtt();
      break;

    case ARDUINO_EVENT_ETH_DISCONNECTED:
      Serial.println("ETH lost connection");

      // ensure we don't reconnect to MQTT when no ETH
      xTimerStop(mqttReconnectTimer, 0);

      break;

    case ARDUINO_EVENT_ETH_STOP:
      Serial.println("ETH stops");

      // ensure we don't reconnect to MQTT when no ETH
      xTimerStop(mqttReconnectTimer, 0); //comentado para ver se isso impede

      break;
#else

    case SYSTEM_EVENT_ETH_CONNECTED:
      Serial.println(F("ETH Connected"));
      break;

    case SYSTEM_EVENT_ETH_GOT_IP:
      Serial.println("ETH connected");
      Serial.println("IP address: ");
      Serial.println(ETH.localIP());
      connectToMqtt();
      break;

    case SYSTEM_EVENT_ETH_DISCONNECTED:
    case SYSTEM_EVENT_ETH_STOP:
      Serial.println("ETH lost connection");

      // ensure we don't reconnect to MQTT when no ETH
      xTimerStop(mqttReconnectTimer, 0); //comentado para ver se isso impede

      break;
```

```
#endif

    default:
      break;
  }
}


void printSeparationLine()
{
  Serial.println("************************************************");
}

// Função chamada pelo timer para tentar reconectar
void tryReconnect() {
  if (!mqttClient.connected()) {
    unsigned long now = millis();
    if ((now - reconnectAttemptStart) <= maxReconnectTime) {
      Serial.println("Tentando reconectar ao MQTT...");
      connectToMqtt();
    } else {
      Serial.println("Tentativas de reconexão ao MQTT encerradas após 15 minutos.");
      xTimerStop(mqttReconnectTimer, 0); // Para o timer
      isReconnectTimerActive = false;
    }
  }
}


void onMqttConnect(bool sessionPresent)
{
  Serial.println("Connected to MQTT.");
  Serial.print("Session present: ");
  Serial.println(sessionPresent);
}

void onMqttDisconnect(AsyncMqttClientDisconnectReason reason) {
  Serial.println("Desconectado do MQTT.");

  if (!isReconnectTimerActive) {
    reconnectAttemptStart = millis(); // Inicia o período de tentativa de reconexão
    xTimerStart(mqttReconnectTimer, 0); // Inicia o timer para tentar reconectar
    isReconnectTimerActive = true;
  }
}

void connectToMqtt() {
  Serial.println("Conectando ao MQTT...");
  mqttClient.connect();
}

void onMqttSubscribe(const uint16_t& packetId, const uint8_t& qos)
{
  Serial.println("Subscribe acknowledged.");
```

```
  Serial.print("  packetId: ");
  Serial.println(packetId);
  Serial.print("  qos: ");
  Serial.println(qos);
}

void onMqttUnsubscribe(const uint16_t& packetId)
{
  Serial.println("Unsubscribe acknowledged.");
  Serial.print("  packetId: ");
  Serial.println(packetId);
}

void onMqttMessage(char* topic, char* payload, const AsyncMqttClientMessageProperties&
properties,
                   const size_t& len, const size_t& index, const size_t& total)
{
  (void) payload;

  Serial.println("Publish received.");
  Serial.print("  topic: ");
  Serial.println(topic);
  Serial.print("  qos: ");
  Serial.println(properties.qos);
  Serial.print("  dup: ");
  Serial.println(properties.dup);
  Serial.print("  retain: ");
  Serial.println(properties.retain);
  Serial.print("  len: ");
  Serial.println(len);
  Serial.print("  index: ");
  Serial.println(index);
  Serial.print("  total: ");
  Serial.println(total);
}


void onMqttPublish(const uint16_t& packetId)
{
  Serial.println("Publish acknowledged.");
  Serial.print("  packetId: ");
  Serial.println(packetId);
}
```

Fonte: Autoria própria.

ANEXO B – PROGRAMA CONTROLADOR PID COM ARDUINO NANO

O anexo B ilustra o programa de monitoramento, comunicação MODBUS e comunicação MQTT dos dados encapsulados em JSON na íntegra, utilizando-se linguagem C++ aplicada ao framework Arduino IDE.

```
/*
Nome: Programa Controlador PFC
Versão: 2.0
Discente: Luiz Eduardo Acciari.
RA: 182270629.
Orientador: Prof. Dr. José Roberto Ribeiro Bortoleto.
Co-orientador: Prof. Dr. Everson Martins.
Tipo: Projeto Final de Curso.
Graduação: Engenharia de Controle e Automação.

