UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

Murilo Henrique Pasini Trevisan

Depuração de sistemas embarcados em tempo real com uma abordagem não intrusiva

São Carlos
2025

Murilo Henrique Pasini Trevisan

Depuração de sistemas embarcados em tempo real com uma abordagem não intrusiva

Monografia apresentada ao Curso de Engenharia elétrica – ênfase em eletrônica, da Escola de Engenharia de São Carlos da Universidade de São Paulo, como parte dos requisitos para obtenção do título de Engenheiro Eletricista.

Orientador(a): Prof. Dr. Pedro de Oliveira Conceição Junior

São Carlos
2025

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

Trevisan, Murilo Henrique Pasini
T814d Depuração de sistemas embarcados em tempo real com uma abordagem não intrusiva / Murilo Henrique Pasini Trevisan; orientador Pedro de Oliveira Conceição Junior. São Carlos, 2025.

Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2025.

1. Sistemas embarcados. 2. Depuração. 3. Tempo real. 4. Não intrusiva. 5. Ferramentas. I. Titulo.

Eduardo Graziosi Silva - CRB - 8/8907

FOLHA DE APROVAÇÃO

Nome: Murilo Henrique Pasini Trevisan

Título: "Depuração de sistemas embarcados em tempo real com uma abordagem não intrusiva"

Trabalho de Conclusão de Curso defendido e aprovado em 27 / 11 / 2025,

com NOTA 10 (DEZ), pela Comissão Julgadora:

Prof. Dr. Pedro de Oliveira Conceição Junior - Orientador - SEL/EESC/USP

Prof. Dr. Maximiliam Luppe - SEL/EESC/USP

Dr. Leonardo Mariano Gomes - EESC/SEL

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Professor Associado José Carlos de Melo Vieira Júnior

Dedico este trabalho à minha família, cujo apoio incondicional foi fundamental durante os desafios acadêmicos. Ao meu orientador, Prof. Dr. Pedro de Oliveira Conceição Junior, pela orientação técnica e paciência durante o desenvolvimento deste projeto. E a todos os engenheiros que diariamente enfrentam os desafios silenciosos da depuração de sistemas embarcados.

AGRADECIMENTOS

Agradeço primeiramente a minha família e todo o seu suporte no meu desenvolvimento que culmina ao ponto que estou na elaboração deste trabalho.
A minha irmã, que me auxiliou com idéias e suporte para elaboração do texto deste projeto.
Ao meu orientador, Prof Pedro de Oliveira Conceição Junior, que me auxiliou no desenvolvimento deste trabalho e me orientou sobre as melhores práticas para a entrega do projeto de conclusão de curso.
A todos que me acompanharam no meu processo de desenvolvimento e me auxiliaram a alcançar meus objetivos.

"O perfil psicológico [de um programador] é principalmente a capacidade de alternar entre níveis de abstração, do baixo nível ao alto nível. Para ver algo no pequeno e para ver algo no grande."

Donald E. Knuth

RESUMO

Trevisan, M. P. **Depuração de sistemas embarcados em tempo real com uma abordagem não intrusiva.** 2025. Monografia (Trabalho de Conclusão de Curso) – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2025.

Este trabalho aborda os desafios e as soluções relacionadas à depuração de sistemas embarcados em tempo real, com foco em técnicas não intrusivas. O objetivo principal é apresentar métodos que permitam a análise e correção de falhas sem interferir na operação normal do sistema, garantindo a previsibilidade e a confiabilidade exigidas por aplicações críticas. São explorados os conceitos fundamentais de sistemas embarcados e tempo real, os principais problemas enfrentados durante a depuração, bem como ferramentas modernas que viabilizam a observação do comportamento do sistema sem afetar sua execução. Ao final é realizado um estudo comparativo de 4 métodos de depuração, levantando métricas de tempo de execução da tarefa e tempo entre execuções das tarefas, assim como o consumo de memória. Este estudo contribui para o entendimento de práticas mais seguras e eficientes na engenharia de sistemas embarcados.

Palavras-chave: Sistemas embarcados. Depuração. Tempo real.. Não intrusiva. Ferramentas.

ABSTRACT

Trevisan, M. P. **Debugging embedded systems in real time with a non-intrusive approach**. 2025. Monografia (Trabalho de Conclusão de Curso) – Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2025.

This work addresses the challenges and solutions related to debugging real-time embedded systems, focusing on non-intrusive techniques. The main goal is to present methods that enable the analysis and correction of failures without interfering with the system's normal operation, ensuring the predictability and reliability required by critical applications. Fundamental concepts of embedded and real-time systems are explored, as well as the main problems encountered during debugging and modern tools that allow observation of system behavior without affecting its execution. At the end, a comparative study of 4 debugging methods is carried out, collecting metrics of task execution time and time between task executions, as well as memory consumption. This study contributes to the understanding of safer and more efficient practices in embedded systems engineering.

Keywords: Embedded systems. Real-time. Debugging. Non-intrusive. Tools.

LISTA DE FIGURAS

SUMÁRIO

1 INTRODUÇÃO

A indústria eletrônica tem apresentado um crescimento significativo nas últimas décadas, impulsionado principalmente pela integração de sistemas computacionais em uma vasta gama de produtos de uso cotidiano.

Dispositivos como automóveis, eletrodomésticos, equipamentos médicos, sistemas de automação residencial e industrial, entre outros, vêm incorporando soluções baseadas em eletrônica digital, o que resulta em produtos mais eficientes, acessíveis e com maior valor agregado (BARROS; CAVALCANTE, 2010).

Sistemas de computação, antes restritos a ambientes corporativos ou domésticos em forma de desktops e servidores, tornaram-se onipresentes, embutidos de maneira discreta em equipamentos que muitas vezes não aparentam conter qualquer tipo de inteligência computacional.

Essa tendência se reflete nos dados de produção: enquanto milhões de processadores são fabricados anualmente para computadores pessoais, bilhões de microcontroladores e processadores dedicados são empregados em sistemas embarcados voltados a aplicações específicas. Essa disparidade demonstra a escala e a importância crescente dos sistemas embarcados no cenário tecnológico atual (BARROS; CAVALCANTE, 2010).

O avanço desses sistemas, no entanto, trouxe consigo novos desafios no processo de desenvolvimento, especialmente pela necessidade de integração de componentes heterogêneos — como circuitos analógicos, sensores, atuadores e unidades de processamento digital — e pela crescente demanda por confiabilidade, baixo consumo energético e operação em tempo real.

Além disso, os requisitos de custo e tempo de mercado impõem restrições rígidas, tornando o projeto de sistemas embarcados uma tarefa complexa e estratégica (BARROS; CAVALCANTE, 2010).

