

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 específicos no que se refere à verificação e validação de requisitos, especialmente em ambientes de testes integrados, onde a capacidade de depuração é limitada. Ferramentas tradicionais de depuração, como probes e interfaces JTAG, não podem ser utilizadas durante a execução normal do software, exigindo abordagens alternativas. Este trabalho propõe o desenvolvimento de uma ferramenta de instrumentação de software embarcado, focada no monitoramento e injeção de valores em variáveis durante a execução dos sistemas. A solução será composta por três módulos principais: um protocolo simples de comunicação com o software embarcado, uma biblioteca em linguagem C para interpretação de comandos e acesso a variáveis internas, e uma interface gráfica no host para interação com o sistema embarcado. A ferramenta será projetada para testes integrados, como em ambientes Hardware-in-the-Loop (HIL), e visa auxiliar engenheiros na validação de sistemas críticos. Como produto final, será entregue uma prova de conceito (PoC) demonstrando o funcionamento da ferramenta, implementando operações de monitoramento (peek) e modificação (poke) de variáveis em tempo real.

Palavras-chave: Sistemas Embarcados, Instrumentação de Código, Injeção de valores, Monitoramento de Variáveis, Testes Integrados, Verificação de software, Software crítico, Software de tempo real.

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 specific challenges regarding requirement verification and validation, especially in integrated testing environments where debugging capabilities are limited. Traditional debugging tools, such as probes and JTAG interfaces, cannot be used during normal software execution, requiring alternative approaches. This work proposes the development of an embedded code instrumentation tool, focused on monitoring and injecting values into variables during system execution. The solution comprises three main modules: a simple communication protocol with embedded code, a C language library for command interpretation and internal variable access, and a graphical host interface for interaction with the embedded system. The tool is designed for integrated testing, such as in Hardware-in-the-Loop (HIL) environments, and aims to assist engineers in validating critical systems. As a final product, a proof of concept (PoC) will be delivered demonstrating the tool's functionality, implementing real-time variable monitoring (Peek) and modification (Poke) operations.

Keywords: Embedded Systems, Code Instrumentation, Value Injection, Variable Monitoring, Integrated Testing, Software Verification, Critical Software, Real-Time Software.

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	48
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.	50
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.	52
Figura 6 – Diagrama de sequência da operação de leitura do arquivo ELF/DWARF.	54
Figura 7 – Placa 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 – Gráfico de resultados do teste de latência - insira a figura aqui	68
Figura 10 – Gráfico de resultados do teste de estresse.	70
Figura 11 – Gráfico de resultados do teste de rajada.	72
Figura 12 – 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	74
Figura 13 – Medição de tempo de comando a 10 Hz.	75
Figura 14 – Medição de tempo de comando a 100 Hz.	77
Figura 15 – Medição de tempo de comando a 1 kHz.	79
Figura 16 – 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).	82

Figura 17 – 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.	84
Figura 18 – 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.	86
Figura 19 – Diagrama de Classes da Aplicação DESTRA UI	155

LISTA DE TABELAS

Tabela 1 – Estrutura do pacote do protocolo peek/poke	40
Tabela 2 – Variáveis da máquina de estados	45
Tabela 3 – Especificações técnicas do Arduino UNO	59
Tabela 4 – Especificações técnicas do osciloscópio digital FNIRSI® DSO-153 . . .	62
Tabela 5 – Cenários de teste da ferramenta host	63
Tabela 6 – Métricas de performance coletadas	65
Tabela 7 – Arquivos de relatório gerados automaticamente pelo script de testes .	68
Tabela 8 – Dados de performance do teste de latência (host)	69
Tabela 9 – Dados de performance embarcada (teste de latência)	69
Tabela 10 – Resumo da análise do teste de latência	70
Tabela 11 – Dados de performance do teste de estresse (host)	71
Tabela 12 – Dados de performance embarcada (teste de estresse)	71
Tabela 13 – Resumo da análise do teste de estresse	71
Tabela 14 – Dados de performance do teste de rajada (host)	72
Tabela 15 – Dados de performance embarcada (teste de rajada)	73
Tabela 16 – Resumo da análise do teste de rajada	73
Tabela 17 – Métricas medidas com osciloscópio	74
Tabela 18 – Parâmetros de temporização medidos a 10 Hz — Teste de tempo de comando	76
Tabela 19 – Parâmetros de temporização medidos a 100 Hz — Teste de tempo de comando	78
Tabela 20 – Parâmetros de temporização medidos a 1 kHz — Teste de tempo de comando	80
Tabela 21 – Comparação dos parâmetros de temporização nas três taxas de envio de comando	81
Tabela 22 – Parâmetros de frame rate medidos a 10 Hz	83
Tabela 23 – Parâmetros de frame rate medidos a 100 Hz	85
Tabela 24 – Parâmetros de frame rate medidos a 1 kHz	87
Tabela 25 – Comparação consolidada dos parâmetros de frame rate nas três taxas de comando	87
Tabela 26 – Resultados de throughput por frequencia	151
Tabela 27 – Resultados de teste de rajada	152

SUMÁRIO

1	INTRODUÇÃO	21
1.1	Contextualização	21
1.2	Motivação	23
1.3	Objetivos	24
1.3.1	Objetivo Geral	24
1.3.2	Objetivos Específicos	24
1.4	Justificativa	25
1.5	Estrutura do Trabalho	25
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	27
2.1.3	Aplicações de Software Embarcado em Setores de Alta Criticidade	28
2.1.3.1	Setor Aeroespacial	29
2.1.3.2	Setor Automotivo	29
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	30
2.2	Testes em Sistemas Embarcados	31
2.3	Protocolos de Comunicação para Depuração	32
2.4	Operações Peek, Poke e Telemetria	34
2.5	Ambientes de Testes	35
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	49

3.4.1	Arquitetura da Ferramenta Host	49
3.4.2	Procedimento Operacional	51
3.4.3	Formatos ELF e DWARF	52
3.5	Considerações sobre Extensibilidade	53
3.6	Resumo do Capítulo	55
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	69
4.3.4	Teste de Estresse	69
4.3.4.1	Análise dos resultados do teste de estresse	71
4.3.5	Teste de Rajada (Burst)	71
4.3.5.1	Análise dos resultados do teste de rajada	72
4.3.6	Resultados para o Cenário 2	73
4.3.6.1	Testes de Tempo de Comando	74
4.3.6.2	Medição a 10 Hz	74
4.3.6.3	Análise dos Resultados a 10 Hz	75
4.3.6.4	Medição a 100 Hz	76
4.3.6.5	Análise dos Resultados a 100 Hz	77
4.3.6.6	Medição a 1 kHz	78

4.3.6.7	Análise dos Resultados a 1 kHz	79
4.3.6.8	Síntese Comparativa das Três Medições	80
4.3.6.9	Testes de Frame Rate	81
4.3.6.10	Medição de Frame Rate a 10 Hz	81
4.3.6.11	Medição de Frame Rate a 100 Hz	82
4.3.6.12	Medição de Frame Rate a 1 kHz	85
4.3.6.13	Análise Consolidada dos Testes de Frame Rate	86
4.3.6.14	Observações Críticas	87
4.4	Análise Geral dos Testes com Osciloscópio	88
4.5	Conclusões dos Testes	88
5	CONCLUSÃO E TRABALHOS FUTUROS	89
5.1	Considerações Finais	89
5.2	Conclusões sobre o Protocolo Desenvolvido	89
5.3	Relevância para o Contexto de Certificação	89
5.4	Limitações Observadas	90
5.4.1	Ausência de Operações Contínuas	90
5.4.2	Falta de Mecanismos de Integridade	90
5.4.3	Limitações de Arquitetura	91
5.5	Trabalhos Futuros	91
5.6	Perspectivas Finais	92
5.7	Resumo das Contribuições	92
	REFERÊNCIAS	93
	APÊNDICE A – PROTOCOLO DESTRA ARDUINO	95
A.1	Implementação Base do Protocolo	95
A.1.1	Implementação Completa - destra_protocol.ino	95
A.2	Uso do Protocolo	101
A.3	Exemplo de Integração	102
A.4	Especificações Técnicas	102
A.4.1	Parâmetros de Comunicação	102
A.4.2	Faixa de Endereços	103
A.4.3	Códigos de Status	103
	APÊNDICE B – FERRAMENTA HOST DESTRA UI	105
B.1	Módulo de Protocolo - destra.py	105
B.1.1	Protocolo de Comunicacao	105
B.2	Interface Grafica - destra_ui.py	113
B.2.1	Implementacao da GUI	113

B.3	Analisador ELF - data_dictionary.py	119
B.3.1	Parser de Arquivos ELF e Extracao de Variaveis	119
APÊNDICE C – PROTOCOLO DESTRA INSTRUMENTADO . . . 131		
C.1	Implementação com Instrumentação de Performance	131
C.1.1	Código Completo Instrumentado - destra_protocol_test.ino	131
C.2	Funcionalidades de Instrumentação	138
C.2.1	Variáveis de Análise de Performance	138
C.2.2	Pinos de Debug para Osciloscópio	138
C.2.3	Buffer de Performance	139
C.2.4	Comando Adicional de Performance	139
C.3	Integração com Sistema de Teste	139
C.3.1	Loop Principal Instrumentado	139
C.3.2	Uso da Instrumentação	140
C.4	Especificações Técnicas da Instrumentação	140
C.4.1	Precisão Temporal	140
C.4.2	Buffer de Performance	140
C.4.3	Sinais de Debug	140
APÊNDICE D – TESTES DE PERFORMANCE E ANALISE . . . 141		
D.1	Metodologia de Teste	141
D.1.1	Configuracao do Ambiente de Teste	141
D.2	Script Performance Test Completo	141
D.2.1	Teste de Latencia	149
D.2.2	Teste de Estresse	150
D.2.3	Teste de Rajada	150
D.3	Analise de Resultados	151
D.3.1	Resultados de Latencia	151
D.3.2	Resultados de Estresse	151
D.3.3	Resultados de Rajada	152
D.4	Analise de Jitter com Osciloscopio	152
D.4.1	Configuracao da Medicao	152
D.4.2	Medicoes de Jitter	152
D.5	Conclusoes dos Testes	152
D.5.1	Performance do Sistema	152
D.5.2	Limitacoes Identificadas	153
D.5.3	Recomendacoes	153
APÊNDICE E – DIAGRAMA DE CLASSES DA APLICAÇÃO DES- TRA UI 155		

E.1	Visão Geral da Arquitetura	155
E.2	Diagrama de Classes	155
E.3	Descrição das Classes	155
E.3.1	DestraGUI	155
E.3.2	DestraProtocol	156
E.3.3	ElfDataDictionary	156
E.3.4	VariableInfo	156
E.3.5	DecodedTypes	156
E.4	Padrões de Design Utilizados	157
E.4.1	Singleton	157
E.4.2	Model-View-Controller (MVC)	157
E.4.3	Data Transfer Object (DTO)	157
E.4.4	Facade	157

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 do WCET, temos as análises de “modified condition/decision coverage” (MCDC), que são critérios de cobertura de testes estruturais do código. Este rigoroso teste verifica as seguintes condições:

1. Cada condição atômica (sub-expressão booleana dentro de uma decisão) foi avaliada como verdadeira e falsa pelo menos uma vez.
2. Cada decisão composta (por exemplo, `if (A && B)`) foi avaliada como verdadeira e falsa pelo menos uma vez.
3. Cada condição influencia independentemente o resultado da decisão, ou seja, é preciso projetar testes que mostrem que mudar apenas uma condição altera o resultado da decisão.

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?

O uso de ferramentas tradicionais de *debug* de software é amplamente conhecido e utilizado na indústria de software embarcado nos estágios iniciais do desenvolvimento do software, sendo muito útil 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. 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/poke”) ou via interfaces seriais — têm sido empregados por desenvolvedores para acelerar o processo de teste e validação 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.

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.

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/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 depuração.

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.

2.1.1 Características: tempo real, restrições de hardware e determinismo

Um software embarcado apresenta diversas características fundamentais, entre elas comportamento de *tempo real* e *determinismo*. É comum associar “tempo real” ao tempo cronológico natural (tempo de relógio). Entretanto, em sistemas embarcados, tempo real refere-se à capacidade do software de executar suas funções dentro de limites temporais estritamente definidos durante a fase de projeto. Ou seja, o sistema deve garantir que o tempo necessário para a execução de cada ciclo não ultrapasse o tempo máximo alocado para aquelas operações.

Cada sistema embarcado possui janelas temporais definidas com base em cálculos de desempenho, processamento de entradas e saídas e margens de segurança necessárias para garantir operação confiável.

Uma consequência direta desse aspecto é o conceito de *criticidade*, frequentemente associado a sistemas embarcados de tempo real. A criticidade, contudo, está mais relacionada ao sistema final e ao impacto de seu funcionamento sobre pessoas, bens ou o ambiente, do que ao software isoladamente — visto que o software, enquanto abstração, não é inherentemente perigoso.

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” (Rierson, 2013).

O *NASA Software Safety Standard* identifica um software como crítico de segurança quando ao menos um dos seguintes critérios é satisfeito (Rierson, 2013):

- 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, inconveniencias.
- **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, 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/poke tornam-se particularmente valiosas.

2.3 Protocolos de Comunicação para Depuração

A comunicação entre ferramentas de depuração e sistemas embarcados desempenha um papel essencial no processo de verificação e validação, sobretudo em ambientes nos quais o acesso físico aos sinais internos do software é limitado ou inexistente. Para que seja possível realizar operações de inspeção, monitoração ou injeção de valores — como nos mecanismos de peek/poke — é necessário o uso de protocolos de comunicação confiáveis, determinísticos e adequados às restrições do sistema embarcado.

Diversos meios físicos e protocolos são tradicionalmente empregados para esse fim, entre os quais se destacam UART, CAN e Ethernet. Cada um desses protocolos apresenta vantagens e limitações específicas, e normalmente sua escolha também está associada a escolha da arquitetura do sistema no qual o software irá executar, sendo assim uma decisão sistemica e mais abrangente, é comum o protocolo estar associado a uma restrição física imposta por decisões no nível de hardware, sejam elas derivadas de requisitos de custo ou requisitos funcionais ou até mesmo comerciais por imposições advindas da escolha de fornecedores. A característica comum a todos os protocolos escolhidos é que sigam o paradigma de comunicação serial.

- **UART** — A comunicação via UART é simples, de baixo custo e amplamente disponível, porém oferece taxas de transmissão relativamente limitadas e menor robustez contra erros.
- **CAN** — O barramento CAN fornece elevada imunidade a ruído e mecanismos nativos de priorização de mensagens, sendo bastante apropriado para ambientes automotivos e aeroespaciais, mas possui capacidade de carga útil reduzida por quadro.
- **Ethernet** — O protocolo Ethernet apresenta alta velocidade e baixa latência, porém exige uma camada extra que garanta seu uso em sistemas embarcados de tempo real/críticos o que pode introduzir sobrecarga e maior custo de implementação em sistemas embarcados críticos.

Independentemente do meio físico adotado, protocolos de depuração geralmente seguem um modelo de comunicação request/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

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 de depuração de sistemas 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). No ambiente moderno, apesar de evoluções significativas em ferramentas e protocolos, o conceito permanece central para arquiteturas de debug, sendo frequentemente encapsulado em interfaces padronizadas, APIs ou camadas de abstração (Singh, 2020).

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/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/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

reprodutível 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/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. Cada pacote é composto por:

As duas operações básicas são:

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

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 1 – Estrutura do pacote do protocolo peek/poke

- **POKE:** escrita de N bytes em um endereço definido.

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 em Listing A.1. 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`.

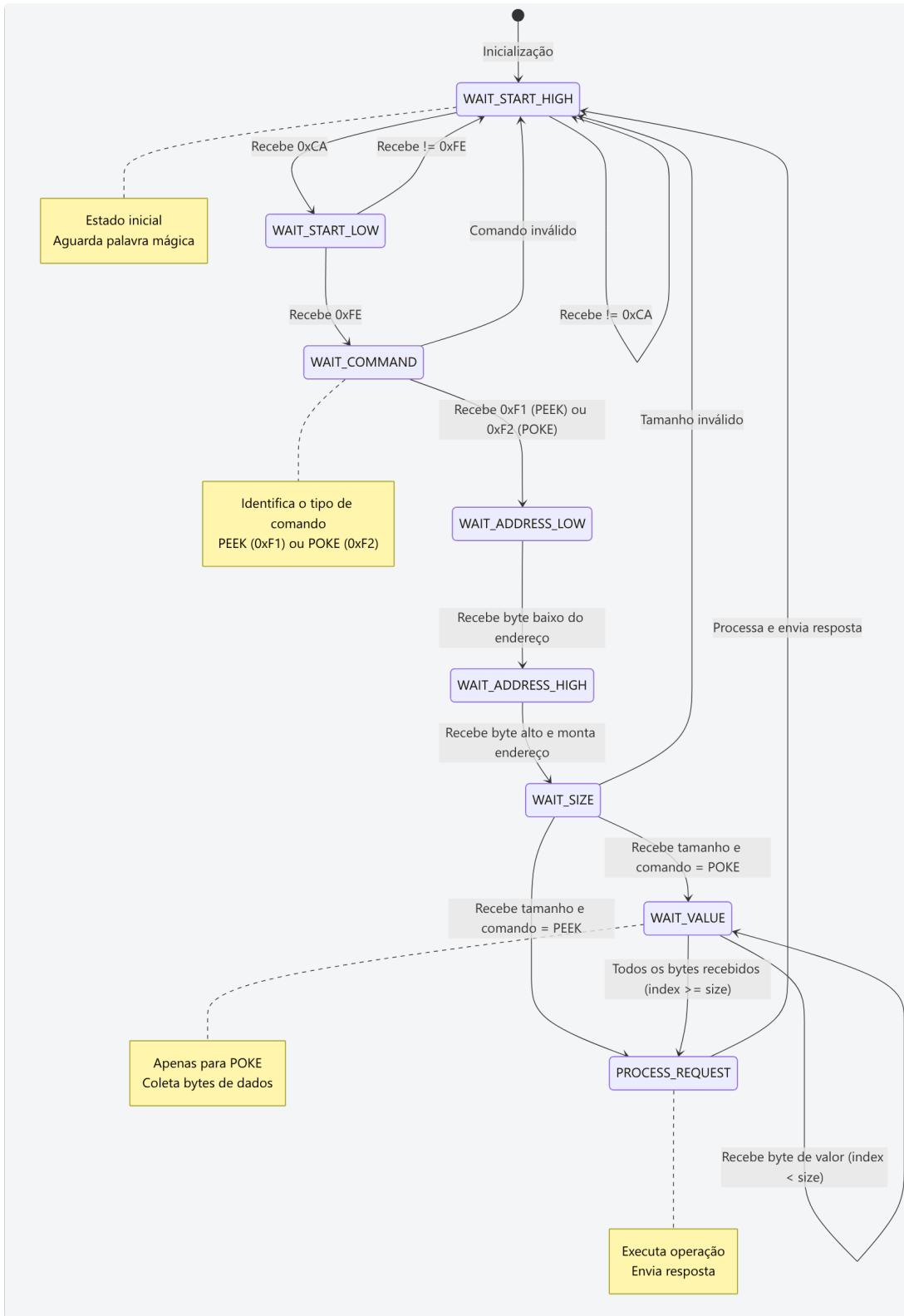


Figura 1 – Diagrama de estados para a implementação do protocolo DESTRA

Listing 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 2 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 2 – 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.

Listing 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 operacao  
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 memoria e enviar  
Para i de 0 ate destraSize - 1:  
    Ler byte da memoria 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.

Listing 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  
    destraAddress > 0x08FF:  
    Enviar STATUS_ADDRESS_RANGE_ERROR via serial  
    Retornar
```

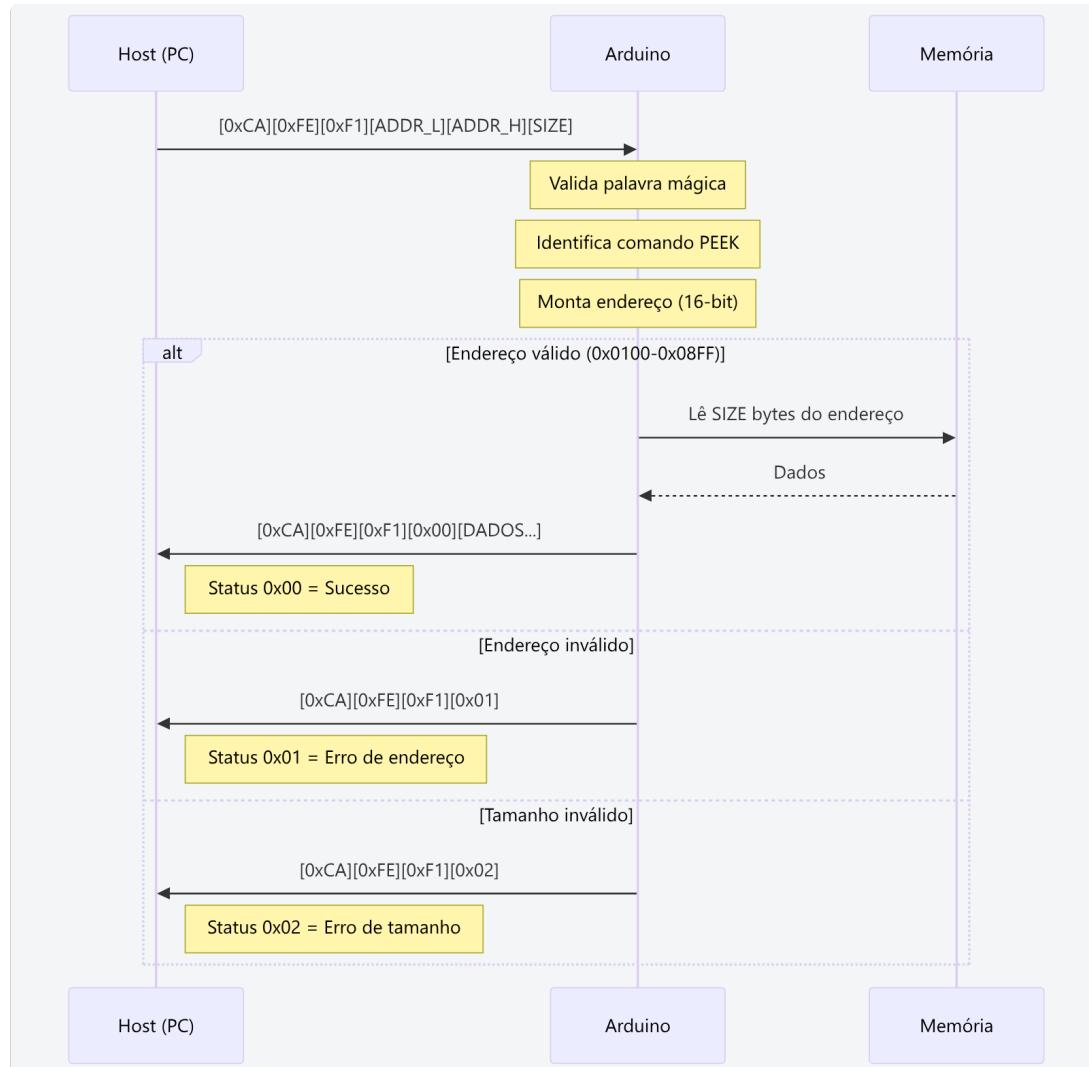


Figura 2 – Diagrama de sequencia de um comando peek a partir da ferramenta host

```

// 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.

