

João Eduardo Moya

**Implementação de uma Plataforma Didática de Automação
Industrial Utilizando Raspberry Pi com Integração
Multiprotocolo e Monitoramento**

UBERLÂNDIA

2026

João Eduardo Moya

Implementação de uma Plataforma Didática de Automação Industrial Utilizando Raspberry Pi com Integração Multiprotocolo e Monitoramento

Trabalho de Conclusão de Curso da Engenharia de Controle e Automação da Universidade Federal de Uberlândia - UFU - Câmpus Santa Mônica, como requisito para a obtenção do título de Graduação em Engenharia de Controle e Automação

Universidade Federal de Uberlândia – UFU

Faculdade de Engenharia Elétrica

Orientador: Prof. Dr. Fábio Vincenzi Romualdo da Silva

UBERLÂNDIA

2026

MOYA, João Eduardo

Implementação de uma Plataforma Didática de Automação Industrial
Utilizando Raspberry Pi com Integração Multiprotocolo e Monitoramento
/ **João Eduardo Moya. - UBERLÂNDIA, 2026**

Orientador: Prof. Dr. Fábio Vincenzi Romualdo da Silva

Trabalho de Conclusão de Curso - Universidade Federal de
Uberlândia - UFU Faculdade de Engenharia Elétrica. **2026.**

1. Automação Industrial 2. Raspberry Pi 3. Node-RED I. Prof. Dr. Fábio
Vincenzi Romualdo da Silva. II. Universidade Federal de Uberlândia. III.
Faculdade de Engenharia Elétrica. IV. Engenharia de Controle e Automação.

*Dedico este trabalho a
todos que me apoiaram.*

Agradecimentos

Agradeço, primeiramente, a Deus, por me conceder saúde, força, sabedoria e perseverança para superar os desafios encontrados ao longo desta caminhada e por tornar possível a conclusão desta importante etapa da minha vida.

Aos meus pais, Roberto e Carolina, expresso minha profunda gratidão pelo amor, apoio, incentivo e ensinamentos que sempre me ofereceram. Sou imensamente grato por estarem ao meu lado em todos os momentos, servindo como base fundamental para minha formação pessoal e acadêmica.

À minha namorada, Valentina, agradeço o carinho, compreensão, paciência e apoio constante durante toda essa trajetória, especialmente nos momentos mais difíceis.

Ao meu orientador, Professor Fábio Vincenzi, deixo meu sincero agradecimento pela orientação, dedicação, disponibilidade e por toda a ajuda prestada ao longo do desenvolvimento deste trabalho. Sua contribuição foi fundamental para a realização deste TCC.

Agradeço também a todos os professores que fizeram parte da minha formação acadêmica, pelos ensinamentos, orientações e conhecimentos compartilhados ao longo do curso, contribuindo de maneira significativa para minha trajetória.

*“Por vezes sentimos que aquilo
que fazemos não é senão uma gota no oceano. Mas
sem ela o oceano seria menor.”,
(Madre Teresa de Calcutá)*

Resumo

Este projeto tem como objetivo a implementação e a validação de uma arquitetura didática de automação industrial baseada em Raspberry Pi, com ênfase na integração multiprotocolo aplicada ao ensino de disciplinas como Redes Industriais I, Informática Industrial II e Sistemas Embarcados II. A solução é estruturada a partir de uma Raspberry Pi 3, utilizada como unidade central de processamento, responsável tanto pelo acionamento de um motor de indução por meio de uma interface de potência quanto pela aquisição de dados de sensores industriais, incluindo temperatura via 1-Wire, temperatura e umidade via I²C e sinal de rotação obtido por meio de um encoder incremental. A proposta concentra-se na demonstração prática da capacidade da Raspberry Pi de operar simultaneamente como servidor e nó de aquisição de dados em diferentes protocolos, permitindo a comunicação entre camadas de automação e aplicações de supervisão por meio de Modbus TCP, OPC UA, MQTT e HTTP. Os dados e comandos são disponibilizados para supervisão, controle e visualização em ambientes como Node-RED e ScadaBR, incluindo a construção de dashboards e interfaces gráficas para acompanhamento das variáveis de processo e do estado do atuador. Espera-se, como resultado, uma plataforma de baixo custo, segura e flexível, capaz de apoiar atividades laboratoriais ao integrar, em um único sistema, aquisição de dados, controle e comunicação multiprotocolo. Dessa forma, o trabalho contribui para a formação prática de estudantes em tecnologias de automação e conectividade associadas à Indústria 4.0, oferecendo um ambiente replicável para demonstrações, testes e validação de arquiteturas industriais modernas.

Palavras-chave: Automação industrial; Raspberry Pi; Node-RED; ScadaBR; Modbus; MQTT; OPCUA

Abstract

This project aims to implement and validate a didactic industrial automation architecture based on the Raspberry Pi, with emphasis on multiprotocol integration applied to teaching courses such as Industrial Networks I, Industrial Informatics II, and Embedded Systems II. The solution is built around a Raspberry Pi 3 used as the central processing unit, responsible for driving an induction motor through a power interface and acquiring data from industrial sensors, including temperature via 1-Wire, temperature and humidity via I²C, and rotational speed obtained from an incremental encoder. The proposal focuses on a practical demonstration of the Raspberry Pi's ability to operate simultaneously as a data acquisition node and as a server under different protocols, enabling communication between automation layers and supervisory applications through Modbus TCP, OPC UA, MQTT, and HTTP. Data and commands are made available for monitoring, control, and visualization in environments such as Node-RED and ScadaBR, including the development of dashboards and graphical interfaces to track process variables and actuator status. The expected outcome is a low-cost, secure, and flexible platform capable of supporting laboratory activities by integrating, within a single system, data acquisition, control, and multiprotocol communication. In this way, the work contributes to the practical training of students in automation and connectivity technologies associated with Industry 4.0, providing a replicable environment for demonstrations, testing, and validation of modern industrial architectures.

Keywords: Industrial automation; Raspberry Pi; Node-RED; ScadaBR; Modbus; MQTT; OPC UA.

Lista de ilustrações

Figura 1: Raspberry Pi 3 Model B	17
Figura 2: Sensor 1-Wire.....	21
Figura 3: Sensor AHT10	22
Figura 4: Encoder E6B2-CWZ5B.....	23
Figura 5: Módulo AOD4184.....	23
Figura 6: Roda e suporte	24
Figura 7: Diagrama da montagem.....	26
Figura 8: Montagem física	27
Figura 9: Diagrama do Processo	30
Figura 10: Terminal SSH.....	32
Figura 11: Acesso Raspberry utilizando o VNC.....	32
Figura 12: Criação Ambiente Venv	34
Figura 13: Entradas e Saídas.....	35
Figura 14: Definição do mapa Modbus TCP no servidor Python (Coils e Holding Registers).....	36
Figura 15: Definição de host e porta do servidor Modbus TCP no código-fonte.....	37
Figura 16: Inicialização do servidor Modbus TCP e definição dos valores iniciais no banco de dados interno.....	38
Figura 17: Importação da classe Server da biblioteca python-opcua.	39
Figura 18: Definição do endpoint e do namespace do servidor OPC UA.	40
Figura 19: Estrutura de inicialização e criação do espaço de endereçamento do servidor OPC UA.....	41
Figura 20: Rotina de encerramento seguro do servidor e limpeza de GPIO.	42
Figura 21: Trecho de código para criação da árvore de nós e variáveis do servidor OPC UA e impressão do mapa de verificação no terminal.	43
Figura 22: Trecho de código com a configuração do cliente MQTT, parâmetros do broker e tópicos utilizados para comando, estado e telemetria.	45
Figura 23: Rotinas de publicação e callbacks do cliente MQTT para assinatura do tópico de comando, aplicação do estado e republicação com retenção.	47
Figura 24: Inicialização do cliente MQTT, associação de callbacks, conexão ao broker e publicação do estado inicial do sistema.	48
Figura 25: Iniciando Servidor Modbus	49
Figura 26: Flow Modbus no Node-RED.....	51
Figura 27: Dashboard Node-RED Modbus.....	52

Figura 28: Data Sources ScadaBR.....	53
Figura 29: Configuração do Data Source Modbus TCP no ScadaBR e mapeamento dos data points para bobinas e registradores holding	54
Figura 30: Watch list do ScadaBR com pontos Modbus TCP do motor e sensores, apresentando valores e instante de atualização	55
Figura 31: Representação gráfica no ScadaBR para o protocolo Modbus TCP, com comando do motor, indicação de estado e gráficos de tendência para temperatura, umidade e RPM.	55
Figura 32: Terminal SSH inicializando o servidor OPC UA.....	57
Figura 33: Flow OPC UA no Node-RED	59
Figura 34: OPC UA Browser.....	60
Figura 35: Debug OPC UA Browser com Node ID inicial de referência ao servidor Parte 1.....	60
Figura 36: Debug OPC UA Browser com Node ID inicial de referência ao servidor Parte 2.....	61
Figura 37: Debug OPC UA Browser com Node ID (ns=2,i=1) do objeto “Motor”	62
Figura 38: Debug OPC UA Browser com Node ID (ns=2,i=2) do objeto “Sensors” Parte1	63
Figura 39: Debug OPC UA Browser com Node ID (ns=2,i=2) do objeto “Sensors” Parte2	63
Figura 40: Debug OPC UA Browser com Node ID (ns=2,i=3) do objeto “Inputs”	64
Figura 41: Dashboard Node-RED OPC UA	65
Figura 42: Terminal SSH inicializando MQTT	66
Figura 43: Flow MQTT no Node-RED	67
Figura 44: Dashboard Node-RED MQTT	68
Figura 45: Flow Node-RED MQTT-HTTP	70
Figura 46: Dashboard HTML	70
Figura 47: Gateway MQTT-HTTP	72
Figura 48: Fonte de dados RPI_HTTP configurada no ScadaBR utilizando o recurso Recuperador HTTP.	72
Figura 49: Configuração do Recuperador HTTP (RPI_HTTP) no ScadaBR com consulta periódica ao endpoint do Node-RED	73
Figura 50: Watch list ScadaBR.....	74
Figura 51: Dashboard utilizando os dados do recuperador HTML	75
Figura 52: Dashboard criada no Node-RED visualizada no ScadaBR	76
Figura 53: Saída do encoder vista no osciloscópio	77
Figura 54: Saída do terminal do servidor Modbus com leitura de pulsos e cálculo de RPM	78
Figura 55: Registro gráfico do RPM no ScadaBR durante o intervalo de observação	79

Sumário

1.	INTRODUÇÃO	13
1.1	JUSTIFICATIVA.....	14
1.2	OBJETIVOS	14
1.2.1	<i>Objetivos Específicos</i>	<i>15</i>
2.	REFERENCIAL TEÓRICO	17
2.1	RASPBERRY PI 3	17
2.2	NODE-RED	18
2.3	SCADABR	18
2.4	MODBUS.....	19
2.5	MQTT.....	19
2.6	HTTP.....	19
2.7	OPC UA.....	20
2.8	SENSOR DE TEMPERATURA 1-WIRE DS18B20.....	20
2.9	SENSOR AHT10	21
2.10	ENCODER E6B2-CWZ5B.....	22
3.	METODOLOGIA	26
3.1	VISÃO GERAL.....	26
3.1.1	<i>Montagem do sistema</i>	<i>26</i>
3.1.2	<i>Diagrama do processo</i>	<i>29</i>
3.1.3	<i>Acesso à Raspberry utilizando SSH e VNC.....</i>	<i>31</i>
3.1.4	<i>Habilitação do SSH e VNC na Raspberry Pi.....</i>	<i>31</i>
3.1.5	<i>Acesso remoto via SSH.....</i>	<i>32</i>
3.1.6	<i>Acesso remoto via VNC.....</i>	<i>32</i>
3.1.7	<i>Ambiente virtual Python (venv)</i>	<i>33</i>
3.2	ENTRADAS E SAÍDAS DA RASPBERRY PI.....	34
3.3	IMPLEMENTAÇÃO DOS SERVIDORES	36
3.3.1	<i>Servidor Modbus</i>	<i>36</i>
3.3.2	<i>Servidor OPC UA.....</i>	<i>39</i>
3.3.3	<i>Servidor MQTT.....</i>	<i>44</i>
3.4	APLICAÇÃO PROTOCOLO MODBUS	49
3.4.1	<i>Iniciando Servidor Modbus.....</i>	<i>49</i>
3.4.2	<i>Integração Modbus TCP no Node-RED e construção da dashboard.....</i>	<i>50</i>
3.4.3	<i>Dashboard Node-RED Modbus.....</i>	<i>51</i>
3.4.4	<i>Modbus ScadaBR.....</i>	<i>52</i>
3.5	APLICAÇÃO PROTOCOLO OPC UA.....	56
3.5.1	<i>Inicialização do servidor OPC UA na Raspberry Pi</i>	<i>56</i>
3.5.2	<i>OPC UA no Node-RED: leitura, escrita e supervisão.....</i>	<i>58</i>
3.5.3	<i>Descoberta de NodeIDs com OPC UA Browser</i>	<i>59</i>
3.5.4	<i>Dashboard OPC UA</i>	<i>64</i>
3.6	APLICAÇÃO PROTOCOLO MQTT	65
3.6.1	<i>Inicialização Servidor MQTT.....</i>	<i>65</i>
3.6.2	<i>Flow MQTT no Node-RED</i>	<i>66</i>
3.6.3	<i>Dashboard MQTT.....</i>	<i>68</i>
3.6.4	<i>MQTT–HTTP para supervisão no ScadaBR</i>	<i>69</i>
3.6.5	<i>Integração MQTT–HTTP para supervisão no ScadaBR</i>	<i>71</i>
3.6.6	<i>Gateway MQTT–HTTP.....</i>	<i>71</i>
3.6.7	<i>Protocolo HTTP no ScadaBR.....</i>	<i>72</i>

4.	RESULTADOS E DISCUSSÕES	77
4.1	VALIDAÇÃO DA IMPLEMENTAÇÃO DO ENCODER INCREMENTAL	77
4.2	VALIDAÇÃO DA COMUNICAÇÃO MULTIPROTOCOLO.....	80
5.	CONCLUSÃO E TRABALHOS FUTUROS.....	81
	REFERÊNCIAS.....	83
	APÊNDICE A – CÓDIGO SERVIDOR MODBUS.....	86
	APÊNDICE B – CÓDIGO SERVIDOR OPC UA	91
	APÊNDICE C – CÓDIGO SERVIDOR MQTT	97

1. Introdução

O avanço tecnológico das últimas décadas impulsionou o surgimento da chamada Indústria 4.0, marcada pela integração entre sistemas físicos e digitais, pela comunicação em tempo real entre dispositivos e pela utilização intensiva de tecnologias como Internet das Coisas (IoT), computação em nuvem, inteligência artificial e protocolos de comunicação industrial (ACATECH, 2013). Nesse contexto, a automação industrial tem se tornado cada vez mais conectada e inteligente, exigindo do engenheiro de controle e automação um domínio prático não apenas dos sistemas de controle, mas também da integração de dados e da interoperabilidade entre dispositivos.

Ao longo da história da automação, o avanço da complexidade dos processos e das distâncias entre operação e controle levou da lógica local e dos painéis de instrumentos ao conceito de supervisão: em vez de comandar cada ação manualmente, o operador passa a supervisionar tarefas, delegando execuções a sistemas no chão de fábrica e intervindo por exceção. Essa ideia foi formulada ainda nos anos 1960 como “human supervisory control” que é uma resposta prática a limitações de latência e à necessidade de controle por objetivos, com o computador local executando trechos da tarefa e reportando ao humano (FERRELL, 1967). Na indústria, o mesmo princípio se materializou nos sistemas SCADA, que consolidam medições distribuídas, exibem o estado do processo em tempo quase real, registram históricos, gerenciam alarmes e permitem comandos remotos, funções que se tornaram estruturantes à medida que redes, RTUs/PLCs e centros de operação se difundiram (NIST, 2023)

O ensino tradicional de automação, entretanto, ainda é fortemente baseado em simulações e laboratórios de alto custo, o que limita o acesso dos estudantes a experiências práticas com protocolos industriais e sistemas SCADA (Supervisory Control and Data Acquisition). Dessa forma, a criação de kits didáticos de baixo custo, que permitam a experimentação real de conceitos de instrumentação, controle e comunicação de dados, surge como uma alternativa eficaz para aproximar o ambiente acadêmico da realidade industrial.

Nesse cenário, a Raspberry Pi se destaca como uma plataforma acessível e versátil para o desenvolvimento de sistemas embarcados e de automação. Por meio de seu sistema operacional baseado em Linux e de sua capacidade de integração com sensores, atuadores e redes industriais, a Raspberry Pi pode ser utilizada como unidade central de controle em projetos educacionais e

industriais, possibilitando o uso de ferramentas de código aberto como Node-RED e ScadaBR (OPENJS FOUNDATION; IBM, 2025; SCADABR PROJECT, 2025).

1.1 Justificativa

A realização deste trabalho justifica-se pela necessidade de ampliar a prática experimental no ensino de automação industrial, especialmente em temas que envolvem integração entre sensores, atuadores, protocolos de comunicação e supervisão. Embora os conteúdos teóricos sejam fundamentais, a compreensão efetiva de arquiteturas de automação torna-se mais consistente quando o estudante pode observar, configurar e testar o funcionamento real dos elementos que compõem o sistema. Estudos recentes sobre educação em engenharia e Indústria 4.0 indicam que soluções didáticas de baixo custo e abordagem prática favorecem o aprendizado aplicado e aproximam a formação acadêmica das demandas tecnológicas atuais (PAJPACH, 2022).

Além disso, laboratórios convencionais de automação frequentemente dependem de equipamentos proprietários e de maior custo, o que pode limitar a quantidade de bancadas disponíveis e a flexibilidade para experimentação. Nesse contexto, a adoção de uma plataforma baseada em componentes acessíveis e ferramentas abertas apresenta-se como alternativa relevante para uso acadêmico, pois permite reproduzir funções típicas de sistemas industriais com menor custo de implementação e maior possibilidade de adaptação. Trabalhos recentes mostram que soluções laboratoriais modulares e economicamente viáveis têm sido empregadas justamente para viabilizar atividades práticas em automação e engenharia aplicada (AL-YAMAN, 2025).

Dessa forma, este projeto justifica-se por propor uma bancada didática capaz de reunir, em um único ambiente experimental, aquisição de dados, comunicação multiprotocolo e supervisão, contribuindo para o desenvolvimento de competências práticas importantes na formação em Engenharia de Controle e Automação.

1.2 Objetivos

O presente trabalho tem como objetivo implementar e validar uma arquitetura didática de automação industrial baseada em Raspberry Pi 3, empregada como unidade central de controle, aquisição de dados e comunicação, com foco na integração multiprotocolo. A proposta visa apoiar o ensino prático nas disciplinas de Redes Industriais I, Informática Industrial II e Sistemas

Embarcados II, demonstrando, em um mesmo ambiente experimental, a interoperabilidade entre diferentes camadas de automação e supervisão.

O sistema integra sensores, atuadores e interfaces supervisórias desenvolvidas em Node-RED e ScadaBR, permitindo a demonstração e comparação de protocolos industriais e de Internet das Coisas, como Modbus TCP, OPC UA, MQTT e HTTP. A arquitetura contempla o acionamento de uma carga representativa do motor por meio de interface de potência, bem como a aquisição de variáveis de processo, incluindo temperatura via 1-Wire, temperatura e umidade via I²C, e velocidade de rotação obtida por meio encoder acoplado ao eixo do motor.

Dessa forma, o projeto busca consolidar um ambiente modular, acessível e de baixo custo, capaz de reproduzir cenários reais de automação industrial e de integração de sistemas, evidenciando a viabilidade da Raspberry Pi como plataforma multiprotocolo para monitoramento, supervisão e experimentos em laboratório.

1.2.1 Objetivos Específicos

- Implementar uma arquitetura didática de automação industrial baseada em Raspberry Pi 3 como unidade central de aquisição de dados, controle e comunicação multiprotocolo;
- Integrar e validar a aquisição de variáveis de processo por meio de sensores digitais, incluindo temperatura via 1-Wire (DS18B20), temperatura e umidade via I²C (AHT10) e velocidade de rotação por um encoder;
- Desenvolver a lógica de comando e monitoramento do atuador e das variáveis medidas, implementando a publicação e a disponibilização dos dados em múltiplos protocolos;
- Implementar e testar um servidor Modbus TCP na Raspberry Pi, definindo um mapa de variáveis em coils e holding registers, e realizar a integração com o Node-RED e ScadaBR para supervisão e controle;
- Implementar e testar um servidor OPC UA na Raspberry Pi, estruturando o *address space* em objetos e variáveis, e integrar a leitura e escrita com o Node-RED para supervisão;

- Implementar e testar a comunicação via MQTT, incluindo a operação do broker na Raspberry Pi, a organização de tópicos para variáveis e comandos e a integração com o Node-RED;
- Implementar uma interface de monitoramento via HTTP utilizando o Node-RED, gerando página HTML para visualização das variáveis publicadas;
- Implementar e validar a integração via HTTP entre o Node-RED e o ScadaBR, disponibilizando as variáveis em endpoints HTTP e realizando a leitura no ScadaBR por meio de recuperador HTTP;
- Realizar testes funcionais para verificar consistência dos valores, acionamento do atuador, confiabilidade e estabilidade de operação nos diferentes protocolos;

2. Referencial Teórico

2.1 Raspberry Pi 3

A Raspberry Pi 3 Model B (Figura 1) é um computador de placa única de baixo custo que reúne, em um mesmo hardware compacto, recursos de processamento, armazenamento, rede e entradas/saídas de propósito geral. Diferentemente de um microcontrolador clássico, executa um sistema operacional Linux (Raspberry Pi OS), permitindo instalar serviços e bibliotecas de nível de aplicação para aquisição de dados, comunicação e supervisão. Essa característica a torna especialmente adequada ao ensino de Engenharia de Controle e Automação, pois possibilita que o estudante percorra, em um único equipamento, o ciclo completo que vai do sensor à visualização e à tomada de decisão (RASPBERRY PI FOUNDATION, 2023).

No aspecto técnico, a placa utiliza o SoC Broadcom BCM2837 com CPU ARM Cortex-A53 de 64 bits, quatro núcleos a 1,2 GHz, e 1 GB de memória LPDDR2. A conectividade nativa inclui Ethernet 10/100, Wi-Fi 2,4 GHz (802.11n) e Bluetooth 4.1/BLE, além de HDMI e quatro portas USB 2.0 para periféricos. O armazenamento é feito em cartão microSD, que abriga o sistema operacional e os dados do usuário. Para integração ao nível de campo, a placa oferece 40 pinos GPIO compatíveis com I²C, SPI, UART, PWM e 1-Wire, bem como interfaces CSI para câmera e DSI para display, possibilitando ligação direta a sensores, atuadores e módulos de expansão típicos de laboratórios didáticos (RASPBERRY PI LTD, 2016).

Em ambientes de ensino e P&D, a combinação entre Linux, GPIOs acessíveis e rede integrada permite instalar e executar com facilidade ferramentas como Node-RED, brokers MQTT, servidores HTTP/REST e bibliotecas de OPC UA, além de bases de dados e pacotes científicos. A documentação oficial do Node-RED inclui guia específico para execução na Raspberry Pi, o que reforça sua vocação como plataforma didática para orquestração e supervisão em pequena escala (OPENJS FOUNDATION; IBM, 2025). Essa dualidade entre “computador completo” e “plataforma de I/O” simplifica a criação de experimentos reproduzíveis que envolvem instrumentação, intertravamentos, registro histórico e integração multiprotocolo, sem a necessidade de um computador externo.

Figura 1: Raspberry Pi 3 Model B



Fonte: (RASPBERRY PI LTD, 2016).

2.2 Node-RED

O Node-RED é um ambiente de programação por fluxos, com editor web e execução sobre Node.js, que permite compor aplicações orientadas a eventos por meio da conexão de “nós” funcionais. Concebido originalmente na IBM e atualmente mantido pela OpenJS Foundation, combina um runtime leve, modelo assíncrono e armazenamento dos fluxos em JSON, o que o torna adequado tanto ao edge quanto à nuvem, para aquisição, transformação e visualização quase em tempo real (OPENJS FOUNDATION; IBM, 2025). Para fins didáticos, a documentação oficial inclui guias específicos de instalação e uso em Raspberry Pi, reduzindo a barreira para montar fluxos de aquisição, intertravamentos básicos e exposição de dados em protocolos comuns da automação e do IoT, com apoio de uma comunidade ativa e material atualizado.

2.3 ScadaBR

O ScadaBR é um SCADA livre e de código aberto amplamente utilizado no Brasil para supervisão, aquisição de dados e automação, com interface web, cadastro de pontos, histórico e alarmística, o que o torna apropriado para ensino e P&D (SCADABR PROJECT, 2025). A documentação pública detalha recursos e integrações, como Modbus IP/Serial operando como

fonte de dados por *polling*, além de procedimentos de instalação, criação de telas e gráficos, o que facilita conectar controladores reais ou simulados em atividades de laboratório (SCADABR PROJECT, 2016).

2.4 Modbus

Modbus é um protocolo de aplicação amplamente adotado na automação industrial, estruturado no modelo cliente–servidor e organizado em áreas de dados como coils, discrete inputs, holding registers e input registers; sua simplicidade conceitual, catálogo estável de *function codes* e mapeamento direto para palavras de 16 bits explicam a longevidade e a ampla interoperabilidade entre CLPs, IHMs e sistemas supervisórios (MODBUS ORGANIZATION, 2012). Na variante Modbus/TCP, o transporte sobre TCP/IP preserva a semântica da pilha clássica, favorecendo a integração com redes Ethernet industriais e plataformas SCADA, além de manter uma referência consolidada para implementadores e integradores por meio dos guias oficiais (MODBUS ORGANIZATION, 2024).

2.5 MQTT

MQTT é um protocolo de mensagens leve, orientado a publish/subscribe, padronizado pela OASIS, projetado para redes com largura de banda limitada, latência variável e dispositivos de recursos restritos em IoT/IIoT (OASIS, 2014; OASIS, 2019). As versões 3.1.1 e 5.0 constituem a base normativa do ecossistema, enquanto o portal do projeto agrega materiais introdutórios e referências de uso em arquiteturas de telemetria industrial (MQTT PROJECT, 2025).