De acordo com De Micheli (1996) existem três classes de sistemas digitais: emulação e sistemas de prototipação, sistemas de computação de propósito geral e sistemas embarcados (embedded systems).

Os sistemas de emulação e prototipação baseiam-se em tecnologias de hardware reprogramáveis, no qual o hardware pode ser reconfigurado pela utilização

de ferramentas de síntese. Tais sistemas requerem usuários especialistas e são utilizados para a validação de sistemas digitais.

Um sistema embarcado (*embedded system*) é um sistema computacional projetado para realizar uma função específica dentro de um sistema maior. Tais sistemas estão presentes em praticamente todos os dispositivos eletrônicos modernos.

Podem ser encontrados em produtos de consumo (como telefones celulares, câmeras digitais e videogames portáteis), em eletrodomésticos (como fornos de micro-ondas, máquinas de lavar e sistemas de climatização), em equipamentos de escritório (como impressoras e copiadoras), e em veículos automotivos (controlando freios ABS, injeção eletrônica, suspensão ativa, entre outros) (BARROS; CAVALCANTE, 2010).

Estudos apontam que, já na década de 1990, o número de sistemas embarcados em uma única residência norte-americana ultrapassava o número de computadores pessoais, com expectativa de crescimento exponencial nos anos seguintes. Em setores como o automotivo, por exemplo, o custo médio da eletrônica embarcada por veículo aumentou significativamente ao longo dos anos, refletindo a complexidade e o papel essencial desses sistemas nas funcionalidades modernas (BARROS; CAVALCANTE, 2010).

Os sistemas embarcados possuem características que os distinguem de outras categorias de sistemas computacionais. Em geral, apresentam (BARROS; CAVALCANTE, 2010):

1. **Funcionalidade dedicada**: são projetados para executar uma tarefa específica de forma repetitiva e previsível;

2. **Altas restrições de projeto**: precisam atender simultaneamente a critérios rigorosos de custo, tamanho físico, consumo energético, desempenho e confiabilidade;

3. **Operação em tempo real**: muitas aplicações exigem resposta imediata a eventos do ambiente, como é o caso de sistemas de controle automotivo, médicos ou industriais.

Dessa forma, o desenvolvimento de sistemas embarcados exige metodologias e ferramentas específicas, que levem em conta tanto os requisitos funcionais quanto as restrições não funcionais impostas pela aplicação. Neste contexto, aspectos como a depuração e validação do software embarcado, especialmente em sistemas que operam sob restrições de tempo real, tornam-se temas centrais para garantir o correto funcionamento e a confiabilidade do produto final.

1.1 Justificativa

Com o crescimento da complexidade dos sistemas embarcados, a etapa de depuração tem se tornado cada vez mais crítica no ciclo de desenvolvimento. Nos sistemas tradicionais, ferramentas de depuração como breakpoints, watchpoints e análise de logs são amplamente utilizadas.

No entanto, em sistemas embarcados com restrições de tempo real, o uso dessas ferramentas pode introduzir perturbações que afetam diretamente o comportamento temporal e funcional do sistema. Isso compromete a confiabilidade do processo de verificação e pode mascarar erros, dificultando sua identificação e correção.

Nesse cenário, torna-se essencial a adoção de técnicas de depuração não intrusiva, que possibilitem a observação do comportamento do sistema sem interferir em sua execução. Estas técnicas vêm ganhando relevância tanto na academia quanto na indústria, uma vez que permitem diagnósticos mais precisos e seguros em ambientes críticos.

A escolha deste tema se justifica, portanto, pela sua atualidade, relevância prática e pelo impacto direto na qualidade e segurança dos produtos desenvolvidos.

1.2 Objetivos

1.2.1 Objetivo Geral

O objetivo geral desse trabalho é apresentar e discutir métodos de depuração não intrusivas aplicadas a sistemas embarcados em tempo real, destacando sua importância, aplicações, vantagens e limitações. Essa demonstração é feita a partir da teoria da literatura sobre o tema e da demonstração prática desses métodos em um projeto de software de tempo real.

1.2.2 Objetivos Específicos

Para alcançar o objetivo geral desse trabalho, tem-se como objetivos específicos os seguintes tópicos:

- Conceituar sistemas embarcados e suas particularidades quanto ao projeto e à execução em tempo real;
- Apontar os principais desafios enfrentados no processo de depuração de sistemas embarcados;
- Explorar técnicas de depuração não intrusiva;
- Identificar as ferramentas e metodologias mais utilizadas na indústria e na pesquisa acadêmica.
- Realizar uma análise comparativa de diferentes métodos para demonstração dos conceitos apresentados.

1.3 Estrutura do Trabalho

Este trabalho está organizado da seguinte forma:

- **Capítulo 1 – Introdução:** apresenta o contexto geral da pesquisa, justificativa, objetivos e organização do trabalho.

- **Capítulo 2 – Fundamentos de Sistemas Embarcados:** discute os conceitos fundamentais sobre sistemas embarcados, arquitetura, classificações e características principais.

- **Capítulo 3 – Desafios na Depuração de Sistemas em Tempo Real:** aborda as limitações e complexidades da depuração em sistemas com restrições temporais.

- **Capítulo 4 – Abordagens de Depuração Não Intrusiva:** explora os métodos não intrusivos disponíveis, tecnologias utilizadas e aplicações práticas.

- **Capítulo 5 – Avaliação experimental de métodos de depuração em sistemas embarcados de tempo real:** compara quantitativamente alguns dos métodos apresentados, demonstrando vantagens e desvantagens de cada aplicação e benchmark de soluções.

- **Capítulo 6 – Resultados:** Realiza a análise dos resultados obtidos na seção anterior e discute o *trade-off* entre os diferentes métodos a partir dos dados coletados.

- **Capítulo 7 – Conclusão:** sintetiza os principais pontos discutidos, apresenta considerações finais e possíveis direções para trabalhos futuros.

2 FUNDAMENTOS DE SISTEMAS EMBARCADOS

2.1 Conceito de Sistemas Embarcados

Sistemas embarcados são sistemas computacionais projetados para desempenhar funções específicas, geralmente como parte integrante de dispositivos maiores. Eles combinam hardware e software dedicados para atender a requisitos particulares de desempenho, consumo de energia, tamanho e confiabilidade (SILVA JÚNIOR, 2018).