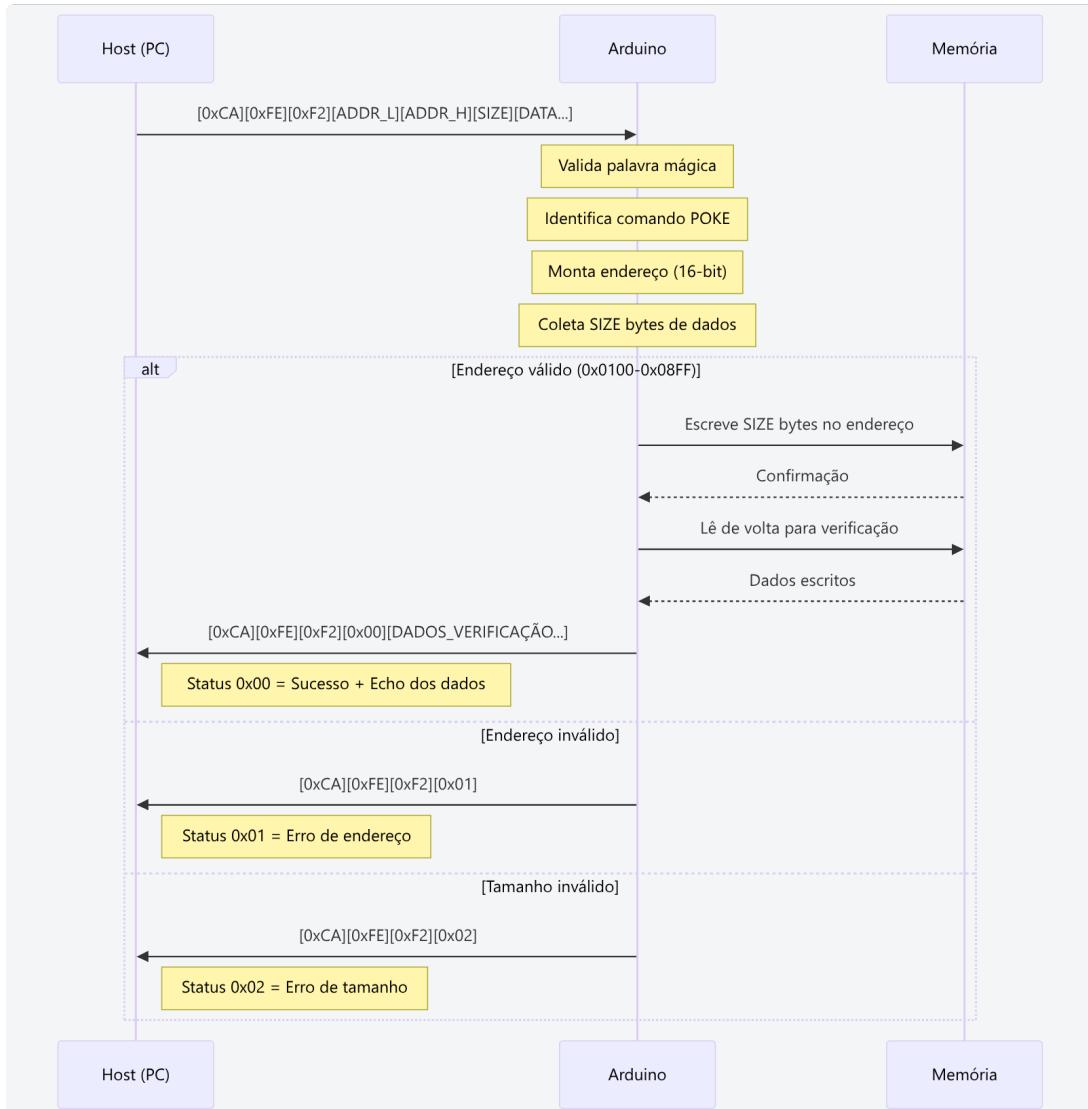


Figura 3 – Diagrama de sequencia de um comando poke a partir da ferramenta host

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*.
- 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:

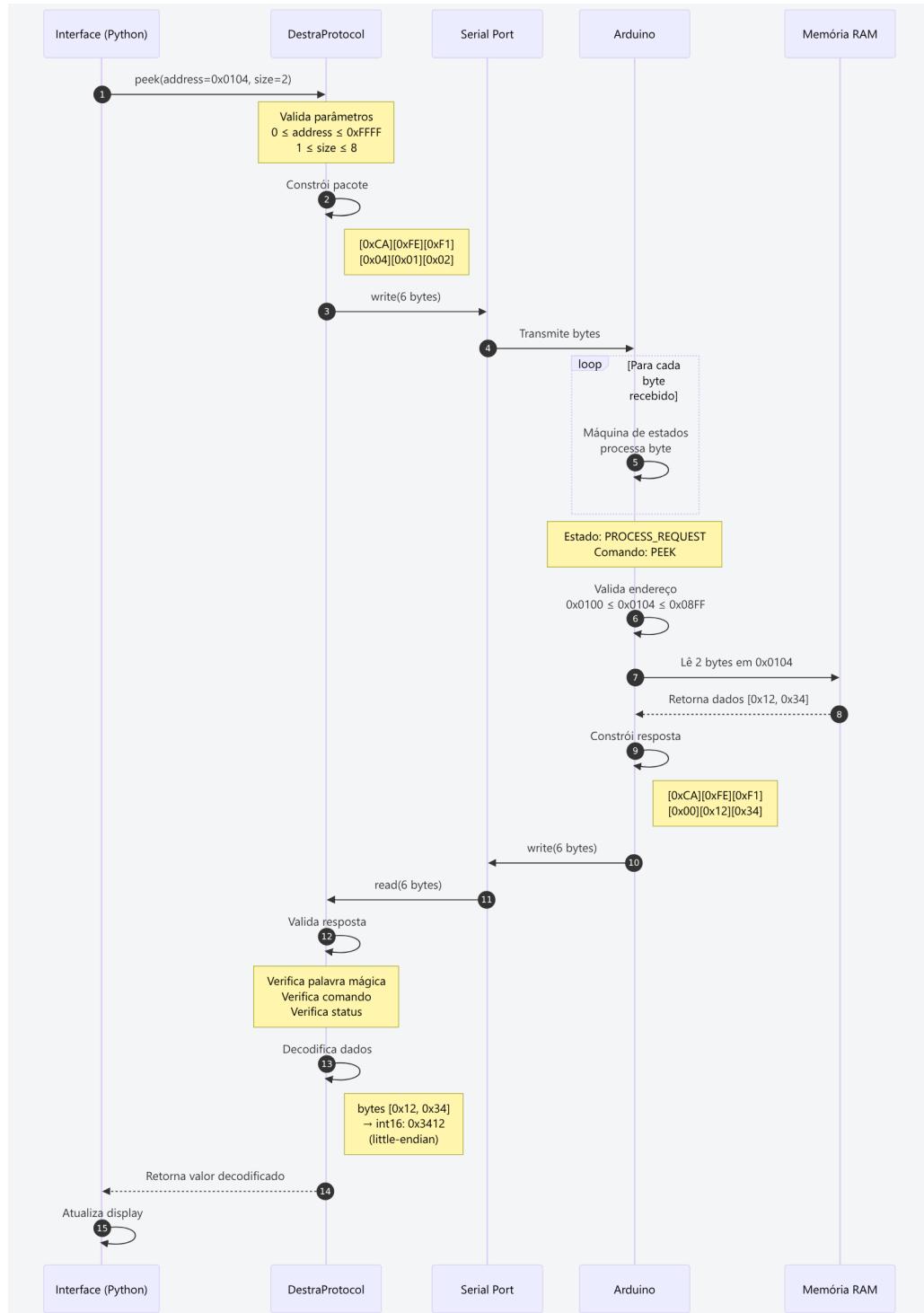


Figura 4 – Diagrama de sequência da operação `peek`. O fluxo ilustra as interações temporais entre a interface DESTRA UI, o módulo de protocolo Python, a comunicação serial UART, e a máquina de estados do sistema embarcado, demonstrando o ciclo completo request/response desde a requisição do usuário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

destra.py: Contém a implementação do protocolo proposto neste trabalho, análogo ao que foi apresentado anteriormente. A listagem deste script é apresentada em Listing B.1.

destra_ui.py: Implementa a interface gráfica da aplicação juntamente com suas ações. A listagem deste script é apresentada em Listing B.2.

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 em Listing B.3.

3.4.2 Procedimento Operacional

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

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 destra.
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.

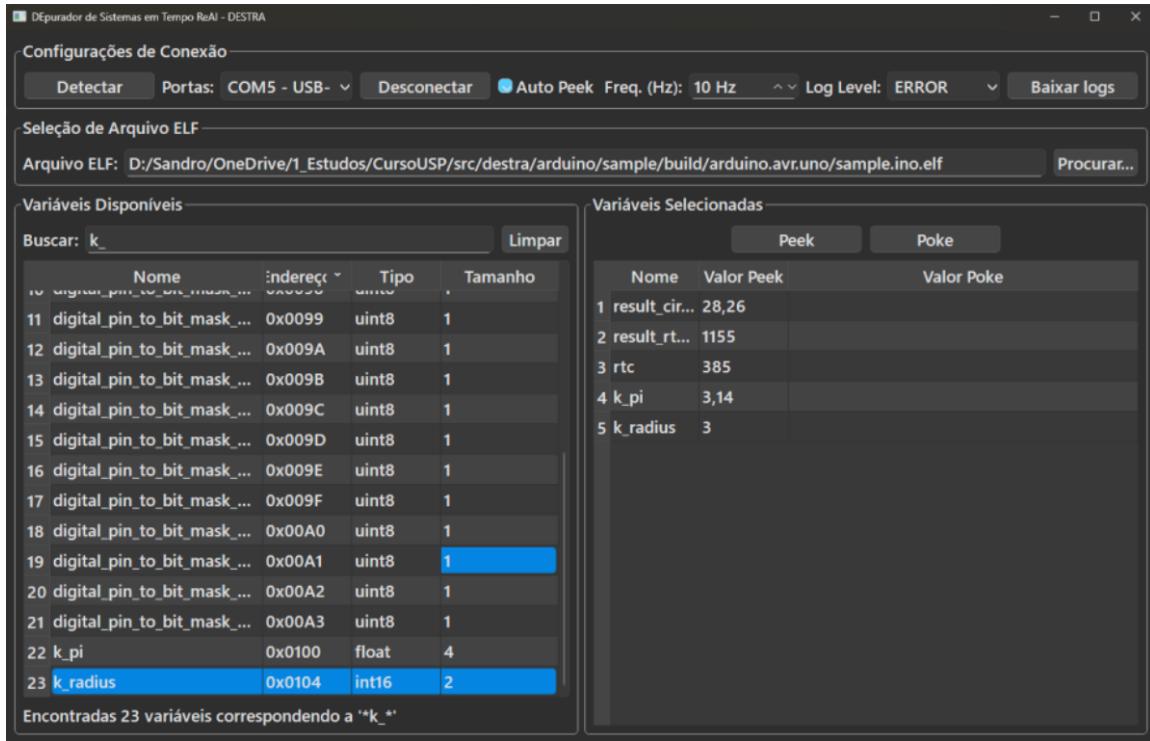


Figura 5 – Interface gráfica da ferramenta DESTRA UI. A tela apresenta os elementos principais para operação da ferramenta: (a) seleção de porta serial, (b) carregamento de arquivo ELF/DWARF, (c) painel de busca e seleção de variáveis, (d) tabela de monitoramento com operações de peek e poke, e (e) console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

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, 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).

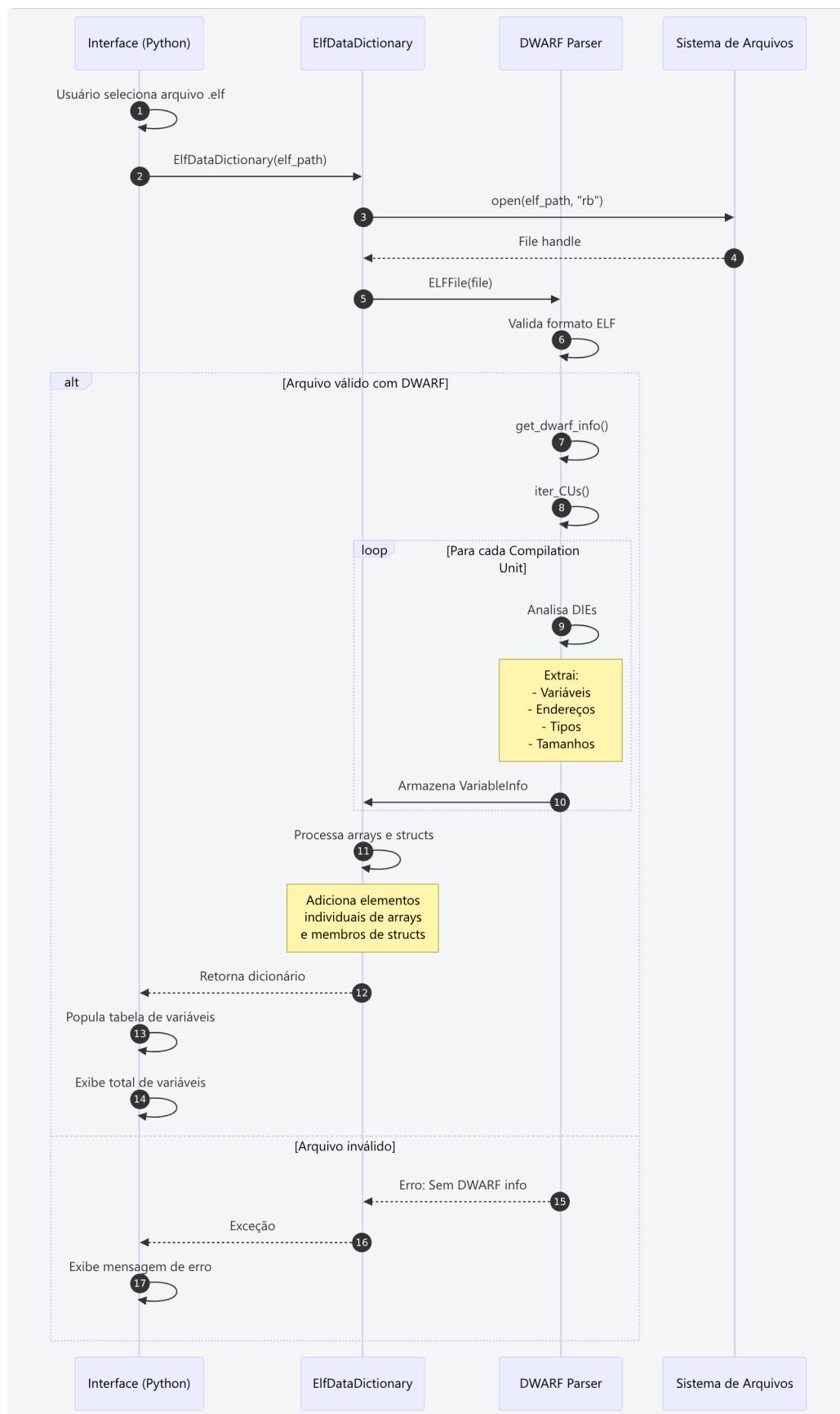


Figura 6 – Diagrama de sequência da operação de leitura do arquivo ELF/DWARF.

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/poke mostraram-se eficazes como 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.

A distinção clara entre software embarcado e ferramenta host garante separação de responsabilidades, facilitando a manutenção, evolução e adaptação para diferentes plataformas de hardware. Por fim, a integração com a ferramenta host em Python e a interface 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 consolidou-se como um recurso que une praticidade, eficiência e potencial de evolução, estabelecendo a base para futuras melhorias no contexto de depuração e instrumentação de sistemas críticos.

A implementação completa do protocolo DESTRA, incluindo o firmware embarcado, a especificação dos comandos, a ferramenta host e os casos de teste utilizados durante o desenvolvimento, está disponibilizada em um repositório público mantido pelo autor (Fadiga, 2025). Esse repositório contém todos os artefatos necessários para reprodução, validação e extensão da solução proposta, permitindo que pesquisadores e engenheiros possam inspecionar o código-fonte integral, acompanhar o histórico de desenvolvimento e utilizar a ferramenta como base para trabalhos futuros.

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.

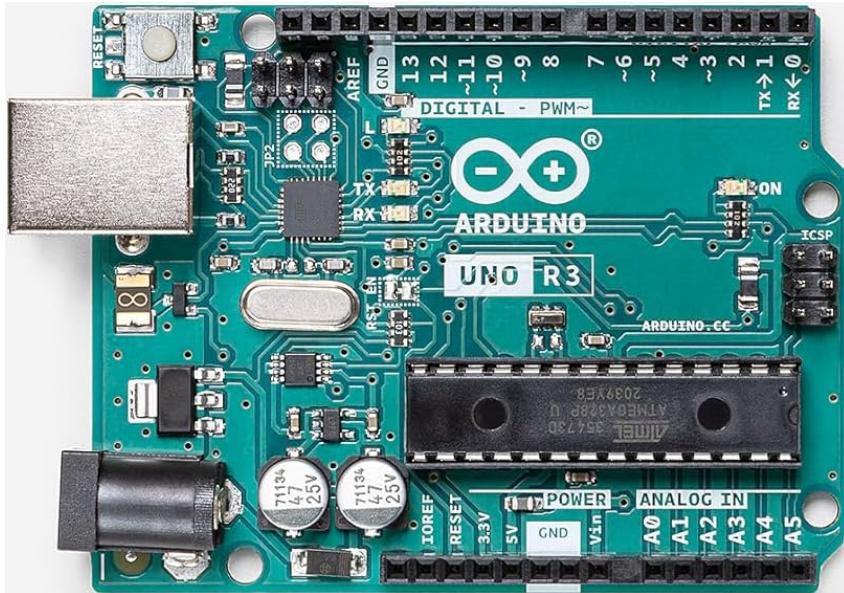


Figura 7 – Placa Arduino UNO. A imagem destaca os principais componentes: (a) microcontrolador ATmega328P, (b) conexão USB, (c) pinos digitais de I/O, (d) entradas analógicas, (e) botão de reset, (f) regulador de tensão, e (g) cristal oscilador de 16 MHz.

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 3 – 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:

Listing 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

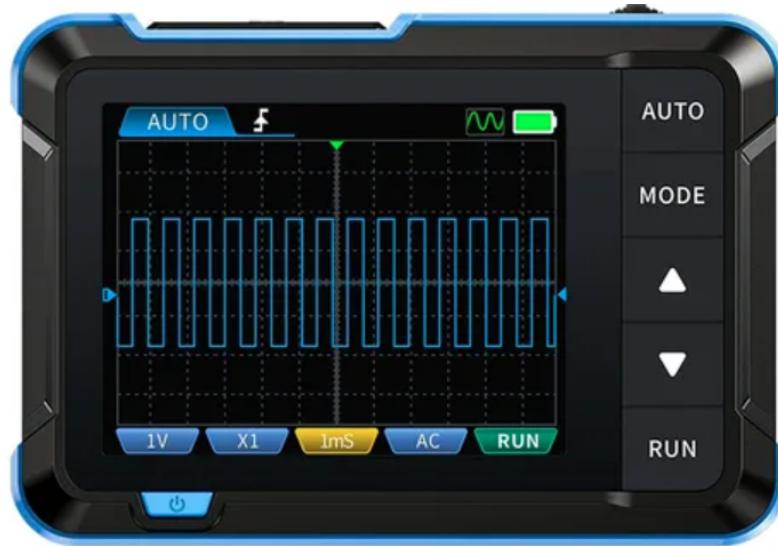


Figura 8 – Osciloscópio digital FNIRSI® DSO-153 2-em-1

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 4 – 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 em Listing D.1.

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 5 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 5 – 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:

Listing 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:

Listing 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 6 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 6 – 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:

Listing 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.

Listing 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 7 – 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

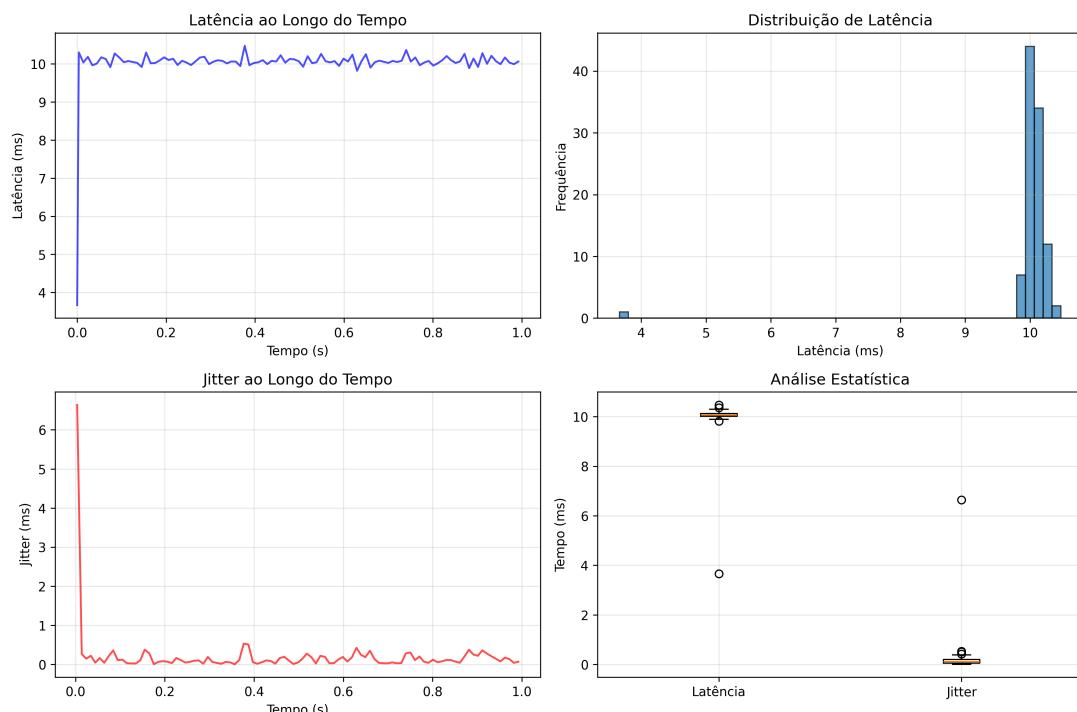


Figura 9 – Gráfico de resultados do teste de latência - insira a figura aqui

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

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 8 – Dados de performance do teste de latência (host)

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 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 9 – 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.

4.3.4 Teste de Estresse

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 10 – Resumo da análise do teste de latência

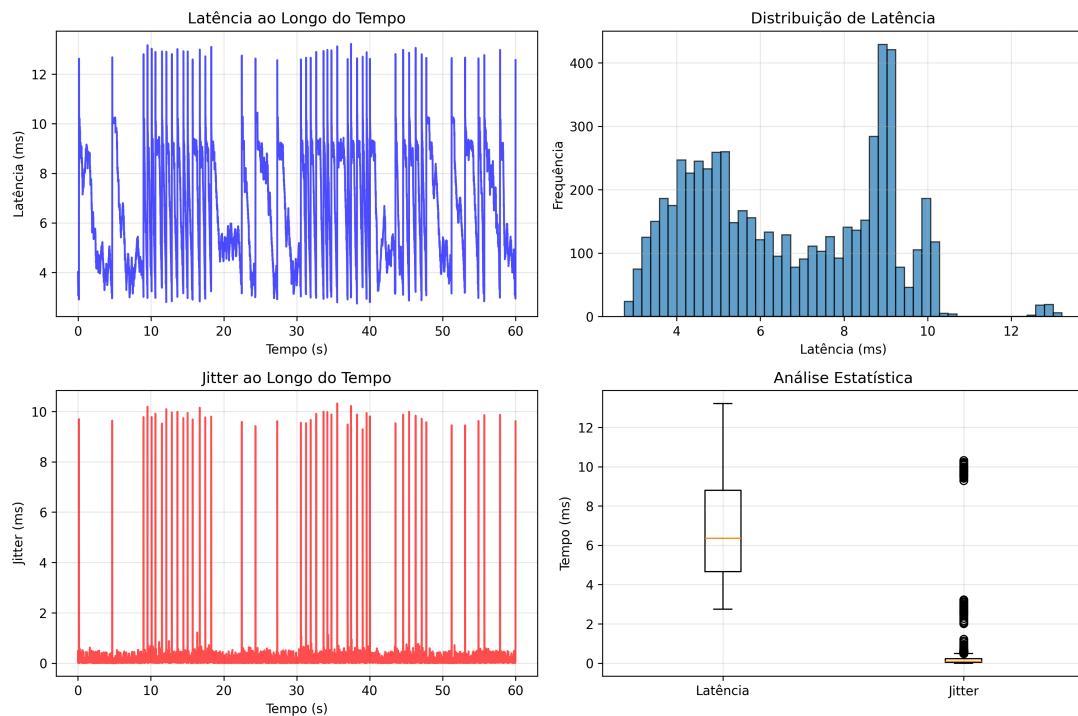


Figura 10 – Gráfico de resultados do teste de estresse.