2.6 HTTP

O HTTP é um protocolo de aplicação sem estado que define semântica de métodos, cabeçalhos e códigos de status para transferência de representações entre clientes e servidores. Embora concebido para a Web, sua ubiquidade e suporte nativo em sistemas operacionais e ferramentas o tornaram alicerce comum para APIs industriais e serviços de borda, permitindo consulta de estados e emissão de comandos com *payloads* JSON. A especificação moderna publicada pela IETF consolida a semântica no RFC 9110, estabelecendo a base normativa para integrações RESTful em contextos educacionais e industriais (IETF, 2022; RFC EDITOR, 2022).

2.7 OPC UA

O protocolo OPC UA combina modelo de informação rico (AddressSpace tipado), serviços cliente–servidor e mecanismos de segurança integrados, além do paradigma Publish–Subscribe para distribuição eficiente de dados. Ao substituir o endereçamento “por registradores” por nós com tipos, atributos e relações, aproxima os dados de processo de representações semânticas adequadas à integração de nível superior e à análise. A Parte 1 da especificação apresenta o escopo, os objetivos de design, o modelo de segurança e o papel de Subscriptions para entrega de atualizações sem *polling* intenso, sendo a porta de entrada recomendada para projetos de integração (OPC FOUNDATION, 2021; OPC FOUNDATION, 2024).

2.8 Sensor de temperatura 1-Wire DS18B20

O barramento 1-Wire é uma solução de comunicação serial projetada para interligar dispositivos digitais utilizando, tipicamente, um único condutor de dados mais o referencial de terra, permitindo a conexão de múltiplos sensores no mesmo fio com baixo custo e simplicidade de cabeamento. Nesse arranjo, o mestre de comunicação controla a temporização do sinal e os dispositivos escravos respondem conforme o protocolo, sendo que cada dispositivo possui um identificador único que permite seu endereçamento individual dentro da rede. Originalmente concebido para comunicação em curtas distâncias e integração de memória e sensores em uma única porta de microcontrolador, o 1-Wire também pode ser estendido a topologias maiores quando são observados cuidados de projeto relacionados a topologia, capacitância e integridade de sinal (ANALOG DEVICES, 2001).

No contexto do kit didático, foi empregado o sensor DS18B20 (Figura 2), um termômetro digital 1-Wire amplamente utilizado em aplicações embarcadas. O DS18B20 fornece a medição de temperatura em formato digital, com possibilidade de ajuste de resolução e comunicação diretamente pelo barramento 1-Wire, o que facilita a integração com a Raspberry Pi por meio do subsistema 1-Wire do sistema operacional. Entre características relevantes para aplicações didáticas e laboratoriais, destacam-se a identificação única do dispositivo no barramento, a operação com alimentação convencional ou alimentação parasita, e a disponibilização do valor de temperatura em registradores internos acessíveis por comandos 1-Wire.

Do ponto de vista de instalação, a rede 1-Wire requer atenção ao pull-up no sinal de dados, às distâncias de cabeamento e à quantidade de dispositivos conectados, pois a capacitância total e reflexões podem degradar os tempos de subida e a leitura correta de bits. Em redes pequenas, como as de bancada, a implementação costuma ser direta; contudo, em extensões maiores, recomenda-se evitar topologias em estrela sem comutação e observar critérios de “peso” e “raio” da rede, conforme discutido em guias de projeto para confiabilidade do 1-Wire.

Figura 2: Sensor 1-Wire



Fonte: (MakerHero, 2026).

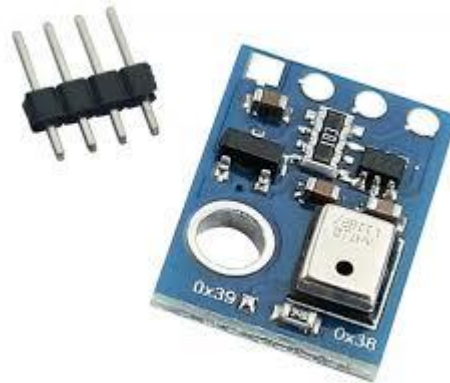
2.9 Sensor AHT10

O AHT10, mostrado na Figura 3 é um sensor digital integrado para medição de temperatura e umidade relativa do ar, cuja saída já é disponibilizada em formato digital, reduzindo a necessidade de condicionamento analógico externo. O dispositivo combina um elemento sensor capacitivo para umidade e um sensor de temperatura em um mesmo encapsulamento, incorporando também processamento interno para fornecer leituras calibradas. Por utilizar comunicação digital, o AHT10 é adequado para aplicações embarcadas e didáticas nas quais se deseja simplificar a instrumentação e reduzir suscetibilidade a interferências típicas de sinais analógicos (ASAIR, 2018.).

A comunicação do AHT10 é realizada pelo barramento I²C, permitindo a conexão do sensor à Raspberry Pi com apenas duas linhas de sinal, SDA e SCL, além da alimentação e referência de terra. Em implementações práticas, o AHT10 geralmente opera no endereço 0x38, e sua integração exige que o dispositivo seja instanciado explicitamente, visto que nem sempre é possível identificá-lo com confiabilidade apenas por varredura automática do barramento (KERNEL.ORG, 2024).

No desenvolvimento do kit, o AHT10 foi utilizado como sensor ambiental, fornecendo as variáveis de temperatura e umidade para supervisão e comparação entre diferentes protocolos de comunicação. Em termos de software, a leitura pode ser realizada por bibliotecas de alto nível que encapsulam a comunicação I²C e disponibilizam diretamente as grandezas físicas, o que facilita o uso em prototipagem e em atividades de laboratório (ADAFRUIT, 2025).

Figura 3: Sensor AHT10



Fonte: (USINAINFO, 2026).

2.10 Encoder E6B2-CWZ5B

Para a medição da velocidade de rotação do motor, foi utilizado um encoder incremental da família Omron E6B2-C, modelo E6B2-CWZ5B. Esse tipo de encoder é amplamente empregado em aplicações de automação para conversão do movimento rotacional em pulsos elétricos digitais, possibilitando a obtenção de informações de velocidade e deslocamento angular (OMRON, 2023).

No presente projeto, foi adotado um encoder com resolução de 10 pulsos por rotação (10 P/R). Dessa forma, a cada volta completa do eixo do encoder são gerados dez pulsos elétricos, os quais podem ser lidos e processados pela Raspberry Pi. A utilização de um encoder com essa

resolução mostrou-se suficiente para a finalidade didática do trabalho, embora resoluções maiores proporcionem uma leitura mais refinada da velocidade.

A Figura 4 apresenta o encoder utilizado no sistema. Conforme o datasheet da família E6B2-C, o modelo E6B2-CWZ5B opera com alimentação entre 12 V e 24 V em corrente contínua.

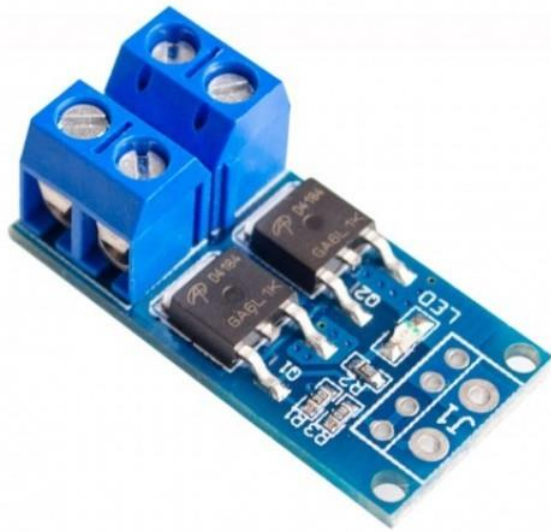
Figura 4: Encoder E6B2-CWZ5B



Fonte: (Lemaqs Automação, 2026).

Em razão dessa característica elétrica, foi utilizada uma fonte de 12 V para alimentar o encoder durante os ensaios. Entretanto, como a Raspberry Pi trabalha com nível lógico de 3,3 V, tornou-se necessário utilizar um módulo de interface para adequação do sinal. No projeto, foi empregado o módulo AOD4184, responsável por reduzir o nível de tensão do sinal proveniente do encoder, originalmente em 12 V, para um valor compatível com a entrada GPIO da Raspberry Pi. A Figura 5 apresenta o módulo utilizado nessa etapa de condicionamento do sinal.

Figura 5: Módulo AOD4184



Fonte: (SAWERS, 2026).

A ligação elétrica foi realizada da seguinte forma: o fio marrom do encoder foi conectado ao terminal positivo da fonte de 12 V, enquanto o fio azul foi conectado ao GND da alimentação. O sinal de pulsos foi obtido a partir do canal A do encoder e encaminhado ao módulo Módulo AOD4184, sendo posteriormente conectado à entrada digital da Raspberry Pi. Os demais canais do encoder não foram utilizados nesta etapa do trabalho, uma vez que o objetivo principal consistiu na medição da velocidade de rotação do motor.

Do ponto de vista mecânico, o encoder não foi acoplado diretamente ao eixo do motor. A Figura 6 apresenta o conjunto mecânico utilizado no acoplamento indireto entre o motor e o encoder.

Figura 6: Roda e suporte



Fonte: (PRIMTEC, 2026).

3. Metodologia

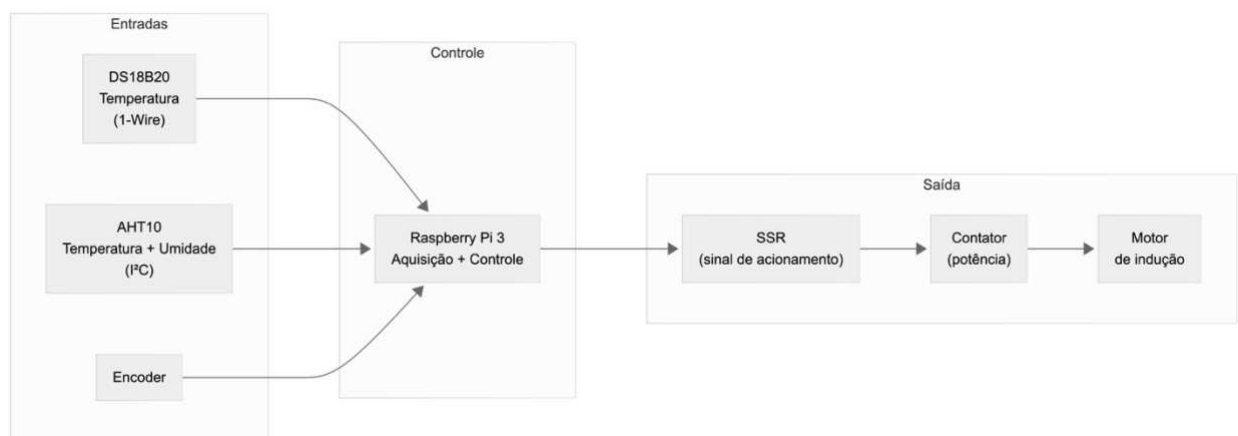
3.1 Visão Geral

Este tópico apresenta a visão geral da metodologia, contextualizando a montagem do sistema, o diagrama de processo e os procedimentos de configuração e acesso à Raspberry Pi. A sequência de subseções descreve como a plataforma foi implementada e preparada para a execução dos servidores e para a integração com as camadas de supervisão.

3.1.1 Montagem do sistema

A montagem do kit didático foi desenvolvida de forma a evidenciar a separação entre o circuito de controle e o circuito de potência, visando maior segurança operacional, organização física e facilidade de manutenção. A arquitetura geral do sistema é apresentada na Figura 7, na qual se observa a disposição funcional das entradas, da unidade de controle e das saídas. Nessa estrutura, a Raspberry Pi 3 atua como unidade central de aquisição e processamento, recebendo sinais dos sensores e enviando comandos ao estágio de acionamento do motor.

Figura 7: Diagrama da montagem



Fonte: Elaborado pelo autor (2026).

No circuito de potência, o acionamento do motor foi realizado por meio de um contator comandado indiretamente pela Raspberry Pi através de um relé de estado sólido. Com essa configuração, a placa opera apenas com sinais de baixa potência em seus pinos GPIO, enquanto o contator executa a comutação da carga de potência do motor. Essa solução contribui para reduzir

a exposição da unidade de controle a ruídos elétricos e transientes típicos de cargas indutivas, além de tornar a estrutura mais modular para futuras adaptações.

Para a instrumentação do sistema, foram integrados sensores digitais conectados à Raspberry Pi. A medição da temperatura foi realizada com o sensor DS18B20, utilizando o barramento 1-Wire, enquanto a temperatura e a umidade do ambiente foram obtidas por meio do sensor AHT10, conectado via barramento I²C. A variável de rotação foi implementada com o uso de um encoder incremental acoplado indiretamente ao eixo do motor, permitindo a obtenção de pulsos para o cálculo da velocidade de rotação (RPM). A leitura desses pulsos foi processada pela Raspberry Pi e utilizada para calcular a rotação do motor com base na contagem de eventos em janelas de tempo, possibilitando a supervisão do comportamento dinâmico do sistema nas interfaces desenvolvidas.

A Figura 8 apresenta a montagem física final do protótipo. Nela, é possível observar a base estrutural utilizada para fixação do motor, do contator, da Raspberry Pi, da protoboard e dos módulos auxiliares de interface. Também se destaca o conjunto mecânico empregado no acoplamento indireto do encoder ao eixo do motor. A disposição dos componentes foi planejada de modo a facilitar a identificação visual dos blocos funcionais e favorecer o uso em ambiente de laboratório.

Figura 8: Montagem física



Fonte: Elaborado pelo autor (2026).

A organização física do sistema buscou, portanto, conciliar funcionalidade e caráter didático, tornando mais clara a relação entre os elementos de campo, o sistema de controle e as camadas de supervisão e comunicação implementadas no projeto.

Para a transmissão do movimento do motor ao encoder, foi utilizado um suporte contendo uma roda de 6,5 cm de diâmetro (Figura 6), mantida em contato com o eixo do motor, cujo diâmetro é de 1,6 cm.

Como o contato ocorre entre superfícies cilíndricas, a relação entre a rotação do encoder e a rotação do motor pode ser determinada pela razão entre os diâmetros:

$$\frac{n_{enc}}{n_{mot}} = \frac{d_{mot}}{d_{roda}}$$

Em que n_{enc} é a rotação do encoder, n_{mot} é a rotação do motor, d_{mot} é o diâmetro do eixo do motor e d_{roda} é o diâmetro da roda acoplada ao encoder.

Substituindo-se os valores do projeto:

$$\frac{n_{enc}}{n_{mot}} = \left(\frac{1,6}{6,5} \right) = 0,2462$$

Considerando a rotação nominal do motor de 3450 rpm, conforme o datasheet do motor adotado no projeto, obtém-se para o encoder:

$$n_{enc} = 3450 * 0,2462 = 849,2 \text{ rpm}$$

Logo, para uma rotação nominal de 3450 rpm no motor, o encoder passa a girar aproximadamente a 849,2 rpm.

Como o encoder adotado fornece 10 pulsos por rotação, o número de pulsos equivalente a uma volta do motor pode ser estimado por:

$$PPR_{mot} = 10 * \left(\frac{1,6}{6,5}\right) = 2,4615$$

Portanto, o sistema passa a ter aproximadamente 2,46 pulsos por volta do eixo do motor. A partir disso, a frequência de pulsos esperada em regime nominal é:

$$f = \frac{(3450 * 2,4615)}{60} = 141,5 \frac{pulsos}{s}$$

Dessa forma, em uma janela de aquisição de 10 segundos, espera-se a contagem aproximada de:

$$N = 141,5 * 10 = 1415 \text{ pulsos}$$

3.1.2 Diagrama do processo

O diagrama mostrando na Figura 9 apresenta a arquitetura geral e a forma como os sinais físicos, os servidores de comunicação e as camadas de supervisão se relacionam. Na parte superior esquerda, encontra-se a planta didática, composta pelos sensores DS18B20 para temperatura via barramento 1-Wire, AHT10 para temperatura e umidade via barramento I²C e o encoder incremental acoplado ao sistema para obtenção dos pulsos de rotação, além do bloco de acionamento do motor por GPIO, representado por SSR e contator. Esses elementos convergem para a Raspberry Pi, responsável pela aquisição dos dados e pela lógica de controle, concentrando as leituras de sensores e o comando de acionamento do atuador em um único ponto de processamento.

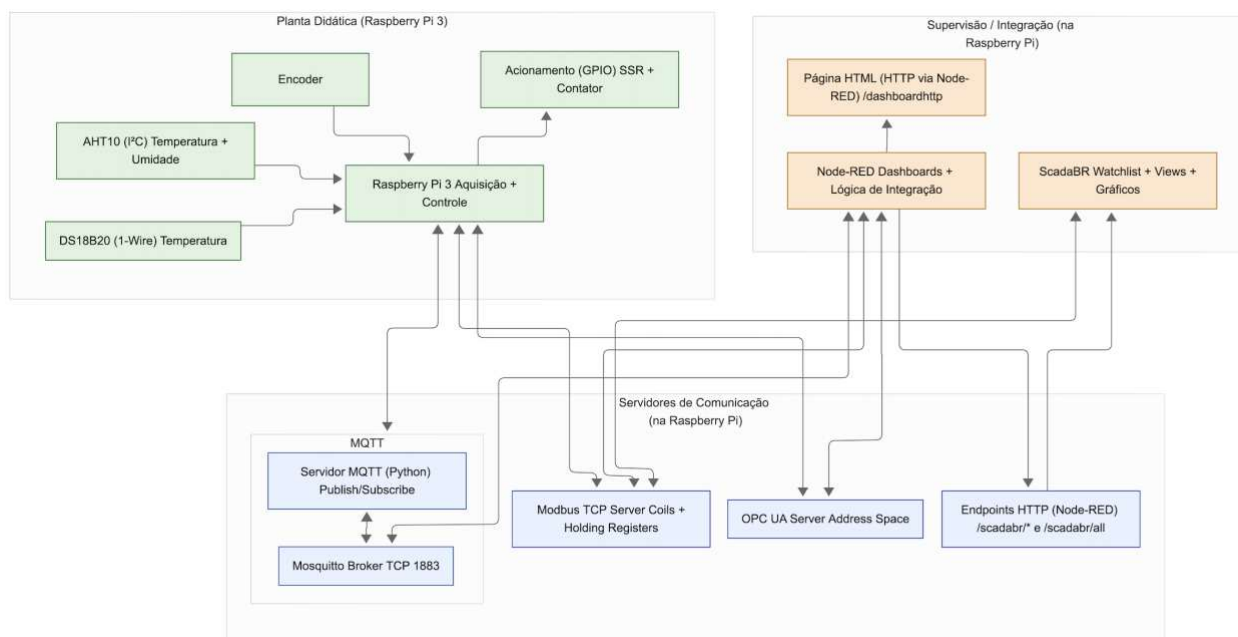
Na região inferior do diagrama, são agrupados os servidores de comunicação executados na Raspberry Pi, que expõem as variáveis do sistema por diferentes protocolos. O servidor Modbus TCP disponibiliza o comando e o estado do motor em bobinas e as variáveis de processo em registradores holding. Em paralelo, o servidor OPC UA organiza as mesmas grandezas em um *address space* estruturado por objetos e variáveis, permitindo leitura contínua e escrita do comando do motor. Para o protocolo MQTT, o diagrama separa a execução do cliente publicador e assinante

em Python, responsável por publicar e receber mensagens nos tópicos do projeto, e o broker Mosquitto, que opera como intermediário de mensagens na porta padrão TCP 1883. Essa separação evidencia que o MQTT depende de um componente central de roteamento de mensagens, ao contrário de Modbus TCP e OPC UA, que expõem diretamente um servidor de dados para os clientes.

À direita, o diagrama destaca a camada de supervisão e integração, também executada na Raspberry Pi. O Node-RED atua como ambiente de integração ao consumir dados via MQTT e OPC UA, bem como ao interagir com o Modbus TCP, alimentando dashboards de supervisão e consolidando o estado das variáveis. Adicionalmente, o Node-RED disponibiliza uma página HTML acessível por HTTP, construída a partir das últimas leituras recebidas, configurada no endpoint indicado, o que caracteriza uma etapa de supervisão leve baseada em navegador. Por fim, o ScadaBR é apresentado como supervisório com recursos de *watchlist*, visualizações e gráficos, recebendo dados tanto por Modbus TCP quanto por HTTP, neste último caso por meio de endpoints publicados pelo Node-RED, representados no diagrama como rotas destinadas à integração com o supervisório.

De forma síntese, o diagrama evidencia um fluxo em três camadas: aquisição e controle no nível físico, exposição das variáveis por múltiplos protocolos no nível de comunicação e, no nível superior, supervisão e integração por Node-RED e ScadaBR. Essa organização permite comparar, em um mesmo arranjo experimental, as diferenças de abordagem entre protocolos orientados a registradores, modelos de informação estruturados e comunicação orientada a mensagens, mantendo um conjunto comum de variáveis e comandos ao longo de toda a arquitetura.

Figura 9: Diagrama do Processo



Fonte: Elaborado pelo autor (2026).

3.1.3 Acesso à Raspberry utilizando SSH e VNC

Para permitir o acesso remoto à Raspberry Pi durante o desenvolvimento do projeto, foi necessário habilitar dois serviços nativos do sistema: SSH (acesso ao terminal via rede) e VNC (acesso gráfico ao desktop). Esses recursos facilitam a execução dos scripts, instalação de pacotes, monitoramento dos servidores e configuração do Node-RED e do ScadaBR sem a necessidade de manter teclado e monitor conectados diretamente à placa.

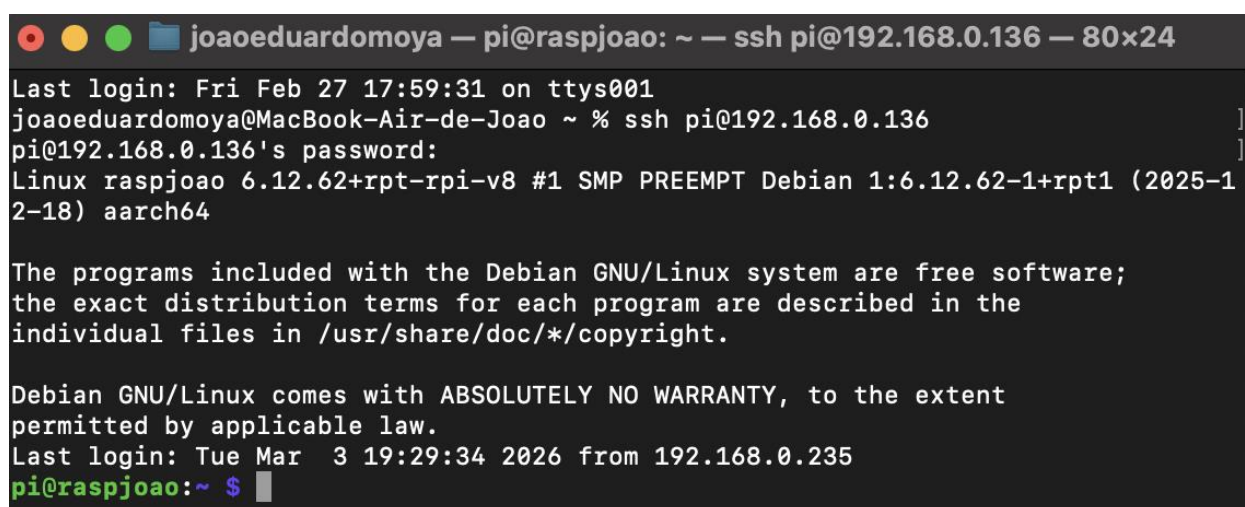
3.1.4 Habilitação do SSH e VNC na Raspberry Pi

A ativação pode ser feita de duas formas principais: pela interface gráfica do Raspberry Pi OS ou via terminal. No modo gráfico, acessa-se o menu de configurações do sistema e habilitam-se explicitamente as opções SSH e VNC na seção de interfaces. Alternativamente, pelo terminal, o procedimento é realizado utilizando a ferramenta de configuração do sistema, que permite ativar esses serviços de maneira simples e segura.

3.1.5 Acesso remoto via SSH

O SSH mostrado na Figura 10 foi utilizado para acessar a Raspberry Pi por linha de comando a partir do computador principal, permitindo executar e encerrar os servidores em Python, editar arquivos, verificar endereços de rede e consultar logs do sistema. Esse método é leve, rápido e adequado para tarefas de desenvolvimento e depuração.

Figura 10: Terminal SSH



```
joaoeduardomoya — pi@raspjoao: ~ — ssh pi@192.168.0.136 — 80x24
Last login: Fri Feb 27 17:59:31 on ttys001
joaoeduardomoya@MacBook-Air-de-Joao ~ % ssh pi@192.168.0.136
pi@192.168.0.136's password:
Linux raspjoao 6.12.62+rpt-rpi-v8 #1 SMP PREEMPT Debian 1:6.12.62-1+rpt1 (2025-12-18) aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

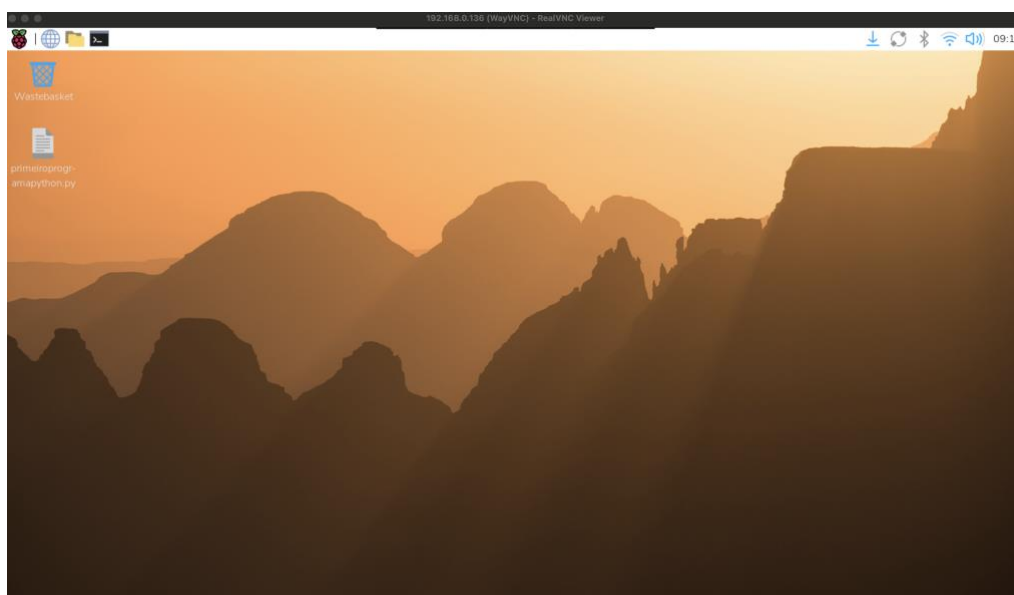
Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Tue Mar 3 19:29:34 2026 from 192.168.0.235
pi@raspjoao:~ $
```

Fonte: Elaborado pelo autor (2026).

