



Universidade Federal de Uberlândia

Faculdade de Engenharia Elétrica

**Ferramenta embarcada para auxílio à depuração e automação
de testes em sistemas de IoT**

Luiz Pedro Bittencourt Souza

Uberlândia, MG

2026

Luiz Pedro Bittencourt Souza

**Ferramenta embarcada para auxílio à depuração e automação de testes em
sistemas de IoT**

Trabalho de Conclusão de Curso do Curso de Engenharia de Controle e Automação, como requisito parcial para a obtenção do Título de Bacharel em Engenharia de Controle e Automação, Faculdade de Engenharia Elétrica da Universidade Federal de Uberlândia.

Orientador: Prof. Dr. Marcelo Barros de Almeida

Uberlândia, MG

2026

Agradecimentos

Primeiramente, agradeço aos meus pais, Luiz Américo e Karina Teresa, pelo apoio incondicional e por sempre me incentivarem a seguir meus sonhos, cultivando em mim a dedicação e o foco nos estudos. Mãe, pai, sem vocês eu não chegaria nem perto de onde estou hoje.

Agradeço à minha namorada, Fernanda Carvalho, por ter sido minha grande companheira, não apenas durante a graduação, mas na vida. Seu apoio, incentivo e motivação foram essenciais para as minhas conquistas. Você foi a primeira a acreditar em mim.

Agradeço aos meus grandes amigos e parceiros de curso, Felipe e Rafael, com os quais tive o privilégio de compartilhar quase todas as disciplinas, trabalhos e provas. Vocês tornaram essa jornada mais leve e memorável.

Agradeço à equipe de desenvolvimento do Laboratório de Mecânica das Estruturas, Lucão, Lofrano, Lukinhas, Stephan, Felipe, Márcio, Marcelo, Vitão, Carlos e Dudu, por todo o aprendizado e pelos momentos, tanto desafiadores quanto divertidos, que vivemos juntos.

Por fim, expresso minha gratidão ao meu orientador, Prof. Dr. Marcelo Barros de Almeida, por acreditar no meu potencial. Sua orientação e a confiança no meu trabalho foram fundamentais para a conclusão desta etapa.

Meus mais sinceros agradecimentos a todos que, de alguma forma, fizeram parte desta trajetória.

*"Scientists study the world as it is,
engineers create the world that never has been."*

Theodore von Kármán

RESUMO

O avanço da Internet das Coisas (IoT) tem impulsionado o desenvolvimento de sistemas embarcados cada vez mais complexos, exigindo soluções que simplifiquem o processo de validação, depuração e análise de dados. Neste contexto, este trabalho propõe o desenvolvimento de um sistema embarcado voltado ao monitoramento remoto e à automação de testes, atuando como uma ferramenta de apoio no ciclo de desenvolvimento de dispositivos conectados. O sistema é capaz de registrar, classificar e transmitir eventos assíncronos gerados pelo *firmware*, possibilitando a análise em tempo real ou posterior por meio de um *data logger* integrado. A comunicação com o ambiente externo foi estruturada utilizando o protocolo MQTT, de modo a segmentar os diferentes tipos de dados adquiridos em tópicos distintos, como eventos de diagnóstico, variáveis de processo e *logs* de desempenho. Para isso, houve o empregamento do *framework* ESP-IDF, explorando suas bibliotecas de conectividade, suporte ao protocolo MQTT e ao sistema de arquivos. O projeto visa fornecer uma plataforma flexível, de baixo custo e facilmente adaptável, que auxilie engenheiros e pesquisadores na depuração e validação de sistemas voltados ao ecossistema IoT.

Palavras-chave: IoT; sistemas embarcados; MQTT; depuração; automação de testes; *data logger*.

ABSTRACT

The rapid growth of the Internet of Things (IoT) has driven the development of increasingly complex embedded systems, demanding solutions that simplify data analysis, validation, and debugging processes. In this context, this work proposes the development of an embedded system aimed at remote monitoring and test automation, serving as a support tool in the development cycle of connected devices. The system will be capable of recording, classifying, and transmitting asynchronous events generated by the firmware, enabling real-time or offline analysis through an integrated data logger. Communication with the external environment will be structured using the MQTT protocol, allowing the separation of acquired data into distinct topics such as diagnostic events, process variables, and performance logs. The ESP-IDF framework will be employed, leveraging its connectivity libraries, MQTT protocol support, and file system management. The project aims to provide a flexible, low-cost, and easily adaptable platform to assist engineers and researchers in debugging and validating IoT oriented systems.

Keywords: IoT; embedded systems; MQTT; debugging; test automation; data logger.

LISTA DE FIGURAS

Figura 1	Diagrama do IO/MUX.....	15
Figura 2	Diagrama de periféricos ESP32-S3FH4R2.....	16
Figura 3	Esquema de ligação e formato de <i>frame</i> UART.....	17
Figura 4	Diagrama da saída <i>Push-pull</i>	18
Figura 5	Diagrama da saída <i>Open Drain</i>	18
Figura 6	Esquema de ligação do barramento SDMMC.	19
Figura 7	Representação de uma rede de conexões MQTT.....	20
Figura 8	Representação da estrutura de um tópico MQTT	21
Figura 9	Escalonador de tarefas em um RTOS.....	23
Figura 10	Estruturas de filas em um RTOS.....	24
Figura 11	Estruturas de semáforos em um RTOS.	24
Figura 12	Placa desenvolvida com base no ESP32-S3FH4R2.....	28
Figura 13	Diagrama de arquitetura de <i>firmware</i>	29
Figura 14	Estrutura do diretório do projeto	32
Figura 15	Trecho de código responsável pela inicialização dos módulos do sistema...	33
Figura 16	Portal de configuração de rede	34
Figura 17	Portal de configuração de <i>hardware</i>	35
Figura 18	Trecho de definição de tópicos MQTT.....	37
Figura 19	Lógica de preenchimento do buffer circular por UART.....	39
Figura 20	Lógica de processamento de dados da UART	40
Figura 21	Serviço de interrupção do GPIO	42
Figura 22	Tarefa de armazenamento e publicação de mensagem do GPIO.....	43
Figura 23	Estrutura de salvamento de mensagens.....	44
Figura 24	Cabeçalho e conteúdo das mensagens	45
Figura 25	Saída no terminal do script.....	47
Figura 26	Arquivos gerados após o processamento dos logs binários.....	47
Figura 27	Sistema de monitoramento fotovoltaico	50
Figura 28	Variáveis aquisitionadas pelo data logger durante a operação.....	51
Figura 29	Extração de <i>logs</i> remotos pelo <i>datalogger</i>	54

Figura 30	Extração de <i>logs</i> remotos pelo <i>datalogger</i>	54
-----------	--	----

LISTA DE ABREVIATURAS E SIGLAS

ADC	Analog-to-Digital Converter
API	Application Programming Interface
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
ESP-IDF	Espressif IoT Development Framework
FW	Firmware
GCC	GNU Compiler Collection
GPIO	General Purpose Input/Output
HTTP	Hypertext Transfer Protocol
HW	Hardware
I/O	Input/Output
I2C	Inter-Integrated Circuit
IoT	Internet of Things
IRAM	Internal Random Access Memory
JSON	JavaScript Object Notation
LED	Light Emitting Diode
LoRa	Long Range
MCUs	Microcontrolador
MQTT	Message Queuing Telemetry Transport
NVS	Non-Volatile Storage
PID	Proportional-Integral-Derivative
QoS	Quality of Service
RAM	Random Access Memory
ROM	Read-Only Memory
RTOS	Real Time Operating Systems
SDMMC	Secure Digital Multimedia Card
SiP	System-in-Package
SoC	System on a Chip
SPI	Serial Peripheral Interface
SSID	Service Set Identifier
SW	Software
TCP	Transmission Control Protocol
TLS	Transport Layer Security
IP	Internet Protocol
UART	Universal Asynchronous Receiver-Transmitter

SUMÁRIO

1	INTRODUÇÃO	11
1.1	Justificativas	11
1.2	Objetivos	12
1.2.1	Objetivo Geral	12
1.2.2	Objetivos Específicos	12
2	REFERENCIAL TEÓRICO	13
2.1	Levantamento de Tecnologias	13
2.1.1	Microcontrolador ESP32-S3FH4R2	14
2.1.1.1	O Periférico UART	16
2.1.1.2	O periférico GPIO	17
2.1.1.3	O periférico SDMMC	19
2.1.2	O Protocolo MQTT	20
2.1.3	Sistemas Operacionais de Tempo Real	22
2.1.3.1	Tarefas	22
2.1.3.2	Filas	23
2.1.3.3	Semáforos	24
2.1.3.4	FreeRTOS no Ecossistema Espressif	25
2.1.4	Arquitetura de <i>software</i>	25
3	METODOLOGIA	27
3.1	Hardware do sistema	27
3.2	Desenvolvimento do <i>Firmware</i>	28
3.2.1	Módulo principal	32
3.2.2	Módulo de <i>WiFi</i>	33
3.2.3	Módulo de <i>MQTT</i>	36
3.2.3.1	Roteamento de tópicos MQTT com uso de X-Macro	37
3.2.3.2	Broker MQTT EMQX	38
3.2.4	Módulo de Comunicação UART	38
3.2.5	Módulo de GPIO	41

3.2.6	Estrutura das mensagens dos dados	43
3.2.6.1	Registro em cartão SD	45
3.2.6.2	Publicação via MQTT	46
3.3	Ferramentas de <i>Software</i> de Apoio	46
3.3.1	Decodificador e visualizador de logs binários	46
4	RESULTADOS E DISCUSSÕES	49
4.1	Caso de Teste 1: Monitoramento fotovoltaico com Telemetria LoRa	49
4.2	Caso de Teste 2: Monitoramento e Interação Remota com Sis- tema de Câmara Fria	52
4.3	Caso de Teste 3: Coleta Remota de <i>Logs</i> UART em Sistema de Monitoramento Inercial	53
4.4	Testes de Estresse do Sistema	55
5	CONCLUSÃO	57

1 INTRODUÇÃO

O aumento da complexidade dos sistemas embarcados e conectados ao ecossistema da *Internet of Things* (IoT, no português, Internet das Coisas) tem impulsionado a necessidade de metodologias mais eficazes para validação e verificação de *firmware*. À medida que os dispositivos se tornam mais inteligentes e integrados, cresce também o desafio de garantir a confiabilidade e a robustez do *software* embarcado. Nesse cenário, a automação de testes surge como uma ferramenta essencial para reduzir o tempo de desenvolvimento, minimizar falhas humanas e assegurar maior cobertura durante o processo de validação.

O processo de automatizar testes permite que rotinas de verificação sejam executadas de forma sistemática e repetível, possibilitando a identificação antecipada de erros e a coleta de métricas de desempenho. Sendo assim, essa abordagem mostra-se especialmente relevante em aplicações embarcadas, onde os ciclos de desenvolvimento são curtos e as margens de erro são reduzidas. Desse modo, a automação de testes bem planejada é um investimento que aumenta a eficiência e a confiabilidade do produto final, ao mesmo tempo em que reduz custos de manutenção e retrabalho [1].

Por outro lado, executar checagens experimentais de longa duração apresenta desafios consideráveis relacionados à coleta, sincronização e confiabilidade das informações obtidas ao longo do tempo. Além disso, pequenas falhas de comunicação ou interrupções na aquisição podem comprometer a validade de todo o ensaio, exigindo sistemas de monitoramento capazes de operar de forma estável, registrar dados continuamente e oferecer meios claros de supervisão e diagnóstico durante o processo.

1.1 Justificativas

O desenvolvimento de sistemas embarcados voltados para aplicações de IoT demanda processos de teste e validação cada vez mais rigorosos. Em muitos casos, o comportamento do *firmware* só pode ser avaliado de forma confiável após longos períodos de operação contínua, o que torna inviável a realização de testes exclusivamente manuais. A ausência de mecanismos automáticos de coleta e supervisão dificulta a identificação de falhas intermitentes, degradações de desempenho e anomalias que podem surgir apenas em cenários de uso prolongado.

Dessa forma, torna-se necessário o uso de ferramentas que possibilitem a automação

do processo de monitoramento e aquisição de dados durante os testes. Uma solução capaz de registrar eventos assíncronos, organizar as informações em categorias distintas e disponibilizar os dados de forma acessível contribui significativamente para a eficiência da validação de *firmware*. Além disso, a integração com protocolos modernos de comunicação, como o MQTT, permite a distribuição e segregação de diferentes tipos de dados, por exemplo, métricas de desempenho, logs de eventos e variáveis de sensores, facilitando o acompanhamento remoto e o diagnóstico em tempo real.

A adoção de uma ferramenta de monitoramento automatizado também reduz o risco de perda de informações durante longos ciclos de teste e amplia a rastreabilidade dos resultados obtidos. Portanto, essa metodologia é especialmente vantajosa em ambientes de pesquisa e desenvolvimento, onde múltiplas iterações de *firmware* precisam ser avaliadas em condições controladas e reproduzíveis. Assim, o projeto proposto justifica-se pela necessidade de otimizar o processo de depuração e validação experimental, fornecendo uma base sólida para o desenvolvimento de sistemas embarcados mais robustos, confiáveis e alinhados às demandas do ecossistema IoT.

1.2 Objetivos

1.2.1 Objetivo Geral

O principal objetivo deste trabalho é aplicar conhecimentos de redes de comunicação, arquitetura e engenharia de *software*, sistemas operacionais de tempo real e eletrônica embarcada no desenvolvimento de um sistema de baixo custo voltado a auxiliar o processo de criação e validação de dispositivos destinados ao mercado de IoT.

1.2.2 Objetivos Específicos

É possível seccionar o objetivo principal dentro dos seguintes objetivos específicos:

- Estudo aprofundado de arquitetura de *software*.
- Entendimento de protocolos de rede.
- Construção de um *firmware* robusto e modular que visa segurança, escalabilidade e aplicação de boas práticas.

2 REFERENCIAL TEÓRICO

2.1 Levantamento de Tecnologias

A consolidação das práticas de IoT tem promovido uma integração cada vez mais estreita entre sistemas físicos e digitais, demandando arquiteturas de *software* capazes de lidar com requisitos de conectividade, escalabilidade e confiabilidade. Segundo Bahga e Madisetti (2015), a IoT representa um ecossistema de dispositivos inteligentes, sensores e atuadores interconectados por redes de comunicação, que coletam, processam e compartilham dados para apoiar decisões automatizadas ou assistidas.

Portanto, esses dispositivos são frequentemente baseados em plataformas de *hardware* embarcado, que executam tarefas de aquisição, processamento local e transmissão de dados para sistemas remotos. Nesse contexto, o *data logger* surge como um elemento muito importante na arquitetura de um sistema IoT, pois atua como o componente responsável pela coleta e armazenamento de informações provenientes de sensores, viabilizando o monitoramento contínuo e a análise posterior dos dados.

No entanto, para que um *data logger* seja eficiente e confiável, é necessário um projeto de *software* que vá além da simples captura de amostras. Nesse sentido, arquitetura de *software* não se restringe à organização do código-fonte, mas representa um conjunto de decisões estruturais que impactam diretamente a capacidade de um sistema de evoluir, escalar e manter sua qualidade ao longo do tempo. [2]

Assim, o desenvolvimento de um *data logger* para aplicações embarcadas e IoT deve considerar princípios de arquitetura de *software* que permitam modularidade, separação de responsabilidades e baixo acoplamento entre os componentes. Desse modo, tal estratégia facilita a implementação de camadas independentes para aquisição de dados, armazenamento local e comunicação com serviços em nuvem, garantindo maior flexibilidade e reuso de código.

Bahga e Madisetti (2015) também ressaltam que a eficiência de uma arquitetura IoT depende da definição clara de camadas funcionais, percepção, rede e aplicação de protocolos padronizados de comunicação, como MQTT e HTTP. Nessa perspectiva, o *data logger* ocupa uma posição intermediária entre a camada de percepção e a de rede, atuando como um nó inteligente capaz de adquirir dados e otimizar o tráfego de informações para os servidores remotos.

