Gabriel Felipe de Souza

Ferramenta Embarcada para Diagnóstico e Simulação de Comunicação Modbus

Uberlândia, MG

2026

Gabriel Felipe de Souza

Ferramenta Embarcada para Diagnóstico e Simulação de Comunicação Modbus

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

Universidade Federal de Uberlândia - UFU
Faculdade de Engenharia Elétrica - FEELT

Orientador Prof. Dr. Daniel Pereira de Carvalho

Uberlândia, MG
2026

A Deus quem me proveu esta vida e todas as maravilhas dela.

À minha família que sempre me apoiou, de todas as formas e por todos os meios.

À minha companheira, que sempre esteve ao meu lado, me motivando e acreditando em meu esforço.

À Universidade Federal de Uberlândia e todos os seus docentes, que me instruíram a chegar até o final desta graduação.

Agradecimentos

Agradeço a Deus e à Nossa Senhora Aparecida, pela força e proteção ao longo dessa caminhada.

Aos meus pais, Delvone Hermes e Eleníce de Fátima, deixo um agradecimento especial por todo o esforço, dedicação e apoio à minha educação, não apenas durante a graduação, mas por toda a minha vida. Sem vocês, nada disso seria possível.

À minha companheira, Amanda Pereira, agradeço com muito carinho por estar ao meu lado em todos os momentos, pela paciência, incentivo e por tornar essa jornada mais leve e significativa.

À Universidade Federal de Uberlândia, em especial à Faculdade de Engenharia Elétrica e aos professores, agradeço pelos ensinamentos e pela formação que me proporcionaram na área de Engenharia de Controle e Automação.

Por fim, agradeço à equipe da ElectricFish Energy Inc., e em especial ao Nélio Junior, pela confiança e pela oportunidade de fazer parte de uma empresa do Vale do Silício, contribuindo de forma importante para a minha formação profissional.

Resumo

Este trabalho apresenta o desenvolvimento de uma ferramenta embarcada, portátil e multifuncional para análise, registro e simulação de comunicações Modbus RTU, voltada à manutenção e ao diagnóstico de redes industriais em campo. Baseado em um microcontrolador STM32F407 com interface gráfica em display touchscreen, o dispositivo opera em três modos selecionáveis em tempo de execução: Mestre, Escravo e Sniffer/datalogger, com suporte às interfaces RS-232 e RS-485. O firmware foi estruturado em uma Camada de Abstração de Hardware (HAL) própria, baseada em ponteiros opacos e injeção de dependências, facilitando a manutenção e a portabilidade do código. Os resultados confirmaram a operação correta nos três modos, com captura íntegra do tráfego e gravação contínua dos quadros em cartão SD com timestamps. Como contribuição, o trabalho entrega uma ferramenta compacta para apoiar atividades de manutenção, diagnóstico, testes e ensino relacionados a redes Modbus RTU.

Palavras-chave: Modbus RTU; Automação Industrial; Comunicação Serial; Sistemas Embarcados; STM32.

Abstract

This work presents the development of a portable and multifunctional embedded tool for Modbus RTU communication analysis, logging, and simulation, aimed at the maintenance and diagnostics of industrial networks in the field. Based on an STM32F407 microcontroller with a touchscreen graphical interface, the device operates in three modes selectable at runtime: master, slave, and sniffer/datalogger, with support for RS-232 and RS-485 interfaces. The firmware was structured around a custom Hardware Abstraction Layer (HAL) based on opaque pointers and dependency injection, easing maintenance and code portability. The results confirmed correct operation in all three modes, with reliable traffic capture and continuous frame logging to an SD card with timestamps. As a contribution, this work delivers a compact tool to support maintenance, diagnostics, testing, and educational activities related to Modbus RTU networks.

Keywords: Modbus RTU; Industrial Automation; Serial Communication; Embedded Systems; STM32.

Lista de ilustrações

Figura 1 – Participação de mercado das redes industriais em 2025. 16
Figura 2 – Etapas de desenvolvimento da automação nas indústrias de processos. 20
Figura 3 – Diferença entre redes antigas e modernas do tipo Fieldbus. 20
Figura 4 – Transmissão síncrona. 21
Figura 5 – Transmissão assíncrona. 21
Figura 6 – Interface e pinagem do conector DB9S para o padrão RS-232. 22
Figura 7 – Estruturas de barramento Full-Duplex (à esquerda) e Half-Duplex (à direita) em redes RS-485. 23
Figura 8 – Arquitetura de rede Modbus, incluindo as implementações sobre MB+, RS-485 e RS-232. 24
Figura 9 – Modelo de dados Modbus com blocos de memória separados. 25
Figura 10 – Modelo de dados Modbus com bloco de memória único. 26
Figura 11 – Estrutura da PDU de requisição e resposta para a função Read Holding Registers. 27
Figura 12 – Estrutura da PDU de requisição e resposta para a função Write Multiple Registers. 27
Figura 13 – Algoritmo iterativo de cálculo de CRC por meio de deslocamento de bits e operações XOR. 29
Figura 14 – Diagrama de blocos do hardware do dispositivo, indicando os barramentos físicos utilizados. 34
Figura 15 – Arquitetura em camadas do firmware desenvolvido. 35
Figura 16 – Estrutura de diretórios do projeto. 36
Figura 17 – Primeira placa de desenvolvimento (STM32-F407VET6 V3.2.1), utilizada na fase inicial de testes da pilha serial. 37
Figura 18 – Segunda placa de desenvolvimento (STM32 F4VE V2.0), adotada na versão final do protótipo. 38
Figura 19 – Módulo de display TFT 3.2 polegadas DevEBox (ILI9341 + XPT2046), utilizado como interface gráfica. 38
Figura 20 – Protótipo final do dispositivo, baseado na placa STM32 F4VE V2.0 com módulos externos MAX485 e MAX3232 acoplados, display ILI9341 e cartão SD instalado. 40
Figura 21 – Fluxograma de captura do Modbus Sniffer, evidenciando as duas fontes de dados (RX e TX) e as duas saídas (log no cartão SD e buffer para a GUI). 52
Figura 22 – Exemplo de teste do hal_uart em modo polling, com transmissão e recepção sincronizadas pelo terminal serial. 58

Figura 23 – Circuito utilizado na primeira placa para controle automático de direção em comunicação RS-485 half-duplex sem pino DE/RE dedicado. . . . 59
Figura 24 – Tela do display ILI9341 dessincronizada após o primeiro ensaio do FSMC. 61
Figura 25 – Exemplo de imagem renderizada com distorção causada pela inversão de endianness. 62
Figura 26 – Ensaio final do sensor de toque XPT2046, com leituras consistentes após correção das anomalias. 64
Figura 27 – Fluxo de teste do modo Escravo configurado na plataforma Node-RED. 65
Figura 28 – Resultado da leitura periódica dos 10 Holding Registers do dispositivo no modo Escravo, observado no Node-RED. 66
Figura 29 – Saída do script Python emulador de escravo durante a validação do Mestre, com quadros recebidos, CRC validado e respostas enviadas. . 67
Figura 30 – Tela do dispositivo no modo Mestre com múltiplas requisições agendadas simultaneamente, para diferentes registros de um escravo. 68
Figura 31 – Trecho do arquivo modbus.log gravado pelo Sniffer no cartão SD, com timestamps, direção dos quadros e conteúdo hexadecimal. 69
Figura 32 – Aba de configuração do barramento serial. 71
Figura 33 – Aba de requisição de funções no modo Mestre. 72
Figura 34 – Aba de modificação de registros no modo Escravo. 73
Figura 35 – Aba do terminal de diagnóstico exibindo os últimos quadros capturados pelo Sniffer. 74

Lista de tabelas

Tabela 1 – Tabelas primárias do modelo de dados Modbus. 25
Tabela 2 – Principais códigos de função Modbus RTU suportados pelo dispositivo. 26

Lista de abreviaturas e siglas

AHB	Advanced High-performance Bus
API	Application Programming Interface
APB1	Advanced Peripheral Bus 1
APB2	Advanced Peripheral Bus 2
BCD	Binary Coded Decimal
BOM	Bill of Materials
BSP	Board Support Package
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
CRC-16	Cyclic Redundancy Check de 16 bits
CRC-32	Cyclic Redundancy Check de 32 bits
DB9S	Conector D-sub de 9 pinos
DCE	Data Circuit-terminating Equipment
DCD	Data Carrier Detect
DE	Driver Enable
DSR	Data Set Ready
DTE	Data Terminal Equipment
DTR	Data Terminal Ready
EMI	Electromagnetic Interference
ESP32	Família de microcontroladores da Espressif
EXTI	External Interrupt
FAT	File Allocation Table
FAT12	FAT versão 12

FAT16	FAT versão 16
FAT32	FAT versão 32
FatFs	Biblioteca FAT Filesystem
FC	Function Code
FMC	Flexible Memory Controller
FSM	Finite State Machine
FSMC	Flexible Static Memory Controller
GND	Ground
GPD	Ground Potential Difference
GPIO	General Purpose Input/Output
GUI	Graphical User Interface
HAL	Hardware Abstraction Layer
HMI	Human Machine Interface
I2C	Inter-Integrated Circuit
IHM	Interface Homem-Máquina
LSB	Least Significant Bit
LSE	Low-Speed External
LSI	Low-Speed Internal
LVGL	Light and Versatile Graphics Library
MB+	Modbus Plus
MISO	Master Input Slave Output
MOSI	Master Output Slave Input
NVIC	Nested Vectored Interrupt Controller
OSI	Open Systems Interconnection
PCB	Printed Circuit Board
PDU	Protocol Data Unit

PLC	Programmable Logic Controller
POSIX	Portable Operating System Interface
PWM	Pulse Width Modulation
RE	Receiver Enable
RI	Ring Indicator
RS-232	Recommended Standard 232
RS-422	Recommended Standard 422
RS-485	Recommended Standard 485
RTC	Real-Time Clock
RTOS	Real-Time Operating System
RTS	Request to Send
RTU	Remote Terminal Unit
RX	Receive
SCADA	Supervisory Control and Data Acquisition
SCK	Serial Clock
SDIO	Secure Digital Input Output
SPI	Serial Peripheral Interface
SS	Slave Select
STM32	Família de microcontroladores da STMicroelectronics
SYSCLK	System Clock
T3.5	Tempo de silêncio Modbus RTU
TC	Transmission Complete
TCP/IP	Transmission Control Protocol / Internet Protocol
TD	Transmitted Data
TTL	Transistor-Transistor Logic
TX	Transmit

UART	Universal Asynchronous Receiver/Transmitter
UL	Unit Load
UNIX	UNIX Time

Sumário

1 INTRODUÇÃO 16
1.1 Justificativas 17
1.2 Objetivos 17
1.2.1 Objetivos Específicos 18

2 REFERENCIAL TEÓRICO 19
2.1 Introdução a Redes Industriais 19
2.1.1 Comunicação Digital 21
2.1.2 RS-232 22
2.1.3 RS-485 22
2.1.4 Protocolo Modbus 24
2.1.4.1 Modelo de dados do Modbus 25
2.1.4.2 Códigos de Função 26
2.1.4.3 Modelo de Endereçamento Modicon 28
2.1.4.4 Verificação de Redundância Cíclica (CRC) 28
2.2 Sistemas Operacionais em Tempo Real (RTOS) 30
2.2.1 Comunicação Serial UART/USART 30
2.2.2 Periféricos do Microcontrolador 30
2.2.2.1 Temporizadores (Timers) 31
2.2.2.2 Interface SPI 31
2.2.2.3 Relógio de Tempo Real (RTC) 31
2.2.2.4 Sistema de Arquivos FAT e Biblioteca FatFs 31
2.2.2.5 Controlador de Memória Estática Flexível (FSMC/FMC) 31

3 METODOLOGIA 33
3.1 Visão Geral do Sistema 33
3.2 Plataforma de Hardware e Componentes Físicos 36
3.3 Arquitetura da Camada HAL 40
3.4 Camada de GPIO (hal_gpio) 41
3.4.1 Mapeamento de Pinos (BSP) 41
3.4.2 Normalização de Níveis Lógicos e Interrupções Externas 41
3.5 Camada UART (hal_uart) 42
3.5.1 Detecção de Fim de Mensagem na UART 42
3.5.2 Controle de Direção Half-Duplex (DE/RE) para RS-485 42
3.6 Camada de Temporização (hal_timer) 43
3.7 Camada de Tempo Global (hal_time) 43

3.8 Camada CRC (hal_crc) 44
3.8.1 Limitação do Periférico CRC do STM32F407 44
3.9 Camada RTC (hal_rtc) 44
3.9.1 Limitação de Resolução em Milissegundos 45
3.10 Camada SPI (hal_spi) 45
3.11 Camada de Armazenamento (hal_storage) 46
3.12 Camada do Display e Controlador Gráfico 46
3.12.1 Aceleração por Mapeamento de Memória (FSMC) 46
3.12.2 Integração do Controlador ILI9341 47
3.13 Dispositivo Escravo (Integração FreeModbus) 47
3.13.1 Camada de Interface Física (Port Layer) 48
3.13.2 Camada de Aplicação e Mapeamento de Memória 48
3.14 Dispositivo Mestre Próprio (ModbusMaster) 48
3.14.1 Agendador (Scheduler) e Fila de Requisições 49
3.14.2 Gerenciamento de Escravos e Estratégia de Backoff 49
3.14.3 Máquina de Estados e Decodificação de Códigos de Função 49
3.14.4 Cache Local de Leituras 50
3.15 Gerenciador Modbus (modbus_manager) 50
3.16 Modbus Sniffer 51
3.16.1 Captura no Fluxo de Recepção (RX) 53
3.16.2 Captura dos Quadros Transmitidos (TX) 53
3.16.3 Desacoplamento por Fila Circular e Duplo Buffer 53
3.17 Interface Gráfica (LVGL) e Controle do Sistema 54
3.17.1 Adaptação de Hardware (Display, Touch e Tick) 54
3.17.2 Configuração Dinâmica e Controle do Sistema 54
3.18 Procedimento de Validação 55

4 RESULTADOS 57
4.1 Validação e Aprimoramento da Camada de Abstração (HAL) . . . 57
4.1.1 Comunicação Serial (UART) e Controle de Fluxo 57
4.1.1.1 Adaptação para Controle de DE/RE no Hardware Final 58
4.1.2 Entradas/Saídas (GPIO) e Temporização 59
4.1.3 Sistema de Arquivos e RTC 59
4.2 Resolução de Falhas Críticas na Integração Gráfica 60
4.2.1 Anomalias no Barramento Paralelo (FSMC) 60
4.2.2 Ruído e Assincronismo no Controlador de Toque (SPI) 63
4.2.3 Integração da Biblioteca LVGL 64
4.3 Validação dos Modos de Operação Modbus 65
4.3.1 Operação como Escravo (Integração FreeModbus) 65
4.3.2 Operação como Mestre Próprio (ModbusMaster) 66

4.3.2.1 Agendamento de Múltiplas Requisições . . . 67
4.3.3 Operação como Sniffer e Datalogger . . . 68
4.3.3.1 Topologia de Captura . . . 68
4.3.3.2 Formato e Capacidade do Log . . . 69
4.3.3.3 Limite de Velocidade Testado . . . 70
4.4 Interface Gráfica Final . . . 70
4.4.1 Aba de Configuração de Rede . . . 70
4.4.2 Aba de Gerenciamento do Mestre . . . 71
4.4.3 Aba de Simulação de Escravo . . . 72
4.4.4 Aba do Terminal de Diagnóstico (Sniffer) . . . 73
4.5 Síntese das Capacidades da Ferramenta . . . 74
4.6 Decisões de Projeto e Limitações da Ferramenta . . . 75
4.6.1 Adoção de Superloop Cooperativo em vez de RTOS . . . 75
4.6.2 Limitações Observadas . . . 75
4.6.3 Síntese dos Resultados . . . 76

5 CONCLUSÃO . . . 77

REFERÊNCIAS BIBLIOGRÁFICAS . . . 79

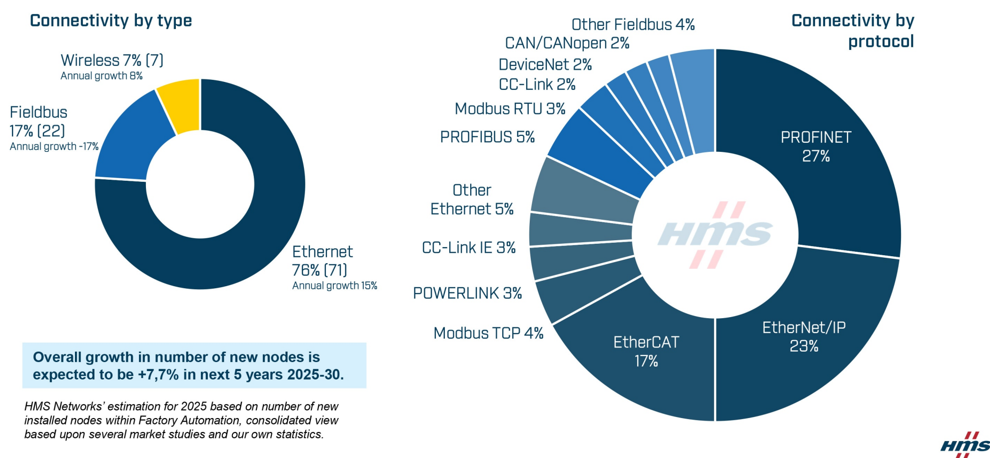
1 Introdução

A comunicação entre dispositivos industriais é um dos pilares da automação moderna. Nesse contexto, o protocolo Modbus, desenvolvido pela Modicon em 1979, consolidou-se como um dos protocolos mais difundidos da automação industrial, destacando-se por sua simplicidade, robustez e facilidade de implementação em diferentes aplicações industriais (SEN, 2014).

Segundo HMS Networks (2025), o Modbus permanece entre os protocolos industriais mais utilizados na automação industrial. O estudo posiciona o Modbus TCP como o quarto protocolo Ethernet industrial mais utilizado (4% do mercado) e o Modbus RTU como o segundo principal protocolo serial (3% do mercado), totalizando 7% de todos os novos nós de rede instalados globalmente.

A Figura 1 apresenta a participação dos principais protocolos de comunicação industrial no mercado global em 2025.

Figura 1 – Participação de mercado das redes industriais em 2025.



Fonte: HMS Networks (2025).

Observa-se que os protocolos baseados em Ethernet industrial dominam as novas instalações, com destaque para PROFINET e EtherNet/IP. Entretanto, o Modbus mantém participação expressiva tanto em aplicações Ethernet quanto seriais, evidenciando sua permanência em sistemas industriais modernos e em aplicações legadas.

Porém, apesar de sua ampla difusão e confiabilidade comprovada, o Modbus RTU apresenta algumas limitações quando comparado a protocolos industriais mais recentes. Sua

estrutura simples, embora vantajosa para compatibilidade e baixo custo de implementação e interpretação técnica, não possui mecanismos nativos de segurança, como autenticação ou criptografia. Além disso, a comunicação baseada no modelo mestre-escravo limita a escalabilidade e cria pontos únicos de falha na rede. Essas características tornam o protocolo mais suscetível a interferências e vulnerabilidades em ambientes interconectados, embora ainda seja amplamente utilizado em redes locais e sistemas isolados, onde sua simplicidade e robustez continuam sendo grandes vantagens (JAKABOCZKI; ADAMKO, 2015).

Neste contexto, o presente trabalho propõe o desenvolvimento de um dispositivo embarcado multifuncional e portátil, projetado para atuar como uma ferramenta de apoio técnico em redes Modbus RTU.

1.1 Justificativas

O protocolo Modbus permanece amplamente utilizado na automação industrial devido à sua simplicidade, robustez e compatibilidade com dispositivos de diferentes fabricantes. Segundo HMS Networks (2025), o protocolo continua presente em uma parcela significativa das novas instalações industriais, demonstrando sua relevância mesmo diante de tecnologias mais recentes.

Entretanto, a manutenção e o diagnóstico de redes Modbus RTU ainda representam desafios para técnicos e engenheiros de campo. Em aplicações industriais, é comum a utilização de dispositivos com diferentes configurações de velocidade, paridade e interfaces físicas, como RS-232 e RS-485, o que aumenta a complexidade na identificação de falhas e na instalação de novos equipamentos.

Atualmente, essas atividades geralmente dependem da utilização de notebooks, conversores USB e softwares específicos de monitoramento. Em muitos casos, essa abordagem reduz a praticidade durante intervenções em campo, principalmente em ambientes com espaço limitado ou de difícil acesso.

Diante desse cenário, este trabalho propõe o desenvolvimento de um dispositivo embarcado portátil e multifuncional para análise, simulação e diagnóstico de redes Modbus RTU. A proposta busca oferecer uma ferramenta compacta e dedicada, capaz de simplificar atividades de manutenção, testes e validação em ambientes industriais e acadêmicos.

1.2 Objetivos

O principal objetivo deste projeto é desenvolver um dispositivo embarcado portátil e multifuncional, baseado em um microcontrolador da família STM32, capaz de simular, analisar, registrar e diagnosticar comunicações Modbus RTU, de forma a agilizar e aprimorar

os processos de implementação, instalação e manutenção de redes industriais ou pequenas plantas que utilizam este protocolo.