Diferentemente dos computadores de uso geral, que suportam múltiplas aplicações e possuem recursos amplamente configuráveis, os sistemas embarcados são altamente especializados. Essa especialização permite que eles operem de forma mais eficiente, com menos recursos e menor custo. Um exemplo simples é o microcontrolador presente em um controle remoto, cujo único propósito é interpretar os comandos do usuário e acionar um transmissor de infravermelho (SILVA JÚNIOR, 2018).

A definição mais técnica apresentada por Toniolo (2018), é que um sistema embarcado é um sistema de computador que faz parte de um produto, sendo projetado para realizar um ou poucos trabalhos específicos, muitas vezes em tempo real.

2.2 Estrutura Geral de um Sistema Embarcado

A estrutura de um sistema embarcado pode variar conforme a complexidade do projeto, mas em geral é composta por:

- **Processador** (CPU, microcontrolador ou DSP): responsável pela execução do programa e controle do sistema;
- **Memória**: armazenamento do firmware (ROM, Flash) e da execução (RAM);
- **Dispositivos de Entrada/Saída**: conectados a sensores, atuadores e outros periféricos;

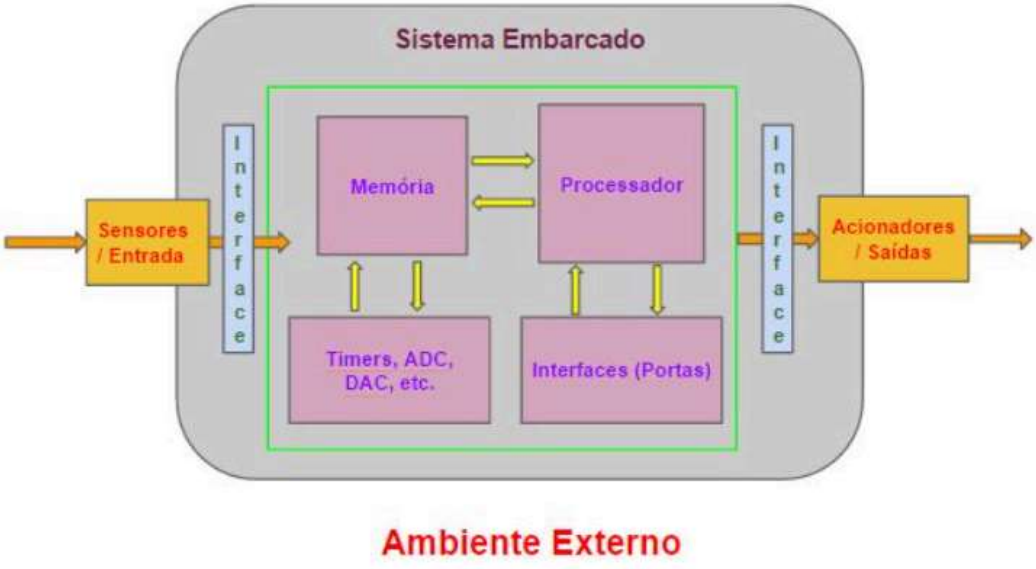
- **Barramentos de comunicação**: como I2C, SPI, CAN, UART, que permitem a troca de dados entre componentes;

- **Sistema de alimentação**: que pode ser desde uma bateria simples até fontes reguladas com proteção térmica;

- **Camadas de software**: incluindo o firmware, drivers, bibliotecas de tempo real (RTOS) e rotinas de inicialização (*bootloaders*).

A figura abaixo representa a arquitetura genérica de um sistema embarcado moderno:

Figura 1 – Arquitetura de um sistema embarcado moderno



Fonte: Reis (2015).

2.3 Classificações dos Sistemas Embarcados

Os sistemas embarcados podem ser classificados de várias maneiras (VARGAS, 2007):

Por complexidade:

- **Sistemas pequenos**: como alarmes e cronômetros, geralmente com 8 bits, pouca RAM e sem sistema operacional;

- **Sistemas médios**: como roteadores Wi-Fi, usando microcontroladores de 16 ou 32 bits e podendo rodar um sistema operacional leve;

- **Sistemas complexos**: como drones, smart TVs ou sistemas automotivos, com processadores avançados e sistemas operacionais embarcados completos (como Linux embarcado, Android ou QNX).

Por criticidade temporal:

- **Tempo real rígido (Hard Real-Time)**: atrasos na execução são inaceitáveis. Exemplo: controle de sistemas aeronáuticos;

- **Tempo real brando (Soft Real-Time)**: atrasos impactam o desempenho, mas não comprometem a funcionalidade. Exemplo: reprodutores de vídeo;

- **Sem tempo real**: não há requisitos temporais estritos. Exemplo: sistemas de atualização de firmware.

Por atualizabilidade:

- **Fixos**: o firmware é carregado uma única vez e raramente é alterado;

- **Atualizáveis**: permitem reprogramação via USB, OTA (Over-the-Air), JTAG, etc;

2.4 Características Principais

Entre as principais características que definem os sistemas embarcados, destacam-se (BARR; MASSA, 2006):

- **Funcionalidade dedicada**: são projetados para uma função específica;
- **Confiabilidade elevada**: devem operar continuamente e sem falhas por longos períodos;
- **Baixo consumo de energia**: especialmente crítico em dispositivos móveis ou autônomos;
- **Redução de custo e tamanho**: para permitir integração em produtos de massa;
- **Desempenho em tempo real**: capacidade de reagir a estímulos ambientais dentro de prazos predefinidos;
- **Alta integração de componentes**: muitos sistemas utilizam SoCs (System-on-Chip), que integram CPU, memória e periféricos em um único chip;
- **Interfaces especializadas**: como PWM para motores, ADC para sensores analógicos, ou interfaces automotivas como LIN e CAN.

2.5 Sistemas Embarcados em Tempo Real

De acordo com Ganssle (2008), sistemas embarcados de tempo real (*real-time embedded systems*) são aqueles que devem produzir respostas dentro de limites temporais rígidos. A capacidade de reagir em tempo hábil é tão importante quanto a correção lógica do resultado.