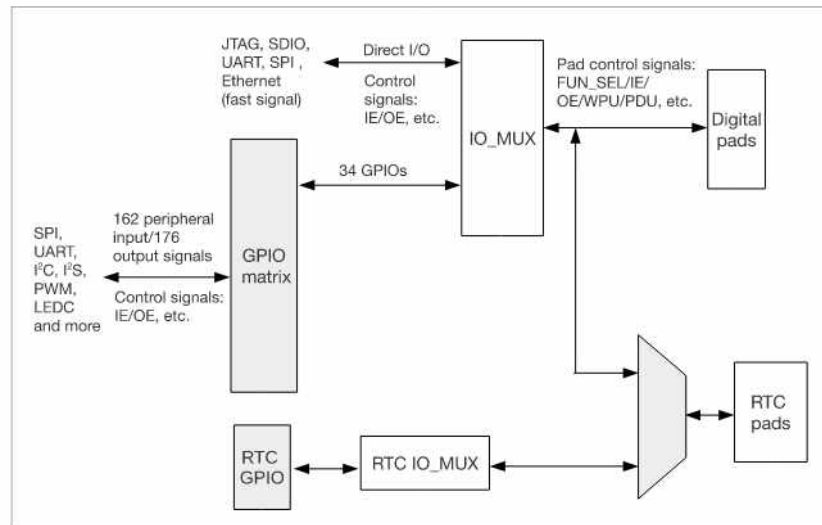
Por fim, ao adotar uma arquitetura de *software* bem definida, o *data logger* pode ser expandido para oferecer funcionalidades adicionais, como registro redundante de dados, tolerância a falhas e sincronização automática com bancos de dados externos. Tais características o tornam um componente essencial não apenas para aquisição de dados, mas também para a confiabilidade e manutenção de sistemas IoT em ambientes reais.

2.1.1 Microcontrolador ESP32-S3FH4R2

O microcontrolador ESP32-S3FH4R2 consiste em um *System on a Chip* (SoC) de alto desempenho que integra o microprocessador Xtensa Dual-Core de 32 bits LX7, capaz de operar em frequências de até 240 MHz para atender às exigências de sistemas embarcados complexos [3]. Esta versão específica do chip é uma solução do tipo *System-in-Package* (SiP) que inclui internamente 4 MB de memória *Flash* e 2 MB de PSRAM, proporcionando o gerenciamento eficiente de grandes volumes de dados e a execução de pilhas de rede robustas sem a necessidade de componentes externos de memória [3].

Um dos diferenciais arquiteturais do ESP32-S3 está no seu subsistema de IO/MUX (*Input/Output Multiplexer*), responsável por realizar o mapeamento entre os pinos físicos do dispositivo e os periféricos internos do SoC [3]. Esse mecanismo confere elevada flexibilidade ao projeto, permitindo que sinais de comunicação, controle e temporização sejam roteados dinamicamente para diferentes GPIOs, reduzindo restrições de pinagem e simplificando o desenvolvimento do *hardware*. Em conjunto com a matriz interna de GPIOs, o IO/MUX possibilita a configuração de funções alternativas, bem como o ajuste de características elétricas dos pinos, como resistores internos de *pull-up* e *pull-down*, modos de saída e capacidade de acionamento, assegurando integridade de sinal e confiabilidade operacional [3].

Figura 1: Diagrama do IO/MUX.

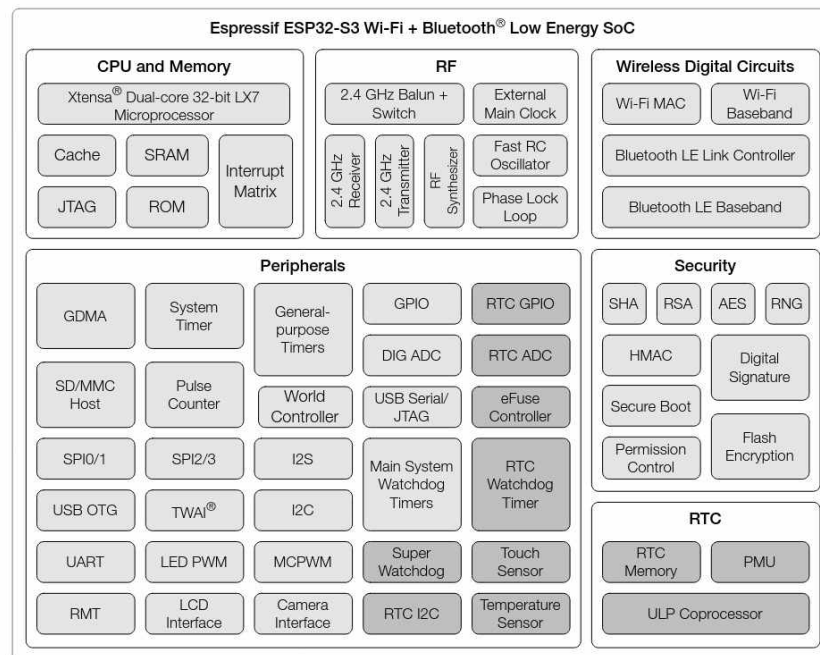


Fonte: ESP32-S3 Series Datasheet Version 2.1 . [3]

No que diz respeito à conectividade, o *hardware* oferece suporte nativo para redes *Wi-Fi* e *Bluetooth*, o que possibilita a integração com protocolos de comunicação modernos, como o MQTT, para a transmissão de dados de diagnóstico e variáveis de processo [3]. A arquitetura interna do SoC disponibiliza uma vasta gama de periféricos de comunicação, incluindo controladores UART, SPI e I2C, além de interfaces analógicas e suporte a barramentos de alta velocidade [3].

Para as funcionalidades de armazenamento de longa duração exigidas pelo *data logger* do projeto, o chip utiliza um controlador de host SDMMC, garantindo integridade e velocidade na escrita dos eventos assíncronos gerados pelo *firmware* por meio de um sistema de arquivos integrado [4].

Figura 2: Diagrama de periféricos ESP32-S3FH4R2



Fonte: ESP32-S3 Series Datasheet Version 2.1 . [3]

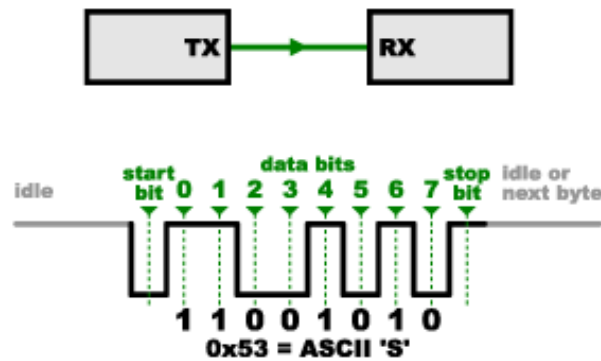
2.1.1.1 O Periférico UART

A UART (Universal Asynchronous Receiver/Transmitter) é um protocolo de comunicação serial assíncrono, muito usado para troca de dados ponto-a-ponto entre dispositivos, como microcontroladores, sensores e módulos externos [5].

O periférico utiliza basicamente duas linhas: TX (transmissão) e RX (recepção), além do GND comum. Por ser assíncrono, não há linha de clock: transmissor e receptor precisam apenas estar configurados com os mesmos parâmetros, como *baud rate*, número de bits de dados, paridade e bits de parada [5].

- 1 bit de start (nível lógico baixo)
- 5 a 9 bits de dados
- 1 bit de paridade (opcional)
- 1 bit obrigatório de stop (podem ser dois)

Figura 3: Esquema de ligação e formato de *frame* UART.

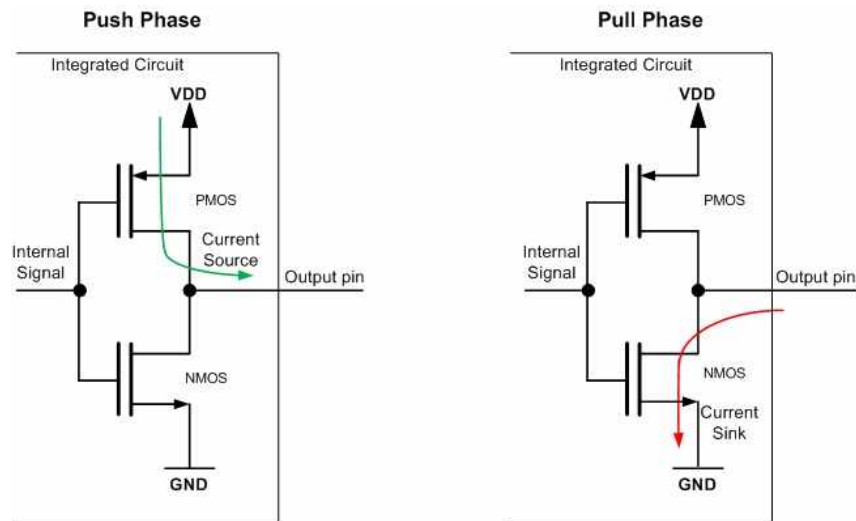


Fonte: Adaptado de [6].

2.1.1.2 O periférico GPIO

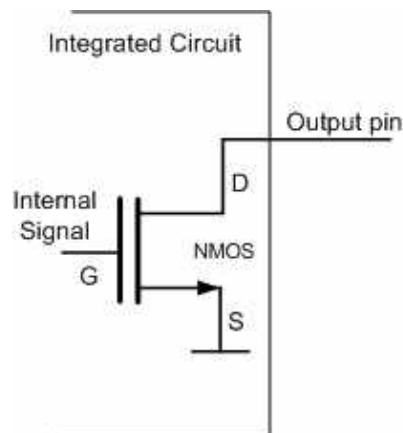
GPIO (General Purpose Input/Output) é o periférico mais básico e versátil de um microcontrolador. Ele permite que um pino funcione como entrada, saída, ou em modos alternativos, dependendo da arquitetura do chip. Como entrada, o GPIO pode ler níveis lógicos (0 ou 1), detectar eventos como bordas de subida/descida e, em muitos MCUs, gerar interrupções. Normalmente há suporte a resistores internos de *pull-up* ou *pull-down* para garantir um nível definido quando o pino está em aberto. Como saída, o GPIO pode forçar níveis alto ou baixo para controlar LEDs, relés, transistores ou servir como sinal de controle para outros circuitos [7].

No modo de saída *push-pull*, o pino GPIO é controlado por dois transistores complementares internos: um responsável por conectar o pino ao nível alto (VDD) e outro ao nível baixo (GND). Essa configuração permite que o microcontrolador force ativamente tanto o nível lógico alto quanto o baixo, resultando em transições rápidas, boa integridade de sinal e maior capacidade de corrente em ambas as direções. Por esse motivo, o modo *push-pull* é amplamente utilizado para acionar LEDs, sinais digitais gerais e interfaces que exigem comutação rápida e bem definida [8].

Figura 4: Diagrama da saída *Push-pull*.

Fonte: Adaptado de [8]

Já no modo de dreno aberto (*open-drain*), o GPIO possui apenas o transistor responsável por conectar o pino ao nível baixo. Quando ativado, o pino é puxado para GND; quando desativado, ele permanece em alta impedância, exigindo um resistor de *pull-up* externo ou interno para que o nível lógico alto seja estabelecido. Essa característica permite que múltiplos dispositivos compartilhem a mesma linha sem conflito elétrico, desde que apenas um deles force o nível baixo por vez [8].

Figura 5: Diagrama da saída *Open Drain*.

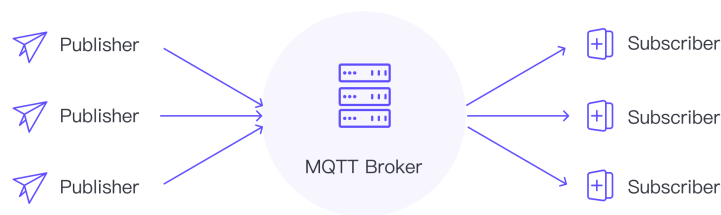
Fonte: Adaptado de [8]

2.1.2 O Protocolo MQTT

O protocolo *Message Queuing Telemetry Transport* (MQTT) consolidou-se como um padrão fundamental em aplicações de Internet das Coisas (IoT) devido à sua leveza e eficiência em dispositivos com recursos computacionais restritos. Originalmente concebido pela IBM na década de 1990, ele foi projetado especificamente para operar sob condições adversas, como redes de alta latência e largura de banda limitada, o que o torna ideal para sistemas embarcados conectados.

Nesse contexto, a operação do MQTT baseia-se no modelo de comunicação publicador/subscritor (*publish/subscribe*), no qual não há uma conexão direta ponto a ponto entre os dispositivos. Em vez disso, a interação é mediada por um intermediário central denominado *broker*, responsável por gerenciar o fluxo de mensagens entre as partes. Enquanto os publicadores enviam dados ao *broker* sem conhecer os destinatários, os elementos interessados inscrevem-se em canais específicos para receber apenas as informações pertinentes. Embora esse paradigma não seja inédito, assemelhando-se à troca de informações em redes *Foundation Fieldbus*, sua aplicação no MQTT otimiza drasticamente o tráfego de dados em ambientes distribuídos.

Figura 7: Representação de uma rede de conexões MQTT



Fonte: Fan Wang, via portal EMQX. [9]

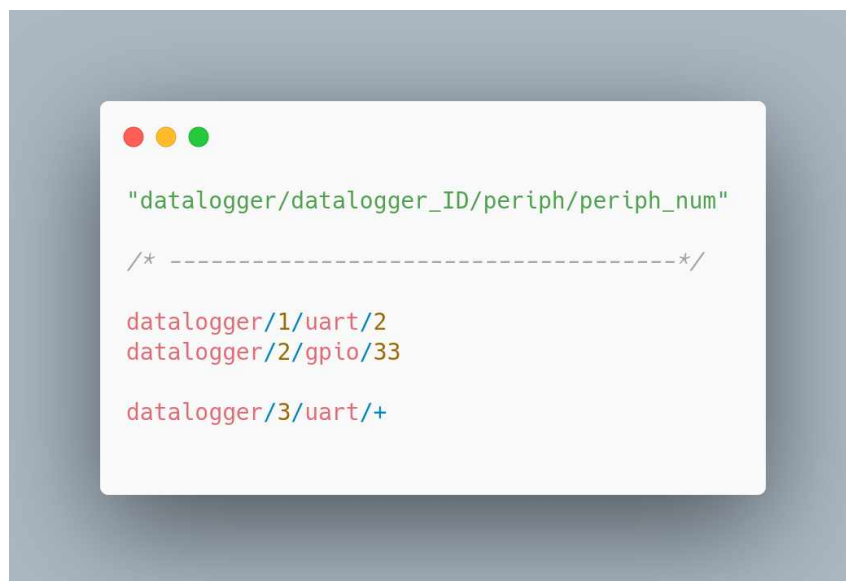
Sob a perspectiva arquitetural, a principal vantagem dessa estrutura é o desacoplamento completo entre os agentes comunicantes, permitindo independência temporal e de endereçamento IP, característica frequentemente inviável no modelo cliente/servidor. Contudo, essa centralização introduz uma vulnerabilidade crítica: o *broker* configura-se como um ponto único de falha (*single point of failure*). Para mitigar esse risco e assegurar a robustez em aplicações industriais, empregam-se estratégias de redundância ou operação

em *clusters*, que garantem alta disponibilidade e escalabilidade ao sistema.

Adicionalmente, o protocolo destaca-se por propriedades técnicas que reforçam sua viabilidade em sistemas de tempo real, como o baixo *overhead*, com cabeçalhos mínimos de apenas 2 *bytes*, e a flexibilidade das camadas de transporte, sendo compatível com TCP/IP e *WebSockets*. Outro diferencial relevante é a disponibilidade de diferentes níveis de Qualidade de Serviço (QoS), que permitem ao desenvolvedor calibrar o rigor da entrega das mensagens conforme a criticidade da aplicação.

