

EESC DCEA

Sandro Fadiga

**Depuração de Sistemas em Tempo Real: Uma Abordagem
de Instrumentação de Código para Testes de Sistemas
Críticos.**

São Carlos

2025

Sandro Fadiga

**Depuração de Sistemas em Tempo Real: Uma Abordagem
de Instrumentação de Código para Testes de Sistemas
Críticos.**

Monografia apresentada ao Curso de Especialização em Sistemas Aeronáuticos da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Especialista em Sistemas Aeronáuticos.

Orientador: Prof. Dr. Glauco Augusto de Paula Caurin

**São Carlos
2025**

AUTORIZO A REPRODUÇÃO TOTAL OU PARCIAL DESTE TRABALHO,
POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS
DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

Ficha catalográfica elaborada pela Biblioteca Prof. Dr. Sérgio Rodrigues Fontes da EESC/USP com os dados inseridos pelo(a) autor(a).

F145d	<p>Fadiga, Sandro Ferraz Martins Depuração de Sistemas em Tempo Real: Uma Abordagem de Instrumentação de Código para Testes de Sistemas Críticos. / Sandro Ferraz Martins Fadiga; orientador Glauco Augusto de Paula Caurin. São Carlos, 2025.</p> <p>Especialização (Especialização em Sistemas aeronáuticos) – Escola de Engenharia de São Carlos da Universidade de São Paulo, 2025.</p> <p>1. Sistemas embarcados. 2. Instrumentação de código. 3. Injeção de valores. 4. Monitoramento de variáveis. 5. Testes integrados. 6. Verificação de software. 7. Software crítico. 8. Software tempo real.</p> <p>I. Título.</p>
-------	---

Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE APROVAÇÃO

DEDICATÓRIA

Aos questionadores, aos inconformados, aos curiosos e aos rebeldes, aqueles que movem o mundo para fora de suas convicções e abrem novos caminhos para o conhecimento.

AGRADECIMENTOS

Agradeço, primeiramente, à minha família, por me incentivar constantemente na busca pelo conhecimento e por acreditar nas minhas capacidades, mesmo nos momentos desafiadores. Ao meu orientador, Prof. Dr. Glauco Augusto de Paula Caurin, pela dedicação, orientação precisa e valiosas contribuições que foram essenciais para o desenvolvimento deste trabalho. Aos demais professores do curso, pela transmissão de conhecimentos fundamentais e pelo suporte acadêmico fornecido ao longo desta jornada.

Aos colegas que tive a oportunidade de conhecer durante o curso, pelas trocas de conhecimentos, discussões enriquecedoras e pelo companheirismo que tornaram essa experiência ainda mais significativa.

Por fim, agradeço a todos que, de alguma forma, contribuíram para a realização deste trabalho.

*“Program testing can show the presence of bugs
but not the absence.”*

(Edger W. Dijkstra, EWD303)

RESUMO

FADIGA, S. F. M. **Depuração de Sistemas em Tempo Real:** Uma Abordagem de Instrumentação de Código para Testes de Sistemas Críticos. 2025. Monografia (Trabalho de Conclusão de Curso) – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2025.

O desenvolvimento de sistemas embarcados de tempo real apresenta desafios significativos no que se refere à verificação e validação de requisitos, especialmente em cenários de teste integrados, nos quais o acesso a estados internos do software é limitado. Ferramentas tradicionais de depuração, como sondas e interfaces JTAG, são amplamente utilizadas nas fases iniciais de desenvolvimento, porém apresentam restrições quando aplicadas a testes de integração ou à execução normal do sistema. Neste contexto, este trabalho propõe, implementa e avalia um protocolo simples de *peek* e *poke* voltado à observação e modificação controlada de estados internos de sistemas embarcados de tempo real, com o objetivo de apoiar atividades de depuração, teste e verificação. O protocolo permite a leitura e a escrita de variáveis internas durante a execução do software, sem a necessidade de interfaces de depuração proprietárias ou intrusivas. Como prova de conceito, o protocolo foi implementado em um microcontrolador de baixo custo e integrado a uma ferramenta cliente no host, possibilitando a interação com o sistema embarcado por meio de comandos determinísticos. O funcionamento da solução foi avaliado em cenários de teste representativos, demonstrando sua aplicabilidade como mecanismo auxiliar de depuração em ambientes de testes integrados. Os resultados indicam que a abordagem proposta pode contribuir para a inspeção de estados internos do software durante a execução, preservando o comportamento temporal do sistema e oferecendo suporte à verificação de requisitos em sistemas embarcados.

Palavras-chave: sistemas de tempo-real; testes integrados; *peek* e *poke*.

ABSTRACT

FADIGA, S. F. M. **Debugging Real-Time Systems: A Code Instrumentation Approach for Testing Critical Systems.** 2025. Monografia (Trabalho de Conclusão de Curso) – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2025.

The development of real-time embedded systems presents significant challenges regarding the verification and validation of requirements, especially in integrated testing scenarios, where access to internal software states is limited. Traditional debugging tools, such as probes and JTAG interfaces, are widely used in the early development phases, but they present restrictions when applied to integration testing or normal system operation. In this context, this work proposes, implements, and evaluates a simple *peek* and *poke* protocol aimed at the controlled observation and modification of internal states of real-time embedded systems, with the objective of supporting debugging, testing, and verification activities. The protocol enables reading and writing of internal variables during software execution, without the need for proprietary or intrusive debugging interfaces. As a proof of concept, the protocol was implemented on a low-cost microcontroller and integrated with a client tool on the host, enabling interaction with the embedded system through deterministic commands. The operation of the solution was evaluated in representative test scenarios, demonstrating its applicability as an auxiliary debugging mechanism in integrated testing environments. The results indicate that the proposed approach can contribute to the inspection of internal software states during execution, while preserving the system's temporal behavior and providing support for requirements verification in embedded systems.

Keywords: real-time systems; integration testing; *peek* and *poke*.

LISTA DE FIGURAS

Figura 1 – Diagrama de estados para a implementação do protocolo DESTRA	43
Figura 2 – Diagrama de sequencia de um comando peek a partir da ferramenta host	47
Figura 3 – Diagrama de sequencia de um comando poke a partir da ferramenta host	49
Figura 4 – Diagrama de sequência da operação peek. O fluxo ilustra as interações temporais entre a interface DESTRA UI, o módulo de protocolo Python, a comunicação serial UART, e a máquina de estados do sistema embarcado, demonstrando o ciclo completo request/response desde a requisição do usuário até a exibição dos dados recuperados da memória.	51
Figura 5 – Interface gráfica da ferramenta DESTRA UI. A tela apresenta os elementos principais para operação da ferramenta: (a) seleção de porta serial, (b) carregamento de arquivo ELF/DWARF, (c) painel de busca e seleção de variáveis, (d) tabela de monitoramento com operações de <i>peek</i> e <i>poke</i> , e (e) console de log para diagnóstico.	53
Figura 6 – Diagrama de sequência da operação de leitura do arquivo ELF/DWARF.	55
Figura 7 – Esquema do Arduino UNO. A imagem destaca os principais componentes: (a) microcontrolador ATmega328P, (b) conexão USB, (c) pinos digitais de I/O, (d) entradas analógicas, (e) botão de reset, (f) regulador de tensão, e (g) cristal oscilador de 16 MHz.	58
Figura 8 – Osciloscópio digital FNIRSI® DSO-153 2-em-1	61
Figura 9 – Latência ao longo do tempo (a)	69
Figura 10 – Distribuição da latência (b)	69
Figura 11 – Jitter ao longo do tempo (c)	70
Figura 12 – Análise estatística da latência (d)	70
Figura 13 – Latência ao longo do tempo (a)	72
Figura 14 – Distribuição da latência (b)	73
Figura 15 – Jitter ao longo do tempo (c)	73
Figura 16 – Análise estatística da latência (d)	74
Figura 17 – Latência ao longo do tempo (a)	75
Figura 18 – Distribuição da latência (b)	76
Figura 19 – Jitter ao longo do tempo (c)	76
Figura 20 – Análise estatística da latência (d)	77
Figura 21 – Setup do osciloscópio conectado ao Arduino é mostrado durante medição de sinais do Arduino, com a sonda conectada a um dos pinos de debug. A tela colorida TFT exibe a forma de onda capturada em tempo real, permitindo análise de frequência, jitter e tempo de processamento de comandos	79

Figura 22 – Medição de tempo de comando a 10 Hz.	80
Figura 23 – Medição de tempo de comando a 100 Hz.	82
Figura 24 – Medição de tempo de comando a 1 kHz.	84
Figura 25 – Medição de frame rate a 10 Hz. A forma de onda quadrada mostra a alternância do pino PIN_FRAME_TOGGLE a cada ciclo do loop. O período medido de 20.18 ms corresponde a dois ciclos completos (uma subida e uma descida).	87
Figura 26 – Medição de frame rate a 100 Hz. A forma de onda mantém a mesma periodicidade observada a 10 Hz, confirmando que o loop principal não é afetado pela taxa de envio de comandos.	89
Figura 27 – Medição de frame rate a 1 kHz. A forma de onda permanece idêntica às medições anteriores, demonstrando que o protocolo DESTRA não introduz degradação temporal no loop principal, mesmo sob taxa extrema de comandos.	91

LISTA DE TABELAS

Tabela 1 – Conjunto mínimo de casos de teste para cobertura MCDC	21
Tabela 2 – Estrutura do pacote do protocolo <i>peek</i> e <i>poke</i>	40
Tabela 3 – Variáveis da máquina de estados	45
Tabela 4 – Especificações técnicas do Arduino UNO	59
Tabela 5 – Especificações técnicas do osciloscópio digital FNIRSI® DSO-153 . .	62
Tabela 6 – Cenários de teste da ferramenta host	63
Tabela 7 – Métricas de performance coletadas	65
Tabela 8 – Arquivos de relatório gerados automaticamente pelo script de testes .	68
Tabela 9 – Dados de performance do teste de latência (host)	71
Tabela 10 – Dados de performance embarcada (teste de latência)	71
Tabela 11 – Resumo da análise do teste de latência	71
Tabela 12 – Dados de performance do teste de estresse (host)	74
Tabela 13 – Dados de performance embarcada (teste de estresse)	75
Tabela 14 – Resumo da análise do teste de estresse	75
Tabela 15 – Dados de performance do teste de rajada (host)	77
Tabela 16 – Dados de performance embarcada (teste de rajada)	78
Tabela 17 – Resumo da análise do teste de rajada	78
Tabela 18 – Métricas medidas com osciloscópio	79
Tabela 19 – Parâmetros de temporização medidos a 10 Hz — Teste de tempo de comando	81
Tabela 20 – Parâmetros de temporização medidos a 100 Hz — Teste de tempo de comando	83
Tabela 21 – Parâmetros de temporização medidos a 1 kHz — Teste de tempo de comando	85
Tabela 22 – Comparação dos parâmetros de temporização nas três taxas de envio de comando	86
Tabela 23 – Parâmetros de frame rate medidos a 10 Hz	88
Tabela 24 – Parâmetros de frame rate medidos a 100 Hz	90
Tabela 25 – Parâmetros de frame rate medidos a 1 kHz	92
Tabela 26 – Comparação consolidada dos parâmetros de frame rate nas três taxas de comando	92

SUMÁRIO

1	INTRODUÇÃO	19
1.1	Contextualização	19
1.2	Motivação	22
1.3	Objetivos	24
1.3.1	Objetivo Geral	24
1.3.2	Objetivos Específicos	24
1.4	Justificativa	24
1.5	Estrutura do Trabalho	24
2	FUNDAMENTAÇÃO	27
2.1	Software Embarcado e Sistemas Críticos	27
2.1.1	Características: tempo real, restrições de hardware e determinismo	27
2.1.2	Software Crítico de Segurança	28
2.1.3	Aplicações de Software Embarcado em Setores de Alta Criticidade	29
2.1.3.1	Setor Aeroespacial	29
2.1.3.2	Setor Automotivo	30
2.1.3.3	Setor Médico-Hospitalar	30
2.1.3.4	Desafios Comuns: Testes e Certificação	30
2.1.4	Importância da Previsibilidade Temporal	31
2.2	Testes em Sistemas Embarcados	32
2.3	Protocolos de Comunicação para Depuração	33
2.4	Operações Peek, Poke e Telemetria	34
2.5	Ambientes de Testes	36
2.6	Trabalhos Relacionados	37
3	DESENVOLVIMENTO DO PROTOCOLO	39
3.1	Arquitetura do Sistema	39
3.2	Especificação do Protocolo	39
3.2.1	Definições de Campos	40
3.2.2	Exemplos de Pacotes	41
3.3	Implementação no Microcontrolador	42
3.3.1	Pseudocódigo da Máquina de Estados	42
3.3.2	Variáveis da Máquina de Estados	45
3.3.3	Processamento do Comando PEEK	45
3.3.4	Processamento do Comando POKE	46
3.4	Implementação no Cliente Host	48

3.4.1	Arquitetura da Ferramenta Host	50
3.4.2	Procedimento Operacional	50
3.4.3	Formatos ELF e DWARF	52
3.5	Considerações sobre Extensibilidade	54
3.6	Resumo do Capítulo	54
4	METODOLOGIA E TESTES	57
4.1	Ambiente de Testes	57
4.1.1	Hardware - Arduino UNO	57
4.1.2	Arduino UNO	57
4.1.2.1	Especificações do Arduino UNO	58
4.1.3	Software Embarcado	58
4.1.3.1	Código de Teste	59
4.1.4	Ferramentas Auxiliares	60
4.1.4.1	Osciloscópio Digital FNIRSI® DSO-153	60
4.1.4.2	Especificações Técnicas do DSO-153	61
4.1.4.3	Ferramenta Host	61
4.2	Cenários de Teste	63
4.2.1	Cenário Host	63
4.2.2	Cenário Embarcado	64
4.2.2.1	Variáveis de Performance	64
4.2.2.2	Pinos de Debug para Osciloscópio	65
4.2.2.3	Métricas Coletadas	65
4.2.2.4	Programa Principal	65
4.3	Resultados Obtidos	66
4.3.1	Resultados para o Cenário 1	67
4.3.2	Arquivos de Relatório Gerados	67
4.3.3	Teste de Latência	68
4.3.3.1	Análise de resultados do teste de latência	71
4.3.4	Teste de Estresse	72
4.3.4.1	Análise dos resultados do teste de estresse	74
4.3.5	Teste de Rajada (Burst)	75
4.3.5.1	Análise dos resultados do teste de rajada	77
4.3.6	Resultados para o Cenário 2	78
4.3.6.1	Testes de Tempo de Comando	79
4.3.6.2	Medição a 10 Hz	79
4.3.6.3	Análise dos Resultados a 10 Hz	80
4.3.6.4	Medição a 100 Hz	81
4.3.6.5	Análise dos Resultados a 100 Hz	82
4.3.6.6	Medição a 1 kHz	83

4.3.6.7	Análise dos Resultados a 1 kHz	84
4.3.6.8	Síntese Comparativa das Três Medições	85
4.3.6.9	Testes de Frame Rate	86
4.3.6.10	Medição de Frame Rate a 10 Hz	86
4.3.6.11	Medição de Frame Rate a 100 Hz	87
4.3.6.12	Medição de Frame Rate a 1 kHz	90
4.3.6.13	Análise Consolidada dos Testes de Frame Rate	91
4.3.6.14	Observações Críticas	92
4.4	Análise Geral dos Testes com Osciloscópio	93
4.5	Conclusões dos Testes	93
5	CONCLUSÃO E TRABALHOS FUTUROS	95
5.1	Considerações Finais	95
5.2	Conclusões sobre o Protocolo Desenvolvido	95
5.3	Relevância para o Contexto de Certificação	95
5.4	Limitações Observadas	96
5.4.1	Ausência de Operações Contínuas	96
5.4.2	Falta de Mecanismos de Integridade	96
5.4.3	Limitações de Arquitetura	97
5.5	Trabalhos Futuros	97
5.6	Perspectivas Finais	98
5.7	Resumo das Contribuições	98
	REFERÊNCIAS	99
	APÊNDICE A – REPOSITÓRIO DO PROJETO DESTRA	101
A.0.1	Estrutura do Repositório	101
A.0.2	Reprodutibilidade	102

1 INTRODUÇÃO

1.1 Contextualização

A depuração de sistemas embarcados críticos de tempo real é de suma importância, pois assegura a segurança, a confiabilidade e a aderência aos requisitos de sistema. No que diz respeito aos requisitos de sistemas que devem ser desdobrados em requisitos de software para sistemas críticos, como os sistemas aeroespaciais aderentes à norma DO-178C, a depender do nível de criticidade do sistema em desenvolvimento é necessário comprovar a rastreabilidade do código associado aos requisitos de software para a devida geração de evidência de cobertura de requisitos.

O desafio, portanto, está associado a como, em testes de caixa preta, comprovar que requisitos estão sendo cumpridos. Para tais testes, usualmente estimula-se o sistema com entradas e coletam-se dados na saída. Este tipo de teste cobre requisitos de sistema no nível de integração de sistemas; porém, estes não são os únicos requisitos a serem verificados em um sistema crítico.

Muitos requisitos detalham estados ou cálculos intermediários de um sistema, de forma que sua aplicação nem sempre é passível de observação na saída do sistema. Existem ainda requisitos emergentes ou derivados, que surgem da necessidade de se criar uma solução no software para resolver um problema gerado pela solução de um requisito de sistema. Para tais requisitos, não é possível realizar medições nem, muitas vezes, estímulos de entrada em um teste de caixa preta.

Em sistemas de tempo real, cálculos e estados internos ao sistema estão associados à dimensão do tempo, usualmente sendo considerado o tempo de execução, ciclo ou de taxa como o tempo máximo permitido para a execução completa do ciclo do software, ou seja:

1. Leitura e processamento de entradas;
2. Cálculos com valores provenientes destas entradas (e valores provenientes de estados anteriores);
3. Gravação em saída dos valores de cálculo.

Neste modelo de execução, o software está submetido aos requisitos que definem margens de segurança, como o *Worst-Case Execution Time* (WCET), o qual define quanto tempo o software pode ocupar no ciclo de execução total do sistema. Este tipo de requisito reserva um slot de tempo no sistema para o qual o software poderá ocupar na execução

do seu ciclo. É necessária uma garantia de que o software em nenhum momento exceda o tempo total alocado, o qual vem acompanhado de uma margem de segurança.

Além da análise de WCET, são empregadas análises de cobertura estrutural do código, como o *Modified Condition/Decision Coverage* (MCDC). Esse critério de cobertura é amplamente adotado em sistemas embarcados críticos, especialmente no contexto da certificação aeronáutica, conforme definido pela norma DO-178C (RTCA, 2011).

O critério MCDC é considerado um dos métodos mais rigorosos de cobertura estrutural, pois exige a verificação detalhada do impacto de cada condição lógica individual no resultado de uma decisão composta, contribuindo para a detecção de falhas que não seriam identificadas por critérios de cobertura menos restritivos (Rierson, 2013).

De forma mais específica, o MCDC impõe que os seguintes requisitos sejam atendidos:

1. Cada condição atômica (subexpressão booleana dentro de uma decisão) seja avaliada como verdadeira e falsa ao menos uma vez;
2. Cada decisão composta seja avaliada como verdadeira e falsa ao menos uma vez;
3. Cada condição atômica demonstre influência independente sobre o resultado da decisão, exigindo a construção de casos de teste nos quais a alteração isolada de uma condição modifique o resultado final da decisão.

Considerando, por exemplo, a decisão lógica apresentada na Listagem 1.1, composta pelas condições atômicas *A*, *B* e *C*, a aplicação do critério MCDC requer a definição de casos de teste que demonstrem que cada uma dessas condições influencia de forma independente o resultado da decisão.

Listagem 1.1 – Exemplo de decisão lógica para análise de MCDC

```
bool evaluate_decision(bool A, bool B, bool C) {  
    if ((A && B) || C) {  
        return true;  
    } else {  
        return false;  
    }  
}
```

Para se obter a cobertura completa de MCDC para as condições definidas no código apresentado na Listagem 1.1, é necessário, no mínimo, um conjunto de quatro casos de teste. Esse conjunto deve ser cuidadosamente projetado de forma a demonstrar, de maneira independente, a influência de cada condição atômica (*A*, *B* e *C*) no resultado da expressão booleana avaliada.

Em conformidade com o critério MCDC, cada condição deve assumir os valores verdadeiro e falso pelo menos uma vez, e deve ser possível evidenciar que a variação isolada de uma única condição é suficiente para alterar o resultado da decisão, mantendo-se as demais condições constantes.

1. Condição A: Caso de teste quando apenas A difere, e a decisão total muda;
2. Condição B: Caso de teste quando apenas B difere, e a decisão total muda;
3. Condição C: Caso de teste quando apenas C difere, e a decisão total muda.

A Tabela 1 apresenta o conjunto mínimo de casos de teste necessário para satisfazer esses critérios de cobertura.

Teste	A	B	C	Resultado
T1	F	V	F	F
T2	V	V	F	V
T3	V	F	F	F
T4	V	F	V	V

Tabela 1 – Conjunto mínimo de casos de teste para cobertura MCDC

Portanto, como podemos verificar se requisitos de sistema, requisitos emergentes e de segurança estão sendo cumpridos pelo software, é necessária a verificação de valores e estados internos do software ao longo do tempo de execução. Assim, a questão central é: Como verificar tais valores sem interferir diretamente nos cálculos de valores e estados do sistema?

Ferramentas tradicionais de *debug* são amplamente utilizadas na indústria de sistemas embarcados, sobretudo nos estágios iniciais do desenvolvimento do software sendo muito úteis para eliminar falhas e detectar erros iniciais de codificação dos requisitos. Ao se escolher a solução de hardware para o sistema, seleciona-se em conjunto, ou mesmo é fornecido pelo fabricante, um kit de desenvolvimento de software composto por:

- Compilador para a linguagem de alto nível (normalmente C ou C++) para o hardware alvo;
- Algum mecanismo de comunicação entre o computador usado para o desenvolvimento do software e o hardware alvo;
- Um sistema de leitura e injeção de valores em memória do hardware;
- Alguma ferramenta de software para visualização e *debug* de código, fazendo uso desta infraestrutura de comunicação e leitura/injeção de valores.

Contudo, as ferramentas de depuração fornecidas pelos fabricantes de microcontroladores, normalmente baseadas em interfaces como JTAG (Joint Test Action Group), SWD (Serial Wire Debug) ou hardware/sondas proprietárias são utilizadas em sua maioria durante as fases iniciais do desenvolvimento do software e seu uso é voltado para identificar e corrigir erros cometidos na codificação dos requisitos de baixo nível (ou nível de software), durante esta etapa testes formais ainda estão em etapa de desenvolvimento — tanto o software quanto os testes partem de um conjunto de requisitos em comum — que acontece em paralelo entre desenvolvimento e verificação.

Além disso, métodos convencionais de depuração — que pausam a execução ou utilizam single-stepping introduzem interferência no comportamento temporal do sistema, alterando sua dinâmica de execução e dificultando a verificação em tempo real que testes de requisitos e testes de integração com hardware na planta real necessitam manter.

Diante dessas limitações, torna-se evidente a necessidade de uma solução de depuração não intrusiva, padronizada e automatizável — capaz de operar sem dependência de sondas específicas nem de interfaces proprietárias, e que permita a execução de testes em ambientes representativos do sistema final.

1.2 Motivação

A depuração de sistemas embarcados críticos, especialmente aqueles de tempo real, enfrenta limitações quando se utilizam apenas técnicas tradicionais de testes de caixa preta. Muitos requisitos, incluindo requisitos derivados ou emergentes, só podem ser verificados por meio da observação de estados e valores internos do software durante sua execução. Ferramentas de depuração fornecidas pelos fabricantes, embora úteis, são geralmente complexas, custosas e voltadas ao desenvolvimento inicial, não sendo adequadas para fases de integração ou testes finais.

Nesse contexto, torna-se relevante desenvolver soluções específicas que permitam a inspeção controlada de estados internos do sistema sem comprometer sua execução. As operações de inspeção e modificação remota de memória — tradicionalmente conhecidas como *peek* (leitura) e *poke* (escrita) — constituem um dos mecanismos mais antigos e difundidos no contexto da depuração de sistemas computacionais e embarcados. Inicialmente popularizadas em plataformas de microcomputadores e linguagens interpretadas, essas operações foram posteriormente incorporadas ao desenvolvimento de sistemas embarcados como um meio simples e direto de acessar variáveis internas durante a execução (Dorfman, 2008). Apesar das evoluções significativas em ferramentas e protocolos ao longo dos anos, o conceito permanece central para arquiteturas de depuração, sendo frequentemente encapsulado em interfaces padronizadas, APIs ou camadas de abstração (Singh, 2020).

Entretanto, em sistemas embarcados críticos — como os empregados na indústria

aeroespacial, automotiva ou médico-hospitalar — tais soluções não podem ser utilizadas diretamente, uma vez que não atendem aos requisitos de certificação impostos por normas regulatórias internacionais. No setor aeronáutico, por exemplo, o DO-178C (Software Considerations in Airborne Systems and Equipment Certification) estabelece critérios rigorosos de desenvolvimento, validação e rastreabilidade de requisitos para softwares embarcados. De forma complementar, o DO-330 (Software Tool Qualification Considerations) regulamenta o uso de ferramentas de apoio, exigindo qualificação formal caso seus resultados interfiram em atividades de verificação.

Embora a literatura acadêmica apresente limitações quanto à documentação formal de técnicas leves de depuração em plataformas de prototipagem, observa-se na prática que dispositivos amplamente utilizados, como Arduino e ESP32, incorporam utilitários de depuração em tempo real em seus ambientes de desenvolvimento. Tais mecanismos — muitas vezes implementados por meio de operações simples de leitura e escrita em memória (*peek* e *poke*) ou via interfaces seriais — têm sido empregados por desenvolvedores para acelerar o processo de depuração e teste durante o protótipo inicial de aplicações embarcadas. Apesar de não constituírem soluções completas de depuração nem substituírem métodos estruturados, essas ferramentas ilustram a demanda crescente por mecanismos acessíveis de inspeção e interação em tempo de execução, especialmente em projetos de propósito geral.

De maneira análoga, outras indústrias impõem normas específicas, como a ISO 26262 no setor automotivo, a IEC 62304 para dispositivos médicos e a IEC 61508 em sistemas de segurança funcional. Diante dessas exigências, fabricantes de microcontroladores e plataformas comerciais não disponibilizam, em geral, soluções de depuração do tipo *peek* e *poke* aptas a serem empregadas diretamente em sistemas críticos.

Ferramentas amplamente difundidas, como o Arduino Serial Monitor, o ESPIDF Monitor ou ambientes de desenvolvimento integrados (IDEs) fornecidos por fornecedores como STMicroelectronics (STM32CubeIDE) e Espressif, não contemplam processos de certificação de software e tampouco oferecem rastreabilidade de requisitos ou cobertura de testes exigida para níveis de criticidade elevados (DAL A e B, conforme o DO-178C).