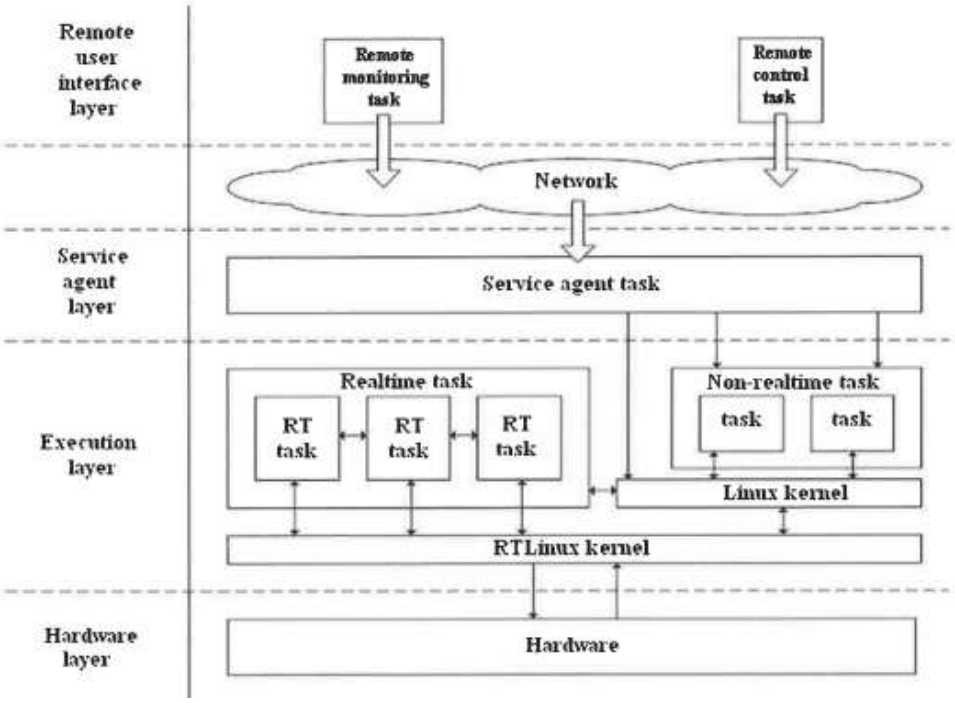
Esses sistemas exigem um controle rigoroso do tempo de execução de tarefas, da prioridade entre processos e do uso de interrupções. Costumam empregar um RTOS (*Real-Time Operating System*), alguns exemplos são FreeRTOS, Zephyr, ChibiOS, RTEMS, entre outros.

- Exemplo: Controle de Airbag

Em caso de colisão, o sistema embarcado do airbag deve processar o impacto, acionar o inflador e liberar o airbag em menos de 50 milissegundos. Um atraso pode tornar o sistema inútil e causar risco de vida. Esse é um exemplo clássico de sistema de tempo real rígido.

Abaixo uma representação de um sistema baseado em tempo real com sistema operacional RTLinux e uma arquitetura de software hierárquica:

Figura 2 – Arquitetura de software hierárquica de tempo real



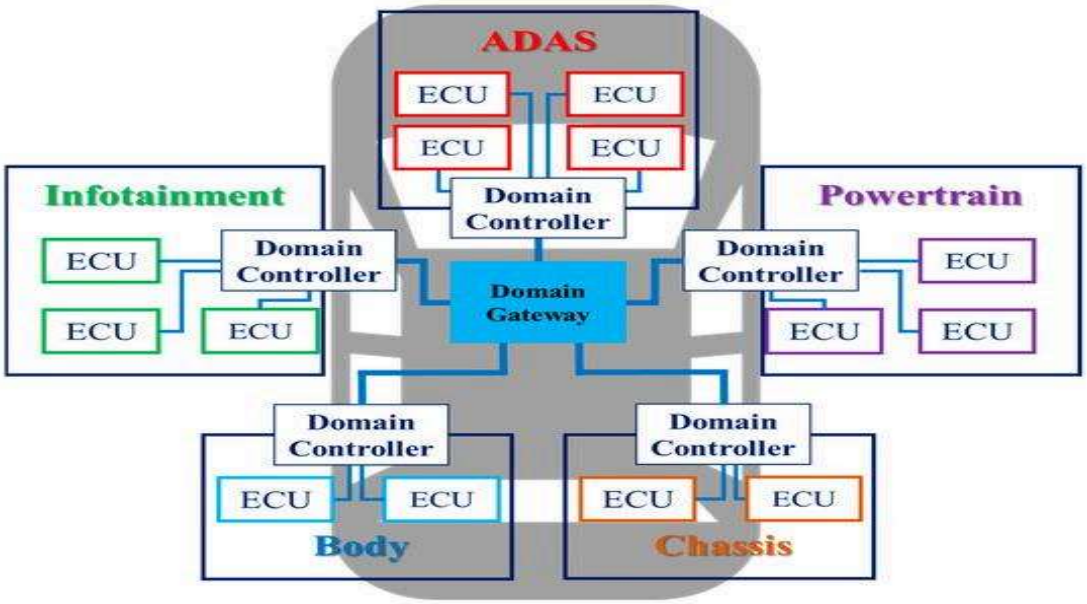
Fonte: HONG (2005).

2.6 Exemplos de Aplicações Reais

A aplicação de sistemas embarcados abrange praticamente todos os setores da economia e da vida cotidiana. Alguns exemplos práticos:

- **Automotivo**: unidades de controle eletrônico (ECU), sistemas de infoentretenimento, sensores de estacionamento, controle de injeção eletrônica, frenagem autônoma, conforme Figura 2.

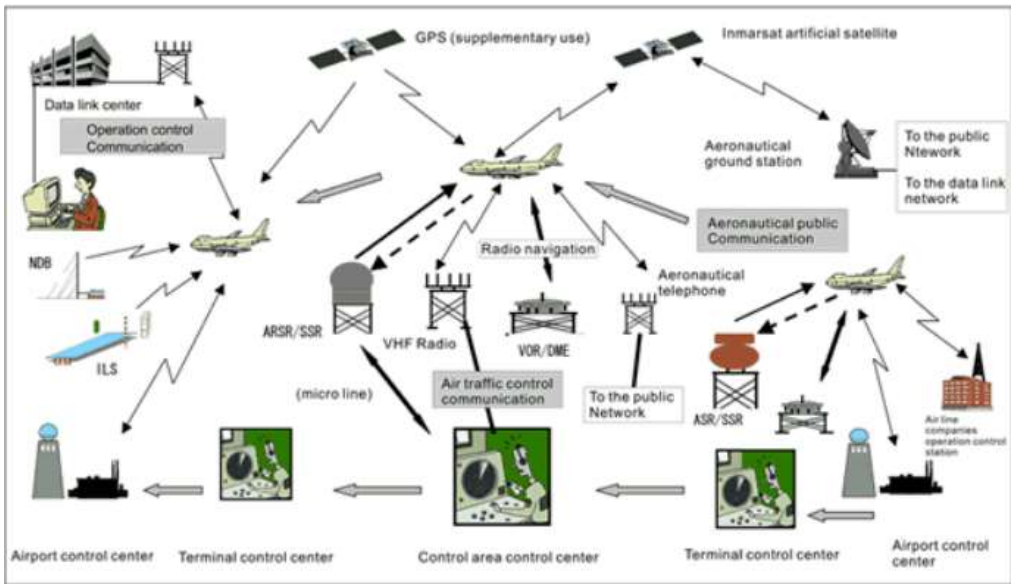
Figura 2 – Arquitetura de rede veicular baseada em zonas.