Quanto à organização das informações, o MQTT utiliza o conceito de tópicos para realizar o endereçamento lógico. Estes são estruturados como sequências de caracteres hierárquicas, permitindo uma segmentação detalhada de fluxos de dados, como diagnósticos de *firmware*, métricas de sensores e logs de desempenho. Ao realizar uma subscrição, o dispositivo sinaliza ao *broker* seu interesse em um caminho específico, sendo notificado instantaneamente sempre que novos dados são publicados naquela ramificação.

Figura 8: Representação da estrutura de um tópico MQTT



Fonte: Autoria Própria (2026).

Por fim, visando a otimização do monitoramento, o protocolo admite o uso de caracteres curinga (*wildcards*). O símbolo de adição (+) é utilizado para substituir um único nível na hierarquia, enquanto a cerquilha (#) permite a assinatura de todos os sub-tópicos subsequentes a partir de um nó específico. Tais funcionalidades conferem ao sistema uma agilidade superior na gestão de eventos assíncronos, consolidando o MQTT

como uma ferramenta modular e extensível para a telemetria moderna.

2.1.3 Sistemas Operacionais de Tempo Real

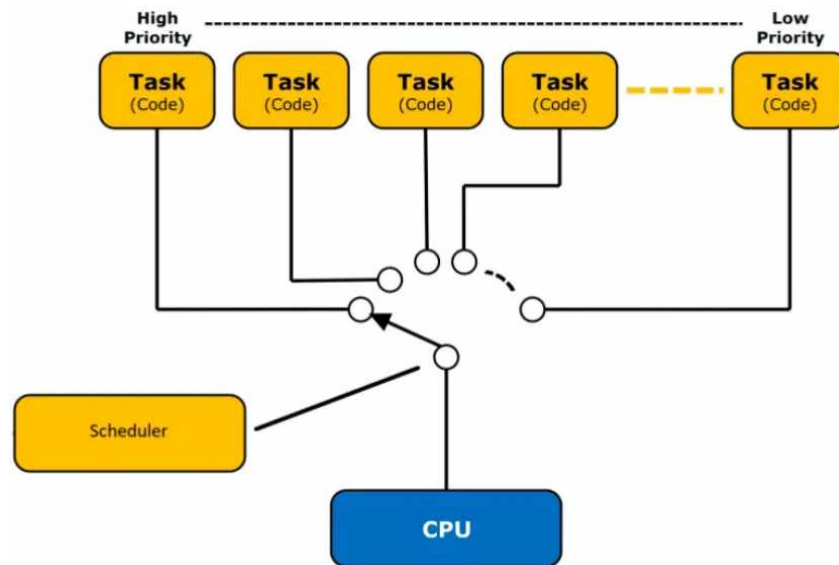
Sistemas Operacionais de Tempo Real, ou RTOS (Real-Time Operating Systems), são projetados para atender aplicações nas quais o tempo de resposta é tão crítico quanto a funcionalidade em si [10]. Diferentemente de sistemas operacionais de propósito geral, um RTOS garante comportamento determinístico, ou seja, a execução de tarefas ocorre dentro de limites temporais previsíveis, mesmo sob condições de alta carga do sistema. Esse tipo de característica é essencial em sistemas embarcados empregados em automação, instrumentação, telecomunicações e controle industrial [11].

Um RTOS organiza a execução do *software* em torno de tarefas concorrentes, cada uma com prioridades bem definidas, e utiliza mecanismos de escalonamento preemptivo para assegurar que atividades críticas sejam atendidas dentro de seus *deadlines*. Além disso, oferece serviços fundamentais como temporização precisa, sincronização entre tarefas, comunicação interprocessos e gerenciamento eficiente de recursos de *hardware*, abstraindo grande parte da complexidade do desenvolvimento de aplicações embarcadas [12].

2.1.3.1 Tarefas

As tarefas (*tasks*) constituem a unidade básica de execução em um RTOS. Cada tarefa representa um fluxo independente de controle, com sua própria pilha e contexto de execução [12]. O escalonador do sistema decide qual tarefa será executada a cada instante com base em critérios como prioridade e estado da tarefa (executando, pronta, bloqueada ou suspensa). Esse modelo permite que múltiplas funcionalidades coexistam de forma organizada e previsível dentro do mesmo sistema.

Figura 9: Escalonador de tarefas em um RTOS.



Fonte: José Moraes, via Portal Embarcados (2018).

2.1.3.2 Filas

As filas (*queues*) são mecanismos de comunicação entre tarefas que permitem a troca segura de dados em ambientes concorrentes. Por meio das filas, uma tarefa pode enviar mensagens ou estruturas de dados para outra, com suporte a bloqueio temporizado e controle de acesso, evitando condições de corrida e simplificando a sincronização entre produtores e consumidores de informação [12].

2.1.3.4 FreeRTOS no Ecosystema Espressif

No contexto dos SoCs da Espressif, como o ESP32-S3, o FreeRTOS desempenha um papel central na arquitetura de *software*. Ele é adotado como o núcleo do sistema operacional dentro do ESP-IDF, servindo como base para a implementação das *stacks* de conectividade, incluindo Wi-Fi, Bluetooth e protocolos de aplicação como MQTT [14]. Essa integração permite que tarefas relacionadas à comunicação, aquisição de dados, armazenamento e controle coexistam de forma organizada e determinística.

A presença do FreeRTOS nas *stacks* da Espressif possibilita o uso eficiente dos recursos de múltiplos núcleos, a priorização de tarefas críticas e a sincronização segura entre componentes do sistema. Dessa forma, o RTOS não apenas facilita o desenvolvimento de aplicações complexas, mas também garante robustez, escalabilidade e previsibilidade temporal, características indispensáveis para sistemas embarcados modernos conectados [14].

2.1.4 Arquitetura de *software*

A arquitetura de *software* é o alicerce que orienta as decisões estruturais e comportamentais de um sistema, definindo como seus componentes interagem, se comunicam e evoluem ao longo do tempo. A arquitetura de *software* deve ser compreendida como um conjunto de decisões técnicas que equilibram qualidades de projeto como desempenho, escalabilidade, manutenibilidade e testabilidade, e não apenas como um diagrama estático de módulos [2]. Essa visão enfatiza o papel do arquiteto como um tomador de decisões que deve considerar tanto as restrições técnicas quanto os requisitos funcionais do sistema. Em sistemas embarcados e distribuídos, como aqueles empregados em aplicações de Internet das Coisas (IoT), tais decisões tornam-se ainda mais críticas, uma vez que envolvem a integração estreita entre *software*, *hardware* e comunicação em tempo real sob severas restrições de processamento, memória e consumo energético.

No contexto do *data logger* proposto neste trabalho, baseado no microcontrolador ESP32-S3FH4R2, a arquitetura de *software* foi concebida para atender simultaneamente aos requisitos de aquisição contínua de dados, registro confiável em armazenamento não volátil e comunicação eficiente com sistemas externos. Para isso, adota-se uma organização modular do *firmware*, na qual funcionalidades como aquisição de sinais, gerenciamento de eventos, armazenamento em cartão SD via SDMMC e comunicação sem fio são implemen-

tadas como componentes logicamente independentes, porém coordenados. Essa separação favorece o baixo acoplamento e a alta coesão, princípios fundamentais para a manutenção e evolução do sistema ao longo do tempo.

A utilização de um sistema operacional de tempo real, especificamente o FreeRTOS, integrado nativamente às *stacks* de desenvolvimento da Espressif, desempenha papel central nessa arquitetura. O FreeRTOS permite a decomposição do sistema em tarefas concorrentes, cada uma responsável por um subconjunto bem definido de funcionalidades, como aquisição de dados, escrita em memória de massa e transmissão via rede. O uso de mecanismos de sincronização, como filas e semáforos, assegura a troca segura de informações entre tarefas e o tratamento determinístico de eventos assíncronos, característica essencial para um sistemas de armazenamento de monitoramento de dados que seja confiável.

Adicionalmente, a arquitetura adotada é fortemente orientada a eventos, alinhando-se ao modelo *publish/subscribe* empregado pelo protocolo MQTT. Esse padrão arquitetural é amplamente utilizado em sistemas IoT por reduzir o acoplamento entre produtores e consumidores de dados e por facilitar a escalabilidade do sistema. No projeto em questão, essa abordagem permite que os dados registrados pelo *data logger* sejam disponibilizados de forma eficiente para ferramentas de análise, monitoramento remoto ou integração com sistemas de automação de testes, sem impactar negativamente as rotinas críticas de aquisição e armazenamento local.

Dessa forma, a arquitetura de *software* do sistema proposto reflete uma combinação equilibrada entre princípios clássicos de engenharia de *software* e práticas consolidadas no desenvolvimento de sistemas embarcados modernos. Essa combinação resulta em um *firmware* robusto, extensível e adequado às demandas de aplicações industriais e experimentais que exigem confiabilidade, previsibilidade temporal e capacidade de evolução contínua.

3 METODOLOGIA

O corrente projeto visa o desenvolvimento de um sistema voltado a auxiliar o processo de criação e validação de dispositivos embarcados. Desse modo, metodologia adotada para o desenvolvimento deste projeto foi estruturada em etapas que abrangem o planejamento, implementação e validação do sistema. As principais fases estão descritas a seguir:

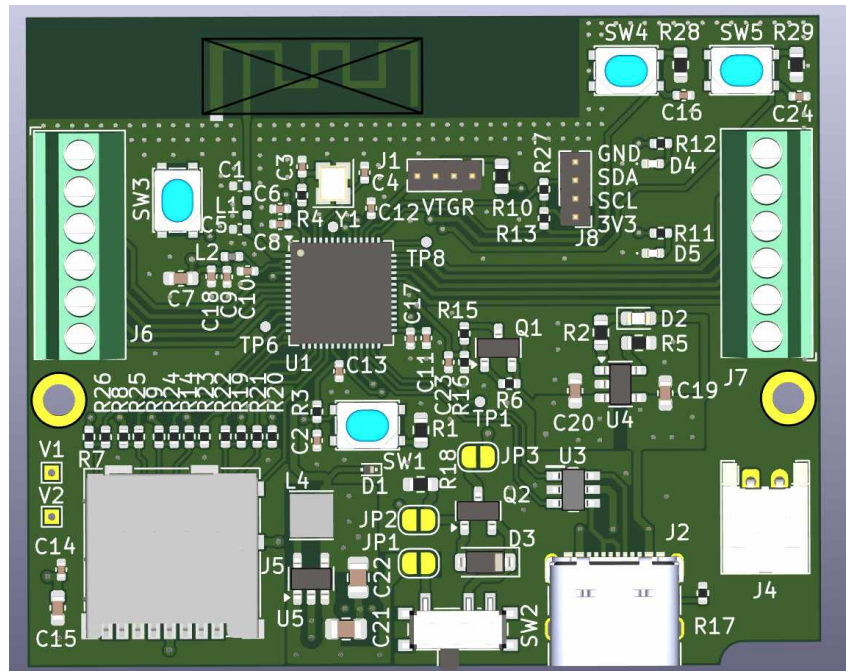
3.1 Hardware do sistema

O *hardware* utilizado neste trabalho foi desenvolvido e montado coletivamente pelos alunos no contexto de uma disciplina específica, ministrada pelo professor responsável, não sendo, portanto, o foco principal deste Trabalho de Conclusão de Curso. Ainda assim, é importante apresentar uma visão geral da plataforma empregada, uma vez que suas características influenciam diretamente as decisões de projeto e a arquitetura do *firmware* desenvolvido.

Após a análise das alternativas de microcontroladores disponíveis no mercado, foi adotado o ESP32-S3FH4R2, principalmente em função de sua conectividade sem fio nativa, que elimina a necessidade de módulos externos para comunicação Wi-Fi. O microcontrolador dispõe de 4 MB de memória *Flash* e 2 MB de PSRAM integradas ao encapsulamento, oferecendo uma base de *hardware* adequada para aplicações que demandam elevada capacidade de processamento e armazenamento temporário de dados.

Essa disponibilidade de memória permite a alocação de *buffers* extensos para tarefas críticas, como captura, processamento e registro de dados, além de viabilizar a implementação de um *firmware* estruturado em múltiplos módulos independentes, com mecanismos de persistência e tolerância a falhas. Outro fator relevante para a escolha do ESP32-S3 foi a presença do subsistema IO/MUX, que possibilita o mapeamento flexível dos periféricos internos para diferentes pinos físicos, reduzindo restrições impostas pelo *hardware* e facilitando tanto o desenvolvimento quanto futuras expansões do sistema.

Figura 12: Placa desenvolvida com base no ESP32-S3FH4R2.

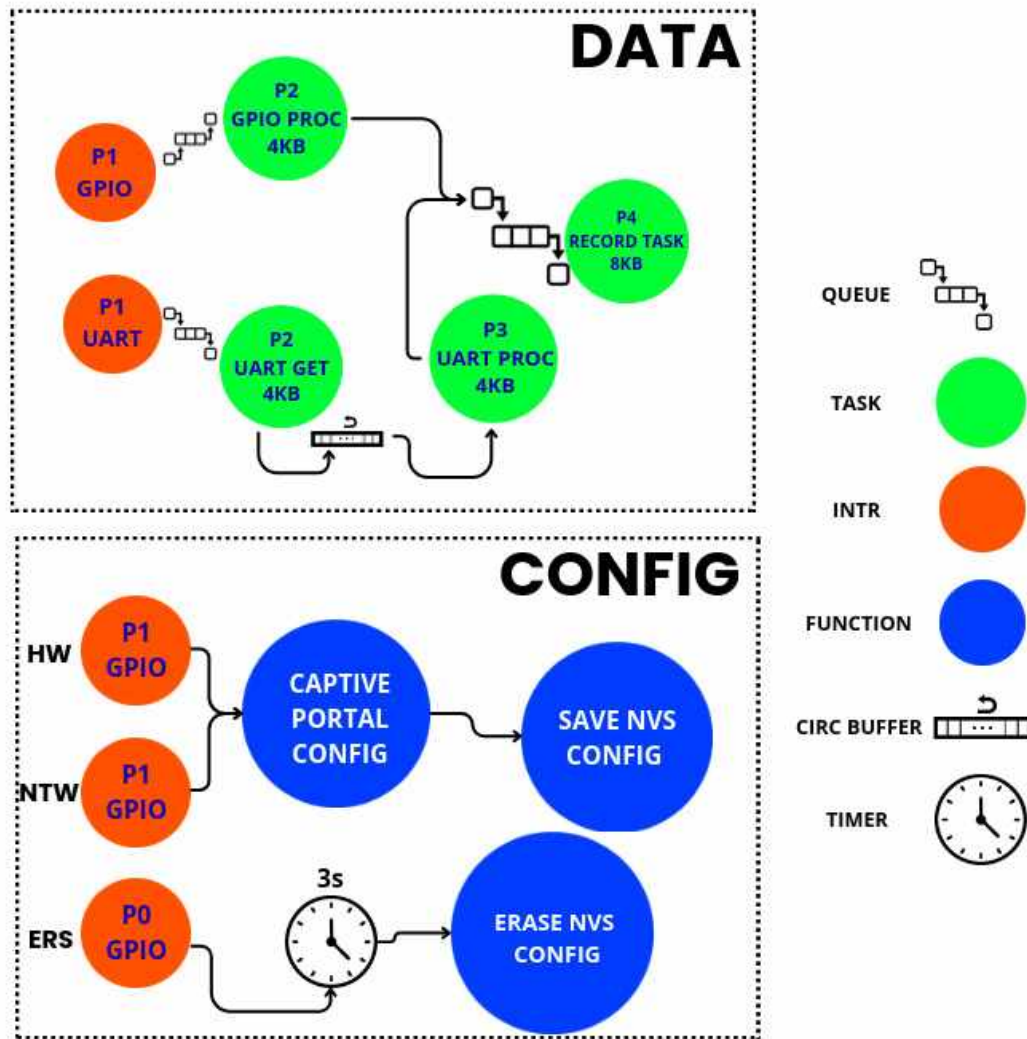


Fonte: Desenvolvido por Barros (2025) [15]

3.2 Desenvolvimento do *Firmware*

O *firmware* desenvolvido neste trabalho representa o núcleo funcional do sistema de *data logging*, sendo responsável por coordenar a aquisição de dados, o gerenciamento de eventos assíncronos, o armazenamento persistente e a comunicação com sistemas externos. Devido à complexidade das funcionalidades envolvidas e à necessidade de comportamento previsível, o *firmware* foi projetado de forma modular, explorando os recursos do microcontrolador ESP32-S3 e o suporte nativo ao sistema operacional de tempo real FreeRTOS.