3.1.6 Acesso remoto via VNC

O VNC (Figura 11) foi utilizado quando era necessário interagir com a interface gráfica da Raspberry Pi, por exemplo para operar o navegador local, ajustar configurações do sistema, visualizar janelas e utilizar ferramentas com interface gráfica. Assim, o acesso remoto tornou-se equivalente a estar fisicamente conectado ao desktop da placa.

Figura 11: Acesso Raspberry utilizando o VNC



Fonte: Elaborado pelo autor (2026).

3.1.7 Ambiente virtual Python (venv)

O projeto utilizou um ambiente virtual Python (venv) (Figura 12) para organizar e isolar todas as bibliotecas necessárias ao funcionamento dos servidores e da leitura de sensores na Raspberry Pi. Em sistemas Linux modernos (como o Raspberry Pi OS), é comum existir uma restrição para instalação de pacotes Python no sistema via pip, justamente para evitar que atualizações ou bibliotecas instaladas manualmente quebrem componentes do próprio sistema operacional. Para contornar isso de forma correta e segura, foi criado um venv dentro da pasta do projeto (por exemplo, `~/tcc/.venv`), que funciona como uma instalação separada de Python, contendo seu próprio interpretador e suas próprias dependências. A instalação/configuração foi feita com o Python já existente na Raspberry, criando o ambiente com `python3 -m venv ~/tcc/.venv`.

Depois disso, o ambiente foi ativado com `source ~/tcc/.venv/bin/activate`, e o terminal passou a exibir `(.venv)` no início da linha, indicando que os comandos `python` e `pip` passaram a apontar para esse ambiente isolado. A partir desse ponto, todas as bibliotecas usadas no projeto por exemplo, servidor Modbus, servidor OPC UA, cliente/biblioteca MQTT e bibliotecas de sensores como AHT10) foram instaladas com o `pip install` dentro do venv, mantendo o sistema operacional limpo e garantindo reprodutibilidade. Finalmente, os servidores foram executados sempre com o ambiente ativado, usando comandos como `python arquivo.py`, assegurando que cada protocolo Modbus, OPC UA e MQTT rodasse na Raspberry Pi com as dependências corretas e sem conflito com o Python global do sistema.

Figura 12: Criação Ambiente Venv

```
Last login: Sun Feb  8 19:41:29 on ttys002
joaoeduardomoya@MacBook-Air-de-Joao ~ % ssh pi@192.168.0.136
pi@192.168.0.136's password:
Linux raspjao 6.12.62+rpt-rpi-v8 #1 SMP PREEMPT Debian 1:6.12.62-1+rpt1 (2025-12-18) aarch64

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Mon Feb  9 18:47:29 2026 from 192.168.0.108
pi@raspjao:~ $ cd ~/tcc
pi@raspjao:~/tcc $ source .venv/bin/activate
(.venv) pi@raspjao:~/tcc $
```

Fonte: Elaborado pelo autor (2026).

3.2 Entradas e Saídas da Raspberry Pi

A Raspberry Pi 3 foi utilizada como unidade central de aquisição e controle, realizando a leitura de sensores digitais e o acionamento do circuito de potência por meio de saídas GPIO, como mostrado na Figura 13. As conexões foram definidas de modo a separar alimentação, comunicação e sinais de controle, visando organização e reprodutibilidade da montagem.

Comunicação 1-Wire

- GPIO4 (DATA 1-Wire): linha de dados do sensor DS18B20, responsável pela medição de temperatura no barramento 1-Wire.
- 3V3 e GND: alimentação e referência elétrica do sensor.

Comunicação I²C

- GPIO2 (SDA): linha de dados do barramento I²C para o sensor AHT10.
- GPIO3 (SCL): linha de clock do barramento I²C para o sensor AHT10.
- 3V3 e GND: alimentação e referência elétrica do sensor.

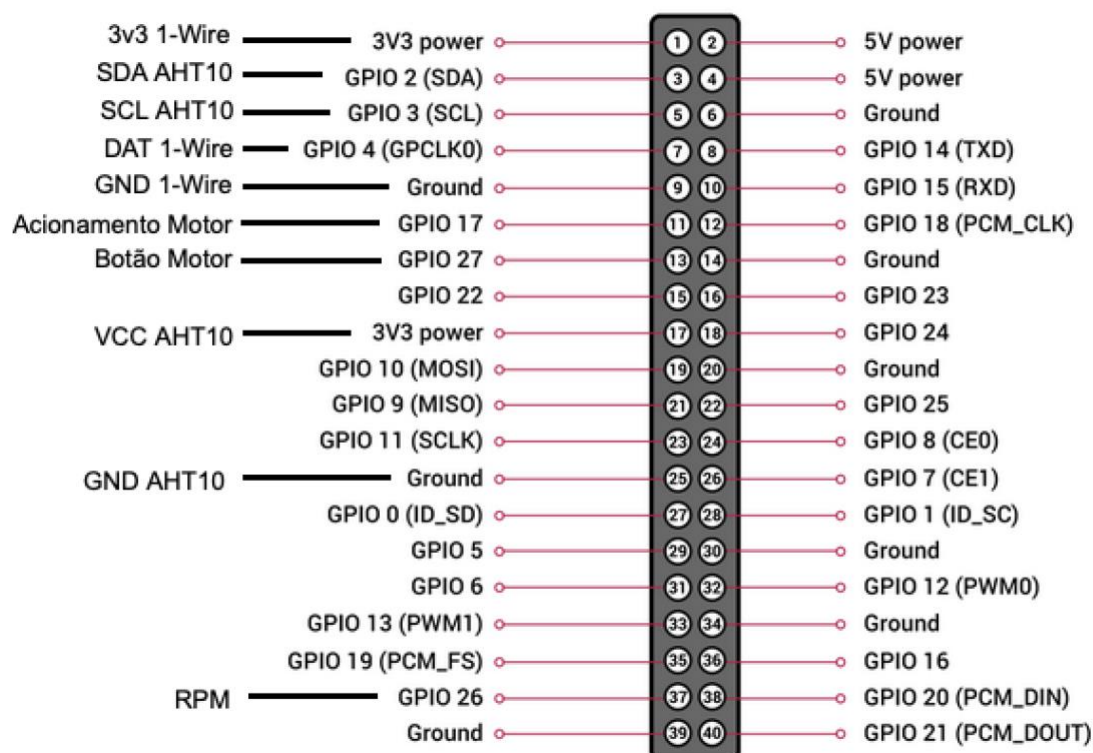
Saída de acionamento do motor

- GPIO17 (saída digital): sinal de comando do acionamento, utilizado para controlar o relé de estado sólido (SSR) e, consequentemente, o contator responsável pela comutação do motor de indução.

Entradas digitais por botão

- GPIO27 (entrada digital): botão local de comando liga/desliga do motor, permitindo alternar o estado de operação (CMD_RUN) diretamente na planta.
- GPIO26 (entrada digital): entrada de pulsos proveniente do encoder incremental, utilizada para contagem de pulsos e cálculo da rotação (RPM) do motor.

Figura 13: Entradas e Saídas



Fonte: Elaborado pelo autor (2026).

3.3 Implementação dos Servidores

3.3.1 Servidor Modbus

Para o protocolo Modbus TCP, foi implementado um servidor em Python executado na Raspberry Pi, responsável por disponibilizar variáveis de comando, estado e medições em uma estrutura compatível com clientes Modbus. Para isso, foi empregada a biblioteca pyModbusTCP, que provê a implementação do servidor Modbus TCP e disponibiliza um banco de dados interno associado às áreas de dados do protocolo, como Coils e Holding Registers. Nesse arranjo, o banco de dados do servidor funciona como um mapa de endereçamento, no qual cada variável do sistema é associada a um endereço Modbus específico, permitindo que aplicações externas realizem leitura e escrita de forma padronizada, conforme a organização definida no mapa do projeto (Figura 14).

O mapa Modbus consiste na definição dos endereços e tipos de dados utilizados para representar as variáveis do sistema. As Coils representam variáveis booleanas e foram utilizadas para comandos e estados digitais, sendo `CMD_RUN` associado à Coil 0 e `ST_MOTOR_ON` associado à Coil 1. No protótipo, o estado `ST_MOTOR_ON` foi configurado para espelhar o comando `CMD_RUN`, de modo a viabilizar a validação do acionamento e da supervisão do motor por clientes Modbus. Em complemento, os Holding Registers representam palavras de 16 bits e foram utilizados para grandezas numéricas, incluindo temperatura, umidade e velocidade, alocadas, respectivamente, em `HR0`, `HR1`, `HR2` e `HR3`.

Como as variáveis analógicas possuem casas decimais, adotou-se uma estratégia de escalonamento para armazenamento em registradores inteiros, multiplicando-se os valores por um fator de 10 antes da escrita no registrador. Dessa forma, por exemplo, uma leitura de 25,3 °C pode ser representada internamente como 253, permitindo que o cliente reconverta o valor para a unidade original por meio da divisão pelo mesmo fator. Por fim, foram definidas constantes simbólicas no código, como `CMD_COIL`, `ST_COIL`, `TEMP_HR` e demais identificadores, com o objetivo de padronizar o acesso aos endereços e reduzir erros de manutenção, evitando o uso recorrente de valores numéricos diretamente no corpo do programa.

Figura 14: Definição do mapa Modbus TCP no servidor Python (Coils e Holding Registers).

```
import time
import glob
import RPi.GPIO as GPIO
from pyModbusTCP.server import ModbusServer

CMD_COIL      = 0
ST_COIL       = 1
TEMP_HR       = 0
AHT_T_HR      = 1
AHT_RH_HR     = 2
RPM_HR        = 3
```

Fonte: Elaborado pelo autor (2026).

O parâmetro HOST = “0.0.0.0” indica que o servidor Modbus TCP ficará em escuta em todas as interfaces de rede disponíveis na Raspberry Pi, como mostrado na Figura 15. Dessa forma, o serviço pode ser acessado tanto localmente, por meio do endereço de loopback, quanto a partir de outros dispositivos na mesma rede, utilizando o endereço IP da Raspberry Pi.

Por outro lado, o parâmetro PORT = 1502 define a porta TCP utilizada pelo serviço. Embora a porta padrão do Modbus TCP seja a 502, optou-se por utilizar a 1502 porque, em sistemas Linux, portas abaixo de 1024 são consideradas portas privilegiadas e, em geral, exigem a execução do processo com permissões elevadas (por exemplo, como *root*) para serem abertas. Assim, ao adotar a porta 1502, foi possível executar o servidor como usuário comum dentro do ambiente virtual Python, reduzindo riscos e simplificando a operação e os testes.

Ressalta-se que essa alteração não modifica o protocolo Modbus TCP, apenas o ponto de acesso (porta) do serviço; em ambiente industrial, caso seja necessário operar na porta padrão 502, pode-se ajustar a configuração e executar o serviço com as permissões adequadas ou aplicar mecanismos do sistema operacional

Figura 15: Definição de host e porta do servidor Modbus TCP no código-fonte.

```
HOST = "0.0.0.0"
PORT = 1502
```

Fonte: Elaborado pelo autor (2026).

No escopo da função `main`, o comando `server = ModbusServer(host=HOST, port=PORT, no_block=True)` instancia o servidor Modbus TCP, associando-o ao endereço e à porta previamente definidos. Em seguida, `server.start()` efetiva a inicialização do serviço, abrindo o socket de comunicação na interface de rede da Raspberry Pi e habilitando o atendimento a requisições Modbus realizadas por clientes externos, como o Node-RED e o ScadaBR, conforme apresentado na Figura 16. O uso de `no_block=True` é relevante porque permite que a aplicação continue executando o laço principal de aquisição e atualização de variáveis enquanto o servidor responde às conexões e solicitações Modbus em paralelo, evitando o bloqueio da execução do programa.

Após o início do servidor, a instrução `db = server.data_bank` fornece acesso ao banco de dados interno mantido pela biblioteca, o qual representa a área de memória do dispositivo Modbus. Esse banco organiza as variáveis do sistema nas regiões típicas do protocolo, como Coils e Holding Registers, sendo o ponto de leitura e escrita efetivamente utilizado pelos clientes Modbus. Assim, as atualizações realizadas pelo código nos Coils e registradores são refletidas diretamente nas leituras efetuadas pelas aplicações supervisórias.

A etapa de inicialização do mapa é realizada por meio de `db.set_coils(0, [False, False])` e `db.set_holding_registers(0, [0, 0, 0, 0])`. No primeiro caso, são inicializados dois Coils a partir do endereço 0, correspondendo às posições do comando e do estado do motor, com valores iniciais em nível lógico falso. No segundo caso, são inicializados quatro Holding Registers a partir do endereço 0, correspondendo às variáveis numéricas do projeto, com valores iniciais nulos. Essa definição explícita garante consistência na partida do sistema, reduzindo a ocorrência de leituras indefinidas e assegurando que o supervisório encontre os endereços configurados com valores válidos desde o início, mesmo antes das rotinas de leitura de sensores atualizarem os registradores durante a execução.

Figura 16: Inicialização do servidor Modbus TCP e definição dos valores iniciais no banco de dados interno.

```
def main():  
    gpio_setup()  
  
    server = ModbusServer(host=HOST, port=PORT, no_block=True)  
    server.start()  
    db = server.data_bank  
  
    db.set_coils(0, [False, False])  
    db.set_holding_registers(0, [0, 0, 0, 0])
```

Fonte: Elaborado pelo autor (2026).

3.3.2 Servidor OPC UA

A instrução `from opcua import Server`, apresentada na Figura 17, realiza a importação da classe `Server` da biblioteca `python-opcua`, responsável por prover os recursos necessários à implementação de um servidor OPC UA em Python. Essa classe encapsula funcionalidades essenciais do protocolo, incluindo configuração de endpoint, gerenciamento de sessão, definição de nós no espaço de endereçamento, tipagem de variáveis e suporte às operações de leitura, escrita e navegação. A partir dessa importação, torna-se possível publicar variáveis de comando, estado e medições do sistema em uma estrutura hierárquica de nós OPC UA, viabilizando o acesso por clientes compatíveis, como o Node-RED e ferramentas de diagnóstico, para fins de supervisão e controle.

Figura 17: Importação da classe `Server` da biblioteca `python-opcua`.

```
from opcua import Server
```

Fonte: Elaborado pelo autor (2026).

O trecho de configuração apresentado na Figura 18 define parâmetros fundamentais para a disponibilização do servidor OPC UA na rede. O valor `ENDPOINT = "opc.tcp://0.0.0.0:4840/tcc/"` estabelece o endereço de serviço do servidor, no qual o prefixo `opc.tcp://` indica o uso do transporte TCP. O endereço `0.0.0.0` determina que o servidor aceitará conexões em todas as interfaces de rede da Raspberry Pi, permitindo acesso tanto local quanto por outros dispositivos na mesma rede. A

porta 4840 é amplamente utilizada como porta padrão em implementações OPC UA, enquanto o sufixo /tcc/ atua como identificador lógico do serviço, facilitando a organização e a distinção do endpoint no contexto do projeto.

Ainda na Figura 18, `NAMESPACE_URI = "tcc"` define o namespace associado ao projeto. No OPC UA, o namespace é utilizado para organizar e separar conjuntos de nós no espaço de endereçamento, evitando conflitos de nomes e permitindo identificar a origem das variáveis publicadas. Após o registro desse namespace no servidor, é atribuído um índice numérico `ns`, e os nós criados passam a ser referenciados com esse índice nos respectivos `NodeIDs`, por exemplo `ns=2`, conforme observado durante a etapa de navegação e identificação dos nós.

Figura 18: Definição do endpoint e do namespace do servidor OPC UA.

```
ENDPOINT = "opc.tcp://0.0.0.0:4840/tcc/"  
NAMESPACE_URI = "tcc"
```

Fonte: Elaborado pelo autor (2026).

A rotina de inicialização do servidor OPC UA apresentada na Figura 19 concentra a criação do serviço, o registro do namespace e a construção da árvore de nós que será consumida pelos clientes. Em `opcua_init()`, a instrução `server = Server()` instancia o servidor OPC UA e `server.set_endpoint(ENDPOINT)` configura o endereço no qual o serviço ficará disponível para conexões. Em seguida, `idx = server.register_namespace(NAMESPACE_URI)` registra o namespace do projeto e retorna o índice numérico associado a esse namespace, o qual passa a compor os identificadores dos nós criados, na forma `ns=<idx>`, conforme observado durante a etapa de navegação e identificação de `NodeIDs`.

Ainda na Figura 19, `objects = server.get_objects_node()` acessa o nó padrão `Objects`, utilizado como raiz para organização do espaço de endereçamento. A partir desse nó foram criados três objetos de agrupamento, `Motor`, `Sensors` e `Inputs`, por meio de `add_object`, de modo a estruturar as variáveis por função e facilitar a navegação em clientes OPC UA. No objeto `Motor` foram inseridas duas variáveis booleanas, `CMD_RUN` e `ST_MOTOR_ON`. A variável `CMD_RUN` foi configurada como escrita com `set_writable()`, permitindo que clientes alterem seu valor para

realizar o acionamento do motor. Já ST_MOTOR_ON foi mantida sem permissão de escrita, caracterizando-a como variável de leitura para supervisão do estado do motor, de modo que o cliente apenas consome o status publicado pelo servidor.

No agrupamento Sensors, foram criadas variáveis numéricas iniciadas com 0.0, como DS18B20_T, AHT10_T e AHT10_RH, o que faz com que sejam publicadas como valores do tipo ponto flutuante, adequados para apresentação em instrumentos e indicadores de supervisão. De forma análoga, o objeto Inputs contém a variável RPM, também numérica, destinada ao valor calculado a partir da entrada de pulsos. Ao final, as mensagens de impressão no terminal registram as condições de inicialização dos sensores e apresentam, de forma explícita, o endpoint, o índice do namespace e os caminhos lógicos dos nós, reduzindo erros de configuração no cliente e servindo como verificação preliminar antes da integração com o Node-RED.

Figura 19: Estrutura de inicialização e criação do espaço de endereçamento do servidor OPC UA.

```
def opcua_init():
    server = Server()
    server.set_endpoint(ENDPOINT)
    idx = server.register_namespace(NAMESPACE_URI)

    objects = server.get_objects_node()

    # Objetos
    motor_obj = objects.add_object(idx, "Motor")
    sensors_obj = objects.add_object(idx, "Sensors")
    inputs_obj = objects.add_object(idx, "Inputs")

    # Motor
    var_cmd = motor_obj.add_variable(idx, "CMD_RUN", False)
    var_cmd.set_writable()
    var_st = motor_obj.add_variable(idx, "ST_MOTOR_ON", False)

    # Sensores
    var_ds = sensors_obj.add_variable(idx, "DS18B20_T", 0.0) # °C
    var_aht_t = sensors_obj.add_variable(idx, "AHT10_T", 0.0) # °C
    var_aht_rh = sensors_obj.add_variable(idx, "AHT10_RH", 0.0) # %

    # Inputs
    var_rpm = inputs_obj.add_variable(idx, "RPM", 0.0)

    print(f"[W1] DS18B20: {w1_path if w1_path else 'NÃO ENCONTRADO'}")
    print("[I2C] AHT10 OK" if aht is not None else "[I2C] AHT10 NÃO INICIALIZOU")
    print(f"[OPCUA] Endpoint: {ENDPOINT}")
    print(f"[OPCUA] Namespace index ({NAMESPACE_URI}): ns={idx}")
    print("[OPCUA] Nodes:")
    print("  Motor/CMD_RUN (writable)")
    print("  Motor/ST_MOTOR_ON (read-only)")
    print("  Sensors/DS18B20_T (Double °C)")
    print("  Sensors/AHT10_T (Double °C)")
    print("  Sensors/AHT10_RH (Double %)")
    print("  Inputs/RPM (Double)")

    return server, var_cmd, var_st, var_ds, var_aht_t, var_aht_rh, var_rpm
```

Fonte: Elaborado pelo autor (2026).

Na Figura 20, a função `on_exit(signum=None, frame=None)` foi implementada como rotina de finalização segura do sistema. Sua finalidade é garantir que, em situações de interrupção do processo por comando do usuário ou por sinal do sistema operacional, o atuador associado ao pino de saída seja colocado em condição segura antes do encerramento do programa. Para isso, o bloco `try` força o nível lógico baixo no pino de acionamento do motor por meio de `GPIO.output(MOTOR_PIN, GPIO.LOW)`, assegurando o desligamento do comando físico, e em seguida executa `GPIO.cleanup()`, que libera a configuração dos pinos e evita que permaneçam travados como saídas em nível alto após a finalização.

Ainda na Figura 20, o tratamento de exceções foi incluído para impedir que falhas durante o processo de limpeza interrompam o encerramento do programa. Ao final, a mensagem impressa no terminal confirma a execução da rotina de desligamento e `sys.exit(0)` finaliza o processo de forma controlada. Em aplicações de automação industrial e prototipagem embarcada, esse tipo de procedimento é relevante por reduzir riscos operacionais, uma vez que evita a permanência de sinais indevidos em saídas de controle quando ocorre interrupção inesperada do software.

Figura 20: Rotina de encerramento seguro do servidor e limpeza de GPIO.

```
def on_exit(signum=None, frame=None):  
    try:  
        GPIO.output(MOTOR_PIN, GPIO.LOW)  
        GPIO.cleanup()  
    except Exception:  
        pass  
    print("\nSaindo... motor OFF, GPIO limpo.")  
    sys.exit(0)
```

Fonte: Elaborado pelo autor (2026).

Na Figura 21, apresenta-se o trecho responsável por instanciar o servidor OPC UA, construir a árvore de objetos e variáveis do projeto e imprimir no terminal um mapa de verificação para validação da inicialização. Inicialmente, o servidor é criado por meio de `Server()`, o endereço de escuta é configurado com `set_endpoint(ENDPOINT)` e, em seguida, o namespace do projeto é registrado com `register_namespace(NAMESPACE_URI)`, retornando o índice `idx`. Esse índice passa a identificar os nós definidos no escopo do projeto, resultando em identificadores no formato `ns=idx`, utilizados pelos clientes durante a navegação e o acesso às variáveis.

O nó padrão `Objects` é obtido por `get_objects_node()`, sendo utilizados objetos lógicos para estruturar as informações em três agrupamentos: `Motor`, `Sensors` e `Inputs`. Essa organização facilita a navegação por clientes OPC UA, pois separa comandos e estados do motor, variáveis de sensores e entradas associadas à velocidade. No agrupamento `Motor`, a variável `CMD_RUN` é criada com valor inicial booleano e recebe `set_writable()`, habilitando escrita por clientes e permitindo o envio de comandos de acionamento. Em contrapartida, `ST_MOTOR_ON` é publicada como variável de leitura, destinada à disponibilização do estado do motor. No agrupamento `Sensors`, são publicadas as variáveis `DS18B20_T`, `AHT10_T` e `AHT10_RH` com valores numéricos de ponto flutuante, adequados à representação de medições. No agrupamento `Inputs`, a variável `RPM` é criada para disponibilizar o valor de rotação calculado a partir da entrada de pulsos.

Por fim, as chamadas `print()` cumprem papel de validação operacional, ao confirmar a detecção do sensor `DS18B20`, a inicialização do `AHT10`, o endpoint ativo, o índice de namespace utilizado e a lista de nós publicados. Esse procedimento reduz inconsistências de configuração no cliente, uma vez que fornece uma referência direta dos caminhos lógicos das variáveis disponibilizadas pelo servidor.

Figura 21: Trecho de código para criação da árvore de nós e variáveis do servidor OPC UA e impressão do mapa de verificação no terminal.

```

# Inicializa server e nodes
server = Server()
server.set_endpoint(ENDPOINT)
idx = server.register_namespace(NAMESPACE_URI)

objects = server.get_objects_node()
motor_obj = objects.add_object(idx, "Motor")
sensors_obj = objects.add_object(idx, "Sensors")
inputs_obj = objects.add_object(idx, "Inputs")

var_cmd = motor_obj.add_variable(idx, "CMD_RUN", False)
var_cmd.set_writable()
var_st = motor_obj.add_variable(idx, "ST_MOTOR_ON", False)

var_ds = sensors_obj.add_variable(idx, "DS18B20_T", 0.0)
var_aht_t = sensors_obj.add_variable(idx, "AHT10_T", 0.0)
var_aht_rh = sensors_obj.add_variable(idx, "AHT10_RH", 0.0)

var_rpm = inputs_obj.add_variable(idx, "RPM", 0.0)

# Mostra mapa
print(f"[W1] DS18B20: {w1_path if w1_path else 'NÃO ENCONTRADO'}")
print("[I2C] AHT10 OK" if aht is not None else "[I2C] AHT10 NÃO INICIALIZOU")
print(f"[OPCUA] Endpoint: {ENDPOINT}")
print(f"[OPCUA] Namespace index ({NAMESPACE_URI}): ns={idx}")
print("[OPCUA] Nodes:")
print("  Motor/CMD_RUN (writable)")
print("  Motor/ST_MOTOR_ON (read-only)")
print("  Sensors/DS18B20_T (Double °C)")
print("  Sensors/AHT10_T (Double °C)")
print("  Sensors/AHT10_RH (Double %)")
print("  Inputs/RPM (Double)")
print("Endpoints other than open requested but private key and certificate are not set.")

main()

```

Fonte: Elaborado pelo autor (2026).

3.3.3 Servidor MQTT

O protocolo MQTT baseia-se em uma arquitetura orientada a mensagens que utiliza um componente central denominado *broker*, responsável por intermediar a troca de informações entre publicadores e assinantes. No desenvolvimento do projeto, o broker adotado foi o Eclipse Mosquitto, instalado e executado na Raspberry Pi. Nessa abordagem, as variáveis do processo são publicadas em tópicos específicos e distribuídas aos clientes que estejam inscritos, caracterizando o modelo publicador/assinante. Assim, os dispositivos não estabelecem comunicação direta entre si, pois o broker passa a concentrar o roteamento das mensagens, permitindo que múltiplos consumidores, como o Node-RED, ferramentas de teste e aplicações supervisórias, recebam os mesmos dados simultaneamente.