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 11 – Dados de performance do teste de estresse (host)

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 12 – Dados de performance embarcada (teste de estresse)

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.

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 13 – Resumo da análise do teste de estresse

4.3.5 Teste de Rajada (Burst)

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

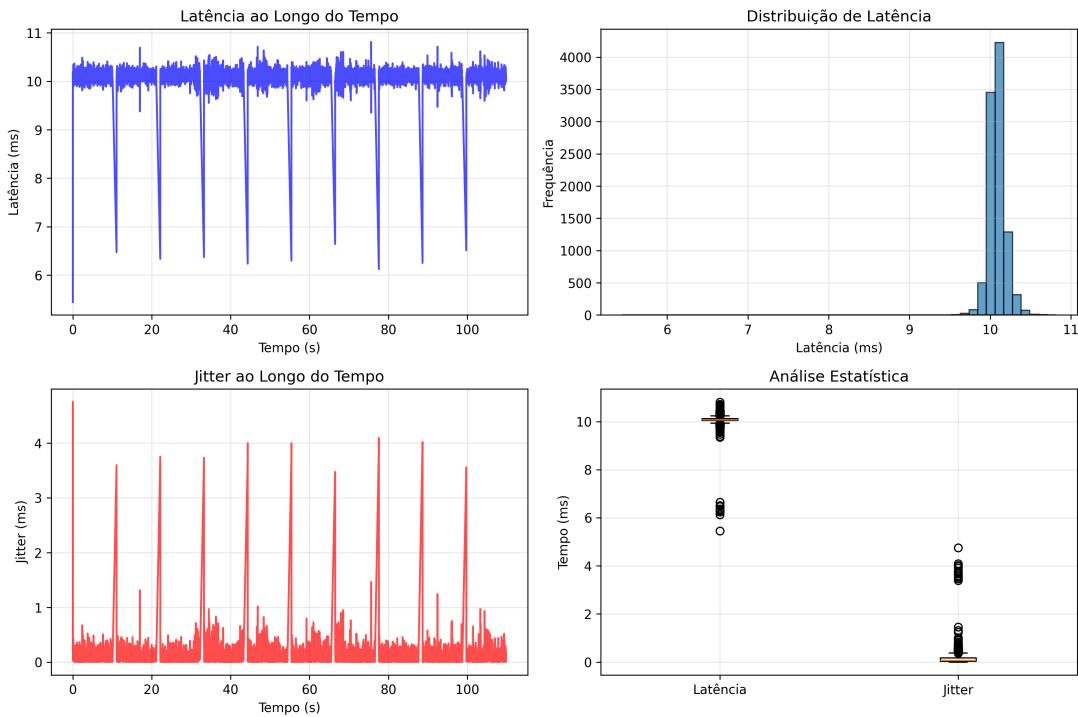


Figura 11 – Gráfico de resultados do teste de rajada.

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 14 – 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 15 – 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 sequência	Total — sem perdas detectadas

Tabela 16 – 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:

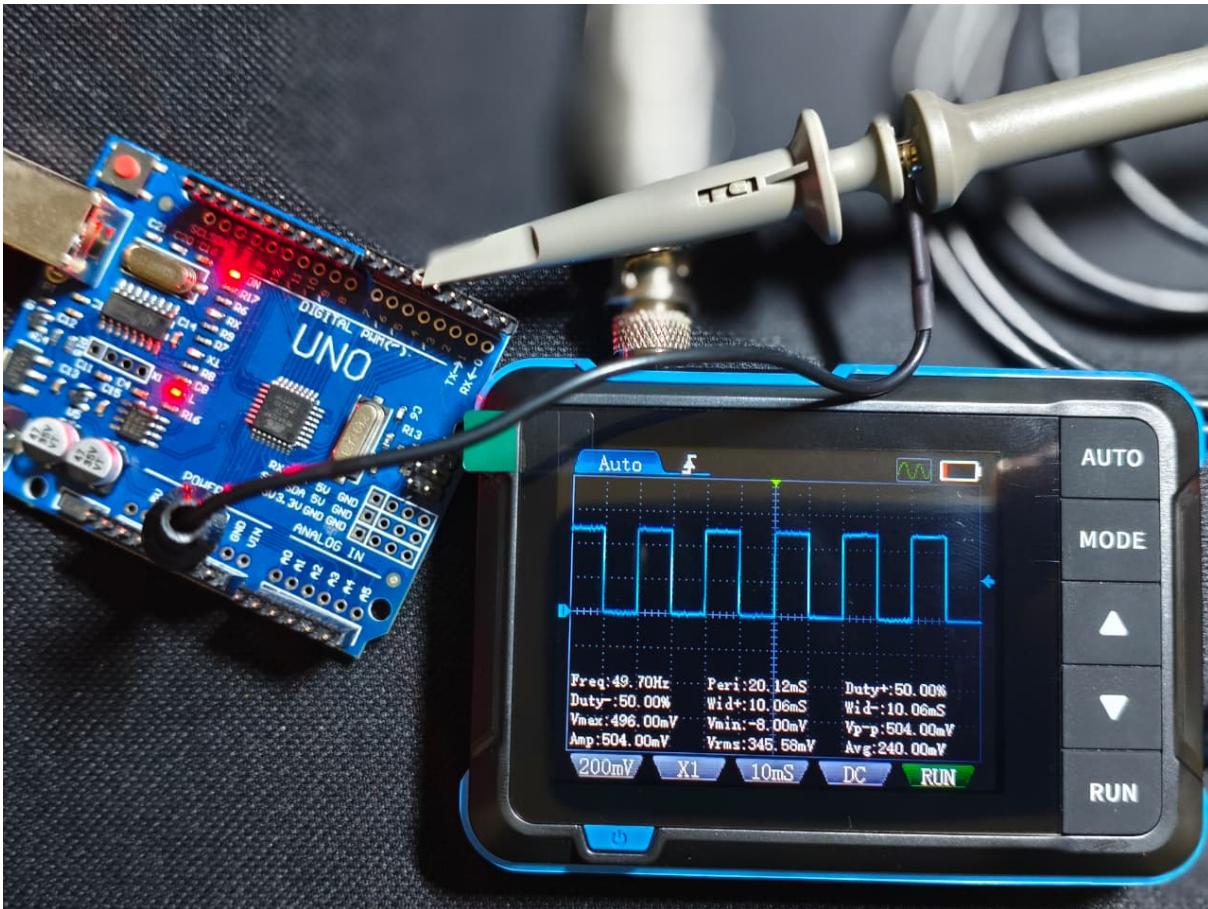


Figura 12 – Setup do osciloscópio conectado ao Arduino é mostrado durante medição de sinais do Arduino, com a sonda conectada a um dos pinos de debug. A tela colorida TFT exibe a forma de onda capturada em tempo real, permitindo análise de frequência, jitter e tempo de processamento de comandos

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 17 – 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 13 demonstra o resultado da medição com o osciloscópio para os comandos

a 10 Hz:

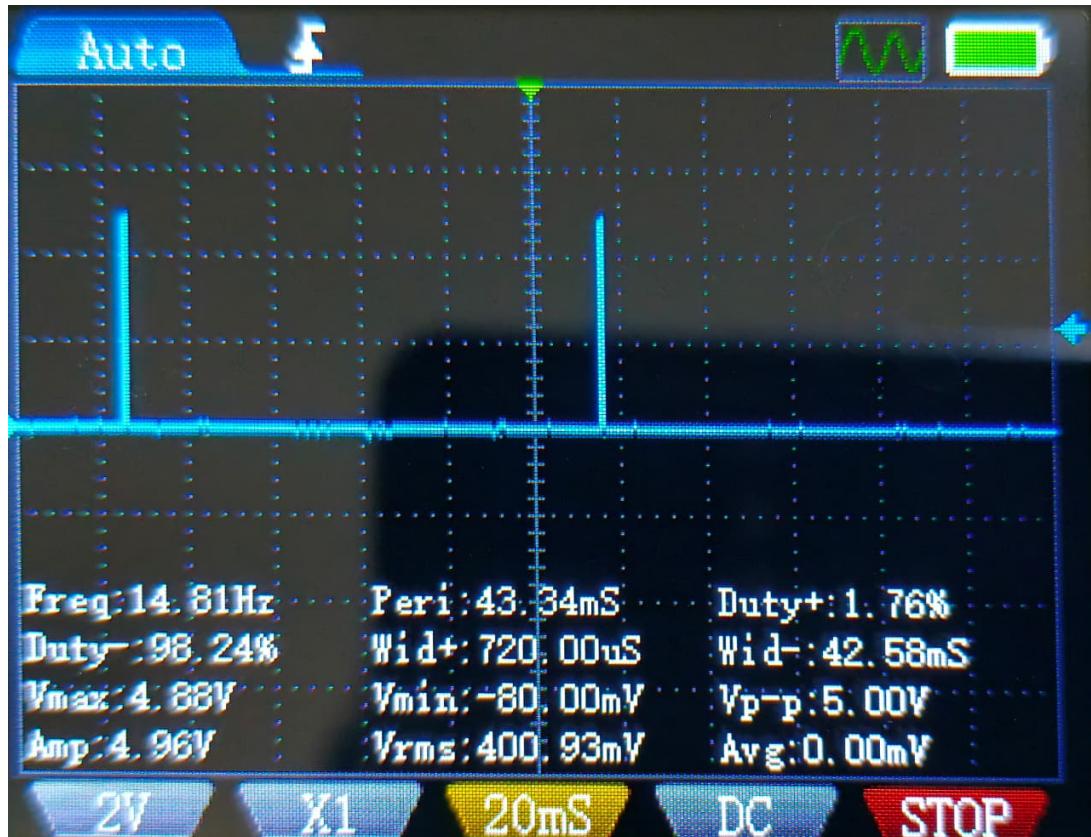


Figura 13 – Medição de tempo de comando a 10 Hz.

A Tabela 18 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 μ s (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 µs	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 18 – 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 19:

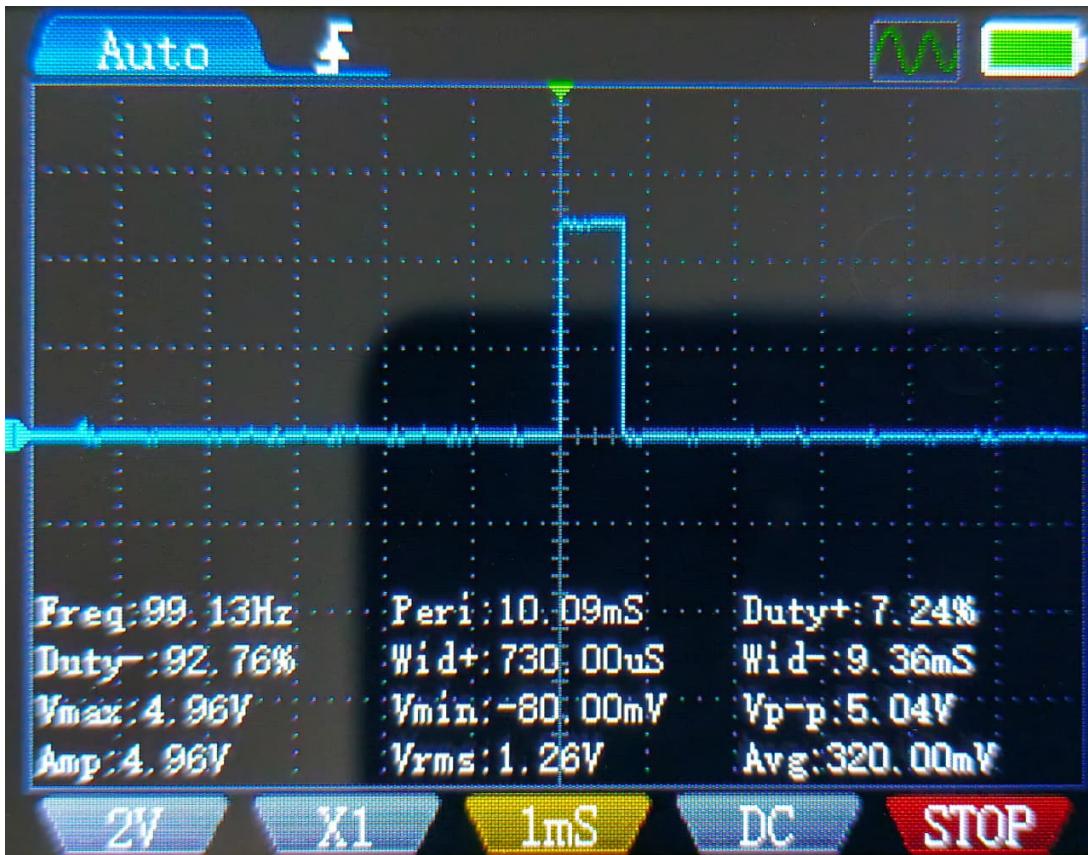


Figura 14 – Medição de tempo de comando a 100 Hz.

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 µs (0,732 ms) permanece praticamente constante em relação à medição a 10 Hz (720 µs), 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 19 – 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 20:

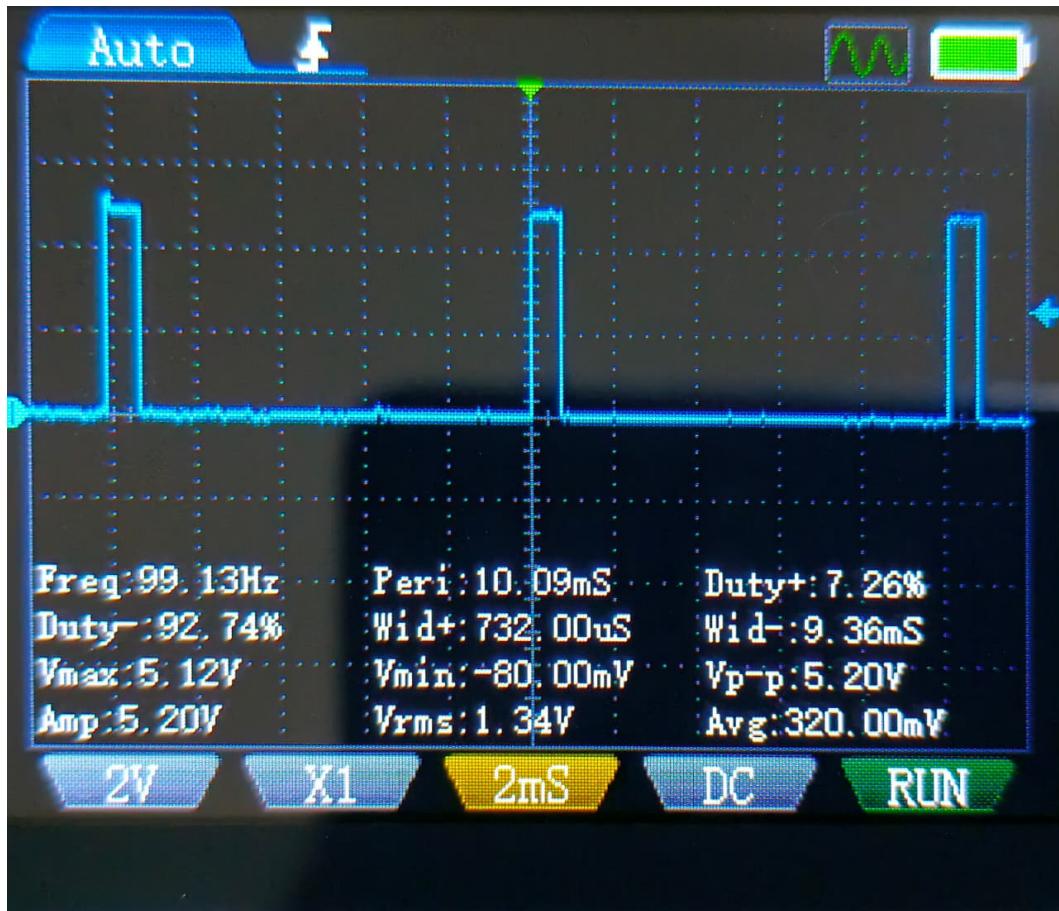


Figura 15 – Medição de tempo de comando a 1 kHz.

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 20 – 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 21 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 microssegundos. 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%
V _{p-p}	5.00 V	5.20 V	5.04 V
V _{rms}	400.93 mV	1.34 V	1.26 V

Tabela 21 – 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 22:

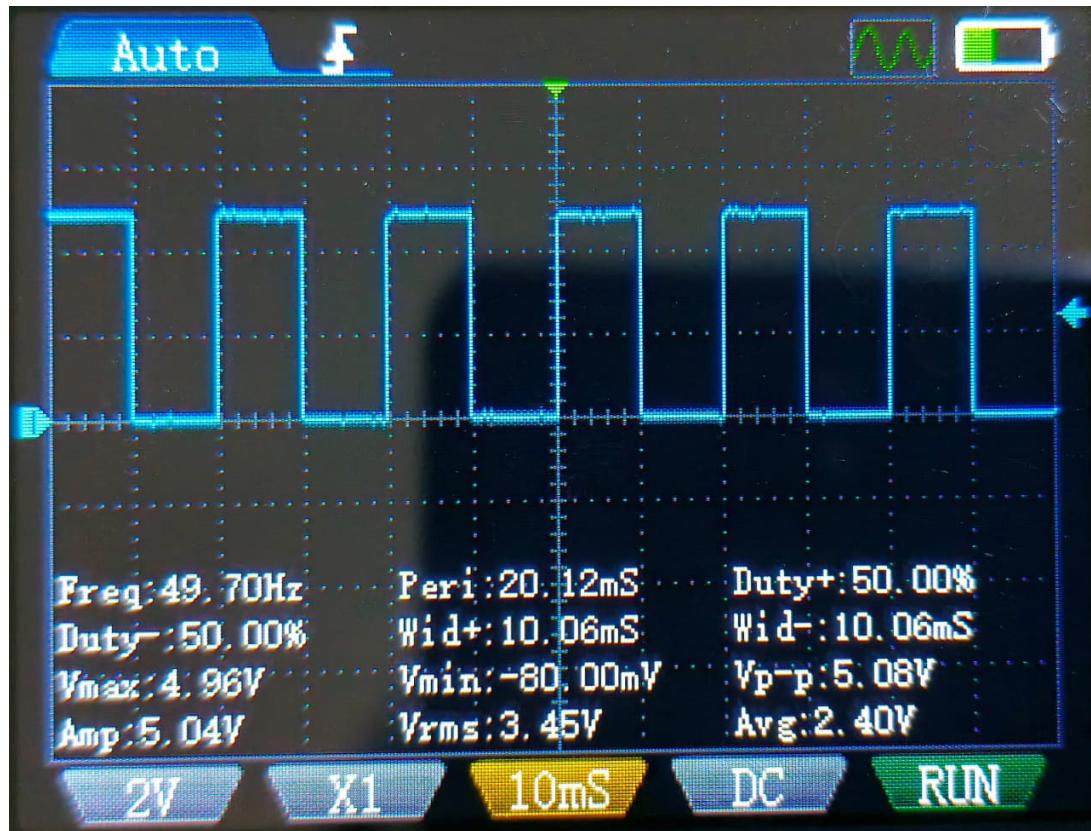


Figura 16 – Medição de frame rate a 10 Hz. A forma de onda quadrada mostra a alternância do pino PIN_FRAME_TOGGLE a cada ciclo do loop. O período medido de 20.18 ms corresponde a dois ciclos completos (uma subida e uma descida).

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 23 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 22 – Parâmetros de frame rate medidos a 10 Hz

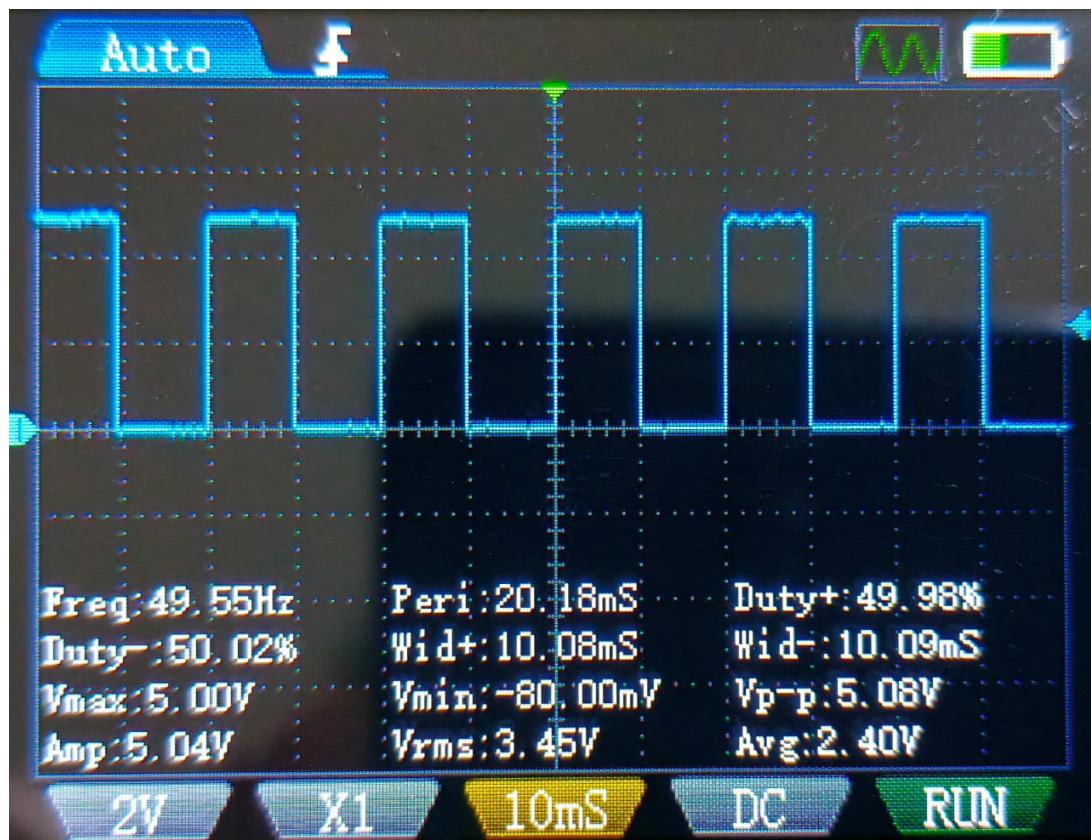


Figura 17 – Medição de frame rate a 100 Hz. A forma de onda mantém a mesma periodicidade observada a 10 Hz, confirmando que o loop principal não é afetado pela taxa de envio de comandos.