A arquitetura do *firmware* foi concebida para separar claramente as responsabilidades de cada subsistema, permitindo que funcionalidades como comunicação, aquisição de dados e armazenamento operem de maneira concorrente e independente. Essa abordagem reduz o acoplamento entre módulos, facilita a manutenção do código e possibilita a expansão futura do sistema sem impacto significativo sobre a estrutura existente.

Figura 13: Diagrama de arquitetura de *firmware*

Fonte: Autoria própria (2026).

O diagrama apresentado na Figura 13 representa a arquitetura de *firmware* do sistema, organizada em dois blocos principais: **DATA** e **CONFIG**. Essa separação tem como objetivo isolar o fluxo principal de aquisição e registro de dados das rotinas responsáveis pela configuração do dispositivo, permitindo que cada subsistema opere de forma mais organizada e independente.

O bloco **DATA** representa o fluxo de aquisição, processamento e registro das informações geradas pelo sistema. Esse fluxo é estruturado como um pipeline no qual interrupções de hardware capturam eventos e encaminham os dados para tarefas do sistema operacional, responsáveis pelo processamento posterior. Nas rotinas de interrupção é exe-

cutado apenas o mínimo necessário, normalmente o envio de eventos para filas (*queues*) ou armazenamento de dados em *buffers* circulares, evitando processamento pesado dentro do contexto de interrupção.

As principais interrupções utilizadas no sistema são associadas aos periféricos GPIO e UART. A partir desses eventos, diferentes tarefas realizam o processamento das informações:

- ***GPIO PROC (P2 – 4 KB de stack)***: Responsável por processar eventos provenientes das interrupções de GPIO. A tarefa recebe os eventos através de uma fila enviada pela ISR e pode realizar filtragem ou interpretação antes de encaminhar os dados para a etapa de registro.
- ***UART GET (P2 – 4 KB de stack)***: Responsável por retirar os bytes recebidos pela UART a partir de um buffer circular. Essa tarefa organiza os dados recebidos e os prepara para processamento posterior.
- ***UART PROC (P3 – 4 KB de stack)***: Realiza o processamento dos dados provenientes da UART, como montagem de pacotes, validação ou interpretação das informações recebidas.
- ***RECORD TASK (P4 – 8 KB de stack)***: Tarefa responsável por consolidar e registrar os dados coletados no sistema. Ela recebe informações das tarefas de processamento e realiza o armazenamento ou organização final dos dados.

As prioridades das tarefas seguem a convenção do sistema em que valores menores indicam maior prioridade. Dessa forma, tarefas com valores maiores possuem prioridade mais baixa, sendo executadas apenas quando não há tarefas de maior prioridade prontas para execução.

O bloco **CONFIG** representa o subsistema responsável pela configuração do dispositivo. Esse modo é ativado a partir da leitura de determinados pinos de entrada do microcontrolador, permitindo que o sistema entre em um modo especial de configuração.

Nesse modo, o *firmware* executa algumas funções específicas:

- ***CAPTIVE PORTAL CONFIG***: Inicia o modo de configuração do sistema, permitindo que o usuário acesse uma interface de configuração para definir parâmetros do dispositivo.

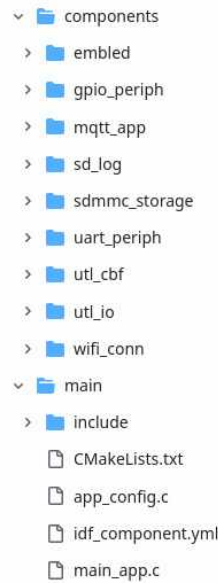
- ***SAVE NVS CONFIG***: Responsável por armazenar as configurações definidas pelo usuário na memória não volátil (NVS), garantindo que elas sejam preservadas após reinicializações.
- ***ERASE NVS CONFIG***: Executada quando o pino de reset de configuração permanece acionado por aproximadamente 3 segundos, apagando as configurações armazenadas e restaurando os parâmetros padrão do sistema.

No nível mais alto, o *firmware* é composto por um conjunto de tarefas executadas sob o FreeRTOS, cada uma responsável por uma função específica do sistema, como gerenciamento da conectividade Wi-Fi, comunicação via protocolo MQTT, recepção e envio de dados por UART, monitoramento de sinais digitais por meio de GPIOs e escrita dos dados no cartão SD utilizando a interface SDMMC. A comunicação entre essas tarefas é realizada por meio de mecanismos de sincronização e troca de dados, como filas e semáforos, garantindo a integridade das informações e o tratamento adequado de eventos concorrentes.

Além disso, o *firmware* incorpora uma camada de abstração para a configuração e o controle dos periféricos, explorando o subsistema IO/MUX do ESP32-S3 para mapear dinamicamente as funções de *hardware* aos pinos físicos. Essa estratégia confere maior flexibilidade ao projeto e reduz a dependência de uma pinagem fixa, aspecto relevante para sistemas experimentais e de testes.

Nas seções subsequentes, são detalhados individualmente os principais módulos que compõem o *firmware*, abordando suas responsabilidades, estrutura interna e interação com os demais componentes do sistema. Essa organização permite uma compreensão gradual do funcionamento do sistema, partindo de uma visão global até a descrição detalhada de cada subsistema implementado, conforme é demonstrado na Figura 14.

Figura 14: Estrutura do diretório do projeto



Fonte: Autoria própria (2026).

3.2.1 Módulo principal

O módulo principal do *firmware* atua como ponto central de inicialização e coordenação do sistema, sendo responsável por preparar o ambiente de execução e acionar os principais subsistemas do *data logger*. Implementado a partir da função *app_init_periph*, conforme o modelo adotado pelo ESP-IDF, esse módulo define a sequência de inicialização dos periféricos e serviços essenciais ao funcionamento da aplicação.

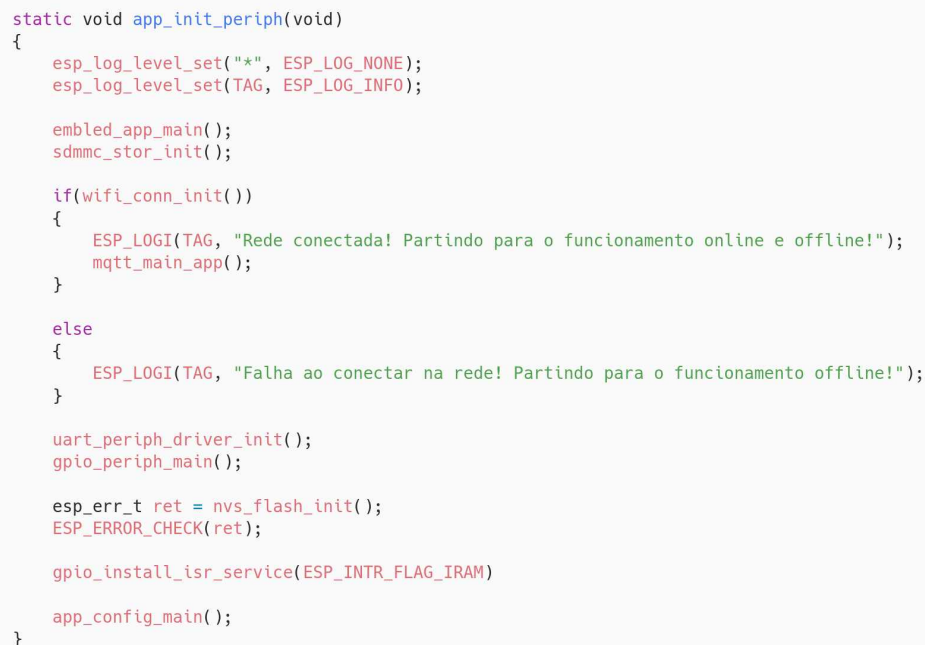
Durante a inicialização, são configurados os níveis de log, garantindo que apenas mensagens relevantes sejam exibidas durante a execução do sistema. Em seguida, são acionados os módulos responsáveis pela inicialização do armazenamento em cartão SD via SDMMC, pela conectividade Wi-Fi e pela comunicação MQTT, permitindo que o sistema opere tanto em modo *online* quanto *offline*, de acordo com a disponibilidade da rede. Essa abordagem assegura flexibilidade operacional e robustez frente a falhas de comunicação.

O módulo principal também é responsável por inicializar os periféricos de comunicação serial e de entrada e saída digital, bem como por configurar serviços fundamentais do sistema, como a memória não volátil (NVS) e o serviço de interrupções para GPIOs.

Por fim, são carregadas as configurações gerais da aplicação, centralizando parâmetros de operação e comportamento do *firmware*.

Além de orquestrar a inicialização, o módulo principal mantém estados internos que indicam o sucesso da configuração de subsistemas críticos, como UART e SDMMC. Esses indicadores podem ser consultados por outros módulos do *firmware*, permitindo decisões condicionais e aumentando a confiabilidade do sistema como um todo.

Figura 15: Trecho de código responsável pela inicialização dos módulos do sistema



```
static void app_init_periph(void)
{
    esp_log_level_set("*", ESP_LOG_NONE);
    esp_log_level_set(TAG, ESP_LOG_INFO);

    embled_app_main();
    sdmmc_stor_init();

    if(wifi_conn_init())
    {
        ESP_LOGI(TAG, "Rede conectada! Partindo para o funcionamento online e offline!");
        mqtt_main_app();
    }

    else
    {
        ESP_LOGI(TAG, "Falha ao conectar na rede! Partindo para o funcionamento offline!");
    }

    uart_periph_driver_init();
    gpio_periph_main();

    esp_err_t ret = nvs_flash_init();
    ESP_ERROR_CHECK(ret);

    gpio_install_isr_service(ESP_INTR_FLAG_IRAM)

    app_config_main();
}
```

Fonte: Autoria própria (2026).

3.2.2 Módulo de *WiFi*

O módulo de conectividade Wi-Fi desempenha um muito importante no *firmware* desenvolvido, sendo responsável não apenas pela comunicação em rede, mas também pela configuração dinâmica do sistema. A *stack* de Wi-Fi implementada foi projetada para oferecer flexibilidade ao usuário final, permitindo que o sistema opere tanto em modo

totalmente *offline* quanto em modo conectado, sem comprometer suas funcionalidades principais de *data logging*.

Durante a inicialização, o sistema verifica a existência de uma configuração válida de rede previamente armazenada na memória não volátil (NVS). Caso essa configuração esteja disponível, o módulo tenta automaticamente estabelecer a conexão com a rede Wi-Fi. Se não houver configuração válida, ou caso a conexão falhe, o sistema disponibiliza um portal captivo, permitindo ao usuário decidir se deseja prosseguir com ou sem conectividade de rede. Ao optar pelo funcionamento com Wi-Fi, o portal fornece interfaces para a inserção do identificador da rede (SSID) e da respectiva senha.

Figura 16: Portal de configuração de rede



The image shows a captive portal interface titled "Datalogger Setup" on a dark blue background. At the top, there is a toggle switch labeled "Modo Offline (Sem Wi-Fi)". Below this, the "SSID" section features a dropdown menu currently displaying "APARTAMENTO-303-2.4 (-91dBm)". Underneath the dropdown is a button labeled "Atualizar Lista" with a circular refresh icon. The "Senha" (Password) section has a text input field with masked characters (dots). At the bottom of the form is a large blue button labeled "Salvar Configuração". The footer of the interface reads "Firmware Config Portal".

Fonte: Autoria própria (2026).

Um diferencial adicional do módulo de Wi-Fi é sua integração com a configuração funcional dos pinos do microcontrolador. Por meio do portal captivo, o usuário pode definir o papel de determinados pinos, selecionando se estes devem operar como GPIOs ou como interfaces UART. Essa abordagem explora a flexibilidade do subsistema IO/MUX do ESP32-S3 e amplia significativamente a adaptabilidade do sistema a diferentes cenários

de uso.

Figura 17: Portal de configuração de *hardware*

The image shows a web interface titled "Hardware Config". It is divided into two main sections: "UART CONFIGURATION" and "GPIO CONFIGURATION".

UART CONFIGURATION

- UART 0** (checked):
 - Baud: 115200
 - TX Pin: 0
 - RX Pin: IO
- UART 1** (unchecked):
 - Baud: 115200
 - TX Pin: IO
 - RX Pin: IO
- UART 2** (unchecked):
 - Baud: 115200
 - TX Pin: IO
 - RX Pin: IO

GPIO CONFIGURATION

- BOARD_1** (checked):
 - Mode: DISABLE
 - Intr: DISABLE
 - Pull Up: DISABLE
 - Pull Down: DISABLE
- BOARD_2** (checked):

Fonte: Autoria própria (2026).

O módulo de Wi-Fi também é responsável por viabilizar o correto funcionamento do protocolo MQTT, fornecendo a infraestrutura de rede necessária para a comunicação assíncrona com sistemas externos. Para melhorar a usabilidade e o diagnóstico do estado do sistema, foram implementados indicadores visuais por meio de LEDs, que sinalizam diferentes estados operacionais, como tentativa de conexão, conexão estabelecida, perda de conexão ativa e operação totalmente *offline*. Esses indicadores permitem ao usuário identificar rapidamente a condição de conectividade do sistema durante sua operação.

3.2.3 Módulo de *MQTT*

O módulo de comunicação MQTT é responsável por viabilizar a troca de dados e comandos entre o *data logger* e sistemas externos, funcionando como uma extensão remota do dispositivo quando a conectividade Wi-Fi está disponível. A implementação utiliza a biblioteca nativa do ESP-IDF, explorando recursos como suporte a TLS, gerenciamento de eventos assíncronos e integração direta com o sistema operacional de tempo real.

A conexão com o *broker* MQTT é estabelecida de forma segura por meio de TLS, utilizando certificado digital embarcado no *firmware*. Após a conexão bem-sucedida, o sistema realiza automaticamente a inscrição em um conjunto de tópicos previamente definidos, permitindo o recebimento de mensagens de configuração, comandos e atualizações de estado. O gerenciamento desses tópicos é feito de forma estruturada, utilizando uma tabela de *handlers* que associa cada tópico a uma função específica de tratamento, o que facilita a extensibilidade da aplicação e a adição de novos serviços MQTT.

As mensagens recebidas são tratadas de maneira centralizada por um mecanismo de filtragem de tópicos, que identifica o sufixo correspondente e encaminha o *payload* para o módulo responsável. Mensagens de configuração permitem a reconfiguração dinâmica de periféricos como UART e GPIO, enquanto mensagens de comando possibilitam a atuação remota sobre o *hardware*, como o acionamento de pinos digitais ou o envio de dados por interfaces seriais. Os dados recebidos nesses tópicos são estruturados em formato JSON, sendo analisados e validados antes da aplicação das configurações ou da execução dos comandos.

No sentido oposto, o módulo MQTT também atua na publicação dos dados coletados pelo sistema. Eventos registrados pelo *data logger*, provenientes de interfaces UART ou de GPIOs, são encapsulados em mensagens MQTT e publicados em tópicos específicos. Cada mensagem inclui informações temporais de alta resolução, baseadas em *timestamps* em microssegundos, seguidas dos dados associados ao evento, garantindo rastreabilidade e sincronização precisa das informações transmitidas.