A operação do broker pode ser verificada por meio do gerenciamento de serviços do sistema e pela confirmação da porta de escuta associada ao protocolo. A porta 1883 corresponde ao padrão do MQTT sobre TCP sem criptografia e foi utilizada nas configurações do projeto. Quando configurado para escutar apenas em 127.0.0.1, o broker restringe o acesso a conexões locais, isto é, à própria Raspberry Pi. Ao permitir escuta em 0.0.0.0, o serviço passa a aceitar conexões provenientes de outras máquinas na rede local, ampliando as possibilidades de integração e testes.

A biblioteca `paho.mqtt.client`, apresentada na Figura 22, fornece os recursos necessários para que aplicações em Python atuem como clientes MQTT. Essa implementação permite estabelecer conexão com o broker, publicar mensagens em tópicos e assinar tópicos para recepção de mensagens, além de suportar um modelo orientado a eventos por meio de rotinas de *callback*, como funções de tratamento de conexão e de recebimento de mensagens. No projeto, essa biblioteca viabilizou tanto a publicação contínua das leituras dos sensores e do estado do motor quanto o recebimento de comandos de acionamento provenientes de clientes externos, como o Node-RED.

Conforme indicado na Figura 22, a configuração do cliente MQTT foi definida apontando para o broker em execução na própria Raspberry Pi, utilizando o endereço 127.0.0.1 e a porta 1883. Em seguida, os tópicos foram organizados de forma hierárquica para separar comandos, estados e medições. Para o motor, foi definido um tópico destinado ao recebimento de comandos de acionamento, `tcc/motor/cmd_run/set`, enquanto os tópicos `tcc/motor/cmd_run` e `tcc/motor/st_motor_on` foram utilizados para publicação do comando atual e do estado do motor, respectivamente. As variáveis de instrumentação foram publicadas em tópicos próprios de telemetria, incluindo `tcc/sensors/ds18b20_t`, `tcc/sensors/aht10_t`, `tcc/sensors/aht10_rh` e `tcc/inputs/rpm`. Essa organização favorece a escalabilidade e a clareza do sistema, uma vez que facilita a identificação das variáveis e a ampliação futura do conjunto de medições.

Figura 22: Trecho de código com a configuração do cliente MQTT, parâmetros do broker e tópicos utilizados para comando, estado e telemetria.

```
import time
import glob
import sys
import signal

import RPi.GPIO as GPIO
import paho.mqtt.client as mqtt

BROKER_HOST = "127.0.0.1"
BROKER_PORT = 1883

TOPIC_BASE = "tcc/motor"
TOPIC_CMD_SET = f"{TOPIC_BASE}/cmd_run/set" # recebe comando
TOPIC_CMD = f"{TOPIC_BASE}/cmd_run" # publica comando (estado)
TOPIC_ST = f"{TOPIC_BASE}/st_motor_on" # publica status do motor

TOPIC_DS18B20 = "tcc/sensors/ds18b20_t"
TOPIC_AHT_T = "tcc/sensors/aht10_t"
TOPIC_AHT_RH = "tcc/sensors/aht10_rh"
TOPIC_RPM = "tcc/inputs/rpm"
```

Fonte: Elaborado pelo autor (2026).

Conforme apresentado na Figura 23, a lógica de comunicação MQTT foi implementada por meio de rotinas de publicação e de tratamento de eventos, organizadas em funções e *callbacks* do cliente. A função `mqtt_publish_states` concentra a publicação do estado do sistema, enviando ao broker os valores atuais de comando e de status do motor nos respectivos tópicos. Nessa publicação, é utilizado o parâmetro `retain=True`, que instrui o broker a armazenar a última mensagem de cada tópico e a disponibilizá-la imediatamente a novos assinantes quando estes se conectarem. Essa estratégia reduz estados indefinidos na inicialização de painéis de supervisão, pois o cliente não precisa aguardar uma nova atualização para receber o último valor válido.

Ainda na Figura 23, o *callback* `on_connect` é executado quando a sessão com o broker é estabelecida com sucesso. Nesse ponto, o cliente realiza a assinatura do tópico de comando destinado ao acionamento do motor, garantindo que mensagens de controle publicadas por outros clientes sejam recebidas pelo servidor. Em seguida, o estado atual do sistema é publicado para o broker, assegurando a coerência inicial entre os tópicos de saída e o estado real mantido pela aplicação.

O *callback* `on_message` é acionado sempre que uma nova mensagem é recebida em um tópico assinado. No caso do projeto, o tratamento é direcionado ao tópico de comando do motor. O conteúdo recebido é interpretado por uma função de conversão para booleano, com rejeição explícita de mensagens inválidas, a fim de evitar alterações indevidas do estado. Após a validação, o novo comando é aplicado ao sistema por meio da rotina responsável pelo acionamento e atualização de variáveis internas. Na sequência, os tópicos de estado são republicados, de forma que o broker reflita imediatamente a alteração realizada, permitindo que o Node-RED e demais clientes apresentem o valor atualizado em tempo compatível com a supervisão.

Figura 23: Rotinas de publicação e callbacks do cliente MQTT para assinatura do tópico de comando, aplicação do estado e republicação com retenção.

```
def mqtt_publish_states(client: mqtt.Client):
    client.publish(TOPIC_CMD, "true" if cmd_run else "false", retain=True)
    client.publish(TOPIC_ST, "true" if st_motor_on else "false", retain=True)

def on_connect(client, userdata, flags, rc):
    print("[MQTT] connected rc=", rc)
    client.subscribe(TOPIC_CMD_SET)
    mqtt_publish_states(client)

def on_message(client, userdata, msg):
    global cmd_run
    if msg.topic == TOPIC_CMD_SET:
        v = parse_bool(msg.payload)
        if v is None:
            print("[MQTT] payload inválido em cmd_run/set:", msg.payload)
            return
        apply_state(v)
        mqtt_publish_states(client)
        print("[MQTT] CMD_RUN <=", v)
```

Fonte: Elaborado pelo autor (2026).

Conforme ilustrado na Figura 24, o cliente MQTT é instanciado por meio da chamada `mqtt.Client()`, que cria o objeto responsável por estabelecer a conexão com o broker e gerenciar a troca de mensagens. Na sequência, são atribuídos ao cliente os *callbacks* `on_connect` e

on_message, de modo que as rotinas de assinatura de tópicos e de processamento de mensagens recebidas sejam executadas automaticamente quando os respectivos eventos ocorrerem.

A conexão é iniciada com `client.connect(BROKER_HOST, BROKER_PORT, 60)`, em que o último parâmetro corresponde ao tempo de *keepalive* em segundos, utilizado para monitoramento e manutenção da sessão. Em seguida, `client.loop_start()` inicia o laço de rede do MQTT em execução paralela, permitindo que a aplicação continue operando no fluxo principal enquanto a biblioteca trata, de forma assíncrona, o envio e o recebimento de mensagens, evitando bloqueios e garantindo a responsividade do sistema.

As mensagens apresentadas no terminal, também mostradas na Figura 24, são empregadas para registrar parâmetros operacionais relevantes, incluindo o endereço do broker, os tópicos assinados para recebimento de comandos, os tópicos publicados para disponibilização de estados e medições, a associação dos pinos GPIO utilizados no protótipo e o tempo configurado para zeramento do RPM na ausência de pulsos. Por fim, é aplicado um estado inicial ao sistema, garantindo a condição segura de partida com o motor desligado, e o estado é publicado imediatamente no broker, assegurando que clientes assinantes, como o Node-RED, recebam valores consistentes desde o início da execução.

Figura 24: Inicialização do cliente MQTT, associação de callbacks, conexão ao broker e publicação do estado inicial do sistema.

```
client = mqtt.Client()
client.on_connect = on_connect
client.on_message = on_message

client.connect(BROKER_HOST, BROKER_PORT, 60)
client.loop_start()

print("[MQTT] Broker:", BROKER_HOST, BROKER_PORT)
print("[MQTT] Motor:")
print("  SUB:", TOPIC_CMD_SET)
print("  PUB:", TOPIC_CMD, TOPIC_ST)
print("[MQTT] Sensores:")
print("  PUB:", TOPIC_DS18B20, TOPIC_AHT_T, TOPIC_AHT_RH, TOPIC_RPM)
print("[GPIO] MOTOR=GPIO17 | BTN_MOTOR=GPIO27 | BTN_PULSE=GPIO26")
print("[RPM] timeout para zerar:", RPM_TIMEOUT_S, "s")

# Estado inicial
apply_state(False)
mqtt_publish_states(client)
```

Fonte: Elaborado pelo autor (2026).

3.4 Aplicação Protocolo Modbus

3.4.1 Iniciando Servidor Modbus

Após acessar a Raspberry Pi por SSH (Figura 25), o servidor Modbus TCP foi iniciado a partir do diretório do projeto, utilizando um ambiente virtual Python para garantir que as dependências do sistema estivessem isoladas e carregadas corretamente. Em seguida, o script do servidor foi executado, inicializando o serviço Modbus com escuta em 0.0.0.0:1502, de modo que o servidor permaneça disponível em todas as interfaces de rede da Raspberry Pi e possa ser acessado por clientes na mesma rede local.

Ao iniciar, o terminal confirma a detecção e operação dos sensores conectados: o DS18B20 é localizado no barramento 1-Wire por meio do arquivo *w1_slave*, e o AHT10 é reconhecido no barramento I²C, indicando que a aquisição de dados está funcional. O sistema também imprime o mapa de endereçamento Modbus, explicitando como cada variável foi disponibilizada para leitura e escrita: o comando de acionamento do motor foi atribuído à Coil 0 (CMD_RUN), o estado do motor à Coil 1 (ST_MOTOR_ON) e as medições foram expostas como Holding Registers com escalonamento ($\times 10$), sendo HR0 para a temperatura do DS18B20, HR1 para a temperatura do AHT10, HR2 para a umidade do AHT10 e HR3 para o valor de RPM.

Por fim, o log apresenta a configuração de GPIOs utilizada na planta didática: GPIO17 representa a saída de acionamento do motor (via SSR e contator), GPIO27 atua como entrada de comando local liga/desliga e GPIO26 é utilizado como entrada de pulsos proveniente do encoder incremental. No código, o cálculo de rotação é realizado pela contagem de pulsos em uma janela de tempo (por exemplo, 10 s), aplicando-se os parâmetros de conversão definidos (como PPR do encoder e fator mecânico do acoplamento) para estimar a rotação do motor. Durante a execução, o terminal imprime continuamente as leituras atualizadas dos sensores e os resultados do cálculo de RPM, evidenciando que o servidor está operacional e publicando variáveis em tempo real para integração com ferramentas como Node-RED e ScadaBR.

Figura 25: Iniciando Servidor Modbus

```
pi@raspjoao:~/tcc $ source .venv/bin/activate
(.venv) pi@raspjoao:~/tcc $ python server_modbus.py
[W1] DS18B20: /sys/bus/w1/devices/28-3ce1d44312df/w1_slave
[I2C] AHT10 OK
[MODBUS] Server ON em 0.0.0.0:1502
[MAPA] Coil0=CMD_RUN | Coil1=ST_MOTOR_ON | HR0=DS18B20_Tx10 | HR1=AHT10_Tx10 | HR2=AHT10_RHx10 | HR3=RPM_MOTOR_x10
[RPM] janela=10s | PPR_encoder=10 | motor/encoder=4.0625
[BOTA0] Toggle CMD_RUN -> True
[MOTOR] ON (GPIO17=HIGH)
[RPM] Pulsos=739 em 10.00s -> Motor=1800.86 rpm -> HR3=18009
[RPM] Pulsos=931 em 10.03s -> Motor=2262.63 rpm -> HR3=22626
[RPM] Pulsos=1492 em 10.04s -> Motor=3621.77 rpm -> HR3=36218
[RPM] Pulsos=1488 em 10.02s -> Motor=3619.37 rpm -> HR3=36194
```

Fonte: Elaborado pelo autor (2026).

3.4.2 Integração Modbus TCP no Node-RED e construção da dashboard

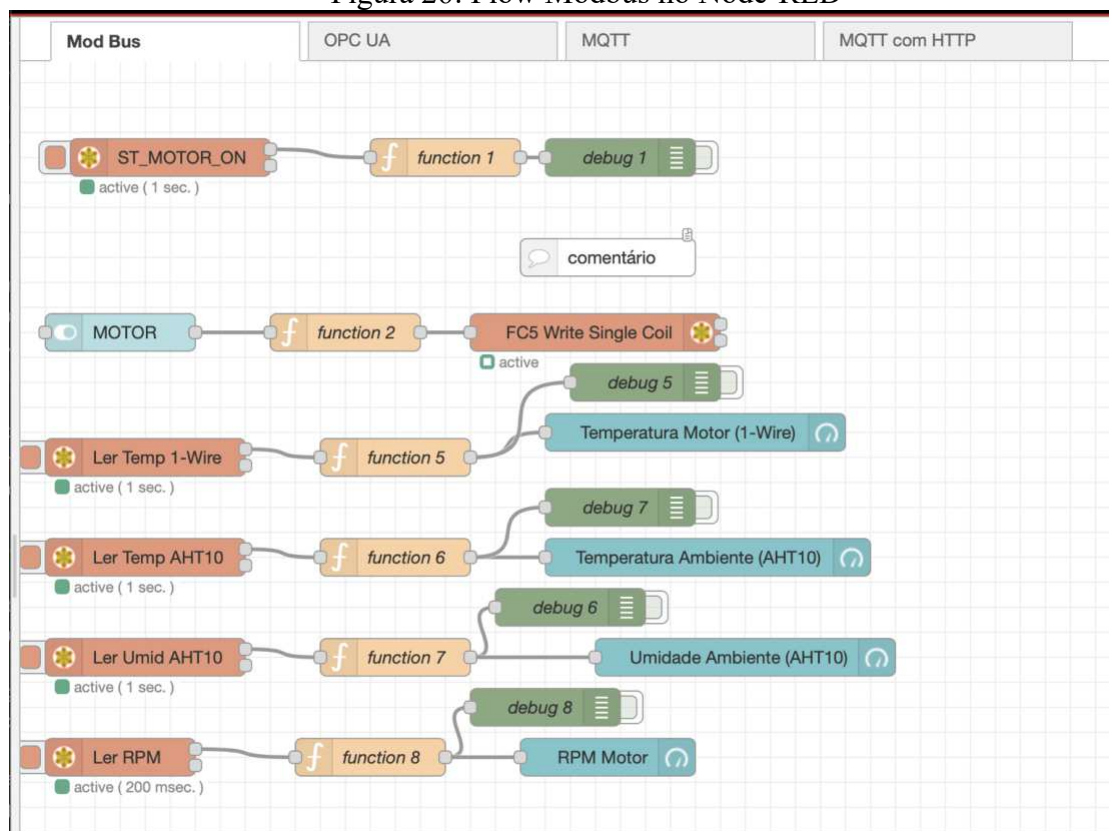
A Figura 26 apresenta o *flow* desenvolvido no Node-RED para comunicação com o servidor Modbus TCP executado na Raspberry Pi, separando claramente as funções de leitura e escrita. Na parte superior, o nó ST_MOTOR_ON realiza a leitura periódica do estado do motor no servidor Modbus e envia o valor para um nó *function* que apenas formata a mensagem, permitindo acompanhar o dado pelo nó de depuração e utilizar o mesmo valor em componentes de supervisão. Esse caminho garante que o Node-Red esteja sempre sincronizado com a variável de status publicada pelo servidor.

Para o comando de acionamento, o fluxo utiliza um controle na dashboard chamado MOTOR, que envia a ação do operador para um nó *function* responsável por converter o comando da interface em um formato compatível com escrita Modbus. Em seguida, o nó FC5 Write Single Coil executa a escrita de uma única bobina no servidor, correspondendo ao registrador de comando do motor, o que permite ligar e desligar o acionamento de forma remota por meio do protocolo Modbus TCP.

As variáveis de processo medidas pelos sensores foram implementadas como leituras independentes: Ler Temp 1-Wire, Ler Temp AHT10, Ler Umid AHT10 e Ler RPM. Cada uma delas executa leituras periódicas do tipo *Holding Register* e encaminha o valor para uma função de ajuste de escala e conversão, já que as medições foram publicadas no servidor em formato inteiro multiplicado por 10. Após a conversão, os valores são exibidos em gauges específicos na dashboard, identificados como Temperatura Motor (1-Wire), Temperatura Ambiente (AHT10), Umidade Ambiente (AHT10) e RPM Motor, permitindo ao usuário visualizar em tempo real tanto as grandezas ambientais quanto a velocidade de rotação medida pelo encoder e convertida em RPM.

Por fim, os nós de debug foram mantidos no fluxo como ferramenta de validação, permitindo confirmar rapidamente se cada leitura ou escrita está chegando ao Node-RED e se as mensagens estão sendo interpretadas corretamente antes de serem exibidas na interface. Essa estrutura modular facilita a expansão do projeto, pois novos sensores podem ser adicionados replicando o mesmo padrão de leitura, conversão e visualização.

Figura 26: Flow Modbus no Node-RED



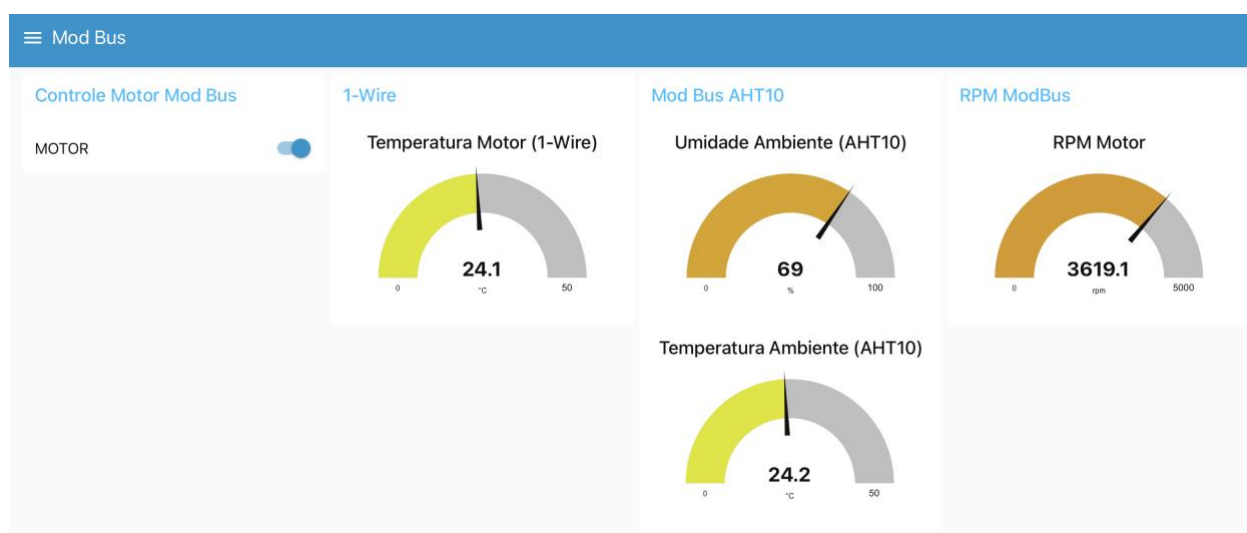
Fonte: Elaborado pelo autor (2026).

3.4.3 Dashboard Node-RED Modbus

A Figura 27 mostra a dashboard construída no Node-RED, apresenta a supervisão e o comando do protótipo via Modbus TCP, consumindo diretamente os dados publicados pelo servidor Modbus rodando na Raspberry Pi. O switch “MOTOR” representa a ação do operador e está ligado a uma escrita Modbus do tipo FC5 (Write Single Coil), que altera a bobina de comando do motor. Em paralelo, o estado do motor é monitorado por uma leitura periódica da bobina de status, permitindo confirmar na interface se o comando foi aplicado corretamente.

Além do acionamento, a dashboard exibe variáveis medidas por sensores em forma de indicadores do tipo gauge. A leitura de Temperatura Motor (1-Wire) corresponde à medição do sensor DS18B20, publicada no servidor como Holding Register HR0 com escala de $^{\circ}\text{C} \times 10$, sendo convertida no fluxo para $^{\circ}\text{C}$ antes de ser exibida. As leituras do AHT10 são separadas em Umidade Ambiente e Temperatura Ambiente, provenientes de HR2 e HR1 respectivamente, também com escala ajustada no Node-RED. Por fim, o gauge RPM Motor representa a variável de velocidade calculada a partir dos pulsos do encoder incremental e disponibilizada no servidor em HR3 com escala $\text{RPM} \times 10$, permitindo demonstrar a aquisição e visualização de uma grandeza dinâmica no mesmo barramento Modbus.

Figura 27: Dashboard Node-RED Modbus



Fonte: Elaborado pelo autor (2026).

3.4.4 Modbus ScadaBR

Na tela da Figura 28 é apresentada a lista de fontes de dados configuradas no projeto. Foram criados dois data sources principais: RPI_Modbus, do tipo Modbus IP, apontando para 127.0.0.1:1502, e RPI_HTTP, do tipo Recuperador HTTP, apontando para um endpoint HTTP hospedado na própria Raspberry Pi via Node-RED. O campo Status indica que as conexões estão

estabelecidas, evidenciado pelo ícone verde, confirmando que o ScadaBR está conseguindo acessar tanto os dados via Modbus TCP quanto via HTTP.

Figura 28: Data Sources ScadaBR



Fonte: Elaborado pelo autor (2026).

Na configuração do data source RPI_Modbus, como vista na Figura 29 o ScadaBR foi parametrizado para atuar como cliente Modbus TCP, realizando leituras e escritas no servidor Modbus executado na Raspberry Pi. O transporte selecionado foi TCP, com Host 127.0.0.1 e Porta 1502, indicando que o ScadaBR está acessando um serviço rodando localmente na própria Raspberry. O período de atualização foi configurado para 1 segundo, permitindo que as variáveis sejam atualizadas quase em tempo real na interface supervisória. Também foram definidos timeout de 500 ms e 2 retentativas, aumentando a robustez da comunicação em caso de perda momentânea de resposta. Essa tela ainda inclui ferramentas de diagnóstico, como leitura direta de faixa de registradores e teste de localizador de ponto, úteis para validar offsets e endereçamentos durante a configuração.

A tabela de Data points mostra como cada variável do seu sistema foi ligada a um endereço Modbus específico. Os sinais digitais do motor foram mapeados em Bobinas (Coils), enquanto as variáveis analógicas foram mapeadas como Registradores Holding. Assim, o comando de acionamento CMD_RUN foi configurado como Coil offset 0, e o status ST_MOTOR_ON como Coil offset 1, refletindo exatamente o mapa do servidor no código. Para as medições, o DS18B20_T foi associado ao Holding Register offset 0, a temperatura do AHT10_T ao offset 1, a umidade AHT10_RH ao offset 2, e o RPM ao offset 3. Dessa forma, o ScadaBR consegue ler e exibir as grandezas físicas e enviar comandos para o processo através de escrita em bobina.

Figura 29: Configuração do Data Source Modbus TCP no ScadaBR e mapeamento dos data points para bobinas e registradores holding

Propriedades do Modbus IP

Nome: RPI_Modbus
 Export ID (XID): DS_011772
 Período de atualização: 1 segundo(s)
 Quantização: ☐
 Timeout (ms): 500
 Retentativas: 2
 Apenas quantidades contíguas: ☐
 Criar pontos de monitor de escravo: ☐
 Máxima contagem de leitura de bits: 2000
 Máxima contagem de leitura de registradores: 125
 Máxima contagem de escrita de registradores: 120
 Tipo de transporte: TCP
 Host: 127.0.0.1
 Porta: 1502
 Encapsulado: ☐
 Criar ponto monitor de conexão: ☐

Níveis de alarme de eventos

Exceção de data source: Nenhum alarme
 Exceção de leitura de data point: Nenhum alarme
 Exceção de escrita em data point: Nenhum alarme

Pesquisa de nós modbus

Pesquisar por nós:
 Nós encontrados:

Leitura de dados Modbus

Id do escravo: 1
 Faixa do registro: Bobina (Coil Status)
 Offset (baseado em 0): 0
 Número de registradores: 100

Teste de localizador de ponto

Id do escravo: 1
 Faixa do registro: Bobina (Coil Status)
 Tipo de dados modbus: Binário
 Offset (baseado em 0): 0
 Bit: 0
 Número de registradores: 0
 Codificação de caracteres: ASCII

Data points

Nome	Tipo de dado	Status	Escravo	Faixa	Offset (baseado em 0)
AHT10_RH	Numérico		1	Registrador holding	2
AHT10_T	Numérico		1	Registrador holding	1
CMD_RUN	Binário		1	Bobina (Coil Status)	0
DS18B20_T	Numérico		1	Registrador holding	0
RPM	Numérico		1	Registrador holding	3
ST_MOTOR_ON	Binário		1	Bobina (Coil Status)	1

Fonte: Elaborado pelo autor (2026).

Após a criação do data source Modbus TCP e o cadastro dos data points, foi utilizada a funcionalidade Watch list do ScadaBR para acompanhar, em tempo real, os valores lidos do servidor Modbus executado na Raspberry Pi (Figura 30). Nessa tela é possível verificar simultaneamente variáveis digitais e analógicas, além do instante da última atualização. Os pontos CMD_RUN e ST_MOTOR_ON aparecem como variáveis binárias, representando respectivamente o comando de acionamento e o estado do motor. Já os valores de temperatura e umidade são exibidos nos pontos DS18B20_T, AHT10_T e AHT10_RH, confirmando a aquisição dos sensores e a entrega desses dados via registradores holding. Por fim, o ponto RPM apresenta o valor de rotação calculado a partir dos pulsos do encoder incremental, demonstrando que também entradas digitais podem ser convertidas em grandezas de interesse e disponibilizadas ao supervisor. A coluna de horário mostra que as leituras estão sendo atualizadas continuamente dentro do período configurado no data source, validando a comunicação Modbus TCP entre aplicação e supervisão.