1.2.1 Objetivos Específicos

- Implementar uma lógica de comunicação Modbus RTU utilizando a biblioteca FreeModbus integrada ao microcontrolador STM32;
- Desenvolver mecanismos de leitura e escrita em dispositivo de armazenamento externo para registro de mensagens e eventos da comunicação Modbus;
- Desenvolver uma interface gráfica intuitiva para configuração, monitoramento e análise da comunicação Modbus RTU;
- Implementar uma técnica de varredura automática para identificação de dispositivos presentes em uma rede Modbus RTU;
- Elaborar uma placa de circuito impresso integrada para validação funcional e análise de viabilidade do dispositivo proposto.

2 Referencial Teórico

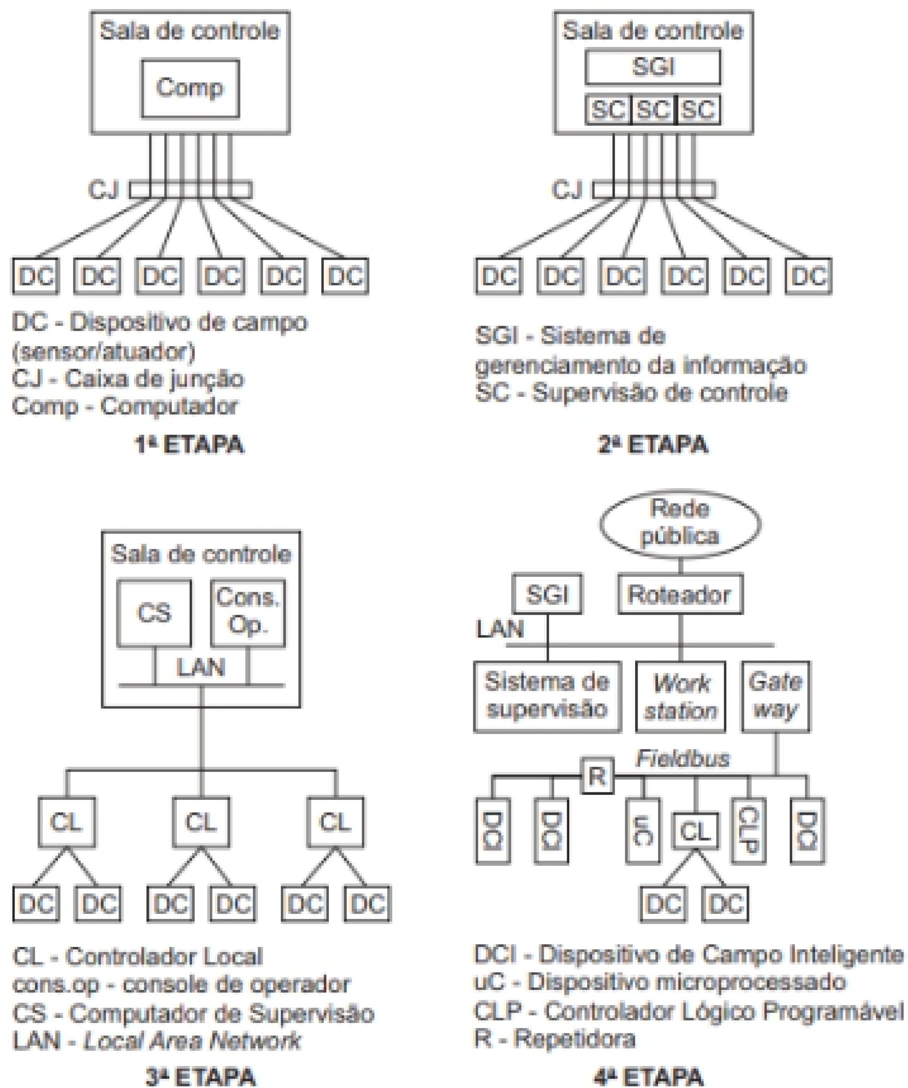
2.1 Introdução a Redes Industriais

As redes industriais constituem um dos pilares da automação moderna, sendo responsáveis pela interligação entre dispositivos de campo, controladores e sistemas de supervisão. Sua evolução está diretamente relacionada ao avanço de componentes eletrônicos, como transistores, circuitos integrados e microprocessadores, que possibilitaram a comunicação eficiente entre máquinas e sistemas computacionais (FILHO, 2014).

Segundo Filho (2014), essa evolução pode ser dividida em quatro etapas principais. A primeira corresponde à comunicação ponto a ponto nos anos 1960, seguida pela arquitetura hierárquica nos anos 1970 e pelos Sistemas Digitais de Controle Distribuído na década de 1980. A quarta etapa marca o surgimento das redes Fieldbus a partir de 1990, que consolidou a comunicação digital no nível de campo, reduzindo cabeamentos e aumentando a integração e o diagnóstico dos sistemas.

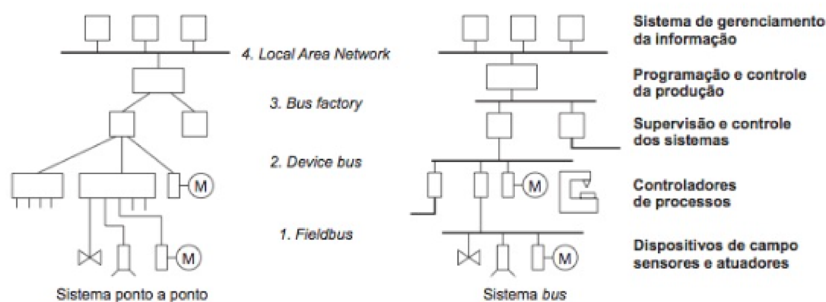
A Figura 2 apresenta essas etapas de desenvolvimento, enquanto a Figura 3 ilustra a transição entre sistemas ponto a ponto e as redes do tipo Fieldbus.

Figura 2 – Etapas de desenvolvimento da automação nas indústrias de processos.



Fonte: Adaptado de Filho (2014).

Figura 3 – Diferença entre redes antigas e modernas do tipo Fieldbus.



Fonte: Adaptado de Filho (2014).

Observa-se nas Figuras 2 e 3 que a evolução das redes industriais caminhou da centralização para a distribuição, substituindo a fiação dedicada por barramentos digitais

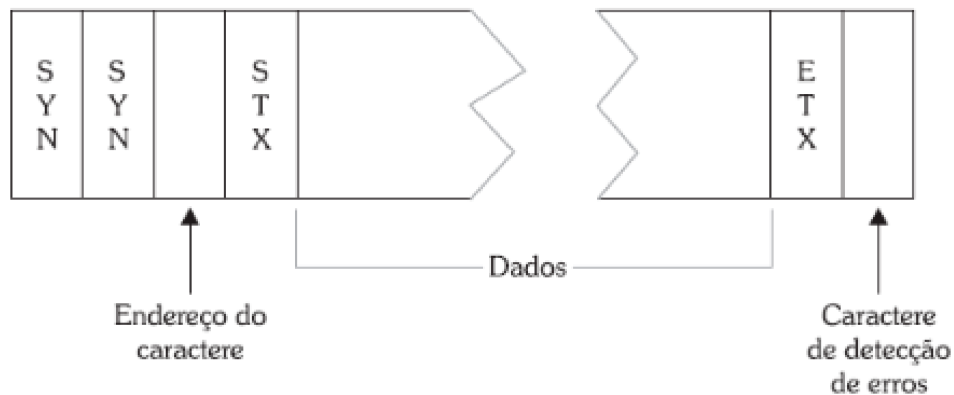
compartilhados. É nesse contexto que protocolos como o Modbus RTU se tornaram fundamentais, ao fornecerem um padrão de comunicação simples e amplamente compatível para interligar dispositivos de campo.

2.1.1 Comunicação Digital

A comunicação digital consiste na transmissão de dados por meio de sinais discretos, geralmente representados por dois níveis de tensão que correspondem aos estados lógicos 0 e 1 (MORAES, 2020).

Segundo Moraes (2020), a transmissão digital pode ocorrer de forma síncrona ou assíncrona. Na transmissão síncrona, os dados são enviados em blocos contínuos, dependendo de um sinal de clock compartilhado entre emissor e receptor. Na transmissão assíncrona, os bytes são transmitidos individualmente, precedidos por um start bit e finalizados por um stop bit, podendo incluir um bit de paridade para verificação de erros. As Figuras 4 e 5 ilustram esses dois modos.

Figura 4 – Transmissão síncrona.



Fonte: Moraes (2020).

Figura 5 – Transmissão assíncrona.

Fonte: Moraes (2020).

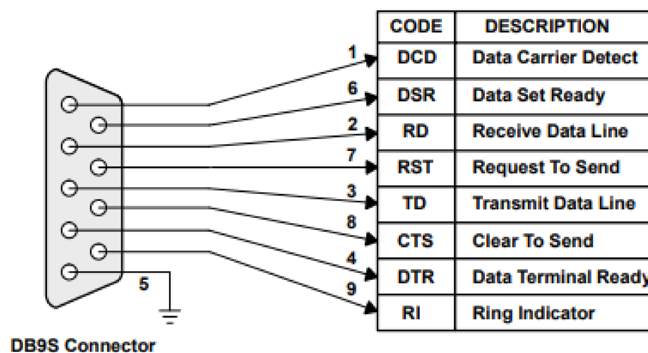
O Modbus RTU adota a transmissão assíncrona, pois ela dispensa um sinal de clock dedicado e simplifica a implementação em sistemas embarcados. Essa escolha é diretamente relevante para o dispositivo desenvolvido neste trabalho, no qual a UART opera sem sincronismo externo, com baudrate configurável pelo usuário.

2.1.2 RS-232

O padrão RS-232 é uma interface serial ponto a ponto que utiliza comunicação single-ended, onde cada sinal é transmitido por um único fio em relação a um terra comum. Por esse motivo, o barramento suporta apenas dois dispositivos e apresenta baixa imunidade a ruídos eletromagnéticos, sendo recomendado para distâncias curtas (Texas Instruments, 2001).

A conexão física mais comum é realizada pelo conector DB9S de 9 pinos, padrão de mercado para essa interface. Em sistemas embarcados modernos, utiliza-se tipicamente apenas três sinais, TX (pino 3), RX (pino 2) e GND (pino 5), dispensando os sinais de controle originalmente definidos para uso com modems analógicos (Texas Instruments, 2001). A Figura 6 apresenta a pinagem padrão do conector DB9S.

Figura 6 – Interface e pinagem do conector DB9S para o padrão RS-232.



Fonte: Texas Instruments (2001).

Apesar de suas limitações em distância e imunidade a ruído, o RS-232 ainda é encontrado em equipamentos industriais legados, o que motivou a inclusão do suporte a esta interface no dispositivo desenvolvido neste trabalho.

2.1.3 RS-485

O padrão RS-485 foi desenvolvido como uma evolução do RS-422, permitindo comunicação multiponto (multidrop) em um barramento compartilhado. Embora seja uma norma estritamente elétrica, que define apenas as características de drivers e receptores, sem estipular conectores ou protocolos de software, o RS-485 tornou-se a interface física padrão

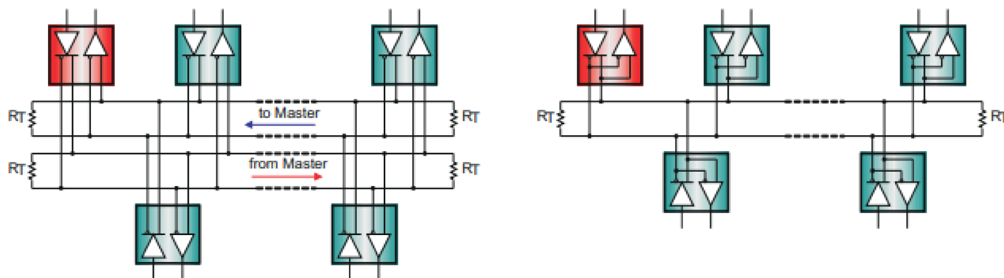
para protocolos de nível superior como Modbus RTU, Profibus e DMX512 (KUGELSTADT, 2021).

O RS-485 utiliza sinalização diferencial balanceada, implementada por meio de um par trançado de fios (denominados A e B). O sinal é representado pela diferença de tensão entre as duas linhas complementares, o que permite que ruídos externos se acoplem igualmente em ambos os condutores e sejam rejeitados pelo receptor. Essa alta imunidade a ruídos viabiliza comunicação confiável em ambientes industriais severos, alcançando distâncias de até 1200 metros (KUGELSTADT, 2021).

Em termos de capacidade de barramento, a norma introduziu o conceito de Carga Unitária (UL, do inglês Unit Load), que representa uma impedância de entrada teórica de aproximadamente 12 kΩ. A especificação original suporta até 32 UL no barramento; contudo, transceptores modernos com frações de carga (como 1/8 UL) elevam essa capacidade teórica para até 256 nós sem necessidade de repetidores (KUGELSTADT, 2021).

O barramento RS-485 pode operar em modo half-duplex ou full-duplex. No modo half-duplex, o mais utilizado em redes Modbus RTU, apenas um par diferencial é empregado, e os dispositivos alternam entre transmissão e recepção. Esse controle de direção é realizado pelos sinais Driver Enable (DE) e Receiver Enable (RE) presentes nos transceptores RS-485, o sinal DE habilita o transmissor e RE habilita o receptor, sendo necessário garantir que apenas um dispositivo transmita por vez no barramento. No modo full-duplex, dois pares diferenciais são utilizados, permitindo transmissão e recepção simultâneas. A Figura 7 ilustra as duas topologias mencionadas.

Figura 7 – Estruturas de barramento Full-Duplex (à esquerda) e Half-Duplex (à direita) em redes RS-485.



Fonte: Kugelstadt (2021).

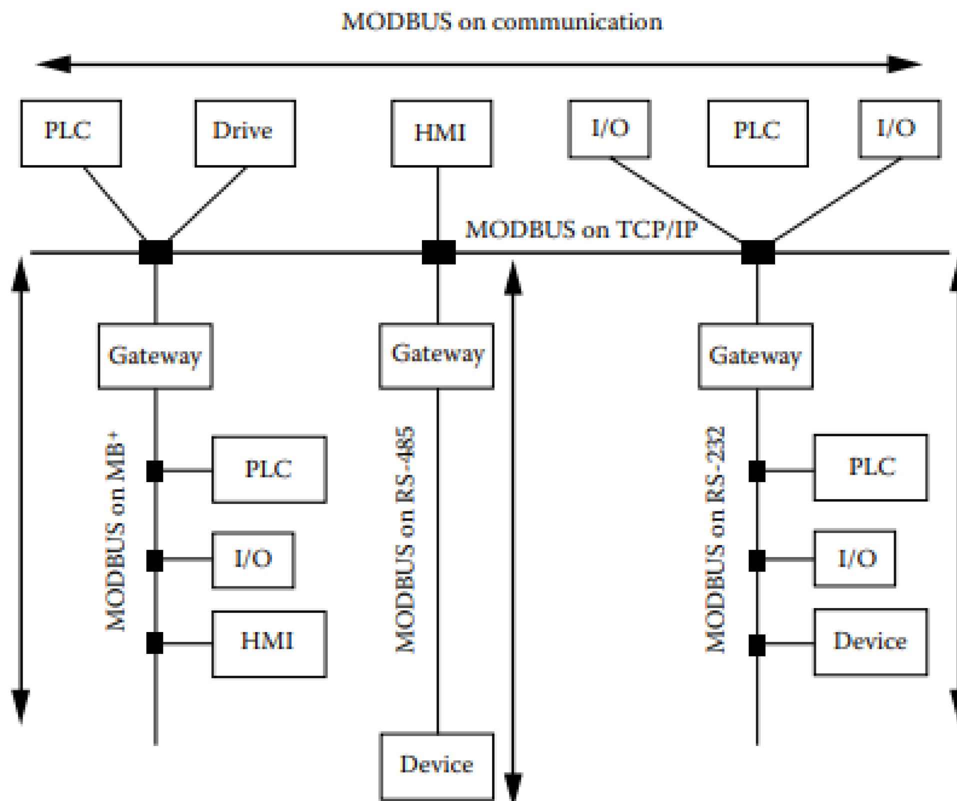
Como evidenciado na Figura 7, a topologia half-duplex com par único é a mais adotada em redes Modbus RTU industriais, pois reduz o custo de cabeamento. O correto gerenciamento dos pinos DE/RE é crítico nessa configuração e foi um dos desafios de hardware enfrentados no desenvolvimento do dispositivo deste trabalho, conforme detalhado no Capítulo 3.

2.1.4 Protocolo Modbus

O protocolo Modbus é um dos padrões mais amplamente utilizados na automação industrial, tendo sido desenvolvido originalmente pela Modicon (atualmente Schneider Electric) em 1979. Trata-se de um protocolo que opera na camada de aplicação (camada 7 do modelo OSI), fornecendo um mecanismo padronizado de comunicação no modelo mestre/escravo (equivalente ao modelo cliente/servidor), o dispositivo mestre inicia todas as requisições de leitura ou escrita, enquanto os dispositivos escravos apenas respondem a essas solicitações (REYNDERS; MACKAY; WRIGHT, 2004).

O protocolo suporta teoricamente até 247 dispositivos escravos e um único mestre, e pode ser implementado sobre diferentes meios físicos, como RS-232, RS-422 ou RS-485, sendo mais comum o uso do RS-485 em ambientes industriais devido à sua imunidade a ruídos (SEN, 2014). A Figura 8 apresenta essa arquitetura de rede.

Figura 8 – Arquitetura de rede Modbus, incluindo as implementações sobre MB+, RS-485 e RS-232.



Fonte: Adaptado de Sen (2014) e Modbus Organization (2012).

Conforme ilustrado na Figura 8, o protocolo Modbus pode ser executado sobre diferentes camadas físicas (RS-232, RS-485 ou Ethernet, no caso do Modbus TCP/IP), mantendo a mesma camada de aplicação. No escopo deste trabalho, o foco é o Modbus RTU sobre RS-485 e RS-232, por serem as interfaces mais comuns em equipamentos

industriais de campo.

2.1.4.1 Modelo de dados do Modbus

O protocolo Modbus organiza os dados internos de um dispositivo em quatro tabelas lógicas, que se distinguem pelo tipo de acesso permitido e pelo tamanho do dado (Modbus Organization, 2012). A Tabela 1 descreve essas tabelas primárias.

Tabela 1 – Tabelas primárias do modelo de dados Modbus.

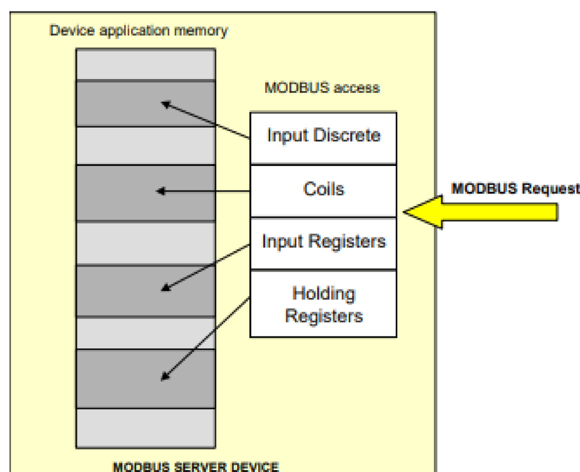
Tabela	**Tipo de dado**	**Tipo de acesso**	**Descrição**
Discrete Inputs	1 bit	Somente leitura	Entradas digitais provenientes do sistema de I/O.
Coils	1 bit	Leitura e escrita	Saídas digitais controláveis pelo mestre.
Input Registers	16 bits	Somente leitura	Valores analógicos provenientes de sensores.
Holding Registers	16 bits	Leitura e escrita	Variáveis internas ou parâmetros de configuração.

Fonte: Modbus Organization (2012).

Uma distinção importante é que, no modelo de dados do usuário, os elementos são numerados de 1 a n. Porém, na PDU (Protocol Data Unit) transmitida na rede, os índices iniciam em zero. Assim, o registro de número X é endereçado na mensagem como X-1 (Modbus Organization, 2012).

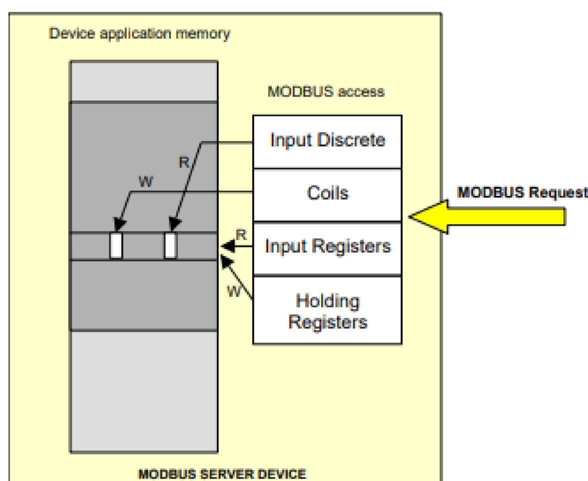
Quanto ao mapeamento interno, o fabricante pode organizar os dados em blocos separados (cada tabela em área de memória distinta) ou em bloco único (todas as tabelas sobrepostas), conforme ilustrado nas Figuras 9 e 10.

Figura 9 – Modelo de dados Modbus com blocos de memória separados.