O módulo foi projetado para operar de forma integrada ao restante do *firmware*, respeitando o estado de conectividade do sistema. Em situações de desconexão do *broker* ou indisponibilidade de rede, o sistema mantém sua operação local de *logging*, enquanto o módulo MQTT gerencia automaticamente os eventos de reconexão quando a conectividade é restabelecida. Essa abordagem assegura robustez operacional e permite que o *data*

logger funcione de maneira contínua tanto em ambientes conectados quanto em cenários totalmente *offline*.

3.2.3.1 Roteamento de tópicos MQTT com uso de X-Macro

O roteamento de tópicos MQTT no *firmware* foi implementado utilizando a técnica de X-Macro, com o objetivo de centralizar a definição dos tópicos e seus respectivos *handlers* em um único ponto do código. Essa abordagem reduz duplicações, evita inconsistências e facilita a manutenção do sistema à medida que novos tópicos são adicionados.

Por meio da X-Macro, cada tópico é definido juntamente com sua função de tratamento, permitindo que a mesma lista seja reutilizada para gerar automaticamente estruturas de dados, *switch cases* ou tabelas de despacho. Dessa forma, o *firmware* mantém uma única fonte de verdade para o mapeamento entre tópicos MQTT e rotinas internas, minimizando erros e aumentando a legibilidade do código.

Figura 18: Trecho de definição de tópicos MQTT

```
#define XMACRO_TOPIC_HANDLER_SUB_LISTS \
    X(CONFIG_REF , "datalogger/config/+", 17, handle_config_message) \
    X(OTA_REF , "datalogger/ota" , 14, handle_ota_message ) \
    X(CMD_REF , "datalogger/cmd/+" , 14, handle_cmd_message )

typedef void (*topic_handler_fn)(const char *suffix, const char *payload);

typedef enum topic_reffs_e
{
    #define X(topic_ref, topic_name, topic_size, topic_handle_func) topic_ref,
    XMACRO_TOPIC_HANDLER_SUB_LISTS
    #undef X

    MAX_TOPICS,
} topic_reffs_t;

typedef struct topic_handler_s
{
    const char *topic;
    size_t topic_len;
    topic_handler_fn func;
} topic_handler_t
```

Fonte: Autoria própria (2026).

Quando uma mensagem MQTT é recebida, o módulo realiza a identificação do tópico correspondente e encaminha o *payload* diretamente para a função associada. O *payload* é então processado de acordo com o contexto do tópico, podendo representar comandos, parâmetros de configuração ou dados de controle. Essa separação clara entre roteamento e processamento garante que cada módulo trate apenas da sua responsabilidade, alinhando-se aos princípios de baixo acoplamento e alta coesão.

3.2.3.2 Broker MQTT EMQX

O *broker* MQTT adotado neste trabalho foi o EMQX Cloud, uma solução de *broker* MQTT totalmente gerenciada que oferece um plano gratuito escalável e substancial, ideal para protótipos e aplicações embarcadas com conectividade esporádica ou contínua. No plano *Serverless* gratuito da EMQX, cada mês inclui uma cota significativa de recursos sem custo, com até 1 milhão de minutos de sessão, 1 GB de tráfego de mensagens e 1 milhão de ações de regras, o que permite manter vários clientes conectados simultaneamente e trocando mensagens por longos períodos sem gerar despesas operacionais [16].

Para dimensionar essa cota, 1 milhão de minutos de sessão pode suportar, por exemplo, vários dispositivos conectados continuamente durante todo um mês, e a cota de 1 GB de tráfego mensal é suficiente para milhões de mensagens MQTT de tamanho típico. Essa configuração gratuita pode ser usada indefinidamente enquanto a utilização se mantiver dentro desses limites.

3.2.4 Módulo de Comunicação UART

O módulo de comunicação UART foi projetado para atuar como uma das principais fontes de aquisição de dados do *data logger*, sendo responsável por receber fluxos assíncronos de informações provenientes de dispositivos externos, como sensores inteligentes, módulos auxiliares ou outros microcontroladores. A implementação foi estruturada para suportar múltiplas instâncias de UART de forma simultânea, explorando os recursos nativos do ESP32-S3 e mantendo escalabilidade e isolamento entre canais de comunicação.

A recepção de dados é baseada no modelo de *driver* assíncrono da ESP-IDF, no qual cada UART habilitada possui uma *task* dedicada responsável por tratar eventos provenientes da fila do *driver*. Os bytes recebidos são armazenados inicialmente em *buffers*

circulares independentes, permitindo desacoplar o tempo de recepção física do processamento lógico dos dados, demonstrado pela Figura 19. Essa abordagem reduz a probabilidade de perda de informação em cenários de alta taxa de transmissão ou carga elevada do sistema, além de facilitar o tratamento incremental dos dados recebidos.

Figura 19: Lógica de preenchimento do buffer circular por UART.



```

while (1)
{
    if(xQueueReceive(uart_queue[uart_num], &event, pdMS_TO_TICKS(10)))
    {
        switch (event.type)
        {
            case UART_DATA:
                int len = uart_read_bytes(uart_num, data, event.size, pdMS_TO_TICKS(100));
                if(len > 0)
                {
                    utl_cbf_put_n(uart_circ_buffers[uart_num], data, len, &written);
                }
            }
        }
    }

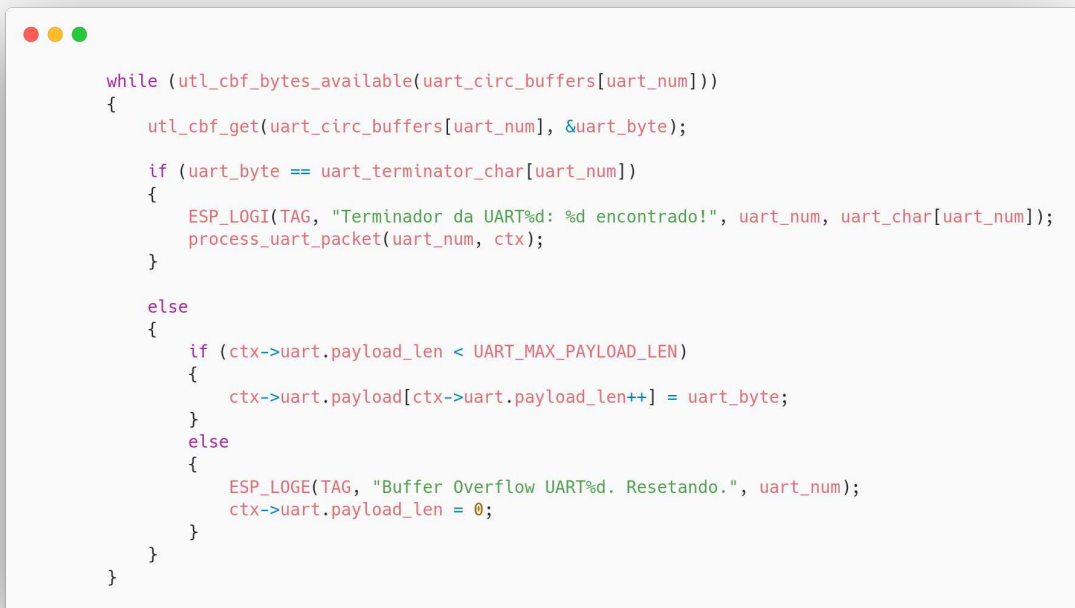
    /* ----- CONTINUAÇÃO DA LÓGICA / TRATAMENTO DE OUTROS CASOS ----- */
}

```

Fonte: Autoria própria (2026).

O processamento dos dados UART é realizado em uma *task* específica de gravação, que consome os bytes do *buffer* circular respeitando um orçamento máximo de processamento por ciclo, garantindo previsibilidade temporal e cooperação com o escalonador do FreeRTOS. Os *frames* são delimitados por um caractere escolhido do usuário, podendo ser 0x10, '\n' ou 0x00, '\0' estratégia que simplifica a detecção de pacotes de tamanho variável e evita dependência de *timeouts*, como é brevemente demonstrado na Figura 20. Uma vez identificado um *frame* completo, os dados são encapsulados em uma estrutura padronizada de log, associando *timestamp* em microssegundos, identificação do periférico e tipo de evento.

Figura 20: Lógica de processamento de dados da UART



```

while (utl_cbf_bytes_available(uart_circ_buffers[uart_num]))
{
    utl_cbf_get(uart_circ_buffers[uart_num], &uart_byte);

    if (uart_byte == uart_terminator_char[uart_num])
    {
        ESP_LOGI(TAG, "Terminador da UART%d: %d encontrado!", uart_num, uart_char[uart_num]);
        process_uart_packet(uart_num, ctx);
    }

    else
    {
        if (ctx->uart.payload_len < UART_MAX_PAYLOAD_LEN)
        {
            ctx->uart.payload[ctx->uart.payload_len++] = uart_byte;
        }
        else
        {
            ESP_LOGE(TAG, "Buffer Overflow UART%d. Resetando.", uart_num);
            ctx->uart.payload_len = 0;
        }
    }
}

```

Fonte: Autoria própria (2026).

Após o encapsulamento, o mesmo pacote de dados é encaminhado simultaneamente para os módulos de armazenamento em cartão SD e de comunicação MQTT, caracterizando uma arquitetura orientada a eventos e com reutilização de dados. Essa duplicação controlada garante que as informações sejam persistidas localmente com baixa latência, ao mesmo tempo em que podem ser transmitidas em tempo real quando houver conectividade de rede disponível.

A configuração das UARTs é totalmente dinâmica e controlada por parâmetros persistentes armazenados em memória não volátil (NVS). Cada instância pode ser habilitada ou desabilitada em tempo de execução, bem como ter seus pinos, taxa de transmissão e estado operacional ajustados sem necessidade de recompilação do *firmware*. Alterações de configuração acionam automaticamente a reinstalação do *driver* e a criação ou remoção das *tasks* associadas, assegurando consistência do sistema durante a operação.

De forma geral, o módulo UART foi concebido para ser robusto, flexível e altamente integrado ao restante do sistema, atendendo aos requisitos de aquisição confiável de dados, baixo acoplamento entre módulos e suporte a múltiplos fluxos de comunicação

simultâneos, características essenciais para o funcionamento do *data logger* proposto neste trabalho.

3.2.5 Módulo de GPIO

O módulo de GPIO foi desenvolvido com o objetivo de permitir a configuração dinâmica dos pinos digitais do sistema, bem como o registro confiável de eventos de mudança de nível lógico. Assim como os demais periféricos, os GPIOs podem ser habilitados ou desabilitados em tempo de execução, a partir de configurações persistidas na memória não volátil (NVS), seja via portal captivo ou por comandos remotos recebidos através do MQTT.

Cada GPIO é tratado internamente por meio de um índice lógico, que é posteriormente mapeado para um pino físico da placa. Essa abordagem desacopla o *firmware* da disposição física do *hardware*, facilitando a manutenção e possíveis revisões da placa. A configuração de cada pino inclui parâmetros como modo de operação, resistores de *pull-up* ou *pull-down* e tipo de interrupção, sendo aplicada dinamicamente sempre que uma nova configuração válida é recebida.

Para a captura de eventos, o módulo utiliza interrupções de *hardware* associadas aos GPIOs configurados. As rotinas de interrupção são mantidas propositalmente simples e executadas em memória IRAM, garantindo baixa latência e previsibilidade temporal. Para evitar múltiplos disparos indesejados decorrentes de ruídos ou *bouncing* mecânico, foi implementado um mecanismo de *debounce* por *software*, baseado em marcação temporal utilizando o temporizador de alta resolução do sistema.

Figura 21: Serviço de interrupção do GPIO

```
static void IRAM_ATTR gpio_isr_handler(void *args)
{
    uint32_t gpio_idx = (uint32_t)args;

    int64_t now = esp_timer_get_time();

    if ((now - last_evt_time[gpio_idx]) < GPIO_DEBOUNCE_US)
    {
        return;
    }

    last_evt_time[gpio_idx] = now;

    gpio_evt_t evt =
    {
        .gpio = (uint8_t)gpio_idx,
        .level = gpio_get_level(gpio_num),
        .time_us = now,
    };

    xQueueSendFromISR(gpio_evt_queue, &evt, &xHigherPriorityTaskWoken);
}
```

Fonte: Autoria própria (2026).

Os eventos capturados nas interrupções são enfileirados e processados por uma *task* dedicada, responsável por registrar os dados no sistema de log e, quando aplicável, encaminhá-los via MQTT. Essa separação entre o contexto de interrupção e o processamento de dados garante maior robustez ao sistema, evitando bloqueios ou sobrecarga no atendimento das interrupções. Cada evento registrado contém o instante de ocorrência, o identificador do pino e o nível lógico detectado, assegurando rastreabilidade e sincronização temporal com os demais periféricos monitorados pelo *datalogger*.

Figura 22: Tarefa de armazenamento e publicação de mensagem do GPIO

```
if (xQueueReceive(gpio_evt_queue, &evt, portMAX_DELAY))
{
    uint8_t gpio_logic = evt.gpio; // índice lógico do gpio

    int real_pin = gpio_pins[gpio_logic];

    ESP_LOGI(TAG, "GPIO%d | level=%d | t=%lld us", real_pin, evt.level, evt.time_us);

    sd_log_msg_t gpio_msg = {0};

    gpio_msg.log_header.header = LOG_PACKET_HEADER_INIT;
    gpio_msg.log_header.time_us = evt.time_us;
    gpio_msg.log_header.log_type = SD_LOG_GPIO;
    gpio_msg.log_header.periph_num = real_pin;

    gpio_msg.gpio.edge = evt.level;
    gpio_msg.gpio.level = evt.level;

    sd_log_data(&gpio_msg);
    mqtt_publish_msg(&gpio_msg);
}
```

Fonte: Autoria própria (2026).

3.2.6 Estrutura das mensagens dos dados

Para padronizar o registro e a transmissão dos eventos capturados pelo sistema, foi definida uma estrutura única de mensagem, denominada *sd_log_msg_t*, responsável por encapsular os dados provenientes dos diferentes periféricos monitorados. Apesar da nomenclatura associada ao armazenamento em cartão SD, essa estrutura é utilizada tanto para persistência local quanto para publicação via MQTT, caracterizando um modelo unificado de representação de eventos.

Figura 23: Estrutura de salvamento de mensagens

```
typedef struct PACKED sd_log_msg_s
{
    sd_log_hdr_t log_header;
    union
    {
        sd_log_uart_t uart;
        sd_log_gpio_t gpio;
    };
} sd_log_msg_t;
```

Fonte: Autoria própria (2026).

A estrutura é composta por um cabeçalho comum (*sd_log_hdr_t*) e por um bloco de dados específico, implementado por meio de uma *union*, que pode representar informações de UART ou de GPIO. O cabeçalho inclui um identificador fixo de validação, um *timestamp* em microssegundos obtido a partir do temporizador de alta resolução do sistema, o tipo de log e o identificador do periférico responsável pelo evento. A utilização do atributo *packed* garante que não haja preenchimento automático (*padding*) entre os campos, assegurando previsibilidade no tamanho da estrutura e consistência na gravação binária em memória externa.

Figura 24: Cabeçalho e conteúdo das mensagens

```

#define PACKED __attribute__((packed))

typedef struct PACKED sd_log_hdr_s
{
    uint32_t header;
    uint64_t time_us;
    uint8_t log_type;
    uint8_t periph_num;
} sd_log_hdr_t;

typedef struct PACKED sd_log_uart_s
{
    uint16_t payload_len;
    uint8_t payload[UART_MAX_PAYLOAD_LEN];
} sd_log_uart_t;

typedef struct PACKED sd_log_gpio_s
{
    uint8_t edge;
    uint8_t level;
} sd_log_gpio_t;

```

Fonte: Autoria própria (2026).