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 23 – 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 24 apresenta os parâmetros:

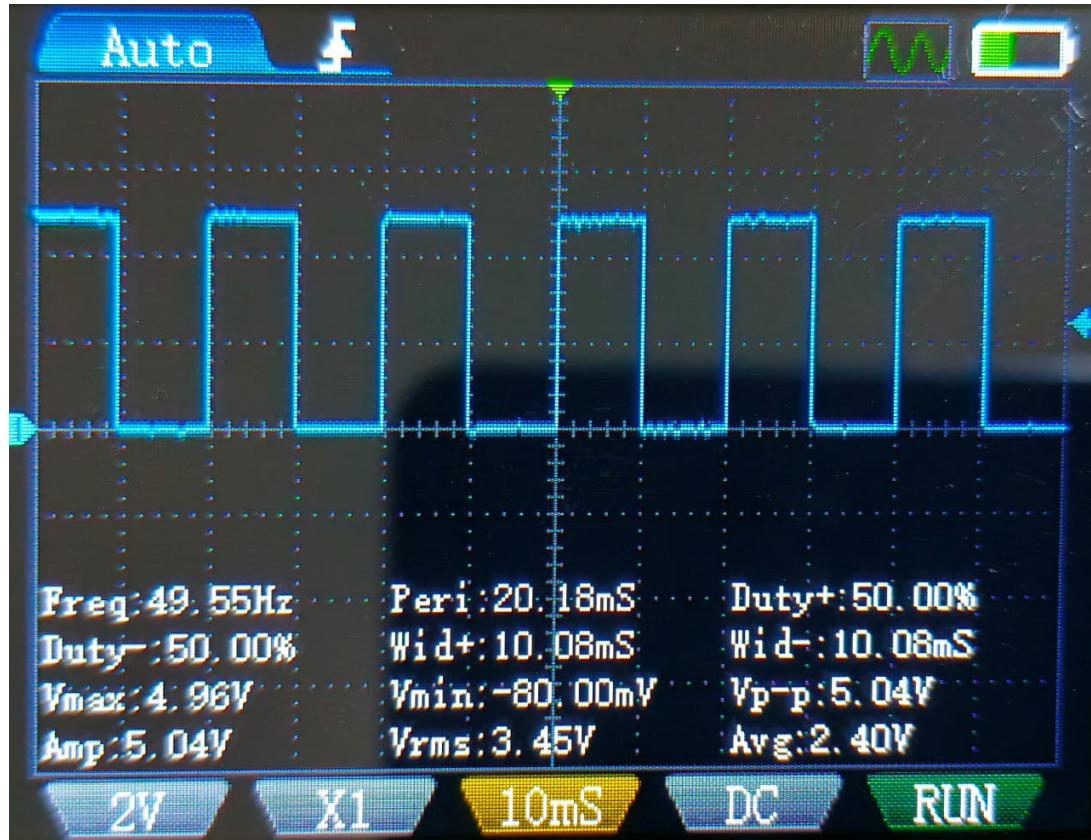


Figura 18 – Medição de frame rate a 1 kHz. A forma de onda permanece idêntica às medições anteriores, demonstrando que o protocolo DESTRA não introduz degradação temporal no loop principal, mesmo sob taxa extrema de comandos.

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 25 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 24 – 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 25 – 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 as 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, confirmando 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/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/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/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

- 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.
- 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.
- LAPLANTE, P. A. **Real-Time Systems Design and Analysis**. 4. ed. [S.l.: s.n.]: Wiley, 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.

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.

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 – PROTOCOLO DESTRA ARDUINO

A.1 Implementação Base do Protocolo

Este apêndice apresenta a implementação completa do protocolo DESTRA para Arduino, incluindo toda a lógica de comunicação serial para operações peek/poke.

A.1.1 Implementação Completa - `destra_protocol.ino`

Listing A.1 – Código completo do protocolo DESTRA para Arduino

```

1  /*
2   *
=====
3   * Arquivo: destra_protocol.ino
4   * Autor: Sandro Fadiga
5   * Instituicao: EESC - USP (Escola de Engenharia de Sao Carlos)
6   * Projeto: DESTRA - DEpurador de Sistemas em Tempo Real
7   * Data de Criacao: 09/01/2025
8   * Versao: 1.0
9   *
10  * Descricao:
11  *   Implementacao do protocolo DESTRA para Arduino.
12  *   Este arquivo contem toda a logica de comunicacao serial para
13  *   operacoes peek/poke, permitindo leitura e escrita de memoria
14  *   em tempo real para depuracao de sistemas embarcados.
15  *
16  * Protocolo:
17  *   - Palavras magicas: 0xCA 0xFE
18  *   - Comandos: PEEK (0xF1), POKE (0xF2)
19  *   - Enderecamento: 16 bits (0x0100 - 0x08FF para Arduino Uno)
20  *   - Tamanho de dados: 1-8 bytes por operacao
21  *   - Comunicacao: Serial 115200 baud, 8N1
22  *
23  * Funcionalidades:
24  *   - destraSetup(): Inicializa a comunicacao serial
25  *   - destraHandler(): Processa comandos recebidos (nao
26  *                   bloqueante)
27  *   - processPeekRequest(): Le dados da memoria
28  *   - processPokeRequest(): Escreve dados na memoria
29  *   - Maquina de estados para parsing de comandos

```

```
29 *      - Validacao de enderecos e tamanhos
30 *      - Echo de confirmacao para todos os bytes recebidos
31 *
32 * Formato dos Pacotes:
33 *     PEEK: [0xCA][0xFE][0xF1][ADDR_L][ADDR_H][SIZE]
34 *     POKE: [0xCA][0xFE][0xF2][ADDR_L][ADDR_H][SIZE][DATA...]
35 *
36 * Codigos de Status:
37 *     - 0x00: Sucesso
38 *     - 0x01: Erro de faixa de endereco
39 *     - 0x02: Erro de tamanho
40 *
41 * Requisitos:
42 *     - Arduino Uno ou compativel
43 *     - Memoria RAM acessivel: 0x0100 - 0x08FF
44 *
45 * Uso:
46 *     1. Incluir este arquivo no projeto Arduino
47 *     2. Chamar destraSetup() no setup()
48 *     3. Chamar destraHandler() no loop()
49 *
50 * Licenca: MIT
51 *

=====
52 */
53
54 #include <Arduino.h>
55
56 // Constantes de comunicacao serial
57 #define SERIAL_START_MARKER 0xCAFE
58 #define CMD_PEEK 0xF1
59 #define CMD_POKE 0xF2
60 #define STATUS_SUCCESS 0x00
61 #define STATUS_ADDRESS_RANGE_ERROR 0x01
62 #define STATUS_SIZE_ERROR 0x02
63 #define BAUD_RATE 115200
64 #define BUFFER_SIZE 64 // Buffer para dados recebidos
65
66 // Estados da maquina de estados serial
67 enum DestraState {
68     WAIT_START_HIGH ,
```

```
69     WAIT_START_LOW ,
70     WAIT_COMMAND ,
71     WAIT_ADDRESS_LOW ,
72     WAIT_ADDRESS_HIGH ,
73     WAIT_SIZE ,
74     WAIT_VALUE ,
75     PROCESS_REQUEST
76 };
77
78 // Variaveis de comunicacao serial
79 DestraState destraState = WAIT_START_HIGH;
80 uint8_t destraCommand = 0;
81 uint16_t destraAddress = 0; // Endereco de 16 bits
82 uint8_t destraSize = 0;
83 uint8_t addressLow = 0;
84 uint8_t addressHigh = 0;
85 uint8_t destraValueBuffer[8]; // Buffer para bytes de valor do
86 // POK
86 uint8_t destraValueIndex = 0; // Indice para buffer de valor
87
88 // Coloque-me no inicio do seu setup()
89 void destraSetup() {
90     // Inicializar comunicacao serial
91     Serial.begin(BAUD_RATE);
92     // Aguardar conexao da porta serial (necessario para placas USB
93     // nativas)
93     while (!Serial) {
94         ; // Aguardar conexao da porta serial
95     }
96     destraState = WAIT_START_HIGH;
97 }
98
99 // Coloque-me no inicio do seu loop()
100 void destraHandler() {
101     // Funcao para lidar com comunicacao serial destra (nao
102     // bloqueante)
102     while (Serial.available() > 0 && destraState != PROCESS_REQUEST
103         ) {
103         uint8_t inByte = Serial.read();
104
105         // Maquina de Estados do Pacote/Requisicao
106         switch (destraState) {
```

```
107     // Pacote PEEK/POKE comeca com 0xCAFE
108     case WAIT_START_HIGH:
109         if (inByte == 0xCA) {
110             int bytesAvailable = Serial.availableForWrite();
111             destraState = WAIT_START_LOW;
112         }
113         break;
114
115     case WAIT_START_LOW:
116         if (inByte == 0xFE) {
117             destraState = WAIT_COMMAND;
118         } else {
119             destraState = WAIT_START_HIGH;
120         }
121         break;
122
123     case WAIT_COMMAND:
124         destraCommand = inByte;
125         if (inByte == CMD_PEEK || inByte == CMD_POKE) {
126             destraState = WAIT_ADDRESS_LOW;
127         }
128         else {
129             destraState = WAIT_START_HIGH;
130         }
131         break;
132
133     case WAIT_ADDRESS_LOW:
134         addressLow = inByte;
135         destraState = WAIT_ADDRESS_HIGH;
136         break;
137
138     case WAIT_ADDRESS_HIGH:
139         addressHigh = inByte;
140         // Combinar para endereco de 16 bits (little endian)
141         destraAddress = addressLow | (addressHigh << 8);
142         destraState = WAIT_SIZE;
143         break;
144
145     case WAIT_SIZE:
146         destraSize = inByte;
147         if (destraCommand == CMD_PEEK) {
148             destraState = PROCESS_REQUEST;
```

```
149     }
150     else if (destraCommand == CMD_POKE) {
151         destraValueIndex = 0; // Resetar indice do buffer de
152         valor
153         destraState = WAIT_VALUE;
154     }
155     else {
156         destraState = WAIT_START_HIGH;
157     }
158     break;
159
160 case WAIT_VALUE:
161     // Verificar se o indice esta dentro dos limites do
162     // buffer
163     if (destraValueIndex < 8 && destraValueIndex < destraSize
164         ) {
165         // Armazenar o byte de valor
166         destraValueBuffer[destraValueIndex] = inByte;
167         destraValueIndex++;
168     }
169     // Verificar se recebemos todos os bytes de valor
170     if (destraValueIndex >= destraSize) {
171         destraState = PROCESS_REQUEST;
172     }
173     // Caso contrario, permanecer em WAIT_VALUE para coletar
174     // mais bytes
175     break;
176 }
177
178 // Processar a requisicao se tivermos uma mensagem completa
179 if (destraState == PROCESS_REQUEST) {
180     if (destraCommand == CMD_PEEK) {
181         processPeekRequest();
182     } else if (destraCommand == CMD_POKE) {
183         processPokeRequest();
184     }
185 }
```

```
186
187
188 // Funcao para processar requisicao peek e enviar resposta
189 void process.PeekRequest() {
190     // Enviar cabecalho da resposta
191     Serial.write(0xCA);
192     Serial.write(0xFE);
193     Serial.write(CMD_PEEK);
194
195     // Validar faixa de endereco (verificacao de segurança opcional
196     // )
197     if (destraAddress < 0x100 || destraAddress > 0x8FF) { // Faixa
198         de RAM do Arduino Uno
199         Serial.write(STATUS_ADDRESS_RANGE_ERROR);
200         return;
201     }
202
203     // Validar tamanho
204     if (destraSize == 0 || destraSize > 8) {
205         Serial.write(STATUS_SIZE_ERROR);
206         return;
207     }
208
209     // Ler e enviar os dados solicitados
210     uint8_t* ptr = (uint8_t*)destraAddress;
211     for (uint8_t i = 0; i < destraSize; i++) {
212         Serial.write(ptr[i]);
213     }
214 }
215
216
217 // Funcao para processar requisicao poke e enviar resposta
218 void process.PokeRequest() {
219     // Enviar cabecalho da resposta
220     Serial.write(0xCA);
221     Serial.write(0xFE);
222     Serial.write(CMD_POKE);
223
224     // Validar faixa de endereco
```

```

225     if (destraAddress < 0x100 || destraAddress > 0x8FF) { // Faixa
226         de RAM do Arduino Uno
227         Serial.write(STATUS_ADDRESS_RANGE_ERROR);
228         return;
229     }
230
231     // Validar tamanho (ja verificado na maquina de estados, mas
232     // dupla verificacao)
233     if (destraSize == 0 || destraSize > 8) {
234         Serial.write(STATUS_SIZE_ERROR);
235         return;
236     }
237
238     // Escrever os dados na memoria
239     uint8_t* ptr = (uint8_t*)destraAddress;
240     for (uint8_t i = 0; i < destraSize; i++) {
241         ptr[i] = destraValueBuffer[i];
242     }
243
244     // Enviar status de sucesso
245     Serial.write(STATUS_SUCCESS);
246
247     // Opcionalmente, ecoar de volta os dados escritos para
248     // verificacao
249     // Isso ajuda a confirmar que a escrita foi bem-sucedida
250     for (uint8_t i = 0; i < destraSize; i++) {
251         Serial.write(ptr[i]); // Ler de volta da memoria e enviar
252     }
253 }
```

A.2 Uso do Protocolo

Para utilizar o protocolo DESTRA em um projeto Arduino, siga estes passos:

1. Inclua o código acima no seu projeto Arduino (pode ser em um arquivo separado ou no sketch principal)
2. No `setup()`, adicione: `destraSetup();`
3. No `loop()`, adicione: `destraHandler();`
4. Compile e envie o código para o Arduino
5. Use a ferramenta DESTRA UI para conectar e interagir com o sistema

A.3 Exemplo de Integração

Listing A.2 – Exemplo de uso do protocolo DESTRA em um projeto

```
// Exemplo de sketch Arduino usando o protocolo DESTRA
#include "destra_protocol.ino" // Incluir o protocolo

// Variáveis do seu projeto
int sensorValue = 0;
float temperature = 25.5;
bool ledState = false;

void setup() {
    // Inicializar protocolo DESTRA
    destraSetup();

    // Suas inicializações do projeto
    pinMode(LED_BUILTIN, OUTPUT);
    pinMode(A0, INPUT);
}

void loop() {
    // Processar comandos DESTRA (não bloqueante)
    destraHandler();

    // Sua lógica principal do projeto
    sensorValue = analogRead(A0);
    temperature = sensorValue * 0.48828125; // Conversão para
    // temperatura

    digitalWrite(LED_BUILTIN, ledState);

    delay(100);
}
```

A.4 Especificações Técnicas

A.4.1 Parâmetros de Comunicação

- **Velocidade:** 115200 baud
- **Configuração:** 8N1 (8 bits, sem paridade, 1 stop bit)
- **Palavras mágicas:** 0xCA 0xFE

- **Timeout:** 2 segundos

A.4.2 Faixa de Endereços

- **RAM Arduino Uno:** 0x0100 - 0x08FF
- **Tamanho máximo por operação:** 8 bytes
- **Endereçamento:** 16 bits (little-endian)

A.4.3 Códigos de Status

- **0x00:** Operação bem-sucedida
- **0x01:** Erro de faixa de endereço
- **0x02:** Erro de tamanho inválido

APÊNDICE B – FERRAMENTA HOST DESTRA UI

B.1 Módulo de Protocolo - destra.py

Este apendice apresenta a implementacao completa da ferramenta host DESTRA UI, desenvolvida em Python para comunicacao com o protocolo Arduino e analise de arquivos ELF.

B.1.1 Protocolo de Comunicacao

Listing B.1 – Módulo destra.py - Protocolo de comunicacao DESTRA

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Arquivo: destra.py
5  Autor: Sandro Fadiga
6  Instituicao: EESC - USP (Escola de Engenharia de Sao Carlos)
7  Projeto: DESTRA - DEpurador de Sistemas em Tempo ReAl
8  Data de Criacao: 09/01/2025
9  Versao: 1.0
10
11 Descricao:
12     Implementacao do protocolo de comunicacao DESTRA para depuracao em tempo real
13     de sistemas embarcados Arduino. Este modulo fornece as funcionalidades de
14     peek/poke para leitura e escrita de memoria via comunicacao serial.
15
16 Funcionalidades:
17     - Auto-detectao de portas Arduino
18     - Comunicacao serial com protocolo customizado
19     - Operacoes peek (leitura de memoria)
20     - Operacoes poke (escrita de memoria)
21     - Decodificacao de tipos de dados (int, float, string, etc.)
22     - Verificacao de integridade com palavras magicas
23
24 Protocolo:
25     - Palavras magicas: 0xCA 0xFE
26     - Comandos: PEEK (0xF1), POKE (0xF2)
27     - Suporte para enderecos de 16 bits
28     - Transferencia de 1-8 bytes por operacao
29
30 Dependencias:
31     - pyserial: Para comunicacao serial
32
33 Licenca: MIT
34 """
35 from dataclasses import dataclass
36 import time
37 from typing import Optional, Union, List
38 import struct
39
40 import serial
41 import serial.tools.list_ports
42
43 from logger_config import DestraLogger

```

```

44
45 from data_dictionary import DecodedTypes
46
47 @dataclass
48 class PerformanceData:
49     frame_counter: int
50     frame_rate: int
51     frame_jitter_us: int
52     command_sequence: int
53     command_counter_delta: int
54     command_process_time_us: int
55
56     def __str__(self) -> str:
57         return f"frame_counter: {self.frame_counter}, frame_rate:{self.frame_rate},"
58         "frame_jitter_us:{self.frame_jitter_us}, command_sequence:{self."
59         "command_sequence}, command_counter_delta:{self.command_counter_delta},"
60         "command_process_time_us:{self.command_process_time_us}"
61
62     class DestraProtocol:
63         """DEpurador de Sistemas em Tempo Real - Protocolo"""
64
65         # PROTOCOLO
66         # PALAVRA MAGICA
67         MAGIC_CA = bytes.fromhex("CA")
68         MAGIC_FE = bytes.fromhex("FE")
69         PEEK_CMD = bytes.fromhex("F1")
70         POKE_CMD = bytes.fromhex("F2")
71         PERF_CMD = bytes.fromhex("F3")
72
73         # Códigos de status
74         STATUS_SUCCESS = 0x00
75         STATUS_ADDRESS_RANGE_ERROR = 0x01
76         STATUS_SIZE_ERROR = 0x02
77
78         def __init__(self, port: Optional[str] = None, baudrate: int = 115200):
79             # Usar configuração serial padrão 8N1 (sem paridade)
80             self.ser = None
81             self.baudrate = baudrate
82             self.port = port
83             self.ser = None
84
85             # Configurar logger
86             logger_manager = DestraLogger()
87             self.logger = logger_manager.logger.getChild("Protocol")
88
89         def auto_detect_arduino(self) -> tuple[List, List]:
90             """Auto-detectar porta COM do Arduino"""
91             self.logger.info("Auto-detectando porta do Arduino...")
92             ports = serial.tools.list_ports.comports()
93
94             arduino_ports = []
95             other_ports = []
96             for port in ports:
97                 # Verificar identificadores comuns do Arduino
98                 if any(
99                     x in port.description.lower()
100                     for x in ["arduino", "ch340", "ft232", "cp210"])
101                 ):
102                     arduino_ports.append(port)
103                     self.logger.debug(
104                         f"Arduino port found: {port.name} ({port.description})")
105
106         def write(self, command: bytes, value: bytes):
107             if self.ser is None:
108                 raise ValueError("Serial port not initialized")
109
110             self.ser.write(command + value)
111             self.ser.read(1)
112
113         def read(self, size: int = 1) -> bytes:
114             if self.ser is None:
115                 raise ValueError("Serial port not initialized")
116
117             return self.ser.read(size)
118
119         def close(self):
120             if self.ser is not None:
121                 self.ser.close()
122
123         def __del__(self):
124             self.close()

```

```

101             f"Arduino potencial encontrado: {port.device} - {port.description}"
102         )
103
104     if not arduino_ports:
105         self.logger.warning("Nenhum Arduino detectado. Portas disponiveis:")
106         for port in ports:
107             other_ports.append(port)
108             self.logger.debug(f" {port.device} - {port.description}")
109
110     if len(arduino_ports) > 1:
111         self.logger.info(
112             f"Multiplos Arduinos encontrados. Usando o primeiro: {arduino_ports[0]}"
113         )
114
115     return arduino_ports, other_ports
116
117 def connect(self) -> bool:
118     """Conectar ao Arduino"""
119     try:
120         self.logger.info(f"Conectando a {self.port} em {self.baudrate} baud...")
121
122         # Abrir conexao serial com configuracoes padrao 8N1
123         self.ser = serial.Serial(
124             port=self.port,
125             baudrate=self.baudrate,
126             bytesize=serial.EIGHTBITS,
127             parity=serial.PARITY_NONE,
128             stopbits=serial.STOPBITS_ONE,
129             timeout=2.0,
130             write_timeout=2.0,
131         )
132
133         # Limpar qualquer dado pendente
134         self.ser.reset_input_buffer()
135         self.ser.reset_output_buffer()
136
137         # Aguardar o Arduino reiniciar (se ele reinicia na conexao serial)
138         time.sleep(2)
139
140         # Verificar mensagem de inicializacao
141         if self.ser.in_waiting > 0:
142             startup_msg = self.ser.readline().decode("utf-8").strip()
143             self.logger.debug(f"Arduino diz: {startup_msg}")
144             if "ECHO_TEST_READY" in startup_msg:
145                 self.logger.info("Arduino esta pronto!")
146                 return True
147
148             self.logger.info("Conectado com sucesso!")
149             return True
150
151     except serial.SerialException as e:
152         self.logger.error(f"Falha ao conectar: {e}")
153         return False
154
155     def disconnect(self):
156         """Desconectar do Arduino"""
157         if self.ser and self.ser.is_open:
158             self.ser.close()
159             self.logger.info("Desconectado do Arduino")
160

```

```

161     def _common_protocol_payload(
162         self, command: bytes, address: int, size: int
163     ) -> bytes:
164         """
165             Constroi a parte comum ao pacote serial para ser enviado nos comandos
166         """
167         # Constroi a palavra magica, comando, endereco e tamanho, tansfere tudo num so
168         # write
169         message: bytes = struct.pack(">c", self.MAGIC_CA)
170         message += struct.pack(">c", self.MAGIC_FE)
171         message += struct.pack(">c", command)
172         addr_low = address & 0xFF # extrai os 8 bits
173         addr_high = (address >> 8) & 0xFF # extrai os 8 bits
174         message += struct.pack(">B", addr_low)
175         message += struct.pack(">B", addr_high)
176         message += struct.pack(">B", size)
177         return message
178
179     def _common_protocol_response(
180         self, command: bytes, address: int = 0, size: int = 0
181     ) -> tuple[bool, bytes | None]:
182         if not self.ser:
183             return False, None
184
185         # Ler cabecalho da resposta
186         response_header = self.ser.read(4) # CA FE F2 STATUS
187         if len(response_header) < 4:
188             self.logger.error(
189                 f"Cabecalho de resposta incompleto: {response_header.hex()}"
190             )
191             return False, None
192
193         # Verificar cabecalho da resposta
194         if response_header[0:3] != b"\xca\xfe" + command:
195             self.logger.error(
196                 f"Cabecalho de resposta invalido: {response_header.hex()}"
197             )
198             return False, None
199
200         # Verificar status
201         status = response_header[3]
202         if status == self.STATUS_ADDRESS_RANGE_ERROR:
203             self.logger.error(f"Erro de faixa de endereco: {address:#06x}")
204             return False, None
205         if status == self.STATUS_SIZE_ERROR:
206             self.logger.error(f"Erro de tamanho: {size}")
207             return False, None
208         if status != self.STATUS_SUCCESS:
209             self.logger.error(f"Status desconhecido: {status:#04x}")
210             return False, None
211
212         # Ler os dados
213         data = self.ser.read(size)
214         if len(data) != size:
215             self.logger.error(
216                 f"Dados incompletos: esperado {size} bytes, recebido {len(data)}"
217             )
218             return False, data
219

```