Fonte: Adaptado de KIM et al., 2023.

- **Aeronáutica**: sistemas de navegação, controle de estabilidade, comunicação via satélite, sistemas redundantes de voo, conforme Figura 3.

Figura 3 – Arquitetura de integração aeronáutica



Fonte: Liasch (2025).

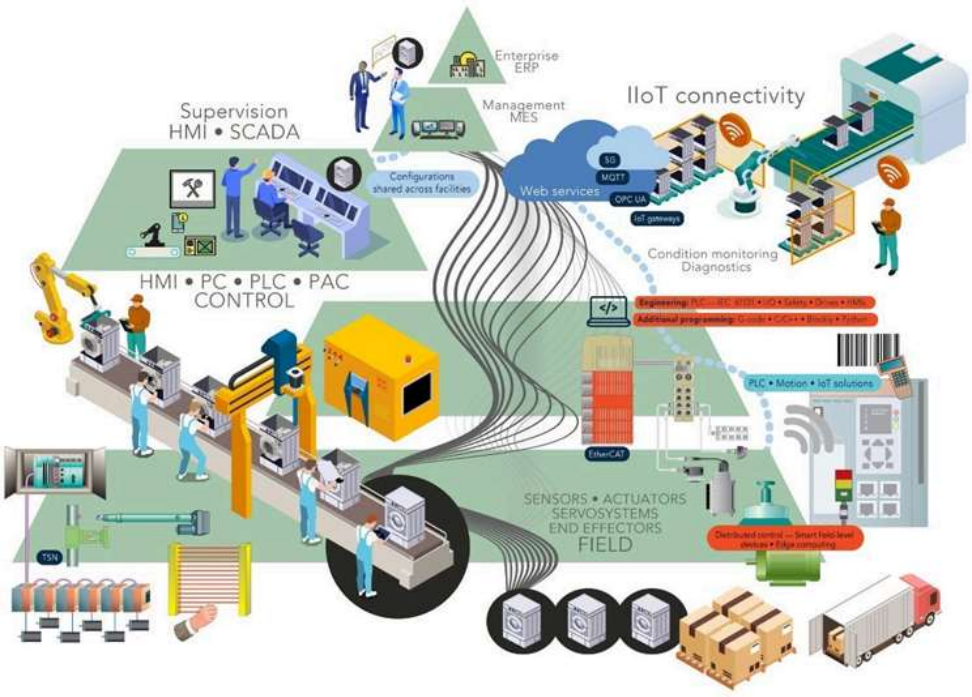
- **Médico-hospitalar**: monitores cardíacos, oxímetros, bombas de insulina, equipamentos de imagem, conforme Figura 4.

Figura 4 – Sistema médico-hospitalar

Fonte: Prolife (2025).

- **Indústria 4.0**: sensores IoT, atuadores inteligentes, robôs industriais, CLPs com Ethernet industrial, conforme Figura 5.

Figura 5 – Indústria 4.0



Fonte: Castro, Pacheco, Pinheiro (2023).

- **Consumo**: geladeiras inteligentes, smart TVs, dispositivos vestíveis (wearables), dispositivos Alexa/Google Home, conforme Figura 6.

Figura 6 – Itens de consumo

Fonte: Unipar (2025).

2.7 Tecnologias e Tendências Atuais

Com a evolução da tecnologia, algumas tendências vêm moldando o futuro dos sistemas embarcados HEATH (2002):

- **Internet das Coisas (IoT)**: sistemas embarcados conectados em rede, com capacidade de comunicação, coleta e processamento de dados remotos;
- **Inteligência Artificial embarcada** (*Edge AI*): uso de algoritmos de aprendizado de máquina localmente em dispositivos com baixa potência;
- **Cibersegurança embarcada**: aumento da preocupação com a proteção contra ataques em sistemas críticos;
- **Virtualização e containers**: uso de ferramentas como Docker e VxWorks para isolamento de funções em sistemas críticos;
- **Plataformas abertas**: uso crescente de sistemas baseados em Linux embarcado, como Yocto e Buildroot.

2.8 Considerações Finais

Os sistemas embarcados são parte essencial da infraestrutura tecnológica moderna. Sua evolução contínua é impulsionada por avanços em microeletrônica, redes sem fio, software embarcado e tecnologias de integração. O entendimento de seus fundamentos é indispensável para enfrentar os desafios de desenvolvimento, especialmente quando se trata de sistemas críticos com requisitos de tempo real.

Com a expansão do número de dispositivos inteligentes conectados e autônomos, espera-se que a demanda por soluções embarcadas eficientes e seguras continue crescendo de forma acelerada nos próximos anos.

3 OS DESAFIOS NA DEPURAÇÃO DE SISTEMAS EMBARCADOS EM TEMPO REAL

3.1 Introdução

Depurar sistemas embarcados sempre foi uma tarefa desafiadora, e essa complexidade se intensifica quando o sistema possui requisitos de tempo real. A necessidade de garantir respostas dentro de prazos rígidos, aliada à escassez de recursos computacionais e à interação com o mundo físico, cria um cenário em que os métodos tradicionais de depuração nem sempre são aplicáveis ou seguros (Ganssle, 2008; Barr; Massa, 2006).

Neste capítulo, serão discutidos os principais desafios enfrentados durante o processo de depuração de sistemas embarcados em tempo real, ilustrando as limitações das abordagens convencionais e a necessidade de técnicas especializadas que minimizem interferências no funcionamento do sistema.

3.2 Restrições de Tempo Real

Sistemas embarcados de tempo real são altamente sensíveis a atrasos. A simples inserção de um ponto de parada (*breakpoint*), ou a utilização de ferramentas de instrumentação em tempo de execução, pode causar perda de *deadlines*, falhas no comportamento esperado ou até travamentos (Marwedel, 2010).

Esse cenário é especialmente crítico em aplicações como:

- Sistemas automotivos (ex: freio ABS);
- Dispositivos médicos (ex: marca-passos);
- Aeronaves (ex: controle de voo automático).

Essas aplicações exigem que qualquer observação ou modificação de variáveis ocorra de forma não intrusiva, sem afetar o tempo de resposta do sistema (Yiu, 2015).