3.2.6.1 Registro em cartão SD

No contexto do armazenamento local, a função *sd_log_data()* é responsável por validar e persistir os dados no cartão SD utilizando escrita binária sequencial. Para eventos de UART, são gravados o cabeçalho, o tamanho do *payload* e os dados recebidos. Para eventos de GPIO, são gravados o cabeçalho e a estrutura correspondente ao estado do pino. Esse modelo garante rastreabilidade completa dos eventos, incluindo marcação temporal precisa e identificação inequívoca da origem dos dados.

A gravação é realizada de forma fragmentada (cabeçalho e conteúdo), o que oferece maior controle sobre validações e reduz o risco de inconsistências em caso de falhas parciais de escrita. Além disso, o uso de um identificador fixo no cabeçalho permite que arquivos binários sejam posteriormente analisados e validados durante a etapa de pós-processamento.

3.2.6.2 Publicação via MQTT

Embora a mesma estrutura de mensagem seja utilizada internamente para organizar os dados, o envio via MQTT não requer a inclusão explícita do cabeçalho binário. Isso ocorre porque a identificação do periférico e do tipo de evento já está implícita na hierarquia de tópicos utilizada pelo sistema. As mensagens são publicadas em tópicos organizados da mesma maneira mostrada na Figura 8.

Nessa organização, o tipo de periférico e seu identificador são definidos pelo próprio tópico MQTT, tornando redundante a transmissão de campos como *log_type* e *periph_num* dentro do *payload*. Assim, para comunicação em rede, apenas os dados relevantes ao evento são serializados e transmitidos, reduzindo o volume de tráfego e tornando a mensagem mais eficiente.

Essa distinção evidencia uma decisão arquitetural importante:

- Para armazenamento local, prioriza-se integridade e rastreabilidade completa.
- Para transmissão em rede, prioriza-se eficiência e contextualização via hierarquia de tópicos.

3.3 Ferramentas de *Software* de Apoio

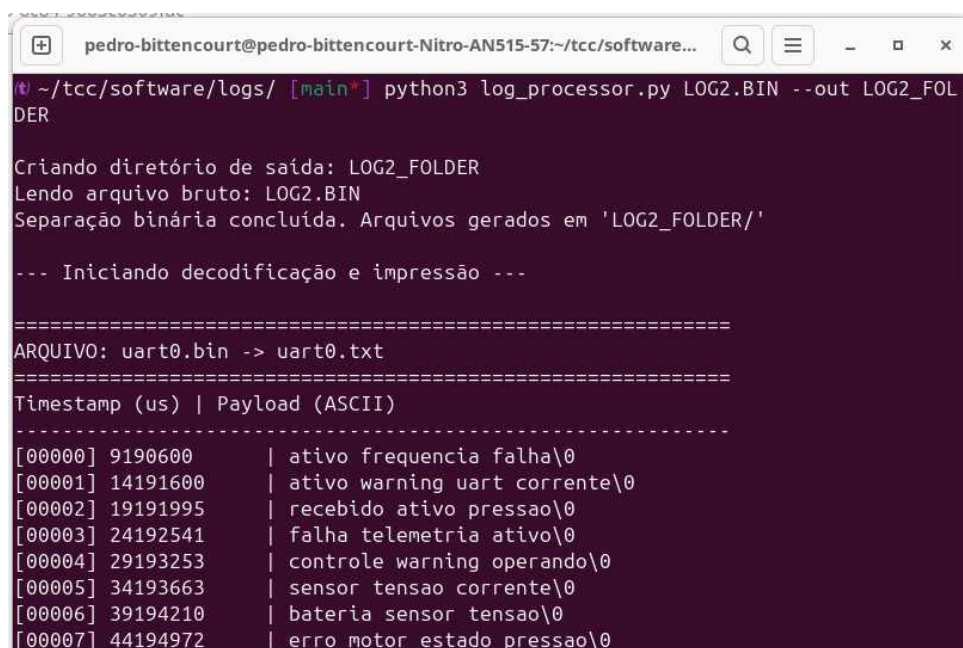
Além do *firmware* embarcado responsável pela aquisição, processamento e transmissão dos dados, foi desenvolvido um conjunto de ferramentas auxiliares em *software* com a finalidade de apoiar as etapas de teste, validação e análise do sistema. Essas ferramentas foram implementadas de modo a permitir tanto a inspeção *offline* dos dados armazenados quanto o monitoramento e a interação em tempo real por meio da infraestrutura de comunicação adotada. Dessa forma, estabeleceu-se um ambiente complementar ao dispositivo embarcado, possibilitando maior controle experimental, rastreabilidade dos eventos e verificação sistemática do funcionamento global do *datalogger*.

3.3.1 Decodificador e visualizador de logs binários

O primeiro componente desse conjunto consiste em um *script* de processamento e decodificação de logs binários, *log_processor.py* responsável por interpretar os arquivos

brutos gerados pelo sistema no cartão de memória. A ferramenta realiza a leitura estruturada dos registros conforme o protocolo definido no *firmware*, separa os eventos por tipo e periférico e gera arquivos organizados para análise posterior. Além disso, converte os dados para formato textual legível, preservando os *timestamps* e representando adequadamente os *payloads* recebidos, inclusive caracteres não imprimíveis. Essa ferramenta desempenhou papel fundamental na validação da integridade do armazenamento e na análise temporal dos eventos registrados.

Figura 25: Saída no terminal do script



```

pedro-bittencourt@pedro-bittencourt-Nitro-AN515-57:~/tcc/software/...
~/tcc/software/logs/ [main*] python3 log_processor.py LOG2.BIN --out LOG2_FOLDER
DER

Criando diretório de saída: LOG2_FOLDER
Lendo arquivo bruto: LOG2.BIN
Separação binária concluída. Arquivos gerados em 'LOG2_FOLDER/'

--- Iniciando decodificação e impressão ---

=====
ARQUIVO: uart0.bin -> uart0.txt
=====
Timestamp (us) | Payload (ASCII)
-----
[00000] 9190600      | ativo frequencia falha\0
[00001] 14191600     | ativo warning uart corrente\0
[00002] 19191995     | recebido ativo pressao\0
[00003] 24192541     | falha telemetria ativo\0
[00004] 29193253     | controle warning operando\0
[00005] 34193663     | sensor tensao corrente\0
[00006] 39194210     | bateria sensor tensao\0
[00007] 44194972     | erro motor estado pressao\0

```

Fonte: Autoria própria (2026).

Figura 26: Arquivos gerados após o processamento dos logs binários



```

pedro-bittencourt@pedro-bittencourt-Nitro-AN515-57:~/tcc/software/logs
~/tcc/software/logs/ [main*] ls -l
total 20
-rw-r--r-- 1 pedro-bittencourt pedro-bittencourt 1623 fev  6 12:56 LOG2.BIN
drwxrwxr-x 2 pedro-bittencourt pedro-bittencourt 4096 fev 22 11:44 LOG2_FOLDER
-rw-rw-r-- 1 pedro-bittencourt pedro-bittencourt 8265 fev 18 18:56 log_processor.py
~/tcc/software/logs/ [main*] ls -l LOG2_FOLDER
total 8
-rw-rw-r-- 1 pedro-bittencourt pedro-bittencourt 1329 fev 22 11:44 uart0.bin
-rw-rw-r-- 1 pedro-bittencourt pedro-bittencourt 2137 fev 22 11:44 uart0.txt
~/tcc/software/logs/ [main*]

```

Fonte: Autoria própria (2026).

A segunda ferramenta corresponde a um *script* de subscrição MQTT, desenvolvido para monitorar em tempo real os tópicos nos quais o sistema publica os dados coletados. Por meio dessa aplicação, é possível acompanhar o fluxo de mensagens, verificar a organização hierárquica dos tópicos, validar o conteúdo das publicações e confirmar o correto funcionamento da camada de comunicação. Essa abordagem permite observar o comportamento do dispositivo em operação *online*, facilitando a identificação de inconsistências e a avaliação da confiabilidade da transmissão.

Por fim, foi implementado um *script* destinado ao envio de comandos ao sistema, utilizando o mesmo mecanismo de comunicação empregado para a publicação dos dados. Essa ferramenta possibilita a realização de testes controlados de configuração e operação, permitindo simular cenários de uso e validar a correta interpretação e execução das instruções pelo *firmware*. Dessa maneira, estabelece-se um canal bidirecional de interação, essencial para testes funcionais completos e para a verificação do comportamento dinâmico do sistema desenvolvido.

4 RESULTADOS E DISCUSSÕES

Esta seção apresenta a evolução experimental do sistema desenvolvido, evidenciando o processo contínuo de aprimoramento ao longo de sua concepção e implementação. Mais do que uma validação pontual, os ensaios realizados refletiram uma abordagem iterativa, na qual cada teste contribuiu para a identificação de limitações, oportunidades de melhoria e novas funcionalidades a serem incorporadas. A cada etapa de experimentação, ajustes arquiteturais e refinamentos de *software* foram realizados com base em análise crítica dos resultados obtidos, buscando ampliar a robustez, a flexibilidade e a confiabilidade do sistema.

Nesse contexto, os experimentos não apenas avaliaram desempenho quanto à recepção de dados, integridade de armazenamento e publicação remota, mas também impulsionaram a incorporação de novas estratégias operacionais e funcionalidades adicionais. Assim, os resultados apresentados contemplam tanto testes controlados quanto aplicações práticas em cenários reais, demonstrando como o sistema evoluiu progressivamente a partir da experiência adquirida em campo e da constante reflexão sobre possíveis melhorias e expansões de uso.

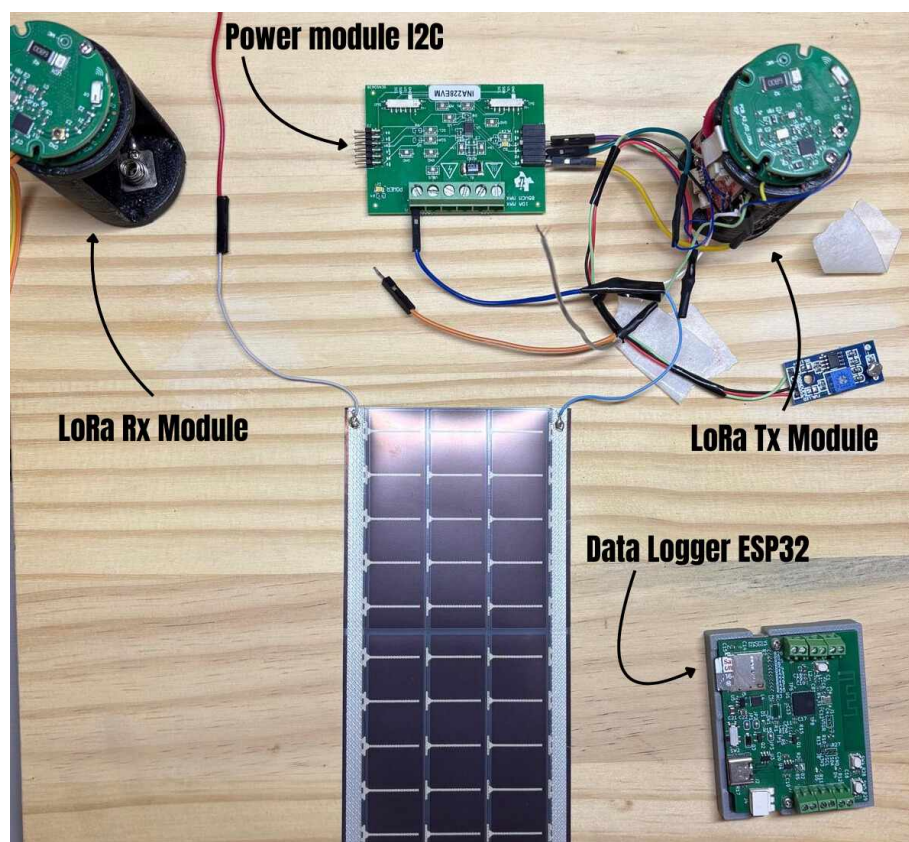
4.1 Caso de Teste 1: Monitoramento fotovoltaico com Telemetria LoRa

Como primeira validação prática da arquitetura proposta, o *datalogger* foi integrado a um sistema de telemetria sem fio de baixo consumo destinado ao monitoramento de painéis fotovoltaicos em ambientes remotos. Este ensaio teve como objetivo avaliar o desempenho do sistema em condições reais de operação, caracterizadas por restrições energéticas, ausência de infraestrutura de comunicação convencional e necessidade de funcionamento contínuo.

A arquitetura experimental foi composta por duas unidades principais. O nó transmissor (TX), totalmente autoalimentado por um painel solar flexível, operava sem baterias ou alimentação externa. Equipado com microcontrolador STM32 e sensores dedicados, o módulo realizava medições de tensão, corrente, potência e temperatura do painel fotovoltaico, além da iluminância ambiente. As variáveis eram adquiridas e organizadas em *frames* transmitidos a cada cinco segundos por meio de comunicação LoRa em 2,4 GHz, priorizando eficiência energética e alcance estendido.

O nó receptor (RX), alimentado por bateria, recebia os pacotes LoRa e os encaminhava via interface UART ao *datalogger* baseado em ESP32-S3. Nesta etapa, o sistema desenvolvido assumia papel central, realizando o processamento dos dados recebidos e implementando estratégia de registro duplo: armazenamento local em cartão SD e publicação simultânea via Wi-Fi para servidor remoto por meio do protocolo MQTT. Essa abordagem introduziu redundância operacional, reduzindo o risco de perda de informação e permitindo visualização remota em tempo real por meio de painel *web*.

Figura 27: Sistema de monitoramento fotovoltaico

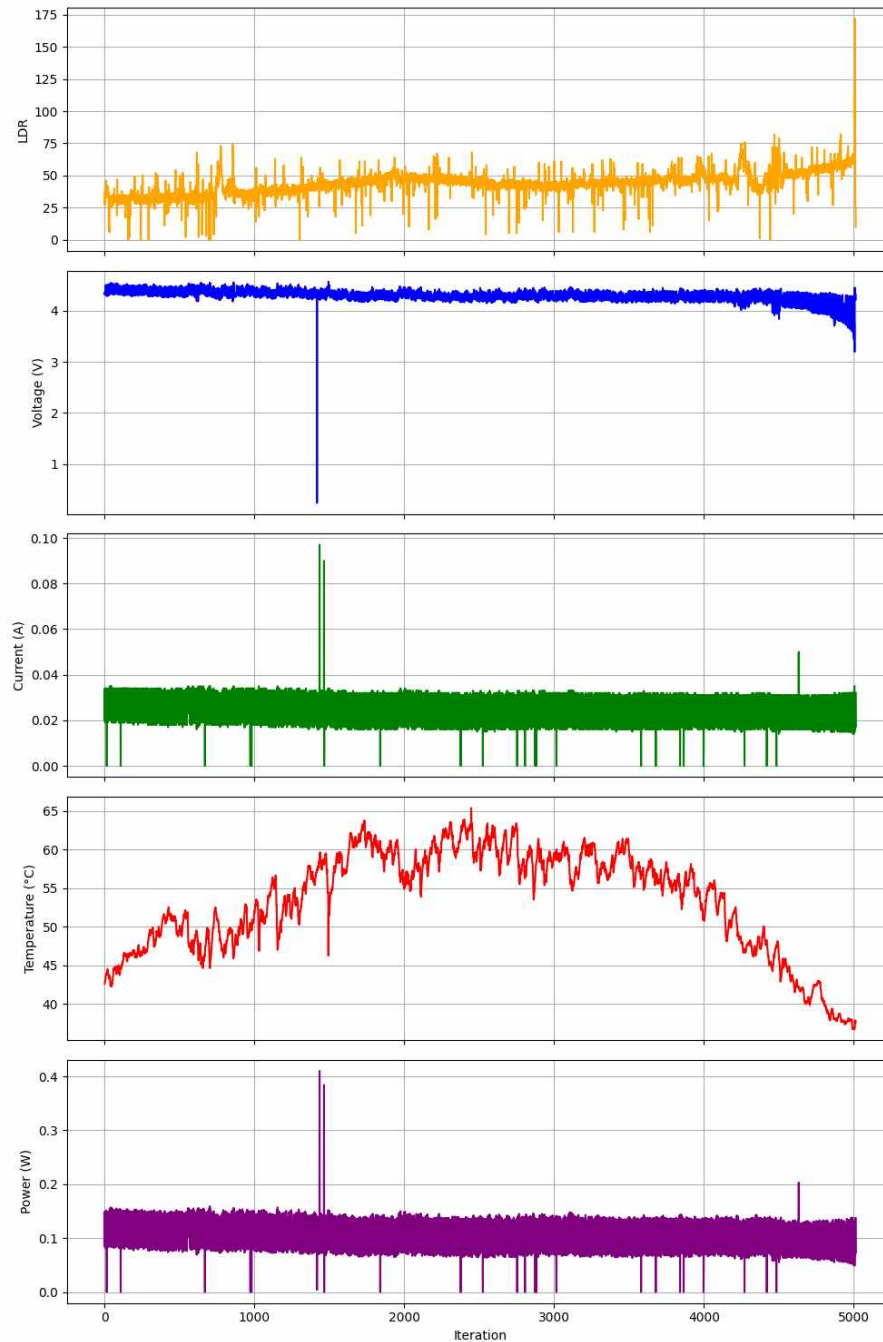


Fonte: Adaptado de [17] (2025).