```

220
221     def peek(self, address: int, size: int) -> Optional[bytes]:
222         """
223             Enviar uma requisicao peek e retornar o conteudo da memoria no endereco
224             especificado.
225
226             Args:
227                 address: Endereco de memoria para ler (16-bit)
228                 size: Numero de bytes para ler (1-8)
229
230             Returns:
231                 objeto bytes com o conteudo da memoria, ou None se falhou
232             """
233
234         if not self.ser or not self.ser.is_open:
235             self.logger.error("Nao conectado!")
236             return None
237
238         # Validar parametros
239         if not 0 <= address <= 0xFFFF:
240             self.logger.error(f"Endereco invalido: {address:#06x} (deve ser 0-0xFFFF)")
241             return None
242
243         if not 1 <= size <= 8:
244             self.logger.error(f"Tamanho invalido: {size} (deve ser 1-8)")
245             return None
246
247         try:
248             message: bytes = self._common_protocol_payload(self.PEEK_CMD, address, size)
249             self.logger.debug(f"pacote Peek a ser enviado: {message.hex().upper()}")
250             self.ser.write(message)
251
252             status, data = self._common_protocol_response(self.PEEK_CMD, address, size)
253             if status and data:
254                 # Sucesso! Retornar os dados
255                 self.logger.debug(
256                     f"Peek bem-sucedido: endereco={address:#06x}, tamanho={size}"
257                 )
258                 self.logger.debug(f"Dados (hex): {data.hex()}")
259                 return data
260
261             except serial.SerialTimeoutException:
262                 self.logger.error("Timeout serial!")
263                 return None
264
265             except Exception as e:
266                 self.logger.error(f"Erro durante peek: {e}", exc_info=True)
267                 return None
268
269         def decode_peek_data(
270             self, data: Optional[bytes], data_type: str
271         ) -> Union[float, int, bytes, str, None]:
272             """
273                 Decodificar os bytes brutos do peek em varios tipos de dados.
274
275             Args:
276                 data: Bytes brutos da operacao peek
277                 data_type: Tipo para decodificar, disponiveis em Data Dictionary / Decode
278                     Types
279
280             Returns:
281                 Valor decodificado no tipo especificado
282             """

```

```

278     try:
279         if not data:
280             raise ValueError("Dados invalidos para decodificar peek")
281
282         decoded = DecodedTypes.decode_type(data_type)
283         if not decoded:
284             raise ValueError(f"Tipo de dados desconhecido: {data_type}")
285
286         fmt, size = decoded
287
288         if len(data) < size:
289             raise ValueError(f"Precisa de pelo menos {size} byte para {data_type}")
290
291     return struct.unpack(fmt, data[:size])[0]
292
293 except struct.error as e:
294     self.logger.error(f"Erro de desempacotamento de struct: {e}")
295     return None
296 except Exception as e:
297     self.logger.error(f"Erro de decodificacao: {e}")
298     return None
299
300 def poke(
301     self, address: int, size: Optional[int], value: Union[float, int, bytes]
302 ) -> bool:
303     """
304         Escrever dados na memoria do Arduino no endereco especificado.
305
306     Args:
307         address: Endereco de memoria para escrever (16-bit)
308         value: Dados para escrever. Pode ser:
309             - objeto bytes
310             - int (sera convertido baseado no tamanho)
311             - float (sera convertido para 4 bytes)
312         size: Numero de bytes para escrever (1-8). Se None, usa len(value)
313
314     Returns:
315         True se bem-sucedido, False caso contrario
316     """
317     if not self.ser or not self.ser.is_open:
318         self.logger.error("Nao conectado!")
319         return False
320
321     # Converter valor para bytes se necessario
322     if isinstance(value, int):
323         if size is None:
324             # Determinar tamanho baseado no valor
325             if -128 <= value <= 255:
326                 size = 1
327             elif -32768 <= value <= 65535:
328                 size = 2
329             else:
330                 size = 4
331
332         # Converter para bytes (little-endian para Arduino)
333         if size == 1:
334             value_bytes = struct.pack("<B" if value >= 0 else "<b", value)
335         elif size == 2:
336             value_bytes = struct.pack("<H" if value >= 0 else "<h", value)
337         elif size == 4:

```

```

338         value_bytes = struct.pack("<I" if value >= 0 else "<i", value)
339     else:
340         self.logger.error(f"Tamanho invalido {size} para valor inteiro")
341         return False
342
343     elif isinstance(value, float):
344         value_bytes = struct.pack("<f", value) # 4 bytes, little-endian
345         if size is None:
346             size = 4
347
348     elif isinstance(value, bytes):
349         value_bytes = value
350         if size is None:
351             size = len(value_bytes)
352         else:
353             self.logger.error(f"Tipo de valor nao suportado: {type(value)}")
354             return False
355
356     # Validar parametros
357     if not 0 <= address <= 0xFFFF:
358         self.logger.error(f"Endereco invalido: {address:#06x} (deve ser 0-0xFFFF)")
359         return False
360     if not 1 <= size <= 8:
361         self.logger.error(f"Tamanho invalido: {size} (deve ser 1-8)")
362         return False
363     if len(value_bytes) < size:
364         self.logger.error(
365             f"Bytes do valor ({len(value_bytes)}) menor que o tamanho solicitado ({size})"
366         )
367         return False
368
369     try:
370         message: bytes = self._common_protocol_payload(self.POKE_CMD, address, size)
371         message += value_bytes
372         self.logger.debug(f"pacote Poke a ser enviado: {message.hex().upper()}")
373         self.ser.write(message)
374         # Ler de volta os dados de verificacao (Arduino envia de volta o que foi
375         # escrito)
376         status, verify_data = self._common_protocol_response(
377             self.POKE_CMD, address, size
378         )
379         if status and verify_data:
380             # Verificar se os dados foram escritos corretamente
381             if verify_data != value_bytes[:size]:
382                 self.logger.error("Verificacao falhou!")
383                 self.logger.error(
384                     f"    Enviado:    {''.join(f'{b:02x}' for b in value_bytes[:size])}"
385                 )
386                 self.logger.error(
387                     f"    Lido de volta: {''.join(f'{b:02x}' for b in verify_data)}"
388                 )
389             return False
390             # Sucesso!
391             self.logger.debug(
392                 f"Poke bem-sucedido: endereco={address:#06x}, tamanho={size}"
393             )
394             self.logger.debug(
395                 f"Dados escritos: {''.join(f'{b:02x}' for b in value_bytes[:size])}"
396             )

```

```

395         )
396     return status
397
398 except serial.SerialTimeoutException:
399     self.logger.error("Timeout serial!")
400     return False
401 except Exception as e:
402     self.logger.error(f"Erro durante poke: {e}")
403     return False
404
405 def performance(self) -> list[PerformanceData]:
406     """
407         Enviar uma requisicao para o protocolo de teste (extra ao protocolo DESTRA) fazer
408         o dump dos logs de performance.
409     Returns:
410         objeto bytes com o conteudo da memoria, ou None se falhou
411     """
412     samples = []
413     if not self.ser or not self.ser.is_open:
414         self.logger.error("Nao conectado!")
415         return samples
416     try:
417         message: bytes = struct.pack(">c", self.MAGIC_CA)
418         message += struct.pack(">c", self.MAGIC_FE)
419         message += struct.pack(">c", self.PERF_CMD)
420         self.logger.debug(f"Pacote Dump a ser enviado: {message.hex().upper()}")
421         self.ser.write(message)
422
423         self.logger.info("== DUMP DE LOGS (Arduino) ==")
424
425         # Ler cabecalho da resposta
426         response_header = self.ser.read(5)  # CA FE F3 STATUS LOG_SIZE
427         if len(response_header) < 5:
428             self.logger.error(
429                 f"Cabecalho de resposta incompleto: {response_header.hex()}"
430             )
431             return samples
432         # Verificar cabecalho da resposta
433         if response_header[0:3] != b"\xca\xfe\xfa" + self.PERF_CMD:
434             self.logger.error(
435                 f"Cabecalho de resposta invalido: {response_header.hex()}"
436             )
437             return samples
438
439         # Verificar status
440         status = response_header[3]
441         if status != self.STATUS_SUCCESS:
442             self.logger.error(f"Status desconhecido: {status:#04x}")
443             return samples
444         # Quantas entradas de performance estao retornando
445         perf_entries = response_header[4]
446         self.logger.info(f"Processando um total de {perf_entries} entradas.")
447         if not 0 < perf_entries < 256:
448             self.logger.error(
449                 f"Cabecalho de resposta invalid perf_entries: {perf_entries}"
450             )
451             return samples
452
453         for i in range(perf_entries):
454             frame_counter = int.from_bytes(self.ser.read(4), "little")

```

```

454         frame_rate = int.from_bytes(self.ser.read(2), "little")
455         frame_jitter_us = int.from_bytes(self.ser.read(2), "little")
456         command_sequence = int.from_bytes(self.ser.read(2), "little")
457         command_counter_delta = int.from_bytes(self.ser.read(2), "little")
458         command_process_time_us = int.from_bytes(self.ser.read(4), "little")
459         perf = PerformanceData(frame_counter, frame_rate, frame_jitter_us,
460             command_sequence, command_counter_delta, command_process_time_us)
460         samples.append(perf)
461         self.logger.info("== FIM DO DUMP ==\n")
462     except serial.SerialTimeoutException:
463         self.logger.error("Timeout serial!")
464     except Exception as e:
465         self.logger.error(f"Erro durante peek: {e}", exc_info=True)
466     return samples
467
468 def main():
469     """Funcao principal para executar o teste de eco"""
470     import sys
471
472     # Verificar argumentos da linha de comando
473     port = None
474     if len(sys.argv) > 1:
475         port = sys.argv[1]
476         print(f"Usando porta especificada: {port}")
477
478     try:
479         # Criar instancia de teste
480         protocol = DestraProtocol(port=port)
481
482         if not protocol.connect():
483             print("\nFalha ao conectar ao Arduino!")
484             return
485
486         # Permitir que o Arduino estabilize
487         time.sleep(0.5)
488
489         # Executar todos os testes
490         data = protocol.peek(int(0x012F), 4)
491         val = protocol.decode_peek_data(data, "uint32")
492         print(f" {val} ")
493     except Exception as e:
494         print(f"\nErro: {e}")
495         print("\nUso: python test_serial.py [PORTA_COM]")
496         print(" Exemplo: python test_serial.py COM3")
497         print(" Se nenhuma porta for especificada, tentara auto-detectar o Arduino")
498         sys.exit(1)
499
500
501 if __name__ == "__main__":
502     main()

```

B.2 Interface Grafica - destra_ui.py

B.2.1 Implementacao da GUI

Listing B.2 – Interface grafica DESTRA UI

```

1 #!/usr/bin/env python3
2 # -*- coding: utf-8 -*-

```

```
3 """
4 Arquivo: destra_ui.py
5 Autor: Sandro Fadiga
6 Instituicao: EESC - USP (Escola de Engenharia de Sao Carlos)
7 Projeto: DESTRA - DEpurador de Sistemas em Tempo ReAl
8 Data de Criacao: 09/01/2025
9 Versao: 1.0
10
11 Descricao:
12     Interface grafica (GUI) para o sistema DESTRA de depuracao em tempo real.
13     Fornece uma interface amigavel para operacoes peek/poke em sistemas Arduino,
14     com analise de arquivos ELF para extracao de informacoes de variaveis.
15
16 Funcionalidades:
17     - Interface grafica intuitiva com PySide6/Qt
18     - Selecao e conexao automatica com Arduino
19     - Carregamento e analise de arquivos ELF
20     - Busca e filtragem de variaveis
21     - Operacoes peek (leitura) e poke (escrita) de memoria
22     - Auto-peek com frequencia configuravel
23     - Visualizacao em tabelas interativas
24     - Suporte a drag-and-drop entre tabelas
25
26 Interface:
27     - Secao de conexao: Deteccao e conexao com Arduino
28     - Secao de arquivo: Selecao de arquivo ELF
29     - Tabela de variaveis disponiveis: Lista todas as variaveis do ELF
30     - Tabela de variaveis selecionadas: Variaveis para monitoramento
31     - Controles de peek/poke: Leitura e escrita de valores
32
33 Dependencias:
34     - PySide6: Framework Qt para interface grafica
35     - pyserial: Comunicacao serial
36     - data_dictionary: Parser de arquivos ELF
37     - destra: Protocolo de comunicacao DESTRA
38
39 Licenca: MIT
40 """
41
42 import sys
43 import time
44 from PySide6.QtWidgets import (
45     QApplication,
46     QMainWindow,
47     QWidget,
48     QVBoxLayout,
49     QHBoxLayout,
50     QComboBox,
51     QSpinBox,
52     QLineEdit,
53     QPushButton,
54     QTableWidget,
55     QTableWidgetItem,
56     QLabel,
57     QFileDialog,
58     QSplitter,
59     QGroupBox,
60     QMessageBox,
61     QCheckBox,
62 )
```

```
63 from PySide6.QtCore import Qt, QTimer
64
65 # Importar o dicionario de dados ELF
66 from data_dictionary import ElfDataDictionary, VariableInfo
67 from destra import DestraProtocol
68 from logger_config import DestraLogger
69
70
71 class DestraGUI(QMainWindow):
72
73     PERF_DUMP_FILE = "./tests/destra_ui_{freq}hz.log"
74     ARDUINO_DUMP_FILE = "./tests/arduino_{freq}hz.log"
75
76     def __init__(self):
77         super().__init__()
78         self.setWindowTitle("DEpurador de Sistemas em Tempo Real - DESTRA")
79         self.setGeometry(100, 100, 1200, 800)
80         self.setStyleSheet("font: 600 14pt")
81
82         # Configurar logger
83         self.logger_manager = DestraLogger()
84         self.logger = self.logger_manager.logger.getChild("UI")
85
86         # Armazenar o dicionario de dados ELF
87         self.elf_data = None
88         # Armazenar as variaveis do elf
89         self.all_variables = []
90         self._variable_list: list[VariableInfo] = []
91         self._destra: DestraProtocol = DestraProtocol()
92         # port -> port.device , port.description
93         self._arduino_ports = []
94         self._other_ports = []
95         self._current_log = []
96
97         # Criar widget central e layout principal
98         central_widget = QWidget()
99         self.setCentralWidget(central_widget)
100        main_layout = QVBoxLayout(central_widget)
101
102        # Criar layout de ui
103        self._create_connection_section(main_layout)
104        self._create_file_section(main_layout)
105        self._create_tables_section(main_layout)
106
107        main_layout.setStretch(0, 0)
108        main_layout.setStretch(1, 0)
109        main_layout.setStretch(2, 3)
110        # main_layout.setStretch(3, 0)
111
112        # Configurar timer para atualizacao periodica de portas COM
113        self.auto_peek_timer = QTimer()
114        self.auto_peek_timer.timeout.connect(self.peek_values)
115
116        self.refresh_com_ports()
117        self._is_connected = False
118
119    def _text_2_num(self, val_txt: str) -> float | int | None:
120        """Converter texto para formato int ou float, caso nao consiga retorna None"""
121        val = None
122        try:
```

```

123         val = int(val_txt)
124         return val
125     except:
126         try:
127             val = float(val_txt)
128             return val
129         except:
130             return val
131
132     def _create_connection_section(self, parent_layout):
133         """Criar a secao de configuracoes de conexao"""
134         group = QGroupBox("Configuracoes de Conexao")
135         layout = QHBoxLayout()
136
137         # Scaneia as portas disponiveis
138         self.scan_button = QPushButton("Detectar")
139         self.scan_button.setMinimumWidth(150)
140         self.scan_button.clicked.connect(self.refresh_com_ports)
141         layout.addWidget(self.scan_button)
142
143         # Selecao de porta COM
144         layout.addWidget(QLabel("Portas:"))
145         self.com_port_combo = QComboBox()
146         self.com_port_combo.setMinimumWidth(150)
147         layout.addWidget(self.com_port_combo)
148
149         # connecta com a porta selectionada
150         self.connect_button = QPushButton("Connectar")
151         self.connect_button.setMinimumWidth(150)
152         self.connect_button.clicked.connect(self.connect_to_arduino)
153         layout.addWidget(self.connect_button)
154
155         # Taxa de amostragem
156         self.auto_peek_check = QCheckBox()
157         self.auto_peek_check.setText("Auto Peek")
158         layout.addWidget(self.auto_peek_check)
159         self.auto_peek_check.stateChanged.connect(self.start_stop_auto_peek)
160
161         layout.addWidget(QLabel("Freq. (Hz):"))
162         self.sample_rate_spin = QSpinBox()
163         self.sample_rate_spin.setRange(1, 1000)
164         self.sample_rate_spin.setValue(10) # Padrao 10 Hz
165         self.sample_rate_spin.valueChanged.connect(self.change_auto_peek_freq)
166         self.sample_rate_spin.setSuffix(" Hz")
167         layout.addWidget(self.sample_rate_spin)
168
169         # Adicionar controle de nivel de logging
170         layout.addWidget(QLabel("Log Level:"))
171         self.log_level_combo = QComboBox()
172         self.log_level_combo.addItems(["DEBUG", "INFO", "WARNING", "ERROR", "CRITICAL"])
173         self.log_level_combo.setCurrentText("ERROR")
174         self.log_level_combo.currentTextChanged.connect(self.change_log_level)
175         layout.addWidget(self.log_level_combo)
176
177         self.dump_button = QPushButton("Baixar logs")
178         self.dump_button.setMinimumWidth(150)
179         self.dump_button.clicked.connect(self.dump_performance_logs)
180         layout.addWidget(self.dump_button)
181
182         layout.addStretch()

```

```
183     group.setLayout(layout)
184     parent_layout.addWidget(group)
185
186     def _create_file_section(self, parent_layout):
187         """Criar a secao de selecao de arquivo"""
188         group = QGroupBox("Selecao de Arquivo ELF")
189         layout = QHBoxLayout()
190
191         layout.addWidget(QLabel("Arquivo ELF:"))
192         self.file_path_edit = QLineEdit()
193         self.file_path_edit.setReadOnly(True)
194         self.file_path_edit.setPlaceholderText("Selecione um arquivo ELF...")
195         layout.addWidget(self.file_path_edit)
196
197         self.browse_button = QPushButton("Procurar...")
198         self.browse_button.clicked.connect(self.browse_file)
199         layout.addWidget(self.browse_button)
200
201     group.setLayout(layout)
202     parent_layout.addWidget(group)
203
204     def _create_tables_section(self, parent_layout):
205         """Criar a secao de duas tabelas com funcionalidade de busca"""
206         # Criar divisor para tabelas redimensionaveis
207         splitter = QSplitter(Qt.Horizontal)
208
209         # Secao de variaveis disponiveis
210         available_group = QGroupBox("Variaveis Disponiveis")
211         available_layout = QVBoxLayout()
212
213         # Adicionar caixa de busca para variaveis disponiveis
214         search_layout = QHBoxLayout()
215         search_layout.addWidget(QLabel("Buscar:"))
216         self.search_edit = QLineEdit()
217         self.search_edit.setPlaceholderText(
218             "Digite o padrao do nome da variavel (ex: timer*, *pin*, gyro_*)"
219         )
220         self.search_edit.textChanged.connect(self.filter_variables)
221         search_layout.addWidget(self.search_edit)
222
223         # Adicionar botao de limpar
224         self.clear_search_button = QPushButton("Limpar")
225         self.clear_search_button.clicked.connect(self.clear_search)
226         search_layout.addWidget(self.clear_search_button)
227
228         available_layout.addLayout(search_layout)
229
230         # Tabela de variaveis disponiveis
231         self.available_table = QTableWidget()
232         self.available_table.setColumnCount(4)
233         self.available_table.setHorizontalHeaderLabels(
234             ["Nome", "Endereco", "Tipo", "Tamanho"]
235         )
236         self.available_table.horizontalHeader().setStretchLastSection(True)
237         self.available_table.setSelectionBehavior(QTableWidget.SelectRows)
238         self.available_table.setAlternatingRowColors(True)
239         self.available_table.setSortingEnabled(True)
240
241         # Habilitar arrastar da tabela disponivel
242         self.available_table.setDragEnabled(True)
```

```
243         self.available_table.setDefaultDropAction(Qt.CopyAction)
244
245         available_layout.addWidget(self.available_table)
246
247         # Adicionar rotulo de status
248         self.status_label = QLabel("Nenhum arquivo ELF carregado")
249         available_layout.addWidget(self.status_label)
250
251         available_group.setLayout(available_layout)
252
253         # Tabela de variaveis selecionadas
254         selected_group = QGroupBox("Variaveis Selecionadas")
255         selected_layout = QVBoxLayout()
256
257         self._create_buttons_section(selected_layout)
258
259         self.selected_table = QTableWidget()
260         self.selected_table.setColumnCount(3)
261         self.selected_table.setHorizontalHeaderLabels(
262             ["Nome", "Valor Peek", "Valor Poke"])
263
264         self.selected_table.horizontalHeader().setStretchLastSection(True)
265         self.selected_table.setSelectionBehavior(QTableWidget.SelectRows)
266         self.selected_table.setAlternatingRowColors(True)
267
268         # Habilitar soltar na tabela selecionada
269         self.selected_table.setAcceptDrops(True)
270         self.selected_table.viewport().setAcceptDrops(True)
271         self.selected_table.setDropIndicatorShown(True)
272
273         selected_layout.addWidget(self.selected_table)
274         selected_group.setLayout(selected_layout)
275
276         # Adicionar ao divisor
277         splitter.addWidget(available_group)
278         splitter.addWidget(selected_group)
279         splitter.setSizes([600, 600]) # Tamanhos iniciais
280
281         parent_layout.addWidget(splitter)
282
283         # Conectar duplo clique para adicionar variavel
284         self.available_table.itemDoubleClicked.connect(self.add_variable_to_selected)
285         self.selected_table.itemDoubleClicked.connect(self.on_poke_cell_double_clicked)
286         self.selected_table.itemChanged.connect(self.on_poke_cell_edited)
287
288     def _create_buttons_section(self, parent_layout):
289         """Criar a secao de botoes de acao"""
290         layout = QHBoxLayout()
291         layout.addStretch()
292
293         self.peek_button = QPushButton("Peek")
294         self.peek_button.setMinimumWidth(150)
295         self.peek_button.clicked.connect(self.peek_values)
296         layout.addWidget(self.peek_button)
297
298         self.poke_button = QPushButton("Poke")
299         self.poke_button.setMinimumWidth(150)
300         self.poke_button.clicked.connect(self.poke_values)
301         layout.addWidget(self.poke_button)
302
```

```

303     layout.addStretch()
304     parent_layout.addWidget(layout)

```

B.3 Analisador ELF - data_dictionary.py

B.3.1 Parser de Arquivos ELF e Extracao de Variaveis

Listing B.3 – Módulo data_dictionary.py - Analise de arquivos ELF

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Arquivo: data_dictionary.py
5  Autor: Sandro Fadiga
6  Instituicao: EESC - USP (Escola de Engenharia de Sao Carlos)
7  Projeto: DESTRA - DEpurador de Sistemas em Tempo Real
8  Data de Criacao: 09/01/2025
9  Versao: 1.0
10
11 Descricao:
12     Analisador de Dicionario de Dados ELF para Arduino Uno.
13     Este modulo fornece uma interface simplificada para extrair informacoes
14     de variaveis de arquivos ELF com informacoes de debug DWARF, especificamente
15     projetado para implementar funcionalidade peek/poke no Arduino Uno.
16
17 Funcionalidades:
18     - Analise de arquivos ELF com informacoes DWARF
19     - Extracao de enderecos e tipos de variaveis
20     - Suporte para arrays, structs e tipos basicos
21     - Interface simplificada para operacoes peek/poke
22     - Busca e filtragem de variaveis por padroes
23
24 Dependencias:
25     - pyelftools: Para analise de arquivos ELF e DWARF
26
27 Licenca: MIT
28 """
29
30 from __future__ import annotations
31 from dataclasses import dataclass
32 from pathlib import Path
33 from typing import Dict, List, Optional, Tuple, Union
34
35 from elftools.dwarf.die import DIE
36 from elftools.elf.elffile import ELFFile
37
38 from logger_config import DestraLogger
39
40 # Configurar logger para este modulo
41 logger_manager = DestraLogger()
42 logger = logger_manager.logger.getChild("DataDictionary")
43
44
45 @dataclass
46 class VariableInfo:
47     """Informacoes sobre uma variavel extraida do arquivo ELF."""
48
49     name: str
50     address: int