Fonte: Modbus Organization (2012).

Figura 10 – Modelo de dados Modbus com bloco de memória único.



Fonte: Modbus Organization (2012).

As Figuras 9 e 10 evidenciam que o Modbus oferece flexibilidade ao desenvolvedor na organização interna da memória. O dispositivo desenvolvido neste trabalho adota o modelo de blocos separados, com vetores estáticos independentes para cada tabela primária, o que facilita a validação e o isolamento de acesso.

2.1.4.2 Códigos de Função

Cada requisição Modbus é identificada por um Function Code (FC), um campo de um byte presente na PDU que indica a operação desejada pelo mestre (Modbus Organization, 2012). A Tabela 2 resume os códigos de função suportados pelo dispositivo deste trabalho.

Tabela 2 – Principais códigos de função Modbus RTU suportados pelo dispositivo.

Código	**Nome**	**Operação**
01 (0x01)	Read Coils	Leitura de saídas digitais (bobinas).
02 (0x02)	Read Discrete Inputs	Leitura de entradas digitais.
03 (0x03)	Read Holding Registers	Leitura de registradores de retenção (16 bits).
04 (0x04)	Read Input Registers	Leitura de registradores de entrada (16 bits).
05 (0x05)	Write Single Coil	Escrita em uma única bobina.
06 (0x06)	Write Single Register	Escrita em um único registrador.
15 (0x0F)	Write Multiple Coils	Escrita em múltiplas bobinas consecutivas.
16 (0x10)	Write Multiple Registers	Escrita em múltiplos registradores consecutivos.

Fonte: Modbus Organization (2012).

A Figura 11 apresenta a estrutura da PDU para a função 03 (Read Holding

Registers), uma das mais utilizadas no modo Mestre deste dispositivo, e a Figura 12 ilustra a função 16 (Write Multiple Registers), utilizada para escrita em massa de parâmetros.

Figura 11 – Estrutura da PDU de requisição e resposta para a função Read Holding Registers.

Request

Function code	1 Byte	**0x03**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 125 (0x7D)

Response

Function code	1 Byte	**0x03**
Byte count	1 Byte	2 x **N***
Register value	**N*** x 2 Bytes	

***N** = Quantity of Registers

Error

Error code	1 Byte	**0x83**
Exception code	1 Byte	01 or 02 or 03 or 04

Here is an example of a request to read registers 108 – 110:

Request		**Response**	
Field Name	*(Hex)*	*Field Name*	*(Hex)*
Function	**03**	Function	**03**
Starting Address Hi	**00**	Byte Count	**06**
Starting Address Lo	**6B**	Register value Hi (108)	**02**
No. of Registers Hi	**00**	Register value Lo (108)	**2B**
No. of Registers Lo	**03**	Register value Hi (109)	**00**
		Register value Lo (109)	**00**
		Register value Hi (110)	**00**
		Register value Lo (110)	**64**

Fonte: Modbus Organization (2012).

Figura 12 – Estrutura da PDU de requisição e resposta para a função Write Multiple Registers.

Request

Function code	1 Byte	**0x10**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	0x0001 to 0x007B
Byte Count	1 Byte	2 x **N***
Registers Value	**N*** x 2 Bytes	value

***N** = Quantity of Registers

Response

Function code	1 Byte	**0x10**
Starting Address	2 Bytes	0x0000 to 0xFFFF
Quantity of Registers	2 Bytes	1 to 123 (0x7B)

Fonte: Modbus Organization (2012).

As Figuras 11 e 12 mostram que as PDUs seguem uma estrutura compacta, formada por endereço inicial, quantidade de itens e dados de 16 bits em formato Big-Endian. Esse formato simples e determinístico é um dos fatores que facilitam a implementação do protocolo em microcontroladores com recursos limitados.

2.1.4.3 Modelo de Endereçamento Modicon

Para simplificar a interface com sistemas SCADA e IHMs, a indústria adotou a "Convenção de Endereçamento Modicon", que atribui prefixos numéricos para identificar visualmente a tabela de destino. Os prefixos são 0xxxx para Coils, 1xxxx para Discrete Inputs, 3xxxx para Input Registers e 4xxxx para Holding Registers (Modicon, 1996).

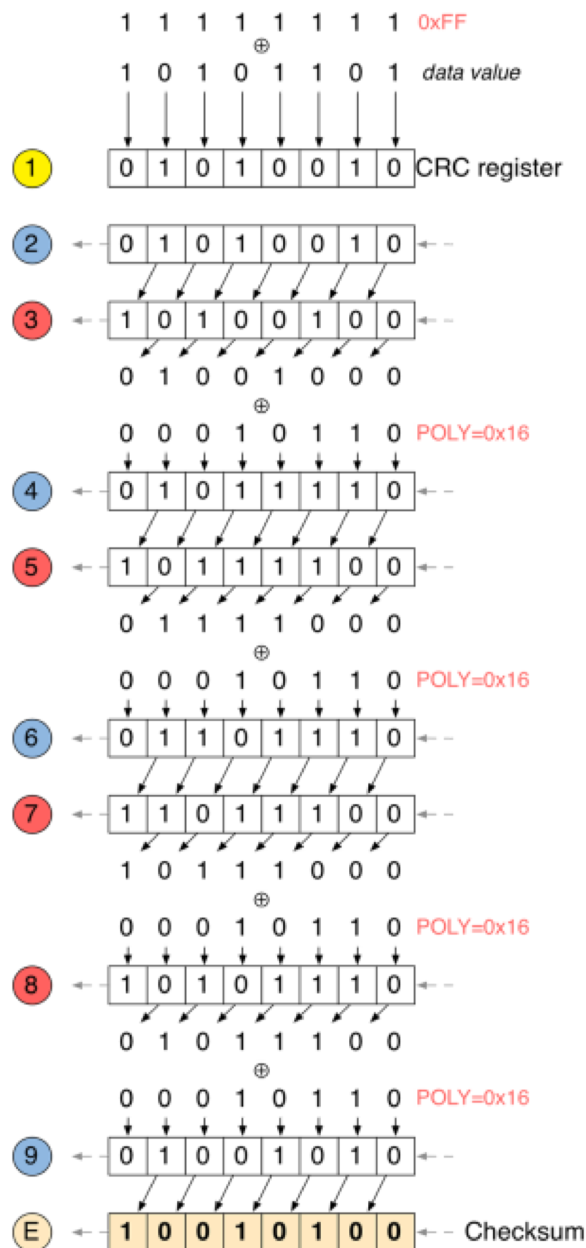
É importante destacar que esses prefixos são apenas marcações visuais na camada de aplicação e não compõem o campo de endereço transmitido fisicamente. A seleção da tabela é determinada exclusivamente pelo código de função inserido na PDU (Modicon, 1996).

2.1.4.4 Verificação de Redundância Cíclica (CRC)

Para garantir a integridade das mensagens transmitidas, o Modbus RTU utiliza a Verificação de Redundância Cíclica de 16 bits (CRC-16). O transmissor calcula um checksum a partir dos bytes da mensagem e o anexa ao final do quadro; o receptor repete o cálculo e compara os valores para detectar eventuais erros de transmissão (NOVIELLO, 2018).

O cálculo CRC-16 do Modbus RTU utiliza o polinômio 0x8005 (representado de forma invertida como 0xA001), aplicado por meio de operações XOR e deslocamento de bits sobre cada byte da mensagem, como ilustrado na Figura 13.

Figura 13 – Algoritmo iterativo de cálculo de CRC por meio de deslocamento de bits e operações XOR.



Fonte: Noviello (2018).

Conforme evidenciado na Figura 13, o algoritmo processa a mensagem bit a bit no registrador de deslocamento, aplicando XOR com o polinômio gerador sempre que o bit ejetado for '1'. O microcontrolador STM32F407 utilizado neste trabalho possui um periférico de CRC em hardware, que acelera esse cálculo; contudo, como o hardware do STM32F407 suporta apenas o polinômio CRC-32 de forma nativa, o CRC-16 do Modbus foi implementado por software (STMicroelectronics, 2024).

2.2 Sistemas Operacionais em Tempo Real (RTOS)

Os sistemas operacionais de tempo real (RTOS) são projetados para aplicações embarcadas que exigem respostas determinísticas a eventos externos. O principal objetivo não é executar tarefas o mais rápido possível, mas garantir que cada evento ocorra dentro de um prazo predefinido (DENARDIN; BARRIQUELLO, 2019).

Segundo Denardin e Barriquello (2019), existem dois tipos principais. No soft real-time a perda de prazos causa degradação de desempenho, enquanto no hard real-time o descumprimento dos tempos pode comprometer o sistema inteiro. Um RTOS organiza a execução em tarefas com prioridades definidas, utilizando mecanismos como semáforos, mutex e filas de mensagens para sincronização e troca de dados entre tarefas.

Embora o dispositivo desenvolvido neste trabalho não utilize um RTOS, o conhecimento desses conceitos é relevante pois a arquitetura de software adotada, baseada em um superloop cooperativo com máquinas de estado, foi projetada de forma a facilitar uma eventual migração para um ambiente com RTOS em trabalhos futuros.

2.2.1 Comunicação Serial UART/USART

Conforme Noviello (2018), as interfaces UART (Universal Asynchronous Receiver/Transmitter) e USART são os periféricos internos responsáveis por converter bytes paralelos em um fluxo serial de sinais elétricos, e vice-versa. Na comunicação assíncrona (UART), os dispositivos operam em uma taxa de transmissão (baud rate) previamente acordada, com quadros compostos por Start Bit, bits de dados, bit de paridade opcional e Stop Bits.

Essas interfaces operam com níveis lógicos locais (TTL), sendo associadas a transceivers externos para implementar padrões físicos como RS-232 e RS-485. Em redes RS-485 half-duplex, o controle rigoroso da direção é crítico. Microcontroladores da família STM32 oferecem suporte nativo ao Hardware Driver Enable (DE), permitindo que a própria USART gerencie o pino de direção do transceptor e garanta os tempos de silêncio exigidos pelo Modbus RTU (NOVIELLO, 2018).

2.2.2 Periféricos do Microcontrolador

Os periféricos descritos nesta seção são utilizados diretamente no dispositivo desenvolvido. Os temporizadores realizam o controle de temporização do protocolo, o SPI atende ao controlador de toque, o RTC fornece os timestamps dos registros, o sistema de arquivos FAT atua no armazenamento em cartão SD e o FSMC realiza a interface com o display gráfico.

2.2.2.1 Temporizadores (Timers)

Os timers são contadores autônomos cujo incremento é derivado de um clock de referência dividido por um prescaler configurável (NOVIELLO, 2018). No contexto deste trabalho, os temporizadores são utilizados principalmente para implementar o intervalo de silêncio T3.5 exigido pelo Modbus RTU, tempo mínimo de 3,5 caracteres entre quadros, operando em modo one-shot com interrupção ao fim da contagem.

2.2.2.2 Interface SPI

A interface SPI (Serial Peripheral Interface) é um protocolo serial síncrono que opera em full-duplex, composto por quatro sinais: SCK (clock), MOSI, MISO e SS (NOVIELLO, 2018). No dispositivo deste trabalho, o barramento SPI é dedicado à interface com o controlador de toque XPT2046, responsável por capturar as coordenadas de interação do usuário na tela.

2.2.2.3 Relógio de Tempo Real (RTC)

O RTC (Real-Time Clock) é um periférico dedicado ao controle contínuo de data e hora, projetado para operar de forma independente do processador principal, inclusive com a alimentação principal desligada, desde que uma bateria de backup esteja conectada (NOVIELLO, 2018). No dispositivo deste trabalho, o RTC é utilizado para gerar timestamps precisos nos registros do barramento Modbus capturados pelo sniffer, permitindo rastreabilidade temporal dos eventos de comunicação salvos no cartão SD.

2.2.2.4 Sistema de Arquivos FAT e Biblioteca FatFs

O FAT (File Allocation Table) é um sistema de arquivos amplamente adotado em mídias removíveis como cartões SD, graças à sua simplicidade e compatibilidade universal (CHAN, 2021). Para viabilizar sua implementação em microcontroladores com restrições de memória, utiliza-se a biblioteca FatFs, desenvolvida por Elm Chan, uma implementação em C, portável e otimizada, integrada oficialmente ao ecossistema de bibliotecas HAL da STMicroelectronics (NOVIELLO, 2018).

No escopo deste trabalho, a FatFs possibilita a gravação incremental dos quadros Modbus capturados em um arquivo de log no cartão SD, viabilizando o armazenamento contínuo e estruturado dos dados de diagnóstico da rede.

2.2.2.5 Controlador de Memória Estática Flexível (FSMC/FMC)

O FSMC (Flexible Static Memory Controller) é um periférico do STM32F407 que permite mapear dispositivos externos, como memórias e controladores de display, diretamente no espaço de endereçamento do processador (STMicroelectronics, 2008). Isso

viabiliza a transferência de dados para a tela por meio de simples operações de escrita em memória, sem intervenção de software para cada sinal do barramento paralelo.

No dispositivo deste trabalho, o FSMC foi empregado para realizar a interface com o controlador gráfico ILI9341 do display touchscreen, garantindo a largura de banda necessária para a atualização fluida da interface gráfica (STMicroelectronics, 2020).

3 Metodologia

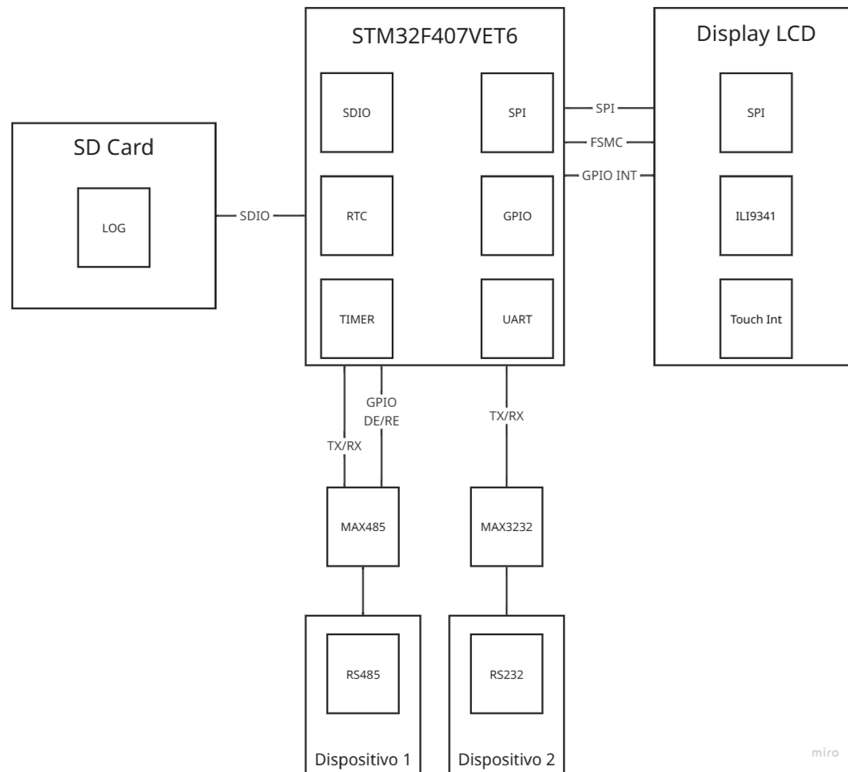
3.1 Visão Geral do Sistema

A ferramenta de diagnóstico desenvolvida neste trabalho consiste em um dispositivo embarcado portátil capaz de operar sobre redes Modbus RTU em três modos distintos, configuráveis em tempo de execução como dispositivo mestre, enviando requisições periódicas ou pontuais para escravos; como dispositivo escravo, simulando registros internos para responder a um mestre; e como sniffer, registrando todo o tráfego do barramento serial em arquivo de log no cartão SD com timestamps precisos.

Do ponto de vista do usuário, o fluxo de operação é direto. Após a alimentação do dispositivo, o usuário configura na tela touchscreen os parâmetros do barramento serial (baudrate, paridade, stop bits), seleciona a interface física (RS-232 ou RS-485) e o modo lógico de operação (Mestre, Escravo ou apenas Monitoramento). No modo Mestre, é possível agendar requisições periódicas ou pontuais para qualquer escravo da rede. No modo Escravo, o usuário pode editar os registros internos diretamente pela interface gráfica. Em qualquer modo, a tela permite visualizar as últimas mensagens trafegadas no barramento, e a totalidade do tráfego é gravada incrementalmente no cartão SD.

A Figura 14 apresenta o diagrama de blocos do dispositivo, com os principais componentes físicos e os barramentos que os interligam ao microcontrolador.

Figura 14 – Diagrama de blocos do hardware do dispositivo, indicando os barramentos físicos utilizados.

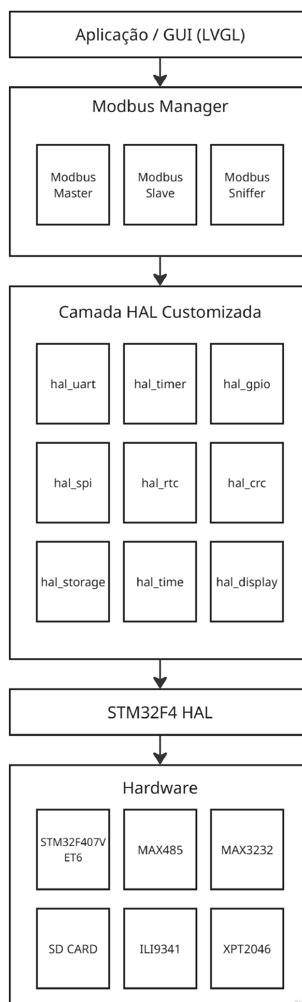


Fonte: Elaborado pelo autor.

Como evidenciado na Figura 14, cada periférico do dispositivo utiliza um barramento físico distinto do microcontrolador. O display ILI9341 é acessado pelo controlador FSMC (mapeamento direto em memória, alta largura de banda), o controlador de toque XPT2046 utiliza SPI, o cartão SD opera por SDIO em modo de 4 bits e os transceptores MAX485 e MAX3232 são acoplados às USARTs do microcontrolador. Essa separação por barramento foi um dos critérios que orientaram a escolha da plataforma de hardware, descrita na próxima seção.

Do ponto de vista de software, o sistema foi dividido em três níveis. O primeiro nível é uma Camada de Abstração de Hardware (HAL) desenvolvida de forma customizada para lidar com os componentes físicos. O segundo nível é a Pilha de Protocolo, com os módulos responsáveis pela comunicação Modbus (escravo, mestre e sniffer) e pelo gerenciamento dos modos de operação. O terceiro nível engloba a Aplicação e a Interface Gráfica, com as quais o usuário interage. Essa divisão estruturada foi essencial para o avanço do trabalho, pois permitiu que cada parte do código fosse desenvolvida, atualizada e testada individualmente antes da integração de todos os componentes. A Figura 15 apresenta essa organização em camadas.

Figura 15 – Arquitetura em camadas do firmware desenvolvido.

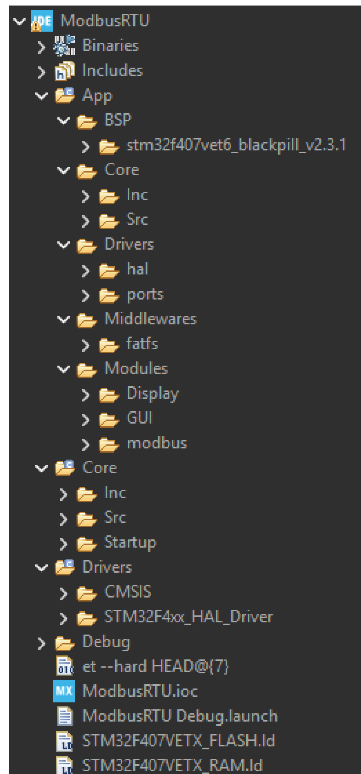


Fonte: Elaborado pelo autor.

A Figura 15 mostra que a Aplicação e a interface gráfica não interagem diretamente com os periféricos do microcontrolador, e sim com a Camada HAL customizada. Essa separação foi essencial para que cada parte do código fosse desenvolvida e testada individualmente, e permite que toda a plataforma de hardware seja substituída no futuro (por exemplo, migrando para um ESP32) sem que a aplicação superior sofra alterações estruturais.

O código-fonte resultante deste trabalho pode ser consultado no repositório online do projeto (SOUZA, 2026). A Figura 16 apresenta a estrutura de diretórios resultante.

Figura 16 – Estrutura de diretórios do projeto.



Fonte: Elaborado pelo autor.