Dessa forma, torna-se necessária a criação interna (*in-house*) de soluções de depuração adaptadas (*tailored*) ao sistema em desenvolvimento, de modo que o protocolo de comunicação, a arquitetura de software embarcado e as ferramentas de suporte sejam especificados, implementados e verificados em conformidade com o nível de garantia de desenvolvimento (Design Assurance Level - DAL) aplicável. Esse processo garante que cada requisito funcional seja rastreado, testado e documentado, permitindo que a ferramenta de depuração não comprometa a conformidade regulatória nem a integridade do sistema em operação.

1.3 Objetivos

1.3.1 Objetivo Geral

Propor, implementar e validar um protocolo simples de *peek* e *poke* para observação e modificação controlada de estados internos em sistemas embarcados de tempo real, a fim de apoiar atividades de teste e verificação.

1.3.2 Objetivos Específicos

- Estudar os conceitos de depuração em sistemas embarcados e protocolos de comunicação aplicáveis;
- Definir os requisitos e a arquitetura de um protocolo de *peek* e *poke*;
- Implementar o protocolo em um microcontrolador de baixo custo (ex.: Arduino);
- Desenvolver uma ferramenta cliente para interação com o protocolo;
- Avaliar o funcionamento do protocolo em cenários de teste representativos.

1.4 Justificativa

A proposta deste trabalho se justifica pela necessidade de ferramentas acessíveis, específicas e de baixo custo para o teste e depuração de sistemas embarcados críticos. Diferentemente de soluções comerciais, que exigem infraestrutura complexa, o protocolo aqui desenvolvido visa oferecer uma alternativa prática tanto em contextos acadêmicos quanto em aplicações industriais. Assim, contribui para a formação de profissionais em sistemas embarcados, possibilita o aprofundamento em técnicas de teste e depuração, e ainda pode ser expandido em pesquisas futuras, incluindo integração com ambientes de simulação e ferramentas de verificação formal.

1.5 Estrutura do Trabalho

Este trabalho está organizado da seguinte forma:

- **Capítulo 1 – Introdução:** apresenta a contextualização do tema, a motivação, os objetivos, a justificativa e a organização do trabalho.
- **Capítulo 2 – Fundamentação Teórica:** descreve os conceitos de sistemas embarcados de tempo real, técnicas de depuração e protocolos de comunicação, além de trabalhos relacionados.
- **Capítulo 3 – Desenvolvimento do Protocolo:** detalha a arquitetura, a especificação e a implementação do protocolo proposto, tanto no microcontrolador quanto na ferramenta cliente.

- **Capítulo 4 – Metodologia e Testes:** apresenta o ambiente de experimentação, a metodologia de avaliação e os resultados obtidos.
- **Capítulo 5 – Conclusão e Trabalhos Futuros:** reúne as conclusões do trabalho, suas limitações e as perspectivas de continuidade.

2 FUNDAMENTAÇÃO

2.1 Software Embocado e Sistemas Críticos

Segundo (Electrical; Engineers, 2000), software embarcado é aquele projetado para operar em um sistema computacional integrado a um sistema maior, geralmente sujeito a restrições de tempo real, memória e energia, além de interagir diretamente com o ambiente físico por meio de sensores e atuadores.

De forma mais simples, pode-se afirmar que o software embarcado é desenvolvido de maneira conjunta ao hardware no qual será executado. Em contraste com softwares voltados a computadores pessoais — nos quais a portabilidade entre plataformas similares é frequentemente possível, mesmo diante de diferenças de hardware — os sistemas embarcados são projetados para solucionar problemas de engenharia específicos. Consequentemente, o software precisa ser concebido de modo a satisfazer requisitos funcionais, temporais e estruturais definidos pela aplicação final, refletindo a forte dependência entre software e hardware (Laplante; Ovaska, 2011).

2.1.1 Características: tempo real, restrições de hardware e determinismo

Um software embarcado apresenta diversas características fundamentais, entre elas o comportamento de *tempo real* e o *determinismo*. É comum associar o termo “tempo real” ao tempo cronológico natural (tempo de relógio). Entretanto, em sistemas embarcados, tempo real refere-se à capacidade do sistema de produzir respostas corretas dentro de limites temporais previamente especificados, definidos ainda na fase de projeto (Kopetz, 2011).

Dessa forma, um sistema de tempo real não é caracterizado apenas por sua rapidez, mas principalmente pela previsibilidade temporal de sua execução. O sistema deve garantir que o tempo necessário para a execução de cada ciclo de processamento não ultrapasse o tempo máximo alocado para essas operações, considerando margens de segurança adequadas (Burns; Wellings, 2001).

Cada sistema embarcado possui janelas temporais definidas com base em análises de desempenho, taxas de amostragem, processamento de entradas e saídas e requisitos de segurança. Nesse contexto, o determinismo desempenha papel central, uma vez que assegura que, para um mesmo conjunto de entradas e condições iniciais, o sistema apresente comportamento temporal e funcional previsível, independentemente de variações internas de execução (Wilhelm *et al.*, 2008).

Uma consequência direta dessas características é o conceito de *criticidade*, frequentemente associado a sistemas embarcados de tempo real. A criticidade, entretanto,

está relacionada ao impacto potencial de falhas do sistema sobre pessoas, bens ou o meio ambiente, e não exclusivamente ao software isoladamente. Em sistemas críticos de segurança (*safety-critical systems*), falhas de software podem contribuir diretamente para condições perigosas, o que justifica a adoção de normas rigorosas de desenvolvimento, verificação e validação (Rierson, 2013).

2.1.2 Software Crítico de Segurança

Segundo (Rierson, 2013), software crítico de segurança (*safety-critical software*) pode ser definido a partir do conceito mais amplo de segurança. Uma definição geral para segurança é a “ausência de condições que possam causar morte, lesão, enfermidade, dano ou perda de equipamentos ou propriedade, ou prejuízo ao meio ambiente” (Rierson, 2013). Já a definição específica de software crítico de segurança é mais subjetiva.

O Institute of Electrical and Electronic Engineers (IEEE) define software crítico de segurança como:

“Software cujo uso em um sistema pode resultar em risco inaceitável. Software crítico de segurança inclui software cuja operação, ou falha em operar, pode levar a um estado perigoso, software destinado a recuperar-se de estados perigosos e software destinado a mitigar a severidade de um acidente”.

O *NASA Software Safety Standard* identifica um software como crítico de segurança quando ao menos um dos seguintes critérios é satisfeito:

- O software causa ou contribui para uma condição de sistema que, caso não seja controlada, possa resultar em morte, lesão grave ou danos significativos ao equipamento, à propriedade ou ao meio ambiente.
- O software é responsável por detectar, monitorar ou controlar estados perigosos, condições anômalas ou situações que envolvam risco para pessoas, equipamentos ou missões.
- O software executa funções necessárias para mitigar, isolar ou responder a falhas do sistema ou do hardware, prevenindo que tais falhas se tornem condições inseguras.
- O software fornece informações críticas utilizadas para tomada de decisão operacional, segurança de voo, controle de missão ou manobras que, se incorretas, podem provocar um resultado perigoso.
- O software é utilizado em sistemas de apoio à missão, simulação, teste ou validação, cujo comportamento incorreto possa induzir erros em sistemas críticos de voo, operação ou segurança.

Portanto quando abordado neste texto os termos mais abrangentes como software embarcado ou software de tempo real estarão referindo-se ao termo mais específico e de interesse deste trabalho: software crítico de segurança.

2.1.3 Aplicações de Software Embarcado em Setores de Alta Criticidade

A crescente complexidade dos produtos tecnológicos, associada à pressão por ciclos de desenvolvimento mais curtos e maior capacidade de atualização, tem impulsionado o uso intensivo de software embarcado em diferentes segmentos industriais. O software, por ser mais flexível, reconfigurável e economicamente vantajoso do que soluções puramente baseadas em hardware, tornou-se elemento central no projeto de sistemas modernos.

Embora diversos setores utilizem software embarcado, três deles se destacam pela elevada dependência de funções automatizadas e pelo impacto direto sobre a segurança humana: o setor aeroespacial, o automotivo e o médico-hospitalar. Esses domínios compartilham um aspecto fundamental: a necessidade de garantir níveis rigorosos de confiabilidade e segurança, o que exige a adoção de normas específicas para desenvolvimento, verificação, validação e certificação de software crítico.

2.1.3.1 Setor Aeroespacial

No setor aeroespacial, especialmente na aviação comercial, o software embarcado está presente em uma ampla gama de sistemas, desde módulos de entretenimento de cabine até sistemas de controle de voo, navegação, freios e gerenciamento de potência. A criticidade de cada função determina o rigor necessário no processo de desenvolvimento, sendo esses níveis formalizados pelo *Design Assurance Level* (DAL), conforme estabelecido pela norma DO-178C.

Os níveis de criticidade variam de DAL A a DAL E, onde:

- **DAL A - Catastrófico:** condições de falha que podem resultar em catástrofe.
- **DAL B - Severa:** condições de falha que podem resultar em falha severa do sistema, porém não catastrófica.
- **DAL C - Maior:** condições de falha que podem resultar em degradação importante do desempenho.
- **DAL D - Menor:** falhas que têm efeitos menores sobre a operação, inconveniências.
- **DAL E - Sem Efeito:** falhas sem impacto na segurança operacional.

Sistemas classificados nos níveis A e B exigem evidências extensivas de verificação, rastreabilidade e cobertura estrutural, refletindo a criticidade de suas funções. Eventos

recentes na indústria reforçam a importância de processos rigorosos de certificação e verificação de software em sistemas de controle de voo.

2.1.3.2 Setor Automotivo

No setor automotivo, a incorporação de sistemas embarcados evoluiu drasticamente nas últimas décadas, abrangendo desde sistemas de infoentretenimento até algoritmos de controle de estabilidade, freios ABS, *airbags*, direção assistida e funções autônomas. Da mesma forma que na aviação, a segurança dos ocupantes e ambiente em torno ao veículo é o fator determinante para o rigor do processo de desenvolvimento.

A norma de referência nesse domínio é a ISO 26262, que define os níveis de criticidade do *Automotive Safety Integrity Level* (ASIL), que vão de ASIL A (menor severidade) até ASIL D (maior severidade). Os requisitos incluem análise de riscos, definição de métricas de segurança, validação e rastreabilidade de requisitos, e estratégias de teste orientadas a segurança funcional. Casos amplamente documentados na literatura técnica demonstram como falhas de software podem impactar diretamente a segurança veicular, reforçando a importância da conformidade com padrões rigorosos.

2.1.3.3 Setor Médico-Hospitalar

Na indústria médica, dispositivos embarcados tornaram-se essenciais para monitoramento, diagnóstico e tratamento. Equipamentos como bombas de infusão, respiradores, sistemas de radioterapia e equipamentos de imagem utilizam software que deve operar com precisão, confiabilidade e tolerância a falhas.

As normas IEC 62304 e ISO 14971 regulamentam o ciclo de vida do software e a análise de riscos para dispositivos médicos, estabelecendo requisitos de engenharia, verificação e manutenção. Eventos históricos envolvendo falhas de software em máquinas terapêuticas de alta energia ilustram a gravidade potencial de erros em sistemas deste setor e motivaram o desenvolvimento das normas atualmente vigentes.

2.1.3.4 Desafios Comuns: Testes e Certificação

Apesar das diferenças entre os setores, um desafio comum permeia o desenvolvimento de software crítico de segurança: a necessidade de gerar evidências objetivas de segurança por meio de testes e campanhas de ensaios. Testes não apenas demonstram o atendimento aos requisitos definidos em projeto, mas também fornecem a documentação necessária a autoridades certificadoras para comprovar conformidade com normas de segurança. Em ambientes de alta criticidade, a ausência de testes rigorosos compromete não apenas o produto final, mas a viabilidade de sua certificação e entrada em operação.

2.1.4 Importância da Previsibilidade Temporal

A previsibilidade temporal é um dos pilares fundamentais no desenvolvimento de sistemas embarcados de tempo real, especialmente quando esses sistemas operam em contextos críticos à segurança. Em tais aplicações, não basta que o software produza resultados corretos: é igualmente essencial que esses resultados sejam entregues dentro de limites temporais previamente especificados. Assim, além de requisitos funcionais, sistemas de tempo real possuem requisitos temporais que devem ser rigorosamente atendidos (Laplante; Ovaska, 2011).

Em sistemas embarcados determinísticos, cada tarefa deve executar dentro de uma janela de tempo definida durante o projeto, frequentemente denominada *tempo de ciclo* ou *deadline*. O não cumprimento dessas restrições pode resultar em instabilidade, perda de controle, violação de margens de segurança ou até falhas catastróficas, dependendo do contexto operacional. Por essa razão, a análise do *Worst-Case Execution Time* (WCET) é central nesses sistemas, pois fornece limites superiores para a duração de cada tarefa, garantindo que o sistema como um todo permaneça dentro de sua operação segura (Wilhelm *et al.*, 2008).

Do ponto de vista da certificação, normas como DO-178C para sistemas aeronáuticos, ISO 26262 para sistemas automotivos e IEC 62304 para dispositivos médicos enfatizam a necessidade de demonstrar que o comportamento temporal do software é previsível e analisável. Embora tais normas não utilizem explicitamente o termo “previsibilidade temporal”, elas exigem evidências de determinismo, ausência de comportamentos assíncronos não analisados, controle sobre latências internas e conformidade com requisitos temporais definidos. Esses requisitos mostram que a previsibilidade temporal é, de fato, um elemento estruturante do processo de verificação e validação em sistemas críticos.

Garantir previsibilidade temporal implica o monitoramento de métricas como latência, jitter, tempo de resposta e ocupação do ciclo de execução. Variações inesperadas nesses parâmetros podem indicar problemas de implementação, interferências entre tarefas, condições de corrida, priorização inadequada ou fluxo de execução inefficiente. Dessa forma, ferramentas capazes de observar, registrar e analisar esses fenômenos tornam-se essenciais durante o processo de teste e integração.

Nesse contexto, a ferramenta desenvolvida neste trabalho apoia diretamente a análise de previsibilidade temporal, permitindo a inspeção em tempo real de variáveis internas, estados intermediários e comportamento do ciclo de execução. Ao oferecer visibilidade sobre características temporais do software — sem interferir significativamente em sua operação — a ferramenta contribui para a identificação precoce de problemas, para a avaliação de requisitos temporais e para a produção de evidências de conformidade exigidas em processos de certificação.

2.2 Testes em Sistemas Embarcados

Podem-se dividir os testes de software em sistemas embarcados em dois grandes grupos: testes de requisitos de alto nível e testes de requisitos de baixo nível. A execução adequada desses dois grupos é essencial para comprovar a aderência a normas de certificação, como a DO-178C. Embora os testes representem apenas uma parte do esforço total de verificação exigido para a certificação de software crítico, eles constituem uma etapa central e complexa — sobretudo os testes de requisitos de alto nível, os quais derivam dos requisitos de sistema e devem demonstrar a capacidade do software de se integrar corretamente ao sistema do qual fará parte.

O objetivo dos testes de software é identificar erros introduzidos durante a implementação dos requisitos na linguagem de programação de alto nível, incluindo erros lógicos, de uso da linguagem e funcionais. Quanto maior o rigor na definição dos casos de teste, maior será a cobertura obtida e, consequentemente, maior a probabilidade de detectar falhas durante o desenvolvimento.

Os testes são organizados em três fases principais:

- **Testes de Desenvolvimento** — Nesta fase, o programador traduz os requisitos de alto e baixo nível para a linguagem de programação adotada. Os testes são executados em plataformas de desenvolvimento, que geralmente possuem baixa representatividade em relação ao sistema final.
- **Testes de Requisitos de Software** — Nesta etapa, os testes são elaborados com base nos requisitos utilizados na codificação. Para cada requisito são definidos casos de teste específicos, podendo haver mais de um caso por requisito.
- **Testes de Integração** — Os testes de integração são realizados em ambientes representativos do sistema final, abrangendo principalmente os requisitos de alto nível e os requisitos de sistema. Nessa fase, não é permitido o uso de instrumentos ou técnicas que possibilitem a coleta de informações internas do software de forma intrusiva, de modo a não interferir na execução normal do sistema.

O presente trabalho tem como objetivo auxiliar as etapas de testes de requisitos e, especialmente, de testes de integração, nas quais o uso de kits de desenvolvimento deixa de ser adequado e o ambiente de testes deve apresentar maior representatividade em relação ao sistema final. Em sistemas aeroespaciais, é comum que a fase final de testes ocorra em um ambiente denominado Iron Bird, que consiste em um laboratório contendo todos os sistemas reais da aeronave conectados e configurados para ensaios de integração. Nesse ambiente, não é possível empregar técnicas de depuração que interrompam a execução do

software para inspeção de seu estado interno. Assim, ferramentas de *peek* e *poke* tornam-se particularmente valiosas.

2.3 Protocolos de Comunicação para Depuração

Diversos meios físicos e protocolos de comunicação são tradicionalmente empregados para a troca de dados entre sistemas embarcados e ferramentas externas de teste e depuração. Entre os mais utilizados destacam-se UART, CAN e Ethernet, amplamente documentados na literatura técnica e em normas industriais (Tanenbaum; Wetherall, 2011; Bosch, 1991; Institute of Electrical and Electronics Engineers, 2018).

Cada um desses protocolos apresenta vantagens e limitações específicas, e sua escolha está fortemente associada à arquitetura do sistema no qual o software será executado. Trata-se, portanto, de uma decisão sistêmica e abrangente, frequentemente condicionada por restrições físicas impostas no nível de hardware, sejam elas derivadas de requisitos de custo, requisitos funcionais ou ainda de decisões comerciais associadas à escolha de fornecedores. Uma característica comum aos protocolos considerados neste contexto é a adoção do paradigma de comunicação serial.

- **UART** — A comunicação via *Universal Asynchronous Receiver/Transmitter* (UART) caracteriza-se pela simplicidade de implementação, baixo custo e ampla disponibilidade em microcontroladores comerciais. Contudo, apresenta taxas de transmissão relativamente limitadas, ausência de mecanismos nativos de detecção e correção de erros e menor robustez em ambientes sujeitos a interferências eletromagnéticas (Tanenbaum; Wetherall, 2011; Stallings, 2017).
- **CAN** — O barramento *Controller Area Network* (CAN) foi projetado para ambientes industriais e automotivos, oferecendo elevada imunidade a ruído, mecanismos de arbitragem determinística e priorização de mensagens. Essas características o tornam adequado a sistemas distribuídos de tempo real; entretanto, sua capacidade de carga útil por quadro é limitada, o que restringe sua aplicação em cenários que demandam maior volume de dados (Bosch, 1991; ISO, 2015).
- **Ethernet** — O protocolo Ethernet disponibiliza altas taxas de transmissão e baixa latência, sendo amplamente utilizado em sistemas embarcados modernos. Todavia, seu uso em sistemas de tempo real ou críticos requer camadas adicionais ou extensões específicas — como protocolos determinísticos ou perfis de tempo real — a fim de garantir previsibilidade temporal, o que pode introduzir sobrecarga e maior complexidade de implementação (Institute of Electrical and Electronics Engineers, 2018; Kopetz, 2011).

Independentemente do meio físico adotado, protocolos de depuração geralmente seguem um modelo de comunicação do tipo *request* e *response*, no qual uma ferramenta externa envia comandos e o dispositivo embarcado responde de forma determinística. Esse paradigma facilita a rastreabilidade das interações e permite que comandos sejam estruturados de forma padronizada, com semântica clara e verificável. Para garantir a correta interpretação das mensagens, é fundamental definir mecanismos de framing, incluindo marcação de início e fim de pacote, numeração de sequência e regras explícitas de encapsulamento dos dados.

A confiabilidade da comunicação depende também da detecção de erros. Por esse motivo, técnicas de checagem de integridade, como *Checksums* ou *Cyclic Redundancy Checks* (CRC), são amplamente utilizadas. O CRC destaca-se pela capacidade de detectar padrões de erro específicos, sendo adequado para ambientes sujeitos a interferências eletromagnéticas, comuns em sistemas embarcados (Koopman; Chakravarty, 2004). A adoção de CRCs reduz a probabilidade de aceitação de quadros corrompidos e contribui diretamente para a robustez geral do protocolo.

Além das questões relacionadas à integridade dos dados, protocolos de depuração devem lidar com desafios práticos, tais como ruído no canal de comunicação, perda de pacotes, duplicação de mensagens e troca na ordem de chegada. Esses fenômenos podem resultar de limitações físicas, contenção no barramento, falhas temporárias ou restrições de temporização. Por esse motivo, estratégias como retransmissão de quadros, temporização com timeouts, confirmação positiva (ACK) e numeração sequencial são frequentemente adotadas para garantir a operação correta do sistema.

A discussão destes elementos constitui a base técnica necessária para o desenvolvimento de um protocolo personalizado de comunicação para depuração, adaptado às restrições e requisitos de sistemas embarcados críticos. A compreensão das características, limitações e mecanismos de confiabilidade dos protocolos existentes permite projetar soluções mais robustas, adequadas ao contexto aeroespacial e alinhadas às práticas recomendadas de verificação e certificação.

2.4 Operações Peek, Poke e Telemetria

A operação *peek* consiste na leitura remota de posições de memória, registradores ou variáveis internas do software embarcado, proporcionando visibilidade do estado interno sem interromper a execução. Essa capacidade é fundamental em sistemas de tempo real, nos quais interrupções artificiais podem distorcer o comportamento do software ou mascarar defeitos críticos (Zuberi; Goddyn; Ghalwash, 1999). Já a operação *poke* permite a escrita remota de valores nesses mesmos elementos, possibilitando injeção de estados, alteração temporária de parâmetros e verificação de respostas do sistema frente a condições específicas. Ambas as operações se tornaram parte essencial de arquiteturas de depuração

on-chip, como JTAG, Nexus e ARM CoreSight, que utilizam conceitos semelhantes para acesso estruturado e controlado ao hardware e ao software durante a execução (Hopkins; McDonald-Maier, 2006).

Complementarmente, a telemetria desempenha um papel distinto e igualmente importante. Diferentemente das operações on-demand de *peek* e *poke*, a telemetria consiste no envio assíncrono, periódico ou orientado a eventos de dados internos do software para um sistema externo de monitoramento. Essa estratégia é amplamente empregada em sistemas críticos e de tempo real, pois permite observar o comportamento do sistema ao longo do tempo, sem interferência perceptível no fluxo de execução. A telemetria é essencial para detectar anomalias, verificar tendências e avaliar o cumprimento de requisitos temporais — aspectos particularmente relevantes em sistemas críticos conforme descrito em normas como DO-178C (RTCA, 2011).

Apesar de sua utilidade, operações remotas de inspeção e escrita introduzem riscos inerentes. O acesso arbitrário à memória pode comprometer a integridade dos dados, violar funções de segurança ou alterar fluxos de controle essenciais ao comportamento correto do software. Esses riscos já são amplamente documentados em literatura de depuração embarcada, que destaca a necessidade de mecanismos de proteção, autenticação e delimitação rigorosa das regiões de memória acessíveis (Schneider; Fraleigh, 2004). Em sistemas críticos, esses controles são ainda mais importantes, dado que falhas induzidas por operações de debug podem gerar violações de requisitos funcionais e de segurança (Rierson, 2013). Assim, operações de *peek* e *poke* devem ser implementadas de forma atômica, com validação rigorosa e sem possibilidade de interferir em seções de código sensíveis ou temporizações essenciais.

Outro aspecto relevante é que operações contínuas de *peek* e *poke* — isto é, a tentativa de monitorar uma variável por meio de leituras repetitivas — não são adequadas para sistemas em tempo real. Além de aumentar a carga no canal de comunicação, leituras frequentes podem introduzir atrasos ou sobrecarga no firmware, comprometendo o determinismo do sistema. Por esse motivo, ferramentas adequadas à instrumentação de sistemas críticos empregam telemetria estruturada e coleta orientada a eventos, em vez de leituras incessantes (Christof, 2013).

No contexto das ferramentas contemporâneas, diversas arquiteturas implementam mecanismos equivalentes aos descritos. O PX4, por exemplo, utiliza o uORB como middleware de mensagens, disponibilizando mecanismos de depuração baseados em tópicos internos, embora não forneça acesso direto à memória como um *peek* e *poke* tradicional. Microcontroladores STM32 oferecem depuração via GDB Server incorporado, permitindo inspeção de registradores e variáveis durante pausa do processador, mas sem capacidade integrada de depuração não intrusiva em execução contínua. Já o Microchip Data Visualizer opera por meio de canais de dados instrumentados, fornecendo telemetria estruturada e

escrita remota de parâmetros, aproximando-se de um sistema híbrido entre telemetria e controle.

2.5 Ambientes de Testes

Do ponto de vista metodológico, diferentes abordagens de teste são empregadas conforme o grau de conhecimento interno do sistema. Testes caixa-preta avaliam apenas entradas e saídas, sem acesso ao código ou à lógica interna, sendo úteis para validação funcional e verificação de requisitos. Já os testes caixa-branca exploram a estrutura interna do software, permitindo avaliar caminhos de execução, condições e variáveis internas — abordagem essencial para conformidade com normas como DO-178C e ISO 26262, que exigem cobertura de código em diferentes níveis de certificação. Em sistemas embarcados, a combinação de ambas as abordagens é comum, dado que testes funcionais isolados não são suficientes para capturar anomalias relacionadas a concorrência, temporização ou integração com o hardware.

No contexto de sistemas embarcados modernos, técnicas avançadas como uso de ambientes PIL (Processor-in-the-Loop), SIL (Software-in-the-Loop), HIL (Hardware-in-the-Loop) e *Iron Bird* (este último mais comumente encontrado na indústria Aeroespacial) ampliam significativamente a capacidade de validação. Testes PIL executam o código no processador real ou simulado, assegurando correção funcional e temporal da implementação final. Testes SIL simulam outros sistemas aos quais o software terá que interagir em ambiente nativo ou emulado, permitindo validações rápidas antes da integração com hardware real. Finalmente, testes HIL permitem avaliar o sistema embarcado interagindo com modelos de planta, sensores simulados e atuadores virtuais, sendo amplamente utilizados na indústria aeronáutica, automotiva e de robótica para validar cenários complexos e dinâmicos com alta fidelidade. Testes em Iron Bird são a última etapa de validação da integração entre sistemas antes de se realizar ensaios em voo (novamente tomando-se a indústria aeroespacial como referência) ensaios no Iron Bird são comuns para testes de release de software antes de serem levados ao veículo final, nestes testes é possível capturar as situações reais de interação entre os sistemas, onde sensores e atuadores reais aumentam a confiabilidade dos testes.

Portanto é comum que os ambientes sejam utilizados sequencialmente no desenvolvimento do sistema embarcado, de forma que a cada etapa aumenta-se a representatividade final do ambiente de testes, sendo assim, é possível mitigar possíveis erros em etapas de custo mais baixo até que se tenha uma confiança maior no software, a uma ordem natural ao se fazer uso destes ambientes de testes é descrita abaixo:

$$[\text{PIL}] \rightarrow [\text{SIL}] \rightarrow [\text{HIL}] \rightarrow [\text{Iron Bird}]$$

A ferramenta de depuração proposta neste trabalho tem como finalidade ser

empregada em todas as etapas do processo descrito anteriormente, possibilitando a execução de testes baseados no protocolo e permitindo sua portabilidade entre diferentes ambientes de verificação. Essa característica favorece a reutilização sistemática dos testes, contribuindo diretamente para o aumento da confiabilidade do sistema final.