Os testes foram conduzidos ao ar livre, sob condições de céu aberto, permitindo a observação do comportamento do sistema ao longo de todo o período de incidência solar. O módulo transmissor operou de forma autônoma durante todo o intervalo de iluminação natural, sustentado exclusivamente pela energia colhida pelo painel. A recepção via UART, o armazenamento em memória externa e a publicação MQTT ocorreram de maneira contínua e estável, sem registro de perda de pacotes durante os ensaios. Os dados

obtidos refletiram de forma coerente a variação diária da irradiância solar, da temperatura do módulo e das grandezas elétricas medidas, evidenciando a consistência da cadeia completa de aquisição, transmissão e registro [17].

Figura 28: Variáveis aqisitadas pelo data logger durante a operação



Fonte: Autoria própria (2026).

Os resultados deste caso de teste demonstram a viabilidade do sistema para moni-

toramento contínuo de instalações fotovoltaicas distribuídas, validando tanto sua arquitetura de *software* quanto sua robustez operacional em campo. Além disso, os dados e análises decorrentes deste experimento foram consolidados em artigo técnico aceito para apresentação na IEEE Applied Power Electronics Conference (APEC), reforçando a relevância técnica e científica da solução desenvolvida.

4.2 Caso de Teste 2: Monitoramento e Interação Remota com Sistema de Câmara Fria

O segundo caso de teste teve como objetivo avaliar a capacidade do *datalogger* de operar como interface de monitoramento e controle em um sistema remoto pré-montado, desprovido de acesso direto à internet. Diferentemente do primeiro ensaio, cujo foco estava na aquisição e publicação contínua de dados ambientais, este experimento buscou validar a comunicação bidirecional, permitindo não apenas o registro das variáveis do sistema, mas também a interação ativa por meio do envio de comandos remotos.

O sistema monitorado consistia em uma unidade de resfriamento de câmara controlada por um microcontrolador STM32U585. O dispositivo possuía circuito dedicado para leitura de temperatura interna e circuito de atuação baseado em módulo Peltier, responsável pela regulação térmica. Em seu *firmware* embarcado, havia um controlador PID implementado para manter a temperatura em torno de um valor de referência configurável. O controlador transmitia continuamente, via interface UART, informações de estado contendo variáveis como temperatura atual, *setpoint*, erro de controle e sinal de atuação aplicado ao módulo Peltier.

O *datalogger* foi conectado diretamente à interface serial do controlador, assumindo a função de *gateway* local. Os dados recebidos eram processados, registrados em cartão SD e organizados para posterior análise. Como o sistema não possuía conectividade própria, o *datalogger* passou a desempenhar também o papel de ponte de comunicação remota, permitindo que comandos fossem enviados ao controlador por meio da infraestrutura MQTT sempre que houvesse conexão disponível.

O protocolo de controle adotado baseava-se em comandos no formato AT. O controlador aguardava instruções contendo o estado operacional do sistema (ON ou OFF) e, quando ativado, o novo valor de temperatura de referência a ser atingido. Ao receber comandos provenientes da rede, o *datalogger* os interpretava e os encaminhava via UART ao

STM32U585, estabelecendo um fluxo bidirecional completo: aquisição contínua de dados do sistema térmico e reconfiguração remota de seus parâmetros de operação.

Os testes foram conduzidos em operação contínua da câmara fria, permitindo observar a dinâmica do controlador PID sob diferentes valores de referência. As variáveis registradas evidenciaram o comportamento típico de sistemas de controle térmico, com redução gradual do erro até estabilização em torno do *setpoint* definido. A troca de comandos remotos mostrou-se estável, sem inconsistências na recepção ou execução das instruções, demonstrando a capacidade do *datalogger* de integrar monitoramento e supervisão ativa mesmo em sistemas originalmente não conectados.

Este caso de teste valida a aplicação do *datalogger* como elemento intermediário de supervisão e controle em sistemas embarcados isolados, ampliando sua funcionalidade além do registro passivo de dados e consolidando sua utilização como *gateway* inteligente para dispositivos industriais ou laboratoriais.

4.3 Caso de Teste 3: Coleta Remota de *Logs* UART em Sistema de Monitoramento Inercial

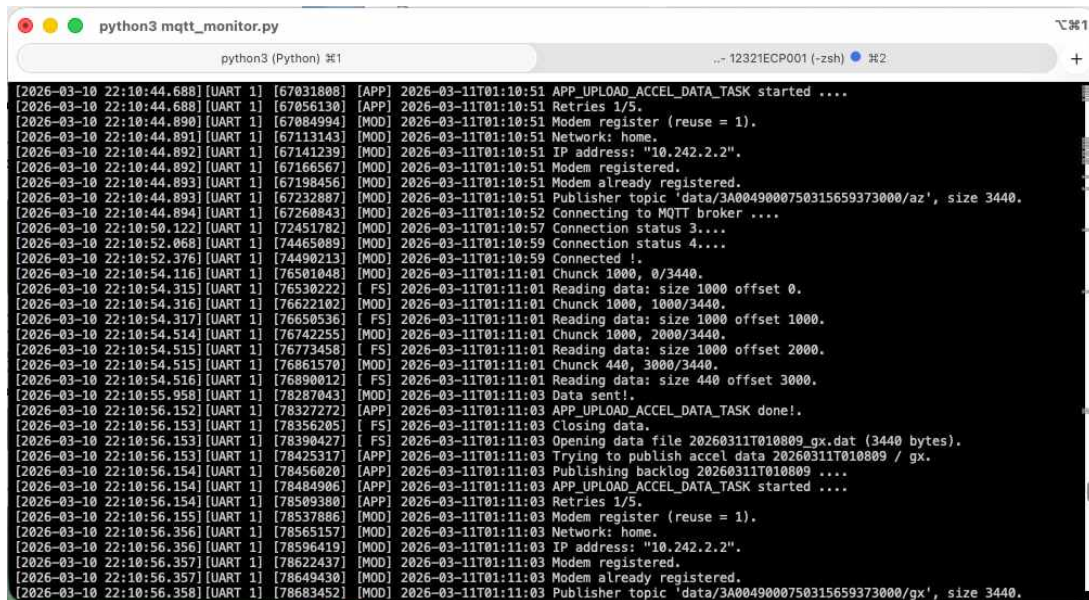
O terceiro caso de teste consistiu na utilização do *datalogger* como ferramenta de coleta e transmissão remota de *logs* de depuração em um sistema embarcado submetido a condições de operação em campo. Diferentemente dos experimentos anteriores, neste cenário o dispositivo não foi utilizado diretamente como fonte primária de aquisição de sensores, mas sim como plataforma auxiliar de monitoramento e registro de dados gerados por outro sistema.

O sistema monitorado era um dispositivo embarcado destinado à aquisição de formas de onda provenientes de sensores inerciais, especificamente acelerômetros e giroscópios. Esse equipamento realizava a captura e transmissão dos dados por meio de rede celular LTE-M, sendo projetado para operar em ambiente externo sob exposição direta ao sol. Durante o processo de validação em campo, tornou-se necessário acompanhar os *logs* de depuração enviados pela interface UART do sistema, utilizados para verificar o comportamento interno do *firmware* e o funcionamento dos algoritmos de aquisição e transmissão.

Entretanto, devido às condições de operação do dispositivo, instalado em local exposto e distante da infraestrutura convencional de desenvolvimento, a utilização de

um computador diretamente conectado à interface serial tornava-se impraticável. Nesse contexto, o *datalogger* desenvolvido neste trabalho foi empregado como uma solução intermediária para captura e transmissão remota desses *logs*.

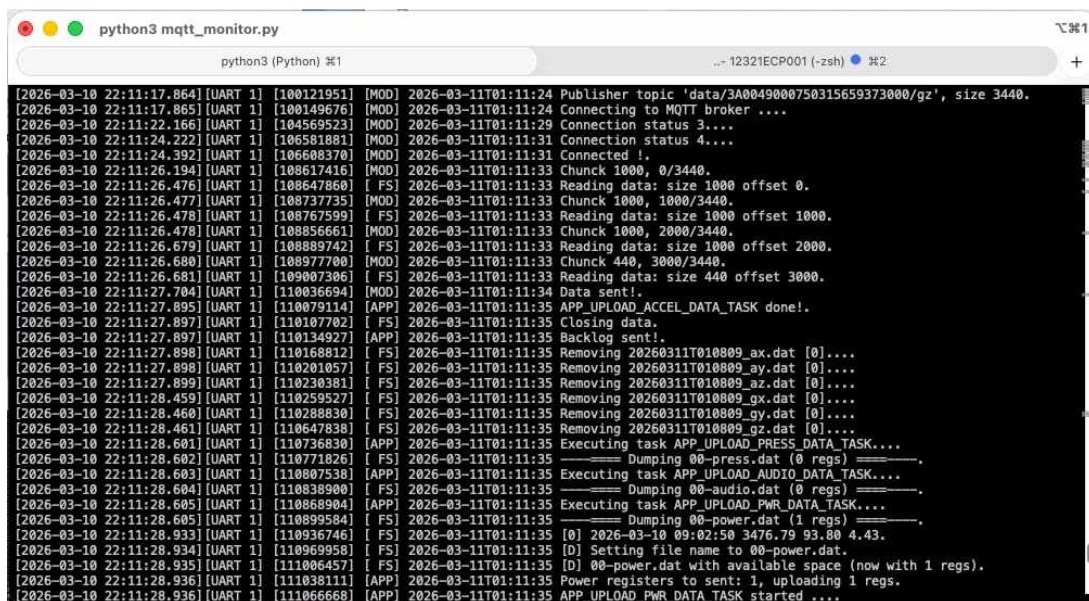
Figura 29: Extração de *logs* remotos pelo *datalogger*



```
python3 mqtt_monitor.py
python3 (Python) #1
... 12321ECP001 (-zsh) #2
[2026-03-10 22:10:44.688] [UART 1] [67031808] [APP] 2026-03-11T01:10:51 APP_UPLOAD_ACCEL_DATA_TASK started ....
[2026-03-10 22:10:44.688] [UART 1] [67056130] [APP] 2026-03-11T01:10:51 Retries 1/5.
[2026-03-10 22:10:44.890] [UART 1] [67084994] [MOD] 2026-03-11T01:10:51 Modem register (reuse = 1).
[2026-03-10 22:10:44.891] [UART 1] [67113143] [MOD] 2026-03-11T01:10:51 Network: home.
[2026-03-10 22:10:44.892] [UART 1] [67141239] [MOD] 2026-03-11T01:10:51 IP address: "10.242.2.2".
[2026-03-10 22:10:44.892] [UART 1] [67166567] [MOD] 2026-03-11T01:10:51 Modem registered.
[2026-03-10 22:10:44.893] [UART 1] [67198456] [MOD] 2026-03-11T01:10:51 Modem already registered.
[2026-03-10 22:10:44.894] [UART 1] [67232887] [MOD] 2026-03-11T01:10:51 Publisher topic 'data/3A0049000750315659373000/az', size 3440.
[2026-03-10 22:10:50.122] [UART 1] [72451782] [MOD] 2026-03-11T01:10:52 Connecting to MQTT broker ....
[2026-03-10 22:10:52.068] [UART 1] [74465089] [MOD] 2026-03-11T01:10:52 Connection status 3....
[2026-03-10 22:10:52.376] [UART 1] [74490213] [MOD] 2026-03-11T01:10:52 Connection status 4....
[2026-03-10 22:10:54.116] [UART 1] [76501048] [MOD] 2026-03-11T01:11:01 Connected !.
[2026-03-10 22:10:54.315] [UART 1] [76530222] [FS] 2026-03-11T01:11:01 Chunk 1000, 0/3440.
[2026-03-10 22:10:54.316] [UART 1] [76622102] [MOD] 2026-03-11T01:11:01 Reading data: size 1000 offset 0.
[2026-03-10 22:10:54.317] [UART 1] [76650536] [FS] 2026-03-11T01:11:01 Chunk 1000, 1000/3440.
[2026-03-10 22:10:54.514] [UART 1] [76742255] [MOD] 2026-03-11T01:11:01 Reading data: size 1000 offset 1000.
[2026-03-10 22:10:54.515] [UART 1] [76773458] [FS] 2026-03-11T01:11:01 Chunk 1000, 2000/3440.
[2026-03-10 22:10:54.515] [UART 1] [76861570] [MOD] 2026-03-11T01:11:01 Reading data: size 1000 offset 2000.
[2026-03-10 22:10:54.516] [UART 1] [76890012] [FS] 2026-03-11T01:11:01 Chunk 440, 3000/3440.
[2026-03-10 22:10:55.958] [UART 1] [78287043] [MOD] 2026-03-11T01:11:03 Data sent!.
[2026-03-10 22:10:56.152] [UART 1] [78327272] [APP] 2026-03-11T01:11:03 APP_UPLOAD_ACCEL_DATA_TASK done!.
[2026-03-10 22:10:56.153] [UART 1] [78356205] [FS] 2026-03-11T01:11:03 Closing data.
[2026-03-10 22:10:56.153] [UART 1] [78390427] [FS] 2026-03-11T01:11:03 Opening data file 20260311T010809_gx.dat (3440 bytes).
[2026-03-10 22:10:56.153] [UART 1] [78425317] [APP] 2026-03-11T01:11:03 Trying to publish accel data 20260311T010809 / gx.
[2026-03-10 22:10:56.154] [UART 1] [78456020] [APP] 2026-03-11T01:11:03 Publishing backlog 20260311T010809 ....
[2026-03-10 22:10:56.154] [UART 1] [78484906] [APP] 2026-03-11T01:11:03 APP_UPLOAD_ACCEL_DATA_TASK started ....
[2026-03-10 22:10:56.154] [UART 1] [78509380] [APP] 2026-03-11T01:11:03 Retries 1/5.
[2026-03-10 22:10:56.155] [UART 1] [78537886] [MOD] 2026-03-11T01:11:03 Modem register (reuse = 1).
[2026-03-10 22:10:56.356] [UART 1] [78565157] [MOD] 2026-03-11T01:11:03 Network: home.
[2026-03-10 22:10:56.356] [UART 1] [78596419] [MOD] 2026-03-11T01:11:03 IP address: "10.242.2.2".
[2026-03-10 22:10:56.357] [UART 1] [78622437] [MOD] 2026-03-11T01:11:03 Modem registered.
[2026-03-10 22:10:56.357] [UART 1] [78649430] [MOD] 2026-03-11T01:11:03 Modem already registered.
[2026-03-10 22:10:56.358] [UART 1] [78683452] [MOD] 2026-03-11T01:11:03 Publisher topic 'data/3A0049000750315659373000/gx', size 3440.
```

Fonte: Autoria própria (2026).