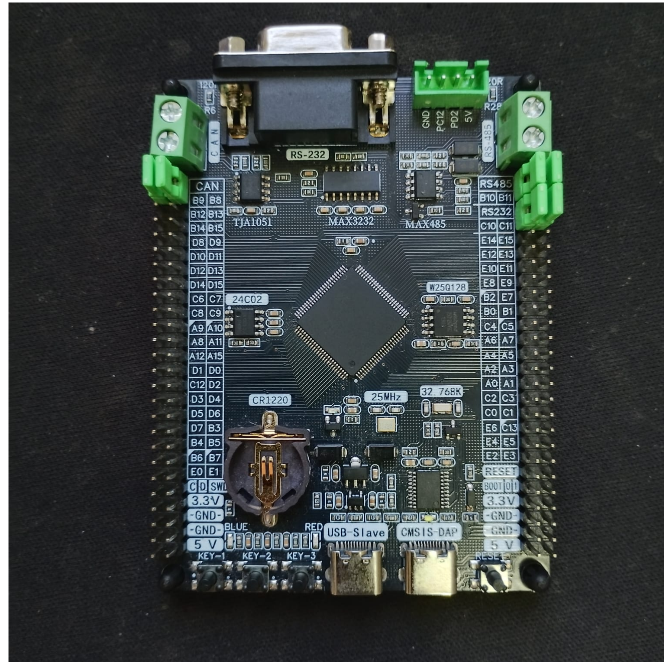
Conforme ilustrado na Figura 16, os arquivos foram organizados em diretórios separados para cada camada lógica do sistema, refletindo a divisão arquitetural apresentada anteriormente. O diretório *ports* concentra as implementações específicas do hardware (STM32), enquanto *modules* reúne bibliotecas portáveis como o controlador do display ILI9341 e a pilha FreeModbus.

3.2 Plataforma de Hardware e Componentes Físicos

Para o desenvolvimento e validação do sistema foram utilizadas duas placas de desenvolvimento distintas, ambas equipadas com o microcontrolador STM32F407VET6, e o uso sequencial dessas duas placas refletiu a estratégia incremental de desenvolvimento adotada no projeto.

A primeira placa utilizada, denominada nesta documentação como STM32-F407VET6 V3.2.1, foi adquirida no mercado paralelo e não consta nos catálogos públicos de hardware como o stm32-base.org. Essa placa foi escolhida para a primeira fase de desenvolvimento, focada na pilha de comunicação serial, pois integra nativamente em sua PCB os transceptores MAX3232 (para RS-232) e MAX485 (para RS-485), além de um conector DB9 padrão para RS-232, o que dispensou o uso de cabos e módulos adicionais nessa etapa. A Figura 17 apresenta essa primeira placa.

Figura 17 – Primeira placa de desenvolvimento (STM32-F407VET6 V3.2.1), utilizada na fase inicial de testes da pilha serial.

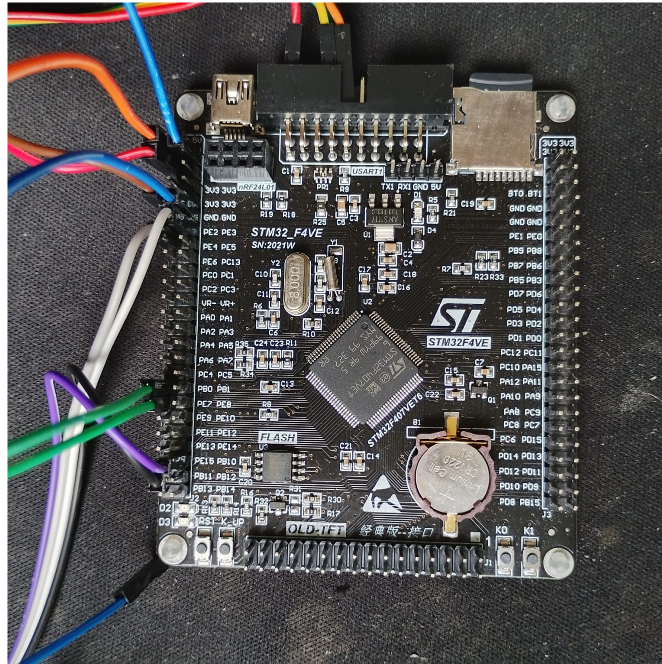


Fonte: Elaborado pelo autor.

Uma particularidade importante dessa primeira placa é que o controle de direção do barramento RS-485 (pinos DE/RE do MAX485) não é gerenciado via software pelo microcontrolador. O acionamento é realizado por um circuito elétrico passivo composto por diodo e capacitor, que ativa e desativa os pinos automaticamente a partir da análise elétrica do sinal de transmissão. Essa solução de hardware funciona, mas elimina o controle determinístico exigido pelo Modbus RTU, e exigiu, posteriormente, adaptações na camada UART quando o projeto migrou para a segunda placa.

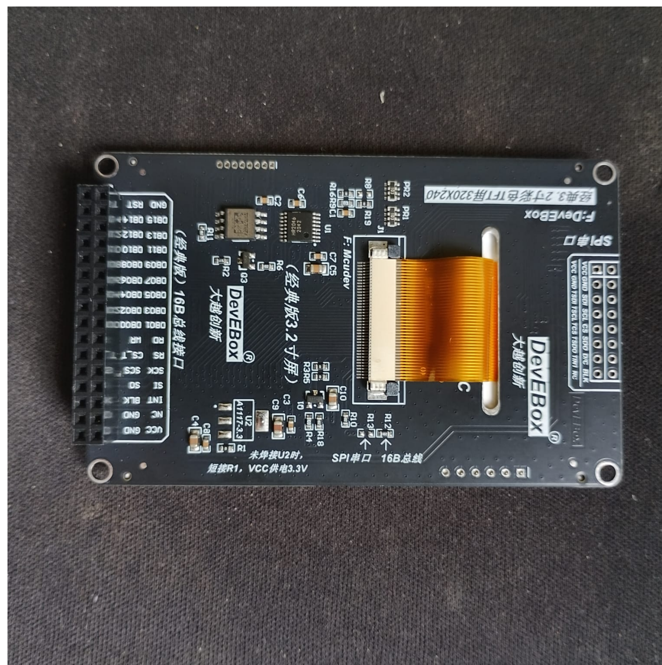
A segunda placa utilizada, modelo STM32 F4VE V2.0 (catalogada em stm32-base.org), é apresentada na Figura 18 e possui uma PCB construída com foco no acoplamento direto a interfaces gráficas, dispondo de barramento dedicado para o display ILI9341 com sensor de toque XPT2046 (compatível com o módulo TFT da fabricante DevEBox, apresentado na Figura 19), além de conector físico nativo para cartão SD e suporte para a bateria de backup do RTC. Em contrapartida, essa placa não integra transceptores seriais, exigindo o uso de módulos externos.

Figura 18 – Segunda placa de desenvolvimento (STM32 F4VE V2.0), adotada na versão final do protótipo.



Fonte: Elaborado pelo autor.

Figura 19 – Módulo de display TFT 3.2 polegadas DevEBox (ILI9341 + XPT2046), utilizado como interface gráfica.



Fonte: Elaborado pelo autor.

A escolha das duas placas seguiu a estratégia de desenvolvimento sequencial. Toda a primeira fase do trabalho, dedicada à validação das camadas de UART, timer, time, GPIO, interrupções, CRC e da pilha Modbus em modo Escravo e Mestre, foi realizada

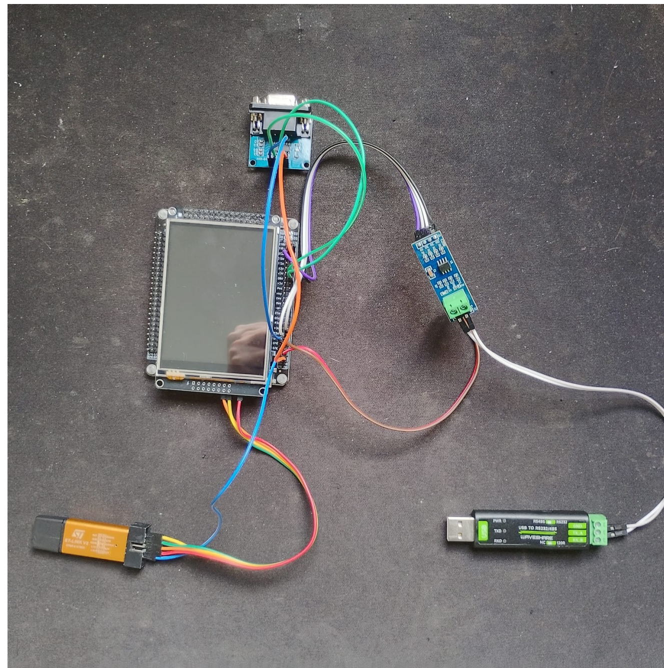
na primeira placa, aproveitando seus transceptores integrados. Na segunda fase, focada na integração do cartão SD, do RTC com logging e da interface gráfica touchscreen, o projeto migrou para a segunda placa. Como essa segunda placa não possui transceptores integrados, foram adicionados módulos externos baseados nos CIs MAX485 e MAX3232. A migração também exigiu a adaptação da camada UART, pois agora o controle dos pinos DE/RE do MAX485 passou a ser feito explicitamente via software, ao contrário do circuito passivo da primeira placa.

A solução ideal seria fundir as duas placas em uma PCB própria, integrando transceptores, conectores RS-232 e RS-485 com saídas adequadas, suporte a cartão SD, interface para o display touchscreen e dip switches para terminação do barramento (não presente nas duas placas avaliadas). O desenvolvimento dessa PCB dedicada foi listado como trabalho futuro.

Além das placas de desenvolvimento, os testes da pilha de comunicação exigiram a utilização de um conversor USB para RS-232/RS-485 da fabricante Waveshare. Esse equipamento permitiu conectar a rede serial do dispositivo a um computador, possibilitando o monitoramento do tráfego e a injeção de pacotes por meio de softwares de simulação. A função desse conversor no projeto foi análoga ao uso convencional do Serial.print em plataformas Arduino, enquanto o desenvolvedor Arduino usa o monitor serial para depurar variáveis, este trabalho utilizou o conversor para observar diretamente o barramento e validar se a camada UART estava transmitindo, recebendo e respeitando os tempos do protocolo corretamente.

A Figura 20 apresenta o protótipo final montado, baseado na segunda placa com os módulos externos acoplados. Observa-se nela a integração final dos componentes, com a placa STM32 F4VE V2.0 ao centro, com o display ILI9341 acoplado ao barramento FSMC, o cartão SD inserido no slot nativo, e os módulos MAX485 e MAX3232 conectados às USARTs do microcontrolador para fornecer as interfaces RS-485 e RS-232 do dispositivo.

Figura 20 – Protótipo final do dispositivo, baseado na placa STM32 F4VE V2.0 com módulos externos MAX485 e MAX3232 acoplados, display ILI9341 e cartão SD instalado.



Fonte: Elaborado pelo autor.

3.3 Arquitetura da Camada HAL

Um dos principais requisitos arquiteturais do projeto foi garantir que o código lógico da ferramenta de diagnóstico pudesse ser migrado para outras plataformas (como a família ESP32) no futuro, com o mínimo de retrabalho. Para atingir esse objetivo foi desenvolvido um HAL próprio, baseado na utilização de Ponteiros Opacos.

Nessa abordagem, os arquivos de cabeçalho (como hal_uart.h) expõem apenas as assinaturas das funções e as definições de tipos genéricos, ocultando completamente as estruturas de dados específicas do hardware. A implementação real, que manipula os registradores e as bibliotecas da fabricante, fica restrita ao diretório *ports* (como hal_uart_stm32f4.c). Dessa forma, a aplicação superior interage apenas com o contrato genérico, desconhecendo o microcontrolador subjacente.

Além da portabilidade, essa arquitetura tem outra vantagem prática. Ao impedir o cruzamento de responsabilidades entre HALs, ela mantém o controle estrito do que cada periférico está fazendo, simplificando a manutenção do código e o diagnóstico de falhas durante o desenvolvimento.

3.4 Camada de GPIO (hal_gpio)

A camada de abstração de GPIO foi desenvolvida utilizando ponteiros associados a uma Tabela de Funções, dividindo o gerenciamento dos pinos em três subcamadas, a Interface Pública (hal_gpio.h), a Implementação da Porta (hal_gpio_stm32f4.c) e o Pacote de Suporte à Placa (bsp_gpio.c).

No contexto deste trabalho, a camada GPIO é utilizada para um conjunto pequeno e bem definido de funções, controle dos pinos DE/RE do RS-485, Chip Select (CS) do controlador de toque XPT2046, leitura do pino de interrupção do toque (T_PEN) e reset do display. O botão de reset físico da placa atua diretamente sobre o pino NRST do microcontrolador, sem necessidade de tratamento via GPIO.

3.4.1 Mapeamento de Pinos (BSP)

A tradução dos identificadores lógicos em pinos físicos do microcontrolador foi isolada na camada de Board Support Package (BSP). Nessa camada, um vetor estático armazena o mapeamento exato da porta, do pino físico e a configuração do controlador de interrupções (NVIC).

Essa separação foi essencial no projeto porque, conforme descrito na seção anterior, a primeira e a segunda placa possuíam roteamento de pinos diferente. A migração entre placas exigiu apenas alterações pontuais na tabela de mapeamento do BSP, sem impacto na lógica do driver ou nas rotinas da aplicação principal.

3.4.2 Normalização de Níveis Lógicos e Interrupções Externas

No ambiente de hardware, é comum existirem circuitos que operam com lógica invertida (sinais ativos em nível baixo). Para mitigar essa complexidade na aplicação, o BSP define um parâmetro booleano de polaridade para cada pino. Nas rotinas de leitura e escrita, o driver consulta esse parâmetro antes de atuar sobre o registrador físico, garantindo que, na aplicação, "ativar" sempre signifique habilitar o dispositivo, independentemente da polaridade elétrica real.

O módulo também provê o tratamento assíncrono de interrupções externas (EXTI), fundamental para a detecção de toques no display (pino T_PEN). A interface permite que a aplicação registre funções de retorno (callbacks) específicas para cada GPIO. No nível do hardware, o driver intercepta a rotina global de tratamento de interrupção, identifica qual pino físico gerou o evento e executa apenas o callback registrado para aquele pino lógico.

3.5 Camada UART (hal_uart)

A camada de comunicação serial foi projetada para suportar o controle de fluxo externo (Half-Duplex) de transceptores RS-485 e RS-232, e para implementar regras dinâmicas de recepção de dados diretamente no driver, atendendo às exigências temporais do Modbus RTU. Seguindo o padrão de Ponteiros Opacos, a camada UART delega o acesso físico à implementação concreta para o STM32F4, suportando operações em modo polling e, primariamente, um modo não-bloqueante orientado a interrupções.

3.5.1 Detecção de Fim de Mensagem na UART

Um dos grandes desafios na implementação de protocolos de comunicação serial é determinar o momento exato em que uma mensagem completa foi recebida. Para tornar a camada UART reutilizável, sua arquitetura foi desenvolvida para suportar três estratégias independentes de finalização de recepção, configuráveis em tempo de execução:

- **Por Caractere Específico**: o driver analisa cada byte recebido na interrupção. Se o caractere coincidir com o delimitador configurado (como \n ou \r), a recepção é dada como encerrada e a aplicação é notificada via callback.
- **Por Tamanho Fixo**: a recepção é considerada completa quando o contador interno de bytes atinge um limite exato.
- **Por Inatividade de Comunicação**: estratégia mandatória para o Modbus RTU, cuja especificação exige que o fim de um quadro seja determinado por um intervalo de silêncio no barramento equivalente a, no mínimo, 3,5 caracteres.

Para implementar a finalização por tempo sem acoplar a UART a um periférico de timer específico, optou-se pela injeção de funções de callback. A aplicação superior injeta no driver UART dois ponteiros de função, sendo um para iniciar e outro para parar um temporizador genérico. A cada byte recebido, a UART invoca a função injetada de reinício do timer; se o fluxo cessar, o temporizador externo expira e aciona o callback de finalização da mensagem. Essa abordagem mantém a UART agnóstica em relação às fontes de tempo do sistema.

3.5.2 Controle de Direção Half-Duplex (DE/RE) para RS-485

Outro desafio inerente ao padrão RS-485 é a necessidade de controle físico da direção do barramento. O transmissor deve habilitar o pino Driver Enable (DE = 1) antes de enviar os dados e retornar ao modo de escuta (DE = 0) rigorosamente após a conclusão física do envio do último bit.

Em vez de acoplar à camada UART o controle direto de GPIOs, a solução arquitetural empregou a mesma estratégia de injeção de callbacks utilizada para o timer. Durante a inicialização do driver, a aplicação fornece uma função de callback encarregada de alternar os estados lógicos do controle de fluxo. A lógica interna da UART comanda essa função de forma autônoma, imediatamente antes de transmitir um dado, a UART invoca a função injetada solicitando a ativação do canal (DE = 1); uma vez que o hardware sinalize a conclusão da transmissão pelo flag TC (Transmission Complete), a UART injeta novamente a função solicitando o retorno ao modo de recepção (DE = 0).

Vale destacar que tanto o controle de timeout T3.5 quanto o controle DE/RE adotam o mesmo padrão, a UART não conhece diretamente o timer nem o GPIO, e sim recebe da aplicação superior os ponteiros de função para essas operações. Esse padrão de injeção de dependências preservou a portabilidade do driver base e foi essencial para permitir que a primeira placa (com controle DE/RE automático em hardware) e a segunda placa (com controle DE/RE via software) compartilhassem o mesmo código de UART.

3.6 Camada de Temporização (hal_timer)

A camada de timer foi desenvolvida para gerenciar eventos temporizados de forma assíncrona, não bloqueante e orientada a interrupções, atendendo principalmente aos requisitos do intervalo T3.5 do Modbus RTU. A interface pública expõe os temporizadores como instâncias opacas, ocultando os periféricos físicos subjacentes (TIM2, TIM3 e TIM5 da família STM32F4). O driver de baixo nível normaliza o prescaler dinamicamente para fornecer uma resolução nativa de 1 μs, independentemente das frequências de clock configuradas no microcontrolador.

O módulo suporta dois modos de operação, Periódico em que recarrega automaticamente o contador e gera interrupções cíclicas (utilizado pela base de tempo do LVGL), e One-Shot, que dispara uma única interrupção e para o timer imediatamente. O modo One-Shot é o utilizado pelo Modbus RTU para detecção do silêncio T3.5, a cada byte recebido na UART o timer é reiniciado, e se o silêncio no barramento atingir o limite configurado, o callback injetado sinaliza o fim da mensagem de forma desacoplada.

3.7 Camada de Tempo Global (hal_time)

Enquanto a camada hal_timer gera eventos assíncronos guiados por interrupções, a camada hal_time fornece o tempo absoluto do sistema para medições de intervalo, controle de timeouts por software e sincronização de fluxo. A interface do driver expõe apenas duas primitivas, a obtenção do tempo atual em milissegundos e a execução de atrasos.

No port desenvolvido para a família STM32, a base de tempo é atrelada ao SysTick,

configurado para gerar interrupções de 1 ms. A função de leitura do tempo atual em milissegundos é o pilar das rotinas não-bloqueantes do projeto, as máquinas de estado registram o instante inicial e, a cada ciclo do laço principal, calculam a diferença temporal para verificar se um timeout foi atingido. O módulo também fornece uma função de atraso bloqueante, utilizada exclusivamente na inicialização de periféricos externos (como o controlador ILI9341), que exigem tempos de acomodação física antes de aceitar comandos.

3.8 Camada CRC (hal_crc)

Para assegurar a confiabilidade dos pacotes Modbus e das operações de armazenamento, foi desenvolvida a camada de abstração para o cálculo de Códigos de Verificação de Redundância Cíclica. O módulo suporta os dois padrões mais adotados em sistemas embarcados, o CRC-16 (mandatório no Modbus RTU) e o CRC-32 (utilizado em validações de blocos no cartão SD). O driver público expõe uma função unificada que direciona a execução para a implementação mais eficiente disponível no hardware.

3.8.1 Limitação do Periférico CRC do STM32F407

O microcontrolador STM32F407 possui um periférico acelerador de CRC nativo, capaz de processar palavras de 32 bits em pouquíssimos ciclos de clock. Contudo, durante o desenvolvimento do driver constatou-se uma limitação arquitetural desse periférico, seu polinômio gerador é fixo em hardware, correspondendo exclusivamente ao padrão Ethernet/MPEG-2 (polinômio 0x04C11DB7 de 32 bits). Além disso, o periférico não oferece suporte nativo para a reversão da ordem dos bits.

Como o protocolo Modbus RTU exige um cálculo de 16 bits baseado no polinômio 0x8005 (representado de forma invertida como 0xA001), o periférico do STM32F407 tornou-se incompatível para a validação dos quadros seriais. Para solucionar esse impasse sem alterar a interface genérica da camada CRC, optou-se por uma arquitetura híbrida, quando a aplicação requisita CRC-16, o driver desvia o fluxo para uma rotina em C que implementa o cálculo padrão do Modbus RTU; quando a aplicação demanda cálculos de 32 bits (eventuais checagens de integridade no cartão SD), o driver habilita o periférico de hardware do STM32F407.

3.9 Camada RTC (hal_rtc)

Para viabilizar o registro de eventos (data logging) e a rastreabilidade das mensagens capturadas no barramento, foi desenvolvida a camada de Relógio de Tempo Real (hal_rtc). É importante distinguir esse módulo da camada hal_time, enquanto a hal_time gerencia

tempo relativo de máquina, a hal_rtc é responsável pelo tempo absoluto e legível (data e hora).