3.3 Visibilidade Limitada

Diferente de sistemas desktop, que oferecem interfaces gráficas e monitoramento por software, muitos sistemas embarcados operam com:

- Ausência de display ou interface gráfica;
- Acesso restrito à memória ou registros internos;
- Comunicação limitada (ex: UART, SPI);
- Falta de sistema operacional completo (*bare-metal*).

A consequência é a baixa visibilidade do estado interno do sistema, dificultando a compreensão do que está acontecendo durante a execução (Wolf, 2001).

3.4 Interferência da Depuração no Comportamento

Grande parte das ferramentas de depuração tradicionais, como printf ou *breakpoints* em IDEs (ex: Keil, IAR, MPLAB X), altera o comportamento do sistema. Isso ocorre por diversos motivos:

- O tempo gasto para imprimir dados pode atrasar a execução;
- A parada do sistema interrompe a comunicação com sensores;
- A adição de código de depuração altera o consumo de memória e CPU.

Esse fenômeno é conhecido como efeito Heisenbug, quando o erro desaparece ao tentar observá-lo (Barr e Massa, 2006).

Em sistemas críticos, essa intrusão pode mascarar ou provocar falhas perigosas (Ganssle, 2008).

3.5 Escassez de Recursos Computacionais

Sistemas embarcados são frequentemente otimizados para operar com o mínimo possível de recursos, como:

- Pouca memória RAM (às vezes menos de 32 KB);
- Armazenamento limitado;
- CPU com *clock* baixo;
- Ausência de unidades de ponto flutuante ou MMU.

Essas limitações tornam inviável o uso de ferramentas complexas de *logging* ou análise em tempo real (Wolf, 2001; Yiu, 2015).

3.6 Concorrência e Sincronização

Muitos sistemas embarcados de tempo real utilizam multitarefa por meio de RTOS ou interrupções. Isso introduz desafios como:

- Corridas críticas entre tarefas;
- Dificuldade de reproduzir bugs não determinísticos;
- Interrupções assíncronas que afetam a ordem de eventos;
- *Deadlocks* e *starvation.*

A depuração precisa lidar com o comportamento concorrente do sistema sem interromper sua ordem natural de execução (Labrosse, 2002; Tanenbaum e

Bos, 2015). Além disso, como alerta Koopman (2021), bugs de concorrência em sistemas críticos são frequentemente irreproduzíveis em testes, exigindo técnicas como mutexes ou desabilitação de interrupções para mitigação.

3.7 Falhas Intermitentes e Difíceis de Reproduzir

Outro problema comum é que muitos erros só ocorrem em condições muito específicas (ex: falhas de comunicação sob carga pesada, leituras erradas sob determinada temperatura). Esses erros intermitentes:

- São difíceis de capturar com técnicas tradicionais;
- Podem não ocorrer durante os testes controlados;
- Requerem longos períodos de monitoramento.

Nesses casos, a depuração precisa incluir mecanismos de rastreamento contínuo ou análise posterior do comportamento do sistema.

3.8 Acesso Físico Restrito

Em muitas situações, o sistema está fisicamente inacessível após ser instalado (ex: sensores industriais em locais perigosos, sondas espaciais, implantes médicos). Isso dificulta o uso de ferramentas que dependem de acesso físico direto, como JTAG, SWD ou portas seriais.

Essa limitação exige soluções de monitoramento remoto ou depuração via telemetria, que não dependam de conexão direta com o dispositivo.

3.9 Testes com Ambiente Realista

A simulação ou emulação de ambientes de tempo real nem sempre é fiel. Por exemplo:

- Sensores reais têm ruído, latência e falhas que simuladores não reproduzem;

- Cargas variáveis na alimentação podem afetar o comportamento;

- Eventos assíncronos (como falhas de comunicação) são difíceis de prever.

A depuração precisa considerar o contexto real em que o sistema está inserido, o que torna necessário usar técnicas de depuração *in situ* (Marwedel, 2010; Ganssle, 2008).

3.10 Considerações Finais

A depuração de sistemas embarcados em tempo real exige um equilíbrio delicado entre observação e interferência. Métodos tradicionais, como *breakpoints* ou *prints*, são muitas vezes inadequados ou perigosos nesses contextos. Por isso, o desenvolvimento de métodos não intrusivos de depuração torna-se essencial para garantir segurança, confiabilidade e cumprimento dos requisitos temporais (Ganssle, 2008; Barr e Massa, 2006).

4 MÉTODOS DE DEPURAÇÃO NÃO INTRUSIVOS

4.1 Introdução

Como discutido anteriormente, a depuração de sistemas embarcados em tempo real impõe desafios significativos relacionados a restrições temporais, visibilidade limitada e interferência no comportamento do sistema. Diante disso, surgem técnicas de depuração não intrusiva, que permitem monitorar e analisar o sistema sem comprometer sua funcionalidade ou desempenho (GANSSLE, 2008).

Este capítulo apresenta os principais métodos não intrusivos aplicáveis à depuração de sistemas embarcados em tempo real, com ênfase em rastreamento, monitoramento passivo, análise fora de linha e uso de ferramentas especializadas (BARR; MASSA, 2006).

4.2 Conceito de Depuração Não Intrusiva

A depuração não intrusiva é caracterizada pela observação do comportamento do sistema sem alteração de sua lógica, tempo de resposta ou consumo de recursos significativos. Em vez de interferir diretamente na execução do software, essas técnicas geralmente envolvem a coleta de dados por hardware dedicado, espionagem de barramentos (bus sniffing), uso de registradores de eventos e monitoramento externo por meio de portas de debug como JTAG ou SWD (MARWEDEL, 2010). Koopman (2021) reforça que a depuração eficaz em sistemas embarcados priorize a observação sem intrusão, citando rastreamento por hardware ou logs em tempo real como métodos essenciais para diagnóstico em ambientes críticos.