A automação de testes em sistemas embarcados constitui um elemento fundamental para assegurar confiabilidade, repetibilidade e rastreabilidade ao longo do ciclo de desenvolvimento. No contexto do protocolo aqui proposto, tais benefícios são potencializados, uma vez que o conjunto de comandos determinísticos admitido pela ferramenta viabiliza a criação de **scripts** de teste estruturados. Esses **scripts** permitem a execução procedural e reproduzível dos ensaios nos diversos ambientes mencionados, favorecendo padronização, auditoria e integração com fluxos modernos de verificação.

2.6 Trabalhos Relacionados

Diversas ferramentas e metodologias têm sido propostas para apoiar o processo de teste, depuração e verificação de software embarcado, especialmente em sistemas críticos, nos quais a previsibilidade temporal, a rastreabilidade e a não intrusividade são requisitos fundamentais. Entre as soluções existentes destacam-se depuradores JTAG/SWD, registradores de eventos (*trace loggers*), analisadores lógicos, barramentos proprietários de depuração e plataformas de teste baseadas em hardware de bancada. Embora amplamente utilizadas nas fases iniciais de desenvolvimento, essas ferramentas apresentam limitações significativas quando aplicadas em ambientes representativos ou integrados, como observado por Rierson (Rierson, 2013).

Uma limitação recorrente envolve o caráter intrusivo das técnicas tradicionais de depuração. Métodos como *breakpoints*, *single stepping* e instrumentação direta alteram o comportamento temporal e o fluxo de execução do software, inviabilizando sua aplicação em contextos de certificação e operação real. Essa restrição é particularmente relevante em sistemas embarcados de missão crítica, nos quais a temporização deve ser preservada integralmente (NASA, 2008). Além disso, muitas dessas ferramentas dependem de acesso físico ao microcontrolador ou a interfaces disponíveis apenas em protótipos, tornando seu uso inviável em plataformas integradas de grande complexidade, como aeronaves e sistemas aeroespaciais.

Outras abordagens incluem mecanismos proprietários de telemetria, depuração via CAN ou UART e serviços internos de diagnóstico. Contudo, tais soluções geralmente carecem de padronização, são pouco extensíveis e nem sempre são projetadas com foco em segurança, integridade dos dados ou tolerância a falhas. A ausência de protocolos formais de integridade, autenticação e controle de sessão limita sua utilização em ambientes críticos, onde requisitos rigorosos são estabelecidos por normas como a DO-178C (RTCA, 2011).

Diante dessas limitações, observa-se um *gap* técnico relevante: a falta de uma ferramenta capaz de prover depuração não intrusiva, execução remota de comandos, coleta estruturada de dados e operação em ambientes representativos do sistema final, preservando previsibilidade temporal e confiabilidade operacional. A literatura e padrões industriais recomendam métodos de verificação independentes, rastreáveis e com impacto mínimo no comportamento do software, reforçando essa lacuna (Electrical; Engineers, 2000).

A solução proposta neste trabalho busca preencher esse espaço ao fornecer uma ferramenta modular, orientada a comandos remotos, baseada em protocolo estruturado e voltada ao apoio de testes e integração em sistemas críticos. Ao posicionar-se entre depuradores tradicionais e ferramentas proprietárias de telemetria, a abordagem apresentada contribui para o estado da arte ao combinar não intrusividade, padronização e extensibilidade, alinhando-se às melhores práticas recomendadas por normas consolidadas da engenharia de software crítico.

3 DESENVOLVIMENTO DO PROTOCOLO

Este capítulo descreve o processo de concepção, definição e implementação do protocolo de comunicação *peek* e *poke*, desenvolvido como solução simplificada para depuração e interação com sistemas embarcados. O objetivo é oferecer uma abordagem padronizada e de baixo custo, permitindo leitura e escrita em variáveis de memória do sistema embarcado de forma controlada e portátil.

3.1 Arquitetura do Sistema

A arquitetura proposta é composta por dois elementos principais:

- **Software embarcado (protocolo C/DESTRA):** responsável por interpretar comandos recebidos via interface serial e executar operações de leitura (*peek*) ou escrita (*poke*) em endereços de memória previamente mapeados.
- **Ferramenta host (cliente Python/DESTRA UI):** responsável por enviar comandos, exibir resultados ao usuário e automatizar sequências de acesso.

A comunicação é estabelecida por meio de uma porta serial (UART), amplamente disponível em plataformas de prototipagem, o que garante a portabilidade da solução. O diagrama simplificado da arquitetura pode ser representado como:

[Usuário] ↔ [Cliente Python] ^{UART} ↔ [Handler Peek-Poke] ↔ [Memória MCU]

Para que os comandos sejam transmitidos entre um host, aplicativo manipulado por um usuário, e o sistema embarcado, foi especificado um protocolo que estabelece comandos de *peek* e *poke*. Estes comandos são codificados de acordo com as regras estabelecidas pelo protocolo e transmitidos via interface serial para o sistema embarcado, que por sua vez faz uso de uma máquina de estados para decodificar o protocolo e processar os comandos.

3.2 Especificação do Protocolo

O protocolo foi projetado para ser mínimo e determinístico, a fim de reduzir o *overhead* de comunicação e facilitar implementações em ambientes de recursos restritos.

As duas operações básicas (comandos) são:

- **PEEK:** leitura de N bytes a partir de um endereço definido.
- **POKE:** escrita de N bytes em um endereço definido.

Cada pacote é composto pela estrutura descrita na Tabela 2.

Campo	Tamanho (bytes)	Descrição
Palavra Mágica	2	Conjunto de bytes usados para identificar um comando
Comando	1	Identificação do tipo de operação
Endereço	2-4	Posição de memória da variável
Tamanho	1	Tamanho (em bytes) do tipo da variável
Valor	N	Dados (no caso de comando poke)

Tabela 2 – Estrutura do pacote do protocolo *peek* e *poke*

Esse formato reduz a ambiguidade e facilita a interpretação tanto no sistema embarcado quanto na ferramenta cliente. O protocolo funciona em um sistema de Requisição e Resposta (Request/Response), onde a ferramenta host realiza requisições para o sistema embarcado, e este responde às requisições de acordo com o comando contido na requisição. As respostas são um echo do cabeçalho inicial da requisição (Palavra Mágica, Comando) adicionado um byte de Status (sucesso ou falha). No caso do comando peek, o valor da variável em memória é retornado. No caso do comando poke, um echo do valor recebido é retornado para certificar que os dados recebidos pelo sistema embarcado estão de acordo. Desta forma, a ferramenta host consegue identificar que seu comando foi aceito e processado pelo sistema embarcado.

3.2.1 Definições de Campos

O protocolo utiliza algumas definições padrão para poder identificar os bytes e a estrutura dos comandos recebidos:

Palavra Mágica: Os 2 bytes são definidos como uma constante com dois tokens em hexadecimal que em conjunto recriam a palavra “Café”: 0xCA, 0xFE. O sistema embarcado espera esta sequência e ao recebê-la avança a máquina de estados para a espera do Comando.

Comando: Definido em apenas um byte: 0xF1 para o comando de peek e 0xF2 para o comando de poke.

Endereço: Dividido em dois bytes para a transmissão de valores multi-byte, mantendo compatibilidade nativa com a arquitetura AVR do Arduino UNO (little-endian). Esta escolha elimina overhead de conversão no sistema embarcado, otimizando o desempenho.

Tamanho: 1 byte para descrever o tamanho do tipo da variável contida no endereço especificado pelo comando (1, 2, 4 ou 8 bytes).

Valor: Este campo só é necessário no comando de poke, pois é nele que é transmitido o valor a ser sobreescrito no endereço especificado no comando.

3.2.2 Exemplos de Pacotes

Exemplo de pacote para comando PEEK:

Considerando uma variável com endereço 0x0104, tipo `int16` e tamanho 2 bytes:

CA FE F1 04 01 02

Decomposição:

- CA FE: Palavras mágicas
- F1: Comando PEEK
- 04: Byte baixo do endereço (LSB)
- 01: Byte alto do endereço (MSB)
- 02: Tamanho (2 bytes)

Nota: o endereço 0x0104 é transmitido como 04 01 (little-endian).

Exemplo de pacote para comando POKE:

Considerando a mesma variável com endereço 0x0104, tipo `int16`, tamanho 2 bytes e escrevendo o valor 4 (0x0004):

CA FE F2 04 01 02 04 00

Decomposição:

- CA FE: Palavras mágicas
- F2: Comando POKE
- 04: Byte baixo do endereço (LSB)
- 01: Byte alto do endereço (MSB)
- 02: Tamanho (2 bytes)
- 04 00: Valor 0x0004 em little-endian

Com estas definições, temos um protocolo simples, porém eficaz, para o processamento de requisições e envio de respostas.

3.3 Implementação no Microcontrolador

No software embarcado, o protocolo foi implementado com uma função que é executada no laço principal do microcontrolador (Arduino Uno, na versão de referência). O fluxo básico segue as seguintes etapas:

1. Aguardar bytes recebidos pela interface Serial/UART.
2. Interpretar a sequência de bytes recebidas de forma a detectar o início do cabeçalho e determinar se trata-se de uma operação *peek* ou *poke*.
3. Executar a leitura no caso de *peek* no endereço de memória recebido, ou escrita, no caso de *poke*, do valor recebido no endereço especificado.
4. Retornar uma resposta estruturada ao cliente, confirmando o resultado da operação.

A implementação foi concebida de forma modular, permitindo que novos comandos sejam adicionados sem impacto significativo no desempenho. Uma máquina de estados realiza a interpretação dos dados recebidos pela porta serial. O mecanismo de leitura e escrita da serial não faz parte do escopo deste trabalho, sendo utilizado o mecanismo já disponibilizado pelo sistema embarcado (Arduino UNO).

A máquina de estados criada é capaz de processar tanto os comandos de *peek* quanto os de *poke*. O processamento é feito por meio da tokenização dos bytes recebidos pela porta serial. Cada transição é um ponto de verificação para o próximo byte a ser recebido. A máquina de estados fica em um modo de espera, comparando os bytes recebidos com o que está definido para a transição ao próximo estado. Em caso positivo (o byte esperado corresponde ao byte codificado para a transição), a máquina avança ao próximo estado e assim sucessivamente. Ao final do processamento de um cabeçalho completo, faz-se a chamada para o processamento do comando, e a máquina de estados volta à condição inicial de espera.

Por meio da Figura 1, é possível observar o comportamento determinístico da máquina de estados, onde cada estado aguarda um byte específico para avançar à etapa subsequente.

3.3.1 Pseudocódigo da Máquina de Estados

O mecanismo é implementado dentro de uma rotina chamada `destraHandler` em um arquivo a ser incluído no projeto Arduino (`destra_protocol.ino`), cuja implementação está apresentada no Apêndice A. Esta função é um loop que tem como regra de parada a entrada de dados da Serial e o estado atual da máquina de estados sendo diferente de `PROCESS_REQUEST`. A cada byte recebido através da interface serial o loop avalia o

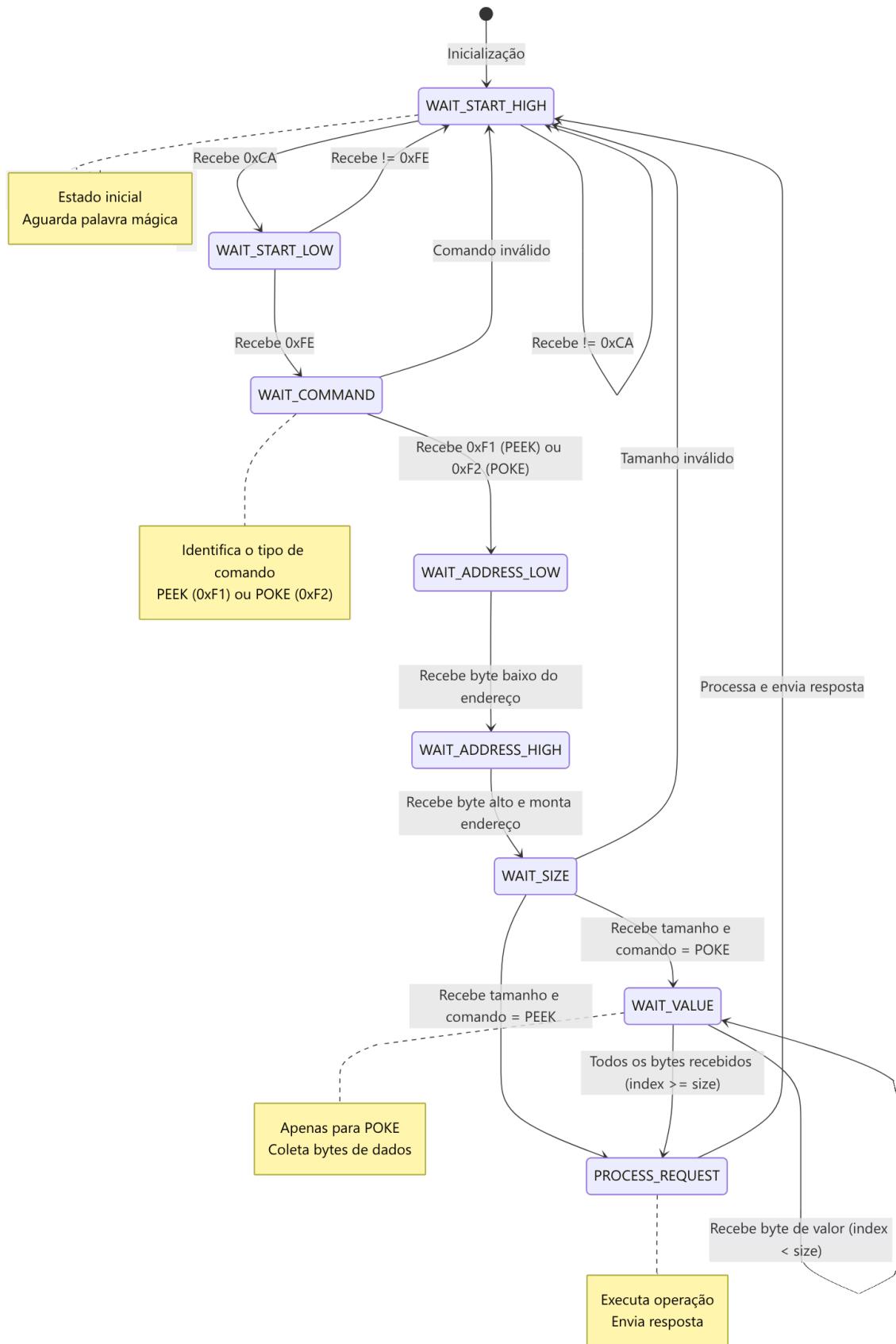


Figura 1 – Diagrama de estados para a implementação do protocolo DESTRA

estado atual compara com a regra da máquina de estados e avança em caso afirmativo para o próximo estado, ao final o processamento do comando é realizado, a Figura 1 e a Listagem 3.1 apresentam este algoritmo em detalhe.

Listagem 3.1 – Pseudocódigo da máquina de estados do protocolo

```
Funcao destraHandler():
    Enquanto houver bytes disponiveis na Serial
        e destraState != PROCESS_REQUEST:
            Ler proximo byte -> inByte
            Se destraState == WAIT_START_HIGH:
                Se inByte == 0xCA:
                    destraState = WAIT_START_LOW
                Senao:
                    destraState = WAIT_START_HIGH
            Senao se destraState == WAIT_START_LOW:
                Se inByte == 0xFE:
                    destraState = WAIT_COMMAND
                Senao:
                    destraState = WAIT_START_HIGH
            Senao se destraState == WAIT_COMMAND:
                destraCommand = inByte
                Se destraCommand == CMD_PEEK ou
                    destraCommand == CMD_POKE:
                        destraState = WAIT_ADDRESS_LOW
                Senao:
                    destraState = WAIT_START_HIGH
            Senao se destraState == WAIT_ADDRESS_LOW:
                addressLow = inByte
                destraState = WAIT_ADDRESS_HIGH
            Senao se destraState == WAIT_ADDRESS_HIGH:
                addressHigh = inByte
                Combinar addressLow e addressHigh ->
                    destraAddress (little-endian)
                destraState = WAIT_SIZE
            Senao se destraState == WAIT_SIZE:
                destraSize = inByte
                Se destraCommand == CMD_PEEK:
                    destraState = PROCESS_REQUEST
                Senao se destraCommand == CMD_POKE:
                    destraValueIndex = 0
```

```

        destraState = WAIT_VALUE
Senao:
        destraState = WAIT_START_HIGH
Senao se destraState == WAIT_VALUE:
    Se destraValueIndex < 8 e
        destraValueIndex < destraSize:
            destraValueBuffer[destraValueIndex]
                = inByte
            destraValueIndex++
    Se destraValueIndex >= destraSize:
        destraState = PROCESS_REQUEST

```

3.3.2 Variáveis da Máquina de Estados

A Tabela 3 apresenta as variáveis utilizadas na implementação da máquina de estados:

Variável	Descrição
destraState	Estado atual. Inicializada com WAIT_START_HIGH
destraCommand	Comando recebido (CMD_PEEK ou CMD_POKE)
addressLow	Byte menos significativo (LSB) do endereço
addressHigh	Byte mais significativo (MSB) do endereço
destraAddress	Endereço de 16 bits combinado (little-endian)
destraSize	Tamanho da operação em bytes (1 a 8)
destraValueBuffer [8]	Buffer temporário para armazenar bytes de valor
destraValueIndex	Índice de controle do buffer de valores

Tabela 3 – Variáveis da máquina de estados

3.3.3 Processamento do Comando PEEK

O processamento do comando de *peek* é feito em uma rotina chamada `process.PeekRequest`. Esta rotina, assim que chamada, faz a gravação de volta na serial do echo do cabeçalho recebido, realiza uma checagem de validação para o endereço de memória e o tamanho recebidos, em seguida grava o status da operação. Em caso de sucesso, grava o valor do endereço requisitado na serial.

Listagem 3.2 – Pseudocódigo do processamento de PEEK

Função `process.PeekRequest ()`:

```

// Enviar cabeçalho da resposta
Enviar 0xCA via serial
Enviar 0xFE via serial
Enviar CMD_PEEK via serial

```

```
// Validar faixa de endereço
Se destraAddress < 0x0100 ou
    destraAddress > 0x08FF:
    Enviar STATUS_ADDRESS_RANGE_ERROR via serial
    Retornar

// Validar tamanho da operação
Se destraSize <= 0 ou destraSize > 8:
    Enviar STATUS_SIZE_ERROR via serial
    Retornar

// Enviar status de sucesso
Enviar STATUS_SUCCESS via serial

// Ler dados da memória e enviar
Para i de 0 ate destraSize - 1:
    Ler byte da memória em
        destraAddress + i -> valor
    Enviar valor via serial
```

A Figura 2 apresenta o diagrama de sequência das operações envolvidas na execução de um comando *peek*, desde o envio da requisição pela ferramenta host até o retorno da resposta pelo sistema embarcado, detalhando as etapas de processamento intermediárias.

3.3.4 Processamento do Comando POKE

O processamento do comando *poke* é realizado em uma rotina chamada `processPokeRequest`. Nesta rotina, temos um processamento do cabeçalho, validação e status de resposta análogo ao processamento do *peek*. Em seguida, a rotina faz a sobreescrita dos bytes de valor recebidos no endereço de memória especificado pelo comando recebido.

Listagem 3.3 – Pseudocódigo do processamento de POKE

Função `processPokeRequest ()`:

```
// Enviar cabeçalho da resposta
Enviar 0xCA via serial
Enviar 0xFE via serial
Enviar CMD_POKE via serial

// Validar faixa de endereço
Se destraAddress < 0x0100 ou
```

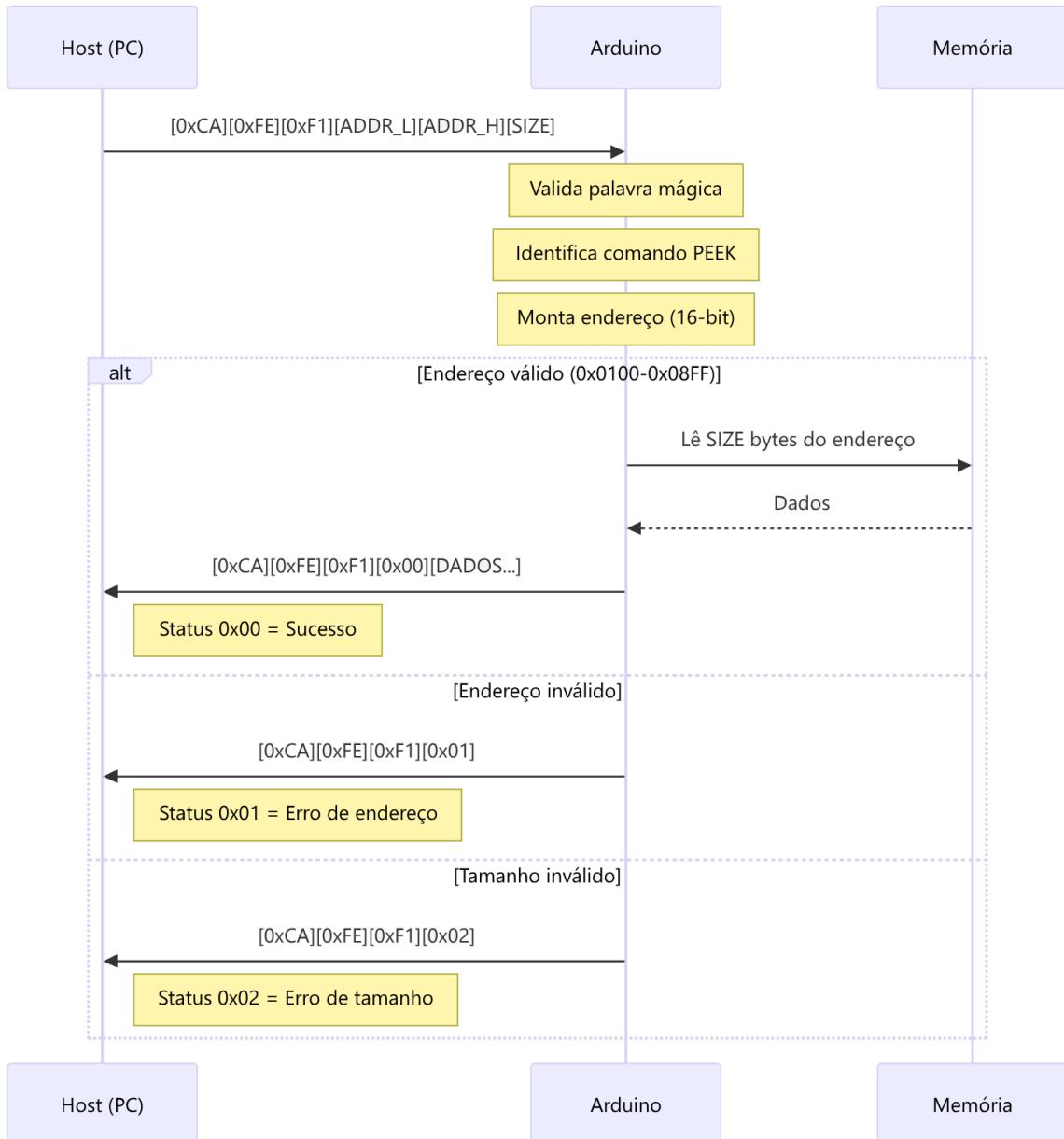


Figura 2 – Diagrama de sequencia de um comando peek a partir da ferramenta host

```
destraAddress > 0x08FF:  
    Enviar STATUS_ADDRESS_RANGE_ERROR via serial  
    Retornar  
  
// Validar tamanho da operacao  
Se destraSize <= 0 ou destraSize > 8:  
    Enviar STATUS_SIZE_ERROR via serial  
    Retornar  
  
// Escrever dados na memoria  
Para i de 0 ate destraSize - 1:  
    Escrever destraValueBuffer[ i ]  
        em memoria no endereco destraAddress + i  
  
// Enviar status de sucesso  
Enviar STATUS_SUCCESS via serial  
  
// Ecoar de volta os dados escritos  
Para i de 0 ate destraSize - 1:  
    Ler byte da memoria em  
        destraAddress + i -> valor  
    Enviar valor via serial
```

A Figura 3 apresenta o diagrama de sequência das operações envolvidas na execução de um comando poke, desde o envio da requisição pela ferramenta host até o retorno da resposta pelo sistema embarcado, detalhando as etapas de validação, escrita em memória e confirmação.

3.4 Implementação no Cliente Host

A ferramenta host foi desenvolvida em Python, aproveitando bibliotecas de suporte a comunicação serial (como *pyserial*). Para simplificar a interação, foi criada uma interface gráfica denominada DESTRA UI, que oferece:

- Conexão automática à porta serial.
- Carregamento de símbolos a partir de arquivos ELF/DWARF, permitindo ao usuário trabalhar com nomes de variáveis em vez de endereços.
- Seleção interativa de variáveis para operações de leitura e escrita.
- Seleção de variáveis para o comando peek.

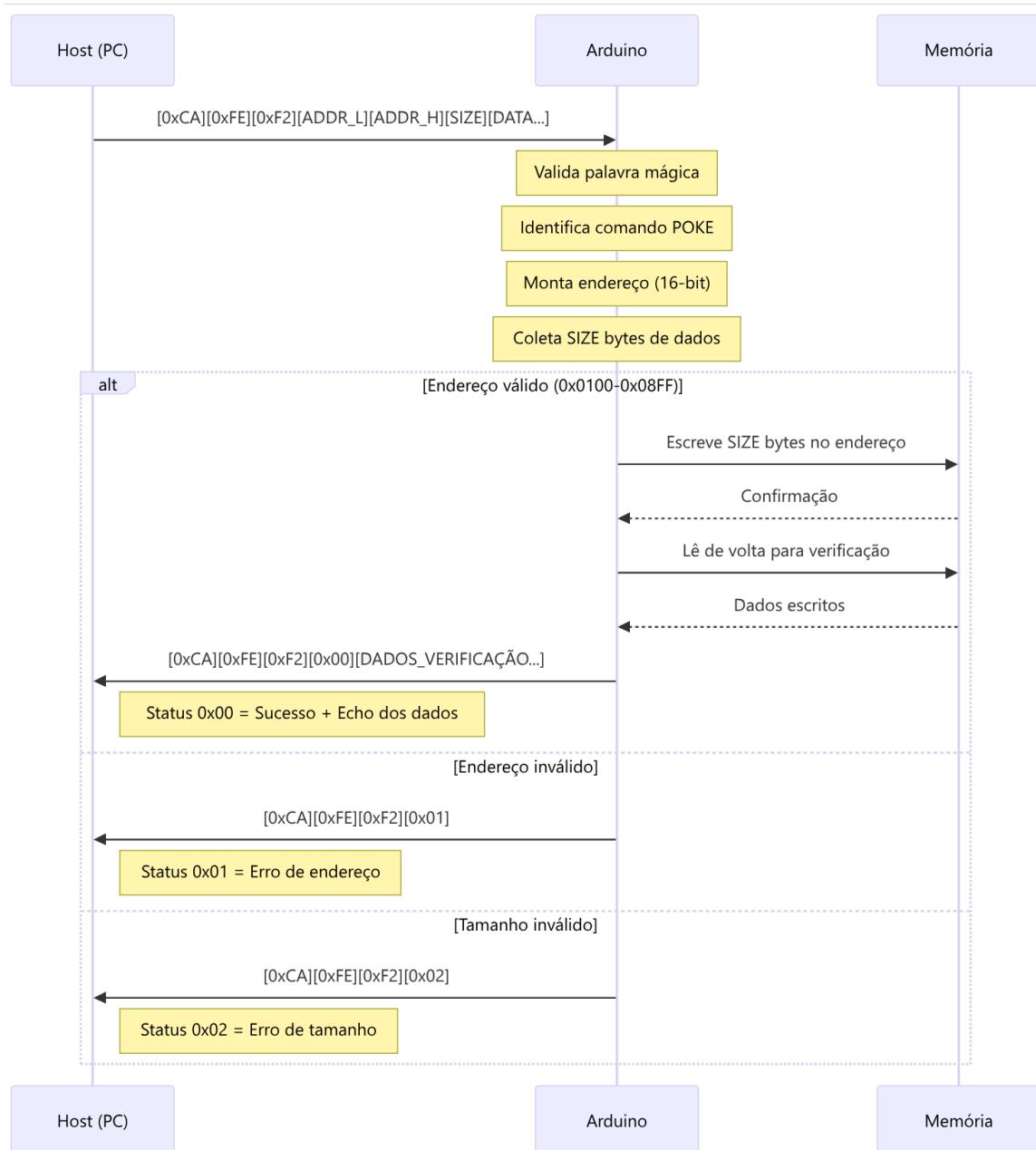


Figura 3 – Diagrama de sequencia de um comando poke a partir da ferramenta host

- Seleção de variáveis e valor para o comando *poke*.
- Um mecanismo simples de *continuous peek* (leitura automática a cada ciclo do valor das variáveis selecionadas). Onde é possível escolher a frequencia de envio de comandos *peek* para o microcontrolador.
- Histórico de comandos e registro de logs para análise.