Códigos de referência:
[1] <https://electronoobs.com/eng_arduino_tut39_code1.php>. Programa-base para controle PID.
[2] <https://github.com/emelianov/modbus-esp8266>. Biblioteca modbus-8266 do Emelianov com suporte para ESP32 e ESP8266, com Master e Slave, funcionamento com MAX485.
*/

/*
 Pinagens:

 Módulo Max6675   ==>   Arduino
 CS             ==>    D10
 SO             ==>    D9
 SCK             ==>    D13
 Vcc            ==>    Vcc (5v)
 Gnd             ==>    Gnd

 Display I2C     ==>   Arduino
 SCL            ==>    A5
 SDA             ==>    A4
 Vcc            ==>    Vcc (5v)
 Gnd            ==>    Gnd
 */

// Definições:
#define SLAVE_ID 3 //Atribuição de endereço modbus slave a este dispositivo (device ID).

// Importação de bibliotecas:
#include <max6675.h> // Biblioteca do conversor de RTD Tipo K.
#include <Wire.h> // Biblioteca de comunicação do protocolo Wire.
#include <LiquidCrystal_I2C.h> //Biblioteca do Display LCD I2C.
```

```
#include <ModbusRTU.h> // Biblioteca modbus-esp8266 do Emelianov.

// GPIO's de acionamento:
int firing_pin = 3;
int zero_cross = 8;

// GPIO's dos botões:
int increase_pin = 11;
int decrease_pin = 12;

// GPIO's do termopar -> Módulo MAX6675 -> Comunicação SPI
int water_temp_DO = 9;
int water_temp_CS = 10;
int water_temp_CLK = 13;

// Variáveis:
int last_CH1_state = 0;
bool zero_cross_detected = false;
int firing_delay = 7400;

// Variável de ajuste da frequência:
int maximum_firing_delay = 7400; // Valor de 7.4 ms, ou 7400 µs -> Para a frequência de alimentação elétrica de 60 Hz.
/*
Período de 60 Hz -> 16.67 ms
Metade do período -> 8.333 ms
Valor de melhor performance -> 7.400 ms -> 7400 µs.
*/

//Variáveis da função millis -> Melhor alternativa em relação ao delay:
unsigned long previousMillis = 0;
unsigned long currentMillis = 0;

// Variáveis temperatura:
int temp_read_Delay = 500; // Delay da leitura de temperatura.
int water_temperature = 0; // Temperatura do fluido.
int setpoint = 100; // Setpoint de temperatura do fluido.

// Variáveis dos botões:
bool pressed_1 = false;
bool pressed_2 = false;

// Variáveis do controle PID:
float PID_error = 0; // Erro do controle PID.
float previous_error = 0; // Erro prévio do controle PID.
float elapsedTime, Time, timePrev; // Tempo para controle PID.
int PID_value = 0; // Valor PID.

// Constantes do controle PID.
int kp = 203; // Constante proporcional.
int ki= 7.2; // Constante integral.
int kd = 1.04; // Constante derivativa.
int PID_p = 0; // Atuação proporcional.
int PID_i = 0; // Atuação integral.
```

```
int PID_d = 0; // Atuação derivativa.

// Inicialização display LCD:
LiquidCrystal_I2C lcd(0x27,20,4);  //sometimes the adress is not 0x27. Change to 0x3f if it dosn't work.

// Inicialização do módulo MAX6675 para termopar:
MAX6675 thermocouple(water_temp_CLK, water_temp_CS, water_temp_DO);

// Inicialização do Modbus:
ModbusRTU mb;



void setup() {
  // Definição de modo para os pinos:
  pinMode (firing_pin,OUTPUT);
  pinMode (zero_cross,INPUT);
  pinMode (increase_pin,INPUT);
  pinMode (decrease_pin,INPUT);

  // Configuração de interruptores:
  PCICR |= (1 << PCIE0);    //Inicia a varredura deinterruptores PCMSK0.
  PCMSK0 |= (1 << PCINT0);  //Configura a GPIO D8 (entrada zero-cross) como gatilho de mudança de estado do interruptor.
  PCMSK0 |= (1 << PCINT3);  //Configura a GPIO D11 (botão increase) como gatilho de mudança de estado do interruptor.
  PCMSK0 |= (1 << PCINT4);  //Configura a GPIO D12 (botão decrease) como gatilho de mudança de estado do interruptor.

  // Configuração display LCD:
  lcd.init();       // Inicialização da comunicação com o display LCD
  lcd.backlight();  // Liga a luz traseira do display LCD.

  //Configuração MODBUS:
  Serial.begin(9600, SERIAL_8N1); //Baud rate da interface RS-485.
  mb.begin(&Serial,5); //MAX-485 interface half-duplex, necessita de pino de comutação de TX/RX especificado. RE/DE do módulo. Direciona a comunicação.
  mb.slave(); //Inicializa o modo Slave do Modbus.
}



void loop() {
  currentMillis = millis(); // Salva o valor do timer de loop. Tempo lento, pois a leitura do sensor é de 100 ms.
  if(currentMillis - previousMillis >= temp_read_Delay){
    previousMillis += temp_read_Delay; // Incrementa o tempo para o próximo loop

    water_temperature = thermocouple.readCelsius(); // Temperatura em °C.