Figura 30: Extração de *logs* remotos pelo *datalogger*



```
python3 mqtt_monitor.py
python3 (Python) #1
... 12321ECP001 (-zsh) #2
[2026-03-10 22:11:17.864] [UART 1] [100121951] [MOD] 2026-03-11T01:11:24 Publisher topic 'data/3A0049000750315659373000/gz', size 3440.
[2026-03-10 22:11:17.865] [UART 1] [100149676] [MOD] 2026-03-11T01:11:24 Connecting to MQTT broker ....
[2026-03-10 22:11:22.166] [UART 1] [104569523] [MOD] 2026-03-11T01:11:29 Connection status 3....
[2026-03-10 22:11:24.222] [UART 1] [106581881] [MOD] 2026-03-11T01:11:31 Connection status 4....
[2026-03-10 22:11:24.392] [UART 1] [106608370] [MOD] 2026-03-11T01:11:31 Connected !.
[2026-03-10 22:11:26.194] [UART 1] [108617416] [MOD] 2026-03-11T01:11:33 Chunk 1000, 0/3440.
[2026-03-10 22:11:26.476] [UART 1] [108647860] [FS] 2026-03-11T01:11:33 Reading data: size 1000 offset 0.
[2026-03-10 22:11:26.477] [UART 1] [108737735] [MOD] 2026-03-11T01:11:33 Chunk 1000, 1000/3440.
[2026-03-10 22:11:26.478] [UART 1] [108767599] [FS] 2026-03-11T01:11:33 Reading data: size 1000 offset 1000.
[2026-03-10 22:11:26.478] [UART 1] [108856661] [MOD] 2026-03-11T01:11:33 Chunk 1000, 2000/3440.
[2026-03-10 22:11:26.679] [UART 1] [108889742] [FS] 2026-03-11T01:11:33 Reading data: size 1000 offset 2000.
[2026-03-10 22:11:26.680] [UART 1] [108977700] [MOD] 2026-03-11T01:11:33 Chunk 440, 3000/3440.
[2026-03-10 22:11:26.681] [UART 1] [109007306] [FS] 2026-03-11T01:11:33 Reading data: size 440 offset 3000.
[2026-03-10 22:11:27.704] [UART 1] [110036694] [MOD] 2026-03-11T01:11:34 Data sent!.
[2026-03-10 22:11:27.895] [UART 1] [110079114] [APP] 2026-03-11T01:11:35 APP_UPLOAD_ACCEL_DATA_TASK done!.
[2026-03-10 22:11:27.897] [UART 1] [110107702] [FS] 2026-03-11T01:11:35 Closing data.
[2026-03-10 22:11:27.897] [UART 1] [110134927] [APP] 2026-03-11T01:11:35 Backlog sent!.
[2026-03-10 22:11:27.898] [UART 1] [110168812] [FS] 2026-03-11T01:11:35 Removing 20260311T010809_ax.dat [0]....
[2026-03-10 22:11:27.898] [UART 1] [110201057] [FS] 2026-03-11T01:11:35 Removing 20260311T010809_ay.dat [0]....
[2026-03-10 22:11:27.899] [UART 1] [110230381] [FS] 2026-03-11T01:11:35 Removing 20260311T010809_az.dat [0]....
[2026-03-10 22:11:28.459] [UART 1] [110259527] [FS] 2026-03-11T01:11:35 Removing 20260311T010809_gx.dat [0]....
[2026-03-10 22:11:28.460] [UART 1] [110288830] [FS] 2026-03-11T01:11:35 Removing 20260311T010809_gy.dat [0]....
[2026-03-10 22:11:28.461] [UART 1] [110647838] [FS] 2026-03-11T01:11:35 Removing 20260311T010809_gz.dat [0]....
[2026-03-10 22:11:28.601] [UART 1] [110736830] [APP] 2026-03-11T01:11:35 Executing task APP_UPLOAD_PRESS_DATA_TASK....
[2026-03-10 22:11:28.602] [UART 1] [110771826] [FS] 2026-03-11T01:11:35 Dumping 00-power.dat (0 regs)....
[2026-03-10 22:11:28.603] [UART 1] [110807538] [APP] 2026-03-11T01:11:35 Executing task APP_UPLOAD_AUDIO_DATA_TASK....
[2026-03-10 22:11:28.604] [UART 1] [110838900] [FS] 2026-03-11T01:11:35 Dumping 00-audio.dat (0 regs)....
[2026-03-10 22:11:28.605] [UART 1] [110869904] [APP] 2026-03-11T01:11:35 Executing task APP_UPLOAD_PWR_DATA_TASK....
[2026-03-10 22:11:28.605] [UART 1] [110899584] [FS] 2026-03-11T01:11:35 Dumping 00-power.dat (1 regs)....
[2026-03-10 22:11:28.933] [UART 1] [110936746] [FS] 2026-03-11T01:11:35 [0] 2026-03-10 09:02:50 3476.79 92.80 4.43.
[2026-03-10 22:11:28.934] [UART 1] [110969958] [FS] 2026-03-11T01:11:35 [D] Setting file name to 00-power.dat.
[2026-03-10 22:11:28.935] [UART 1] [111006457] [FS] 2026-03-11T01:11:35 [D] 00-power.dat with available space (now with 1 regs).
[2026-03-10 22:11:28.936] [UART 1] [111038111] [APP] 2026-03-11T01:11:35 Power registers to sent: 1, uploading 1 regs.
[2026-03-10 22:11:28.936] [UART 1] [111066668] [APP] 2026-03-11T01:11:35 APP_UPLOAD_PWR_DATA_TASK started ....
```

Fonte: Autoria própria (2026).

O dispositivo foi conectado à interface UART do sistema monitorado, passando a atuar como um coletor de dados serial. As mensagens de depuração recebidas eram processadas e registradas localmente, além de serem encaminhadas via rede por meio do protocolo MQTT. Dessa forma, tornou-se possível acompanhar remotamente o fluxo de mensagens geradas pelo sistema em tempo real, sem necessidade de modificações significativas no *firmware* do equipamento monitorado.

Essa estratégia permitiu transformar a saída de depuração UART em um canal de telemetria remoto, possibilitando a análise do comportamento do sistema durante sua operação em campo. Além de facilitar o processo de diagnóstico e validação do dispositivo monitorado, o experimento demonstrou a flexibilidade do *datalogger* desenvolvido, evidenciando sua capacidade de atuar não apenas como registrador de dados, mas também como ferramenta de apoio ao desenvolvimento e depuração de sistemas embarcados em ambientes de difícil acesso.

Os resultados obtidos reforçam a utilidade da arquitetura proposta em cenários práticos de engenharia, nos quais a observabilidade do sistema é frequentemente limitada pelas condições físicas de instalação. Nesse contexto, a possibilidade de coletar e transmitir remotamente *logs* de depuração amplia significativamente a capacidade de diagnóstico e manutenção de dispositivos embarcados operando em campo.

4.4 Testes de Estresse do Sistema

Com o objetivo de avaliar os limites operacionais do *datalogger* e verificar sua robustez sob condições de alta atividade, foram realizados testes de estresse envolvendo múltiplas interfaces de entrada operando simultaneamente. Esses ensaios buscaram reproduzir cenários em que diferentes fontes de dados geram eventos concorrentes, exigindo do sistema capacidade adequada de processamento, organização e registro das informações.

O experimento foi conduzido utilizando três interfaces UART e quatro entradas GPIO operando em paralelo, o *clock* do sistema foi ajustado para 160 MHz, sendo um misto entre consumo e eficiência. As interfaces UART foram configuradas para receber fluxos contínuos de dados provenientes de dispositivos externos, enquanto os pinos GPIO foram excitados por sinais digitais periódicos com frequência controlada, simulando eventos externos de aquisição.

Durante os ensaios com três interfaces UART ativas simultaneamente, o sistema

apresentou funcionamento estável para taxas de transmissão de até 230400 baud em cada canal. Nessas condições, os dados recebidos foram corretamente processados e registrados, sem ocorrência de perdas de informação ou inconsistências no fluxo de armazenamento. Em um cenário adicional, utilizando apenas uma interface UART, o sistema foi capaz de operar próximo ao limite prático do barramento serial, atingindo taxas próximas de 1 Mbps. Esse resultado evidencia que o limite observado no cenário com múltiplos canais está associado principalmente à concorrência de eventos e ao processamento simultâneo das diferentes interfaces.

Para as entradas digitais, os testes foram realizados com quatro pinos GPIO submetidos a sinais periódicos com frequência crescente. O sistema apresentou detecção confiável das transições de borda para frequências de até aproximadamente 500 Hz por canal quando todos os quatro pinos estavam ativos simultaneamente. Em um cenário com apenas um canal de entrada, foi possível operar com frequências próximas de 1 kHz mantendo a detecção correta dos eventos.

Esse limite é considerado adequado para a classe de aplicação proposta, uma vez que, para frequências mais elevadas de eventos digitais, a própria fabricante do microcontrolador recomenda o uso de periféricos dedicados de contagem de pulsos, como o módulo *Pulse Counter* (PCNT) [14]. O periférico PCNT é projetado especificamente para contagem de eventos de alta frequência em sinais digitais, realizando essa tarefa por meio de hardware dedicado e reduzindo significativamente a carga de processamento do núcleo principal. Diferentemente da abordagem baseada em interrupções de GPIO, o PCNT permite acumular contagens de pulsos diretamente no hardware e gerar interrupções apenas quando determinados limiares são atingidos, sendo mais apropriado para aplicações que envolvam medições de frequência ou contagem intensiva de pulsos.

Os resultados obtidos indicam que a arquitetura de *firmware*, baseada em interrupções leves e processamento posterior por tarefas do sistema operacional, é capaz de lidar de forma eficiente com múltiplas fontes concorrentes de dados. Mesmo sob condições de elevada atividade nas interfaces de comunicação e de entrada digital, o sistema manteve comportamento estável e previsível, demonstrando capacidade adequada para aplicações que envolvam aquisição simultânea de dados provenientes de diferentes periféricos.

5 CONCLUSÃO

O desenvolvimento do *datalogger* resultou em um sistema modular e funcional, focado na integração entre aquisição de dados, armazenamento local e comunicação em rede. Para atingir a estabilidade necessária, foi preciso estruturar o *firmware* em camadas e implementar o desacoplamento entre as rotinas de interrupção e o processamento principal. Essa organização tornou-se um requisito fundamental para garantir a previsibilidade do sistema e viabilizar manutenções futuras.

Atingir esse resultado, no entanto, envolveu desafios técnicos significativos que exigiram revisões e refatorações constantes da arquitetura. A definição do formato do frame binário para o armazenamento no cartão SD, por exemplo, demandou iterações sucessivas para encontrar um equilíbrio viável entre a eficiência de escrita e a garantia de integridade dos dados. Além disso, o gerenciamento de recursos do microcontrolador, englobando a alocação de memória, dimensionamento de filas, *buffers* e o tratamento de conexões de rede, provou-se uma etapa complexa e crítica para evitar falhas em operação contínua.

Durante a resolução desses problemas, as decisões de projeto precisaram alinhar as restrições de *hardware* com as necessidades do cliente final. A concepção da arquitetura global foi guiada pelo objetivo de entregar um produto configurável e intuitivo, o que adicionou uma camada extra de complexidade ao desenvolvimento do firmware para mascarar as dificuldades técnicas do usuário.

Em suma, o projeto exigiu um balanço contínuo entre os rigores da engenharia de sistemas embarcados e a usabilidade prática. O resultado entrega uma plataforma estável e alinhada aos requisitos propostos, estabelecendo uma base de código organizada que permite a acomodação de novas funcionalidades e a adaptação para diferentes cenários de aplicação no futuro. Todos os arquivos desenvolvidos no trabalho, *hardware*, *firmware* e ferramentas de *software* podem ser encontrados em [18].

Como continuidade do trabalho, prevê-se a implementação de atualização remota de firmware (OTA), incluindo o desenvolvimento de uma ponte que permita a distribuição das imagens para múltiplos dispositivos. Paralelamente, pretende-se expandir o sistema de decodificação para suportar outros protocolos, como I2C e SPI, aumentando a capacidade de interpretar e registrar dados provenientes de diferentes periféricos.

REFERÊNCIAS BIBLIOGRÁFICAS

- [1] AXELROD, A. *Complete Guide to Test Automation: Techniques, Practices, and Patterns for Building and Maintaining Effective Software Projects*. [S.l.]: Apress, 2018.
- [2] RICHARDS, M.; FORD, N. *Fundamentals of software architecture*. [S.l.]: O'Reilly, 2020.
- [3] Espressif Systems. *ESP32-S3 Series Datasheet*. [S.l.], 2023. Dual-core Xtensa LX7, Wi-Fi, Bluetooth, IO/MUX, SDMMC. Available at: <<https://www.espressif.com>>.
- [4] SD ASSOCIATION. *SD Specifications Part 1 Physical Layer Simplified Specification*. 2022. SD, SDHC, SDXC bus architecture and protocol.
- [5] STALLINGS, W. *Data and Computer Communications*. 10. ed. [S.l.]: Pearson, 2013.
- [6] GRUSIN, M. *Serial Interfaces*. 2026. Acesso em: 8 fev. 2026. Available at: <<https://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi>>.
- [7] VALVANO, J. W. *Embedded Systems: Real-Time Interfacing to Arm Cortex-M Microcontrollers*. [S.l.]: CreateSpace, 2016.
- [8] TECH, P. O. . *Open Drain Output vs. Push-Pull Output*. 2026. Acesso em: 8 fev. 2026. Available at: <<https://open4tech.com/open-drain-output-vs-push-pull-output/>>.
- [9] WANG, F. *Comparison of Open Source MQTT Brokers 2025*. 2025. Acesso em: 7 fev. 2026. Available at: <<https://www.emqx.com/en/blog/a-comprehensive-comparison-of-open-source-mqtt-brokers-in-2023>>.
- [10] LI, Q.; YAO, C. *Real-Time Concepts for Embedded Systems*. [S.l.]: CRC Press, 2003.
- [11] LAPLANTE, P. A.; OVASKA, S. J. *Real-Time Systems Design and Analysis*. 4th. ed. [S.l.]: John Wiley & Sons, 2011.
- [12] BARRY, R. *Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide*. [S.l.], 2024. Acessado em: 8 de fevereiro de 2026. Available at: <https://www.freertos.org/Documentation/RTOS_book.html>.

- [13] TANENBAUM, A. S.; BOS, H. *Modern Operating Systems*. 4th. ed. [S.l.]: Pearson, 2014.
- [14] Espressif Systems. *ESP-IDF Programming Guide v5.2.1 Documentation*. [S.l.], 2024. Acessado em: 8 de fevereiro de 2026. Available at: <<https://docs.espressif.com/projects/esp-idf/en/stable/esp32s3/index.html>>.
- [15] ALMEIDA, M. B. de. *Hardware desenvolvido durante a disciplina Tópicos Especiais em Engenharia de Computação: Desenvolvimento de hardware e firmware para IoT*. 2025. Informação pessoal fornecida pelo autor.
- [16] EMQ Technologies Co., Ltd. *EMQX: A Highly Scalable and Distributed MQTT Messaging Broker for IoT*. 2023. <https://www.emqx.io/>. Disponível em: <https://www.emqx.io/>. Acessado em: 8 de fevereiro de 2026.
- [17] PRESTI, F. N. et al. Concurrent power generation and data transmission: A 2.4 ghz lora solar monitoring system. In: *IEEE Applied Power Electronics Conference and Exposition (APEC)*. [S.l.: s.n.], 2026.
- [18] SOUZA, L. P. B. *Datalogger Modular Baseado em ESP32-S3*. 2026. https://github.com/luizpedrobt/bitbox_V1.0. Repositório contendo hardware, software e firmware do projeto. Acesso em: 24 mar. 2026.