Essa abordagem facilita o uso por profissionais que não possuem familiaridade com ferramentas de baixo nível, ao mesmo tempo em que garante flexibilidade para desenvolvedores avançados.

A Figura 4 fornece uma representação esquemática das interações temporais entre os componentes do sistema durante a execução de uma operação *peek*. O diagrama de sequência permite uma compreensão clara e precisa do protocolo de comunicação implementado, evidenciando: (a) o ponto de iniciação no módulo de interface do usuário, (b) o processamento de decodificação e transmissão, (c) o tratamento e execução no microcontrolador, e (d) a retransmissão e decodificação dos dados na ferramenta host. Esta estrutura hierarquizada garante a rastreabilidade completa da operação e facilita a validação e verificação do protocolo.

Este diagrama é fundamental para compreender a arquitetura em camadas do sistema, evidenciando como a separação de responsabilidades entre componentes garante uma solução modular, testável e facilmente extensível.

3.4.1 Arquitetura da Ferramenta Host

A implementação da ferramenta é feita em três scripts Python:

destra.py: Contém a implementação do protocolo proposto neste trabalho, análogo ao que foi apresentado anteriormente. A listagem deste script é apresentada no Apêndice A.

destra.ui.py: Implementa a interface gráfica da aplicação juntamente com suas ações. A listagem deste script é apresentada no Apêndice A.

data_dictionary.py: Implementa um parser do formato ELF/DWARF que cria um dicionário de dados relacionando os nomes de variáveis declarados no código fonte embarcado com seus atributos: endereço em memória, tipo de dado, tamanho. A listagem deste script é apresentada no Apêndice A.

3.4.2 Procedimento Operacional

Para realizar a operação de um comando *peek*, são necessários os seguintes passos:

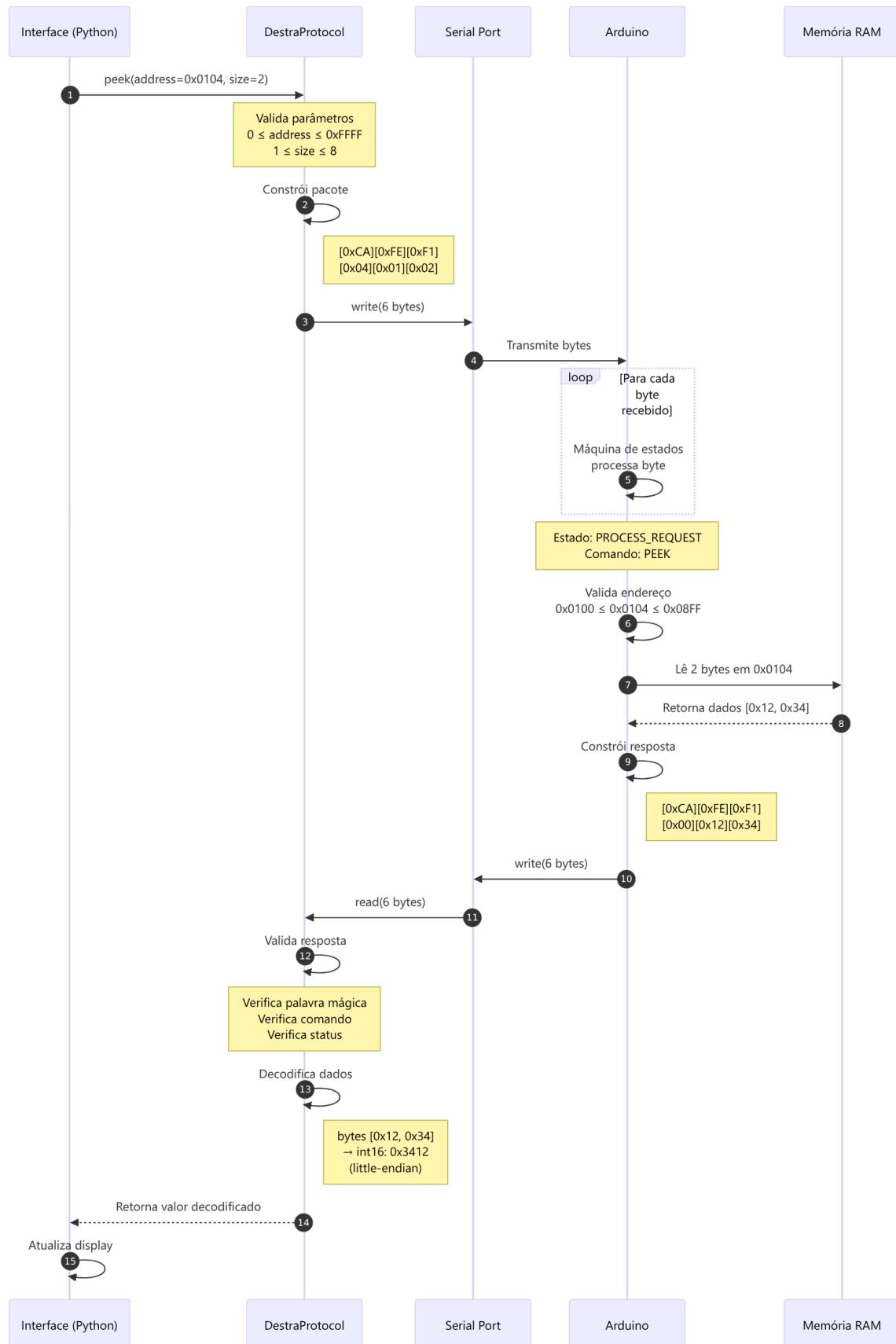


Figura 4 – Diagrama de sequência da operação `peek`. O fluxo ilustra as interações temporais entre a interface DESTRA UI, o módulo de protocolo Python, a comunicação serial UART, e a máquina de estados do sistema embarcado, demonstrando o ciclo completo request/response desde a requisição do usuário até a exibição dos dados recuperados da memória.

1. Conectar fisicamente o host (PC) onde a ferramenta é executada ao Arduino UNO via cabo USB.
2. Compilar para o Arduino UNO o código ao qual se deseja instrumentar, contendo a implementação embarcada do protocolo desta.
3. Utilizando a IDE do Arduino, exporte o arquivo ELF/DWARF.
4. Em sequência, abrir a ferramenta e selecionar a porta de comunicação correta (a ferramenta implementa um script interno que detecta a porta serial associada ao Arduino).
5. Carregar o arquivo ELF/DWARF associado ao código fonte.
6. Selecionar as variáveis de interesse na lista de variáveis à esquerda.
7. Realizar o comando de *peek*.
8. Realizar o comando de *poke* (opcional), preenchendo a célula *poke* disponível na tabela de monitoramento com o valor desejado.

A Figura 5 apresenta a interface gráfica da ferramenta DESTRA UI, evidenciando os principais componentes: o painel de seleção de porta serial, o carregador de arquivo ELF/DWARF, a lista de variáveis disponíveis e a tabela de monitoramento onde é possível realizar operações de *peek* e *poke* de forma intuitiva.

3.4.3 Formatos ELF e DWARF

O formato **Executable and Linkable Format** (ELF) é um padrão amplamente utilizado em sistemas Unix e em arquiteturas embarcadas para representar executáveis, bibliotecas compartilhadas e arquivos objeto. Desenvolvido inicialmente pela Unix System Laboratories e posteriormente adotado pelo projeto System V Release 4 (Committee, 1995), o ELF tornou-se o formato predominante devido à sua portabilidade e flexibilidade.

Estruturalmente, um arquivo ELF organiza-se em seções e segmentos. As seções contêm informações como código executável, dados estáticos e tabelas de símbolos, enquanto os segmentos representam as partes efetivamente carregadas em memória durante a execução. Essa separação permite que o ELF seja utilizado tanto no processo de compilação e linkagem quanto em tempo de execução.

Complementarmente, o **Debugging With Attributed Record Formats** (DWARF) é um padrão de descrição de informações de depuração que pode ser incorporado em arquivos ELF (Committee, 2017). Trata-se de um formato independente da arquitetura e da linguagem de programação, projetado para prover uma representação detalhada da estrutura interna do programa. O DWARF descreve elementos como tipos de dados,

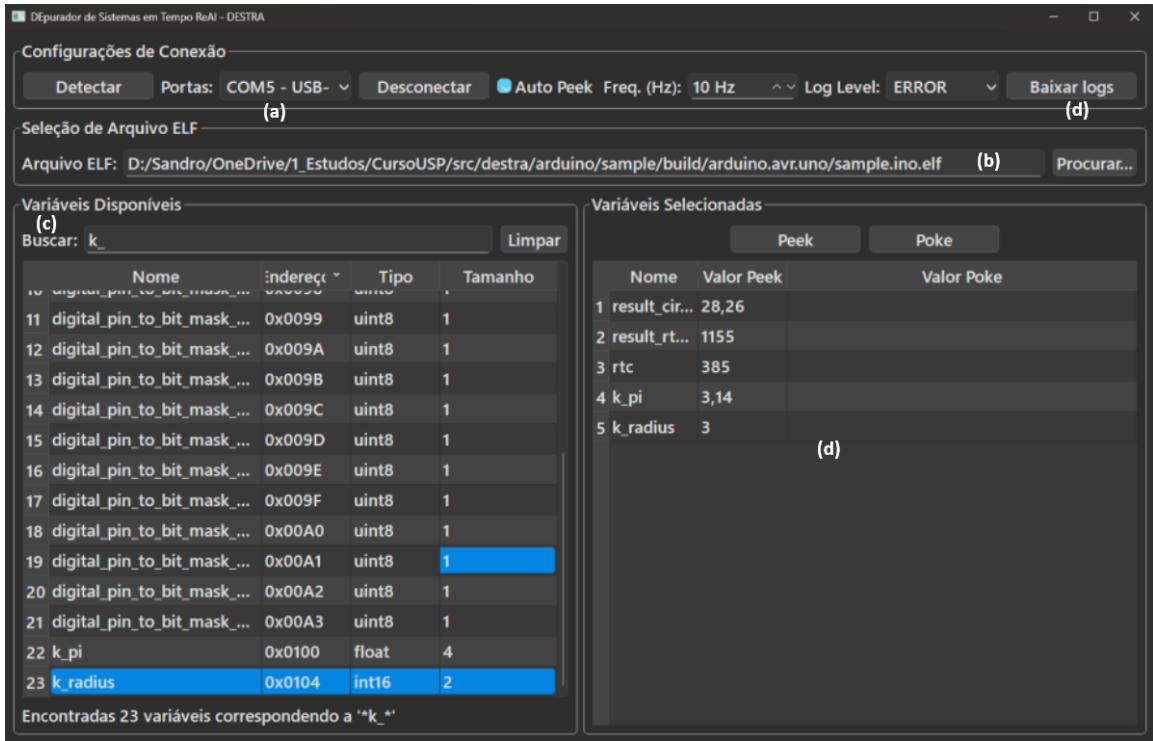


Figura 5 – Interface gráfica da ferramenta DESTRA UI. A tela apresenta os elementos principais para operação da ferramenta: (a) seleção de porta serial, (b) carregamento de arquivo ELF/DWARF, (c) painel de busca e seleção de variáveis, (d) tabela de monitoramento com operações de *peek* e *poke*, e (e) console de log para diagnóstico.

variáveis globais, estruturas, classes, funções, escopos léxicos e até mapeamentos entre instruções de máquina e linhas do código-fonte.

Essas informações são fundamentais para ferramentas de depuração, como debuggers (por exemplo, GDB), que dependem de uma correlação precisa entre o código binário executável e sua representação em alto nível.

Na prática, o ELF atua como o contêiner que organiza e armazena o binário, enquanto o DWARF fornece o conjunto de metadados necessários para inspecionar a execução e o estado do programa. Essa combinação é de particular importância em sistemas embarcados críticos, nos quais o rastreamento de execução, a análise de memória e a validação de fluxos de controle são atividades essenciais para o processo de verificação e certificação.

Portanto, ELF e DWARF podem ser compreendidos como elementos centrais da infraestrutura de desenvolvimento moderno, que oferecem suporte avançado à depuração e análise estática, mas cuja aplicação em sistemas críticos requer adaptação cuidadosa. Sua relevância ultrapassa o ambiente acadêmico e de prototipagem, consolidando-se como referência técnica na indústria de software embarcado.

A Figura 6 descreve a sequência do fluxo de carregamento do dicionário de dados na aplicação ferramenta host.

3.5 Considerações sobre Extensibilidade

Embora o protocolo atual implemente apenas as operações básicas de peek e poke, sua estrutura foi concebida com extensibilidade em mente. Entre as funcionalidades previstas para evolução estão:

- Suporte a múltiplas arquiteturas através da implementação de uma camada de abstração de hardware HAL (*hardware abstraction layer*).
- Inclusão de uma fila de comandos para processamento otimizado em casos de vários comandos chegarem ao mesmo tempo sem risco de perda de comandos.
- Inclusão de um número de sequência no cabeçalho (se a opção acima for implementada).
- Inclusão de mecanismos de integridade (CRC, Checksums).
- Extensão para comandos de *continuous peek* e *continuous poke*.
- Suporte a tipos de dados complexos como structs, unions e arrays.
- Comandos de consulta de outras informações relativas ao sistema embarcado, como CRC do software carregado (muito útil para verificação em testes).

Essas perspectivas reforçam o caráter modular e aberto da solução, permitindo sua adoção em diferentes cenários industriais e acadêmicos.

3.6 Resumo do Capítulo

A concepção e implementação do protocolo de comunicação *peek* e *poke* mostraram-se eficazes como uma solução de baixo custo e alta portabilidade para depuração em sistemas embarcados. O uso de uma máquina de estados simples, aliada a comandos minimamente estruturados, garante determinismo na interpretação das mensagens e confiabilidade na execução das operações de leitura e escrita em memória.

Essa abordagem reduz a complexidade de integração, ao mesmo tempo em que mantém a robustez necessária para cenários de prototipagem e análise em tempo real. A modularidade da solução permite a expansão do protocolo com novos comandos ou mecanismos de segurança sem comprometer o desempenho ou exigir modificações estruturais profundas.

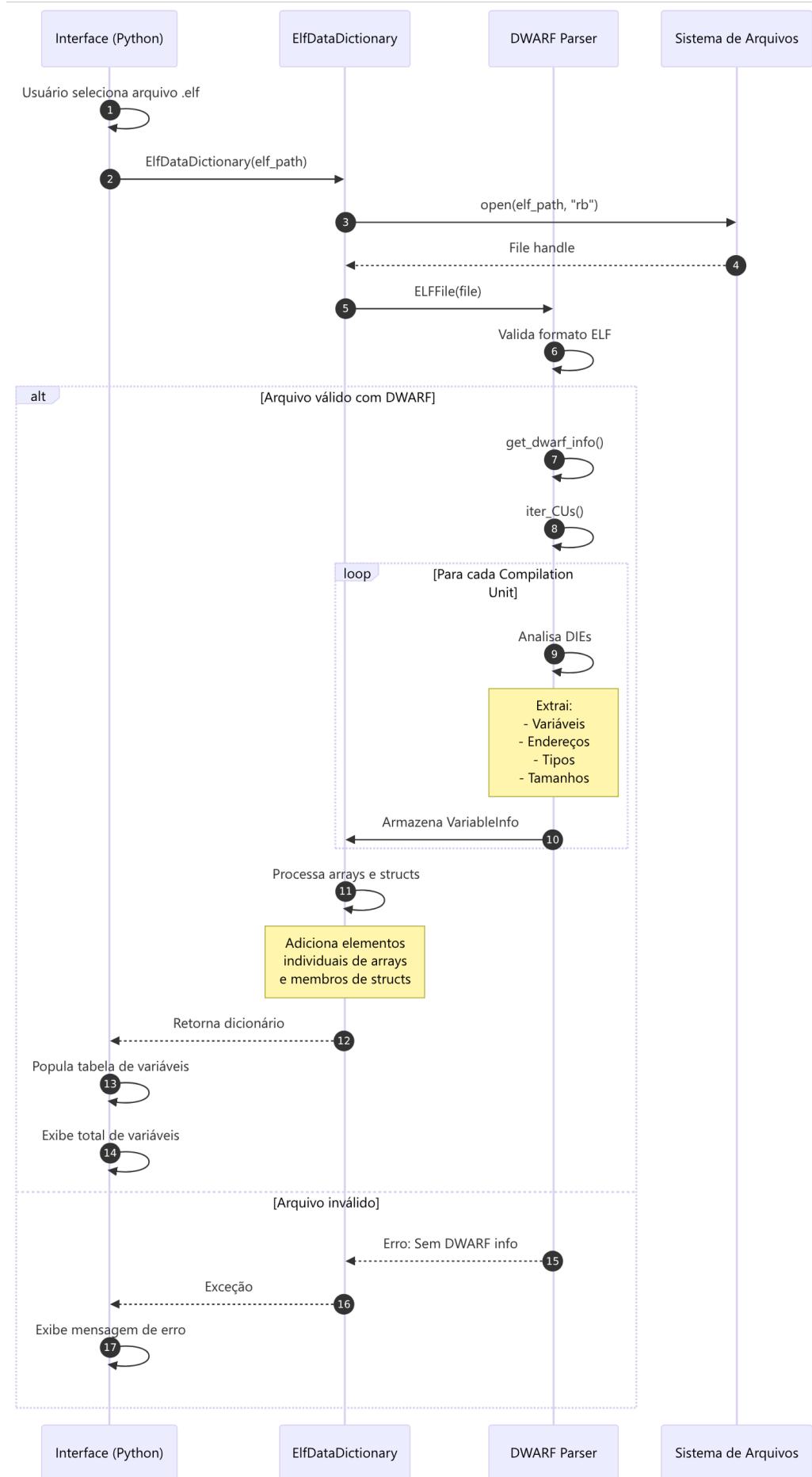


Figura 6 – Diagrama de sequência da operação de leitura do arquivo ELF/DWARF.

A distinção clara entre o software embarcado e a ferramenta host garante a separação de responsabilidades, facilitando a manutenção, a evolução e a adaptação para diferentes plataformas de hardware. Por fim, a integração com a ferramenta host desenvolvida em Python e com a interface gráfica DESTRA UI evidencia a preocupação em tornar a solução acessível tanto para desenvolvedores experientes quanto para usuários com menor familiaridade com ambientes de baixo nível.

Assim, o protocolo consolida-se como um recurso que alia praticidade, eficiência e potencial de evolução, estabelecendo uma base consistente para futuras melhorias no contexto de depuração e instrumentação de sistemas embarcados críticos.¹

¹ A implementação completa do protocolo, incluindo o firmware embarcado, a especificação dos comandos, a ferramenta host e os casos de teste utilizados durante o desenvolvimento, encontra-se disponível em um repositório público, que reúne todos os artefatos necessários para a reprodução, validação e extensão da solução proposta (Fadiga, 2025).

4 METODOLOGIA E TESTES

4.1 Ambiente de Testes

O protocolo desenvolvido neste trabalho foi implementado sobre uma plataforma de hardware específica, o Arduino Uno, escolhido por sua simplicidade arquitetural e previsibilidade de execução. Trata-se de um sistema embarcado que não possui um Sistema Operacional de Tempo Real (RTOS – Real-Time Operating System), operando em modo bare-metal, ou seja, com execução direta sobre o hardware sem a intermediação de camadas de abstração complexas.

Essa característica permite o controle total do fluxo de execução e a implementação direta do protocolo em linguagem C, utilizando a adaptação fornecida pelo ambiente Arduino. Embora o Arduino Uno não disponha de recursos típicos de um RTOS, sua biblioteca padrão oferece um conjunto abrangente de funções de baixo nível que simplificam o desenvolvimento e a manipulação de periféricos.

No contexto desta pesquisa, destaca-se a utilização da interface UART (Universal Asynchronous Receiver-Transmitter), fundamental para a comunicação serial entre o host e o sistema embarcado. Por meio dessa interface, o protocolo DESTRA é transmitido e processado, possibilitando o envio e recebimento de comandos de leitura e escrita de memória de forma estruturada e determinística.

4.1.1 Hardware - Arduino UNO

4.1.2 Arduino UNO

O Arduino UNO é uma plataforma de prototipagem eletrônica de código aberto baseada no microcontrolador ATmega328P. Desenvolvido para facilitar o aprendizado e a prototipagem rápida, o Arduino UNO é amplamente utilizado em projetos educacionais, hobbistas e aplicações embarcadas de baixa complexidade. Sua facilidade de programação, combinada com uma comunidade ativa e extensa documentação, o torna uma escolha popular para sistemas embarcados de prototipagem.

A placa possui 14 pinos digitais de entrada/saída (dos quais 6 podem ser usados como saídas PWM), 6 entradas analógicas, um cristal oscilador de 16 MHz, uma conexão USB para programação e comunicação serial, um botão de reset e um regulador de tensão. A memória Flash de 32 KB (com 0,5 KB reservado para o bootloader), combinada com 2 KB de SRAM e 1 KB de EEPROM, fornece espaço suficiente para aplicações embarcadas moderadamente complexas.

A Figura 7 apresenta a placa Arduino UNO com destaque para seus componentes

principais.

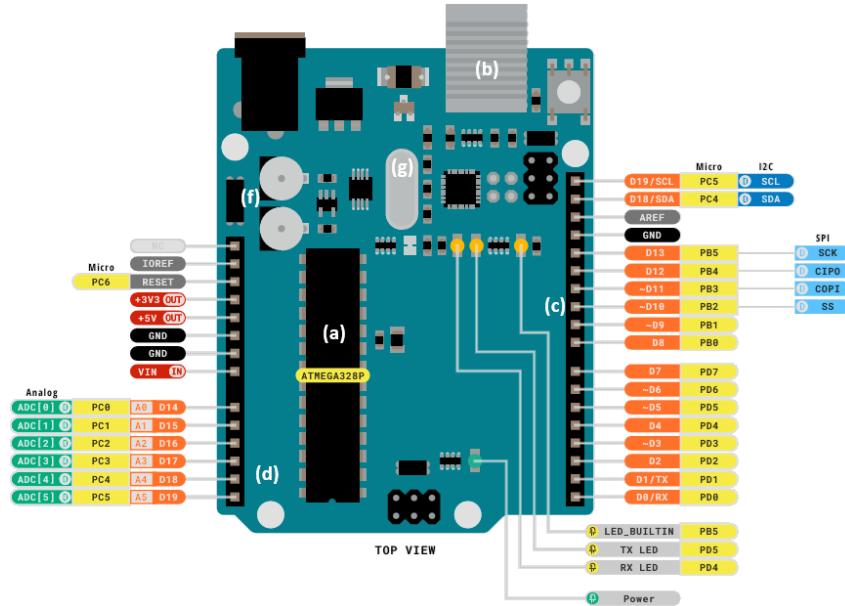


Figura 7 – Esquema do Arduino UNO. A imagem destaca os principais componentes: (a) microcontrolador ATmega328P, (b) conexão USB, (c) pinos digitais de I/O, (d) entradas analógicas, (e) botão de reset, (f) regulador de tensão, e (g) cristal oscilador de 16 MHz.

O Arduino UNO é programado através do Arduino IDE (Integrated Development Environment), uma plataforma open-source baseada em Java que simplifica o processo de escrita, compilação e upload de código. A linguagem de programação é uma variante simplificada de C, o que torna o aprendizado acessível mesmo para usuários sem experiência prévia em programação embarcada.

Para o desenvolvimento deste trabalho, o Arduino UNO foi escolhido como plataforma de implementação do protocolo DESTRA devido à sua ampla disponibilidade, baixo custo, documentação abundante e capacidade de comunicação serial via USB, que permite a integração eficiente com ferramentas host em Python.

4.1.2.1 Especificações do Arduino UNO

4.1.3 Software Embarcado

As versões de software utilizadas no desenvolvimento do protocolo DESTRA e da aplicação de testes foram:

- Arduino IDE 2.3.6
- Firmware: avr-gcc@7.3.0-atmel3.6.1-arduino
- Otimização para Debugging ativada

Componente	Especificação
Microcontrolador	ATmega328P
Memória Flash	32 KB (0,5 KB usados pelo bootloader)
SRAM	2 KB
EEPROM	1 KB (não-volátil)
Tensão de Operação	5V
Tensão de Entrada (recomendada)	7V a 12V
Tensão de Entrada (limites)	6V a 20V
Pinos Digitais de I/O	14 (6 podem ser usados como saídas PWM)
Entradas Analógicas	6
Corrente por Pino de I/O	20 mA
Corrente para Pino de 3.3V	50 mA
Frequência de Clock	16 MHz
Conexão USB	Para programação e alimentação
Protocolos de Comunicação	UART (TX/RX), I2C (SDA/SCL), SPI
Dimensões Físicas	68,6 mm × 53,4 mm
Peso	25 g

Tabela 4 – Especificações técnicas do Arduino UNO

Para a validação do protocolo DESTRA, foi implementado um código simples em Arduino (linguagem similar a C). As variáveis de interesse foram declaradas com o modificador `volatile`, o que em C garante que o compilador não realizará otimizações sobre suas atribuições. Dessa forma, assegura-se que essas variáveis permaneçam acessíveis e devidamente representadas no arquivo ELF/DWARF.