A interface pública expõe estruturas genéricas (data, hora e timestamp), delegando o acesso físico à implementação específica para o STM32F4. No pacote de suporte à placa, o clock do RTC foi configurado para utilizar o Oscilador Externo de Baixa Frequência (LSE), um cristal dedicado de 32,768 kHz, escolhido em detrimento dos osciladores internos (LSI) por sua superior estabilidade térmica e precisão. Por estar localizado no domínio de backup do microcontrolador, o RTC continua contando o tempo mesmo se a alimentação principal for cortada, desde que a bateria conectada ao pino V_BAT esteja ativa.

3.9.1 Limitação de Resolução em Milissegundos

Embora o periférico mantenha o calendário de forma autônoma, ele não fornece um registrador direto de milissegundos. Em vez disso, disponibiliza um registrador SubSeconds, um contador decrescente, que o driver converte em uma contagem crescente de milissegundos para prover timestamps de maior resolução (essenciais para calcular deltas de resposta no tráfego Modbus).

A resolução temporal efetiva é ditada pela configuração dos pré-divisores, o cristal de 32,768 kHz, dividido pelo pré-divisor assíncrono padrão $(127 + 1)$, fornece um clock síncrono de 256 Hz, o que resulta em uma granularidade temporal real de aproximadamente 3,9 ms $(1000/256 \approx 3{,}906$ ms). Essa limitação inerente da arquitetura do RTC do STM32F407 deve ser considerada ao analisar os timestamps dos quadros Modbus capturados.

3.10 Camada SPI (hal_spi)

No contexto deste trabalho, o barramento SPI foi dedicado exclusivamente à interface com o controlador de toque XPT2046 acoplado ao display. O processo de amostragem das coordenadas exige operação Full-Duplex, o microcontrolador envia um byte de comando (selecionando o canal X ou Y) e, simultaneamente, aplica pulsos de clock para que o controlador retorne os bytes da conversão A/D. Para suprir essa demanda, o sistema utiliza a rotina de transmissão e recepção simultânea em modo polling bloqueante, escolha justificada pelo baixo volume de dados trafegados a cada leitura, que dispensa o uso de DMA ou interrupções.

Uma particularidade do SPI é a necessidade de um sinal físico de seleção de escravo (Chip Select). Em vez de delegar esse pino ao próprio periférico SPI, a camada hal_spi foi arquitetada de forma agnóstica, e o controle do CS é realizado pela aplicação superior através de chamadas à camada hal_gpio. Essa estratégia mantém a coerência com o padrão adotado na camada hal_uart (DE/RE) e permite que múltiplos escravos SPI compartilhem o mesmo barramento sem conflitos no driver base.

3.11 Camada de Armazenamento (hal_storage)

Para viabilizar o armazenamento não volátil e estruturado dos registros da rede, desenvolveu-se a camada de armazenamento, dividida em três níveis lógicos, a Interface de Aplicação (hal_storage), o middleware de Sistema de Arquivos (FatFs) e a Camada de Hardware Físico (SDIO/BSP).

A interface pública expõe operações semânticas de alto nível, escrita de novo arquivo, leitura e, crucialmente, escrita incremental (append). A operação de append é fundamental para a função de Logger do protótipo, pois permite que novos quadros Modbus interceptados sejam adicionados ao final de um arquivo de log existente, evitando reescrever megabytes de dados a cada nova captura.

Na camada mais baixa optou-se por não utilizar SPI para a comunicação com o cartão SD; em vez disso, empregou-se o periférico dedicado SDIO do STM32F407 em modo Wide Bus de 4 bits, multiplicando a largura de banda da transferência física e mitigando gargalos de I/O que poderiam travar a recepção de pacotes na UART durante o salvamento de logs densos.

A integração com o sistema de arquivos é realizada via uma camada de tradução, que conecta as chamadas da FatFs à HAL da ST. Para que os arquivos exibam datas corretas quando o cartão SD é inserido em um computador, a função de obtenção de tempo da FatFs foi acoplada à camada hal_rtc, fornecendo o calendário absoluto no formato compactado de 32 bits exigido pela especificação FAT.

3.12 Camada do Display e Controlador Gráfico

Para suportar a Interface Gráfica do dispositivo, foi desenvolvida uma pilha de software de vídeo estruturada em três níveis, a configuração física do barramento, a camada de abstração de comunicação (hal_display) e o driver independente do controlador gráfico ILI9341. O objetivo central foi isolar a lógica de inicialização e desenho do display em relação ao barramento físico utilizado, permitindo o reaproveitamento da biblioteca gráfica em projetos futuros.

3.12.1 Aceleração por Mapeamento de Memória (FSMC)

Para obter taxas de atualização fluidas no display sem onerar a CPU com o chaveamento manual de pinos GPIO, o projeto utilizou o Controlador de Memória Estática Flexível (FSMC) do STM32F407, que permite mapear periféricos externos diretamente no espaço de endereçamento da memória. No pacote de suporte à placa, o display foi conectado ao Banco 1 do FSMC (endereço base 0x60000000).

A lógica arquitetural dessa abordagem está no tratamento do pino RS (que informa ao display se o barramento contém um Comando ou um Dado). O pino RS foi fisicamente conectado ao pino de endereço lógico A18 do FSMC, criando dois ponteiros de memória distintos no firmware:

- Endereço de Comando (0x60000000): ao escrever neste endereço, o bit A18 é 0 e o FSMC informa ao display que a transação é um Comando.
- Endereço de Dado (0x60080000): ao escrever neste endereço, o bit A18 é 1 (2^{19} em endereçamento de bytes) e o FSMC informa ao display que a transação é um Dado.

Com essa topologia, o envio de um pixel de 16 bits para a tela custa apenas um único ciclo de instrução de armazenamento na memória da CPU. Adicionalmente, foi inserido um mecanismo de sincronização configurável que insere micropausas a cada bloco de pixels desenhados, evitando artefatos visuais (tearing) comuns em transferências assíncronas de altíssima velocidade.

3.12.2 Integração do Controlador ILI9341

A integração do display foi estruturada por meio do driver do controlador gráfico ILI9341, alocado no diretório *modules*. Para integrá-lo ao sistema utilizou-se o padrão de Injeção de Dependências através de um arquivo de costura. Durante a inicialização, o sistema empacota as funções genéricas do hal_display dentro de uma estrutura de I/O e a passa para o núcleo do driver. A partir desse momento, o módulo do ILI9341 assume o controle do display, enviando a sequência de comandos hexadecimais de inicialização do painel TFT, configurando a orientação para paisagem (320×240) e disponibilizando primitivas de desenho geométrico por região.

Para acomodar diferentes barramentos na arquitetura portável, criou-se a camada hal_display. Por meio de diretivas de pré-processador, o sistema seleciona em tempo de compilação qual driver concreto será instanciado, garantindo que a aplicação não faça chamadas diretas aos ponteiros de memória do FSMC. Caso a placa seja alterada para um microcontrolador sem FSMC, basta recompilar com a flag adequada para que a mesma API de vídeo seja roteada para um driver alternativo.

3.13 Dispositivo Escravo (Integração FreeModbus)

Para prover ao dispositivo a capacidade de atuar como simulador de dispositivo de campo, optou-se pela integração da biblioteca open-source FreeModbus, amplamente validada na indústria para a operação no modo Escravo. A biblioteca implementa nativamente a lógica do protocolo, a validação de redundância cíclica (CRC) e o roteamento de

códigos de função. A escolha foi também motivada pelo fato do autor possuir experiência prévia com essa biblioteca em outros projetos.

O esforço de desenvolvimento concentrou-se em duas frentes, a adaptação da stack para a arquitetura HAL customizada do projeto e a implementação da camada de aplicação com mapeamento de registradores lógicos.

3.13.1 Camada de Interface Física (Port Layer)

O FreeModbus foi projetado para ser agnóstico em relação ao microcontrolador. Para que ele pudesse controlar o hardware, desenvolveu-se a camada de Port, conectando as exigências da biblioteca às HALs previamente desenvolvidas:

- **Comunicação Serial**: acoplada à camada hal_uart no modo orientado a interrupções. Cada byte recebido aciona a máquina de estados do FreeModbus. Aqui é implementado o controle DE/RE do RS-485 utilizando os callbacks da camada hal_uart. Paralelamente, foram inseridas chamadas ao módulo Sniffer, permitindo que a função de Logger capture o tráfego do Escravo de forma transparente.
- **Temporização T3.5**: instanciou-se a camada hal_timer em modo One-Shot. Cada byte que chega pela serial reinicia o temporizador; se o fluxo cessar, o temporizador expira e sinaliza ao stack que o quadro está completo.

3.13.2 Camada de Aplicação e Mapeamento de Memória

Com a infraestrutura de rede estabelecida, desenvolveu-se a estrutura de dados que simula os sensores e atuadores do Escravo. O espaço de endereçamento foi particionado em vetores estáticos distintos com 512 Coils, 16 Discrete Inputs, 512 Input Registers e 512 Holding Registers. A ponte entre a requisição decodificada pelo FreeModbus e os vetores de memória foi implementada através de funções de callback. Quando o Mestre remoto solicita, por exemplo, a leitura de um registrador Holding, o laço global de eventos invoca o callback correspondente, que calcula o offset no vetor local, valida limites de acesso, empacota os dados em formato Big-Endian e os retorna ao núcleo do protocolo.

Por meio da interface gráfica, o usuário pode editar diretamente os valores desses vetores, simulando o comportamento de um equipamento de campo na rede.

3.14 Dispositivo Mestre Próprio (ModbusMaster)

Diferentemente do modo Escravo (passivo e reativo), a função de Mestre Modbus exige uma postura ativa e coordenadora na rede. O Mestre é o único dispositivo autorizado

a iniciar transações, sendo responsável por orquestrar a varredura dos nós, gerenciar timeouts e lidar com dispositivos inoperantes.

Como a biblioteca FreeModbus não provê suporte nativo robusto para a operação Mestre, desenvolveu-se uma pilha de comunicação própria, inspirada na arquitetura do FreeModbus mas estruturada do zero para operar de forma não bloqueante, modular e orientada a varredura.

3.14.1 Agendador (Scheduler) e Fila de Requisições

Em aplicações de diagnóstico é necessário que a interface gráfica permaneça responsiva enquanto a comunicação serial ocorre em segundo plano. Para isso, o Mestre não escreve diretamente na porta serial, a aplicação injeta requisições de leitura ou escrita em uma Fila de Trabalhos (Job Queue).

Para leituras contínuas (como o monitoramento em tempo real de um medidor de energia em períodos de tempo definidos), implementou-se um Agendador Periódico. O agendador armazena a requisição e a periodicidade desejada e, a cada ciclo do laço principal, verifica via hal_time se o momento de execução foi atingido. Em caso afirmativo, o trabalho é automaticamente injetado na Fila.

3.14.2 Gerenciamento de Escravos e Estratégia de Backoff

Um dos problemas mais críticos em redes seriais multidrop é a degradação de desempenho causada por dispositivos ausentes ou danificados. Se o Mestre tentar ler continuamente um sensor desconectado, o sistema perderá centenas de milissegundos aguardando o timeout a cada requisição, penalizando a varredura dos demais dispositivos saudáveis.

Para mitigar esse gargalo, implementou-se um algoritmo de gerenciamento de estado por nó. O Mestre rastreia falhas individuais de comunicação. Caso um dispositivo falhe consecutivamente além de um limiar predefinido, o Mestre o classifica como offline e aplica uma política de recuo temporário (Backoff). Durante esse período, qualquer requisição destinada àquele escravo falha instantaneamente na fila, permitindo que a banda serial seja integralmente dedicada aos sensores operantes até que o tempo de punição expire.

3.14.3 Máquina de Estados e Decodificação de Códigos de Função

A orquestração das requisições é regida por uma Máquina de Estados Finitos simplificada, com dois estados mutuamente exclusivos:

1. **Estado Ocioso**: o sistema verifica se há trabalhos na fila e se o tempo de silêncio entre quadros foi respeitado. Satisfeitas as condições, a requisição é traduzida em

quadro RTU binário, o CRC-16 é anexado e o pacote é enviado via hal_uart. O estado transita para a espera.

2. **Aguardando Resposta**: a máquina delega o controle de tempo a uma função verificadora de quadro pronto, que opera em conjunto com o temporizador injetado (T3.5) na interrupção da UART. O estado é abandonado se o timeout total da requisição for excedido (erro) ou se um pacote completo for detectado.

Recebido o quadro de resposta, o sistema confirma a integridade do CRC-16, valida o endereço do escravo e verifica o byte de Código de Função. Caso o byte retornado tenha o bit mais significativo em 1 (por exemplo, 0x83 para falha no FC 03), o sistema identifica uma Exceção Modbus e aborta o processamento. Estando a resposta íntegra, um despachante condicional decodifica o formato esperado, os códigos 01 e 02 tratam respostas em bits (Bobinas e Entradas Discretas), códigos 03 e 04 tratam respostas analógicas em Big-Endian, e códigos 05, 06, 0F e 10 tratam confirmações de escrita.

3.14.4 Cache Local de Leituras

Para otimizar o acesso das camadas superiores (como a interface LVGL) aos dados lidos, os valores extraídos das respostas não são apenas retornados via callbacks transitórios, eles são armazenados de forma persistente em estruturas de Cache Analógico e Digital, subdivididas por ID de escravo. Esse mecanismo permite que a GUI solicite o valor atual de um sensor a qualquer instante consultando apenas a memória RAM, desobrigando a interface visual de interagir com as complexidades temporais da rede serial.

3.15 Gerenciador Modbus (modbus_manager)

Para convergir as múltiplas funcionalidades da ferramenta (Mestre, Escravo e Sniffer) em um dispositivo com recursos físicos limitados (uma única porta serial ativa), desenvolveu-se o Gerenciador Modbus. Este módulo atua como o controlador central da arquitetura, permitindo que o algoritmo de comportamento do sistema seja selecionado dinamicamente em tempo de execução.

A máquina de estados global do Gerenciador compreende os estados Inativo, Escravo e Mestre. Quando o usuário, através da interface gráfica, comanda a troca de modo (por exemplo, parar a simulação de Escravo e iniciar o escaneamento como Mestre), uma rotina de inicialização é invocada. O Gerenciador paralisa imediatamente as interrupções de RX e TX em andamento, desfaz as alocações da máquina de estados do FreeModbus e, sequencialmente, injeta os callbacks do driver adequado na camada física.

Esse controle estrito previne colisões de acesso a recursos físicos compartilhados (UART, timer, GPIOs DE/RE) e garante que as duas lógicas de protocolo jamais tentem

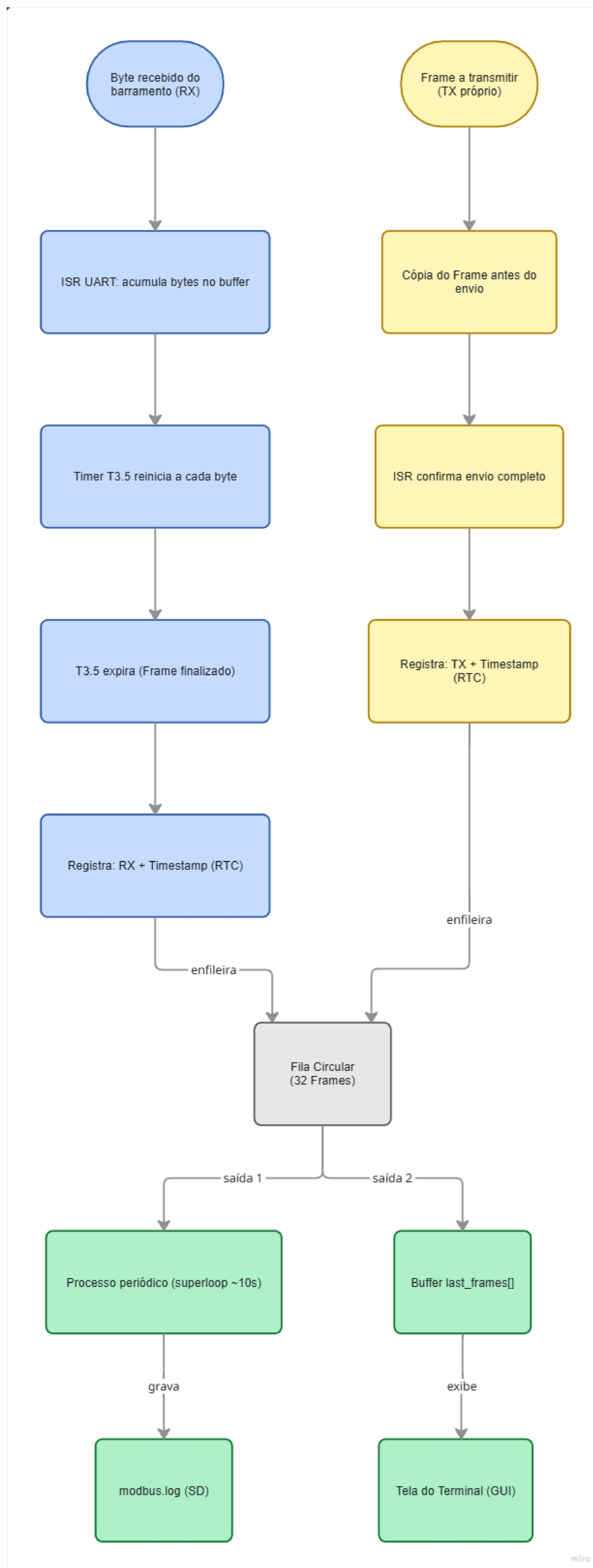
controlar o pino de direção do RS-485 simultaneamente. É também o Gerenciador que aplica os parâmetros configurados pelo usuário na interface (baudrate, paridade, interface física, modo lógico) à camada UART durante a transição entre modos.

3.16 Modbus Sniffer

Para prover a funcionalidade de análise do tráfego, desenvolveu-se o módulo Sniffer, um analisador de rede embutido e não intrusivo. Embora o termo sniffer esteja convencionalmente associado a uma postura puramente passiva de escuta, a abordagem adotada neste trabalho é mais abrangente, o módulo registra tudo o que trafega no barramento serial em que o dispositivo está conectado, incluindo os quadros que o próprio dispositivo transmite quando atua como Mestre ou Escravo. Em outras palavras, o Sniffer escuta o barramento como um todo, e não apenas os dispositivos externos.

Esse comportamento garante que o arquivo de log contenha um histórico fiel de toda a transação Modbus, com requisições e respostas correlacionadas, independentemente de qual extremidade da comunicação foi originada pelo próprio dispositivo. A Figura 21 ilustra esse fluxo unificado de captura.

Figura 21 – Fluxograma de captura do Modbus Sniffer, evidenciando as duas fontes de dados (RX e TX) e as duas saídas (log no cartão SD e buffer para a GUI).



Fonte: Elaborado pelo autor.

Conforme apresentado na Figura 21, o sniffer possui duas fontes de dados (RX externo e TX próprio) e duas saídas (gravação incremental no cartão SD e buffer para visualização na GUI). A arquitetura baseia-se no conceito de injeção no fluxo de dados, o módulo intercepta as chamadas diretamente nas Rotinas de Serviço de Interrupção (ISR) da UART, permitindo a observação passiva do tráfego sem interferir na execução do protocolo Modbus.

3.16.1 Captura no Fluxo de Recepção (RX)

A captura de dados recebidos é realizada byte a byte. No modo Mestre, o Sniffer é acionado pelo callback do gerenciador; no modo Escravo, ele é acionado no port serial do FreeModbus. Em ambos os cenários, o fluxo elétrico aciona a interrupção da UART, que despacha o mesmo byte simultaneamente para a máquina de estados do protocolo e para a função de captura do Sniffer. Essa abordagem elimina a duplicação de drivers e garante que a ferramenta processe exatamente a mesma informação que o núcleo do Modbus.

Como o processo de recepção não possui conhecimento prévio do tamanho da mensagem, o Sniffer utiliza o mesmo mecanismo de delimitação do Modbus RTU, o intervalo de silêncio T3.5. Os bytes recebidos são armazenados sequencialmente em um buffer temporário; quando o temporizador expira, a função de fechamento é acionada, o quadro é considerado fechado, marcado como RX e recebe o timestamp do instante de fechamento.

3.16.2 Captura dos Quadros Transmitidos (TX)

Para a captura de quadros transmitidos pelo próprio dispositivo, a implementação adotou um processo de validação em duas etapas, garantindo que o log contenha apenas dados que efetivamente trafegaram no barramento:

1. **Armazenamento Prévio**: antes de acionar a transmissão na UART, a aplicação invoca a função de armazenamento do Sniffer. O quadro a ser enviado é copiado para um buffer temporário e uma variável de estado é ativada.

2. **Confirmação de Envio**: apenas após a conclusão física da transmissão (sinalizada pelo evento de transmissão completa na interrupção do hardware), a função de confirmação é chamada. O quadro é então consolidado, marcado como TX e despachado para a fila com seu respectivo timestamp.

3.16.3 Desacoplamento por Fila Circular e Duplo Buffer

A escrita no cartão SD é uma operação bloqueante que não pode ser executada dentro de rotinas de interrupção. Para isolar esse tempo de processamento, o Sniffer utiliza

uma Fila Circular com capacidade para 32 quadros. As interrupções de RX e TX limitam-se a empurrar os quadros formatados para essa fila. No loop principal do sistema, a função de processamento consome a fila, converte os bytes em uma string formatada (incluindo data e hora) e realiza a escrita incremental no arquivo modbus.log.