Figura 30: Watch list do ScadaBR com pontos Modbus TCP do motor e sensores, apresentando valores e instante de atualização

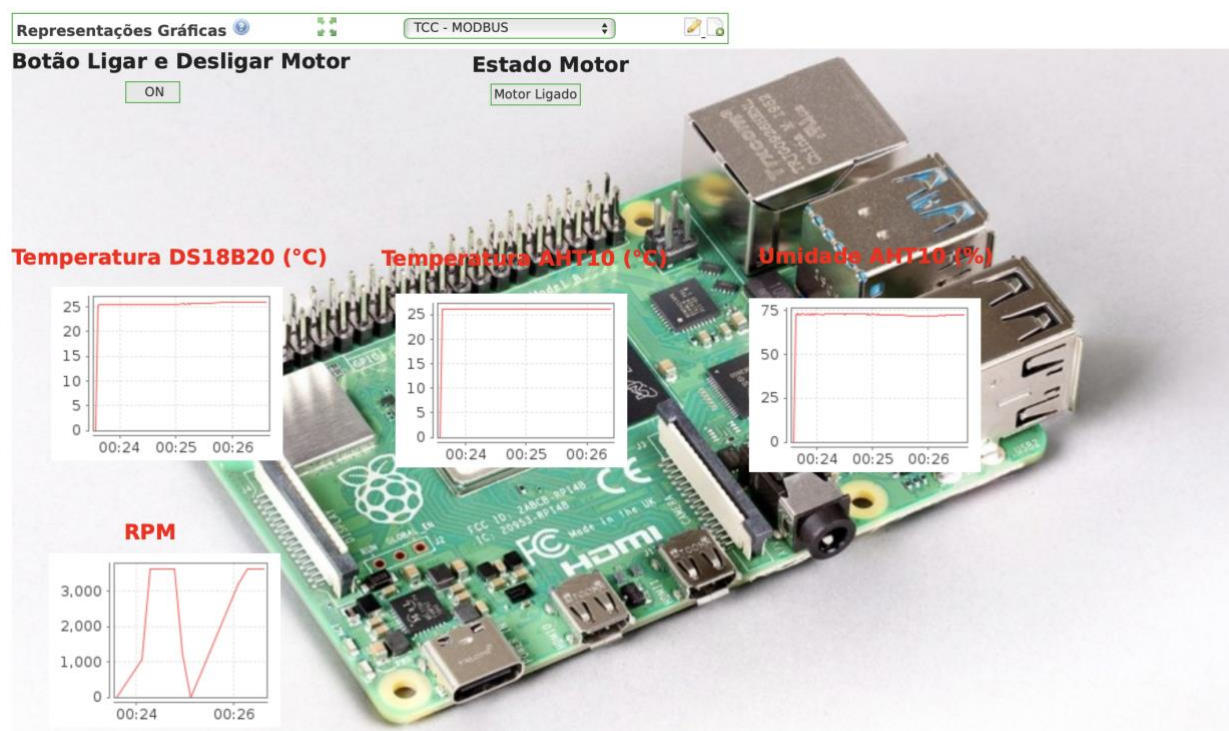
 RPI_Modbus - CMD_RUN	1	00:24:30	   
 RPI_Modbus - ST_MOTOR_ON	1	00:24:30	   
 RPI_Modbus - RPM	3619.3	00:24:30	   
 RPI_Modbus - AHT10_RH	72.8	00:24:30	   
 RPI_Modbus - AHT10_T	26.0	00:24:30	   
 RPI_Modbus - DS18B20_T	25.400000000000002	00:24:30	   

Fonte: Elaborado pelo autor (2026).

Após a validação das leituras na Watch list, foi criada uma Representação Gráfica no ScadaBR para reunir, em uma única tela, os principais elementos de supervisão do protótipo via Modbus TCP (Figura 31). Na parte superior foram inseridos dois componentes relacionados ao acionamento do motor: um botão de comando, que escreve no ponto CMD_RUN para ligar e desligar o motor, e um indicador de estado, vinculado ao ponto ST_MOTOR_ON, que confirma visualmente se o motor se encontra ligado ou desligado.

Em seguida, foram adicionados gráficos de tendência associados aos data points dos sensores e do RPM, permitindo acompanhar a evolução temporal das variáveis. A tela inclui gráficos para a temperatura do DS18B20, temperatura do AHT10, umidade do AHT10 e a rotação em RPM, evidenciando que o supervisor é capaz de integrar variáveis analógicas e digitais provenientes da Raspberry Pi, com atualização contínua e histórico, além de possibilitar intervenção do usuário por meio do comando de acionamento.

Figura 31: Representação gráfica no ScadaBR para o protocolo Modbus TCP, com comando do motor, indicação de estado e gráficos de tendência para temperatura, umidade e RPM.



Fonte: Elaborado pelo autor (2026).

3.5 Aplicação Protocolo OPC UA

3.5.1 Inicialização do servidor OPC UA na Raspberry Pi

A inicialização do servidor OPC UA (Figura 32) foi realizada no terminal da Raspberry Pi, acessando o diretório do projeto e ativando o ambiente virtual Python com `source .venv/bin/activate`. Em seguida, o servidor foi executado pelo comando `python server_opcua.py`, garantindo que as dependências e bibliotecas do projeto fossem carregadas a partir do ambiente isolado, evitando conflitos com pacotes instalados globalmente no sistema operacional. Ao iniciar, o programa confirma o reconhecimento dos sensores conectados ao hardware, indicando a presença do DS18B20 no barramento 1-Wire e a inicialização correta do AHT10 no barramento I²C. Essa etapa valida que a aquisição de dados físicos está disponível antes da publicação das variáveis no servidor OPC UA, permitindo que as medições sejam atualizadas continuamente e consumidas por clientes externos.

O servidor OPC UA foi disponibilizado no endpoint `opc.tcp://0.0.0.0:4840/tcc/`. O endereço 0.0.0.0 indica que o serviço permanece em escuta em todas as interfaces de rede da Raspberry Pi, permitindo acesso a partir de outros dispositivos na mesma rede. A porta 4840 é a porta usual em implementações OPC UA e facilita a compatibilidade com ferramentas e clientes

OPC UA. O sufixo /tcc/ foi utilizado para identificar o contexto do projeto no endereço do servidor. Na sequência, o programa registra o namespace do projeto e informa o índice associado, exibindo Namespace index (tcc): ns=2. Esse índice é utilizado pelos clientes OPC UA para compor o NodeId de cada variável, garantindo que os nós do projeto sejam diferenciados do namespace padrão do OPC UA.

O mapa de nós publicado foi organizado em três grupos principais. Em Motor, foram criadas as variáveis CMD_RUN e ST_MOTOR_ON, em que CMD_RUN é configurada como gravável e permite que um cliente escreva o comando de liga/desliga, enquanto ST_MOTOR_ON permanece somente leitura e representa o estado do motor. Em Sensors, foram disponibilizadas as variáveis DS18B20_T, AHT10_T e AHT10_RH no tipo numérico *Double* para representar temperatura e umidade. Em Inputs, foi publicada a variável RPM também como *Double*, representando a velocidade a partir dos pulsos do encoder. Após a publicação dos nós, o terminal confirma que o servidor entrou em operação com Server STARTED e que está Listening on 0.0.0.0:4840. O aviso relacionado a certificados indica que apenas endpoints abertos estão habilitados, pois não foram definidos chave privada e certificado para comunicação segura; ainda assim, em contexto didático e em rede local, o servidor permanece funcional para testes de leitura e escrita de variáveis via OPC UA.

Por fim, a execução contínua do servidor é evidenciada pela impressão periódica das medições dos sensores e pelo registro do cálculo de rotação. O comando local do motor permanece associado ao GPIO27, enquanto o GPIO26 é utilizado como entrada de pulsos do encoder incremental. A velocidade é obtida por contagem de pulsos em uma janela de aquisição, conforme indicado no terminal (por exemplo, janela de 10 s), com conversão para RPM a partir da resolução do encoder e da relação mecânica adotada, permitindo a supervisão contínua da velocidade do motor.

Figura 32: Terminal SSH inicializando o servidor OPC UA

```
(.venv) pi@raspjoao:~/tcc $ python server_opcua.py
[W1] DS18B20: /sys/bus/w1/devices/28-3ce1d44312df/w1_slave
[I2C] AHT10 OK
[OPCUA] Endpoint: opc.tcp://0.0.0.0:4840/tcc/ ns=2
Endpoints other than open requested but private key and certificate are not set.
Listening on 0.0.0.0:4840
[OPCUA] Server STARTED
[RPM] janela=10s | PPR_encoder=10 | motor/encoder=4.0625
[RPM] Pulsos=0 em 10.00s -> Motor=0.00 rpm
[RPM] Pulsos=0 em 10.00s -> Motor=0.00 rpm
[MOTOR] ON (GPIO17=HIGH)
[RPM] Pulsos=1026 em 10.04s -> Motor=2489.77 rpm
[RPM] Pulsos=1487 em 10.02s -> Motor=3618.75 rpm
[RPM] Pulsos=1492 em 10.05s -> Motor=3619.78 rpm
[RPM] Pulsos=1487 em 10.02s -> Motor=3618.30 rpm
[RPM] Pulsos=1486 em 10.02s -> Motor=3616.56 rpm
```

Fonte: Elaborado pelo autor (2026).

3.5.2 OPC UA no Node-RED: leitura, escrita e supervisão

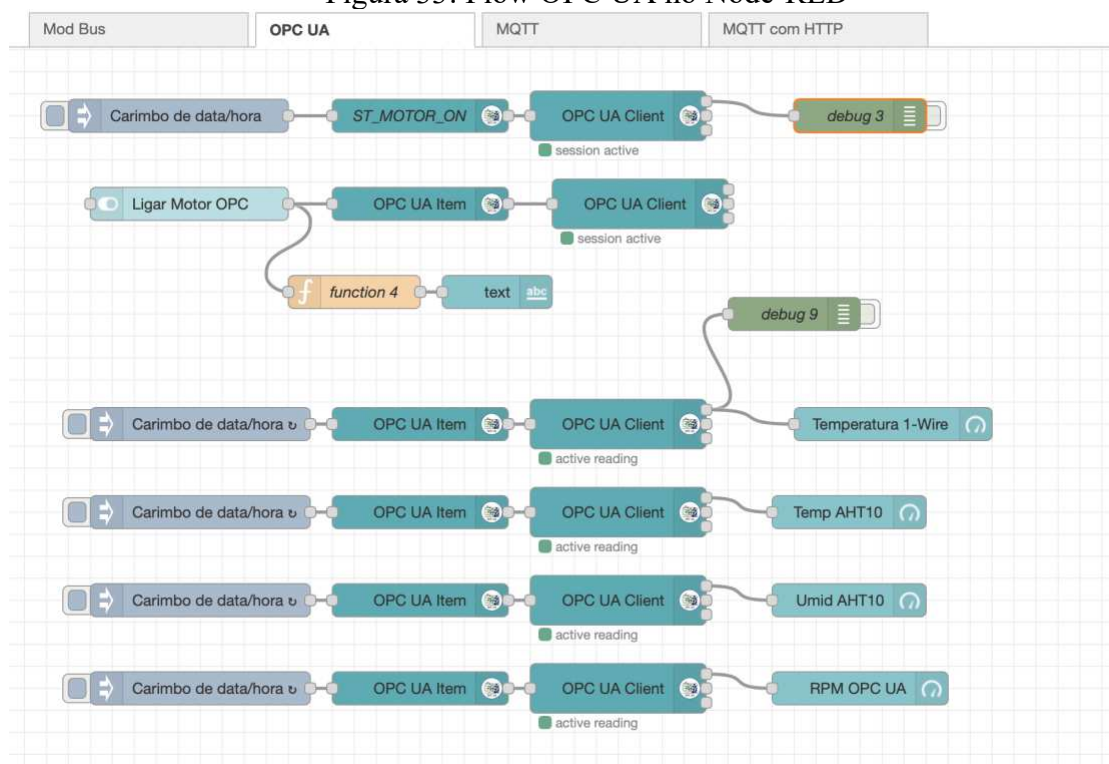
Nesta etapa, foi desenvolvido um flow no Node-RED (Figura 33) para a realização de controle e supervisão utilizando o protocolo OPC UA, com o Node-RED operando como cliente conectado ao servidor executado na Raspberry Pi. A estrutura foi organizada em blocos funcionais, contemplando o acionamento do motor e a aquisição das variáveis de processo. Para a supervisão do estado do motor, foi empregado um gatilho do tipo carimbo de data e hora, responsável por disparar leituras periódicas. O gatilho alimenta um nó OPC UA Item configurado com o NodeID da variável de status ST_MOTOR_ON e, em seguida, o nó OPC UA Client realiza a leitura em sessão ativa, encaminhando o valor obtido para depuração e para a interface de visualização, assegurando atualização contínua na dashboard.

O comando de acionamento do motor foi implementado por meio de um componente de interface do Node-RED, que gera um valor booleano associado ao comando. Esse valor é direcionado a um nó OPC UA Item configurado para escrita, vinculado ao NodeID correspondente à variável CMD_RUN. Na sequência, o nó OPC UA Client mantém a sessão ativa e efetua a escrita no servidor OPC UA, permitindo a alteração remota do comando do motor. Adicionalmente, foi utilizado um bloco de função para tratamento do valor e um componente textual para exibição do estado, de modo a fornecer confirmação visual do acionamento na interface.

De forma análoga, foram implementados fluxos independentes para as variáveis DS18B20_T, AHT10_T, AHT10_RH e RPM. Cada fluxo foi composto por um gatilho periódico, um nó OPC UA Item configurado com o NodeID específico e um nó OPC UA Client em modo de

leitura contínua. Os valores adquiridos foram encaminhados aos instrumentos de visualização do tipo gauge e, quando necessário, a nós de depuração para validação durante a etapa de testes. Essa organização modular facilita a manutenção do sistema e caracteriza o uso do OPC UA no Node-RED tanto para escrita de comandos quanto para leitura contínua de variáveis, configurando uma arquitetura típica de supervisão em sistemas de automação industrial.

Figura 33: Flow OPC UA no Node-RED

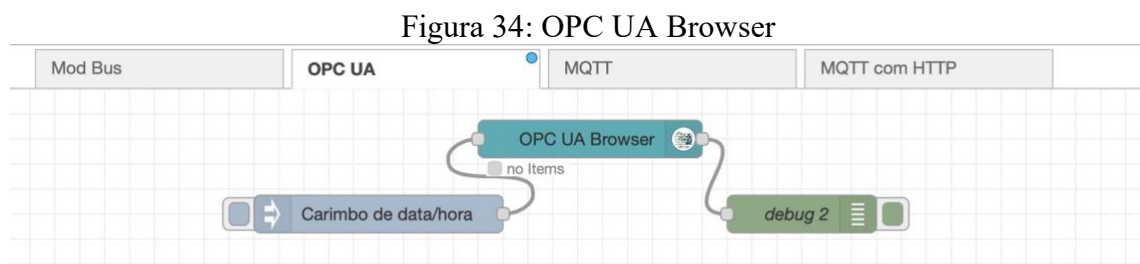


Fonte: Elaborado pelo autor (2026).

3.5.3 Descoberta de NodeIDs com OPC UA Browser

Como o OPC UA organiza variáveis em uma hierarquia de objetos e nós, foi necessário identificar corretamente os NodeIDs (formato ns=X;i=Y) antes de configurar os nós OPC UA Item. Para isso foi utilizado o nó OPC UA Browser no Node-RED, responsável por navegar na estrutura do servidor e retornar, via debug, os objetos e variáveis disponíveis.

O procedimento começou com a execução do Browser na raiz do servidor, utilizando um NodeID base do espaço padrão, o que permitiu visualizar a existência das pastas/objetos principais do projeto, como Motor, Sensors e Inputs, além do nó Server do namespace padrão. A partir dessa lista inicial foi possível confirmar a estrutura do *Address Space* do servidor OPC UA e entender onde cada variável estava organizada.



Fonte: Elaborado pelo autor (2026).

Na Figura 35 e Figura 36 o Browser é executado a partir de um NodeID inicial de referência do servidor, e o resultado é visualizado no painel Debug. O retorno mostra uma lista de objetos encontrados, incluindo o nó padrão Server e os objetos do projeto no namespace do TCC, como Motor, Sensors e Inputs. Essa listagem confirma que o servidor está ativo, que o Node-RED está conseguindo navegar no endpoint e que a estrutura lógica do projeto está corretamente publicada no servidor OPC UA.

Figura 35: Debug OPC UA Browser com Node ID inicial de referência ao servidor Parte 1

```

05/03/2026, 04:27:06  nó: debug 2
msg : Object
  ▾ object
    _msgid: "5da356e22394e91b"
    ▾ payload: array[4]
      ▾ 0: object
        ▾ item: object
          referenceTypeId: "ns=0;i=35"
          isForward: true
          nodeId: "ns=0;i=2253"
          ▾ browseName: object
            namespaceIndex: 0
            name: "Server"
          ▸ displayName: object
            nodeClass: 1
            typeDefinition: "ns=0;i=2004"
            value: null
            dataType: "Null"
        ▾ 1: object
          ▾ item: object
            referenceTypeId: "ns=0;i=35"
            isForward: true
            nodeId: "ns=2;i=1"
            ▾ browseName: object
              namespaceIndex: 2
              name: "Motor"
            ▸ displayName: object
              nodeClass: 1
              typeDefinition: "ns=0;i=58"
              value: null
              dataType: "Null"

```

Fonte: Elaborado pelo autor (2026).

Figura 36: Debug OPC UA Browser com Node ID inicial de referência ao servidor Parte 2

```

  ▾ 2: object
    ▾ item: object
      referenceTypeId: "ns=0;i=35"
      isForward: true
      nodeId: "ns=2;i=2"
      ▾ browseName: object
        namespaceIndex: 2
        name: "Sensors"
      ▸ displayName: object
        nodeClass: 1
        typeDefinition: "ns=0;i=58"
        value: null
        dataType: "Null"
  ▾ 3: object
    ▾ item: object
      referenceTypeId: "ns=0;i=35"
      isForward: true
      nodeId: "ns=2;i=3"
      ▾ browseName: object
        namespaceIndex: 2
        name: "Inputs"
      ▸ displayName: object
        nodeClass: 1
        typeDefinition: "ns=0;i=58"
        value: null
        dataType: "Null"
  topic: ""
  endpoint: "opc.tcp://127.0.0.1:4840/tcc/"

```

Fonte: Elaborado pelo autor (2026).

A Figura 37 mostra o Browser sendo utilizado para entrar no objeto Motor (ns=2,i=1) e listar seus nós internos. No Debug aparecem duas variáveis principais: CMD_RUN e ST_MOTOR_ON, ambas do tipo Boolean. Nessa etapa são identificados os NodeIDs específicos dessas variáveis, que no seu caso ficaram como ns=2;i=4 para CMD_RUN e ns=2;i=5 para ST_MOTOR_ON. A partir desses NodeIDs, o Node-RED consegue implementar o controle do motor escrevendo em CMD_RUN e a supervisão lendo ST_MOTOR_ON.

Figura 37: Debug OPC UA Browser com Node ID (ns=2,i=1) do objeto “Motor”

```
05/03/2026, 04:32:36  nó: debug 2
msg : Object
  object
    _msgid: "9d853e51b2b1bb30"
    payload: array[2]
      0: object
        item: object
          referenceTypeId: "ns=0;i=47"
          isForward: true
          nodeId: "ns=2;i=4"
          browseName: object
            namespaceIndex: 2
            name: "CMD_RUN"
          displayName: object
            nodeClass: 2
            typeDefinition: "ns=0;i=63"
            value: false
            dataType: "Boolean"
      1: object
        item: object
          referenceTypeId: "ns=0;i=47"
          isForward: true
          nodeId: "ns=2;i=5"
          browseName: object
            namespaceIndex: 2
            name: "ST_MOTOR_ON"
          displayName: object
            nodeClass: 2
            typeDefinition: "ns=0;i=63"
            value: false
            dataType: "Boolean"
    topic: ""
    endpoint: "opc.tcp://127.0.0.1:4840/tcc/"
```

Fonte: Elaborado pelo autor (2026).

Na Figura 38 e Figura 39 o Browser é executado dentro do objeto Sensors. O Debug retorna as variáveis publicadas para instrumentação do sistema, incluindo DS18B20_T, AHT10_T e AHT10_RH, agora com tipo Double, pois representam grandezas analógicas de temperatura e umidade em ponto flutuante. Esses NodeIDs são então copiados e usados nos nós OPC UA Item configurados para leitura contínua, permitindo alimentar gauges e displays na dashboard do Node-RED.

Figura 38: Debug OPC UA Browser com Node ID (ns=2,i=2) do objeto “Sensors” Parte1

```

05/03/2026, 04:37:42  nó: debug 2
msg : Object
  object
    _msgid: "f13cc1039ca2b346"
    payload: array[3]
      0: object
        item: object
          referenceTypeId: "ns=0;i=47"
          isForward: true
          nodeId: "ns=2;i=6"
          browseName: object
            namespaceIndex: 2
            name: "DS18B20_T"
          displayName: object
            nodeClass: 2
            typeDefinition: "ns=0;i=63"
            value: 25
            dataType: "Double"
      1: object
        item: object
          referenceTypeId: "ns=0;i=47"
          isForward: true
          nodeId: "ns=2;i=7"
          browseName: object
            namespaceIndex: 2
            name: "AHT10_T"
          displayName: object
            nodeClass: 2
            typeDefinition: "ns=0;i=63"
            value: 24.796295166015625
            dataType: "Double"

```

Fonte: Elaborado pelo autor (2026).

Figura 39: Debug OPC UA Browser com Node ID (ns=2,i=2) do objeto “Sensors” Parte2

```

  2: object
    item: object
      referenceTypeId: "ns=0;i=47"
      isForward: true
      nodeId: "ns=2;i=8"
      browseName: object
        namespaceIndex: 2
        name: "AHT10_RH"
      displayName: object
        nodeClass: 2
        typeDefinition: "ns=0;i=63"
        value: 60.11552810668945
        dataType: "Double"
    topic: ""
    endpoint: "opc.tcp://127.0.0.1:4840/tcc/"

```

Fonte: Elaborado pelo autor (2026).

Na Figura 40 o Browser é executado dentro do objeto Inputs, que agrupa variáveis relacionadas a entradas do sistema. O Debug retorna o nó RPM, também do tipo Double, que representa a rotação obtida a partir dos pulsos do encoder. Após localizar o NodeID correspondente

ao RPM, esse valor passa a ser lido periodicamente pelo Node-RED e enviado para visualização na dashboard.

Figura 40: Debug OPC UA Browser com Node ID (ns=2,i=3) do objeto “Inputs”

```
05/03/2026, 04:42:25  nó: debug 2
msg : Object
  object
    _msgid: "a7445f9577db226e"
    payload: array[1]
      0: object
        item: object
          referenceTypeId: "ns=0;i=47"
          isForward: true
          nodeId: "ns=2;i=9"
          browseName: object
            namespaceIndex: 2
            name: "RPM"
          displayName: object
            nodeClass: 2
            typeDefinition: "ns=0;i=63"
            value: 0
            dataType: "Double"
        topic: ""
        endpoint: "opc.tcp://127.0.0.1:4840/tcc/"
```

Fonte: Elaborado pelo autor (2026).

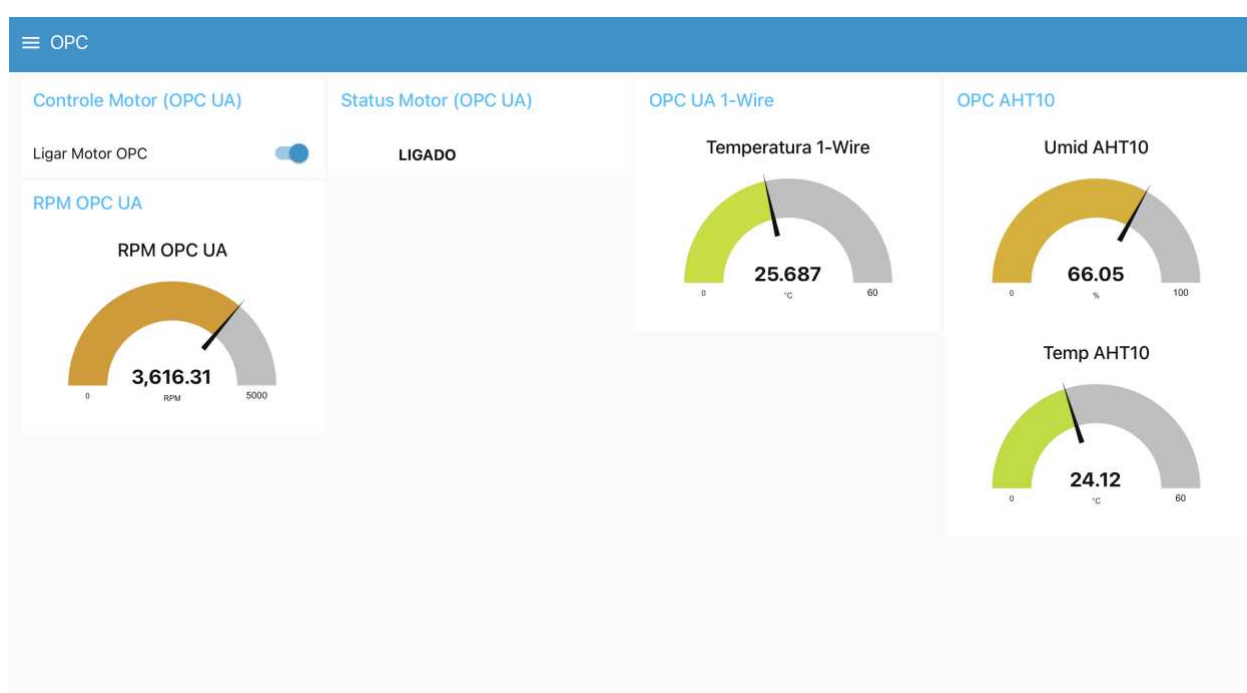
3.5.4 Dashboard OPC UA

Nesta etapa, foi construída uma dashboard no Node-RED (Figura 41) para a supervisão e o controle do sistema por meio do protocolo OPC UA. A interface foi organizada em painéis funcionais, de modo a separar o acionamento do motor, a indicação de estado e a visualização das variáveis medidas. O painel de controle do motor disponibiliza um elemento do tipo chave com operação booleana, responsável por enviar o comando de partida ao servidor OPC UA. Em paralelo, o painel de status apresenta a condição operacional do motor em formato textual, permitindo a confirmação imediata do estado lido a partir da variável de supervisão.

A visualização das grandezas de processo foi realizada por meio de instrumentos do tipo gauge, adequados para acompanhamento em tempo real e interpretação rápida dos valores. A temperatura do motor, obtida pelo sensor DS18B20 via 1-Wire, foi apresentada em escala compatível com o intervalo de operação esperado. As variáveis de ambiente provenientes do sensor

AHT10 foram exibidas em dois instrumentos independentes, contemplando temperatura e umidade relativa. Por fim, o valor de velocidade de rotação do motor, representado em rotações por minuto, foi exibido em instrumento específico para RPM, permitindo a avaliação do comportamento dinâmico do sistema durante os testes. Essa dashboard caracterizou a integração do Node-RED como cliente OPC UA, consolidando em uma mesma interface as funções de comando, supervisão de estado e monitoramento de variáveis do processo.