A implementação segue a estrutura padrão da plataforma Arduino, composta pelas funções principais `setup()` e `loop()`. No processo de inicialização, a função `setup()` realiza a configuração do protocolo por meio da chamada a `destraSetup()`, que também é responsável pela inicialização da porta serial do hardware. Em sequência, a função `loop()` inicia com a chamada a `destraHandler()`, encarregada de processar os comandos do protocolo DESTRA.

4.1.3.1 Código de Teste

Após essa etapa, o código executa algoritmos simples cujo propósito é fornecer um cenário de demonstração para leitura (comando `peek`) e escrita (comando `poke`) de variáveis:

Listagem 4.1 – Código de teste com variáveis para monitoramento

```
// TEST DATA (usamos variáveis como volatile
// para assegurar que estariam no arquivo .elf)
volatile unsigned long rtc = 0;
volatile float k_pi = 3.14f;
volatile uint8_t k_radius = 3;
```

```
volatile float result_circle_area;
volatile unsigned long result_rtc_x_radius;

void calculation() {
    // Exemplos de cálculos com variáveis a
    // serem monitoradas / alteradas
    result_circle_area = k_pi *
        (k_radius * k_radius);
    result_rtc_x_radius = k_radius * rtc;
    rtc += 1;
}
```

A integração entre o código embarcado e o ambiente de testes ocorre por meio da comunicação serial UART, utilizando a porta USB do Arduino UNO. Durante a execução dos testes, a ferramenta realiza o carregamento automático das variáveis presentes no arquivo ELF, utilizando as informações de depuração DWARF para mapear nomes, tipos e endereços de memória.

Esse arranjo experimental permite verificar o funcionamento correto do protocolo DESTRA, avaliando sua capacidade de manipular dados de forma confiável, dentro dos limites de tempo e integridade esperados. Além disso, a comunicação direta com o hardware, sem a intervenção de um sistema operacional, torna o ambiente altamente determinístico, o que facilita a análise de latência, confiabilidade e robustez.

4.1.4 Ferramentas Auxiliares

4.1.4.1 Osciloscópio Digital FNIRSI® DSO-153

Foi utilizado um osciloscópio digital de um canal FNIRSI® DSO-153 2-em-1 Mini 1MHz 5MS/s para monitorar pinos digitais do Arduino, permitindo a geração de formas de onda e medições de tempo/frequência.

A Figura 8 apresenta o osciloscópio FNIRSI® DSO-153 conectado ao Arduino UNO durante uma sessão de medição.

O osciloscópio digital FNIRSI® DSO-153 é um instrumento de medição portátil e compacto, projetado para aplicações de prototipagem, depuração e análise de circuitos eletrônicos. Seu design 2-em-1 oferece funcionalidades de osciloscópio e gerador de forma de onda (waveform generator) integradas em um único dispositivo, tornando-o uma ferramenta versátil para engenheiros e desenvolvedores. Com uma largura de banda máxima de 1 MHz e taxa de amostragem de 5 MS/s (milhões de amostras por segundo), o DSO-153 é adequado para análise de sinais em frequências baixas a médias, tipicamente encontradas em aplicações embarcadas, circuitos digitais de lógica, comunicação serial e controle de tempo real. A tela colorida TFT de alta resolução proporciona visualização clara de formas

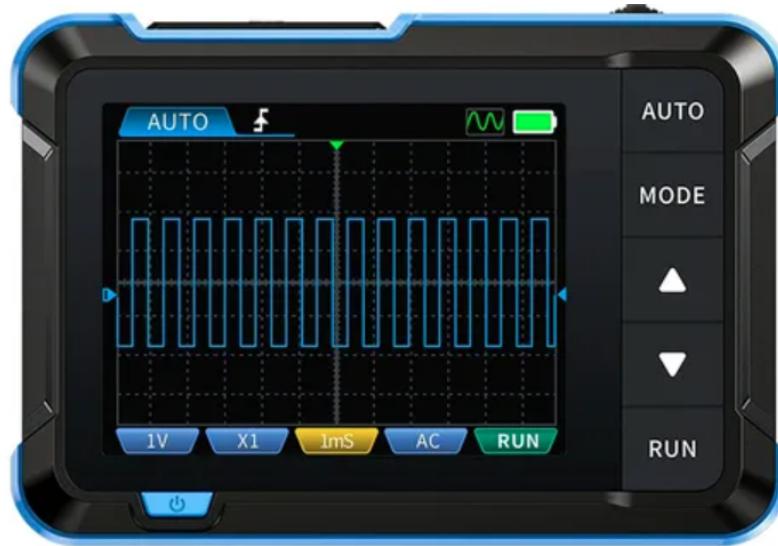


Figura 8 – Osciloscópio digital FNIRSI® DSO-153 2-em-1

de onda, facilitando a identificação de padrões, anomalias e transições de sinal.

O aparelho oferece um único canal de entrada, permitindo monitoramento simultâneo de uma linha de sinal por vez. Apesar dessa limitação, é possível realizar medições sequenciais em diferentes pontos do circuito, rotacionando a sonda entre as posições de teste. A bateria integrada proporciona portabilidade, permitindo seu uso em campo sem necessidade de alimentação externa contínua.

Para o contexto deste trabalho, o osciloscópio DSO-153 foi utilizado para validação física das métricas temporais do protocolo DESTRA, em especial: (i) medição da frequência do loop principal (100 Hz), (ii) quantificação do jitter temporal entre ciclos de execução, e (iii) determinação do tempo de processamento de comandos (command_process_time). As medições realizadas via osciloscópio complementam os dados coletados internamente pelo firmware, fornecendo uma validação independente e confiável do desempenho temporal do sistema.

4.1.4.2 Especificações Técnicas do DSO-153

4.1.4.3 Ferramenta Host

As versões de software utilizadas para o desenvolvimento e execução da ferramenta host foram:

- SO: Windows 11
- Python 3.13.5
- PySerial 3.5
- PySide6 6.9.2

Parâmetro	Valor
Largura de Banda	1 MHz
Taxa de Amostragem Máxima	5 MS/s (Mega amostras por segundo)
Canais de Entrada	1 (monofásico)
Impedância de Entrada	1 MΩ
Acoplamento	AC/DC selecionável
Tela	TFT colorida, alta resolução
Tensão de Entrada (Máxima)	±50V (com atenuação)
Funcionamento	Osciloscópio + Gerador de forma de onda
Alimentação	Bateria integrada ou USB
Portabilidade	Compacta e leve (200g)

Tabela 5 – Especificações técnicas do osciloscópio digital FNIRSI® DSO-153

A ferramenta host DESTRA UI foi desenvolvida em Python, utilizando o framework PySide6 (Qt for Python) para a criação da interface gráfica. A configuração da conexão serial é realizada de forma automática através da detecção das portas associadas a dispositivos Arduino, ou manual, caso múltiplos dispositivos sejam identificados. O parâmetro de baud rate utilizado é 115200 bps (8N1), valor que deve ser idêntico ao configurado no código embarcado.

O carregamento do dicionário de dados é realizado a partir do arquivo ELF gerado durante a compilação do código embarcado. A ferramenta executa automaticamente o parsing das informações de depuração (DWARF debug info), permitindo que as variáveis sejam apresentadas de forma estruturada e pesquisáveis.

As operações principais da ferramenta compreendem:

- **PEEK:** leitura instantânea do valor atual da variável selecionada.
- **POKE:** escrita de um novo valor, acionada por duplo clique na célula correspondente.
- **Auto PEEK:** monitoramento contínuo das variáveis selecionadas, com frequência configurável entre 1 e 100 Hz.

O recurso Auto PEEK implementa um mecanismo semelhante a um *continuous peek*, no qual a ferramenta envia comandos de leitura em intervalos regulares. O gerenciamento das variáveis pode ser feito de forma interativa, possibilitando a remoção de itens da lista ou a edição direta dos valores para sobreescrita.

Além disso, a ferramenta dispõe de configuração de níveis de log — DEBUG, INFO, WARNING e ERROR — que permitem ajustar a verbosidade das mensagens exibidas no console durante a execução.

A integração dessas funcionalidades na ferramenta DESTRA UI representa um avanço significativo na observabilidade e controle de sistemas embarcados durante as fases

de teste e validação. Ao abstrair a complexidade inerente à comunicação de baixo nível, a interface permite que o pesquisador ou engenheiro concentre seus esforços na análise do comportamento funcional do sistema.

4.2 Cenários de Teste

Os testes realizados têm como objetivo avaliar o comportamento do protocolo DESTRA sob diferentes condições de operação, com foco em dois cenários: ferramenta host e embarcado. Cada um desses cenários visa levantar as características de performance do protocolo, avaliando diferentes dimensões sob a perspectiva de testes.

A escolha destas dimensões está diretamente relacionada à viabilidade do uso do protocolo em sistemas embarcados de maior criticidade, nos quais a previsibilidade temporal, a integridade dos dados e a capacidade de recuperação frente a falhas são requisitos essenciais.

4.2.1 Cenário Host

Nos testes a partir do cenário da ferramenta host desenvolvemos um script Python (`performance_tests.py`) complementar à ferramenta host que faz uso do protocolo DESTRA.

O script realiza sequências de comando peek em uma variável inteira de tamanho 4 bytes. A listagem deste script é apresentada no Apêndice A.

Os testes criados no script rodam com sua chamada em linha de comando, realizando as operações automaticamente e gerando arquivos de relatório no formato markdown e gráficos. A Tabela 6 detalha os objetivos de cada um dos testes do script:

Cenário de Teste	Descrição / Objetivo	Métricas Coletadas
Latência	Mede o tempo de ida e volta (round-trip) entre envio de comando e resposta	Latência média, mínima e máxima; desvio padrão; jitter
Estresse	Mede a estabilidade sob carga contínua durante intervalo prolongado	Throughput de comandos; latência média; número de erros
Rajada	Simula envio rápido e consecutivo de comandos	Tempo médio; jitter; throughput máximo; detecção de perdas

Tabela 6 – Cenários de teste da ferramenta host

4.2.2 Cenário Embarcado

Para este cenário, o protocolo DESTRA implementado para o sistema embarcado foi instrumentado para coletar e armazenar dados de performance. O protocolo recebeu um novo comando, 0xF3, para suportar o envio dos dados de performance mediante uma requisição realizada pela ferramenta host/script de testes.

Além da coleta de dados de performance, foram inseridas no protocolo chamadas de ativações de pinos digitais do Arduino em determinados pontos para serem realizadas medições com o osciloscópio.

4.2.2.1 Variáveis de Performance

A Listagem 4.2 apresenta as variáveis criadas para o monitoramento e armazenamento dos dados de performance:

Listagem 4.2 – Variáveis de instrumentação do protocolo

```
// Comando especial para recuperar logs
#define CMD_GET_PERF_LOG 0xF3

volatile unsigned long frameCounter = 0;
volatile uint16_t frameRate = 0;
volatile uint16_t frameJitter = 0;
volatile uint16_t commandSequence = 0;
volatile unsigned long commandStartCounter = 0;
volatile unsigned long commandEndCounter = 0;
volatile unsigned long lastFrameTime = 0;
volatile unsigned long commandReceiveTime = 0;
volatile unsigned long commandProcessTime = 0;
volatile unsigned long lastDeltaTime = 0;

#define PERF_BUFFER_SIZE 100
struct PerfLog {
    unsigned long frameCounter;
    uint16_t frameRate;
    uint16_t frameJitter;
    uint16_t commandSequence;
    uint16_t commandFrameCounterDelta;
    unsigned long commandProcessTime;
};

PerfLog perfBuffer[PERF_BUFFER_SIZE];
uint8_t perfIndex = 0;
```

4.2.2.2 Pinos de Debug para Osciloscópio

Os seguintes pinos foram configurados para medições com osciloscópio:

Listagem 4.3 – Pinos de debug para osciloscópio

```
#define PIN_TRIGGER_RX 2 // Pulso ao receber
#define PIN_TRIGGER_TX 3 // Pulso ao enviar
#define PIN_FRAME_TOGGLE 4 // Toggle cada loop
#define PIN_BUSY 5 // Alto durante proc.

#define PULSE_RX() { digitalWrite(PIN_TRIGGER_RX, \
    HIGH); delayMicroseconds(10); \
    digitalWrite(PIN_TRIGGER_RX, LOW); }
#define PULSE_TX() { digitalWrite(PIN_TRIGGER_TX, \
    HIGH); delayMicroseconds(10); \
    digitalWrite(PIN_TRIGGER_TX, LOW); }
#define TOGGLE_FRAME() { \
    digitalWrite(PIN_FRAME_TOGGLE, \
    !digitalRead(PIN_FRAME_TOGGLE)); }
#define SET_BUSY(state) { \
    digitalWrite(PIN_BUSY, state); }
```

4.2.2.3 Métricas Coletadas

A Tabela 7 apresenta as métricas coletadas durante os testes:

Campo	Descrição	Unidade
frameCounter	Contador absoluto de frames (incrementado a cada iteração do loop)	contagem
frameRate	Frequência de execução do loop principal	Hz
frameJitter	Diferença entre durações consecutivas de frames	μs
commandSequence	Número sequencial do comando recebido	—
commandFrameCounterDelta	Número de frames entre início e fim do comando	contagem
commandProcessTime	Tempo total de processamento de um comando	μs

Tabela 7 – Métricas de performance coletadas

4.2.2.4 Programa Principal

O programa principal (loop) do protocolo DESTRA foi modificado para operar em 100 Hz, de forma a executar sob um tempo pré-definido e com determinismo:

Listagem 4.4 – Loop principal do protocolo DESTRA

```
void loop() {
    unsigned long currentFrameTime = micros();
    unsigned long deltaTime =
        currentFrameTime - lastFrameTime;

    // Calcular framerate (Hz)
    if (deltaTime > 0) {
        frameRate = 1000000.0 / deltaTime;
    }

    // Calcular jitter
    frameJitter = abs((long)deltaTime -
        (long)lastDeltaTime);
    lastDeltaTime = deltaTime;

    // Toggle pino de frame
    TOGGLE_FRAME();
    frameCounter++;

    // Processar comandos DESTRA
    destraHandler();

    // Executar calculos de exemplo
    calculation();

    // Ajustar para ~100Hz (10ms)
    unsigned long elapsed =
        micros() - currentFrameTime;
    if (elapsed < 10000) {
        delayMicroseconds(10000 - elapsed);
    }
    lastFrameTime = currentFrameTime;
}
```

4.3 Resultados Obtidos

A execução dos testes foi realizada conectando-se o Arduino ao host e utilizando as ferramentas descritas nas seções anteriores. Os cenários foram montados da seguinte forma:

- **Cenário 1:** Script Python `performance_tests.py` + código Arduino instrumentado.

- **Cenário 2:** Ferramenta Host (DESTRA UI) + Osciloscópio + código Arduino instrumentado.

4.3.1 Resultados para o Cenário 1

Para capturar os dados para o Cenário de testes 1 o script Python `performance_tests.py` foi executado através do seguinte comando em um "shell":

```
> python .\performance_tests.py COM5
```

Este comando executou automaticamente a sequência dos três testes (Latência, Estresse e Rajada), coletando dados de performance internos e gerando relatórios conforme a saída do script exibida em 4.5.

Listagem 4.5 – Saída do programa `performance_tests.py` durante execução dos testes

```
2025-10-22 21:13:04 - DESTRA.Protocol - INFO - Conectando
2025-10-22 21:13:06 - DESTRA.Protocol - INFO - Conectado com sucesso!
2025-10-22 21:13:06 - DESTRA.Tester - INFO - Iniciando teste de latência: 100 amostras
2025-10-22 21:13:07 - DESTRA.Tester - INFO - Teste de latência concluído
2025-10-22 21:13:07 - DESTRA.Tester - INFO - Iniciando download de performance
2025-10-22 21:13:07 - DESTRA.Protocol - INFO - === DUMP DE LOGS (Arduino) ===
2025-10-22 21:13:07 - DESTRA.Protocol - INFO - Processando 99 entradas
2025-10-22 21:13:07 - DESTRA.Protocol - INFO - === FIM DO DUMP ===
2025-10-22 21:13:11 - DESTRA.Performance - INFO - Gráficos salvos
2025-10-22 21:38:00 - DESTRA.Tester - INFO - Iniciando teste de stress: 60s @ 100Hz
2025-10-22 21:39:00 - DESTRA.Tester - INFO - Teste de stress concluído
2025-10-22 21:39:00 - DESTRA.Tester - INFO - Iniciando download de performance
2025-10-22 21:39:00 - DESTRA.Performance - INFO - Gráficos salvos
2025-10-22 21:57:58 - DESTRA.Tester - INFO - Iniciando teste de burst: 10x1000 cmd's
2025-10-22 21:59:48 - DESTRA.Tester - INFO - Teste de burst concluído
2025-10-22 21:59:48 - DESTRA.Tester - INFO - Iniciando download de performance
2025-10-22 21:59:48 - DESTRA.Protocol - INFO - === DUMP DE LOGS (Arduino) ===
2025-10-22 21:59:48 - DESTRA.Protocol - INFO - Processando 99 entradas
2025-10-22 21:59:48 - DESTRA.Protocol - INFO - === FIM DO DUMP ===
2025-10-22 21:59:49 - DESTRA.Performance - INFO - Gráficos salvos
2025-10-22 21:59:49 - DESTRA.Protocol - INFO - Desconectado do Arduino
```

Nota: Porém é preciso ressaltar que a limitação de memória do Arduino permite guardar somente 100 registros de dados de performance. Logo para os testes mais longos como Estresse e Burst os dados internos são limitados a apenas 100 amostras.

4.3.2 Arquivos de Relatório Gerados

Ao término da execução, o script gerou automaticamente os seguintes arquivos de relatório:

Os timestamps nos nomes dos arquivos (ex: 20251022_211307) permitem identificar rapidamente quando cada teste foi executado, facilitando o controle de versão e rastreabilidade dos resultados.

Arquivo	Descrição
Latencia_*.md	Relatório em Markdown contendo métricas detalhadas do teste de latência (média, mediana, desvio padrão, percentis)
Latencia_*.png	Gráfico visual dos dados de latência e jitter, facilitando análise visual de distribuição e outliers
Estresse_*.md	Relatório em Markdown com resultados do teste de estresse de 60 segundos contínuos
Estresse_*.png	Gráfico de desempenho sob carga contínua, evidenciando estabilidade temporal
Burst_*.md	Relatório em Markdown com análise de 10.000 comandos em modo rajada
Burst_*.png	Gráfico de throughput e latência durante picos de alta frequência de requisições

Tabela 8 – Arquivos de relatório gerados automaticamente pelo script de testes

A seguir apresenta-se um resumo detalhado e análise crítica dos dados obtidos em cada cenário de teste.

4.3.3 Teste de Latência

O gráficos a seguir apresentam resultados do teste de latência: (a) latência ao longo do tempo, (b) distribuição da latência, (c) jitter temporal e (d) análise estatística dos valores medidos, e foram obtidos a partir da execução do script Python `performance_tests.py`.

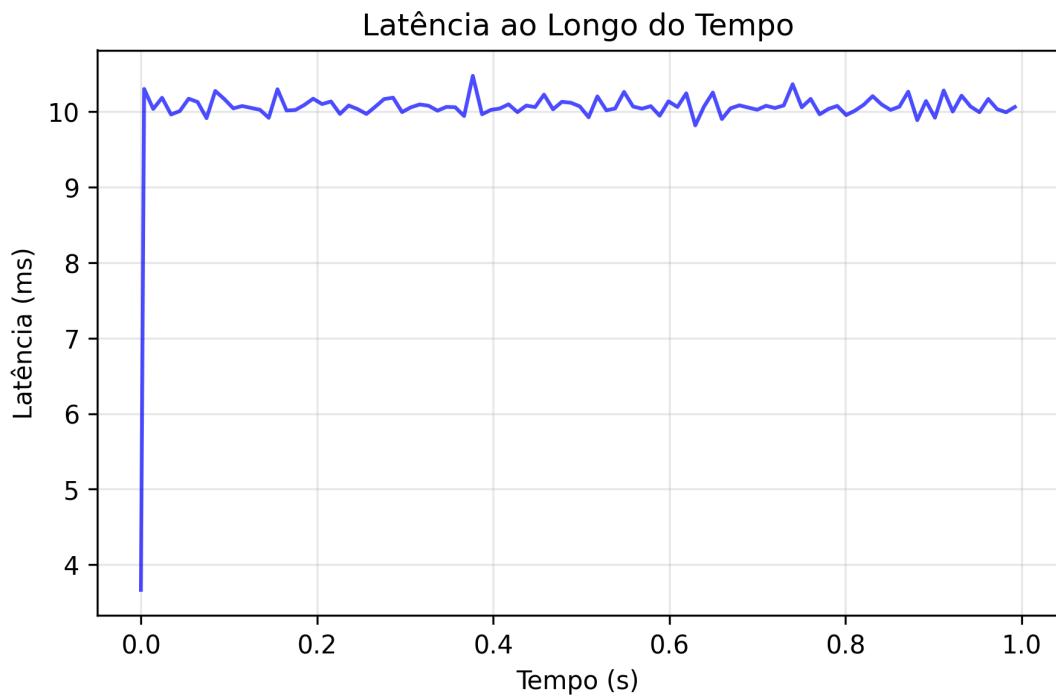


Figura 9 – Latência ao longo do tempo (a)

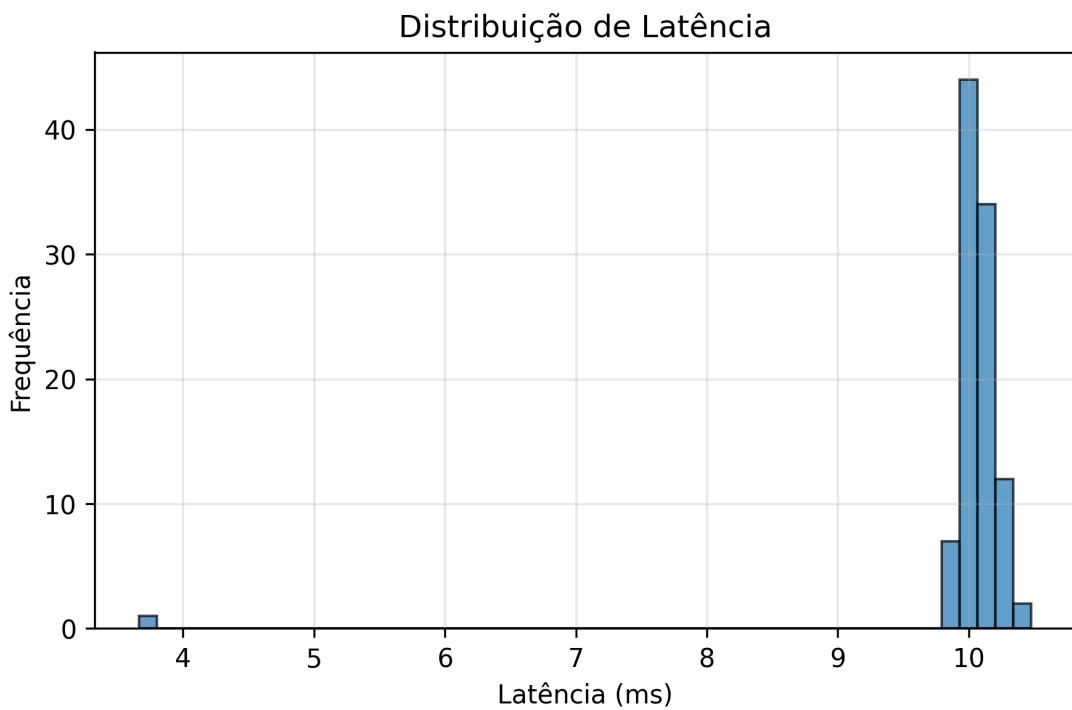


Figura 10 – Distribuição da latência (b)

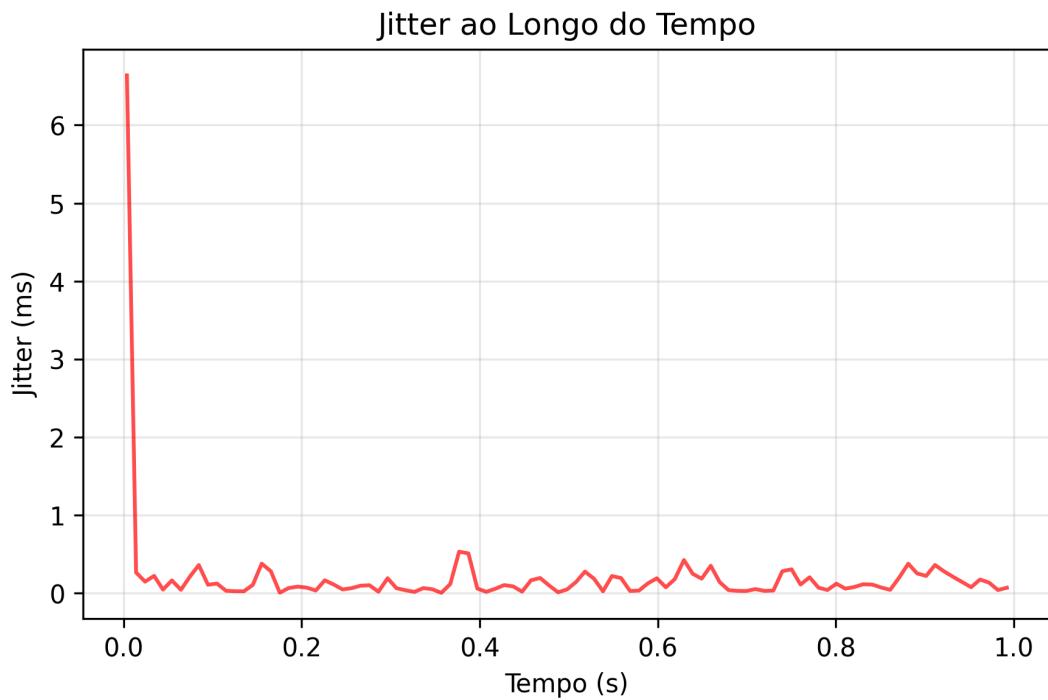


Figura 11 – Jitter ao longo do tempo (c)

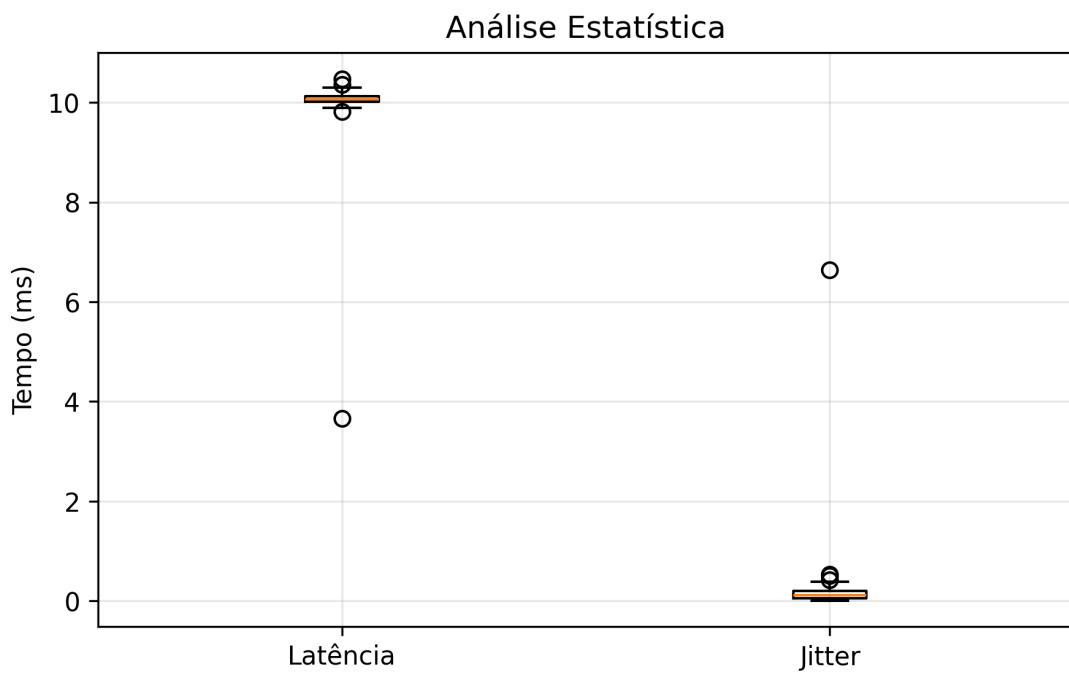


Figura 12 – Análise estatística da latência (d)

Para o teste de latência, foi configurado um número de amostras (operações de peek) igual a 100. Os resultados obtidos foram:

Para os resultados de latência e jitter, a média de frame rate observada foi de aproximadamente 98 a 100 Hz, conforme esperado. O jitter médio permaneceu dentro

Métrica	Valor
Total de Medidas	100
Medições Bem-Sucedidas	100
Taxa de Erro (%)	0.0
Latência Média (ms)	10.016
Latência Mediana (ms)	10.064
Desvio Padrão (ms)	0.647
Latência Mínima (ms)	3.665
Latência Máxima (ms)	10.476
P95 (ms)	10.276
P99 (ms)	10.367
Jitter Médio (ms)	0.201

Tabela 9 – Dados de performance do teste de latência (host)

da faixa de 0,20 ms, indicando estabilidade temporal adequada. O tempo médio de processamento de comando foi inferior a 0,25 ms, mostrando que o protocolo possui sobrecarga mínima.