Para atender simultaneamente à gravação em disco e à GUI, o módulo mantém um segundo espaço de memória contendo os últimos quadros capturados. A interface gráfica consulta este vetor em RAM, permitindo a exibição do tráfego serial em tempo real na tela do dispositivo, sem acarretar concorrência de leitura no sistema de arquivos do cartão SD.

3.17 Interface Gráfica (LVGL) e Controle do Sistema

Para o controle local e a visualização dos dados de diagnóstico, o sistema emprega a biblioteca gráfica LVGL (Light and Versatile Graphics Library). A implementação desta camada atua como o ponto central de controle do dispositivo, sendo responsável por parametrizar as portas seriais, agendar requisições e exibir o tráfego da rede. A integração da biblioteca ao firmware foi dividida em três adaptações, o controle de imagem, entrada de toque e base de tempo.

3.17.1 Adaptação de Hardware (Display, Touch e Tick)

A renderização das telas não ocorre diretamente no display. O LVGL foi configurado para operar com um buffer parcial em RAM equivalente a um décimo da altura da tela. A biblioteca processa os elementos visuais nesse buffer e, ao finalizar uma área, chama a função de transferência, que utiliza as primitivas do controlador ILI9341 para enviar os dados à tela.

A interação do usuário é gerenciada pelo módulo de entrada do LVGL, que realiza a leitura do controlador de toque XPT2046 via SPI. A função de leitura verifica o estado do pino de interrupção e, caso um toque seja detectado, realiza leituras sequenciais via SPI. Os valores analógicos puros são submetidos a uma rotina de calibração geométrica, que corrige as escalas e inverte os eixos para corresponder à orientação de paisagem da tela.

Para manter a precisão das animações e dos eventos de entrada, a biblioteca exige uma base de tempo dedicada. Para evitar concorrência com o protocolo Modbus (que utiliza o timer principal), instanciou-se um segundo temporizador de hardware. Este temporizador gera interrupções periódicas a cada 5 ms, atualizando o relógio interno da interface gráfica.

3.17.2 Configuração Dinâmica e Controle do Sistema

A interface gráfica atua como a interface de configuração principal do dispositivo. Por meio de formulários na tela, o usuário define os parâmetros da comunicação serial

(baudrate, paridade, stop bits, interface física RS-232 ou RS-485) e o modo lógico de operação (Mestre ou Escravo).

Quando o usuário confirma as alterações, o callback associado ao botão consolida os dados em uma estrutura de configuração e aciona a rotina de partida do Gerenciador Modbus. Essa integração permite que o sistema paralise as operações ativas, reconfigure as portas da UART e do temporizador, e alterne o modo de operação em tempo de execução, sem reiniciar o microcontrolador.

No modo Mestre, o LVGL gerencia também a criação de tarefas de leitura e escrita, o usuário preenche os campos com o ID do escravo, código de função, endereço inicial, quantidade de registradores e periodicidade desejada. Ao confirmar, a interface preenche a estrutura de requisição e a envia para o agendador. A partir desse momento, o agendador do Mestre assume o controle, enviando as mensagens na rede no intervalo especificado.

A visualização das mensagens ocorre de forma desacoplada, utilizando duas fontes de dados:

1. **Dados de Registradores (Cache)**: para atualizar mostradores de valores lidos da rede, rotinas periódicas consultam o cache analógico e o cache digital. A tela exibe os últimos valores válidos armazenados em RAM pelo Mestre Modbus.
2. **Diagnóstico do Sniffer**: para a tela de terminal, o LVGL consome o buffer secundário do Sniffer. A interface itera sobre esse vetor para atualizar a lista do terminal, permitindo que o usuário visualize as últimas transações capturadas (com timestamps e marcações RX/TX) ao acionar manualmente o botão de atualização da tela.

Toda a execução do sistema é unificada no laço principal do firmware, no qual o processamento de eventos do LVGL ocorre de forma cooperativa com o processamento do protocolo Modbus e a gravação do log no cartão SD, garantindo o funcionamento determinístico e fluido da ferramenta.

3.18 Procedimento de Validação

Como a especificação Modbus RTU é rigorosa quanto à integridade das mensagens e aos tempos do barramento (qualquer erro de CRC ou de temporização T3.5 resulta em descarte do quadro pelo lado oposto), a própria operação correta com dispositivos externos serve como validação funcional do sistema. O procedimento de validação adotado neste trabalho consistiu em duas frentes principais.

A primeira utilizou o módulo Modbus do ambiente Node-RED como contraparte da rede, configurando o Node-RED como Mestre, foi possível validar a operação do dispositivo no modo Escravo (verificando se as requisições de leitura e escrita eram corretamente

respondidas para cada código de função suportado); e configurando o Node-RED como Escravo simulado, foi possível validar a operação do dispositivo no modo Mestre, observando-se o agendamento periódico, o tratamento de timeouts e o mecanismo de Backoff para escravos inoperantes.

A segunda frente utilizou um script em Python desenvolvido para o projeto, capaz de gerar tráfego controlado e analisar as mensagens recebidas no barramento via conversor USB-serial Waveshare. Esse script permitiu inspecionar quadro a quadro o conteúdo dos pacotes transmitidos pelo dispositivo, confirmando a aderência ao formato Big-Endian, ao endereçamento base zero e ao cálculo do CRC-16 do Modbus RTU. Os resultados detalhados desses testes são apresentados no Capítulo 4.

4 Resultados

Este capítulo apresenta os resultados obtidos com a ferramenta desenvolvida, organizados em três grandes blocos. O primeiro descreve a validação das camadas de abstração de hardware (HAL) e a resolução das principais falhas técnicas encontradas durante a integração dos periféricos. O segundo apresenta a validação dos três modos de operação da ferramenta (Escravo, Mestre e Sniffer/datalogger), com ênfase nesta última, que é a função central de análise de redes. O terceiro consolida a interface gráfica final e discute as decisões de projeto e as limitações observadas.

4.1 Validação e Aprimoramento da Camada de Abstração (HAL)

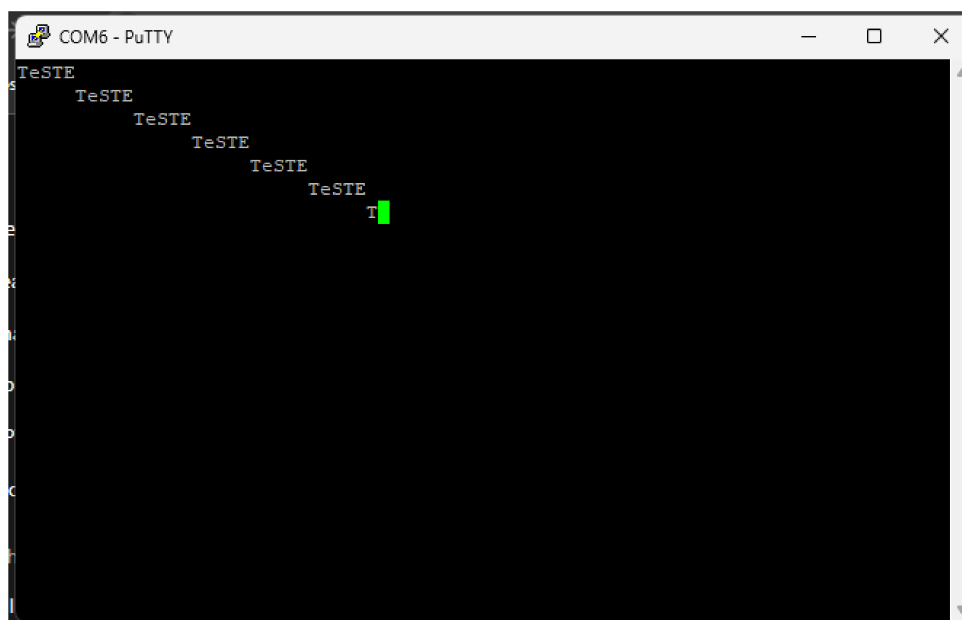
A etapa inicial de testes concentrou-se em atestar a confiabilidade dos drivers de baixo nível. Para cada periférico abstraído, elaborou-se um cenário de validação específico, avaliando-se as estratégias de finalização, controle de fluxo e integração com as demais camadas.

4.1.1 Comunicação Serial (UART) e Controle de Fluxo

O módulo hal_uart foi submetido a testes de comunicação direta com um computador (loopback e troca de mensagens com o software terminal PuTTY) por meio do conversor isolado USB para RS-232/RS-485 da Waveshare. Inicialmente, validou-se a transmissão em modo polling, enviando pacotes periódicos a cada 1 segundo. Durante os testes de recepção, identificou-se a necessidade de aprimorar a máquina de estados da UART para suportar múltiplas estratégias de finalização de quadro.

O driver foi expandido para detectar o fim da recepção por quantidade de bytes, por inatividade do barramento e por caractere terminador. A Figura 22 apresenta um dos ensaios de transmissão e recepção do módulo durante essa etapa.

Figura 22 – Exemplo de teste do hal_uart em modo polling, com transmissão e recepção sincronizadas pelo terminal serial.



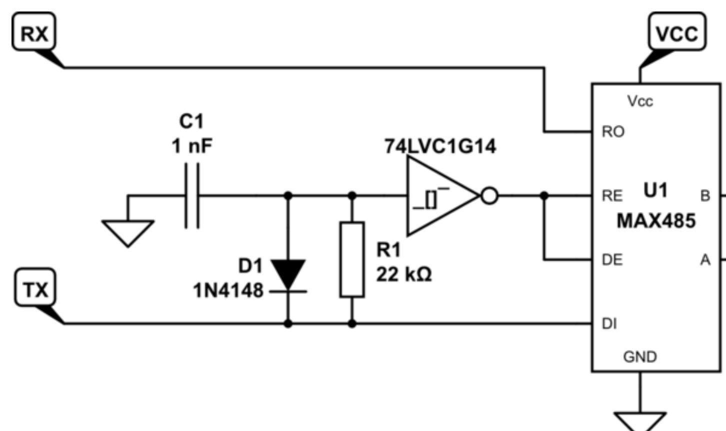
Fonte: Elaborado pelo autor.

Conforme observado na Figura 22, o driver foi capaz de transmitir os pacotes periódicos e processar as respostas recebidas sem perdas, validando o funcionamento básico da camada antes da integração com o protocolo Modbus.

4.1.1.1 Adaptação para Controle de DE/RE no Hardware Final

A migração para a placa de circuito final (STM32 F4VE V2.0 com módulos externos) revelou uma nova exigência elétrica. Diferentemente da primeira placa, que possuía controle automático de direção via circuito RC passivo (Figura 23), a topologia definitiva utilizava um transceptor MAX485 externo sem esse circuito, exigindo o controle rigoroso dos pinos Driver Enable (DE) e Receiver Enable (RE) por software.

Figura 23 – Circuito utilizado na primeira placa para controle automático de direção em comunicação RS-485 half-duplex sem pino DE/RE dedicado.



Fonte: Adaptado de Stack Exchange (2017).

A Figura 23 ilustra a solução passiva da primeira placa, na qual diodos e capacitores controlam a habilitação do transmissor a partir da própria atividade elétrica do sinal. Essa abordagem é simples mas não fornece o controle determinístico exigido pela temporização do Modbus RTU. Graças ao baixo acoplamento da arquitetura HAL, a transição para o controle por software foi resolvida com a injeção de callbacks do módulo hal_gpio diretamente na UART, garantindo o chaveamento Half-Duplex preciso sem necessidade de alterar o código da pilha Modbus.

4.1.2 Entradas/Saídas (GPIO) e Temporização

A validação de entradas digitais focou no comportamento das interrupções externas geradas por botões. O mapeamento físico do botão principal (KEY0) apresentou inversão lógica na placa definitiva. A correção exigiu a reconfiguração do hal_gpio para operar com resistor de elevação interno (Pull-up) e detecção em borda de subida (Rising Edge), comprovando a flexibilidade do parâmetro de polaridade do BSP.

O hal_timer foi validado em conjunto com o hal_gpio, atuando no controle do acionamento de LEDs sinalizadores e na contagem rigorosa de timeouts para a UART, especialmente o intervalo T3.5 do Modbus RTU. Os testes confirmaram que o reinício do timer a cada byte recebido e o disparo de callback ao expirar funcionavam de forma desacoplada e precisa, conforme arquitetado na Metodologia.

4.1.3 Sistema de Arquivos e RTC

O conceito do datalogger foi validado por meio de um ensaio de integração entre o hal_storage, o hal_rtc e o hal_timer. Um temporizador foi configurado para disparar eventos periódicos, momento em que o sistema solicitava o timestamp do RTC e o escrevia em um

arquivo de texto no cartão SD utilizando a camada de armazenamento em modo append. O ensaio confirmou também que os arquivos gerados exibiam datas corretas quando o cartão SD era inserido em um computador, validando a integração da função de tempo da FatFs com o RTC.

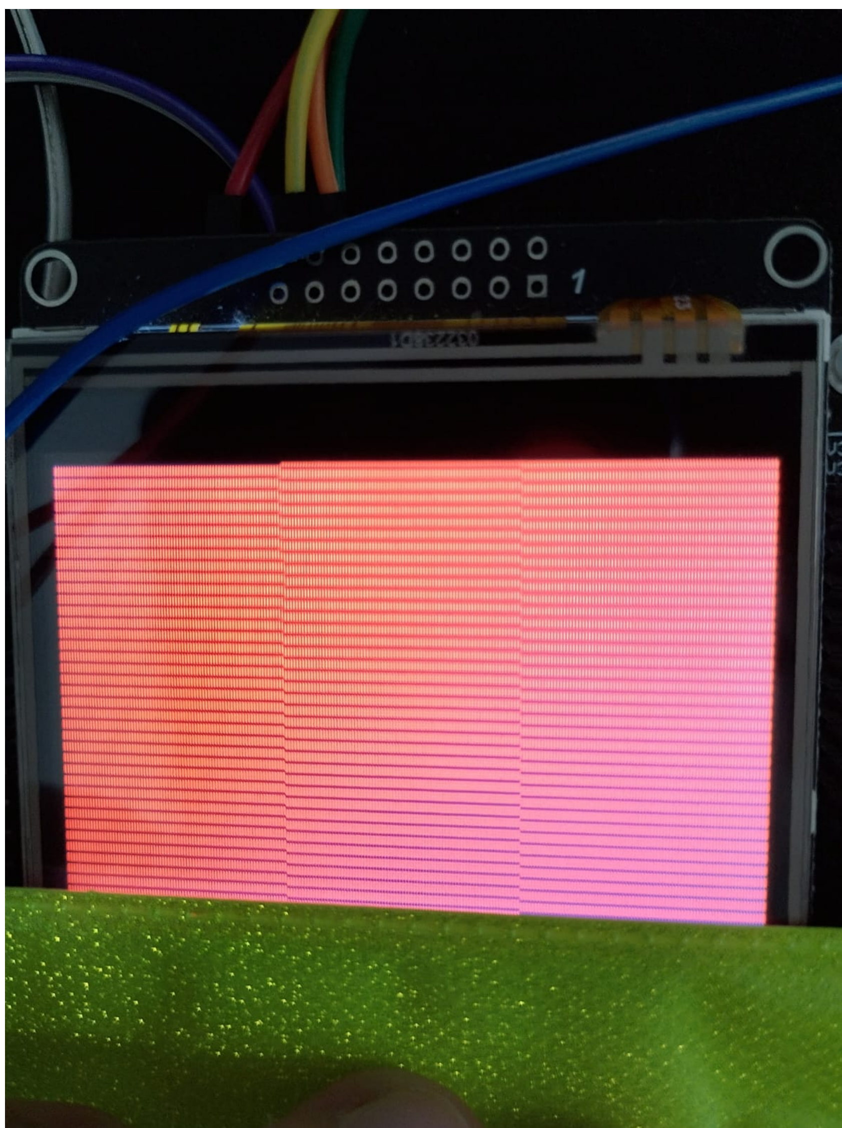
4.2 Resolução de Falhas Críticas na Integração Gráfica

A etapa mais desafiadora do projeto consistiu na estabilização da interface gráfica e na comunicação com os controladores do display e do toque. Esta seção documenta as principais falhas encontradas e as soluções aplicadas, pois representam decisões de engenharia relevantes do trabalho.

4.2.1 Anomalias no Barramento Paralelo (FSMC)

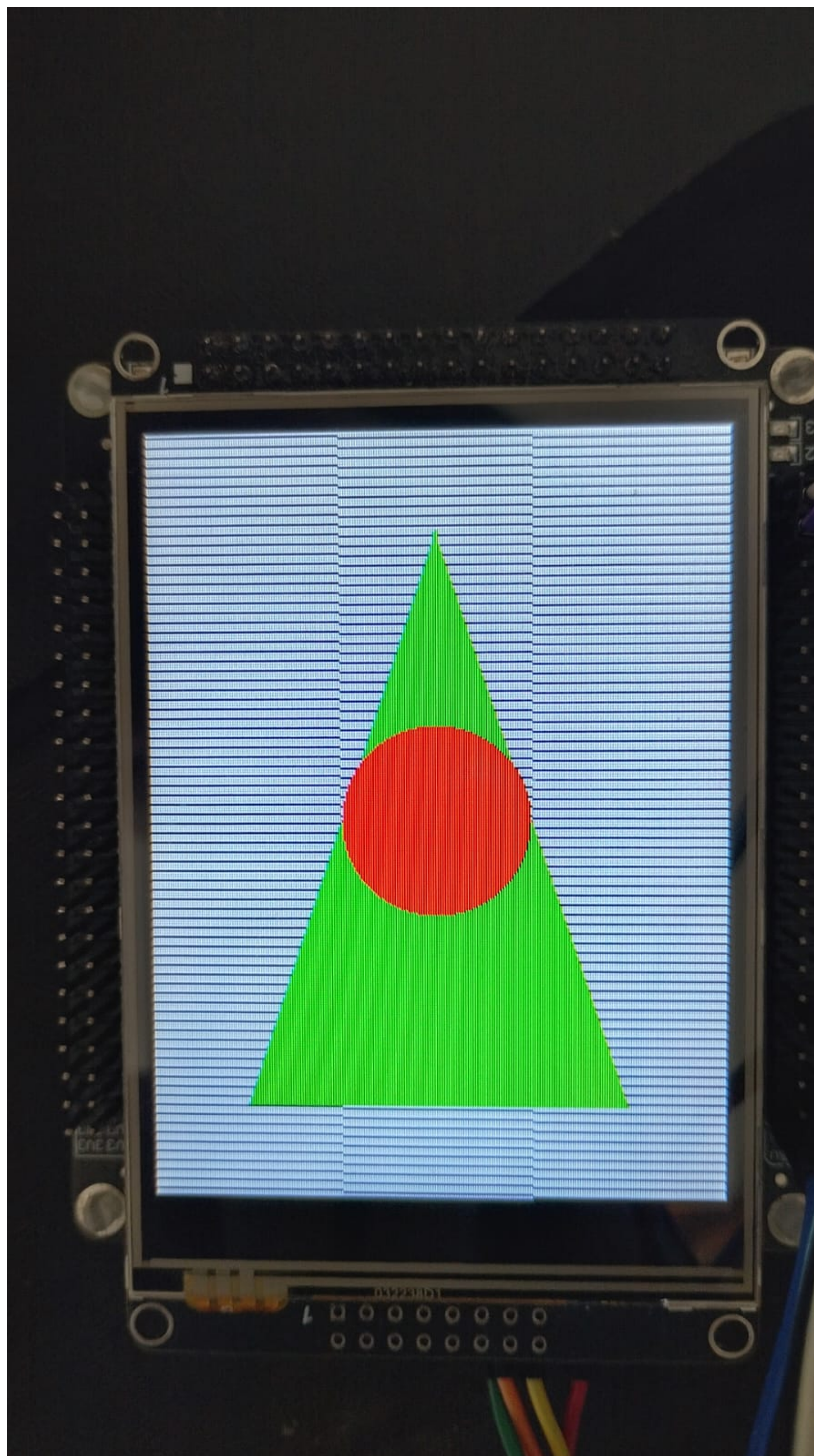
O primeiro ensaio de vídeo, o preenchimento da tela com cores sólidas, resultou em distorção severa, dividindo o display em três blocos desalinhados com espectros de cor incorretos, conforme apresentado nas Figuras 24 e 25.

Figura 24 – Tela do display ILI9341 dessincronizada após o primeiro ensaio do FSMC.



Fonte: Elaborado pelo autor.

Figura 25 – Exemplo de imagem renderizada com distorção causada pela inversão de endianness.



Fonte: Elaborado pelo autor.

A análise lógica das Figuras 24 e 25 revelou uma inconsistência de endianness no barramento de 16 bits, enquanto o controlador ILI9341 aguardava o byte mais significativo nas vias D8 a D15, o FSMC do STM32F4 transmitia nativamente no formato Little-Endian. A correção exigiu a implementação de uma macro de inversão de bytes na escrita dos

registradores de memória do FSMC, restaurando o alinhamento correto e a renderização fiel das imagens.