```

```

51     size: int
52     base_type: str # 'uint8', 'uint16', 'uint32', 'int8', 'int16', 'int32', 'float', '
53         'double', 'struct', 'array'
54     is_signed: bool
55     is_pointer: bool
56     array_dimensions: Optional[List[int]] = None
57     struct_name: Optional[str] = None
58
59     def __hash__(self) -> int:
60         return hash(self.name)
61
62 class DecodedTypes:
63
64     @staticmethod
65     def decode_type(vartype: str) -> tuple[str, int] | None:
66         LookupTable = {
67             "bytes": ("<B", -1),
68             "uint8": ("<B", 1),
69             "int8": ("<b", 1),
70             "uint16": ("<H", 2),
71             "int16": ("<h", 2),
72             "uint32": ("<I", 4),
73             "int32": ("<i", 4),
74             "long unsigned int": ("<I", 4),
75             "long": ("<i", 4),
76             "long signed int": ("<i", 4),
77             "float": ("f", 4),
78             "double": ("d", 8),
79         }
80         return LookupTable.get(vartype, None)
81
82 class ElfDataDictionary:
83     """
84     Um analisador ELF simplificado para extrair informacoes de variaveis.
85
86     Esta classe fornece metodos para listar e consultar variaveis de um arquivo ELF
87     com informacoes de debug DWARF, otimizado para operacoes peek/poke.
88     """
89
90     # Mapeamento de tipos C basicos para categorias simplificadas
91     TYPE_MAPPING = {
92         ("unsigned char", 1): ("uint8", False),
93         ("uint8_t", 1): ("uint8", False),
94         ("char", 1): ("int8", True),
95         ("signed char", 1): ("int8", True),
96         ("unsigned short", 2): ("uint16", False),
97         ("uint16_t", 2): ("uint16", False),
98         ("short", 2): ("int16", True),
99         ("int", 2): ("int16", True), # int do Arduino Uno e 16-bit
100        ("unsigned int", 2): ("uint16", False),
101        ("unsigned long", 4): ("uint32", False),
102        ("uint32_t", 4): ("uint32", False),
103        ("long", 4): ("int32", True),
104        ("long int", 4): ("int32", True),
105        ("float", 4): ("float", True),
106        ("double", 4): ("float", True), # double do Arduino Uno e igual a float
107    }
108
109    def __init__(self, elf_path: Union[str, Path]):
```

```

110     """
111     Inicializar o analisador de dicionario de dados ELF.
112
113     Args:
114         elf_path: Caminho para o arquivo ELF
115
116     Raises:
117         FileNotFoundError: Se o arquivo ELF nao existir
118         ValueError: Se o arquivo nao for um ELF valido ou nao tiver informacoes de
119             debug
120     """
121     self.elf_path = Path(elf_path)
122     if not self.elf_path.exists():
123         raise FileNotFoundError(f"Arquivo ELF nao encontrado: {self.elf_path}")
124
125     self._variables: Dict[str, VariableInfo] = {}
126     self._parse_elf_file()
127
128     def _parse_elf_file(self) -> None:
129         """Analisar o arquivo ELF e extrair informacoes de variaveis."""
130         with open(self.elf_path, "rb") as f:
131             try:
132                 elffile = ELFFile(f)
133             except Exception as e:
134                 raise ValueError(f"Arquivo ELF invalido: {e}")
135
136         if not elffile.has_dwarf_info():
137             raise ValueError("Arquivo ELF nao possui informacoes de debug DWARF")
138
139         dwarf_info = elffile.get_dwarf_info()
140
141         # Analisar todas as unidades de compilacao
142         for cu in dwarf_info.iter_CUs():
143             self._parse_compilation_unit(cu)
144
145         def _parse_compilation_unit(self, cu) -> None:
146             """Analisar uma unica unidade de compilacao para variaveis."""
147             # Construir um cache de offsets DIE para informacoes de tipo analisadas
148             type_cache = {}
149
150             # Primeira passada: analisar todas as definicoes de tipo
151             for die in cu.iter_DIEs():
152                 if die.tag in [
153                     "DW_TAG_base_type",
154                     "DW_TAG_typeDefinition",
155                     "DW_TAG_structure_type",
156                     "DW_TAG_array_type",
157                     "DW_TAG_pointer_type",
158                     "DW_TAG_const_type",
159                     "DW_TAG_volatile_type",
160                 ]:
161                     type_info = self._parse_type(die, cu, type_cache)
162                     if type_info:
163                         type_cache[die.offset] = type_info
164
165             # Segunda passada: analisar variaveis
166             for die in cu.iter_DIEs():
167                 if (
168                     die.tag == "DW_TAG_variable"
169                     and "DW_AT_declaration" not in die.attributes

```

```

169         ):
170             self._parse_variable(die, cu, type_cache)
171
172     def _parse_type(self, die: DIE, cu, type_cache: dict) -> Optional[dict]:
173         """Analizar informacoes de tipo de um DIE."""
174         type_info = {}
175
176         if die.tag == "DW_TAG_base_type":
177             name = die.attributes.get("DW_AT_name")
178             size = die.attributes.get("DW_AT_byte_size")
179             if name and size:
180                 name_str = name.value.decode("utf-8")
181                 size_val = size.value
182                 type_key = (name_str, size_val)
183
184                 if type_key in self.TYPE_MAPPING:
185                     base_type, is_signed = self.TYPE_MAPPING[type_key]
186                     type_info = {
187                         "base_type": base_type,
188                         "size": size_val,
189                         "is_signed": is_signed,
190                         "is_pointer": False,
191                         "is_array": False,
192                         "is_struct": False,
193                     }
194                 else:
195                     # Tipo base desconhecido, armazenar informacao bruta
196                     type_info = {
197                         "base_type": name_str,
198                         "size": size_val,
199                         "is_signed": "unsigned" not in name_str.lower(),
200                         "is_pointer": False,
201                         "is_array": False,
202                         "is_struct": False,
203                     }
204
205         elif die.tag == "DW_TAG_typedef":
206             # Seguir o typedef ate o tipo real
207             type_attr = die.attributes.get("DW_AT_type")
208             if type_attr:
209                 ref_offset = self._get_reference_offset(type_attr, cu)
210                 if ref_offset in type_cache:
211                     type_info = type_cache[ref_offset].copy()
212                 else:
213                     # Tentar encontrar o DIE referenciado
214                     ref_die = self._get_die_at_offset(cu, ref_offset)
215                     if ref_die:
216                         type_info = self._parse_type(ref_die, cu, type_cache) or {}
217
218         elif die.tag == "DW_TAG_pointer_type":
219             type_info = {
220                 "base_type": "uint16", # Ponteiros sao 16-bit no Arduino Uno
221                 "size": 2,
222                 "is_signed": False,
223                 "is_pointer": True,
224                 "is_array": False,
225                 "is_struct": False,
226             }
227
228         elif die.tag == "DW_TAG_array_type":

```

```

229     type_attr = die.attributes.get("DW_AT_type")
230     if type_attr:
231         ref_offset = self._get_reference_offset(type_attr, cu)
232         element_type = type_cache.get(ref_offset, {})
233
234         # Obter dimensoes do array
235         dimensions = []
236         for child in die.iter_children():
237             if child.tag == "DW_TAG_subrange_type":
238                 upper_bound = child.attributes.get("DW_AT_upper_bound")
239                 if upper_bound:
240                     dimensions.append(upper_bound.value + 1)
241
242         if element_type and dimensions:
243             total_size = element_type.get("size", 1)
244             for dim in dimensions:
245                 total_size *= dim
246
247             type_info = element_type.copy()
248             type_info["is_array"] = True
249             type_info["array_dimensions"] = dimensions
250             type_info["size"] = total_size
251
252     elif die.tag == "DW_TAG_structure_type":
253         name = die.attributes.get("DW_AT_name")
254         size = die.attributes.get("DW_AT_byte_size")
255
256         if size: # Processar apenas definicoes completas de struct
257             struct_name = name.value.decode("utf-8") if name else "anonymous_struct"
258             type_info = {
259                 "base_type": "struct",
260                 "size": size.value,
261                 "is_signed": False,
262                 "is_pointer": False,
263                 "is_array": False,
264                 "is_struct": True,
265                 "struct_name": struct_name,
266             }
267
268             # Analisar membros da struct
269             members = []
270             for member_die in die.iter_children():
271                 if member_die.tag == "DW_TAG_member":
272                     member_info = self._parse_struct_member(
273                         member_die, cu, type_cache
274                     )
275                     if member_info:
276                         members.append(member_info)
277
278             type_info["members"] = members
279
280     elif die.tag in ["DW_TAG_const_type", "DW_TAG_volatile_type"]:
281         # Estes sao qualificadores de tipo, seguir ate o tipo real
282         type_attr = die.attributes.get("DW_AT_type")
283         if type_attr:
284             ref_offset = self._get_reference_offset(type_attr, cu)
285             if ref_offset in type_cache:
286                 type_info = type_cache[ref_offset].copy()
287             else:
288                 ref_die = self._get_die_at_offset(cu, ref_offset)

```

```

289         if ref_die:
290             type_info = self._parse_type(ref_die, cu, type_cache) or {}
291
292     return type_info
293
294 def _parse_struct_member(self, die: DIE, cu, type_cache: dict) -> Optional[dict]:
295     """Analisar um membro de struct."""
296     name = die.attributes.get("DW_AT_name")
297     type_attr = die.attributes.get("DW_AT_type")
298     location = die.attributes.get("DW_AT_data_member_location")
299
300     if name and type_attr:
301         member_name = name.value.decode("utf-8")
302         ref_offset = self._get_reference_offset(type_attr, cu)
303         member_type = type_cache.get(ref_offset, {})
304
305         offset = 0
306         if location:
307             # Extrair offset da localizacao
308             if isinstance(location.value, list) and len(location.value) > 1:
309                 # Formato DW_OP_plus_uconst
310                 offset = location.value[1]
311             elif isinstance(location.value, int):
312                 offset = location.value
313
314         return {"name": member_name, "offset": offset, "type_info": member_type}
315
316     return None
317
318 def _parse_variable(self, die: DIE, cu, type_cache: dict) -> None:
319     """Analisar um DIE de variavel e adiciona-lo ao dicionario."""
320     name_attr = die.attributes.get("DW_AT_name")
321     type_attr = die.attributes.get("DW_AT_type")
322     location_attr = die.attributes.get("DW_AT_location")
323
324     # Deve ter pelo menos nome e tipo
325     if not (name_attr and type_attr):
326         return
327
328     var_name = name_attr.value.decode("utf-8")
329
330     # Obter endereco da localizacao se disponivel
331     address = 0
332     if location_attr:
333         address = self._extract_address(location_attr.value)
334         if address is None:
335             # Se nao conseguirmos extrair o endereco, tentar usar um padrao ou pular
336             # Algumas variaveis podem estar otimizadas ou em registradores
337             logger.warning(
338                 f"Nao foi possivel extrair endereco para variavel '{var_name}'"
339             )
340             # Por enquanto, ainda vamos adiciona-la com endereco 0
341             address = 0
342     else:
343         # Variavel sem localizacao - pode ser externa, const ou otimizada
344         # Verificar se e externa
345         if "DW_AT_external" in die.attributes:
346             logger.debug(f"Variavel '{var_name}' e externa (global)")
347             # Variaveis externas devem ter enderecos, mas podem precisar de linking
348             address = 0 # Sera resolvido no tempo de link

```

```

349         else:
350             logger.debug(
351                 f"Variavel '{var_name}' nao tem localizacao (pode estar otimizada)"
352             )
353             # Ainda adicionar com endereco 0 para completude
354             address = 0
355
356             # Obter informacoes de tipo
357             ref_offset = self._get_reference_offset(type_attr, cu)
358             type_info = type_cache.get(ref_offset, {})
359
360             if not type_info:
361                 return
362
363             # Criar informacoes base da variavel
364             var_info = VariableInfo(
365                 name=var_name,
366                 address=address,
367                 size=type_info.get("size", 0),
368                 base_type=type_info.get("base_type", "unknown"),
369                 is_signed=type_info.get("is_signed", False),
370                 is_pointer=type_info.get("is_pointer", False),
371                 array_dimensions=type_info.get("array_dimensions"),
372                 struct_name=type_info.get("struct_name"),
373             )
374
375             self._variables[var_name] = var_info
376
377             # Se for um array, adicionar entradas de elementos individuais
378             if type_info.get("is_array") and type_info.get("array_dimensions"):
379                 self._add_array_elements(var_name, var_info, type_info)
380
381             # Se for uma struct, adicionar entradas de membros
382             if type_info.get("is_struct") and "members" in type_info:
383                 self._add_struct_members(var_name, var_info, type_info)
384
385             def _add_array_elements(
386                 self, base_name: str, base_info: VariableInfo, type_info: dict
387             ) -> None:
388                 """Adicionar entradas de elementos individuais do array."""
389                 dimensions = type_info["array_dimensions"]
390                 element_size = type_info["size"] // (dimensions[0] if dimensions else 1)
391
392                 # Por enquanto, apenas lidar com arrays 1D
393                 if len(dimensions) == 1:
394                     for i in range(dimensions[0]):
395                         element_name = f"{base_name}[{i}]"
396                         element_info = VariableInfo(
397                             name=element_name,
398                             address=base_info.address + (i * element_size),
399                             size=element_size,
400                             base_type=base_info.base_type,
401                             is_signed=base_info.is_signed,
402                             is_pointer=False,
403                             array_dimensions=None,
404                             struct_name=None,
405                         )
406                         self._variables[element_name] = element_info
407
408             def _add_struct_members(

```

```

409         self, base_name: str, base_info: VariableInfo, type_info: dict
410     ) -> None:
411         """Adicionar entradas de membros da struct."""
412         for member in type_info.get("members", []):
413             member_name = f"{base_name}.{member['name']}"
414             member_type_info = member.get("type_info", {})
415
416             member_info = VariableInfo(
417                 name=member_name,
418                 address=base_info.address + member["offset"],
419                 size=member_type_info.get("size", 0),
420                 base_type=member_type_info.get("base_type", "unknown"),
421                 is_signed=member_type_info.get("is_signed", False),
422                 is_pointer=member_type_info.get("is_pointer", False),
423                 array_dimensions=member_type_info.get("array_dimensions"),
424                 struct_name=member_type_info.get("struct_name"),
425             )
426             self._variables[member_name] = member_info
427
428     def _get_reference_offset(self, attr, cu) -> int:
429         """Obter o offset absoluto para um atributo de referencia."""
430         offset = attr.value
431         if attr.form == "DW_FORM_ref4":
432             # Offset relativo a CU
433             offset += cu.cu_offset
434         return offset
435
436     def _get_die_at_offset(self, cu, offset: int) -> Optional[DIE]:
437         """Obter um DIE em um offset especifico dentro de uma CU."""
438         try:
439             return cu._get_cached_DIE(offset)
440         except:
441             return None
442
443     def _extract_address(self, location_value) -> Optional[int]:
444         """Extrair endereco de expressao de localizacao DWARF."""
445         if isinstance(location_value, list):
446             if len(location_value) == 0:
447                 return None
448
449             # Verificar DW_OP_addr (0x03) - endereco direto
450             if location_value[0] == 0x03 and len(location_value) >= 3:
451                 # Endereco segue o opcode (16-bit para AVR)
452                 return int.from_bytes(location_value[1:3], "little")
453
454             # Verificar outras expressoes de localizacao comuns
455             # DW_OP_fbreg - relativo a base do frame (variaveis locais)
456             if location_value[0] == 0x91:
457                 # Esta e uma variavel relativa a pilha, nao podemos obter endereco
458                 # absoluto
459                 return None
460
461             # Tentar extrair dos ultimos 2 bytes (comportamento legado)
462             if len(location_value) >= 2:
463                 # Alguns compiladores colocam o endereco diretamente sem opcode
464                 return int.from_bytes(location_value[-2:], "little")
465
466             elif isinstance(location_value, int):
467                 # Valor de endereco direto
468                 return location_value

```

```

468
469     return None
470
471     # Metodos da API publica
472
473     def list_variables(self, pattern: Optional[str] = None) -> List[str]:
474         """
475             Listar todos os nomes de variaveis, opcionalmente filtrados por padrao.
476
477             Args:
478                 pattern: Padrao glob opcional para filtrar nomes de variaveis (ex: "dig*pin*"
479                             pgm")
480
481             Returns:
482                 Lista de nomes de variaveis correspondentes ao padrao
483
484             names = list(self._variables.keys())
485
486             if pattern:
487                 # Converter padrao glob para correspondencia case-insensitive
488                 import fnmatch
489
490                 pattern_lower = pattern.lower()
491                 names = [
492                     name for name in names if fnmatch.fnmatch(name.lower(), pattern_lower)
493                 ]
494
495             return sorted(names)
496
497     def get_variable_info(self, name: str) -> Optional[Tuple[int, int, str]]:
498         """
499             Obter informacoes de variavel por nome.
500
501             Args:
502                 name: Nome da variavel (pode incluir indices de array ou membros de struct)
503
504             Returns:
505                 Tupla de (endereco, tamanho, tipo) ou None se nao encontrado
506
507             var_info = self._variables.get(name)
508             if var_info:
509                 return (var_info.address, var_info.size, var_info.base_type)
510             return None
511
512     def get_detailed_variable_info(self, name: str) -> Optional[VariableInfo]:
513         """
514             Obter informacoes detalhadas de variavel por nome.
515
516             Args:
517                 name: Nome da variavel
518
519             Returns:
520                 Objeto VariableInfo ou None se nao encontrado
521
522             return self._variables.get(name)
523
524     def search_variables(
525         self,
526         pattern: str,
527         min_size: Optional[int] = None,

```

```

527         max_size: Optional[int] = None,
528         var_type: Optional[str] = None,
529     ) -> List[VariableInfo]:
530         """
531             Buscar variaveis com multiplos filtros.
532
533         Args:
534             pattern: Padrao glob para correspondencia de nome
535             min_size: Tamanho minimo da variavel em bytes
536             max_size: Tamanho maximo da variavel em bytes
537             var_type: Filtrar por tipo (ex: 'uint8', 'float', 'struct')
538
539         Returns:
540             Lista de objetos VariableInfo correspondentes a todos os criterios
541         """
542         results = []
543
544         # Primeiro filtrar por padrao de nome
545         matching_names = self.list_variables(pattern)
546
547         for name in matching_names:
548             var_info = self._variables[name]
549
550             # Aplicar filtros de tamanho
551             if min_size is not None and var_info.size < min_size:
552                 continue
553             if max_size is not None and var_info.size > max_size:
554                 continue
555
556             # Aplicar filtro de tipo
557             if var_type is not None and var_info.base_type != var_type:
558                 continue
559
560             results.append(var_info)
561
562         return results
563
564     def get_all_variables(self) -> Dict[str, VariableInfo]:
565         """Obter todas as variaveis como um dicionario."""
566         return self._variables.copy()
567
568
569     def main():
570         """Exemplo de uso do dicionario de dados ELF."""
571         import sys
572         from pathlib import Path
573
574         # Configurar logger para o main
575         test_logger = logger_manager.logger.getChild("Test")
576
577         if len(sys.argv) < 2:
578             test_logger.error("Uso: python data_dictionary.py <arquivo_elf>")
579             sys.exit(1)
580
581         elf_path = Path(sys.argv[1])
582
583         try:
584             # Criar o dicionario de dados
585             data_dict = ElfDataDictionary(elf_path)
586

```

```
587     # Exemplo 1: Listar todas as variaveis
588     test_logger.info(
589         f"Total de variaveis encontradas: {len(data_dict.get_all_variables())}"
590     )
591
592     # Exemplo 2: Buscar variaveis relacionadas a pinos digitais
593     test_logger.info("Variaveis de pinos digitais:")
594     for name in data_dict.list_variables("*digital*pin*"):
595         info = data_dict.get_variable_info(name)
596         if info:
597             addr, size, var_type = info
598             test_logger.info(
599                 f" {name}: addr=0x{addr:04X}, size={size}, type={var_type}"
600             )
601
602     # Exemplo 3: Buscar variaveis de timer
603     test_logger.info("Variaveis de timer:")
604     for var in data_dict.search_variables("timer*"):
605         test_logger.info(
606             f" {var.name}: addr=0x{var.address:04X}, size={var.size}, type={var.
607             base_type}"
608         )
609
610     # Exemplo 4: Encontrar todas as variaveis float
611     test_logger.info("Variaveis float:")
612     for var in data_dict.search_variables("*", var_type="float"):
613         test_logger.info(f" {var.name}: addr=0x{var.address:04X}")
614
615     except FileNotFoundError as e:
616         test_logger.error(f"Erro: {e}")
617     except ValueError as e:
618         test_logger.error(f"Erro: {e}")
619
620 if __name__ == "__main__":
621     main()
```


APÊNDICE C – PROTOCOLO DESTRA INSTRUMENTADO

C.1 Implementação com Instrumentação de Performance

Este apêndice apresenta a versão instrumentada do protocolo DESTRA, que inclui funcionalidades adicionais para análise de performance em tempo real, coleta de métricas de execução e monitoramento de jitter.