Métrica (Firmware)	Valor
Total de Amostras	99
Frame Jitter Médio (ms)	0.0022
Frame Jitter Mediana (ms)	0.0000
Desvio Padrão (ms)	0.0035
Tempo de Comando Médio (ms)	0.732
Gaps no Frame Counter	0
Gaps na Sequência de Comandos	0

Tabela 10 – Dados de performance embarcada (teste de latência)

4.3.3.1 Análise de resultados do teste de latência

A latência média de 10,016 ms indica um tempo de resposta estável e previsível. O desvio padrão de apenas 0,64 ms demonstra baixa variabilidade temporal. A inexistência de gaps confirma ausência de perda de pacotes. Essa estabilidade sugere que o protocolo DESTRA introduz mínimo overhead de comunicação.

Aspecto	Observação
Latência média (host)	~10.016 ms — consistente, com baixa dispersão
Jitter médio (host)	~0.2 ms — boa estabilidade temporal
Frame rate médio (firmware)	~99 fps — ciclo estável
Tempo médio de comando	~0.731 ms — processamento rápido
Anomalias detectadas	Nenhuma — sequência íntegra

Tabela 11 – Resumo da análise do teste de latência

4.3.4 Teste de Estresse

O gráficos a seguir apresentam resultados do teste de estresse: (a) latência ao longo do tempo, (b) distribuição da latência, (c) jitter temporal e (d) análise estatística dos valores medidos, e foram obtidos a partir da execução do script Python `performance_tests.py`.

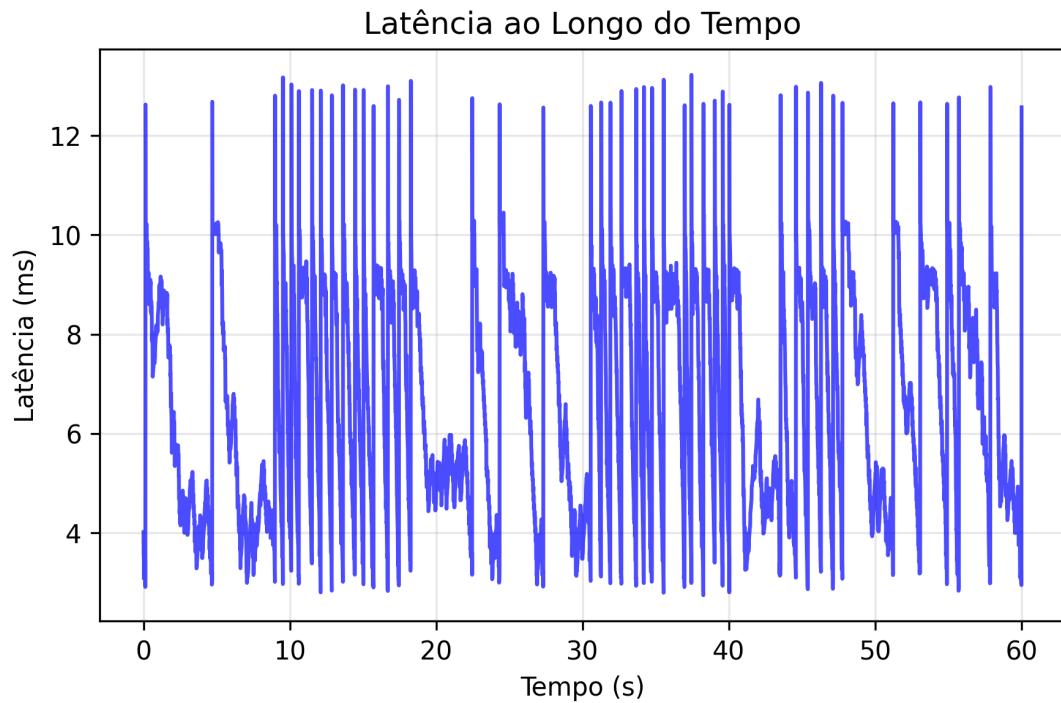


Figura 13 – Latência ao longo do tempo (a)

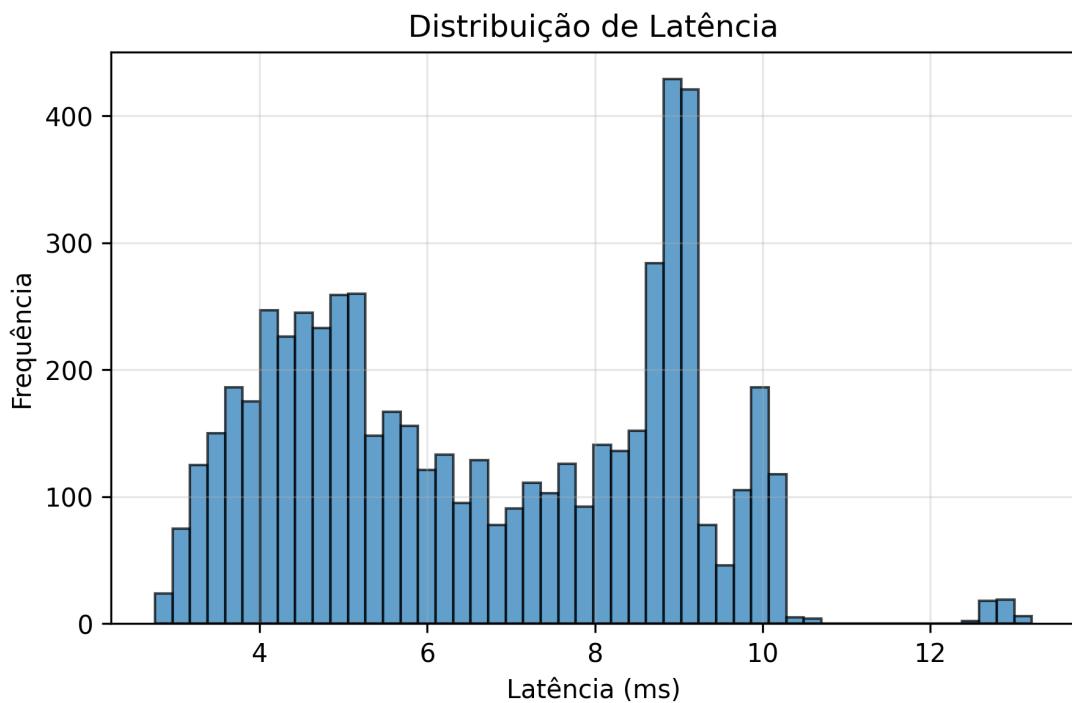


Figura 14 – Distribuição da latência (b)

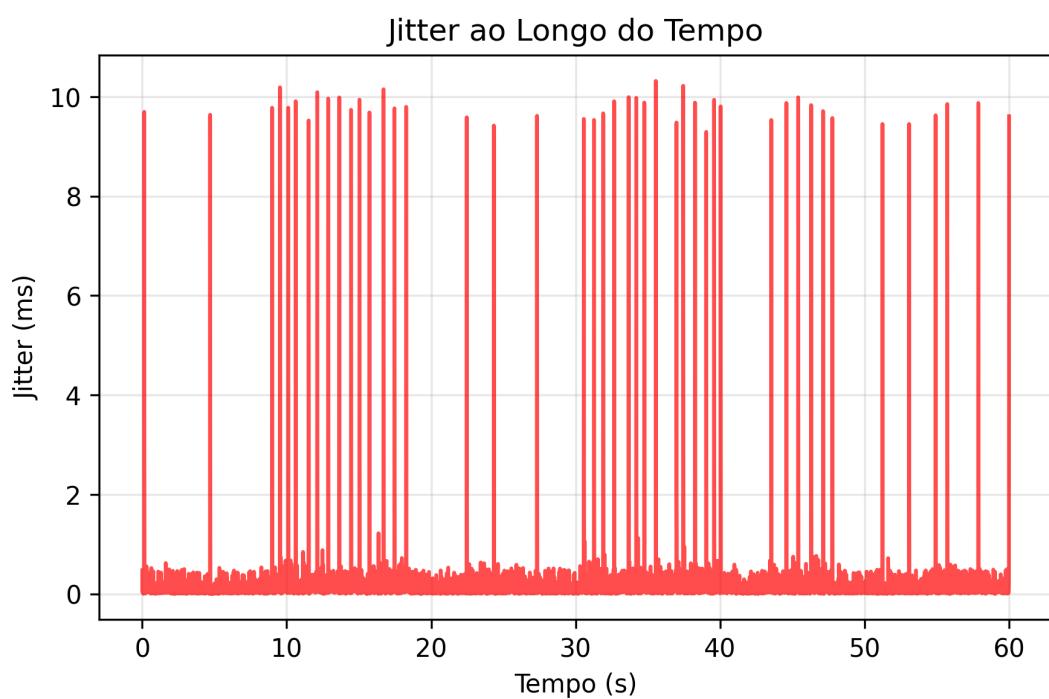


Figura 15 – Jitter ao longo do tempo (c)

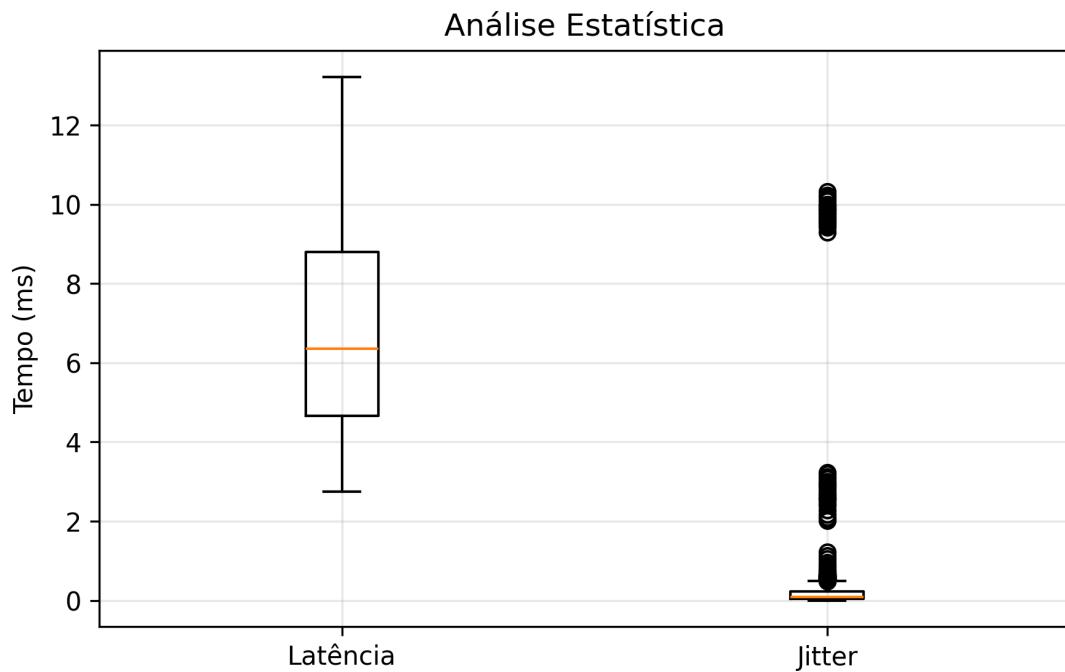


Figura 16 – Análise estatística da latência (d)

Para o teste de estresse, foi configurada uma duração de 60 segundos realizando operações de peek contínuas. Os resultados foram:

Métrica	Valor
Total de Medições	5.897
Medições Bem-Sucedidas	5.897
Taxa de Erro (%)	0.0
Latência Média (ms)	6.628
Latência Mediana (ms)	6.352
Desvio Padrão (ms)	2.213
Latência Mínima (ms)	2.750
Latência Máxima (ms)	13.215
P95 (ms)	10.027
P99 (ms)	10.272
Jitter Médio (ms)	0.236

Tabela 12 – Dados de performance do teste de estresse (host)

4.3.4.1 Análise dos resultados do teste de estresse

Durante 60 segundos de operação contínua, o sistema manteve taxa de erro zero. A pequena variação observada pode ser atribuída a interferências no buffer serial. Apesar desses picos, o frame rate médio permaneceu estável. Nenhum gap de sequência de comandos foi identificado, confirmando integridade lógica do protocolo sob estresse.

Métrica (Firmware)	Valor
Total de Medições	5.905
Taxa de Erro (%)	0.0
Frame Jitter Médio (ms)	0.0023
Frame Rate (fps)	99.0
Comando Process Time Médio (ms)	0.833
Gaps de Frame Counter	90
Gaps de Command Sequence	0

Tabela 13 – Dados de performance embarcada (teste de estresse)

Aspecto	Observação
Latência média (host)	~6,63 ms — dentro do esperado sob carga
Jitter médio (host)	~0.23 ms — boa estabilidade
Frame rate médio (firmware)	~99 fps — ciclo estável
Tempo médio de comando	~0.832 ms — processamento uniforme
Anomalias detectadas	Pequenos gaps pontuais, sem impacto crítico

Tabela 14 – Resumo da análise do teste de estresse

4.3.5 Teste de Rajada (Burst)

O gráficos a seguir apresentam resultados do teste de rajada: (a) latência ao longo do tempo, (b) distribuição da latência, (c) jitter temporal e (d) análise estatística dos valores medidos, e foram obtidos a partir da execução do script Python `performance_tests.py`.

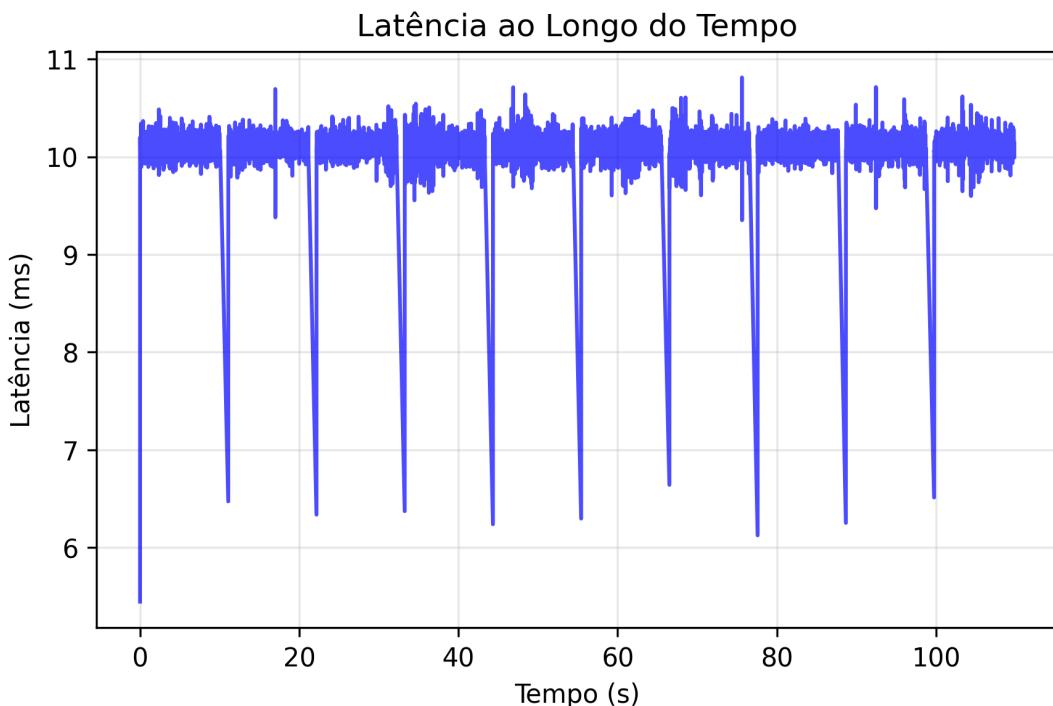


Figura 17 – Latência ao longo do tempo (a)

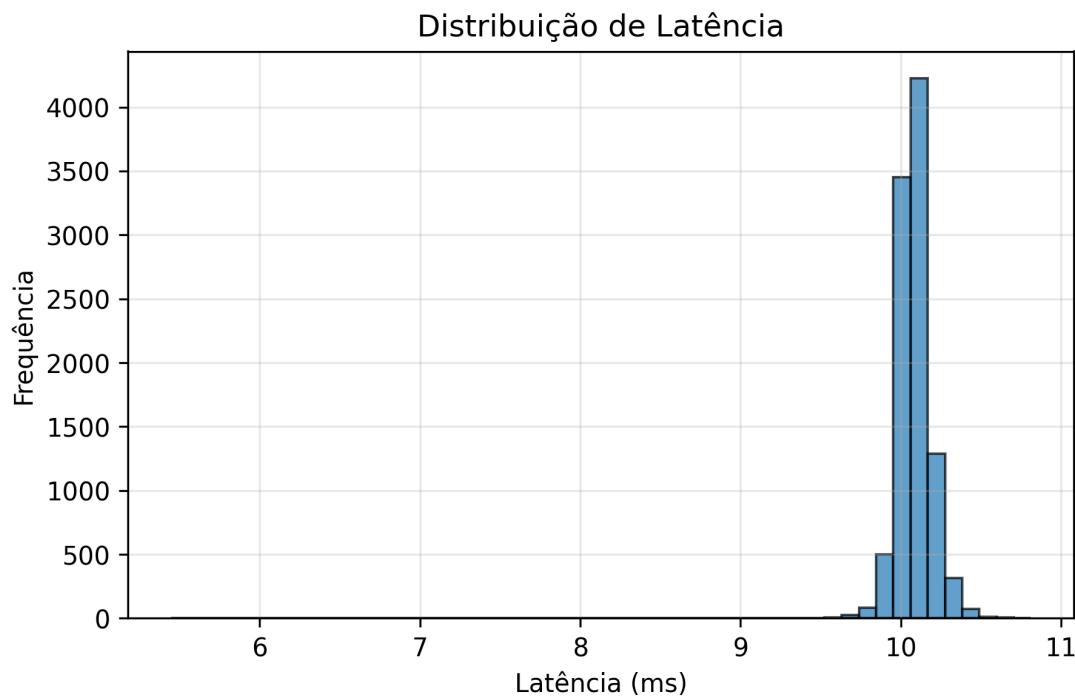


Figura 18 – Distribuição da latência (b)

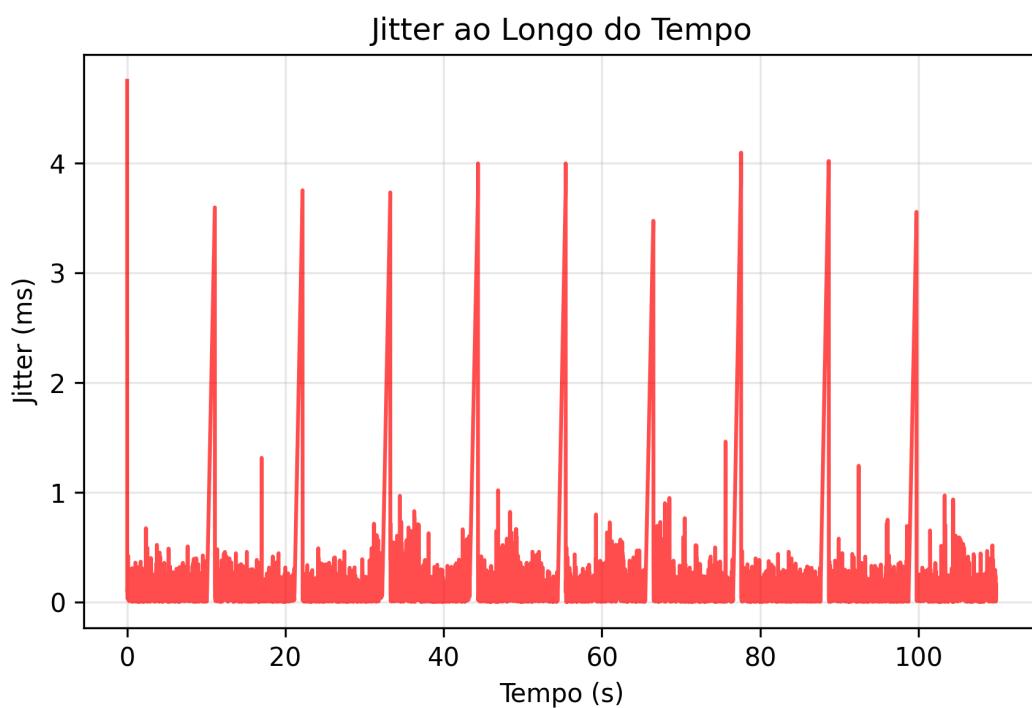


Figura 19 – Jitter ao longo do tempo (c)

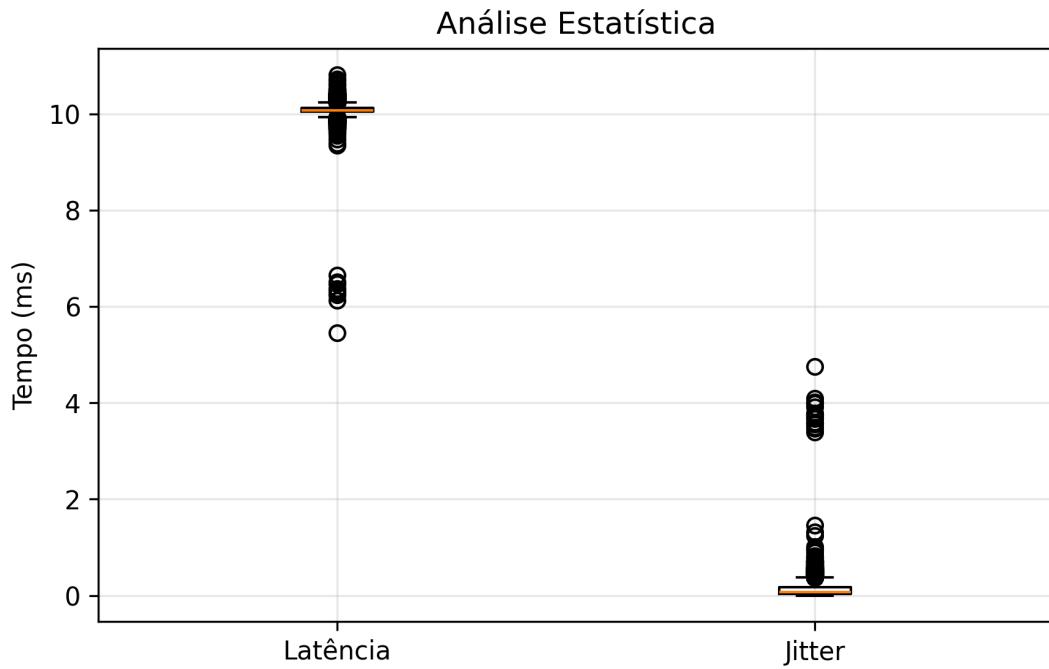


Figura 20 – Análise estatística da latência (d)

O teste de rajada foi executado disparando um total de 10.000 comandos (1.000 em sequências de 10 chamadas). Os resultados foram:

Métrica	Valor
Total de Medições	10.000
Medições Bem-Sucedidas	10.000
Taxa de Erro (%)	0.0
Latência Média (ms)	10.081
Latência Mediana (ms)	10.069
Desvio Padrão (ms)	0.158
Latência Mínima (ms)	5.448
Latência Máxima (ms)	10.811
P95 (ms)	10.262
P99 (ms)	10.374
Jitter Médio (ms)	0.122

Tabela 15 – Dados de performance do teste de rajada (host)

4.3.5.1 Análise dos resultados do teste de rajada

No cenário de rajada, o protocolo processou 10.000 operações sem erros, mantendo latência média de 10,08 ms e jitter inferior a 0,12 ms. Esse comportamento demonstra excelente estabilidade sob alta taxa de requisições.

Métrica (Firmware)	Valor
Total de Amostras	99
Frame Jitter Médio (ms)	0.0021
Frame Rate (fps)	99.0
Comando Process Time Médio (ms)	2.089
Gaps no Frame Counter	0
Gaps na Command Sequence	0

Tabela 16 – Dados de performance embarcada (teste de rajada)

Aspecto	Observação
Latência média (host)	10,08 ms — dentro do esperado
Jitter médio (host)	0,12 ms — estabilidade excelente
Jitter embarcado	0,002 ms — precisão excelente
Tempo de processamento	Média 2,09 ms, eventos até 135 ms
Taxa de erro	0,0 % — nenhuma falha
Integridade de sequênci a	Total — sem perdas detectadas

Tabela 17 – Resumo da análise do teste de rajada

4.3.6 Resultados para o Cenário 2

A execução do Cenário 2 foi feita com o osciloscópio conectado com a sonda as portas do Arduino e para cada teste a sonda foi movida para o pino alvo do teste, a ferramenta host (DESTRA UI) foi utilizada para estimular as saídas dos pinos enviando comandos de peek a taxas variadas entre 10hz, 100hz e 1khz.