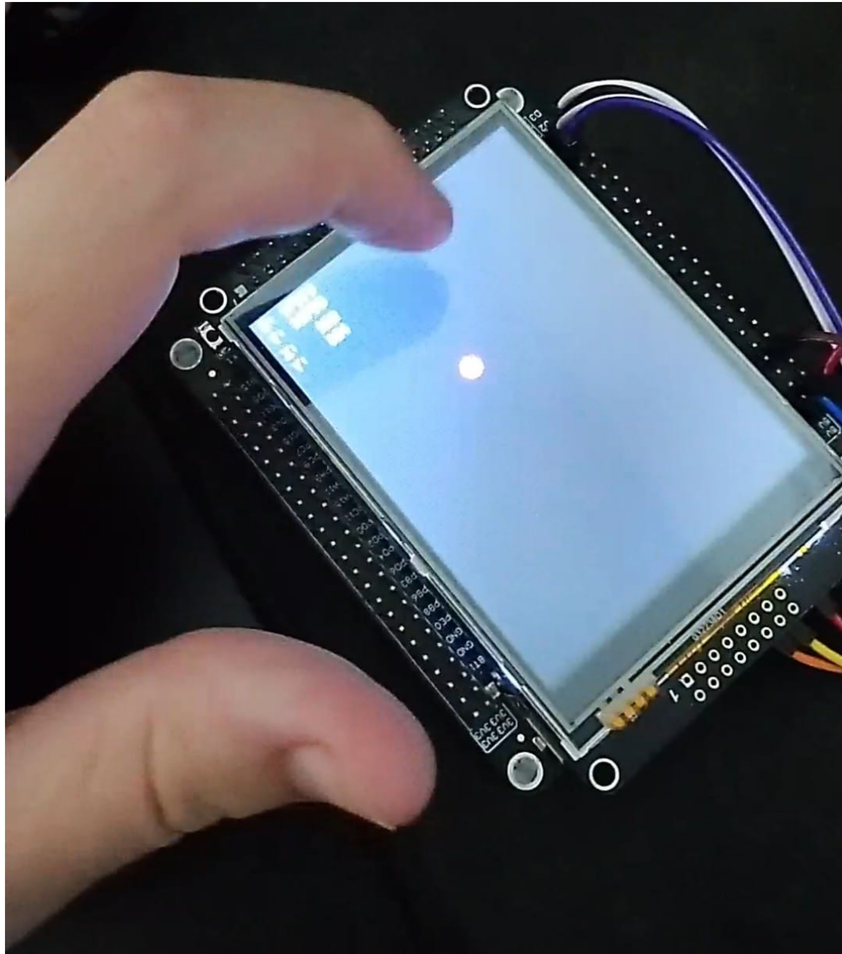
4.2.2 Ruído e Assincronismo no Controlador de Toque (SPI)

A validação inicial do sensor XPT2046, desenhando um círculo branco nas coordenadas lidas, resultou em leituras erráticas concentradas na lateral esquerda da tela. O diagnóstico revelou três anomalias distintas:

- **Transações SPI não atômicas**: o uso sequencial das funções de transmissão e recepção em uma interface full-duplex deixava bytes residuais no FIFO do periférico. O problema foi sanado substituindo-se a operação por uma chamada atômica de transmissão e recepção simultânea, garantindo a sincronização entre a requisição de comando e a resposta do ADC do XPT2046.
- **Inversão dupla do sinal de interrupção**: o pino de detecção de toque (T_PEN) possui lógica invertida (ativo em nível baixo). O BSP já realizava a correção lógica do sinal, mas a camada de entrada aplicava uma segunda negação desnecessária, resultando em leitura ininterrupta de ruído quando não havia toque e bloqueio da leitura durante a interação física.
- **Mascaramento de bits**: o ADC do controlador retorna um valor de 12 bits encapsulado em 16 bits. Um erro inicial no deslocamento das máscaras binárias causava truncamento dos dados brutos. A correção do algoritmo de deslocamento restabeleceu a precisão da leitura.

A Figura 26 apresenta o ensaio final do sensor XPT2046 após a correção das três anomalias.

Figura 26 – Ensaio final do sensor de toque XPT2046, com leituras consistentes após correção das anomalias.



Fonte: Elaborado pelo autor.

Como observado na Figura 26, após as três correções a leitura do toque passou a refletir corretamente a posição do dedo na tela, permitindo a interação fluida com a interface gráfica.

4.2.3 Integração da Biblioteca LVGL

A integração final ocorreu na ramificação principal do repositório. Os primeiros testes da interface apresentaram instabilidade, como campos numéricos que redefiniam seus valores ao serem pressionados e conflitos de parâmetros. A causa raiz foi atribuída à falta de sincronização entre os callbacks orientados a evento do LVGL e o modelo de dados em segundo plano (data binding). O alinhamento das tarefas de atualização de tela com a Fila de Trabalhos do gerenciador Modbus eliminou as condições de corrida, resultando em uma interface robusta e responsiva.

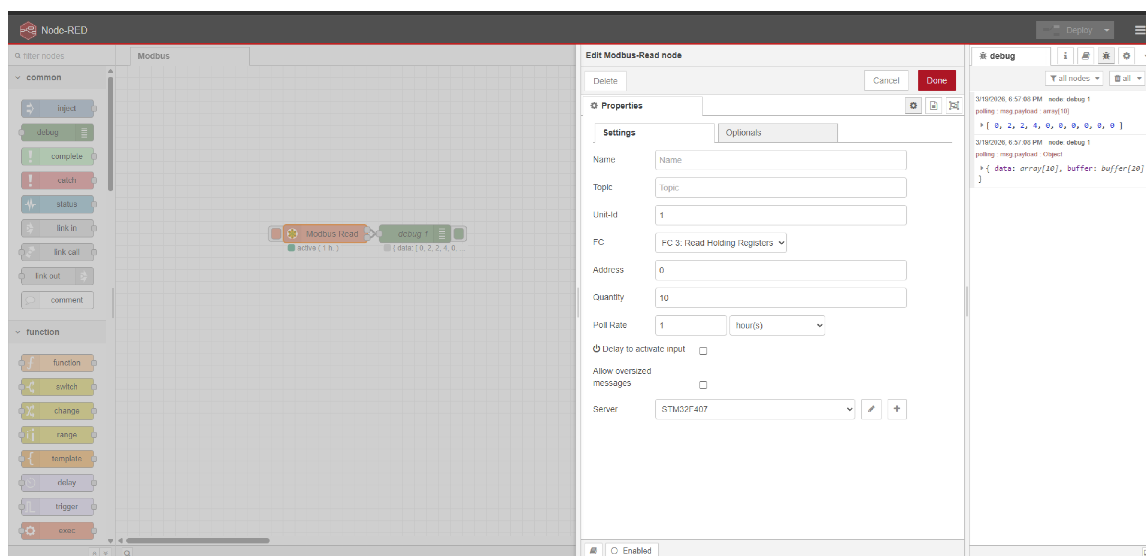
4.3 Validação dos Modos de Operação Modbus

Estabilizada a infraestrutura, os testes avançaram para a validação dos três modos de operação da ferramenta, modo Escravo, Mestre e Sniffer/datalogger. Esta última é a funcionalidade central de análise de redes, e por isso recebe maior detalhamento nesta seção.

4.3.1 Operação como Escravo (Integração FreeModbus)

Para validar a integração da biblioteca FreeModbus, o dispositivo foi configurado com o ID lógico 1 e conectado a um sistema supervisório simulado via plataforma Node-RED, utilizando o pacote node-red-contrib-modbus. O mestre remoto foi parametrizado para requisitar leituras periódicas de 1 Hz nos 10 primeiros Holding Registers. A Figura 27 apresenta a configuração do fluxo no Node-RED.

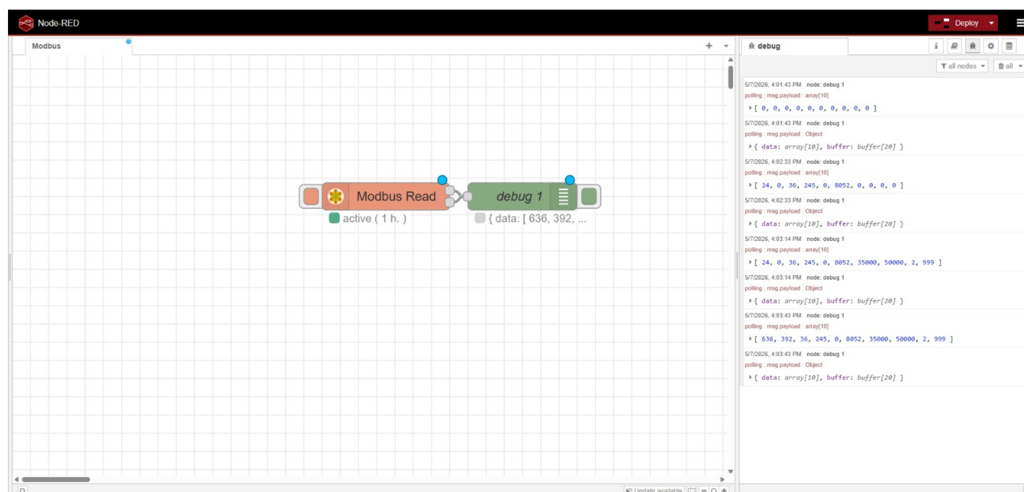
Figura 27 – Fluxo de teste do modo Escravo configurado na plataforma Node-RED.



Fonte: Elaborado pelo autor.

A Figura 28 apresenta o resultado da leitura periódica recebida pelo Node-RED, comprovando que o microcontrolador decodificou corretamente as requisições, acessou os vetores de memória da camada de aplicação e enviou respostas íntegras com os valores configurados pela interface gráfica do dispositivo.

Figura 28 – Resultado da leitura periódica dos 10 Holding Registers do dispositivo no modo Escravo, observado no Node-RED.



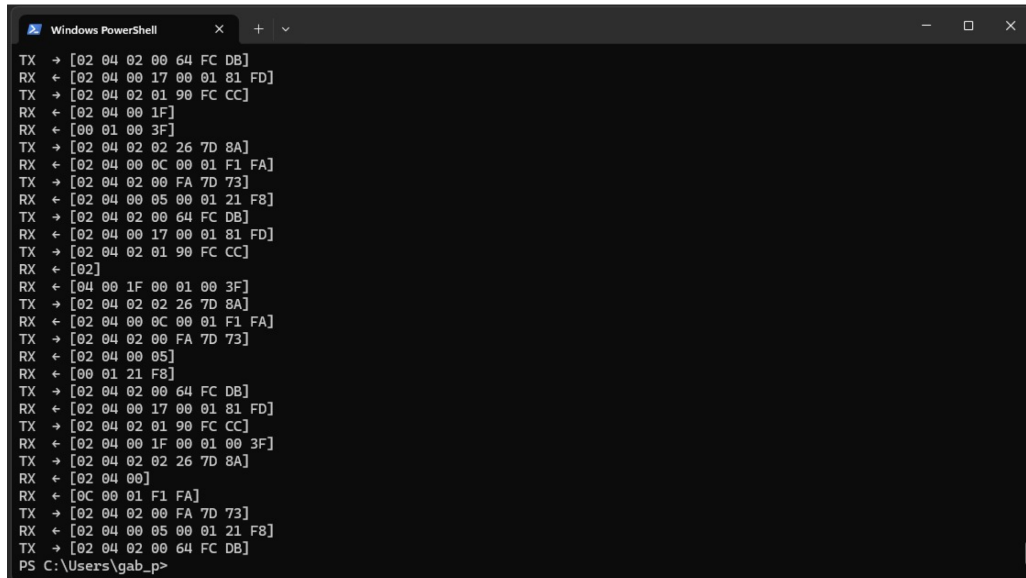
Fonte: Elaborado pelo autor.

A interface gráfica do dispositivo permite ao usuário modificar individualmente os registros de cada uma das quatro tabelas primárias do Modbus, os Coils (saídas digitais, leitura/escrita), Discrete Inputs (entradas digitais, somente leitura), Input Registers (registros analógicos de 16 bits, somente leitura) e Holding Registers (registros analógicos de 16 bits, leitura/escrita). O usuário seleciona o tipo de tabela, o endereço alvo e o novo valor a ser escrito.

4.3.2 Operação como Mestre Próprio (ModbusMaster)

A validação do Mestre Modbus exigiu um ensaio em malha fechada. Desenvolveu-se um script em Python que executa em um computador conectado via conversor USB-serial Waveshare, emulando o comportamento de um escravo genérico. O script valida os quadros recebidos quanto a CRC e endereçamento, e responde com valores predeterminados para cada código de função, permitindo verificar exaustivamente o comportamento do Mestre. A Figura 29 apresenta a saída do script durante um ensaio.

Figura 29 – Saída do script Python emulador de escravo durante a validação do Mestre, com quadros recebidos, CRC validado e respostas enviadas.



Fonte: Elaborado pelo autor.

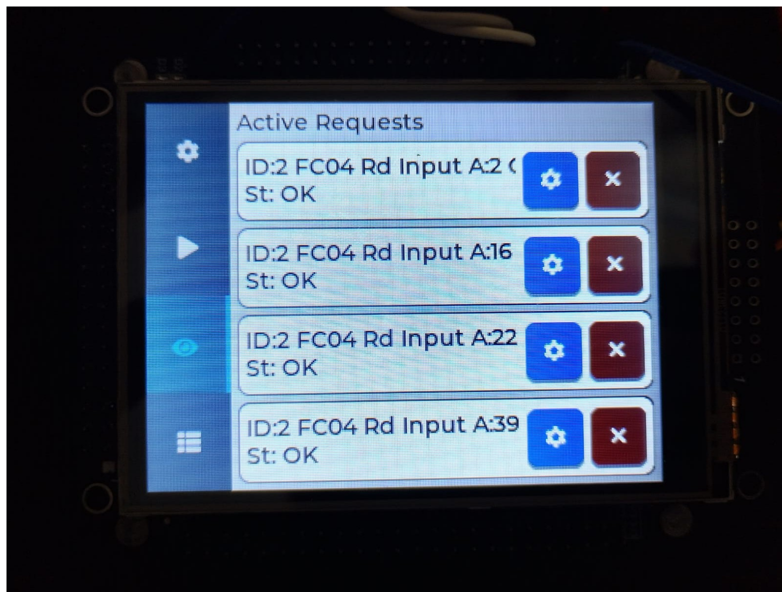
Conforme apresentado na Figura 29, o script confirmou a exatidão da montagem dos quadros pelo Mestre (estrutura da PDU, ordenação Big-Endian dos registradores e cálculo correto do CRC-16), além do respeito ao silêncio T3.5 entre quadros.

4.3.2.1 Agendamento de Múltiplas Requisições

Uma das principais capacidades do Mestre desenvolvido é o agendamento simultâneo de múltiplas requisições periódicas. O sistema mantém uma lista interna de tarefas registradas, cada uma com seus parâmetros (ID do escravo, código de função, endereço inicial, quantidade e período de repetição) e uma marca de tempo da próxima execução. A cada ciclo do laço principal, o agendador verifica quais tarefas atingiram o tempo de execução e as injeta na fila de requisições para envio ao barramento.

A interface gráfica permite que o usuário visualize, modifique e cancele essas tarefas dinamicamente. A Figura 30 apresenta um ensaio com várias requisições agendadas para diferentes registros de um mesmo escravo, ilustrando a capacidade de monitoramento simultâneo.

Figura 30 – Tela do dispositivo no modo Mestre com múltiplas requisições agendadas simultaneamente, para diferentes registros de um escravo.



Fonte: Elaborado pelo autor.

A Figura 30 evidencia que o agendador consegue alternar entre múltiplas requisições de leitura e escrita, respeitando os tempos de cada uma. O sistema foi limitado a 10 requisições simultâneas para garantir margem suficiente de processamento, e o tempo mínimo de repetição configurável é de 100 ms, valor escolhido para preservar a temporização T3.5 do barramento mesmo em cenários com vários escravos respondendo.

Para a escrita Modbus, ao selecionar o código de função apropriado (FC 06 para escrita única ou FC 16 para escrita múltipla) a interface abre dinamicamente os campos para entrada do valor a ser gravado. No caso do FC 16, o usuário primeiro informa a quantidade de registradores a escrever, e a interface gera os campos correspondentes para os valores.

4.3.3 Operação como Sniffer e Datalogger

A funcionalidade de Sniffer é o principal recurso de análise de redes da ferramenta. Diferentemente das funcionalidades anteriores (Mestre e Escravo), que exigem que o dispositivo participe ativamente da comunicação, o Sniffer permite registrar passivamente todo o tráfego do barramento, incluindo conversas entre dispositivos externos.

4.3.3.1 Topologia de Captura

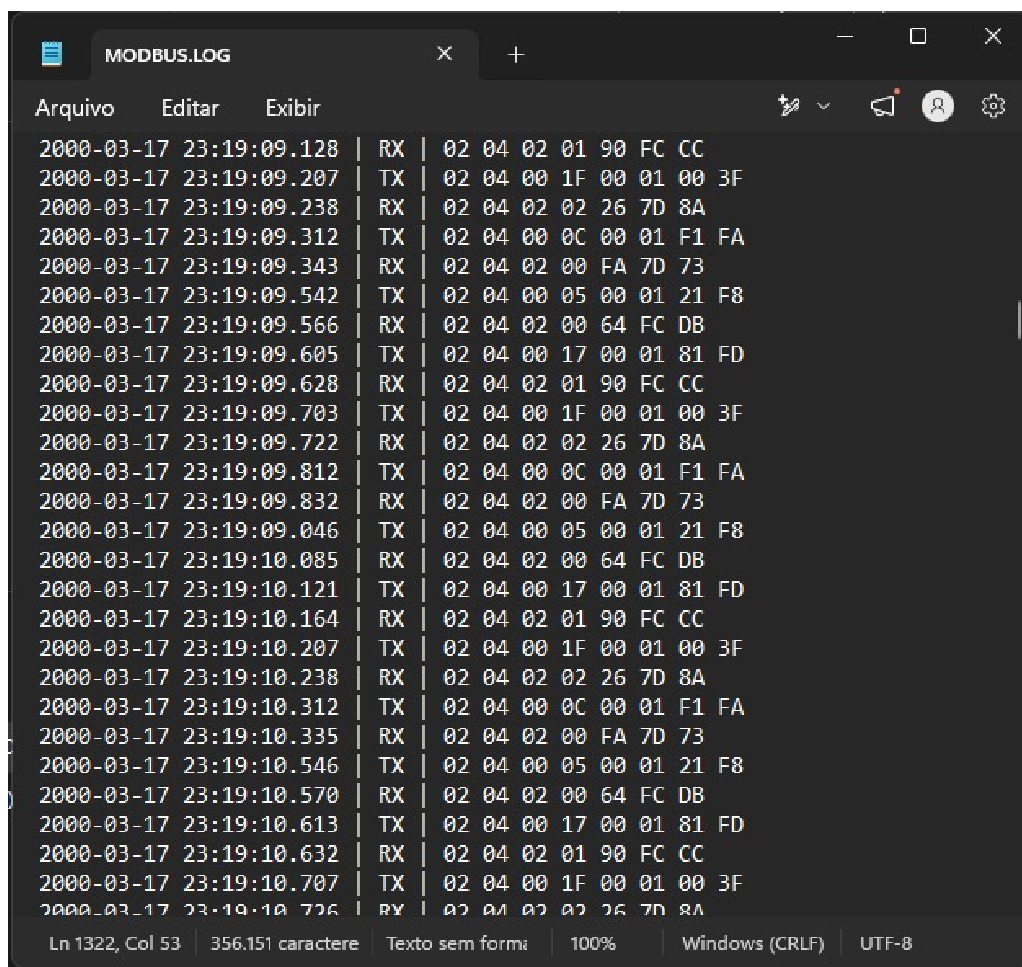
Para capturar uma comunicação entre dois ou mais equipamentos externos, o usuário conecta o dispositivo no mesmo barramento RS-485 (em paralelo) e o configura como Escravo com um ID lógico que não esteja em uso na rede. Nessa configuração,

o dispositivo recebe todos os bytes que trafegam no barramento, mas não responde a nenhuma requisição (já que nenhum mestre está endereçado a ele). A cada novo byte, o timer T3.5 é reiniciado; quando o silêncio do barramento atinge o limite, o quadro é considerado fechado, marcado com a direção (RX), recebe o timestamp do RTC e é enfileirado para gravação no cartão SD. O processo aplica-se de forma idêntica nos modos Mestre e Escravo ativos. Nesses casos, o Sniffer registra também os quadros enviados pelo próprio dispositivo (marcados como TX), conforme detalhado na Metodologia.

4.3.3.2 Formato e Capacidade do Log

O arquivo modbus.log é gravado em modo append no cartão SD, com uma linha por quadro contendo o timestamp absoluto (data e hora com resolução de aproximadamente 3,9 ms, limite do RTC do STM32F407), direção do quadro (RX ou TX) e conteúdo hexadecimal completo. A Figura 31 apresenta um trecho real do arquivo de log extraído do cartão SD após um ensaio.

Figura 31 – Trecho do arquivo modbus.log gravado pelo Sniffer no cartão SD, com timestamps, direção dos quadros e conteúdo hexadecimal.



Fonte: Elaborado pelo autor.

Conforme observado na Figura 31, cada quadro registrado contém todas as informações necessárias para análise posterior em ferramentas externas, o timestamp permite reconstruir a temporalidade da comunicação, a marcação de direção identifica a origem do quadro e o conteúdo hexadecimal completo possibilita decodificar manualmente o código de função, o endereço alvo e os dados transmitidos.