4.3 Rastreio por Hardware (Hardware Tracing)

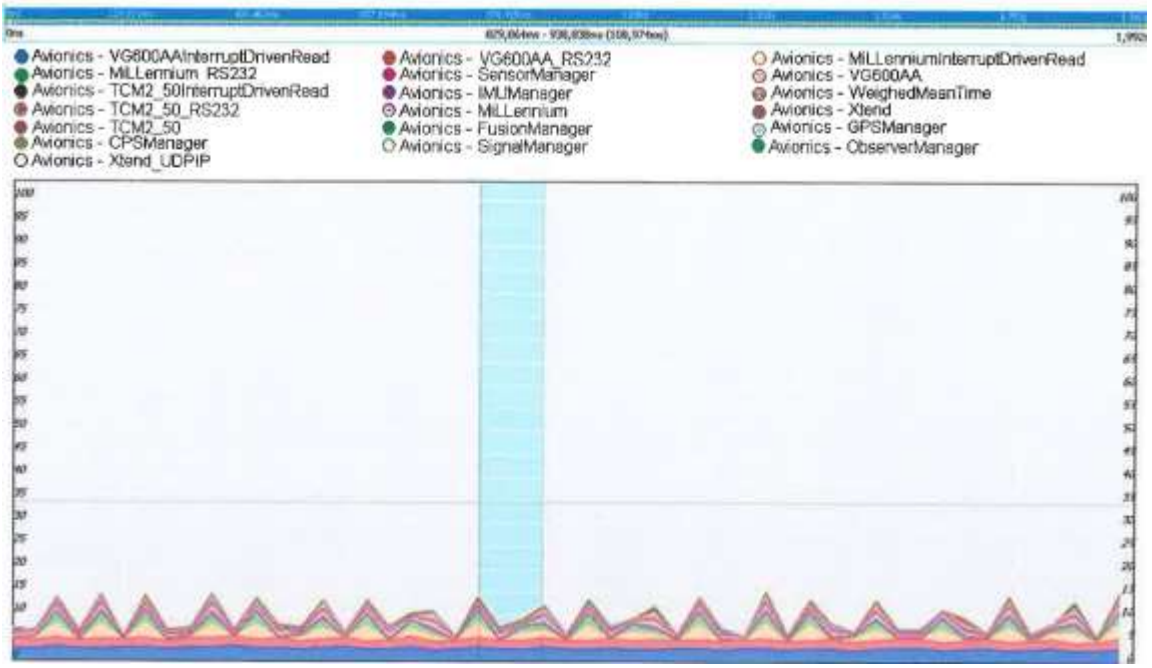
Esse método utiliza recursos embutidos nos microcontroladores para rastrear a execução de instruções, interrupções ou acessos à memória. Entre os principais recursos estão o ETM (Embedded Trace Macrocell), o ITM (Instrumentation Trace

Macrocell) e o SWO (Single Wire Output), todos presentes na arquitetura ARM Cortex-M (YIU, 2015).

Esses mecanismos permitem rastrear o sistema em tempo real com impacto mínimo, pois não requerem modificação no código fonte nem causam atrasos significativos (SEGGER, 2024).

Um exemplo desse rastreio pode ser observado no sistema de tracing de alguns sistemas operacionais de tempo real como o QNX, abaixo uma figura do System Profiler de um sistema de tempo real de um VANT:

Figura 6 – Distribuição de consumo de processamento do projeto



Fonte: AMIANTI (2008).

Amianti (2008) demonstra o uso de ferramentas não intrusivas em sistemas de tempo real, permitindo identificação de gargalos de desempenho, validando a eficácia do método para aplicações críticas como VANTs. Essa abordagem é a base para ferramentas modernas como SEGGER System View ou Percepio TraceAlyzer, extendendo essas ferramentas para microcontroladores ARM Cortex-M.

4.4 Espionagem de Barramentos

Sniffers de barramento observam sinais elétricos de comunicação (como SPI, I²C, UART ou CAN) sem interagir diretamente com os dispositivos conectados.

Ferramentas como Saleae Logic Analyzer e Bus Pirate permitem capturar pacotes de dados, diagnosticar falhas e medir o desempenho da comunicação sem intrusão (SALEAE, 2024).

4.5 Monitoramento por Registradores de Eventos

Muitos microcontroladores modernos incluem buffers circulares ou registradores que armazenam eventos do sistema, como chamadas de funções ou interrupções. Esses dados podem ser acessados por interfaces de depuração sem interromper a execução do firmware, auxiliando na identificação de falhas esporádicas (YIU, 2015).

Esse tipo de recurso também é explorado por ferramentas como o FreeRTOS+Trace.

4.6 Espelhamento de Variáveis (Shadowing)

O shadowing consiste em replicar variáveis críticas em regiões de memória acessíveis externamente. Esse método pode utilizar mecanismos como DMA ou canais de depuração como SWO, para evitar a degradação do desempenho (MARWEDEL, 2010; YIU, 2015).

Soluções comerciais, como o Tracealyzer (PERCEPIO, 2024) citado na seção 4.9 implementam shadowing de variáveis críticas via DMA, reduzindo a carga de CPU durante o monitoramento.

4.7 Depuração por Instrumentação de Hardware Externo

Instrumentos externos, como osciloscópios, analisadores lógicos e câmeras de alta velocidade, também são empregados para observar sinais físicos emitidos pelo sistema. Essa abordagem é comum quando o sistema não permite inserção de código adicional ou está em ambiente crítico (GANSSLE, 2008).

4.8 Análise Fora de Linha (Post-Mortem)

Em sistemas críticos ou inacessíveis, a coleta de dados é feita após uma falha, por meio de armazenamento em memória não volátil. A análise offline permite reconstruir o estado do sistema antes da falha (BARR; MASSA, 2006), sendo uma prática comum em aplicações aeroespaciais ou industriais (MARWEDEL, 2010).

4.9 Soluções Comerciais e Open Source

Ferramentas especializadas ajudam na implementação de depuração não intrusiva:

- **SystemView**, da SEGGER, oferece rastreamento de tarefas com base em ITM/SWO (SEGGER, 2024);

- **Tracealyzer**, da Percepio, fornece visualização gráfica detalhada do comportamento do sistema com suporte a FreeRTOS e outros RTOS (PERCEPIO, 2024). Assim como o System Profiler utilizado em Amianti (2008), ferramentas como o Tracealyzer mantêm o princípio de rastreamento passivo, com atualizações para visualizações modernas e suporte a demais RTOS.

- **FreeRTOS+Trace** é uma solução integrada ao kernel do FreeRTOS, voltada para análise visual e rastreamento;

- **Black Magic Probe** e **OpenOCD** são alternativas open source para debug via JTAG/SWD (BLACK MAGIC PROBE, 2024; OPENOCD, 2024).