Devido ao osciloscópio possuir apenas um canal, foi possível monitorar dois aspectos principais:

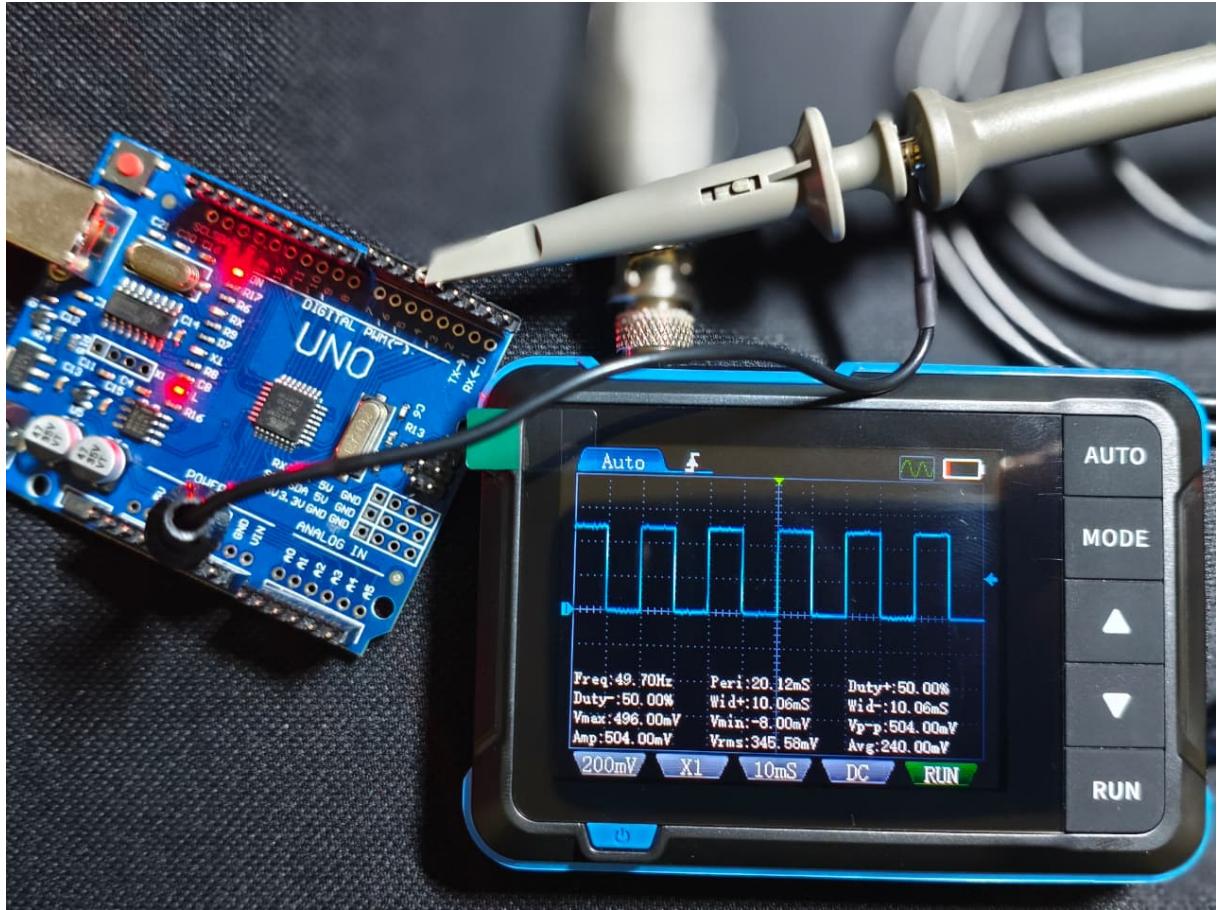


Figura 21 – Setup do osciloscópio conectado ao Arduino é mostrado durante medição de sinais do Arduino, com a sonda conectada a um dos pinos de debug. A tela colorida TFT exibe a forma de onda capturada em tempo real, permitindo análise de frequência, jitter e tempo de processamento de comandos

Métrica	Fonte (pinos)	Descrição
t_{BUSY}	PIN_BUSY	Duração da execução interna do comando
f_{LOOP}	PIN_FRAME_TOGGLE	Frequência do loop (deve ser 100 Hz)
Δf_{LOOP}	PIN_FRAME_TOGGLE	Jitter percentual no ciclo do loop

Tabela 18 – Métricas medidas com osciloscópio

4.3.6.1 Testes de Tempo de Comando

4.3.6.2 Medição a 10 Hz

Para os envios de comandos a 10 Hz, a frequência de detecção de comandos (14,81 Hz) é próxima à frequência de envio e substancialmente inferior à frequência de execução do loop principal (100 Hz). Temos um comando peek sendo processado a cada 10 frames aproximadamente.

A Figura 22 demonstra o resultado da medição com o osciloscópio para os comandos

a 10 Hz:

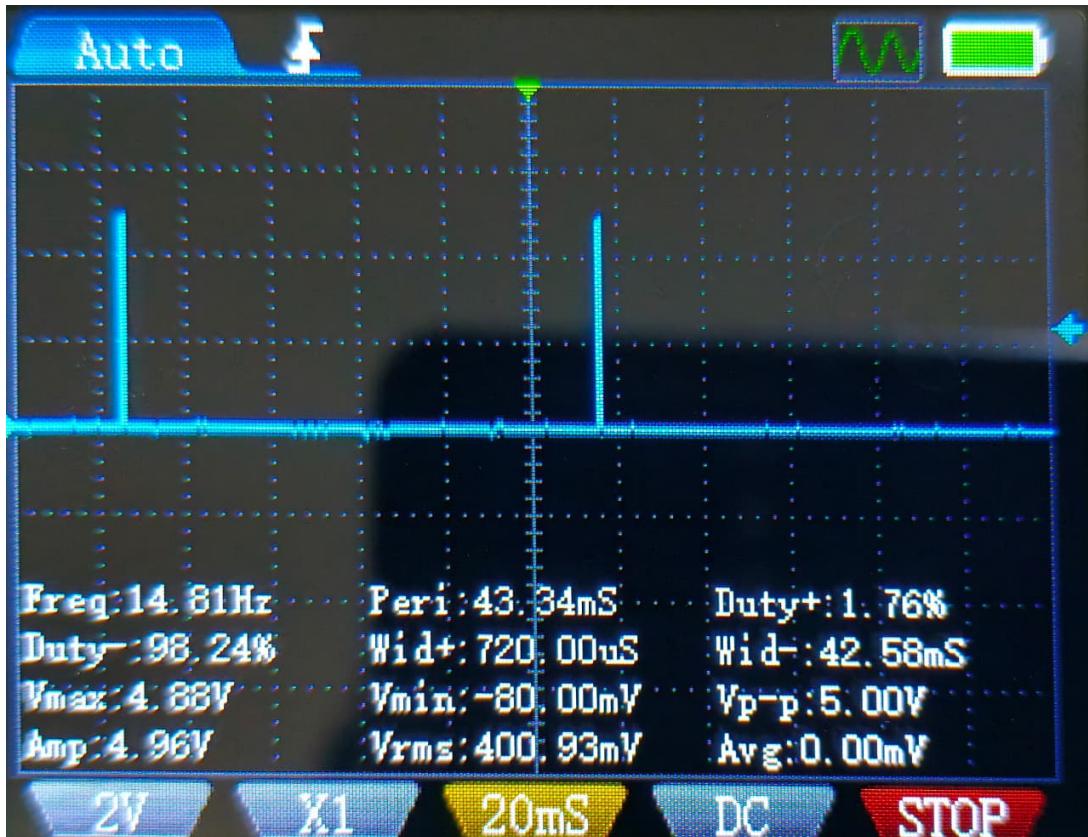


Figura 22 – Medição de tempo de comando a 10 Hz.

A Tabela 19 apresenta os parâmetros de temporização medidos pelo osciloscópio durante este teste:

4.3.6.3 Análise dos Resultados a 10 Hz

Os resultados da medição permitem as seguintes observações:

- A frequência medida de 14.81 Hz está próxima à taxa esperada de envio de comandos (10 Hz), com a diferença explicada pela variabilidade de timing e jitter inerente ao Arduino.
- O período de 43.34 ms corresponde aproximadamente a 4,3 ciclos do loop de 100 Hz (período teórico de 10 ms), indicando que um comando é processado a cada 4-5 iterações do loop.
- A largura do pulso de 720 μs (0,72 ms) representa o tempo de processamento efetivo de um comando, alinhado com os dados coletados internamente pelo firmware ($\text{command_process_time} \approx 0,73 \text{ ms}$).

Parâmetro	Valor	Interpretação
Frequência (Freq)	14.81 Hz	Frequência de detecção dos pulsos de comando (PIN_BUSY)
Período (Peri)	43.34 ms	Intervalo de tempo entre detecções de comandos consecutivos
Duty Cycle	98.24%	Percentual de tempo que o sinal permanece em nível alto
Duty+	1.76%	Percentual de tempo que o pulso fica ativo (comando processando)
Largura do Pulso (Widt)	720.00 μ s	Duração do pulso de processamento do comando
Vmax	4.88 V	Tensão máxima do sinal capturado
Vmin	-80.00 mV	Tensão mínima do sinal (ruído de linha)
Vp-p (Pico a Pico)	5.00 V	Amplitude total da forma de onda
Vrms	400.93 mV	Valor RMS (raiz quadrada média) do sinal
Amplitude (Amp)	4.96 V	Amplitude média do sinal

Tabela 19 – Parâmetros de temporização medidos a 10 Hz — Teste de tempo de comando

- O duty cycle de 98.24% indica que o sistema permanece em espera entre comandos, com apenas 1.76% do tempo dedicado ao processamento ativo, refletindo a eficiência do protocolo em cenários de baixa frequência.
- A amplitude de 5 V confirma que o sinal está dentro dos limites esperados para lógica digital de 5V do Arduino.

4.3.6.4 Medição a 100 Hz

Aumentando a frequência de envios de comandos para 100 Hz, a forma de onda permite a leitura da frequência próxima à execução do loop, indicando pelo menos 1 comando peek por ciclo. Os resultados da medição são apresentados na Tabela 20:

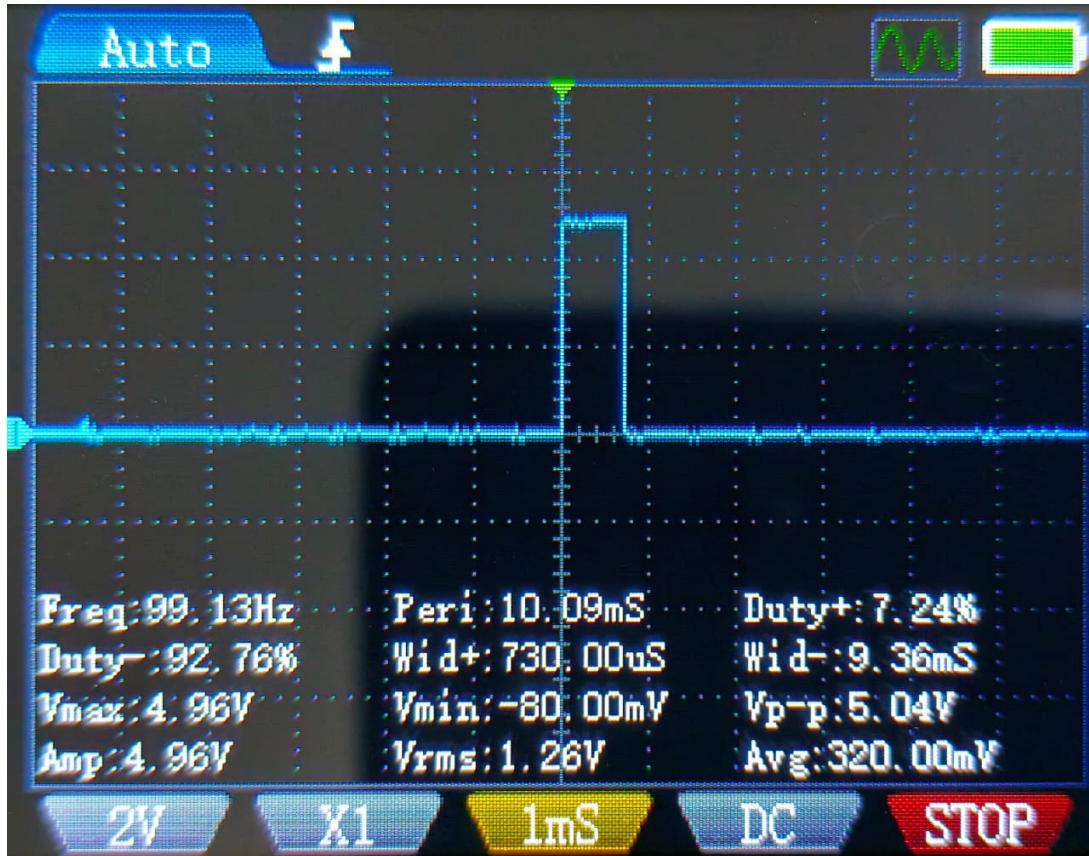


Figura 23 – Medição de tempo de comando a 100 Hz.

4.3.6.5 Análise dos Resultados a 100 Hz

Os resultados da medição a 100 Hz permitem as seguintes observações:

- A frequência medida de 99.13 Hz está extremamente próxima aos 100 Hz esperados pela taxa de envio de comandos, demonstrando excelente alinhamento temporal entre o host e o firmware.
- O período de 10.09 ms corresponde praticamente a um ciclo completo do loop de 100 Hz (período esperado de 10 ms), indicando que em média um comando é processado por iteração do loop.
- A largura do pulso de 732 μ s (0,732 ms) permanece praticamente constante em relação à medição a 10 Hz (720 μ s), reforçando a consistência do tempo de processamento de comandos (command_process_time).
- O duty cycle de 7.26% indica que sob carga de 100 Hz, o sistema dedica aproximadamente 7,26% do tempo ao processamento ativo, deixando margem de segurança para outras operações.
- A amplitude de 5.20 V confirma níveis de sinal saudáveis e sem distorção, mesmo sob taxa de comandos 10x maior que a primeira medição.

Parâmetro	Valor	Interpretação
Frequência (Freq)	99.13 Hz	Frequência de detecção dos pulsos de comando (próxima aos 100 Hz esperados)
Período (Peri)	10.09 ms	Intervalo de tempo entre detecções de comandos consecutivos
Duty Cycle	92.74%	Percentual de tempo que o sinal permanece em nível alto
Duty+	7.26%	Percentual de tempo que o pulso fica ativo (comando processando)
Largura do Pulso (Widt)	732.00 μ s	Duração do pulso de processamento do comando
Vmax	5.12 V	Tensão máxima do sinal capturado
Vmin	-80.00 mV	Tensão mínima do sinal (ruído de linha)
Vp-p (Pico a Pico)	5.20 V	Amplitude total da forma de onda
Vrms	1.34 V	Valor RMS (raiz quadrada média) do sinal
Amplitude (Amp)	5.20 V	Amplitude média do sinal

Tabela 20 – Parâmetros de temporização medidos a 100 Hz — Teste de tempo de comando

- O valor de Vrms de 1.34 V (comparado a 400.93 mV no teste a 10 Hz) reflete o aumento do duty cycle, confirmando que mais tempo está sendo gasto em processamento.

4.3.6.6 Medição a 1 kHz

Para o envio de comandos peek a 1 kHz, o resultado mostra capacidade do protocolo DESTRA de processar um volume relativamente alto de comandos, aproximadamente 10 comandos a cada tick do programa embarcado. Os parâmetros medidos são apresentados na Tabela 21:

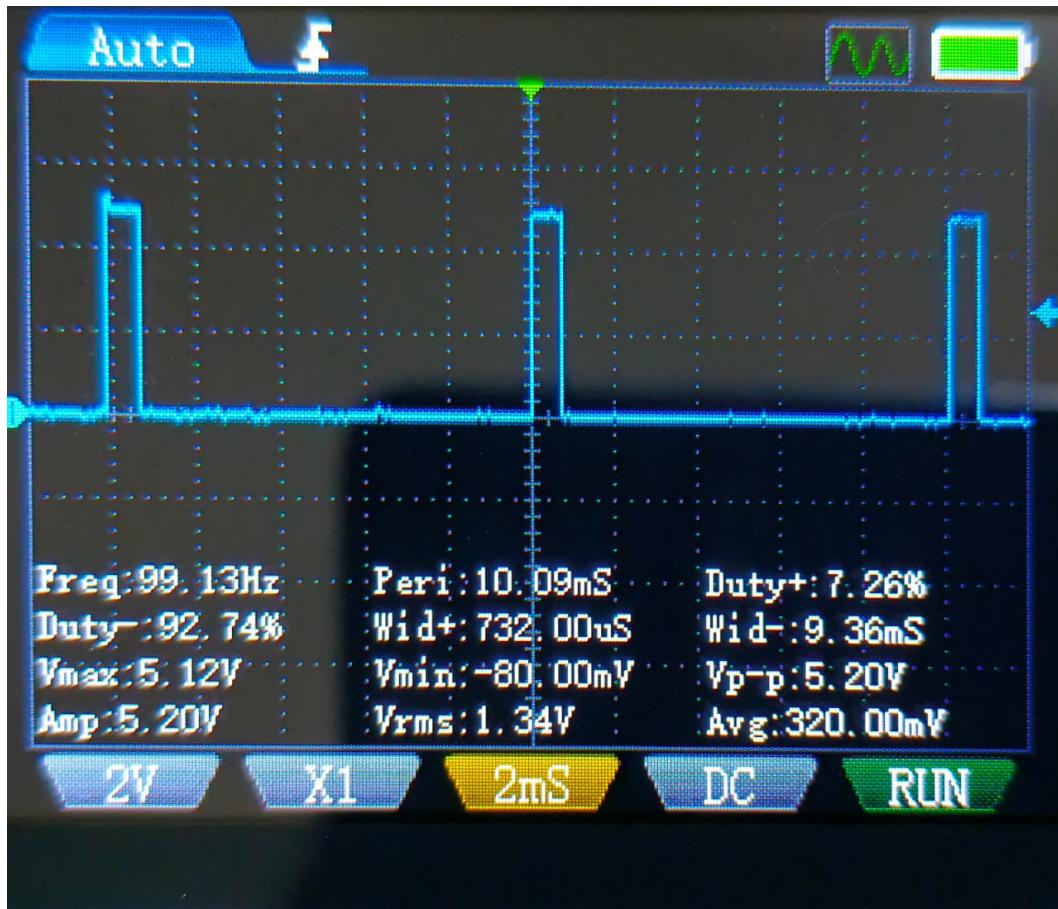


Figura 24 – Medição de tempo de comando a 1 kHz.

4.3.6.7 Análise dos Resultados a 1 kHz

Os resultados da medição a 1 kHz permitem as seguintes observações:

- A frequência medida permanece em 99.13 Hz, refletindo que o osciloscópio está capturando a frequência de agrupamento dos pulsos (10 comandos por ciclo de 10 ms = 1000 comandos/s). A forma de onda mostra pulsos próximos entre si, indicando processamento sequencial rápido.
- O período de 10.09 ms permanece praticamente idêntico às medições anteriores, confirmando que o loop principal mantém sua frequência de 100 Hz mesmo sob carga extrema de 1 kHz.
- A largura do pulso de 730 μs mantém-se estável em relação às medições anteriores, demonstrando que cada comando individual requer aproximadamente o mesmo tempo de processamento, independentemente da frequência de chegada.
- O duty cycle de 7.24% é praticamente idêntico ao da medição a 100 Hz, sugerindo que o sistema está processando os comandos em forma de rajadas (burst), com períodos de inatividade entre os grupos.

Parâmetro	Valor	Interpretação
Frequência (Freq)	99.13 Hz	Frequência de detecção dos pulsos de comando agrupados
Período (Peri)	10.09 ms	Intervalo entre grupos de comandos processados
Duty Cycle	92.76%	Percentual de tempo que o sinal permanece em nível alto
Duty+	7.24%	Percentual de tempo que os pulsos ficam ativos
Largura do Pulso (Widt)	730.00 µs	Duração do pulso de processamento (por comando individual)
Vmax	4.96 V	Tensão máxima do sinal capturado
Vmin	-80.00 mV	Tensão mínima do sinal (ruído de linha)
Vp-p (Pico a Pico)	5.04 V	Amplitude total da forma de onda
Vrms	1.26 V	Valor RMS (raiz quadrada média) do sinal
Amplitude (Amp)	4.96 V	Amplitude média do sinal

Tabela 21 – Parâmetros de temporização medidos a 1 kHz — Teste de tempo de comando

- A forma de onda em dente-de-serra observada na tela reflete exatamente esse comportamento: múltiplos pulsos de comando chegam quase simultaneamente (devido à comunicação serial em 115200 bps), e o firmware os processa sequencialmente dentro de cada ciclo de 10 ms.
- A amplitude de 5.04 V permanece dentro dos limites esperados, confirmando que não há degradação de sinal mesmo sob essa taxa extrema de processamento.

4.3.6.8 Síntese Comparativa das Três Medidas

A Tabela 22 apresenta uma comparação consolidada dos três cenários de teste:

Outra importante observação sobre os gráficos extraídos das medidas de tempo de comando é que a largura da forma de onda, que indica o tempo em que o pino PIN_BUSY ficou ativo (tempo de execução de um comando), permaneceu praticamente constante durante as três medidas, com valor aproximadamente igual ao das medidas instrumentadas do software embarcado, na faixa de 730 microsegundos. Este resultado corrobora a

Parâmetro	10 Hz	100 Hz	1 kHz
Frequência Medida	14.81 Hz	99.13 Hz	99.13 Hz
Período	43.34 ms	10.09 ms	10.09 ms
Largura Pulso	720 μ s	732 μ s	730 μ s
Duty Cycle	1.76%	7.26%	7.24%
Vp-p	5.00 V	5.20 V	5.04 V
Vrms	400.93 mV	1.34 V	1.26 V

Tabela 22 – Comparação dos parâmetros de temporização nas três taxas de envio de comando

precisão e confiabilidade da instrumentação interna do firmware, validando que as métricas coletadas refletem fielmente o comportamento temporal real do sistema.

4.3.6.9 Testes de Frame Rate

O pino PIN_FRAME_TOGGLE é alternado a cada ciclo do loop principal, permitindo medir a frequência de execução e o jitter temporal da aplicação. Esta medida visa detectar o frame rate da aplicação durante a execução dos comandos sob diferentes cargas.

4.3.6.10 Medição de Frame Rate a 10 Hz

A 10 Hz de envio de comandos, o tempo de frame é de 10 ms, refletindo um período baseado na execução do loop principal. Os parâmetros medidos são apresentados na Tabela 23:

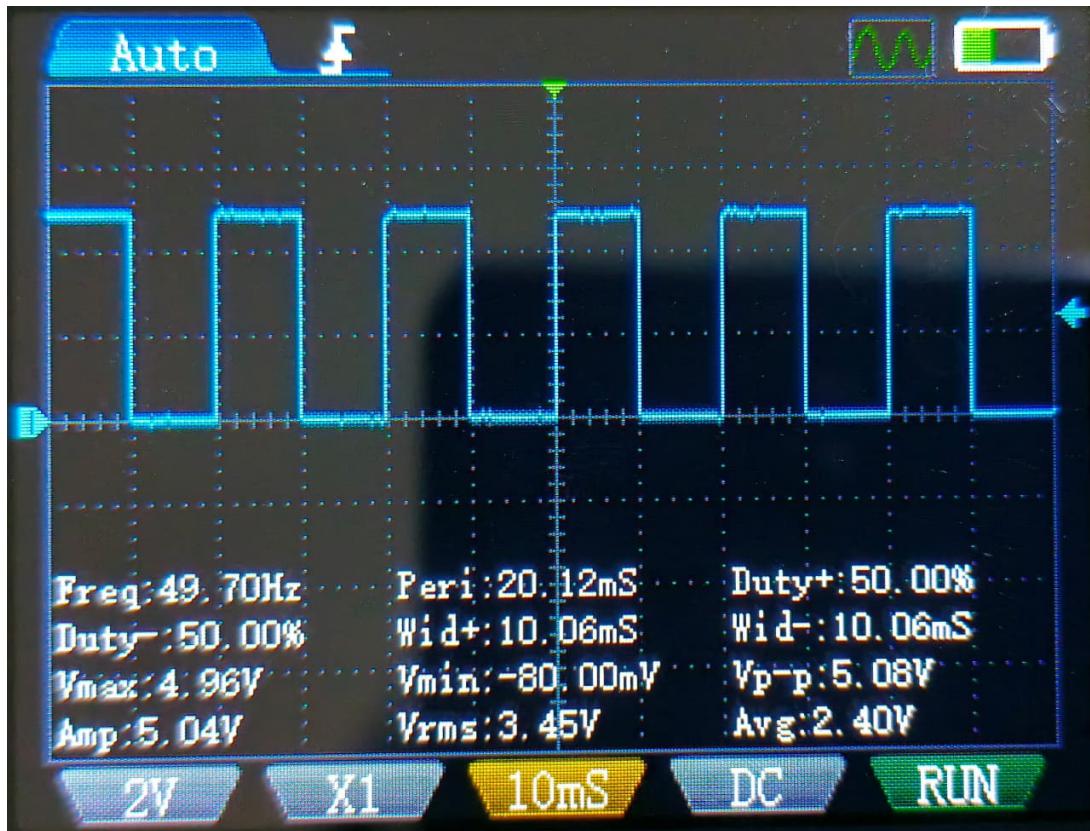


Figura 25 – Medição de frame rate a 10 Hz. A forma de onda quadrada mostra a alternância do pino PIN_FRAME_TOGGLE a cada ciclo do loop. O período medido de 20.18 ms corresponde a dois ciclos completos (uma subida e uma descida).