    PID_error = setpoint - water_temperature; // Calcula o erro PID.

    // Constante integral -> Afeta erros acima de 30 °C:
```

```
   if(PID_error > 30){
     PID_i = 0;
   }

   PID_p = kp * PID_error; // Calcula atuação P (proporcional).
   PID_i = PID_i + (ki * PID_error); // Calcula atuação I (integral).
   timePrev = Time; // Tempo anterior é armazenado antes da leitura de tempo atual.
   Time = millis(); // Leitura atual de tempo.
   elapsedTime = (Time - timePrev) / 1000; // Tempo decorrido.
   PID_d = kd*((PID_error - previous_error)/elapsedTime); // Calcula atuação D (derivativa).
   PID_value = PID_p + PID_i + PID_d; // Calcula o valor PID total.

   // Definição do espectro de delay entre 0 e 7400 µs:
   if(PID_value < 0){
     PID_value = 0;
     }
   if(PID_value > 7400){
     PID_value = 7400;
     }

   // Mostra os valores no display LCD:
   lcd.clear();
   lcd.setCursor(0,0);
   lcd.print("Set: ");
   lcd.setCursor(5,0);
   lcd.print(setpoint);
   lcd.setCursor(0,1);
   lcd.print("Temp: ");
   lcd.setCursor(6,1);
   lcd.print(water_temperature);

   // Envio de dados MODBUS:
   mb.slave(SLAVE_ID); // Cria um slave modbus com ID = SLAVE_ID.

   // Temperatura fluido:
   mb.addIreg(1); //Adiciona ao slave um registro de endereço 1.
   mb.Ireg(1,water_temperature*100); //Atribui ao registro 1 do slave o valor de temperatura
do fluido.

   // Setpoint temperatura fluido:
   mb.addIreg(2); //Adiciona ao slave um registro de endereço 2.
   mb.Ireg(2,setpoint*100); //Atribui ao registro 1 do slave o valor de setpoint de temperatura
do fluido.

   mb.task(); //Necessário para rodar nossa tarefa de slave.
   yield(); //Necessário para rodar nossa tarefa de slave.

   previous_error = PID_error; // Guarda o erro anterior.
  }

  // Se a interrupção do sinal de zero-cross é detectada, cria um pulso de acionamento de 100
µs.
  if (zero_cross_detected)
    {
```

```
    delayMicroseconds(maximum_firing_delay - PID_value); // Delay que controla a
potência AC de saída.
    digitalWrite(firing_pin,HIGH);
    delayMicroseconds(100);
    digitalWrite(firing_pin,LOW);
    zero_cross_detected = false;
   }
}



// Rotina dos vetores de interrupção (D8, D11 e D12):
ISR(PCINT0_vect){
 // Entrada do optoacoplador (zero-cross):
 if(PINB & B00000001){ // Registrador de estados AND, verifica se D8 (zero-cross) está em
nível alto.
   if(last_CH1_state == 0){ // Se for 0, temos mudança de estado.
    zero_cross_detected = true; // Mudança de estado detectada, utiliza-se os pontos de subida
e descida do sinal zero-cross.
   }
 }
 else if(last_CH1_state == 1){ // Se zero-cross estiver em nível 0 o último estado era 1, então
há mudança de estado.
   zero_cross_detected = true; // Mudança de estado detectada, utiliza-se os pontos de subida
e descida do sinal zero-cross.
   last_CH1_state = 0; // Armazena o estado atual como último estado para o próximo loop.
   }
   // Entrada do botão increase:
   if(PINB & B00001000){ // Registrador de estados AND, verifica se D11 (increase) está em
nível alto.
    if (!pressed_1)
    {
     setpoint = setpoint + 5; // Aumenta a temperatura em passos de 5 °C.
     delay(20);
     pressed_1 = true;
    }
   }
   else if (pressed_1)
   {
    pressed_1 = false;
   }

   // Entrada do botão decrease:
   if(PINB & B00010000){ // Registrador de estados AND, verifica se D12 (decrease) está
em nível alto.
    if (!pressed_2)
    {
     setpoint = setpoint - 5; // Diminui a temperatura em passos de 5 °C.
     delay(20);
     pressed_2 = true;
    }
   }
   else if (pressed_2)
   {
```

```
      pressed_2 = false;
    }
}

bool cb(Modbus::ResultCode event, uint16_t transactionId, void* data) { //Função de
detecção de erros.
  if (event != Modbus::EX_SUCCESS) {
    Serial.print("Request result: 0x");
    Serial.print(event, HEX);
  }
  return true;
}
```

Fonte: Autoria própria.