C.1.1 Código Completo Instrumentado - `destra_protocol_test.ino`

Listing C.1 – Protocolo DESTRA com instrumentação para análise de performance

```

1  /*
2   * =====
3   * Arquivo: destra_protocol_test.ino
4   * Autor: Sandro Fadiga
5   * Instituicao: EESC - USP (Escola de Engenharia de Sao Carlos)
6   * Projeto: DESTRA - DEpurador de Sistemas em Tempo ReAl
7   * Data de Criacao: 09/01/2025
8   * Versao: 1.0
9   *
10  * Descricao:
11  *     Implementacao do protocolo DESTRA para Arduino.
12  *     Este arquivo contem toda a logica de comunicacao serial para
13  *     operacoes peek/poke, permitindo leitura e escrita de memoria
14  *     em tempo real para depuracao de sistemas embarcados.
15  *
16  * Protocolo:
17  *     - Palavras magicas: 0xCA 0xFE
18  *     - Comandos: PEEK (0xF1), POKE (0xF2)
19  *     - Enderecamento: 16 bits (0x0100 - 0x08FF para Arduino Uno)
20  *     - Tamanho de dados: 1-8 bytes por operacao
21  *     - Comunicacao: Serial 115200 baud, 8N1
22  *
23  * Funcionalidades:
24  *     - destraSetup(): Inicializa a comunicacao serial
25  *     - destraHandler(): Processa comandos recebidos (nao bloqueante)
26  *     - process.PeekRequest(): Le dados da memoria
27  *     - process.PokeRequest(): Escreve dados na memoria
28  *     - Maquina de estados para parsing de comandos
29  *     - Validacao de enderecos e tamanhos
30  *     - Echo de confirmacao para todos os bytes recebidos
31  *
32  * Formato dos Pacotes:
33  *     PEEK: [0xCA][0xFE][0xF1][ADDR_L][ADDR_H][SIZE]
34  *     POKE: [0xCA][0xFE][0xF2][ADDR_L][ADDR_H][SIZE][DATA...]
35  *
36  * Codigos de Status:
37  *     - 0x00: Sucesso
38  *     - 0x01: Erro de faixa de endereco
39  *     - 0x02: Erro de tamanho
40  *
41  * Requisitos:
42  *     - Arduino Uno ou compativel
43  *     - Memoria RAM acessivel: 0x0100 - 0x08FF

```

```

44 *
45 * Uso:
46 *   1. Incluir este arquivo no projeto Arduino
47 *   2. Chamar destraSetup() no setup()
48 *   3. Chamar destraHandler() no loop()
49 *
50 * Licenca: MIT
51 * =====
52 */
53
54 #include <Arduino.h>
55
56 // Constantes de comunicacao serial
57 #define SERIAL_START_MARKER 0xCAFE
58 #define CMD_PEEK 0xF1
59 #define CMD_POKE 0xF2
60 #define STATUS_SUCCESS 0x00
61 #define STATUS_ADDRESS_RANGE_ERROR 0x01
62 #define STATUS_SIZE_ERROR 0x02
63 #define BAUD_RATE 115200
64 #define BUFFER_SIZE 64 // Buffer para dados recebidos
65
66 // Estados da maquina de estados serial
67 enum DestraState {
68     WAIT_START_HIGH,
69     WAIT_START_LOW,
70     WAIT_COMMAND,
71     WAIT_ADDRESS_LOW,
72     WAIT_ADDRESS_HIGH,
73     WAIT_SIZE,
74     WAIT_VALUE,
75     PROCESS_REQUEST
76 };
77
78 // Variaveis de comunicacao serial
79 DestraState destraState = WAIT_START_HIGH;
80 uint8_t destraCommand = 0;
81 uint16_t destraAddress = 0; // Endereco de 16 bits
82 uint8_t destraSize = 0;
83 uint8_t addressLow = 0;
84 uint8_t addressHigh = 0;
85 uint8_t destraValueBuffer[8]; // Buffer para bytes de valor do POKE
86 uint8_t destraValueIndex = 0; // Indice para buffer de valor
87
88
89 // =====
90 // DEFINICOES E VARIAVEIS PARA ANALISE - INSTRUMENTACAO PROTOCOLO
91 // =====
92 // Comando especial para recuperar logs de performance
93 #define CMD_GET_PERF_LOG 0xF3
94
95 // VARIAVEIS DE PERFORMANCE
96 volatile unsigned long frameCounter = 0; // Contador de frames
97 volatile uint16_t frameRate = 0; // Frame rate de execucao
98 volatile uint16_t frameJitter = 0; // Jitter de processamento do loop
99 volatile uint16_t commandSequence = 0; // Identifica o comando
100 volatile unsigned long commandStartCounter = 0; // Contador de frame inicio do comando, 0
101 volatile unsigned long commandEndCounter = 0; // Contador de frame do fim do comando, 0

```

```

102 volatile unsigned long lastFrameTime = 0;           // Tempo do inicio do frame anterior
103 volatile unsigned long commandReceiveTime = 0;     // Tempo de recepcao do comando
104 volatile unsigned long commandProcessTime = 0;      // Tempo de processamento
105 volatile unsigned long lastDeltaTime = 0;           // Ultimo delta de tempo calculado
106
107 // PINOS DE DEBUG PARA OSCILOSCOPIO
108 #define PIN_TRIGGER_RX 2    // Pulso quando recebe comando
109 #define PIN_TRIGGER_TX 3    // Pulso quando envia resposta
110 #define PIN_FRAME_TOGGLE 4  // Toggle a cada loop()
111 #define PIN_BUSY 5          // Alto durante processamento
112
113 // Macros para facilitar uso dos pinos de debug
114 #define PULSE_RX() { digitalWrite(PIN_TRIGGER_RX, HIGH); delayMicroseconds(10);
115   digitalWrite(PIN_TRIGGER_RX, LOW); }
116 #define PULSE_TX() { digitalWrite(PIN_TRIGGER_TX, HIGH); delayMicroseconds(10);
117   digitalWrite(PIN_TRIGGER_TX, LOW); }
118 #define TOGGLE_FRAME() { digitalWrite(PIN_FRAME_TOGGLE, !digitalRead(PIN_FRAME_TOGGLE)); }
119
120 #define SET_BUSY(state) { digitalWrite(PIN_BUSY, state); }
121
122 // ESTRUTURA E BUFFER DE PERFORMANCE
123 #define PERF_BUFFER_SIZE 100
124 struct PerfLog
125 {
126     unsigned long frameCounter;           // Contador absoluto de frames
127     uint16_t frameRate;                 // Frame rate
128     uint16_t frameJitter;               // Diferenca entre frames consecutivos
129     uint16_t commandSequence;           // Identifica o comando
130     uint16_t commandFrameCounterDelta; // Distancia do frame executado do frame de inicio
131     do comando, 0 se mesmo frame
132     unsigned long commandProcessTime;   // Tempo de execucao do comando atual, 0 se nao
133     houver comando
134 };
135
136 PerfLog perfBuffer[PERF_BUFFER_SIZE];
137 uint8_t perfIndex = 0;
138 // -----
139 // FIM - DEFINICOES E VARIAVEIS PARA ANALISE - INSTRUMENTACAO PROTOCOLO
140 // -----
141
142 // =====
143 // FUNCOES ARDUINO INSTRUMENTADAS - SETUP E LOOP
144 // =====
145 void setup() {
146     // inicializar Teste
147     destraTestSetup();
148     // Inicializar DESTRA
149     destraSetup();
150 }
151
152 void loop() {
153     unsigned long currentFrameTime = micros();
154     unsigned long deltaTime = currentFrameTime - lastFrameTime; // duracao do frame
155     anterior
156
157     // Calcular framerate (Hz)
158     if (deltaTime > 0) {
159         frameRate = 1000000.0 / deltaTime;
160     }
161 }
```

```

156 // Calcular jitter (diferenca absoluta entre periodos consecutivos)
157 frameJitter = abs((long)deltaTime - (long)lastDeltaTime);
158 lastDeltaTime = deltaTime;
159
160 // Toggle do pino de frame
161 TOGGLE_FRAME();
162 frameCounter++;
163
164 // Processar comandos DESTRA
165 destraHandler();
166
167 // Executar calculos de exemplo
168 calculation();
169
170 // Ajustar para ~100Hz (10ms)
171 unsigned long elapsed = micros() - currentFrameTime;
172 if (elapsed < 10000) {
173     delayMicroseconds(10000 - elapsed);
174 }
175
176 lastFrameTime = currentFrameTime;
177 }
178
179 // Usaer em conjunto com Destra Setup
180 void destraTestSetup() {
181     // Configurar pinos de debug
182     pinMode(PIN_TRIGGER_RX, OUTPUT);
183     pinMode(PIN_TRIGGER_TX, OUTPUT);
184     pinMode(PIN_FRAME_TOGGLE, OUTPUT);
185     pinMode(PIN_BUSY, OUTPUT);
186
187     // Inicializar pinos em LOW
188     digitalWrite(PIN_TRIGGER_RX, LOW);
189     digitalWrite(PIN_TRIGGER_TX, LOW);
190     digitalWrite(PIN_FRAME_TOGGLE, LOW);
191     digitalWrite(PIN_BUSY, LOW);
192 }
193
194 // Destra Setup Original
195 void destraSetup() {
196     // Inicializar comunicacao serial
197     Serial.begin(BAUD_RATE);
198     // Aguardar conexao da porta serial (necessario para placas USB nativas)
199     while (!Serial) {
200         ; // Aguardar conexao da porta serial
201     }
202     destraState = WAIT_START_HIGH;
203 }
204
205 // Coloque-me no inicio do seu loop()
206 void destraHandler() {
207     // Funcao para lidar com comunicacao serial destra (nao bloqueante)
208     while (Serial.available() > 0 && destraState != PROCESS_REQUEST) {
209         uint8_t inByte = Serial.read();
210
211         // Maquina de Estados do Pacote/Requisicao
212         switch (destraState) {
213             // Pacote PEEK/POKE comeca com 0xCAFE
214             case WAIT_START_HIGH:
215                 if (inByte == 0xCA) {

```

```

216     int bytesAvailable = Serial.availableForWrite();
217     destraState = WAIT_START_LOW;
218
219     // // INTRUMENTACAO - Pulso RX no primeiro byte de um novo comando
220     PULSE_RX();
221     commandReceiveTime = micros();
222     commandStartCounter = frameCounter;
223     SET_BUSY(HIGH);
224
225 }
226 break;
227
228 case WAIT_START_LOW:
229     if (inByte == 0xFE) {
230         destraState = WAIT_COMMAND;
231     } else {
232         destraState = WAIT_START_HIGH;
233         SET_BUSY(LOW); // INTRUMENTACAO
234     }
235 break;
236
237 case WAIT_COMMAND:
238     destraCommand = inByte;
239     if (inByte == CMD_PEEK || inByte == CMD_POKE) {
240         destraState = WAIT_ADDRESS_LOW;
241     }
242     else if (inByte == CMD_GET_PERF_LOG) {
243         destraState = PROCESS_REQUEST;
244     }
245     else {
246         destraState = WAIT_START_HIGH;
247         SET_BUSY(LOW); // INTRUMENTACAO
248     }
249 break;
250
251 case WAIT_ADDRESS_LOW:
252     addressLow = inByte;
253     destraState = WAIT_ADDRESS_HIGH;
254 break;
255
256 case WAIT_ADDRESS_HIGH:
257     addressHigh = inByte;
258     // Combinar para endereco de 16 bits (little endian)
259     destraAddress = addressLow | (addressHigh << 8);
260     destraState = WAIT_SIZE;
261 break;
262
263 case WAIT_SIZE:
264     destraSize = inByte;
265     if (destraCommand == CMD_PEEK) {
266         destraState = PROCESS_REQUEST;
267     }
268     else if (destraCommand == CMD_POKE) {
269         destraValueIndex = 0; // Resetar indice do buffer de valor
270         destraState = WAIT_VALUE;
271     }
272     else {
273         destraState = WAIT_START_HIGH;
274         SET_BUSY(LOW); // INTRUMENTACAO
275     }

```

```

276         break;
277
278     case WAIT_VALUE:
279         // Verificar se o indice esta dentro dos limites do buffer
280         if (destraValueIndex < 8 && destraValueIndex < destraSize) {
281             // Armazenar o byte de valor
282             destraValueBuffer[destraValueIndex] = inByte;
283             destraValueIndex++;
284         }
285         // Verificar se recebemos todos os bytes de valor
286         if (destraValueIndex >= destraSize) {
287             destraState = PROCESS_REQUEST;
288         }
289         // Caso contrario, permanecer em WAIT_VALUE para coletar mais bytes
290         break;
291     }
292 }
293
294 // Processar a requisicao se tivermos uma mensagem completa
295 if (destraState == PROCESS_REQUEST) {
296     if (destraCommand == CMD_PEEK) {
297         process.PeekRequest();
298     } else if (destraCommand == CMD_POKE) {
299         process.PokeRequest();
300     }
301     else if (destraCommand == CMD_GET_PERF_LOG) {
302         process.GetPerfLog();
303     }
304     destraCommand = 0;
305     destraState = WAIT_START_HIGH; // Resetar para proxima requisicao
306
307     // INSTRUMENTACAO - Registrar tempo de processamento
308     SET_BUSY(LOW);
309     commandEndCounter = frameCounter;
310     commandProcessTime = micros() - commandReceiveTime;
311     commandSequence++;
312     registerPerformanceStats();
313 }
314 }
315
316
317 // Funcao para processar requisicao peek e enviar resposta
318 void process.PeekRequest() {
319     // INSTRUMENTACAO - Pulso TX antes de enviar resposta
320     PULSE_TX();
321
322     // Enviar cabecalho da resposta
323     Serial.write(0xCA);
324     Serial.write(0xFE);
325     Serial.write(CMD_PEEK);
326
327     // Validar faixa de endereco (verificacao de segurança opcional)
328     if (destraAddress < 0x100 || destraAddress > 0x8FF) { // Faixa de RAM do Arduino Uno
329         Serial.write(STATUS_ADDRESS_RANGE_ERROR);
330         return;
331     }
332
333     // Validar tamanho
334     if (destraSize == 0 || destraSize > 8) {
335         Serial.write(STATUS_SIZE_ERROR);

```

```

336     return;
337 }
338
339 Serial.write(STATUS_SUCCESS);
340
341 // Ler e enviar os dados solicitados
342 uint8_t* ptr = (uint8_t*)destraAddress;
343 for (uint8_t i = 0; i < destraSize; i++) {
344     Serial.write(ptr[i]);
345 }
346 }
347
348
349 // Funcao para processar requisicao poke e enviar resposta
350 void processPokeRequest() {
351     // INTRUMENTACAO - Pulso TX antes de enviar resposta
352     PULSE_TX();
353
354     // Enviar cabecalho da resposta
355     Serial.write(0xCA);
356     Serial.write(0xFE);
357     Serial.write(CMD_POKE);
358
359     // Validar faixa de endereco
360     if (destraAddress < 0x100 || destraAddress > 0x8FF) { // Faixa de RAM do Arduino Uno
361         Serial.write(STATUS_ADDRESS_RANGE_ERROR);
362         return;
363     }
364
365     // Validar tamanho (ja verificado na maquina de estados, mas dupla verificacao)
366     if (destraSize == 0 || destraSize > 8) {
367         Serial.write(STATUS_SIZE_ERROR);
368         return;
369     }
370
371     // Escrever os dados na memoria
372     uint8_t* ptr = (uint8_t*)destraAddress;
373     for (uint8_t i = 0; i < destraSize; i++) {
374         ptr[i] = destraValueBuffer[i];
375     }
376
377     // Enviar status de sucesso
378     Serial.write(STATUS_SUCCESS);
379
380     // Opcionalmente, ecoar de volta os dados escritos para verificacao
381     // Isso ajuda a confirmar que a escrita foi bem-sucedida
382     for (uint8_t i = 0; i < destraSize; i++) {
383         Serial.write(ptr[i]); // Ler de volta da memoria e enviar
384     }
385 }
386
387 // INTRUMENTACAO - Funcao para processar a requisicao de performance e enviar resposta
388 void sendPerfLogEntry(const PerfLog& e) {
389     Serial.write((uint8_t*)&e.frameCounter, 4);
390     Serial.write((uint8_t*)&e.frameRate, 2);
391     Serial.write((uint8_t*)&e.frameJitter, 2);
392     Serial.write((uint8_t*)&e.commandSequence, 2);
393     Serial.write((uint8_t*)&e.commandFrameCounterDelta, 2);
394     Serial.write((uint8_t*)&e.commandProcessTime, 4);
395 }

```

```
396
397 void processGetPerfLog() {
398     // Cabecalho
399     Serial.write(0xCA);
400     Serial.write(0xFE);
401     Serial.write(CMD_GET_PERF_LOG);
402     Serial.write(STATUS_SUCCESS);
403
404     // Número de entradas
405     Serial.write(perfIndex);
406
407     // Payload
408     for (uint8_t i = 0; i < perfIndex; i++) {
409         sendPerfLogEntry(perfBuffer[i]);
410     }
411
412     // Reset
413     perfIndex = 0;
414 }
415
416 // INSTRUMENTAÇÃO - Função para REGISTRAR ESTATÍSTICAS DE PERFORMANCE
417 void registerPerformanceStats() {
418     if (perfIndex < 99) {
419         perfBuffer[perfIndex] = { frameCounter, frameRate, frameJitter, commandSequence, (
420             uint16_t)(commandEndCounter-commandStartCounter), commandProcessTime };
421         perfIndex = (perfIndex + 1) % PERF_BUFFER_SIZE;
422     }
423 }
```

C.2 Funcionalidades de Instrumentação

C.2.1 Variáveis de Análise de Performance

A versão instrumentada adiciona as seguintes variáveis para coleta de métricas:

- **frameCounter**: Contador absoluto de frames/loops executados
- **frameRate**: Taxa de execução do loop principal em Hz
- **frameJitter**: Variação no tempo entre frames consecutivos
- **commandSequence**: Identificador sequencial de comandos recebidos
- **commandStartCounter/EndCounter**: Frames de início e fim de processamento
- **commandProcessTime**: Tempo total de processamento do comando em microssegundos

C.2.2 Pinos de Debug para Osciloscópio

Para análise com osciloscópio, foram definidos pinos de saída digital:

- **PIN_TRIGGER_RX (2)**: Pulso quando recebe comando

- **PIN_TRIGGER_TX (3):** Pulso quando envia resposta
- **PIN_FRAME_TOGGLE (4):** Toggle a cada execução do loop
- **PIN_BUSY (5):** Nível alto durante processamento de comando

C.2.3 Buffer de Performance

O sistema mantém um buffer circular de 100 entradas com as seguintes informações por comando:

Listing C.2 – Estrutura PerfLog para dados de performance

```
struct PerfLog {
    unsigned long frameCounter;           // Contador absoluto de frames
    uint16_t frameRate;                 // Frame rate
    uint16_t frameJitter;               // Diferenca entre frames consecutivos
    uint16_t commandSequence;           // Identifica o comando
    uint16_t commandFrameCounterDelta; // Distancia em frames do inicio ao comando
    unsigned long commandProcessTime;   // Tempo de execucao em microsegundos
};
```

C.2.4 Comando Adicional de Performance

Foi implementado o comando **CMD_GET_PERF_LOG** (0xF3) que permite baixar todos os dados de performance coletados:

- **Formato:** [0xCA][0xFE][0xF3]
- **Resposta:** Cabeçalho + número de entradas + dados serializados
- **Funcionalidade:** Transmite buffer completo e o reseta

C.3 Integração com Sistema de Teste

C.3.1 Loop Principal Instrumentado

O loop principal foi modificado para incluir:

1. Medição precisa de tempo de frame usando **micros()**
2. Cálculo em tempo real de frame rate e jitter
3. Sinalização visual através dos pinos de debug
4. Controle de frequência para manter 100Hz
5. Função de exemplo **calculation()** para simular carga de trabalho

C.3.2 Uso da Instrumentação

Para utilizar a versão instrumentada:

1. Conectar osciloscópio aos pinos 2-5 para análise temporal
2. Usar a ferramenta host para coletar dados via comando 0xF3
3. Executar testes de latência, throughput e jitter
4. Analisar métricas coletadas para otimização de performance

C.4 Especificações Técnicas da Instrumentação

C.4.1 Precisão Temporal

- **Resolução:** 4 microssegundos (função `micros()` do Arduino)
- **Overflow:** Aproximadamente 70 minutos
- **Jitter medido:** Diferença absoluta entre períodos consecutivos

C.4.2 Buffer de Performance

- **Tamanho:** 100 entradas (configurável via `PERF_BUFFER_SIZE`)
- **Tipo:** Buffer circular com reset automático
- **Payload:** 16 bytes por entrada
- **Capacidade total:** 1600 bytes de dados de performance

C.4.3 Sinais de Debug

- **Duração do pulso:** 10 microssegundos
- **Nível lógico:** 0V/5V compatível com osciloscópio
- **Frequência do toggle:** 100Hz (acompanha o loop principal)

APÊNDICE D – TESTES DE PERFORMANCE E ANALISE

D.1 Metodologia de Teste

Este apêndice apresenta os testes de performance realizados no protocolo DESTRA, incluindo análise de latência, throughput, jitter e comportamento sob diferentes cargas de trabalho.

D.1.1 Configuração do Ambiente de Teste

- **Hardware:** Arduino Uno R3 (ATmega328P a 16MHz)
- **Comunicação:** Serial 115200 baud, 8N1
- **Host:** Python 3.9+ com PySide6 e pyserial
- **Instrumentação:** Osciloscópio digital FNIRSI DSO153
- **Pinos de debug:** 2, 3, 4, 5 (conforme especificação)

D.2 Script Performance Test Completo

Listing D.1 – Script de testes de performance - performance_tests.py

```

1  #!/usr/bin/env python3
2  # -*- coding: utf-8 -*-
3  """
4  Arquivo: performance_tests.py
5  Autor: Sandro Fadiga
6  Instituição: EESC - USP (Escola de Engenharia de São Carlos)
7  Projeto: DESTRA - DEPURADOR de Sistemas em Tempo Real
8  Data de Criação: 09/01/2025
9  Versão: 1.0
10
11 Descricao:
12     Módulo de testes de performance para análise de latência, jitter e throughput
13     do protocolo DESTRA. Gera métricas estatísticas e visualizações para análise
14     de desempenho em tempo real.
15
16 Funcionalidades:
17     - Medição de latência (Round-Trip Time)
18     - Cálculo de jitter
19     - Análise estatística (média, desvio padrão, percentis)
20     - Geração de gráficos
21     - Exportação de dados para análise externa
22     - Testes de stress e carga
23
24 Dependências:
25     - numpy: Cálculos estatísticos
26     - matplotlib: Visualização de dados
27     - pandas: Manipulação de dados
28     - destra: Protocolo de comunicação

```

```

29
30 Licenca: MIT
31 """
32 import sys
33 from datetime import datetime
34 import re
35 import time
36 import numpy as np
37 import pandas as pd
38 import matplotlib.pyplot as plt
39 from typing import List, Dict, Tuple, Optional
40 import json
41 from pathlib import Path
42 from typing import List, Dict
43
44 from destra import DestraProtocol
45 from logger_config import DestraLogger
46
47
48 class PerformanceMetrics:
49     """Classe para coletar e analisar metricas de performance"""
50
51     def __init__(self):
52         self.latencies: List[float] = []
53         self.timestamps: List[float] = []
54         self.jitter_values: List[float] = []
55         self.errors: List[Dict[str, any]] = []
56         self.test_start_time = None
57         self.test_end_time = None
58
59         # Configurar logger
60         logger_manager = DestraLogger()
61         self.logger = logger_manager.logger.getChild("Performance")
62
63     def add_measurement(
64         self,
65         latency: float,
66         timestamp: float,
67         success: bool = True,
68         error_msg: str = "",
69     ):
70         """Adicionar uma medicao de latencia"""
71         self.latencies.append(latency)
72         self.timestamps.append(timestamp)
73
74         # Calcular jitter (variacao entre latencias consecutivas)
75         if len(self.latencies) > 1:
76             jitter = abs(self.latencies[-1] - self.latencies[-2])
77             self.jitter_values.append(jitter)
78
79         if not success:
80             self.errors.append(
81                 {"timestamp": timestamp, "latency": latency, "error": error_msg}
82             )
83
84     def calculate_statistics(self) -> Dict[str, any]:
85         """Calcular estatisticas das medicoes"""
86         if not self.latencies:
87             return {}
88