Figura 41: Dashboard Node-RED OPC UA



Fonte: Elaborado pelo autor (2026).

3.6 Aplicação Protocolo MQTT

3.6.1 Inicialização Servidor MQTT

A partir do registro apresentado no terminal, foi realizada a inicialização do servidor MQTT na Raspberry Pi (Figura 42) para viabilizar a publicação e a assinatura das variáveis do sistema em tópicos padronizados. Após ativar o ambiente virtual Python e executar o script `server_mqtt.py`, o programa confirmou a detecção dos sensores DS18B20 e AHT10 e estabeleceu

conexão com o broker local no endereço 127.0.0.1 e porta 1883, retornando $rc = 0$, o que indica conexão bem-sucedida.

Na sequência, o terminal apresenta parâmetros relevantes para a medição de rotação, incluindo a janela de aquisição adotada ($janela = 3$ s), a resolução do encoder ($PPR_encoder = 10$) e o fator de conversão utilizado para estimar a rotação do motor a partir da rotação do encoder ($motor/encoder = 4,0625$), consistente com o acoplamento indireto implementado no sistema. Em seguida, são exibidas as leituras sucessivas de rotação obtidas a partir da contagem de pulsos por janela, evidenciando o cálculo do RPM do motor e a atualização contínua do valor. Observa-se que, após a partida do motor (mensagem [MOTOR] ON (GPIO17=HIGH)), o RPM calculado estabiliza em torno de $3,6 \times 10^3$ rpm, com variações pequenas entre as amostras, o que demonstra consistência na aquisição dos pulsos do encoder e no processamento da rotação.

Por fim, o log confirma a execução integrada do sistema, com acionamento do motor pela saída GPIO17 e cálculo de RPM baseado na contagem de pulsos do encoder no intervalo configurado, validando o funcionamento do servidor MQTT para disponibilização da variável de rotação e das demais medições no contexto multiprotocolo do projeto.

Figura 42: Terminal SSH inicializando MQTT

```
(.venv) pi@raspjoao:~/tcc $ python server_mqtt.py
[W1] DS18B20: /sys/bus/w1/devices/28-3ce1d44312df/w1_slave
[I2C] AHT10 OK
/home/pi/tcc/server_mqtt.py:200: DeprecationWarning: Callback API version 1 is deprecated, update to latest version
  client = mqtt.Client()
[MQTT] Broker: 127.0.0.1 1883
[RPM] janela=3s | PPR_encoder=10 | motor/encoder=4.0625
[MQTT] connected rc= 0
[RPM] Pulsos=0 em 3.02s -> Motor=0.00 rpm
[MOTOR] ON (GPIO17=HIGH)
[RPM] Pulsos=297 em 3.02s -> Motor=2397.61 rpm
[RPM] Pulsos=446 em 3.01s -> Motor=3612.36 rpm
[RPM] Pulsos=446 em 3.01s -> Motor=3614.77 rpm
[RPM] Pulsos=448 em 3.02s -> Motor=3619.30 rpm
[RPM] Pulsos=448 em 3.02s -> Motor=3617.41 rpm
[RPM] Pulsos=447 em 3.01s -> Motor=3621.86 rpm
[RPM] Pulsos=448 em 3.02s -> Motor=3614.72 rpm
[RPM] Pulsos=447 em 3.01s -> Motor=3623.74 rpm
```

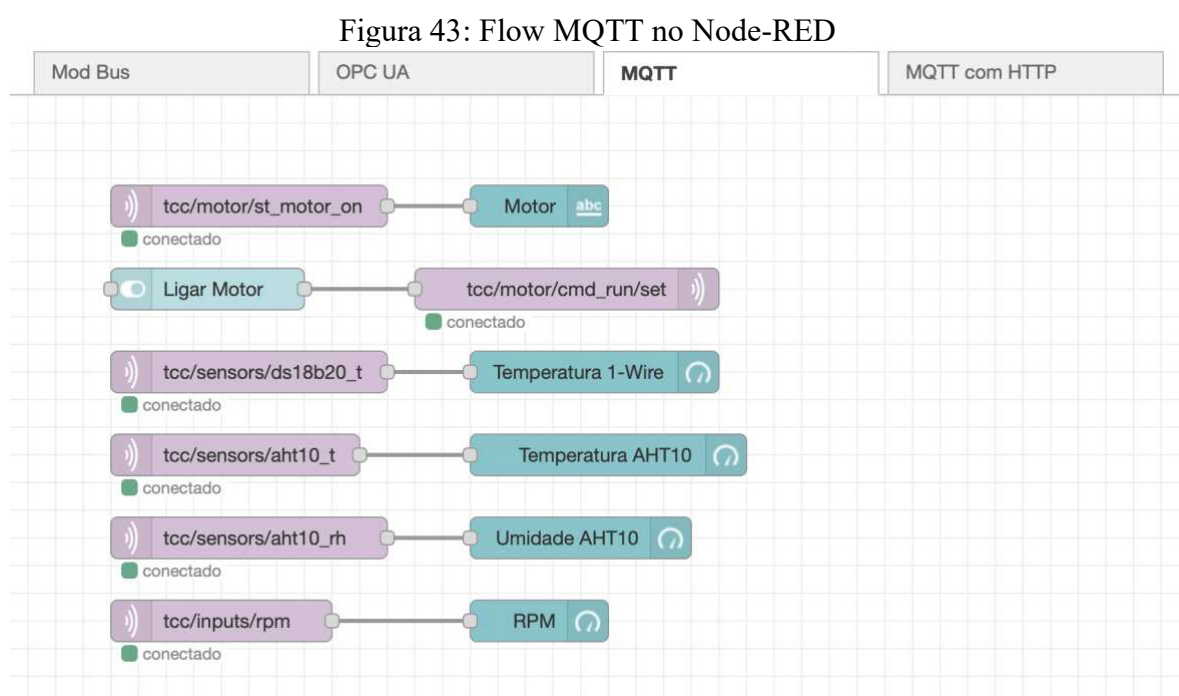
Fonte: Elaborado pelo autor (2026).

3.6.2 Flow MQTT no Node-RED

Nesta etapa foi desenvolvido um *flow* no Node-RED (Figura 43) para supervisão e comando do sistema via protocolo MQTT, utilizando o broker executado localmente na Raspberry Pi. A estratégia adotada consistiu em separar os tópicos de publicação de variáveis de processo dos tópicos de comando, permitindo que o Node-RED atuasse simultaneamente como assinante para monitoramento e como publicador para controle.

O estado do motor foi supervisionado por meio da assinatura do tópico `tcc/motor/st_motor_on`, cujo valor recebido é encaminhado para um elemento de texto na interface, permitindo a visualização imediata da condição operacional do acionamento. Para o comando do motor, foi utilizado um componente do tipo *switch* na dashboard, configurado para publicar valores booleanos no tópico `tcc/motor/cmd_run/set`, o que possibilita o acionamento remoto do sistema a partir da interface, replicando o comportamento de um comando de partida e parada. Dessa forma, o canal de comando permanece desacoplado do canal de status, garantindo consistência de leitura mesmo quando o acionamento ocorre por outras fontes.

Em paralelo ao controle do motor, foram configurados fluxos de leitura para as variáveis sensoriadas. A temperatura medida pelo sensor DS18B20 foi recebida via assinatura do tópico `tcc/sensors/ds18b20_t` e direcionada a um *gauge* identificado como “Temperatura 1-Wire”. A temperatura e a umidade relativas do sensor AHT10 foram recebidas, respectivamente, pelos tópicos `tcc/sensors/aht10_t` e `tcc/sensors/aht10_rh`, sendo também encaminhadas para indicadores do tipo *gauge* na dashboard. Por fim, o valor da velocidade calculada foi monitorado pela assinatura do tópico `tcc/inputs/rpm`, com exibição em um medidor dedicado. Em conjunto, essa configuração permitiu consolidar, em um único painel, o monitoramento contínuo de variáveis do processo e a atuação remota sobre o comando do motor, utilizando a infraestrutura leve e baseada em tópicos do MQTT.



Fonte: Elaborado pelo autor (2026).

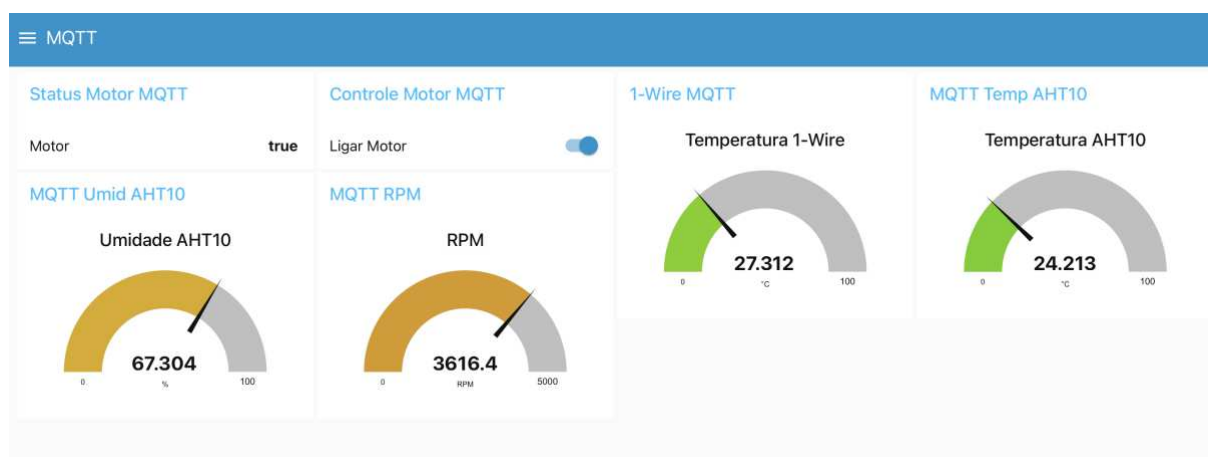
3.6.3 Dashboard MQTT

Aqui foi elaborada a dashboard MQTT (Figura 44) no Node-RED com o objetivo de consolidar, em uma única interface, as funções de controle e monitoramento das variáveis publicadas via broker. A interface foi organizada em blocos funcionais, distinguindo claramente os elementos de supervisão do estado do motor, o envio de comando de acionamento e a visualização das grandezas medidas pelos sensores.

O estado do motor é apresentado em um campo textual denominado “Status Motor MQTT”, alimentado diretamente pela assinatura do tópico de status, permitindo identificar, em tempo real, a condição lógica do acionamento. O controle do motor é realizado por meio de um componente do tipo chave, denominado “Controle Motor MQTT”, que publica valores booleanos no tópico de comando do sistema, viabilizando o acionamento remoto sem a necessidade de acesso direto ao código do servidor.

As variáveis de processo são exibidas por meio de instrumentos do tipo *gauge*, com escalas definidas conforme a natureza de cada grandeza. A temperatura do DS18B20 é apresentada no bloco “1-Wire MQTT”, enquanto a temperatura do AHT10 é exibida em “MQTT Temp AHT10” e a umidade relativa do AHT10 em “MQTT Umid AHT10”. Além disso, o valor de RPM calculado a partir dos pulsos do encoder é mostrado em um medidor dedicado, permitindo acompanhar a variação da velocidade de rotação do motor. Essa estrutura de dashboard garantiu a demonstração prática do MQTT como protocolo orientado a publicação e assinatura, integrando controle e supervisão de forma modular e compatível com aplicações de automação e IoT.

Figura 44: Dashboard Node-RED MQTT



Fonte: Elaborado pelo autor (2026).

3.6.4 MQTT–HTTP para supervisão no ScadaBR

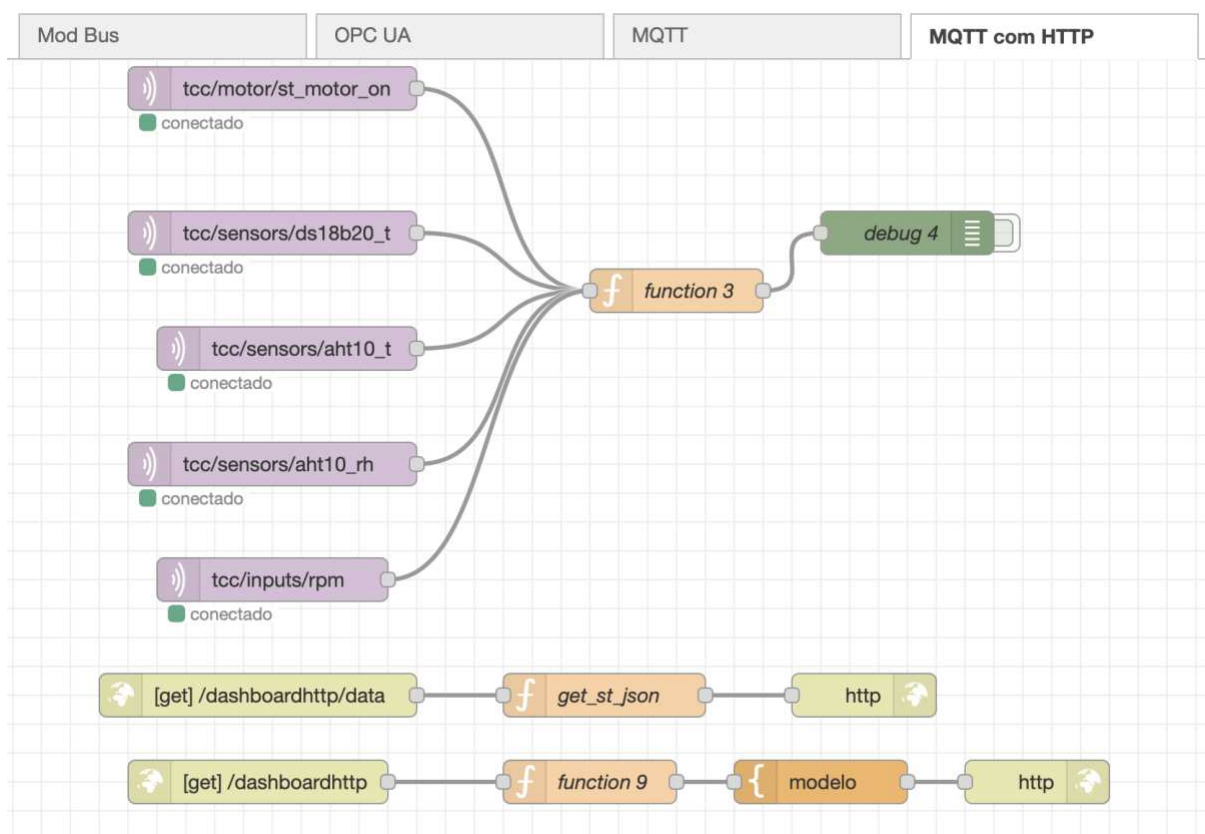
Nesta etapa foi desenvolvida uma página supervisória em HTML no Node-RED (Figura 46) utilizando dados recebidos via MQTT. A Raspberry Pi publica continuamente as variáveis do processo nos tópicos do projeto, como o estado do motor, temperatura do DS18B20, temperatura e umidade do AHT10 e o valor de RPM. No Node-RED, como apresentado na Figura 45 cada tópico é recebido por um nó MQTT-in e encaminhado para um nó Function central, responsável por atualizar um objeto de estado único no contexto do fluxo. Esse objeto consolida o último valor válido de cada variável e registra também um carimbo de tempo da última atualização, garantindo consistência mesmo quando as mensagens MQTT chegam de forma assíncrona.

A partir desse estado consolidado, foram criados dois endpoints HTTP. O primeiro, em GET /dashboardhttp, entrega a página HTML renderizada, atuando como a interface de visualização. O segundo, em GET /dashboardhttp/data, retorna o estado em formato JSON, contendo todas as variáveis em uma única resposta. Dessa forma, a página HTML não depende de múltiplas requisições por variável; em vez disso, consulta periodicamente um único endpoint e recebe todos os dados necessários para atualizar a tela.

A atualização automática da página foi implementada por meio de um script em JavaScript embutido no HTML, que executa requisições periódicas ao endpoint /dashboardhttp/data em intervalos de aproximadamente 1 segundo. A cada ciclo, o navegador recebe o JSON com os valores atuais e atualiza dinamicamente os campos numéricos e os elementos gráficos da interface, como barras de progresso e indicadores de estado. Com essa abordagem, a supervisão via navegador passa a operar em um modelo de atualização contínua, sem necessidade de recarregar

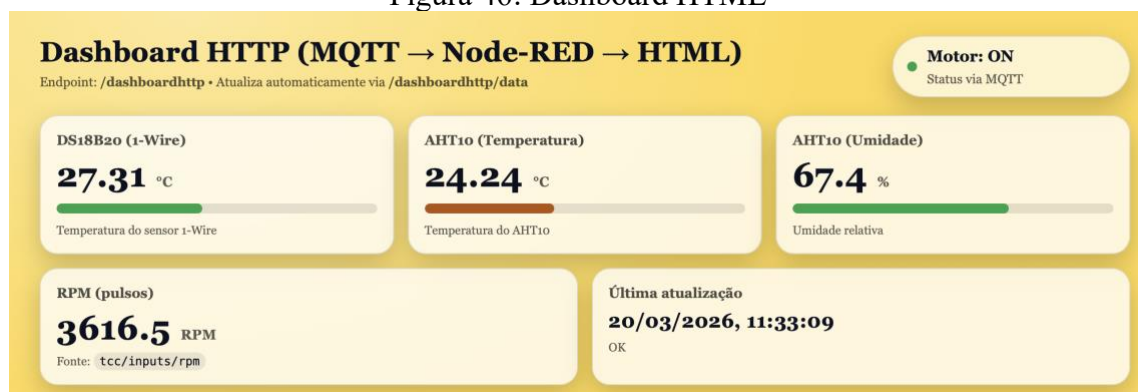
manualmente a página, mantendo a visualização coerente com os dados publicados via MQTT e processados pelo Node-RED.

Figura 45: Flow Node-RED MQTT-HTTP



Fonte: Elaborado pelo autor (2026).

Figura 46: Dashboard HTML



Fonte: Elaborado pelo autor (2026).

3.6.5 Integração MQTT–HTTP para supervisão no ScadaBR

Neste tópico foi implementada uma integração intermediária entre o protocolo MQTT e o sistema supervisor ScadaBR, utilizando o Node-RED como camada de agregação e disponibilização de dados via HTTP. A Raspberry Pi executou o serviço de aquisição e publicação das variáveis em tópicos MQTT, enquanto o Node-RED operou como cliente MQTT em modo subscriber, recebendo continuamente o estado do motor e as variáveis de processo. Essa abordagem permitiu consolidar dados provenientes de múltiplos tópicos em uma única estrutura de estado, facilitando o consumo posterior por sistemas que não se conectam nativamente ao broker MQTT.

No Node-RED, cada variável publicada via MQTT foi recebida por nós de assinatura e encaminhada a um nó de função responsável por atualizar o contexto do fluxo. Nesse contexto, foi mantido um objeto de estado contendo as principais variáveis do sistema, incluindo o estado do motor, temperatura do sensor DS18B20, temperatura e umidade do sensor AHT10, valor de RPM e um carimbo de tempo associado à última atualização. Dessa forma, mesmo com atualizações assíncronas inerentes ao MQTT, o Node-RED passou a disponibilizar o último valor válido de cada grandeza de forma centralizada e consistente.

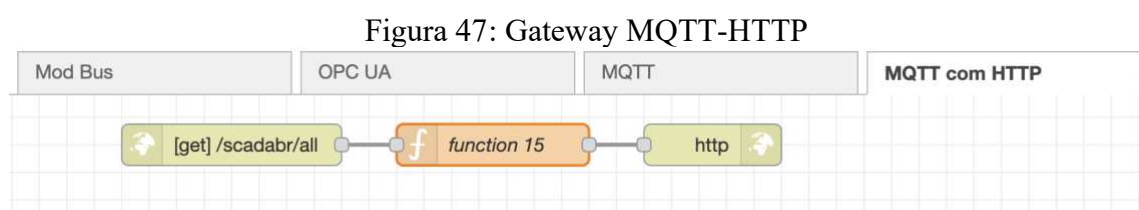
Com o estado consolidado, foi disponibilizado um endpoint HTTP no Node-RED, configurado para responder a requisições do tipo GET retornando os dados em formato JSON. Assim, o ScadaBR passou a obter as variáveis por meio de requisições periódicas ao endpoint, caracterizando um modelo de integração do tipo pull. Esse arranjo permitiu que o supervisor consumisse informações originalmente publicadas via MQTT sem depender de conexão direta com o broker, ampliando a interoperabilidade do sistema.

3.6.6 Gateway MQTT–HTTP

Assim foi implementado um gateway MQTT–HTTP no Node-RED (Figura 47) para permitir que os dados publicados via MQTT fossem disponibilizados por meio de uma interface HTTP compatível com sistemas que operam por requisição, como o ScadaBR. Os valores de processo recebidos nos tópicos MQTT foram previamente consolidados no contexto do fluxo,

mantendo o último valor válido de cada variável. Em seguida, foi criado um endpoint do tipo HTTP In configurado como GET /scadabr/all, responsável por atender solicitações externas.

O nó Function associado a esse endpoint realiza a leitura do estado consolidado armazenado no Node-RED e prepara a resposta no formato adequado, tipicamente um objeto JSON contendo, em uma única estrutura, o estado do motor, as temperaturas, a umidade e o RPM. Por fim, o nó HTTP Response devolve essa informação ao cliente que realizou a requisição. Dessa forma, os dados originalmente distribuídos em múltiplos tópicos MQTT passam a ser consumidos via HTTP em um único ponto de acesso, caracterizando uma integração do tipo pull, na qual o ScadaBR pode consultar periodicamente a URL e atualizar suas variáveis internas sem necessidade de conexão direta ao broker MQTT.

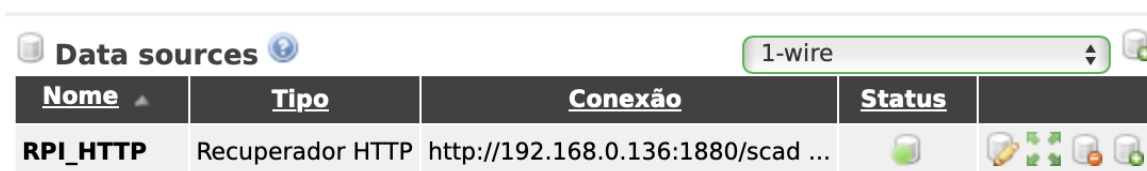


Fonte: Elaborado pelo autor (2026).

3.6.7 Protocolo HTTP no ScadaBR

Na Figura 48 é apresentada a tela de *Data sources* do ScadaBR, na qual foi configurada a fonte de dados RPI_HTTP, do tipo Recuperador HTTP. Observa-se que a comunicação se encontra ativa, indicada pelo ícone de status em verde, evidenciando que o supervisório está conseguindo acessar periodicamente o endpoint HTTP disponibilizado na Raspberry Pi. Essa fonte foi utilizada para integrar ao ScadaBR as variáveis agregadas no Node-RED a partir de publicações MQTT, permitindo que medições e estados do sistema fossem consumidos pelo supervisório sem conexão direta ao *broker* MQTT.

Figura 48: Fonte de dados RPI_HTTP configurada no ScadaBR utilizando o recurso Recuperador HTTP.



Nome	Tipo	Conexão	Status	
RPI_HTTP	Recuperador HTTP	http://192.168.0.136:1880/scad ...		

Fonte: Elaborado pelo autor (2026).

A Figura 49 apresenta a configuração da fonte de dados RPI_HTTP no ScadaBR, implementada por meio do recurso Recuperador HTTP. Foi definida a URL `http://192.168.0.136:1880/scadabr/all`, correspondente ao endpoint disponibilizado no Node-RED para fornecimento dos dados consolidados a partir do MQTT. O período de atualização foi configurado para 1 segundo, caracterizando um mecanismo de aquisição do tipo *pull*, no qual o supervisor realiza requisições periódicas e atualiza seus valores a partir da resposta HTTP. Foram também definidos parâmetros de timeout e retentativas, visando aumentar a robustez da coleta em caso de falhas momentâneas de comunicação.

Na mesma tela, observa-se a criação dos *data points* vinculados à fonte HTTP, cada um configurado com o tipo de dado adequado e associado diretamente ao respectivo campo do JSON retornado pelo Node-RED. Assim, variáveis como `ds18b20_t`, `aht10_t`, `aht10_rh`, `rpm` e `motor_on` passam a ser atualizadas no ScadaBR de forma automática e periódica, permitindo sua utilização em *watch list*, registro histórico e representações gráficas.

Figura 49: Configuração do Recuperador HTTP (RPI_HTTP) no ScadaBR com consulta periódica ao endpoint do Node-RED

Propriedades do Recuperador HTTP

Nome

Export ID (XID)

Período de atualização segundo(s)

URL

Timeout (segundos)

Retentativas

Níveis de alarme de eventos

Falha em recuperar dados Urgente

Exceção em converter data point Urgente

Data points

Nome	Tipo de dado	Status	RegEx
AHT10_RH	Numérico		"aht10_rh"\s*:\s*([-]?[d+](?:\.\d+)?)
AHT10_T	Numérico		"aht10_t"\s*:\s*([-]?[d+](?:\.\d+)?)
DS18B20_T	Numérico		"ds18b20_t"\s*:\s*([-]?[d+](?:\.\d+)?)
MOTOR_ON	Binário		"motor_on"\s*:\s*(true false)
RPM	Numérico		"rpm"\s*:\s*([-]?[d+](?:\.\d+)?)

Fonte: Elaborado pelo autor (2026).

A Figura 50 apresenta a watch list do ScadaBR para a fonte de dados RPI_HTTP, evidenciando a atualização periódica das variáveis obtidas via Recuperador HTTP. Observa-se que os pontos MOTOR_ON, DS18B20_T, AHT10_T, AHT10_RH e RPM são exibidos com seus respectivos valores instantâneos e com o horário da última leitura, confirmando que o supervisório está recebendo corretamente o conteúdo do endpoint HTTP disponibilizado pelo Node-RED. Dessa forma, os dados originalmente publicados em MQTT passam a ser visualizados no ScadaBR por meio de um canal HTTP, mantendo a coerência com as leituras apresentadas nas demais interfaces do projeto e viabilizando a supervisão centralizada das grandezas em tempo quase real.