4.3.6.11 Medição de Frame Rate a 100 Hz

Para 100 Hz, obtemos praticamente a mesma medida, demonstrando que o loop principal mantém sua frequência de execução independentemente da taxa de envio de comandos. A Tabela 24 apresenta os parâmetros:

Parâmetro	Valor	Interpretação
Frequência (Freq)	49.55 Hz	Frequência de alternância do pino (metade da frequência do loop, pois cada ciclo tem uma subida e uma descida)
Período (Peri)	20.18 ms	Intervalo total entre duas transições completas (subida + descida)
Duty Cycle	50.02%	Percentual de tempo em nível alto (sinal simétrico, como esperado)
Largura de Pulso Alto (Widt+)	10.09 ms	Tempo de um ciclo do loop (metade do período total)
Largura de Pulso Baixo (Widt-)	10.09 ms	Tempo de um ciclo do loop (metade do período total)
Vmax	5.00 V	Tensão máxima do sinal
Vmin	-80.00 mV	Tensão mínima do sinal (ruído de linha)
Vp-p (Pico a Pico)	5.08 V	Amplitude total da forma de onda
Vrms	3.45 V	Valor RMS (raiz quadrada média) do sinal
Amplitude (Amp)	5.04 V	Amplitude média do sinal

Tabela 23 – Parâmetros de frame rate medidos a 10 Hz

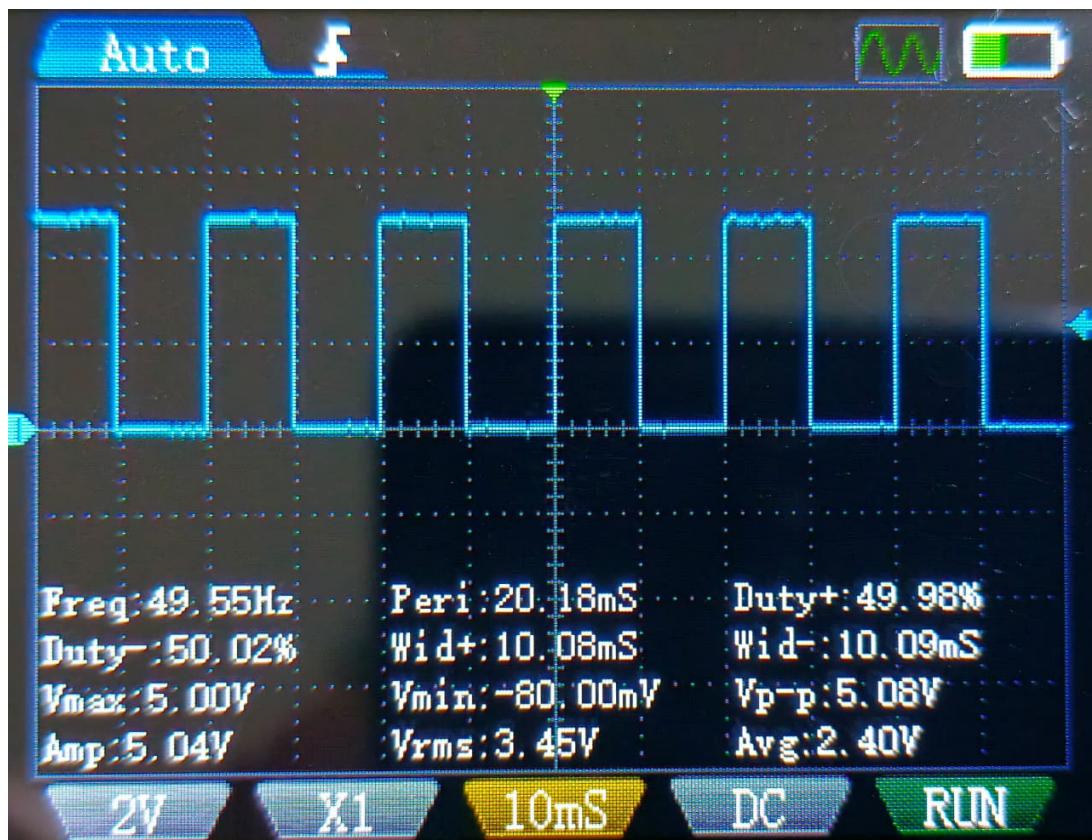


Figura 26 – Medição de frame rate a 100 Hz. A forma de onda mantém a mesma periodicidade observada a 10 Hz, confirmando que o loop principal não é afetado pela taxa de envio de comandos.

Parâmetro	Valor	Interpretação
Frequência (Freq)	49.55 Hz	Frequência de alternância do pino (mesma que a medição a 10 Hz)
Período (Peri)	20.18 ms	Intervalo total entre duas transições completas
Duty Cycle	50.00%	Percentual de tempo em nível alto (sinal perfeitamente simétrico)
Largura de Pulso Alto (Widt+)	10.09 ms	Tempo de um ciclo do loop
Largura de Pulso Baixo (Widt-)	10.09 ms	Tempo de um ciclo do loop
Vmax	4.96 V	Tensão máxima do sinal
Vmin	-80.00 mV	Tensão mínima do sinal
Vp-p (Pico a Pico)	5.04 V	Amplitude total da forma de onda
Vrms	3.45 V	Valor RMS do sinal
Amplitude (Amp)	5.04 V	Amplitude média do sinal

Tabela 24 – Parâmetros de frame rate medidos a 100 Hz

4.3.6.12 Medição de Frame Rate a 1 kHz

A 1 kHz, confirmamos novamente a estabilidade do frame rate, reforçando que o sistema mantém seu clock de 100 Hz mesmo sob carga extrema. A Tabela 25 apresenta os parâmetros:

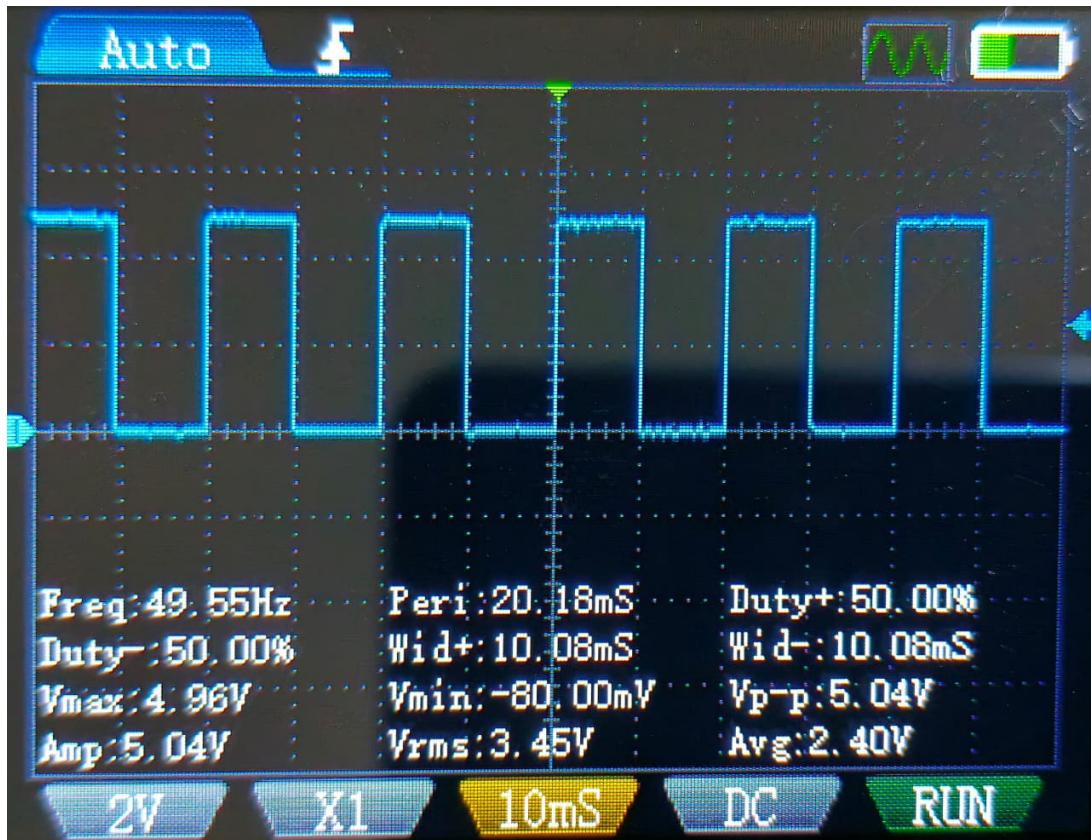


Figura 27 – Medição de frame rate a 1 kHz. A forma de onda permanece idêntica às medições anteriores, demonstrando que o protocolo DESTRA não introduz degradação temporal no loop principal, mesmo sob taxa extrema de comandos.

4.3.6.13 Análise Consolidada dos Testes de Frame Rate

As medições de frame rate complementam as análises de tempo de comando, fornecendo evidência independente da estabilidade temporal do sistema. A Tabela 26 apresenta uma comparação consolidada:

Parâmetro	Valor	Interpretação
Frequência (Freq)	49.70 Hz	Frequência de alternância do pino (praticamente idêntica)
Período (Peri)	20.12 ms	Intervalo total entre duas transições completas
Duty Cycle	50.00%	Percentual de tempo em nível alto (sinal perfeitamente simétrico)
Largura de Pulso Alto (Widt+)	10.06 ms	Tempo de um ciclo do loop
Largura de Pulso Baixo (Widt-)	10.06 ms	Tempo de um ciclo do loop
Vmax	4.96 V	Tensão máxima do sinal
Vmin	-80.00 mV	Tensão mínima do sinal
Vp-p (Pico a Pico)	5.08 V	Amplitude total da forma de onda
Vrms	3.45 V	Valor RMS do sinal
Amplitude (Amp)	5.04 V	Amplitude média do sinal

Tabela 25 – Parâmetros de frame rate medidos a 1 kHz

Parâmetro	10 Hz	100 Hz	1 kHz
Frequência	49.55 Hz	49.55 Hz	49.70 Hz
Período Total	20.18 ms	20.18 ms	20.12 ms
Período Loop	10.09 ms	10.09 ms	10.06 ms
Duty Cycle	50.02%	50.00%	50.00%
Vp-p	5.08 V	5.04 V	5.08 V

Tabela 26 – Comparação consolidada dos parâmetros de frame rate nas três taxas de comando

4.3.6.14 Observações Críticas

- A frequência medida de aproximadamente 49.6 Hz (em vez de 50 Hz teórico) reflete que o osciloscópio está capturando a frequência de alternância do pino, que é metade da frequência do loop ($100 \text{ Hz} \div 2 = 50 \text{ Hz}$). A pequena variação (0,3 Hz) é atribuível à precisão do cristal e ao jitter inerente.
- O período de aproximadamente 10.08 ms por ciclo confirma que o loop está sendo executado em 100 Hz, exatamente como programado. A variação máxima observada é de apenas 0,12 ms entre as medições (10.06 ms a 10.09 ms), representando um

jitter relativo de aproximadamente 1,2%, dentro do esperado para cristais de 16 MHz.

- O duty cycle de praticamente 50% indica que o sinal está perfeitamente simétrico (tempo alto \approx tempo baixo), confirmando que o `TOGGLE_FRAME()` está funcionando corretamente sem distorções.
- A amplitude de sinal de 5.04-5.08 V permanece estável em todas as três medições, confirmando que não há degradação de sinal mesmo sob carga extrema de 1 kHz.
- A extraordinária estabilidade do frame rate (variação máxima de 0,18 ms em 10 ms, ou 1,8%) valida o mecanismo de controle temporal implementado no firmware, que força o loop a executar a cada 10 ms através de `delayMicroseconds()`.

4.4 Análise Geral dos Testes com Osciloscópio

As medições realizadas com o osciloscópio confirmaram integralmente a periodicidade observada via instrumentação de software. A captura do sinal de alternância de frame apresentou uma frequência média de 99.4 Hz (medida direta do loop = 2×49.6 Hz de alternância), com variação máxima de apenas 1.8%, bem dentro do limite esperado para osciladores de cristal de 16 MHz em microcontroladores. Essa medição física corrobora a precisão temporal do sistema e valida completamente a confiabilidade da instrumentação interna do firmware. As variações de jitter registradas pelo software refletem flutuações reais no tempo de execução do loop, não artefatos de medição.

Adicionalmente, a consistência observada entre os três cenários de teste (10 Hz, 100 Hz e 1 kHz) demonstra que o protocolo DESTRA é completamente não-intrusivo quanto ao comportamento temporal do sistema. O loop principal continua executando em sua frequência programada de 100 Hz, independentemente da carga de comandos, confirmado a viabilidade da solução como ferramenta de depuração para sistemas embarcados críticos que exigem determinismo temporal rigoroso.

4.5 Conclusões dos Testes

Os testes realizados demonstram que o protocolo DESTRA funciona de forma estável e previsível em diferentes cenários de carga e frequência. A ausência total de erros de transmissão, combinada com latências consistentes e jitter baixo, evidencia a adequação da solução para aplicações embarcadas de prototipagem e depuração. Os resultados suportam a viabilidade da extensão do protocolo para ambientes críticos, desde que sejam implementados mecanismos adicionais de segurança e integridade conforme discutido no Capítulo 5.

5 CONCLUSÃO E TRABALHOS FUTUROS

5.1 Considerações Finais

A depuração e a verificação de sistemas embarcados críticos de tempo real representam esafios contínuos, especialmente quando há a necessidade de comprovar requisitos funcionais e temporais sem interferir diretamente na execução do software. Este trabalho apresentou o desenvolvimento e validação de um protocolo simples de peek/poke, voltado para observação e modificação controlada de variáveis internas, aplicado a um microcontrolador de baixo custo e com foco em apoio às atividades de teste e certificação conforme os princípios estabelecidos pela DO-178C.

A proposta demonstrou que é possível projetar uma ferramenta de depuração modular, rastreável e de baixo custo, com impacto mínimo sobre o comportamento temporal do sistema embarcado. O uso de comandos estruturados e medições internas de desempenho (via `CMD_GET_PERF_LOG`) permitiu validar a integridade da execução e quantificar a sobrecarga introduzida pela comunicação, confirmando a viabilidade da solução como mecanismo auxiliar em campanhas de teste e integração de software crítico.

5.2 Conclusões sobre o Protocolo Desenvolvido

Os resultados obtidos nos testes de desempenho e estresse mostraram que o protocolo mantém latência média inferior a 10 ms e jitter reduzido, mesmo sob alta carga de comandos, o que evidencia sua estabilidade temporal e robustez de comunicação. A análise embarcada revelou ainda que o tempo de processamento de comandos (`command_process_time`) manteve-se estável, em torno de 0,73 ms, com desvio padrão inferior a 0,003 ms, confirmando a previsibilidade exigida para sistemas de tempo real.

Além de cumprir seu objetivo principal — prover leitura e escrita em variáveis internas de forma segura e controlada —, o protocolo mostrou-se útil como instrumento de coleta de dados para verificação de requisitos temporais, permitindo correlacionar métricas internas (como frame rate e process time) com medições externas via osciloscópio. Essa correlação fornece uma camada adicional de rastreabilidade entre o comportamento observado e o comportamento esperado, o que é fundamental em atividades de verificação e validação sob normas de certificação.

5.3 Relevância para o Contexto de Certificação

No contexto da certificação de software embarcado, a rastreabilidade entre requisitos, código e evidências de teste é um elemento essencial. O protocolo desenvolvido contribui

diretamente para esse processo ao permitir a observação controlada de estados internos sem necessidade de instrumentação invasiva nem dependência de ferramentas proprietárias.

Conforme previsto nas normas (RTCA, 2011) e (RTCA, 2012), a qualificação de ferramentas de apoio depende de seu impacto sobre as atividades de verificação. Nesse sentido, a solução proposta se enquadra como uma ferramenta de suporte não intrusiva, podendo ser utilizada em fases de integração e ensaio funcional para coleta de evidências objetivas de execução de requisitos temporais, análise de desempenho e confirmação de margens de WCET (*Worst Case Execution Time*).

A principal contribuição está em demonstrar que é possível implementar, de forma independente e rastreável, uma ferramenta simples, de arquitetura aberta e documentação completa, capaz de apoiar testes de conformidade e verificação de requisitos em sistemas embarcados críticos, mesmo quando não se dispõe de ferramentas comerciais qualificadas. Isso torna o framework aplicável tanto em ambientes acadêmicos (para ensino de práticas de verificação embarcada) quanto em contextos industriais, como protótipo de infraestrutura de depuração aderente aos processos de certificação.

5.4 Limitações Observadas

Durante a execução dos testes e da validação funcional do protocolo, algumas limitações foram identificadas e são consideradas oportunidades de evolução.

5.4.1 Ausência de Operações Contínuas

A primeira limitação diz respeito à ausência de suporte a operações contínuas de leitura ou escrita (*continuous peek e poke*). Atualmente, cada requisição precisa ser iniciada de forma individual, o que limita o uso do protocolo em medições de alta frequência ou em observações contínuas de variáveis dinâmicas. A inclusão de um modo de operação contínua permitiria capturar séries temporais de forma mais eficiente, reduzindo a sobrecarga de comandos e ampliando o potencial de análise temporal.

5.4.2 Falta de Mecanismos de Integridade

Outra limitação está relacionada à robustez da camada de verificação de integridade de dados. Embora a comunicação se mostre estável nas condições atuais, o protocolo ainda não implementa mecanismos de verificação de redundância cíclica (CRC) ou checagem de erro robusta nos pacotes trocados. A ausência de um CRC dedicado pode, em cenários de ruído eletromagnético ou baud rates mais elevadas, introduzir risco de corrupção silenciosa de dados. A adoção de um CRC-16 ou CRC-32, associado à confirmação de sequência de comandos, representaria um avanço importante na confiabilidade e na segurança da comunicação.

5.4.3 Limitações de Arquitetura

Por fim, destaca-se que o framework, em sua forma atual, foi projetado para ambientes monothread e de comunicação serial simples, o que limita seu uso direto em plataformas com sistemas operacionais de tempo real (RTOS) ou múltiplas tarefas concorrentes. Apesar disso, sua estrutura modular permite fácil extensão para suportar filas de mensagens, buffers circulares e mecanismos de sincronização adequados para tais contextos.

5.5 Trabalhos Futuros

Dando continuidade ao trabalho, propõem-se as seguintes evoluções e aprimoramentos:

1. **Implementação de Operações Contínuas:** Suporte a *continuous peek* e *poke*, permitindo leitura e escrita contínua em variáveis selecionadas com temporização configurável.
2. **Mecanismos de Integridade de Dados:** Inclusão de verificação de redundância cíclica (CRC) nos pacotes de comunicação, garantindo detecção de erros de transmissão e validação completa dos dados trocados.
3. **Suporte a RTOS e Multitarefa:** Extensão do protocolo para ambientes com sistemas operacionais de tempo real, de forma a permitir sua aplicação em sistemas mais complexos e próximos de aplicações aeronáuticas reais.
4. **Integração com Ferramentas de Teste:** Integração com ferramentas de teste automatizado e bancos de requisitos, permitindo o registro automático de medições como evidências de verificação.
5. **Interface Gráfica Avançada:** Desenvolvimento de interface gráfica (GUI) aprimorada para facilitar a configuração, monitoramento e registro de sessões de depuração, com suporte a exportação de relatórios e visualização de históricos.
6. **Qualificação conforme DO-330:** Estudo de qualificação da ferramenta conforme os critérios do DO-330 (RTCA, 2012), avaliando seu enquadramento como ferramenta de apoio à verificação.
7. **Suporte a Interfaces de Alta Velocidade:** Adaptação do protocolo para interfaces mais rápidas (CAN, SPI, Ethernet), de modo a ampliar o alcance e a taxa de atualização para cenários de teste em tempo real.
8. **Validação em Plataformas Adicionais:** Portabilidade do protocolo para outras arquiteturas de microcontroladores (ARM Cortex-M, RISC-V) e sistemas operacionais embarcados.

9. **Documentação e Padronização:** Elaboração de especificação formal do protocolo e publicação de guidelines para implementação em diferentes plataformas, facilitando adoção e contribuições da comunidade.

5.6 Perspectivas Finais

As melhorias propostas fortaleceriam ainda mais o protocolo como instrumento de apoio aos testes de certificação, tornando-o um componente efetivo na geração de evidências rastreáveis e no monitoramento controlado de software crítico em execução.

A consolidação dessas evoluções permitiria ao framework DESTRA ser adotado não apenas como ferramenta de prototipagem e desenvolvimento, mas também como componente qualificado em processos formais de verificação e validação de sistemas embarcados críticos.

Espera-se que este trabalho estimule futuras pesquisas e desenvolvimentos na área de depuração e monitoramento de sistemas embarcados, contribuindo para uma indústria mais segura, rastreável e aderente aos mais rigorosos padrões de certificação internacional.

5.7 Resumo das Contribuições

As principais contribuições deste trabalho podem ser assim resumidas:

- Desenvolvimento de um protocolo simples, determinístico e modular para operações *peek* e *poke* em sistemas embarcados de baixo custo.
- Implementação completa do protocolo em Arduino UNO com instrumentação para coleta de métricas de desempenho.
- Desenvolvimento de ferramenta host em Python com interface gráfica intuitiva, suportando carregamento de símbolos ELF/DWARF e operações de monitoramento em tempo real.
- Validação experimental abrangente através de testes de latência, estresse e rajada, com medições correlacionadas por osciloscópio.
- Demonstração de viabilidade da solução para contextos de certificação de software crítico de segurança, conforme normas DO-178C e DO-330.
- Disponibilização de solução de código aberto e documentação completa para apoio a pesquisa e desenvolvimento acadêmico e industrial.

REFERÊNCIAS

- BOSCH, R. Can specification version 2.0. **Robert Bosch GmbH**, 1991.
- BURNS, A.; WELLINGS, A. **Real-Time Systems and Programming Languages**. 3. ed. Harlow: Addison-Wesley, 2001.
- CHRISTOF, N. **Debugging of Embedded Systems**. 2013. Dissertação (Mestrado) — University of Applied Sciences Technikum Wien, 2013.
- COMMITTEE, D. D. S. **DWARF Debugging Information Format Version 5**. [S.l.], 2017. Padrão DWARF versão 5. Disponível em: <http://dwarfstd.org/>.
- COMMITTEE, T. **Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification**. [S.l.], 1995. Especificação oficial do formato ELF. Disponível em: <https://refspecs.linuxbase.org/elf/elf.pdf>.
- DORFMAN, P. M. From obscurity to utility: Addr, peek, poke as data step programming tools. In: **Proceedings of the NorthEast SAS Users Group Conference (NESUG)**. [S.l.: s.n.], 2008.
- ELECTRICAL, I. of; ENGINEERS, E. **The Authoritative Dictionary of IEEE Standards Terms**. 7. ed. New York: IEEE Press, 2000. IEEE Std 100.
- FADIGA, S. **DESTRA: DEpurador de Sistemas em Tempo Real - Protocolo de Peek/Poke para Arduino**. 2025. Disponível em: <https://github.com/sfadiga/destra>. Repositório GitHub do projeto.
- HOPKINS, A. B. T.; McDONALD-MAIER, K. D. Debug support for complex systems on-chip: a review. **Elsevier Microprocessors and Microsystems**, 2006. Disponível em bases de engenharia eletrônica. Disponível em: https://www.researchgate.net/publication/3351788_Debug_support_for_complex_systems_on-chip_A_review.
- INSTITUTE OF ELECTRICAL AND ELECTRONICS ENGINEERS. **IEEE Standard for Ethernet**. 2018.
- ISO. **Road vehicles — Controller area network (CAN)**. 2015.
- KOOPMAN, P.; CHAKRAVARTY, T. Cyclic redundancy code (crc) polynomial selection for embedded networks. **International Conference on Dependable Systems and Networks (DSN)**, p. 145–154, 2004. Referência sobre seleção e robustez de CRCs.
- KOPETZ, H. **Real-Time Systems: Design Principles for Distributed Embedded Applications**. 2. ed. New York: Springer, 2011.
- LAPLANTE, P. A.; OVASKA, J. S. **Real-Time Systems Design and Analysis**. 4. ed. [S.l.: s.n.]: Wiley-IEEE Press, 2011.
- NASA. **NASA Software Safety Standard**. 2008. Critérios para software considerado safety-critical pela NASA. Disponível em: <https://standards.nasa.gov/standard/nasa/nasa-std-871913>.

RIERSON, L. **Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance**. Boca Raton: CRC Press, 2013. ISBN 978-1-4398-9296-8.

RTCA, I. **DO-178C: Software Considerations in Airborne Systems and Equipment Certification**. 2011. Norma de certificação de software aeroespacial.

RTCA, I. **DO-330: Software Tool Qualification Considerations**. 2012. Norma complementar sobre qualificação de ferramentas de software.

SCHNEIDER, S.; FRALEIGH, L. The ten secrets of embedded debugging. **Embedded.com**, 2004. Disponível em: [embedded.com](http://www.embedded.com).

SINGH, B. Turn antiquated peek and poke interfaces in embedded module to modern web APIs. **International Journal of Innovative Research and Technology**, v. 6, n. 12, 2020.

STALLINGS, W. **Data and Computer Communications**. 10. ed. Boston: Pearson, 2017.

TANENBAUM, A. S.; WETHERALL, D. J. **Computer Networks**. 5. ed. Boston: Pearson, 2011.

WILHELM, R. *et al.* The worst-case execution-time problem – overview of methods and survey of tools. **ACM Transactions on Embedded Computing Systems**, v. 7, n. 3, p. 1–53, 2008.

ZUBERI, K. A.; GODDYN, P.; GHALWASH, A. Debugging real-time systems. **Real-Time Systems Symposium**, 1999.

APÊNDICE A – REPOSITÓRIO DO PROJETO DESTRA

Devido à extensão do código-fonte desenvolvido ao longo deste trabalho — incluindo o firmware embarcado, a ferramenta host, os módulos de instrumentação e os scripts de teste — optou-se por não incluir o código completo como apêndice impresso neste documento.

A disponibilização integral de aproximadamente sessenta páginas de código comprometeria a legibilidade do trabalho e não agregaria valor técnico adicional à análise apresentada nos capítulos anteriores. Em consonância com práticas modernas de engenharia de software e pesquisa experimental, todo o código-fonte foi organizado e disponibilizado em um repositório público versionado.

O repositório oficial do projeto DESTRA está disponível em:

<https://github.com/sfadiga/destra/>

O repositório contém todos os artefatos necessários para reprodução, validação e extensão da solução proposta, incluindo o histórico completo de desenvolvimento.

A.0.1 Estrutura do Re却itório

A organização do repositório segue uma separação clara de responsabilidades entre os componentes do sistema, conforme descrito a seguir:

- `/arduino/` — Código-fonte do firmware embarcado responsável pela implementação do protocolo DESTRA no microcontrolador (desenvolvido em linguagem C do Arduino). Inclui:
 - `/destra_protocol_test/` — Firmware utilizado nos ensaios experimentais e testes de desempenho. Inclui:
 - * `/arduino/destra_protocol_test/destra_protocol_test.ino` — Implementação do protocolo para Arduino instrumentado para testes.
 - * `/arduino/destra_protocol_test/sample.ino` — Exemplo de integração.
 - `/sample/` — Firmware mínimo de exemplo para demonstração e validação funcional do protocolo. Inclui:
 - * `/arduino/sample/destra_protocol.ino` — Implementação do protocolo para Arduino.
 - * `/arduino/sample/sample.ino` — Exemplo de integração.

- */src/* — Código-fonte da aplicação *host*, desenvolvida em Python, responsável pela comunicação com o dispositivo embarcado, coleta de dados, instrumentação e interface gráfica utilizada nos experimentos. Inclui:
 - */src/data_dictionary.py* — Parser de arquivos ELF/DWARF.
 - */src/destra_ui.py* — Interface gráfica (PySide/Qt).
 - */src/destra.py* — Implementação do protocolo no host.
 - */src/logger_config.py* — Sistema de logging.
 - */src/requirements.txt* — Dependências Python.
- */tests/* — Scripts e rotinas de teste empregados na execução dos ensaios descritos neste trabalho, incluindo testes de latência, rajada e análise estatística. Dados legados de testes preliminares são mantidos apenas para referência histórica.
- */logs/* — Arquivos de registro gerados durante a execução dos testes, contendo medições brutas de tempo, eventos de comunicação e informações de depuração, utilizados posteriormente na análise experimental.
- */docs/* — Documentação complementar do projeto, incluindo os arquivos L^AT_EX deste trabalho acadêmico, figuras, dados experimentais consolidados e materiais de apoio à banca examinadora.

A.0.2 Reprodutibilidade

O repositório contém o arquivo `README.md` localizado no diretório raiz do projeto, este inclui instruções detalhadas para: compilação do firmware, execução da ferramenta host e realização dos testes experimentais apresentados neste trabalho, permitindo a completa reproduzibilidade dos resultados obtidos.