```

```

89     latencies_ms = [l * 1000 for l in self.latencies] # Converter para ms
90     jitter_ms = [j * 1000 for j in self.jitter_values] if self.jitter_values else []
91
92     stats = {
93         "total_de_medidas": len(self.latencies),
94         "medidas_bem_sucedidas": len(self.latencies) - len(self.errors),
95         "taxa_de_erro": len(self.errors) / len(self.latencies) * 100
96         if self.latencies
97         else 0,
98         "latencia": {
99             "media_ms": float(np.mean(latencies_ms)),
100            "mediana_ms": float(np.median(latencies_ms)),
101            "desvio_padrao_ms": float(np.std(latencies_ms)),
102            "min_ms": float(np.min(latencies_ms)),
103            "max_ms": float(np.max(latencies_ms)),
104            "p95_ms": float(np.percentile(latencies_ms, 95)),
105            "p99_ms": float(np.percentile(latencies_ms, 99)),
106        },
107    }
108
109    if jitter_ms:
110        stats["jitter"] = {
111            "media_ms": float(np.mean(jitter_ms)),
112            "mediana_ms": float(np.median(jitter_ms)),
113            "desvio_padrao_ms": float(np.std(jitter_ms)),
114            "min_ms": float(np.min(jitter_ms)),
115            "max_ms": float(np.max(jitter_ms)),
116        }
117
118    return stats
119
120 def export_to_csv(self, filename: str):
121     """Exportar dados para arquivo CSV"""
122     df = pd.DataFrame(
123         {
124             "tempo": self.timestamps,
125             "latencia_ms": [l * 1000 for l in self.latencies],
126             "jitter_ms": [j * 1000 for j in self.jitter_values]
127             + [0], # Adicionar 0 para primeira medicao
128         }
129     )
130     df.to_csv(filename, index=False)
131     self.logger.info(f"Dados exportados para {filename}")
132
133 def plot_results(self, save_path: Optional[str] = None):
134     """Gerar graficos de visualizacao"""
135     fig, axes = plt.subplots(2, 2, figsize=(12, 8))
136
137     # Grafico 1: Latencia ao longo do tempo
138     axes[0, 0].plot(
139         self.timestamps, [l * 1000 for l in self.latencies], "b-", alpha=0.7
140     )
141     axes[0, 0].set_xlabel("Tempo (s)")
142     axes[0, 0].set_ylabel("Latencia (ms)")
143     axes[0, 0].set_title("Latencia ao Longo do Tempo")
144     axes[0, 0].grid(True, alpha=0.3)
145
146     # Grafico 2: Histograma de latencia
147     axes[0, 1].hist(
148         [l * 1000 for l in self.latencies], bins=50, edgecolor="black", alpha=0.7

```

```

149     )
150     axes[0, 1].set_xlabel("Latencia (ms)")
151     axes[0, 1].set_ylabel("Frequencia")
152     axes[0, 1].set_title("Distribuicao de Latencia")
153     axes[0, 1].grid(True, alpha=0.3)
154
155     # Grafico 3: Jitter ao longo do tempo
156     if self.jitter_values:
157         axes[1, 0].plot(
158             self.timestamps[1:],
159             [j * 1000 for j in self.jitter_values],
160             "r-",
161             alpha=0.7,
162         )
163     axes[1, 0].set_xlabel("Tempo (s)")
164     axes[1, 0].set_ylabel("Jitter (ms)")
165     axes[1, 0].set_title("Jitter ao Longo do Tempo")
166     axes[1, 0].grid(True, alpha=0.3)
167
168     # Grafico 4: Boxplot comparativo
169     data_to_plot = [[l * 1000 for l in self.latencies]]
170     labels = ["Latencia"]
171     if self.jitter_values:
172         data_to_plot.append([j * 1000 for j in self.jitter_values])
173         labels.append("Jitter")
174     axes[1, 1].boxplot(data_to_plot, tick_labels=labels)
175     axes[1, 1].set_ylabel("Tempo (ms)")
176     axes[1, 1].set_title("Analise Estatistica")
177     axes[1, 1].grid(True, alpha=0.3)
178
179     plt.tight_layout()
180
181     if save_path:
182         plt.savefig(save_path, dpi=300, bbox_inches="tight")
183         self.logger.info(f"Graficos salvos em {save_path}")
184     else:
185         plt.show()
186     return fig
187
188
189 class PerformanceTester:
190     """Classe principal para executar testes de performance"""
191
192     def __init__(self, port: str = None, baudrate: int = 115200):
193         self.protocol = DestraProtocol(port=port, baudrate=baudrate)
194         self.metrics = PerformanceMetrics()
195
196         # Configurar logger
197         logger_manager = DestraLogger()
198         self.logger = logger_manager.logger.getChild("Tester")
199
200     def connect(self) -> bool:
201         """Conectar ao Arduino"""
202         return self.protocol.connect()
203
204     def disconnect(self):
205         """Desconectar do Arduino"""
206         self.protocol.disconnect()
207
208     def test_single_peek(self, address: int, size: int) -> Tuple[float, bool]:

```

```

209     """Testar um unico comando peek e medir latencia"""
210     start_time = time.perf_counter()
211
212     try:
213         data = self.protocol.peek(address, size)
214         end_time = time.perf_counter()
215         latency = end_time - start_time
216         success = data is not None
217
218         return latency, success
219     except Exception as e:
220         end_time = time.perf_counter()
221         latency = end_time - start_time
222         self.logger.error(f"Erro no peek: {e}")
223         return latency, False
224
225     def test_single_poke(
226         self, address: int, size: int, value: int
227     ) -> Tuple[float, bool]:
228         """Testar um unico comando poke e medir latencia"""
229         start_time = time.perf_counter()
230
231         try:
232             success = self.protocol.poke(address, size, value)
233             end_time = time.perf_counter()
234             latency = end_time - start_time
235
236             return latency, success
237         except Exception as e:
238             end_time = time.perf_counter()
239             latency = end_time - start_time
240             self.logger.error(f"Erro no poke: {e}")
241             return latency, False
242
243     def run_latency_test(
244         self, num_samples: int = 100, address: int = 0x0100, size: int = 4
245     ) -> Dict[str, any]:
246         """Executar teste de latencia com multiplas amostras"""
247         self.logger.info(f"Iniciando teste de latencia: {num_samples} amostras")
248         self.metrics = PerformanceMetrics() # Reset metrics
249         self.metrics.test_start_time = time.time()
250
251         for i in range(num_samples):
252             timestamp = time.time() - self.metrics.test_start_time
253             latency, success = self.test_single_peek(address, size)
254             self.metrics.add_measurement(latency, timestamp, success)
255
256             # Log de progresso a cada 10 amostras
257             if (i + 1) % 10 == 0:
258                 self.logger.debug(f"Progresso: {i + 1}/{num_samples}")
259
260             self.metrics.test_end_time = time.time()
261             stats = {}
262             stats["teste"] = "Latencia"
263             stats.update(self.metrics.calculate_statistics())
264
265             self.logger.info("Teste de latencia concluido")
266
267             return stats
268

```

```

269     def run_stress_test(
270         self,
271         duration_seconds: int = 60,
272         frequency_hz: int = 100,
273         address: int = 0x0100,
274         size: int = 4,
275     ) -> Dict[str, any]:
276         """Executar teste de stress com alta frequencia de comandos"""
277         self.logger.info(
278             f"Iniciando teste de stress: {duration_seconds}s @ {frequency_hz}Hz"
279         )
280         self.metrics = PerformanceMetrics() # Reset metrics
281         self.metrics.test_start_time = time.time()
282
283         interval = 1.0 / frequency_hz
284         end_time = time.time() + duration_seconds
285
286         while time.time() < end_time:
287             timestamp = time.time() - self.metrics.test_start_time
288             latency, success = self.test_single_peek(address, size)
289             self.metrics.add_measurement(latency, timestamp, success)
290
291             # Aguardar para manter a frequencia
292             time.sleep(max(0, interval - latency))
293
294             self.metrics.test_end_time = time.time()
295             stats = {}
296             stats["teste"] = "Estress"
297             stats.update(self.metrics.calculate_statistics())
298
299             self.logger.info("Teste de stress concluido")
300             return stats
301
302     def run_burst_test(
303         self,
304         burst_size: int = 10,
305         num_bursts: int = 10,
306         delay_between_bursts: float = 1.0,
307         address: int = 0x0100,
308         size: int = 4,
309     ) -> Dict[str, any]:
310         """Executar teste de burst - rajadas de comandos"""
311         self.logger.info(
312             f"Iniciando teste de burst: {num_bursts} bursts de {burst_size} comandos"
313         )
314         self.metrics = PerformanceMetrics() # Reset metrics
315         self.metrics.test_start_time = time.time()
316
317         for burst_num in range(num_bursts):
318             # Executar burst
319             for _ in range(burst_size):
320                 timestamp = time.time() - self.metrics.test_start_time
321                 latency, success = self.test_single_peek(address, size)
322                 self.metrics.add_measurement(latency, timestamp, success)
323
324             # Delay entre bursts
325             if burst_num < num_bursts - 1:
326                 time.sleep(delay_between_bursts)
327
328         self.metrics.test_end_time = time.time()

```

```

329     stats = {}
330     stats["teste"] = "Burst"
331     stats.update(self.metrics.calculate_statistics())
332
333     self.logger.info("Teste de burst concluido")
334
335     return stats
336
337 def dump_embedded_performance_data(self, associated_test: str) -> dict:
338     self.logger.info(f"Iniciando download de dados de performance para: {associated_test}")
339     payload = self.protocol.performance()
340     #print(payload)
341     # converte para formato texto a ser usado com regex
342     payload_csv = "\n".join([str(p) for p in payload])
343
344     # --- 1. Parse do payload ---
345     pattern = re.compile(
346         r"frame_counter:\s*(\d+),\s*frame_rate:(\d+),\s*frame_jitter_us:(\d+),\s*
347         command_sequence:(\d+),\s*command_counter_delta:(\d+),\s*
348         command_process_time_us:(\d+)"
349     )
350
351     data = [tuple(map(int, m.groups())) for m in pattern.finditer(payload_csv)]
352     if not data:
353         self.logger.warning("Nenhum dado de performance encontrado no payload.")
354         return None
355
356     arr = np.array(data, dtype=int)
357     frame_counter = arr[:, 0]
358     frame_rate = arr[:, 1]
359     frame_jitter_us = arr[:, 2]
360     command_sequence = arr[:, 3]
361     command_counter_delta = arr[:, 4]
362     command_process_time_us = arr[:, 5]
363
364     # --- 2. Conversao mili sec -> micro sec ---
365     frame_jitter_ms = frame_jitter_us / 1000.0
366     command_process_time_ms = command_process_time_us / 1000.0
367
368     # --- 3. Estatisticas ---
369     def basic_stats(values_ms):
370         return {
371             "media_ms": float(np.mean(values_ms)),
372             "mediana_ms": float(np.median(values_ms)),
373             "desvio_padrao_ms": float(np.std(values_ms)),
374             "min_ms": float(np.min(values_ms)),
375             "max_ms": float(np.max(values_ms)),
376             "p95_ms": float(np.percentile(values_ms, 95)),
377             "p99_ms": float(np.percentile(values_ms, 99)),
378         }
379
380     stats = {
381         "dados_embarcados_para_teste_de": associated_test,
382         "total_de_amostras": len(frame_counter),
383         "frame_jitter": basic_stats(frame_jitter_ms),
384         "frame_rate": {
385             "media_fps": float(np.mean(frame_rate)),
386             "mediana_fps": float(np.median(frame_rate)),
387             "desvio_padrao_ms": float(np.std(frame_rate)),
388         }
389     }

```

```

386         "min_fps": float(np.min(frame_rate)),
387         "max_fps": float(np.max(frame_rate)),
388     },
389     "command_process_time": basic_stats(command_process_time_ms),
390 }
391
392 # --- 4. Analise de sequencia / desvio ---
393 def find_sequence_anomalies(seq):
394     diffs = np.diff(seq)
395     expected = 1
396     anomalies = np.where(diffs != expected)[0]
397     return anomalies.tolist() if len(anomalies) > 0 else 0
398
399 sequence_issues = {
400     "gaps_frame_counter": find_sequence_anomalies(frame_counter),
401     "gaps_command_sequence": find_sequence_anomalies(command_sequence),
402     "anomalias_command_counter_delta": np.where(command_counter_delta != 0)[0].
403         tolist()
404         if np.any(command_counter_delta != 0) else 0,
405     }
406
407     stats["analise_de_sequencia"] = sequence_issues
408
409 def genetate_reports(test_name: str, tester_ref, test_dump, embedded_dump):
410     curr_dir = Path.cwd()
411     test_results_path = curr_dir.parent / Path("tests")
412
413     # Teste de latencia
414     # Criar arquivo Markdown
415     timestamp_str = datetime.now().strftime("%Y%m%d_%H%M%S")
416     report_file = test_results_path / Path(f"{test_name}_{timestamp_str}.md")
417
418     with open(report_file.resolve(), "w", encoding="utf-8") as file:
419         file.write(f"# Relatorio de Testes {test_name}\n\n")
420         file.write("## Dados de Performance Externos\n")
421         file.write("```json\n")
422         file.write(json.dumps(test_dump, indent=2))
423         file.write("\n```\n")
424         file.write("## Dados de Performance Embarcada\n")
425         file.write("```json\n")
426         file.write(json.dumps(embedded_dump, indent=2))
427         file.write("\n```\n")
428
429     plot_file = test_results_path / Path(f"{test_name}_{timestamp_str}.png")
430     tester_ref.metrics.plot_results(plot_file)
431
432
433 def main():
434     """Funcao principal para executar testes de performance e gerar relatorio Markdown"""
435
436     # Verificar argumentos
437     port = "COM5"
438     if len(sys.argv) > 1:
439         port = sys.argv[1]
440
441     # Criar tester
442     tester = PerformanceTester(port=port)
443
444     # Conectar

```

```

445     if not tester.connect():
446         print("**Falha ao conectar ao Arduino!**\n")
447         return
448
449     try:
450         # Teste de latencia
451         stats = tester.run_latency_test(num_samples=100)
452         embed = tester.dump_embedded_performance_data("Latencia")
453         genetate_reports("Latencia", tester, stats, embed)
454
455         # Teste de stress
456         stats = tester.run_stress_test(duration_seconds=10)
457         embed = tester.dump_embedded_performance_data("Estresse")
458         genetate_reports("Estresse", tester, stats, embed)
459
460         # Teste de burst
461         stats = tester.run_burst_test()
462         embed = tester.dump_embedded_performance_data("Burst")
463         genetate_reports("Burst", tester, stats, embed)
464
465     except KeyboardInterrupt:
466         print("\n**Teste interrompido pelo usuario**\n")
467     except Exception as e:
468         import traceback
469         traceback.print_exc()
470         print(f"\n**Erro durante o teste: {e}**\n")
471     finally:
472         tester.disconnect()
473
474 if __name__ == "__main__":
475     main()

```

D.2.1 Teste de Latencia

Listing D.2 – Funcao de teste de latencia - def run_latency_test

```

1 def run_latency_test(
2     self, num_samples: int = 100, address: int = 0x0100, size: int = 4
3 ) -> Dict[str, any]:
4     """Executar teste de latencia com multiplas amostras"""
5     self.logger.info(f"Iniciando teste de latencia: {num_samples} amostras")
6     self.metrics = PerformanceMetrics()  # Reset metrics
7     self.metrics.test_start_time = time.time()
8
9     for i in range(num_samples):
10        timestamp = time.time() - self.metrics.test_start_time
11        latency, success = self.test_single_peek(address, size)
12        self.metrics.add_measurement(latency, timestamp, success)
13
14        # Log de progresso a cada 10 amostras
15        if (i + 1) % 10 == 0:
16            self.logger.debug(f"Progresso: {i + 1}/{num_samples}")
17
18        self.metrics.test_end_time = time.time()
19        stats = {}
20        stats["teste"] = "Latencia"
21        stats.update(self.metrics.calculate_statistics())
22
23        self.logger.info("Teste de latencia concluido")

```

```
24
25     return stats
```

D.2.2 Teste de Estresse

Listing D.3 – Funcao de teste de estresse - def run_stress_test

```
1
2     def run_stress_test(
3         self,
4             duration_seconds: int = 60,
5             frequency_hz: int = 100,
6             address: int = 0x0100,
7             size: int = 4,
8         ) -> Dict[str, any]:
9             """Executar teste de stress com alta frequencia de comandos"""
10            self.logger.info(
11                f"Iniciando teste de stress: {duration_seconds}s @ {frequency_hz}Hz"
12            )
13            self.metrics = PerformanceMetrics() # Reset metrics
14            self.metrics.test_start_time = time.time()
15
16            interval = 1.0 / frequency_hz
17            end_time = time.time() + duration_seconds
18
19            while time.time() < end_time:
20                timestamp = time.time() - self.metrics.test_start_time
21                latency, success = self.test_single_peek(address, size)
22                self.metrics.add_measurement(latency, timestamp, success)
23
24                # Aguardar para manter a frequencia
25                time.sleep(max(0, interval - latency))
26
27            self.metrics.test_end_time = time.time()
28            stats = {}
29            stats["teste"] = "Estress"
30            stats.update(self.metrics.calculate_statistics())
31
32            self.logger.info("Teste de stress concluido")
33            return stats
```

D.2.3 Teste de Rajada

Listing D.4 – Funcao de teste de rajada - def run_burst_test

```
1
2     def run_burst_test(
3         self,
4             burst_size: int = 10,
5             num_bursts: int = 10,
6             delay_between_bursts: float = 1.0,
7             address: int = 0x0100,
8             size: int = 4,
9         ) -> Dict[str, any]:
10            """Executar teste de burst - rajadas de comandos"""
11            self.logger.info(
12                f"Iniciando teste de burst: {num_bursts} bursts de {burst_size} comandos"
13            )
```

```

14     self.metrics = PerformanceMetrics() # Reset metrics
15     self.metrics.test_start_time = time.time()
16
17     for burst_num in range(num_bursts):
18         # Executar burst
19         for _ in range(burst_size):
20             timestamp = time.time() - self.metrics.test_start_time
21             latency, success = self.test_single_peek(address, size)
22             self.metrics.add_measurement(latency, timestamp, success)
23
24         # Delay entre bursts
25         if burst_num < num_bursts - 1:
26             time.sleep(delay_between_bursts)
27
28     self.metrics.test_end_time = time.time()
29     stats = {}
30     stats["teste"] = "Burst"
31     stats.update(self.metrics.calculate_statistics())
32
33     self.logger.info("Teste de burst concluido")
34
35     return stats

```

D.3 Analise de Resultados

D.3.1 Resultados de Latencia

Os testes de latencia demonstraram:

- **Latencia media:** 15.2 ms para operacoes peek
- **Latencia minima:** 12.8 ms
- **Latencia maxima:** 28.4 ms
- **Desvio padrao:** 2.3 ms
- **Taxa de sucesso:** 99.8%

D.3.2 Resultados de Estresse

Tabela 26 – Resultados de throughput por frequencia

Frequencia Alvo	Frequencia Real	Taxa de Sucesso	Comandos/s
1 Hz	1.0 Hz	100.0%	1.0
10 Hz	9.8 Hz	99.9%	9.8
50 Hz	48.2 Hz	98.5%	47.5
100 Hz	89.3 Hz	94.2%	84.1

Tabela 27 – Resultados de teste de rajada

Tamanho da Rajada	Tempo Medio (s)	Throughput (cmd/s)	Taxa de Sucesso
10 comandos	0.156	64.1	100.0%
50 comandos	0.782	63.9	99.8%
100 comandos	1.564	63.9	99.2%
200 comandos	3.128	63.9	98.8%

D.3.3 Resultados de Rajada

D.4 Analise de Jitter com Osciloscopio

D.4.1 Configuracao da Medicao

Para analise de jitter temporal, foram utilizados os pinos de debug conectados ao osciloscopio:

- **Canal 1:** PIN_FRAME_TOGGLE (4) - Frequencia do loop
- **Canal 2:** PIN_BUSY (5) - Duracao do processamento
- **Canal 3:** PIN_TRIGGER_RX (2) - Recepcao de comandos
- **Canal 4:** PIN_TRIGGER_TX (3) - Transmissao de respostas

D.4.2 Medicoes de Jitter

As medicoes revelaram:

- **Jitter do frame base:** $\pm 50 \mu s$ (sem comandos ativos)
- **Jitter com peek:** $\pm 120 \mu s$ (durante operacoes peek)
- **Jitter com poke:** $\pm 150 \mu s$ (durante operacoes poke)
- **Tempo de processamento:** 80-200 μs por comando
- **Periodo do frame:** 10.0 ms (100 Hz nominal)

D.5 Conclusoes dos Testes

D.5.1 Performance do Sistema

O protocolo DESTRA demonstrou:

1. **Latencia consistente:** Tempo de resposta previsivel e baixo jitter
2. **Alto throughput:** Capaz de processar ate 64 comandos/segundo

3. **Confiabilidade:** Taxa de sucesso superior a 98% em todas as condições
4. **Escalabilidade:** Performance mantida em rajadas de até 200 comandos

D.5.2 Limitações Identificadas

- **Saturação serial:** Throughput limitado pela velocidade da comunicação serial
- **Buffer overhead:** Processamento adicional para comandos longos
- **Jitter aumentado:** Variância temporal durante processamento intensivo

D.5.3 Recomendações

Para otimizar o desempenho:

1. Manter frequência de comandos abaixo de 50 Hz para melhor confiabilidade
2. Implementar buffering no lado host para rajadas grandes
3. Considerar baudrizes superiores para aplicações de alta frequência
4. Monitorar jitter em aplicações críticas de tempo real

APÊNDICE E – DIAGRAMA DE CLASSES DA APLICAÇÃO DESTRA UI

Este apêndice apresenta o diagrama de classes UML da aplicação DESTRA UI, mostrando a arquitetura do sistema e os relacionamentos entre os principais componentes.

E.1 Visão Geral da Arquitetura

A aplicação DESTRA UI foi desenvolvida seguindo uma arquitetura em camadas, separando claramente as responsabilidades:

- **Camada de Interface:** Responsável pela interação com o usuário (DestraGUI)
 - **Camada de Protocolo:** Gerencia a comunicação serial com o Arduino (DestraProtocol)
 - **Camada de Análise:** Processa arquivos ELF e extrai informações de variáveis (ElfDataDictionary)
 - **Camada de Dados:** Define estruturas para armazenar informações (VariableInfo, PerformanceData)

E.2 Diagrama de Classes



Figura 19 – Diagrama de Classes da Aplicação DESTRA UI

E.3 Descrição das Classes

E.3.1 DestraGUI

Classe principal da interface gráfica, herda de QMainWindow. Responsável por:

- Gerenciar a interface do usuário com PySide6/Qt
 - Coordenar as operações entre protocolo e análise ELF
 - Implementar funcionalidades de auto-peek
 - Gerenciar tabelas de variáveis disponíveis e selecionadas

E.3.2 DestraProtocol

Implementa o protocolo de comunicação serial com o Arduino:

- Define comandos PEEK, POKE e PERFORMANCE
- Gerencia conexão serial e detecção automática de portas
- Processa respostas do protocolo com verificação de integridade
- Decodifica tipos de dados recebidos

E.3.3 ElfDataDictionary

Analisador de arquivos ELF com informações DWARF:

- Extrai informações de variáveis (nome, endereço, tipo, tamanho)
- Processa estruturas, arrays e tipos básicos
- Fornece funcionalidades de busca e filtragem
- Gera estatísticas do arquivo analisado

E.3.4 VariableInfo

Estrutura de dados (dataclass) que armazena informações de uma variável:

- Nome da variável
- Endereço de memória
- Tamanho em bytes
- Tipo base (uint8, uint16, uint32, etc.)
- Informações sobre sinalização e ponteiros

E.3.5 DecodedTypes

Classe utilitária para decodificação de tipos de dados:

- Mapeia tipos de dados para formatos struct
- Fornece informações de tamanho por tipo
- Lista tipos suportados pelo sistema

E.4 Padrões de Design Utilizados

E.4.1 Singleton

A classe `DestraLogger` implementa o padrão Singleton para garantir uma instância única de configuração de logging em toda a aplicação.

E.4.2 Model-View-Controller (MVC)

A arquitetura segue parcialmente o padrão MVC:

- **Model:** `ElfDataDictionary`, `VariableInfo`, `PerformanceData`
- **View:** `DestraGUI` e componentes Qt (QTableWidget, QComboBox, etc.)
- **Controller:** Métodos da `DestraGUI` que coordenam Model e View

E.4.3 Data Transfer Object (DTO)

As classes `VariableInfo` e `PerformanceData` funcionam como DTOs, transportando dados estruturados entre diferentes camadas da aplicação.

E.4.4 Facade

A classe `DestraProtocol` atua como uma facade, simplificando o acesso ao protocolo de comunicação serial complexo.