A gravação no cartão SD não interfere na recepção em tempo real do barramento. Isso foi alcançado por uma estratégia de desacoplamento, as rotinas de interrupção da UART apenas empurram os quadros completos para uma fila circular em RAM, e a escrita efetiva no cartão (operação bloqueante) ocorre no laço principal apenas em momentos sem requisições ativas, em ciclos de aproximadamente 10 segundos. Essa abordagem garante que mesmo gravações longas no SD não causem perda de quadros nem travamento da interface gráfica.

O arquivo de log cresce indefinidamente em modo append até o limite físico do cartão SD; quando o cartão se enche, o sistema simplesmente para de gravar novos quadros, sem sobrescrever os anteriores. Considerando um cartão de 16 GB e um tráfego típico de uma rede Modbus em operação, isso permite muitos dias de captura contínua antes que seja necessário esvaziar o cartão.

4.3.3.3 Limite de Velocidade Testado

Os ensaios com o Sniffer foram realizados em baudrates de 9600 a 115200 bps, valores típicos do Modbus RTU em ambientes industriais. Em todas as velocidades testadas, não foi observada perda perceptível de quadros nem travamento da interface gráfica, mesmo durante a captura contínua de vários minutos. O baudrate máximo de 115200 bps representa o limite efetivamente validado neste trabalho; baudrates superiores não foram avaliados pois extrapolam os valores usuais do protocolo.

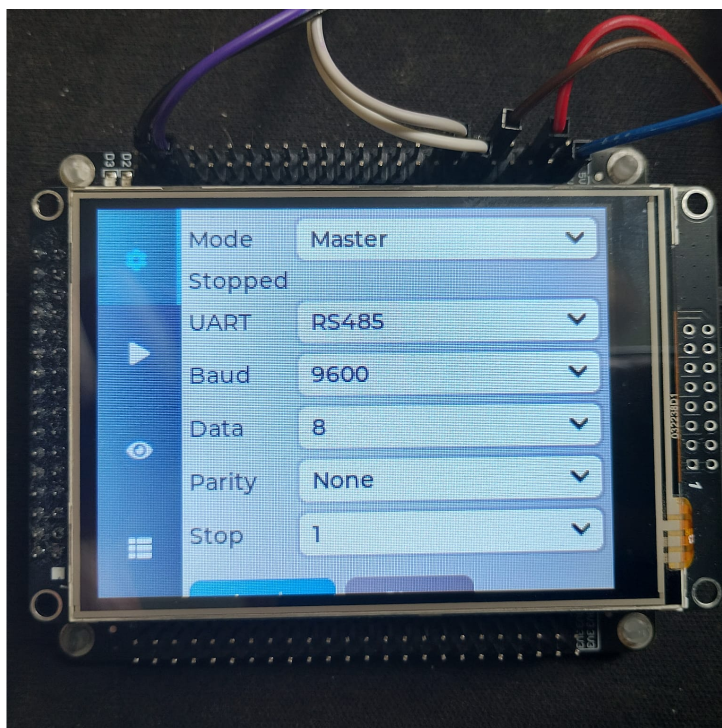
4.4 Interface Gráfica Final

A integração de todos os módulos resultou em uma interface gráfica de usuário fluida, responsiva e dividida em quatro abas funcionais que compõem o produto da ferramenta de diagnóstico.

4.4.1 Aba de Configuração de Rede

Permite a parametrização em tempo real da porta serial física (baudrate, paridade, stop bits), a seleção entre as interfaces RS-485 e RS-232 e a alternância dinâmica entre os modos de operação Mestre e Escravo Modbus. A confirmação dos parâmetros aciona a rotina de partida do gerenciador Modbus, que reconfigura todas as camadas em tempo de execução sem reiniciar o microcontrolador.

Figura 32 – Aba de configuração do barramento serial.



Fonte: Elaborado pelo autor.

4.4.2 Aba de Gerenciamento do Mestre

Habilitada ao operar no modo Mestre, oferece formulários para a criação de requisições de rede parametrizadas por código de função, endereço alvo e modo de execução (envio único ou cíclico). A lista de tarefas ativas permite o controle de periodicidade e o cancelamento de requisições em andamento, conforme apresentado na Figura 33.

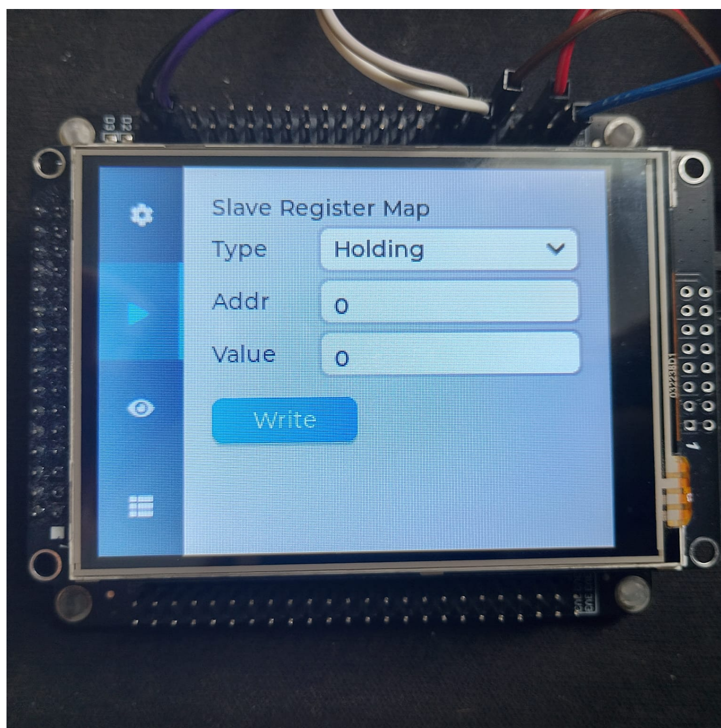
Figura 33 – Aba de requisição de funções no modo Mestre.

Fonte: Elaborado pelo autor.

4.4.3 Aba de Simulação de Escravo

Habilitada ao operar no modo Escravo, apresenta a interface para visualização e modificação manual dos valores alocados nos mapas de registradores internos, viabilizando a simulação de sensores e atuadores de campo. A Figura 34 apresenta a tela.

Figura 34 – Aba de modificação de registros no modo Escravo.



Fonte: Elaborado pelo autor.

4.4.4 Aba do Terminal de Diagnóstico (Sniffer)

Acessível em ambos os modos de operação, atua como visualização em tempo quase real dos dados do barramento. Exibe o histórico formatado dos últimos quadros capturados, marcados com direção (RX/TX), conteúdo hexadecimal e timestamp da transação. Esses dados refletem os mesmos logs gravados persistentemente no cartão SD. A atualização da lista é acionada manualmente pelo usuário através do botão de refresh, escolha de projeto que evita atualizações constantes da tela durante a leitura. A Figura 35 apresenta essa aba em operação.

Figura 35 – Aba do terminal de diagnóstico exibindo os últimos quadros capturados pelo Sniffer.

Fonte: Elaborado pelo autor.

4.5 Síntese das Capacidades da Ferramenta

Consolidando os resultados apresentados nas seções anteriores, a ferramenta validada neste trabalho é baseada no microcontrolador STM32F407VET6 (ARM Cortex-M4 a 168 MHz), suporta as interfaces seriais RS-232 (transceptor MAX3232) e RS-485 (transceptor MAX485) com seleção em tempo de execução, e foi efetivamente testada em baudrates entre 9600 e 115200 bps.

Em termos de funcionalidades Modbus, o dispositivo opera nos três modos previstos (Mestre, Escravo e Sniffer/datalogger) e suporta os códigos de função 01, 02, 03, 04, 05, 06, 15 e 16. No modo Escravo, foram alocados 512 Coils, 16 Discrete Inputs, 512 Input Registers e 512 Holding Registers, todos editáveis pela interface gráfica. No modo Mestre, podem ser agendadas até 10 requisições simultâneas, com período mínimo de 100 ms.

A captura de tráfego pelo Sniffer é registrada em um arquivo de log no cartão SD, acessado via SDIO em modo de 4 bits. Cada quadro recebe um timestamp do RTC, com resolução de aproximadamente 3,9 ms (limitação inerente do periférico do STM32F407). A interação com o usuário é realizada através de um display TFT de 3,2 polegadas (320×240) com toque resistivo XPT2046, e a validação funcional foi conduzida em duas frentes, utilizando o módulo Modbus do Node-RED como contraparte (modo Escravo) e um script Python desenvolvido para o projeto (modo Mestre).

4.6 Decisões de Projeto e Limitações da Ferramenta

Esta seção discute decisões de projeto que moldaram o produto final e aponta as limitações observadas, importantes para o contexto de uso da ferramenta e para nortear trabalhos futuros.

4.6.1 Adoção de Superloop Cooperativo em vez de RTOS

Embora o Referencial Teórico apresente conceitos de Sistemas Operacionais de Tempo Real, o firmware desenvolvido neste trabalho não utiliza um RTOS dedicado. A arquitetura adotada é a de um superloop cooperativo com máquinas de estado, altamente reativo a interrupções de hardware.

A intenção inicial do projeto considerava o uso de um RTOS para gerenciar a concorrência entre a comunicação Modbus e a interface gráfica. Contudo, ao longo do desenvolvimento, observou-se que o volume de troca de mensagens entre essas duas camadas, nos cenários de uso típicos da ferramenta, não justificou a complexidade adicional de gerenciar tarefas, semáforos e mutex de um RTOS. A solução cooperativa baseada em interrupções e máquinas de estado mostrou-se suficiente para atender aos requisitos temporais do Modbus RTU e à fluidez da interface gráfica, simplificando o projeto e o diagnóstico de falhas. A migração para um RTOS dedicado (FreeRTOS ou similar) é citada como trabalho futuro caso a ferramenta seja expandida para múltiplos barramentos simultâneos ou para a integração de funções remotas.

4.6.2 Limitações Observadas

As limitações listadas a seguir foram identificadas durante o desenvolvimento e os ensaios, sendo importantes para delimitar o escopo de aplicação da ferramenta:

- **Validação apenas com dispositivos simulados**: a validação foi realizada utilizando Node-RED e script Python como contrapartes da rede. Equipamentos industriais reais (CLPs, inversores, medidores) não foram testados neste trabalho, e essa validação adicional fortaleceria a confiabilidade da ferramenta em campo.
- **Ausência de geração automática de dados no Escravo**: no modo Escravo, os valores dos registradores são editados manualmente pelo usuário. Não há geração de valores aleatórios, incrementos automáticos ou simulação de comportamento dinâmico de sensores ao longo do tempo.
- **Ausência de configuração por arquivo**: todas as configurações (parâmetros seriais e tarefas do Mestre) são realizadas manualmente pela interface gráfica. Não há suporte para importação ou exportação de arquivos de configuração.

- **Ausência de acesso remoto**: a ferramenta não possui interface Wi-Fi, Bluetooth ou USB de dados. A leitura dos logs requer a remoção física do cartão SD para um computador externo.
- **Atualização manual da tela do Sniffer**: a aba de terminal não atualiza automaticamente os quadros capturados. O usuário precisa acionar manualmente o botão de refresh para visualizar as últimas mensagens.
- **Resolução do timestamp limitada a ∼3,9 ms**: limitação inerente da arquitetura do RTC do STM32F407, conforme descrito na Metodologia.
- **Ausência de terminação interna do barramento**: a terminação RS-485 (resistor de 120 Ω) não está integrada ao módulo MAX485 utilizado, exigindo que o usuário a adicione externamente em redes longas.
- **Alimentação cabeada**: o protótipo atual exige uma fonte externa de 5 V, sem suporte a bateria interna. A operação portátil é tratada como aprimoramento futuro.
- **Stress de tráfego não exaustivo**: embora não tenha sido observada perda de quadros nos ensaios realizados (até 115200 bps), o sistema não foi submetido a testes de stress agressivo (por exemplo, cenários com 10 requisições simultâneas a 100 ms cada e quadros de tamanho máximo). Tais ensaios são recomendados para caracterizar definitivamente a margem operacional da ferramenta.

4.6.3 Síntese dos Resultados

A ferramenta atendeu aos três modos de operação propostos, com validação funcional do Mestre, Escravo e Sniffer em diferentes baudrates. As decisões arquiteturais (HAL com Ponteiros Opacos, injeção de dependências para DE/RE e T3.5, desacoplamento entre interrupções e gravação no SD) provaram ser eficazes durante os ensaios, permitindo que o sistema operasse de forma estável com a interface gráfica responsiva. As limitações apontadas representam direcionamentos claros para a continuidade do trabalho, em especial a validação com equipamentos industriais reais e a expansão das capacidades do Escravo (geração dinâmica de dados) e do Sniffer (acesso remoto aos logs).

5 Conclusão

Este trabalho apresentou o desenvolvimento de uma ferramenta embarcada para diagnóstico e simulação de redes Modbus RTU, motivada pela necessidade prática de uma solução flexível e de baixo custo para a análise e o monitoramento de comunicações industriais. Considera-se que o objetivo geral foi atingido, o protótipo final operou de forma estável em seus três modos previstos, atuando como Mestre, Escravo e Sniffer/datalogger sobre as interfaces RS-232 e RS-485, com configuração dinâmica em tempo de execução pela interface gráfica touchscreen.

As decisões arquiteturais adotadas no firmware viabilizaram esses resultados. A construção de uma Camada de Abstração de Hardware (HAL) própria, baseada em Ponteiros Opacos e injeção de dependências, permitiu o desacoplamento entre a pilha Modbus, os periféricos do microcontrolador e a interface gráfica, facilitando a migração da primeira para a segunda placa de desenvolvimento e abrindo caminho para futuras portabilidades para outras famílias de microcontroladores. A adaptação da biblioteca FreeModbus atendeu aos requisitos de temporização do modo Escravo, enquanto o desenvolvimento de uma pilha Mestre própria, totalmente não-bloqueante, manteve a varredura da rede sem prejudicar a responsividade do sistema. O analisador de rede acoplado ao armazenamento em cartão SD operou de forma desacoplada das interrupções da UART, registrando o tráfego com timestamps sem perda perceptível de quadros até a velocidade máxima testada de 115200 bps. Toda a operação foi orquestrada por uma interface gráfica em LVGL, dividida em quatro abas funcionais que centralizaram a configuração e a visualização dos dados.

Reconhece-se também que o trabalho apresenta limitações importantes para o contexto de uso da ferramenta. A validação foi realizada exclusivamente com dispositivos simulados em software (Node-RED e script em Python), e ensaios com equipamentos industriais reais, como CLPs e medidores Modbus, fortaleceriam a confiabilidade do protótipo em campo. O modo Escravo, em sua versão atual, permite apenas a edição manual e individual dos registradores, sem geração dinâmica de valores que simule o comportamento de sensores reais ao longo do tempo. A leitura dos logs gravados no cartão SD ainda exige a remoção física do cartão, uma vez que não há suporte a comunicação remota (Wi-Fi, Bluetooth ou USB). Tais pontos, longe de invalidar os resultados obtidos, delimitam o escopo do que foi efetivamente entregue e norteiam os próximos passos do projeto.

No escopo de software, propõe-se como trabalhos futuros a validação da ferramenta em conjunto com equipamentos industriais reais, complementando os ensaios com simu-

ladores; a implementação de geração dinâmica de valores no modo Escravo (incremento, decremento e variação aleatória dentro de faixas configuráveis), tornando a simulação mais próxima do comportamento de sensores de campo; o desenvolvimento de uma rotina de varredura ativa de dispositivos presentes na rede, capaz de identificar automaticamente os IDs dos escravos respondentes; o suporte a configuração por arquivo, permitindo importar e exportar conjuntos de tarefas do Mestre e mapas de registradores do Escravo; e a adição de um menu para ajuste do RTC em tempo de execução, eliminando a necessidade de recompilação para alterar data e hora. Caso a ferramenta seja expandida para operar com múltiplos barramentos simultâneos ou para integrar funções de acesso remoto, será pertinente avaliar a migração da arquitetura atual de superloop cooperativo para um Sistema Operacional de Tempo Real (RTOS), que ofereceria escalonamento preemptivo e gerenciamento mais eficiente da concorrência entre comunicação, gravação e renderização gráfica.

No escopo de hardware, o principal trabalho futuro consiste no projeto e fabricação de uma placa de circuito impresso (PCB) dedicada, fundindo as características complementares observadas nas duas placas de desenvolvimento utilizadas neste trabalho. Essa PCB deve contemplar o roteamento otimizado para o display touchscreen, a integração nativa dos transceptores RS-485 e RS-232 com seus respectivos conectores, a inclusão de dip switches para terminação seletiva do barramento RS-485 (atualmente exigida apenas externamente) e os circuitos de proteção elétrica necessários para uso industrial. Adicionalmente, propõe-se a integração de uma bateria recarregável e de um circuito de gerenciamento de energia, eliminando a dependência atual de uma fonte cabeada de 5 V e tornando o dispositivo verdadeiramente portátil.

Em síntese, o trabalho entregou um protótipo funcional que cumpre os objetivos propostos e abre um conjunto claro de direcionamentos para sua continuidade. As decisões de arquitetura tomadas durante o desenvolvimento, somadas à documentação aberta do código-fonte no repositório público (SOUZA, 2026), oferecem uma base reutilizável para que essas evoluções, tanto de software quanto de hardware, possam ser incorporadas de forma incremental, aproximando a ferramenta de uma solução pronta para uso prático em campo.

Referências Bibliográficas

CHAN, E. *FAT Filesystem Specification*. 2021. Acesso em: 17 mar. 2026. Disponível em: <http://elm-chan.org/docs/fat_e.html>. Citado na página 31.

DENARDIN, G. W.; BARRIQUELLO, C. H. *Sistemas Operacionais de Tempo Real e sua Aplicação em Sistemas Embarcados*. Porto Alegre: Editora da UFRGS, 2019. Citado na página 30.

FILHO, G. F. *Automação de Processos e de Sistemas*. 1. ed. São Paulo: Editora Érica, 2014. Citado 2 vezes nas páginas 19 e 20.

HMS Networks. *Industrial Network Market Shares 2025*. 2025. Acesso em: 30 out. 2025. Disponível em: <https://www.hms-networks.com/news/news-details/27-05-2025-hms-networks-report-industrial-trends-2025>. Citado 2 vezes nas páginas 16 e 17.

JAKABOCZKI, G.; ADAMKO, E. Vulnerabilities of modbus rtu protocol – a case study. *University of Debrecen, Faculty of Engineering*, p. 2–3, 2015. Citado na página 17.

KUGELSTADT, T. *The RS-485 Design Guide*. Dallas, TX, 2021. Acesso em: 17 mar. 2026. Disponível em: <https://www.ti.com/lit/an/slla272d/slla272d.pdf>. Citado na página 23.

Modbus Organization. *MODBUS Application Protocol Specification V1.1b3*. [S.l.], 2012. Acesso em: 17 mar. 2026. Disponível em: <https://modbus.org/docs/Modbus_Application_Protocol_V1_1b3.pdf>. Citado 4 vezes nas páginas 24, 25, 26 e 27.

Modicon. *Modicon Modbus Protocol Reference Guide*. North Andover, MA, 1996. Acesso em: 17 mar. 2026. Citado na página 28.

MORAES, A. F. d. *Redes de Computadores*. 2. ed. São Paulo: Editora Érica, 2020. (Série Eixos). Citado na página 21.

NOVIELLO, C. *Mastering STM32*. [S.l.]: Leanpub, 2018. Citado 4 vezes nas páginas 28, 29, 30 e 31.

REYNDERS, D.; MACKAY, S.; WRIGHT, E. *Practical Industrial Data Communications: Best Practice Techniques*. 1. ed. Oxford, UK: Butterworth-Heinemann, 2004. Citado na página 24.

SEN, S. K. *Fieldbus and Networking in Process Automation*. 1. ed. Boca Raton, FL: CRC Press, 2014. Citado 2 vezes nas páginas 16 e 24.

SOUZA, G. F. d. *modbus_rtu_port*. 2026. Disponível em: <https://github.com/GabrielFSs/modbus_rtu_port>. Citado 2 vezes nas páginas 35 e 78.

Stack Exchange. *Is there any way to use Half Duplex RS485 without using a dedicated controller pin?* 2017. Acesso em: 19 mar. 2026. Disponível em: <https://electronics.stackexchange.com/questions/286263/is-there-any-way-to-use-half-duplex-rs485-without-using-a-dedicated-controller-p>. Citado na página 59.

STMicroelectronics. *Overview of the STM32F10xxx flexible static memory controller*. Genebra, 2008. Citado na página 31.

STMicroelectronics. *STM32MP1 Flexible Memory Controller (FMC)*. 2020. Material de treinamento. Acesso em: 17 mar. 2026. Citado na página 32.

STMicroelectronics. *STM32 Reference Manual*. 2024. Manual de referência da família STM32 (ex.: RM0090). Acesso em: 17 mar. 2026. Citado na página 29.

Texas Instruments. *Interface Circuits for TIA/EIA-232-F*. Dallas, 2001. Acesso em: 17 mar. 2026. Disponível em: <https://www.ti.com/lit/an/slla037a/slla037a.pdf>. Citado na página 22.