Figura 50: Watch list ScadaBR

Watch list			
RPI_HTTP - MOTOR_ON	1	00:32:51	
RPI_HTTP - DS18B20_T	26.937	00:32:51	
RPI_HTTP - AHT10_T	26.196	00:32:51	
RPI_HTTP - AHT10_RH	72.026	00:32:51	
RPI_HTTP - RPM	3615.2	00:32:51	

Fonte: Elaborado pelo autor (2026).

A Figura 51 apresenta a representação gráfica no ScadaBR construída com um componente HTML, utilizada como painel de leitura dos data points associados ao Data Source RPI_HTTP. Nesse arranjo, o ScadaBR continua responsável pela aquisição e atualização das variáveis, enquanto o HTML atua como camada de apresentação, organizando os valores em cartões com rótulos e unidades. O painel exibe o estado do motor e as grandezas de processo temperatura do DS18B20, temperatura do AHT10, umidade relativa do AHT10 e RPM, refletindo diretamente os valores armazenados nos respectivos data points e atualizados conforme o período de consulta configurado no Recuperador HTTP. Dessa forma, a interface combina a infraestrutura supervisória do ScadaBR para comunicação e histórico com uma visualização customizada, permitindo uma leitura mais intuitiva e adequada ao contexto didático do experimento.

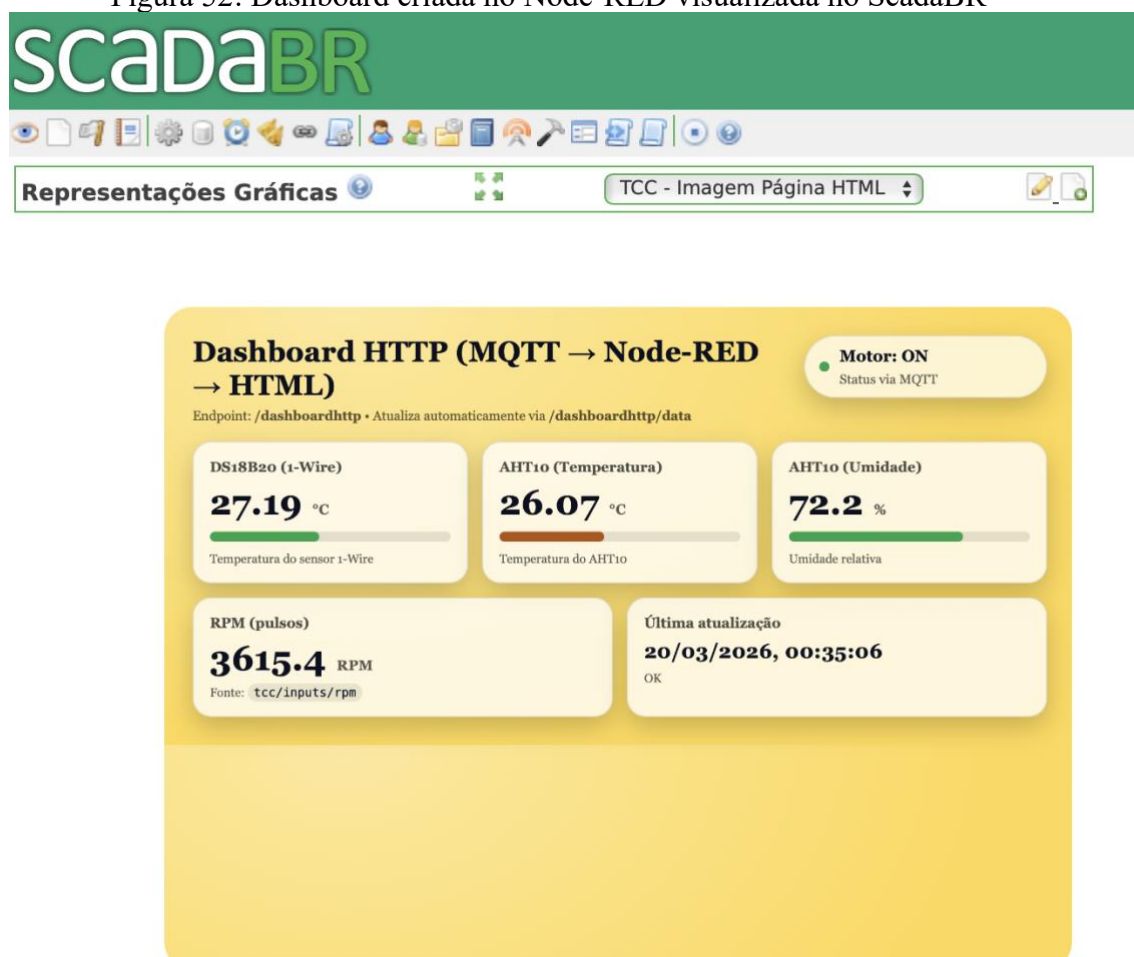
Figura 51: Dashboard utilizando os dados do recuperador HTML



Fonte: Elaborado pelo autor (2026).

A Figura 52 apresenta a página web de supervisão desenvolvida no Node-RED e visualizada no ScadaBR por meio de um componente de exibição de conteúdo HTML. Nessa configuração, o Node-RED disponibiliza a interface no endpoint `http://192.168.0.136:1880/dashboardhttp`, reunindo em uma única tela o estado do motor e as variáveis de processo publicadas via MQTT, como temperatura do DS18B20, temperatura e umidade do AHT10 e RPM. Ao incorporar essa página no ScadaBR, o supervisório passa a centralizar, em um mesmo ambiente, tanto as telas nativas de monitoramento quanto a visualização externa gerada no Node-RED, facilitando a demonstração do conceito de integração multiprotocolo e de composição de interfaces de supervisão a partir de diferentes camadas do sistema.

Figura 52: Dashboard criada no Node-RED visualizada no ScadaBR



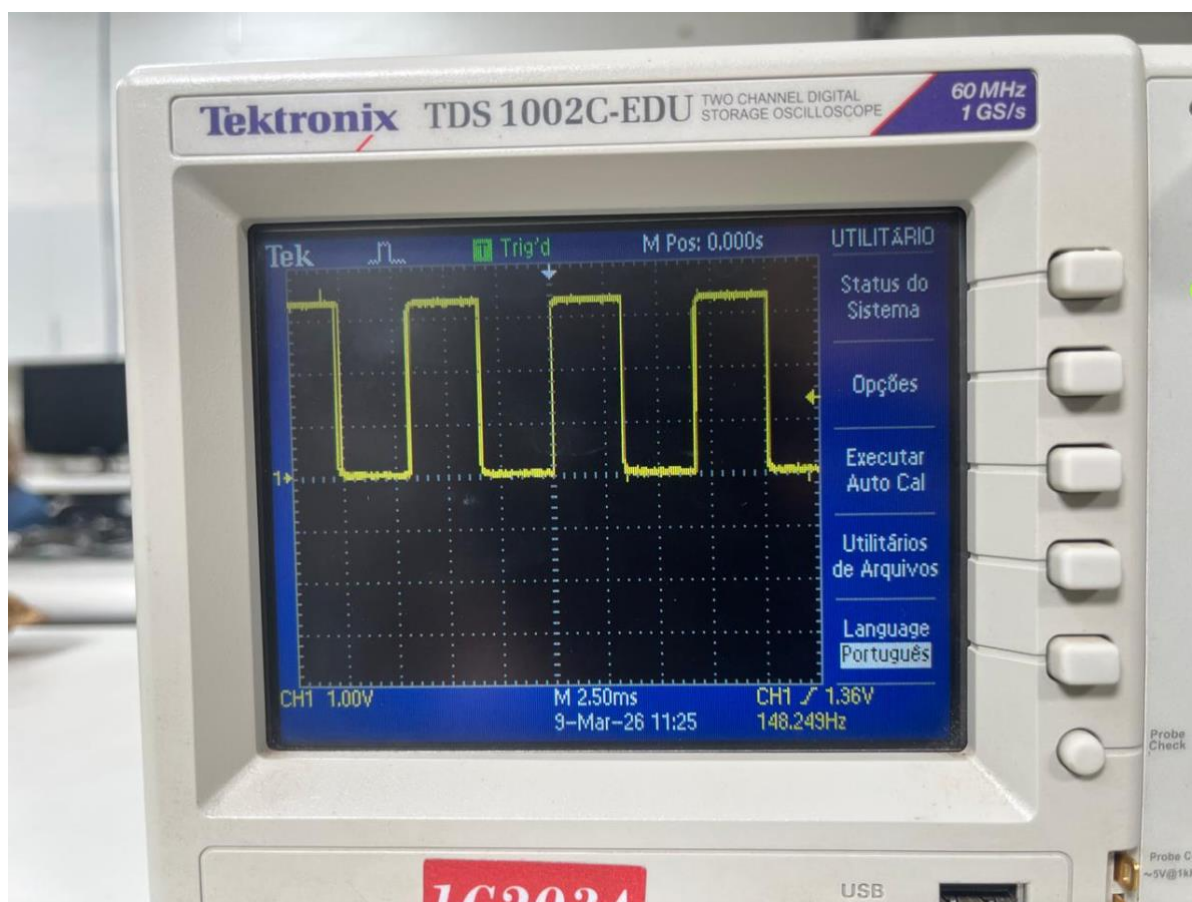
Fonte: Elaborado pelo autor (2026).

4. Resultados e Discussões

4.1 Validação da implementação do encoder incremental

A implantação do encoder incremental no sistema foi precedida por uma etapa de verificação do sinal de saída, com o objetivo de confirmar o funcionamento adequado do dispositivo antes de sua integração definitiva à Raspberry Pi. Para isso, o canal de saída do encoder foi analisado experimentalmente, verificando-se a presença de pulsos regulares e compatíveis com o comportamento esperado para a aplicação. Conforme apresentado na Figura 53, o sinal observado apresentou forma de onda compatível com pulsos digitais, sendo medida uma frequência de aproximadamente 148,249 Hz.

Figura 53: Saída do encoder vista no osciloscópio



Fonte: Elaborado pelo autor (2026).

A frequência f medida no osciloscópio corresponde à taxa de pulsos gerados na saída do encoder. Assim, a rotação do encoder pode ser obtida por:

$$n_{enc} = \frac{60 * f}{PPR}$$

Em que n_{enc} é a rotação do encoder (rpm), f é a frequência (Hz) e PPR é a resolução do encoder (pulsos por rotação). Considerando $f=148,249$ Hz e $PPR=10$, obtém-se:

$$n_{enc} = \frac{60 * 148.249}{10} = 889.49 \text{ rpm}$$

No sistema desenvolvido, o encoder foi acoplado indiretamente ao eixo do motor por meio de uma roda de diâmetro $d_{roda} = 6,5$ cm, em contato com o eixo do motor de diâmetro $d_{eixo} = 1,6$ cm. Desse modo, a relação geométrica entre as rotações é dada por:

$$\frac{n_{enc}}{n_{mot}} = \frac{d_{eixo}}{d_{roda}} = \frac{1.6}{6.5} = 0.2462$$

Logo, a rotação do motor pode ser estimada por:

$$n_{mot} = \frac{n_{enc}}{0.2462} = n_{enc} * 4.0625$$

Portanto:

$$n_{mot} = 889.49 * 4.0625 = 3613.5 \text{ rpm}$$

Após essa verificação inicial, o encoder foi integrado ao sistema por meio do servidor Modbus executado na Raspberry Pi. Conforme mostrado na Figura 54, o servidor registra a contagem de pulsos por janela de tempo e a conversão correspondente para RPM do motor, disponibilizando o valor no registrador Holding associado à variável de rotação. Observa-se que os valores calculados permaneceram na faixa de aproximadamente 3616 rpm, em concordância com a estimativa obtida a partir da frequência medida no osciloscópio e da relação geométrica de acoplamento.

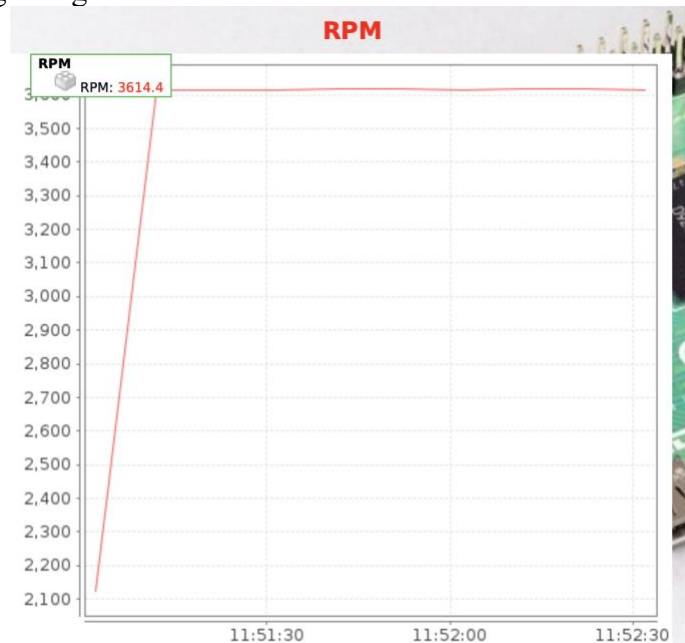
Figura 54: Saída do terminal do servidor Modbus com leitura de pulsos e cálculo de RPM

```
(.venv) pi@raspjoao:~/tcc $ python server_modbus.py
[W1] DS18B20: /sys/bus/w1/devices/28-3ce1d44312df/w1_slave
[I2C] AHT10 OK
[MODBUS] Server ON em 0.0.0.0:1502
[MAPA] Coil0=CMD_RUN | Coil1=ST_MOTOR_ON | HR0=DS18B20_Tx10 | HR1=AHT10_Tx10 | HR2=AHT10_RHx10 | HR3=RPM_MOTOR_x10
[RPM] janela=10s | PPR_encoder=10 | motor/encoder=4.0625
[MOTOR] ON (GPIO17=HIGH)
[RPM] Pulsos=885 em 10.01s -> Motor=2156.10 rpm -> HR3=21561
[RPM] Pulsos=1485 em 10.01s -> Motor=3617.60 rpm -> HR3=36176
[RPM] Pulsos=1487 em 10.02s -> Motor=3617.69 rpm -> HR3=36177
[RPM] Pulsos=1497 em 10.09s -> Motor=3617.40 rpm -> HR3=36174
[RPM] Pulsos=1490 em 10.03s -> Motor=3619.48 rpm -> HR3=36195
[RPM] Pulsos=1484 em 10.00s -> Motor=3616.96 rpm -> HR3=36170
[RPM] Pulsos=1490 em 10.04s -> Motor=3616.79 rpm -> HR3=36168
[RPM] Pulsos=1492 em 10.05s -> Motor=3617.26 rpm -> HR3=36173
[RPM] Pulsos=1491 em 10.04s -> Motor=3619.50 rpm -> HR3=36195
[RPM] Pulsos=1491 em 10.05s -> Motor=3617.72 rpm -> HR3=36177
[RPM] Pulsos=1487 em 10.02s -> Motor=3618.52 rpm -> HR3=36185
[RPM] Pulsos=1495 em 10.08s -> Motor=3615.72 rpm -> HR3=36157
```

Fonte: Elaborado pelo autor (2026).

Adicionalmente, foi realizado um experimento de observação contínua do RPM durante aproximadamente 2 minutos, com aquisição dos dados via Modbus e supervisão no ScadaBR. Conforme apresentado na Figura 55, a rotação medida manteve-se estável ao longo do intervalo, evidenciando consistência na aquisição dos pulsos, no processamento do RPM e na disponibilização da variável para supervisão.

Figura 55: Registro gráfico do RPM no ScadaBR durante o intervalo de observação



Fonte: Elaborado pelo autor (2026).

De forma geral, os resultados obtidos indicam coerência entre a frequência medida diretamente no osciloscópio, o cálculo teórico do RPM do motor a partir da relação geométrica de acoplamento e o valor supervisionado nas interfaces do sistema. Assim, a implementação da medição de rotação por encoder incremental mostrou-se satisfatória para a finalidade didática do

projeto, permitindo a visualização em tempo real do RPM e sua integração com as camadas de comunicação e supervisão definidas na arquitetura.

4.2 Validação da comunicação multiprotocolo

Além dos resultados associados à medição de rotação, foi verificado o funcionamento da comunicação entre os diferentes protocolos implementados na arquitetura proposta. Durante os ensaios, constatou-se que as variáveis de processo foram adquiridas, processadas e disponibilizadas de forma consistente pelas diferentes camadas do sistema, evidenciando a integração entre sensores, unidade de controle e interfaces de supervisão.

No protocolo Modbus TCP, observou-se que o mapeamento das variáveis em *coils* e *holding registers* viabilizou operações de leitura e escrita de maneira estável, permitindo tanto o acionamento do motor quanto a supervisão das grandezas medidas. De forma análoga, o OPC UA apresentou desempenho adequado, possibilitando a organização estruturada das variáveis no servidor e sua leitura por ferramentas de integração. Quanto ao MQTT, verificou-se a publicação eficiente das variáveis em tópicos específicos, caracterizando o modelo *publish/subscribe* e permitindo a distribuição dos dados aos clientes inscritos. Por fim, a comunicação via HTTP também foi validada, possibilitando a disponibilização de dados para visualização e integração com a camada supervisória.

De modo geral, os ensaios realizados indicaram êxito na comunicação entre os protocolos empregados. As trocas de dados ocorreram em conformidade com a arquitetura definida, sem comprometer a leitura das variáveis nem a lógica de acionamento. Esse resultado é relevante para a proposta didática do trabalho, pois demonstra, em uma única bancada experimental, a coexistência e a interoperabilidade entre diferentes protocolos de automação industrial e de aplicações de Internet das Coisas.

Assim, em conjunto com a implantação do encoder incremental, a validação multiprotocolo evidencia que o sistema foi capaz de adquirir sinais de campo, processá-los na Raspberry Pi e disponibilizá-los adequadamente por múltiplos meios de comunicação, reforçando a viabilidade da solução como recurso experimental para o ensino de automação, aquisição de sinais, redes industriais e supervisão.

5. Conclusão e Trabalhos Futuros

Conclui-se que foi possível implementar e validar uma plataforma didática de automação industrial baseada em Raspberry Pi 3, com integração multiprotocolo e recursos de monitoramento, atendendo ao objetivo de demonstrar, em um mesmo ambiente experimental, diferentes paradigmas de comunicação e supervisão empregados na automação industrial e em aplicações de IoT/IIoT. A solução desenvolvida contemplou o acionamento do motor de indução e a aquisição de variáveis de processo a partir de sensores digitais, incluindo temperatura via 1-Wire, temperatura e umidade via I²C e velocidade de rotação por encoder incremental, viabilizando a observação prática do ciclo de aquisição, processamento, disponibilização e visualização dos dados.

Ressalta-se, ainda, o sucesso na implantação do encoder incremental para medição de rotação. A verificação do sinal de saída em osciloscópio indicou pulsos regulares e compatíveis com o comportamento esperado, e a conversão dos pulsos em RPM apresentou valores coerentes com a estimativa teórica baseada na frequência medida e na relação geométrica do acoplamento indireto. Além disso, a aquisição contínua do RPM ao longo do intervalo de observação evidenciou estabilidade do valor supervisionado, confirmando a consistência da leitura, do processamento e da disponibilização dessa variável para as camadas de comunicação e supervisão do sistema.

No âmbito das comunicações, o Modbus TCP operou a partir de um mapeamento explícito das variáveis em *coils* e *holding registers*, permitindo leitura e escrita por endereçamento direto e assegurando compatibilidade com ferramentas supervisórias. Em paralelo, o OPC UA evidenciou um modelo mais estruturado, com organização hierárquica das variáveis em objetos e nós no servidor, acessados por NodeId, possibilitando supervisão e controle por meio do Node-RED como cliente OPC UA. Adicionalmente, o MQTT foi empregado na publicação das variáveis em tópicos, caracterizando o modelo *publish/subscribe* e favorecendo a integração com aplicações distribuídas.

Como contribuição de integração, o Node-RED atuou como camada intermediária de agregação e supervisão, consolidando dados recebidos via MQTT e disponibilizando-os por HTTP. Esse arranjo permitiu a integração com o ScadaBR por meio do Recuperador HTTP, evidenciando a interoperabilidade entre protocolos e a possibilidade de integração com sistemas que não possuem suporte nativo ao broker MQTT, mantendo simultaneamente diferentes caminhos de supervisão no projeto e fortalecendo o caráter comparativo da bancada.

Por fim, os resultados indicam que a plataforma apresenta caráter modular e replicável em contexto acadêmico, podendo servir como base para atividades práticas em redes industriais, sistemas embarcados e supervisão. Como trabalhos futuros, recomenda-se a implementação de mecanismos de segurança e autenticação nos serviços de comunicação, bem como a integração com serviços em nuvem para disponibilização e monitoramento de dados em tempo real, além da ampliação do conjunto com novos dispositivos e cenários de experimentação, de modo a aumentar o realismo e o valor pedagógico da solução.

Referências

ACATECH – National Academy of Science and Engineering. *Recommendations for implementing the strategic initiative INDUSTRIE 4.0 – Final report of the Industrie 4.0 Working Group*. Munich, 2013. Disponível em: <https://en.acatech.de/publication/recommendations-for-implementing-the-strategicinitiative-industrie-4-0-final-report-of-the-industrie-4-0-working-group/>. Acesso em: 26 out. 2025.

ADAFRUIT. *Introduction — Adafruit AHTx0 Library*. 2025. Disponível em: <https://docs.circuitpython.org/projects/ahtx0/>. Acesso em: 6 mar. 2026.

AL-YAMAN, M.; ALSWAITI, D.; ALSHARKAWI, A.; AL-TAEE, M. A cost-effective modular laboratory solution for industrial automation and applied engineering education. *MethodsX*, v. 14, p. 103388, 2025. Disponível em: <https://www.sciencedirect.com/science/article/pii/S2215016125002341>. Acesso em: 5 mar. 2026.

ANALOG DEVICES. *Guidelines for Reliable Long Line 1-Wire Networks*. 16 nov. 2001. Disponível em: <https://www.analog.com/en/resources/technical-articles/guidelines-for-reliable-long-line-1wire-networks.html>. Acesso em: 3 mar. 2026.

ASAIR. *AHT10: Temperature and Humidity Sensor* (manual técnico). 2018. Disponível em: <https://altronics.cl/uploads/AHT10.pdf>. Acesso em: 6 mar. 2026.

FERRELL, W. R.; SHERIDAN, T. B. Supervisory Control of Remote Manipulation. *IEEE Spectrum*, v. 4, n. 10, p. 81–88, out. 1967. Disponível em: <https://ntrs.nasa.gov/api/citations/19690006779/downloads/19690006779.pdf>. Acesso em: 26 out. 2025.

IETF – Internet Engineering Task Force. *RFC 9110 – HTTP Semantics*. 2022. Disponível em: <https://www.rfc-editor.org/rfc/rfc9110>. Acesso em: 25 out. 2025.

KERNEL.ORG. *Kernel driver aht10* (documentação técnica). 2024. Disponível em: <https://www.kernel.org/doc/html/v6.1/hwmon/aht10.html>. Acesso em: 6 mar. 2026.

LEMAQS AUTOMAÇÃO. *Encoder incremental rotativo Omron E6B2-CWZ5B (produto 9875781)*. 2026. Disponível em: <https://www.lemasautomacao.com.br/encoderincrementalrotativo600pulsos6b2-cwz5b/prod-9875781/>. Acesso em: 19 mar. 2026.

MAKERHERO. *Sensor de temperatura DS18B20 à prova d'água*. 2026. Disponível em: <https://www.makerhero.com/produto/sensor-de-temperatura-ds18b20-a-prova-dagua/>. Acesso em: 19 mar. 2026.

MODBUS ORGANIZATION. *Modbus Application Protocol Specification, V1.1b3*. 2012. Disponível em: https://www.modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf. Acesso em: 25 out. 2025.

MODBUS ORGANIZATION. *Modbus Specifications and Implementation Guides*. 2024. Disponível em: <https://www.modbus.org/modbus-specifications>. Acesso em: 25 out. 2025.

MQTT PROJECT. *MQTT – The Standard for IoT Messaging*. 2025. Disponível em: <https://mqtt.org/>. Acesso em: 25 out. 2025.

NIST – National Institute of Standards and Technology. *SP 800-82 Rev. 3: Guide to Operational Technology (OT) Security*. Gaithersburg, 2023. Disponível em: <https://csrc.nist.gov/pubs/sp/800/82/r3/final>. Acesso em: 26 out. 2025.

OASIS. *MQTT Version 3.1.1 – OASIS Standard*. 2014. Disponível em: <https://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf>. Acesso em: 25 out. 2025.

OASIS. *MQTT Version 5.0 – OASIS Standard*. 2019. Disponível em: <https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf>. Acesso em: 25 out. 2025.

OMRON. *E6B2-C incremental 40-mm-dia. rotary encoder: lineup and specifications*. [s.l.]: Omron, 2023. Disponível em: <https://www.ia.omron.com/products/family/487/specification.html>. Acesso em: 5 mar. 2026.

OPC FOUNDATION. *OPC UA Part 1: Overview and Concepts (Version 1.04)*. 2021. Disponível em: <https://reference.opcfoundation.org/Core/Part1/v104/docs/>. Acesso em: 25 out. 2025.

OPC FOUNDATION. *OPC UA Part 1: Overview and Concepts (Version 1.05)*. 2024. Disponível em: <https://reference.opcfoundation.org/Core/Part1/v105/docs/>. Acesso em: 25 out. 2025.

OPENJS FOUNDATION; IBM. *Node-RED – Getting started: Running on Raspberry Pi*. 2025. Disponível em: <https://nodered.org/docs/getting-started/raspberrypi>. Acesso em: 25 out. 2025.

PAJPACH, M.; HAFFNER, O.; KUČERA, E.; DRAHOŠ, P. Low-Cost Education Kit for Teaching Basic Skills for Industry 4.0 Using Deep-Learning in Quality Control Tasks. *Electronics*, v. 11, n. 2, p. 230, 2022. Disponível em: <https://www.mdpi.com/2079-9292/11/2/230>. Acesso em: 5 mar. 2026.

PRIMTEC. *Kit suporte metálico roldana para encoder incremental com amortecimento de impacto*. 2026. Disponível em: <https://www.primtec.com.br/kit-suporte-metalico-roldana-para-encoder-incremental-com-amortecimento-de-impacto>. Acesso em: 19 mar. 2026.

RASPBERRY PI FOUNDATION. *Raspberry Pi OS – Documentation*. 2023. Disponível em: <https://www.raspberrypi.com/documentation/computers/os.html>. Acesso em: 25 out. 2025.

RASPBERRY PI LTD. *Raspberry Pi 3 Model B – Specifications*. 2016. Disponível em: <https://www.raspberrypi.com/products/raspberry-pi-3-model-b/>. Acesso em: 25 out. 2025.

RFC EDITOR. *Information on RFC 9110*. 2022. Disponível em: <https://www.rfc-editor.org/info/rfc9110>. Acesso em: 25 out. 2025.

SAWERS. *Módulo MOSFET AOD4184A PWM*. 2026. Disponível em: <https://sawers.com.pe/modulo-mosfet-aod4184a-pwm/>. Acesso em: 19 mar. 2026.

SCADABR PROJECT. *Site oficial do ScadaBR*. 2025. Disponível em: <https://scadabr.org/>. Acesso em: 25 out. 2025.

SCADABR PROJECT (SourceForge). *Manual ScadaBR (PDF)*. 2016. Disponível em: <https://sourceforge.net/projects/scadabr/files/Software/Docs/Manual%20ScadaBR.pdf/download>. Acesso em: 25 out. 2025.

USINAINFO. *Sensor AHT10 de alta precisão para medir temperatura e umidade*. 2025. Disponível em: <https://www.usinainfo.com.br/sensor-de-temperatura/sensor-aht10-de-alta-precisao-para-medir-temperatura-e-umidade-5691.html>. Acesso em: 19 mar. 2026.

APÊNDICE A – Código Servidor Modbus

```
#!/usr/bin/env python3
import time
import glob
import threading
import RPi.GPIO as GPIO
import lgpio
from pyModbusTCP.server import ModbusServer

# CONFIG GERAL

GPIO_LED = 17
GPIO_BTN = 27
GPIO_PULSE = 26

HOST = "0.0.0.0"
PORT = 1502

W1_BASE = "/sys/bus/w1/devices"

# Encoder / RPM
WINDOW_S = 10.0
PPR_ENCODER = 10.0      # pulsos por volta do encoder (canal usado)
DIAM_MOTOR = 1.6        # cm
DIAM_RODA = 6.5         # cm
MOTOR_PER_ENCODER = DIAM_RODA / DIAM_MOTOR # 4.0625

# Modbus map
CMD_COIL = 0
ST_COIL = 1
TEMP_HR = 0
AHT_T_HR = 1
AHT_RH_HR = 2
RPM_HR = 3

# GPIO / ESTADO

cmd_run = False
st_motor_on = False
```

```
def gpio_setup():
    GPIO.setmode(GPIO.BCM)
    GPIO.setwarnings(False)
    GPIO.setup(GPIO_LED, GPIO.OUT)
    GPIO.output(GPIO_LED, GPIO.LOW)
    GPIO.setup(GPIO_BTN, GPIO.IN, pull_up_down=GPIO.PUD_UP)

def set_motor(on: bool):
    GPIO.output(GPIO_LED, GPIO.HIGH if on else GPIO.LOW)

def apply_state(new_cmd: bool):
    global cmd_run, st_motor_on
    new_cmd = bool(new_cmd)
    if new_cmd == cmd_run:
        return
    cmd_run = new_cmd
    st_motor_on = cmd_run
    set_motor(st_motor_on)
    print(f'[MOTOR] {'ON ' if st_motor_on else 'OFF'} (GPIO{GPIO_LED}={'HIGH' if st_motor_on else 'LOW'})")

# DS18B20 (1-Wire)

def find_ds18b20_path():
    devs = glob.glob(f'{W1_BASE}/28-*/w1_slave')
    return devs[0] if devs else None

def read_ds18b20_c(path):
    try:
        with open(path, "r") as f:
            lines = f.read().strip().splitlines()
            if len(lines) < 2 or "YES" not in lines[0] or "t=" not in lines[1]:
                return None
            t_milli = int(lines[1].split("t=")[1])
            return t_milli / 1000.0
    except Exception:
        return None

# AHT10 (I2C)
```

```
def setup_aht10():
    import board
    import busio
    import adafruit_ahtx0
    i2c = busio.I2C(board.SCL, board.SDA)
    return adafruit_ahtx0.AHTx0(i2c)

# Encoder (lgpio) -> contador

class PulseCounterLgpio:
    def __init__(self, gpiochip: int, pin: int):
        self.gpiochip = gpiochip
        self.pin = pin
        self.h = None
        self.cb = None
        self.lock = threading.Lock()
        self.count = 0

    def _cb(self, chip, gpio, level, tick):
        if level == 0: # queda
            with self.lock:
                self.count += 1

    def start(self):
        self.h = lgpio.gpiochip_open(self.gpiochip)
        lgpio.gpio_claim_alert(self.h, self.pin, lgpio.FALLING_EDGE, lgpio.SET_PULL_UP)
        self.cb = lgpio.callback(self.h, self.pin, lgpio.FALLING_EDGE, self._cb)

    def read_and_reset(self) -> int:
        with self.lock:
            c = self.count
            self.count = 0
        return c

    def stop(self):
        try:
            if self.cb:
                self.cb.cancel()
        except Exception:
            pass
        try:
```

```
        if self.h is not None:
            lgpio.gpiochip_close(self.h)
    except Exception:
        pass

# Modbus helpers

def get_coil(db, addr: int) -> bool:
    vals = db.get_coils(addr, 1)
    return bool(vals and vals[0])

def set_coil(db, addr: int, value: bool):
    db.set_coils(addr, [bool(value)])

# MAIN

def main():
    global cmd_run, st_motor_on

    gpio_setup()

    server = ModbusServer(host=HOST, port=PORT, no_block=True)
    server.start()
    db = server.data_bank

    db.set_coils(0, [False, False])
    db.set_holding_registers(0, [0, 0, 0, 0])

    w1_path = find_ds18b20_path()
    print(f"[W1] DS18B20: {w1_path if w1_path else 'NÃO ENCONTRADO'}")

    try:
        aht = setup_aht10()
        print("[I2C] AHT10 OK")
    except Exception as e:
        aht = None
        print(f"[I2C] AHT10 não inicializou: {e}")

    print(f"[MODBUS] Server ON em {HOST}:{PORT}")
```

```
print("[MAPA] Coil0=CMD_RUN | Coil1=ST_MOTOR_ON | HR0=DS18B20_Tx10 | HR1=AHT10_Tx10 |  
HR2=AHT10_RHx10 | HR3=RPM_MOTOR_x10")  
  
print(f"[RPM] janela={WINDOW_S:.0f}s | PPR_encoder={PPR_ENCODER:g} |  
motor/encoder={MOTOR_PER_ENCODER:.4f}")  
  
counter = PulseCounterLgpio(gpiochip=0, pin=GPIO_PULSE)  
counter.start()  
  
last_btn = 1  
last_toggle_time = 0.0  
  
last_ds_time = 0.0  
last_aht_time = 0.0  
  
win_start = time.time()  
last_rpm_x10 = None  
  
try:  
    while True:  
        now = time.time()  
  
        # 1) comando via Modbus (coil0)  
        cmd_from_modbus = get_coil(db, CMD_COIL)  
  
        # 2) botão local (toggle)  
        btn = GPIO.input(GPIO_BTN)  
        if btn == 0 and last_btn == 1 and (now - last_toggle_time) > 0.25:  
            cmd_from_modbus = not cmd_from_modbus  
            set_coil(db, CMD_COIL, cmd_from_modbus)  
            last_toggle_time = now  
            print(f"[BOTA0] Toggle CMD_RUN -> {cmd_from_modbus}")  
            last_btn = btn  
  
        # 3) aplica estado  
        if cmd_from_modbus != cmd_run:  
            apply_state(cmd_from_modbus)  
  
        # 4) status coil1  
        set_coil(db, ST_COIL, st_motor_on)  
  
        # 5) DS18B20 (1s)  
        if w1_path and (now - last_ds_time) >= 1.0:
```

```
t = read_ds18b20_c(w1_path)
if t is not None:
    db.set_holding_registers(TEMP_HR, [int(round(t * 10))])
    last_ds_time = now

# 6) AHT10 (1s)
if aht and (now - last_aht_time) >= 1.0:
    try:
        t2 = float(aht.temperature)
        rh = float(aht.relative_humidity)
        db.set_holding_registers(AHT_T_HR, [int(round(t2 * 10))])
        db.set_holding_registers(AHT_RH_HR, [int(round(rh * 10))])
    except Exception as e:
        print("[AHT10] erro leitura:", e)
    last_aht_time = now

# 7) RPM por janela (10s)
if (now - win_start) >= WINDOW_S:
    dt = now - win_start
    pulses = counter.read_and_reset()

    enc_rpm = (pulses / dt) * 60.0 / PPR_ENCODER if (dt > 0 and PPR_ENCODER > 0) else 0.0
    motor_rpm = enc_rpm * MOTOR_PER_ENCODER

    rpm_x10 = int(round(motor_rpm * 10))
    db.set_holding_registers(RPM_HR, [rpm_x10])

    if rpm_x10 != last_rpm_x10:
        print(f"[RPM] Pulsos={pulses} em {dt:.2f}s -> Motor={motor_rpm:.2f} rpm -> HR3={rpm_x10}")
        last_rpm_x10 = rpm_x10

    win_start = now

    time.sleep(0.01)

except KeyboardInterrupt:
    print("\n[MODBUS] Parando...")
finally:
    try:
        server.stop()
    except Exception:
        pass
```

```
    try:
        counter.stop()
    except Exception:
        pass

    try:
        GPIO.output(GPIO_LED, GPIO.LOW)
    except Exception:
        pass

    GPIO.cleanup()

if __name__ == "__main__":
    main()
```

APÊNDICE B – Código Servidor OPC UA

```
#!/usr/bin/env python3
import time
import sys
import signal
import glob
import threading

import RPi.GPIO as GPIO
import lgpio
from opcua import Server

# CONFIG GERAL

GPIO_LED = 17
GPIO_BTN = 27
GPIO_PULSE = 26

ENDPOINT = "opc.tcp://0.0.0.0:4840/tcc/"
NAMESPACE_URI = "tcc"

W1_BASE = "/sys/bus/w1/devices"

# Encoder / RPM
WINDOW_S = 10.0
PPR_ENCODER = 10.0      # pulsos por volta do encoder (canal usado)
DIAM_MOTOR = 1.6        # cm
DIAM_RODA = 6.5         # cm
MOTOR_PER_ENCODER = DIAM_RODA / DIAM_MOTOR # 4.0625

# GPIO / ESTADO

cmd_run = False
st_motor_on = False

def gpio_setup():
    GPIO.setwarnings(False)
    GPIO.setmode(GPIO.BCM)
```

```
GPIO.setup(GPIO_LED, GPIO.OUT, initial=GPIO.LOW)
GPIO.setup(GPIO_BTN, GPIO.IN, pull_up_down=GPIO.PUD_UP)

def set_motor(on: bool):
    GPIO.output(GPIO_LED, GPIO.HIGH if on else GPIO.LOW)

def apply_state(new_cmd: bool):
    global cmd_run, st_motor_on
    new_cmd = bool(new_cmd)
    if new_cmd == cmd_run:
        return
    cmd_run = new_cmd
    st_motor_on = cmd_run
    set_motor(st_motor_on)
    print(f'[MOTOR] {\'ON \' if st_motor_on else \'OFF\'} (GPIO{GPIO_LED}={\'HIGH\' if st_motor_on else \'LOW\'})')

# DS18B20 (1-Wire)

def find_ds18b20_path():
    devs = glob.glob(f'{W1_BASE}/28-*/w1_slave')
    return devs[0] if devs else None

def read_ds18b20_c(path):
    try:
        with open(path, "r") as f:
            lines = f.read().strip().splitlines()
            if len(lines) < 2 or "YES" not in lines[0] or "t=" not in lines[1]:
                return None
            t_milli = int(lines[1].split("t=")[1])
            return t_milli / 1000.0
    except Exception:
        return None

# AHT10 (I2C)

def setup_aht10():
    import board
    import busio
    import adafruit_ahtx0
```

```
i2c = busio.I2C(board.SCL, board.SDA)
return adafruit_ahx0.AHTx0(i2c)

# Encoder (lgpio) -> contador

class PulseCounterLgpio:
    def __init__(self, gpiochip: int, pin: int):
        self.gpiochip = gpiochip
        self.pin = pin
        self.h = None
        self.cb = None
        self.lock = threading.Lock()
        self.count = 0

    def _cb(self, chip, gpio, level, tick):
        if level == 0: # queda
            with self.lock:
                self.count += 1

    def start(self):
        self.h = lgpio.gpiochip_open(self.gpiochip)
        lgpio.gpio_claim_alert(self.h, self.pin, lgpio.FALLING_EDGE, lgpio.SET_PULL_UP)
        self.cb = lgpio.callback(self.h, self.pin, lgpio.FALLING_EDGE, self._cb)

    def read_and_reset(self) -> int:
        with self.lock:
            c = self.count
            self.count = 0
        return c

    def stop(self):
        try:
            if self.cb:
                self.cb.cancel()
        except Exception:
            pass
        try:
            if self.h is not None:
                lgpio.gpiochip_close(self.h)
        except Exception:
            pass
```

```
# OPC UA (criação da árvore)

def build_opcua():
    server = Server()
    server.set_endpoint(ENDPOINT)
    idx = server.register_namespace(NAMESPACE_URI)

    objects = server.get_objects_node()
    motor_obj = objects.add_object(idx, "Motor")
    sensors_obj = objects.add_object(idx, "Sensors")
    inputs_obj = objects.add_object(idx, "Inputs")

    var_cmd = motor_obj.add_variable(idx, "CMD_RUN", False)
    var_cmd.set_writable()
    var_st = motor_obj.add_variable(idx, "ST_MOTOR_ON", False)

    var_ds = sensors_obj.add_variable(idx, "DS18B20_T", 0.0)
    var_aht_t = sensors_obj.add_variable(idx, "AHT10_T", 0.0)
    var_aht_rh = sensors_obj.add_variable(idx, "AHT10_RH", 0.0)

    var_rpm_motor = inputs_obj.add_variable(idx, "RPM_MOTOR", 0.0)

    print(f"[OPCUA] Endpoint: {ENDPOINT} ns={idx}")
    print("[OPCUA] Nodes: Motor/CMD_RUN (writable), Motor/ST_MOTOR_ON, Sensors/*, Inputs/RPM_MOTOR")
    return server, var_cmd, var_st, var_ds, var_aht_t, var_aht_rh, var_rpm_motor

def on_exit(signum=None, frame=None):
    try:
        GPIO.output(GPIO_LED, GPIO.LOW)
        GPIO.cleanup()
    except Exception:
        pass
    print("\nSaindo... motor OFF, GPIO limpo.")
    sys.exit(0)

# MAIN

def main():
    global cmd_run, st_motor_on
```

```
signal.signal(signal.SIGINT, on_exit)
signal.signal(signal.SIGTERM, on_exit)

gpio_setup()

# sensores
w1_path = find_ds18b20_path()
print(f"[W1] DS18B20: {w1_path if w1_path else 'NÃO ENCONTRADO'}")

try:
    aht = setup_aht10()
    print("[I2C] AHT10 OK")
except Exception as e:
    aht = None
    print(f"[I2C] AHT10 não inicializou: {e}")

# servidor OPC UA
server, var_cmd, var_st, var_ds, var_aht_t, var_aht_rh, var_rpm_motor = build_opcua()
server.start()
print("[OPCUA] Server STARTED")
print(f"[RPM] janela={WINDOW_S:.0f}s | PPR_encoder={PPR_ENCODER:g} |
motor/encoder={MOTOR_PER_ENCODER:.4f}")

# contador de pulsos (lgpio)
counter = PulseCounterLgpio(gpiochip=0, pin=GPIO_PULSE)
counter.start()

# estado inicial
apply_state(False)
var_cmd.set_value(False)
var_st.set_value(False)

last_btn = 1
last_toggle_time = 0.0

last_ds_time = 0.0
last_aht_time = 0.0

win_start = time.time()

try:
```

```
while True:
    now = time.time()

    # 1) botão local (toggle)
    btn = GPIO.input(GPIO_BTN)
    if btn == 0 and last_btn == 1 and (now - last_toggle_time) > 0.25:
        apply_state(not cmd_run)
        var_cmd.set_value(cmd_run)
        last_toggle_time = now
    last_btn = btn

    # 2) comando remoto via OPC UA
    desired = bool(var_cmd.get_value())
    if desired != cmd_run:
        apply_state(desired)

    # 3) publica status
    var_st.set_value(st_motor_on)

    # 4) DS18B20 (1s)
    if w1_path and (now - last_ds_time) >= 1.0:
        t = read_ds18b20_c(w1_path)
        if t is not None:
            var_ds.set_value(float(t))
        last_ds_time = now

    # 5) AHT10 (1s)
    if aht and (now - last_aht_time) >= 1.0:
        try:
            t2 = float(aht.temperature)
            rh = float(aht.relative_humidity)
            var_aht_t.set_value(t2)
            var_aht_rh.set_value(rh)
        except Exception as e:
            print("[AHT10] erro leitura:", e)
        last_aht_time = now

    # 6) RPM por janela (10s)
    if (now - win_start) >= WINDOW_S:
        dt = now - win_start
        pulses = counter.read_and_reset()
```

```
enc_rpm = (pulses / dt) * 60.0 / PPR_ENCODER if (dt > 0 and PPR_ENCODER > 0) else 0.0
motor_rpm = enc_rpm * MOTOR_PER_ENCODER

var_rpm_motor.set_value(float(motor_rpm))

print(f"[RPM] Pulsos={pulses} em {dt:.2f}s -> Motor={motor_rpm:.2f} rpm")

win_start = now

time.sleep(0.01)

except KeyboardInterrupt:
    print("\n[OPCUA] Parando...")
finally:
    try:
        server.stop()
    except Exception:
        pass
    try:
        counter.stop()
    except Exception:
        pass
    on_exit()

if __name__ == "__main__":
    main()
```

APÊNDICE C – Código Servidor MQTT

```
#!/usr/bin/env python3
import time
import glob
import sys
import signal
import threading
import RPi.GPIO as GPIO
import lgpio
import paho.mqtt.client as mqtt

# CONFIG GERAL

GPIO_LED = 17
GPIO_BTN = 27
GPIO_PULSE = 26

W1_BASE = "/sys/bus/w1/devices"

WINDOW_S = 3.0
PPR_ENCODER = 10.0
DIAM_MOTOR = 1.6
DIAM_RODA = 6.5
MOTOR_PER_ENCODER = DIAM_RODA / DIAM_MOTOR

BROKER_HOST = "127.0.0.1"
BROKER_PORT = 1883

TOPIC_BASE = "tcc/motor"
TOPIC_CMD_SET = f"{TOPIC_BASE}/cmd_run/set"
TOPIC_CMD = f"{TOPIC_BASE}/cmd_run"
TOPIC_ST = f"{TOPIC_BASE}/st_motor_on"

TOPIC_DS18B20 = "tcc/sensors/ds18b20_t"
TOPIC_AHT_T = "tcc/sensors/aht10_t"
TOPIC_AHT_RH = "tcc/sensors/aht10_rh"
TOPIC_RPM = "tcc/inputs/rpm"
```

```
# ESTADO

cmd_run = False
st_motor_on = False

def parse_bool(payload: bytes):
    s = payload.decode(errors="ignore").strip().lower()
    if s in ("1", "true", "on", "ligar", "start", "run"):
        return True
    if s in ("0", "false", "off", "desligar", "stop"):
        return False
    return None

# GPIO

def gpio_setup():
    GPIO.setwarnings(False)
    GPIO.setmode(GPIO.BCM)
    GPIO.setup(GPIO_LED, GPIO.OUT, initial=GPIO.LOW)
    GPIO.setup(GPIO_BTN, GPIO.IN, pull_up_down=GPIO.PUD_UP)

def gpio_cleanup():
    try:
        GPIO.output(GPIO_LED, GPIO.LOW)
    except Exception:
        pass
    GPIO.cleanup()

def set_motor(on: bool):
    GPIO.output(GPIO_LED, GPIO.HIGH if on else GPIO.LOW)

def apply_state(new_cmd: bool):
    global cmd_run, st_motor_on
    new_cmd = bool(new_cmd)
    if new_cmd == cmd_run:
        return
    cmd_run = new_cmd
    st_motor_on = cmd_run
    set_motor(st_motor_on)
    print(f"[MOTOR] {'ON' if st_motor_on else 'OFF'} (GPIO{GPIO_LED}={'HIGH' if st_motor_on else 'LOW'})")
```

```
# DS18B20

def find_ds18b20_path():
    devs = glob.glob(f"{W1_BASE}/28-*/w1_slave")
    return devs[0] if devs else None

def read_ds18b20_c(path):
    try:
        with open(path, "r") as f:
            lines = f.read().strip().splitlines()
            if len(lines) < 2 or "YES" not in lines[0] or "t=" not in lines[1]:
                return None
            t_milli = int(lines[1].split("t=")[1])
            return t_milli / 1000.0
    except Exception:
        return None

# AHT10

def setup_aht10():
    import board
    import busio
    import adafruit_ahtx0
    i2c = busio.I2C(board.SCL, board.SDA)
    return adafruit_ahtx0.AHTx0(i2c)

# Encoder (lgpio)

class PulseCounterLgpio:
    def __init__(self, gpiochip: int, pin: int):
        self.gpiochip = gpiochip
        self.pin = pin
        self.h = None
        self.cb = None
        self.lock = threading.Lock()
        self.count = 0

    def _cb(self, chip, gpio, level, tick):
        if level == 0:
```

```
        with self.lock:
            self.count += 1

def start(self):
    self.h = lgpio.gpiochip_open(self.gpiochip)
    lgpio.gpio_claim_alert(self.h, self.pin, lgpio.FALLING_EDGE, lgpio.SET_PULL_UP)
    self.cb = lgpio.callback(self.h, self.pin, lgpio.FALLING_EDGE, self._cb)

def read_and_reset(self) -> int:
    with self.lock:
        c = self.count
        self.count = 0
    return c

def stop(self):
    try:
        if self.cb:
            self.cb.cancel()
    except Exception:
        pass
    try:
        if self.h is not None:
            lgpio.gpiochip_close(self.h)
    except Exception:
        pass

# MQTT callbacks

def mqtt_publish_states(client: mqtt.Client):
    client.publish(TOPIC_CMD, "true" if cmd_run else "false", retain=True)
    client.publish(TOPIC_ST, "true" if st_motor_on else "false", retain=True)

def on_connect(client, userdata, flags, rc):
    print("[MQTT] connected rc=", rc)
    client.subscribe(TOPIC_CMD_SET)
    mqtt_publish_states(client)

def on_message(client, userdata, msg):
    if msg.topic == TOPIC_CMD_SET:
        v = parse_bool(msg.payload)
        if v is None:
```

```
        print("[MQTT] payload inválido:", msg.payload)
        return
    apply_state(v)
    mqtt_publish_states(client)
    print("[MQTT] CMD_RUN <=", v)

def on_exit(signum=None, frame=None):
    print("\nSaindo...")
    try:
        gpio_cleanup()
    except Exception:
        pass
    sys.exit(0)

# MAIN

def main():
    signal.signal(signal.SIGINT, on_exit)
    signal.signal(signal.SIGTERM, on_exit)

    gpio_setup()

    w1_path = find_ds18b20_path()
    print(f"[W1] DS18B20: {w1_path if w1_path else 'NÃO ENCONTRADO'}")

    try:
        aht = setup_aht10()
        print("[I2C] AHT10 OK")
    except Exception as e:
        aht = None
        print(f"[I2C] AHT10 não inicializou: {e}")

    client = mqtt.Client()
    client.on_connect = on_connect
    client.on_message = on_message
    client.connect(BROKER_HOST, BROKER_PORT, 60)
    client.loop_start()

    print("[MQTT] Broker:", BROKER_HOST, BROKER_PORT)
    print(f"[RPM] janela={WINDOW_S:.0f}s | PPR_encoder={PPR_ENCODER:g} |
motor/encoder={MOTOR_PER_ENCODER:.4f}")
```

```
apply_state(False)
mqtt_publish_states(client)

counter = PulseCounterLgpio(gpiochip=0, pin=GPIO_PULSE)
counter.start()

last_btn = 1
last_toggle_time = 0.0

last_ds_time = 0.0
last_aht_time = 0.0

win_start = time.time()
rpm_motor = 0.0

try:
    while True:
        now = time.time()

        # botão local
        btn = GPIO.input(GPIO_BTN)
        if btn == 0 and last_btn == 1 and (now - last_toggle_time) > 0.25:
            apply_state(not cmd_run)
            mqtt_publish_states(client)
            last_toggle_time = now
        last_btn = btn

        # DS18B20
        if w1_path and (now - last_ds_time) >= 1.0:
            t = read_ds18b20_c(w1_path)
            if t is not None:
                client.publish(TOPIC_DS18B20, f"{t:.3f}", retain=True)
            last_ds_time = now

        # AHT10
        if aht and (now - last_aht_time) >= 1.0:
            try:
                t2 = float(aht.temperature)
                rh = float(aht.relative_humidity)
                client.publish(TOPIC_AHT_T, f"{t2:.3f}", retain=True)
                client.publish(TOPIC_AHT_RH, f"{rh:.3f}", retain=True)
```

```
except Exception as e:
    print("[AHT10] erro leitura:", e)
    last_aht_time = now

# RPM janela (3s)
if (now - win_start) >= WINDOW_S:
    dt = now - win_start
    pulses = counter.read_and_reset()

    enc_rpm = (pulses / dt) * 60.0 / PPR_ENCODER if (dt > 0 and PPR_ENCODER > 0) else 0.0
    rpm_motor = enc_rpm * MOTOR_PER_ENCODER

    client.publish(TOPIC_RPM, f"{rpm_motor:.1f}", retain=True)
    print(f"[RPM] Pulsos={pulses} em {dt:.2f}s -> Motor={rpm_motor:.2f} rpm")

    win_start = now

    time.sleep(0.02)

except KeyboardInterrupt:
    print("\n[MQTT] saindo...")
finally:
    try:
        counter.stop()
    except Exception:
        pass
    try:
        client.loop_stop()
        client.disconnect()
    except Exception:
        pass
    gpio_cleanup()

if __name__ == "__main__":
    main()
```