A seguir uma tabela comparativa demonstrando vantagens, limitações e aplicação típica de cada uma das ferramentas citadas:

Ferramenta	Vantagens	Limitações	Aplicação típica
Tracealyzer	-Visualização gráfica de tasks, semáforos e filas em RTOS -Suporte a FreeRTOS, Zephyr, ThreadX -Baixo overhead (<1% CPU)	-Custo comercial (≈$2k/licença) -Requer hardware específico para tracing completo	Sistemas médicos, automotivos
SystemView	-Gratuita para uso não comercial -Integração nativa com ITM/SWO (ARM Cortex) -Timeline interativa de eventos	-Limitação a microcontroladores ARM -Configuração complexa para sistemas multicore	Drones, IoT industrial
FreeRTOS+Trace	-Integrado ao kernel do FreeRTOS -Logs de contexto mínimo (sem modificação de código)	-Sem visualização gráfica nativa -Depende de ferramentas externas para análise	Prototipagem rápida
OpenOCD	-Open source -Suporte a múltiplas arquiteturas (ARM, RISC-V) -Depuração via JTAG/SWD	-Sem recursos avançados de tracing -Curva de aprendizado íngreme	Sistemas acadêmicos/low-cost
Black Magic Probe	-Depuração sem driver adicional -Compatível com GDB -Baixo custo (hardware aberto)	-Baixo custo (hardware aberto)-Funcionalidades básicas (sem tracing) -Limitado a microcontroladores específicos	Projetos makers/educacionais

4.10 Considerações Finais

A depuração não intrusiva é essencial para garantir o diagnóstico seguro e preciso de sistemas embarcados em tempo real. Combinando técnicas de rastreamento por hardware, espionagem de barramentos, monitoramento de variáveis e análise pós-falha, é possível obter visibilidade detalhada do sistema sem afetar sua operação. O uso dessas abordagens deve ser considerado desde o início do projeto, integrando os recursos necessários ao hardware e firmware.

5 AVALIAÇÃO EXPERIMENTAL DE MÉTODOS DE DEPURAÇÃO EM SISTEMAS EMBARCADOS DE TEMPO REAL

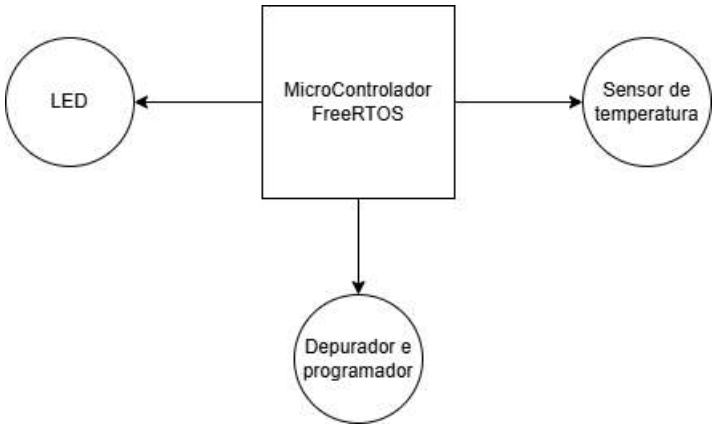
Para avaliar alguns dos diferentes métodos de depuração apresentados nesse projeto, foi elaborado um software embarcado, em um sistema de tempo real, visando avaliar os resultados e comparar os impactos na execução das tarefas e complexidade da solução. Para este experimento foram usadas somente soluções que não dependem de componentes externos ou softwares licenciados.

5.1 Metodologia experimental

Este capítulo apresenta uma avaliação experimental comparativa de diferentes métodos de depuração em sistemas embarcados de tempo real. A metodologia adotada tem como objetivo comparar quantitativamente o desempenho e a interferência causada por distintas técnicas de depuração, com base em métricas objetivas mensuráveis em um sistema de teste controlado.

Segue um diagrama da estrutura do ensaio executado:

Figura 7 – Diagrama de conexões da bancada de testes



Fonte: Elaborado pelo autor.

5.1.1 Objetivo principal

Comparar quantitativamente diferentes métodos de depuração em um sistema embarcado de tempo real, avaliando seu impacto no comportamento temporal do sistema e na capacidade de detecção de anomalias.

Os desafios na depuração de sistemas embarcados de tempo real, discutidos no capítulo 3 nas seções 3.2 e 3.4 motivaram a seleção das métricas de avaliação.

O objetivo é metrificar precisamente o impacto desses desafios quando diferentes técnicas de depuração são aplicadas.

5.1.2 Hipótese de pesquisa

Métodos de depuração não intrusivos causam menor interferência no comportamento temporal do sistema quando comparados a métodos tradicionais intrusivos, permitindo uma análise mais precisa e confiável do funcionamento do sistema sem comprometer seus requisitos temporais críticos.

5.1.3 Variáveis independentes

As variáveis independentes deste projeto correspondem aos diferentes métodos de depuração avaliados e apresentados no capítulo 4 deste projeto, sendo:

1. Logger Uart bloqueante na tarefa de controle:

Implementação direta de instruções de log na tarefa crítica do sistema.

2. Logger Uart bloqueante em tarefa de menor prioridade:

Utilização de uma tarefa de baixa prioridade para obtenção das tarefas críticas.

3. Logger Uart DMA na tarefa de controle:

Acesso direto às variáveis do sistema através da interface de depuração serial sem interromper execução.

4. Logger Uart DMA em tarefa de menor prioridade:

Implementação de um buffer circular esvaziado por DMA para minimizar interferência.

Portanto, será avaliado um método intrusivo, sendo o método 1, a ser avaliado como grupo controle para as demais métricas. As demais métricas são baseadas nas técnicas apresentadas nas seções 3 e 4 deste projeto.

5.1.4 Variáveis dependentes

As variáveis dependentes se referem as métricas quantitativas que serão utilizadas para avaliar o desempenho e impacto de cada método

1. tempo de resposta da tarefa crítica:

Medido em ticks do sistema operacional e com o auxílio de um timer ajustado para medições de microssegundos. Esta métrica avalia o impacto direto dos métodos de depuração no tempo de execução da tarefa crítica, que deve permanecer dentro de limites estritos para garantir o comportamento em tempo real do sistema.

2. Consumo de CPU:

Calculado como o tempo que a tarefa demora para ser executada, esta métrica também é obtida com os ticks do sistema operacional e a partir do timer configurado para microssegundos.

3. Uso de memória:

Quantificado em kilobytes de RAM e Flash necessários para implementação de cada método. Esta métrica é obtida através da análise do mapa de memória gerado pelo linker, comparando o consumo de cada método para o projeto final.

5.1.5 Procedimento experimental

O experimento será conduzido em um sistema de teste baseado em um microcontrolador STM32F401RE (ARM Cortex-M4, 8MHz) executando o sistema operacional de tempo real FreeRTOS v10.3.1. A aplicação consiste em um sistema de controle de temperatura com uma tarefa crítica de período fixo (5ms) e uma tarefa secundária para obtenção de dados.

A tarefa secundária foi definida como menor prioridade no sistema, e com período fixo de 200ms.

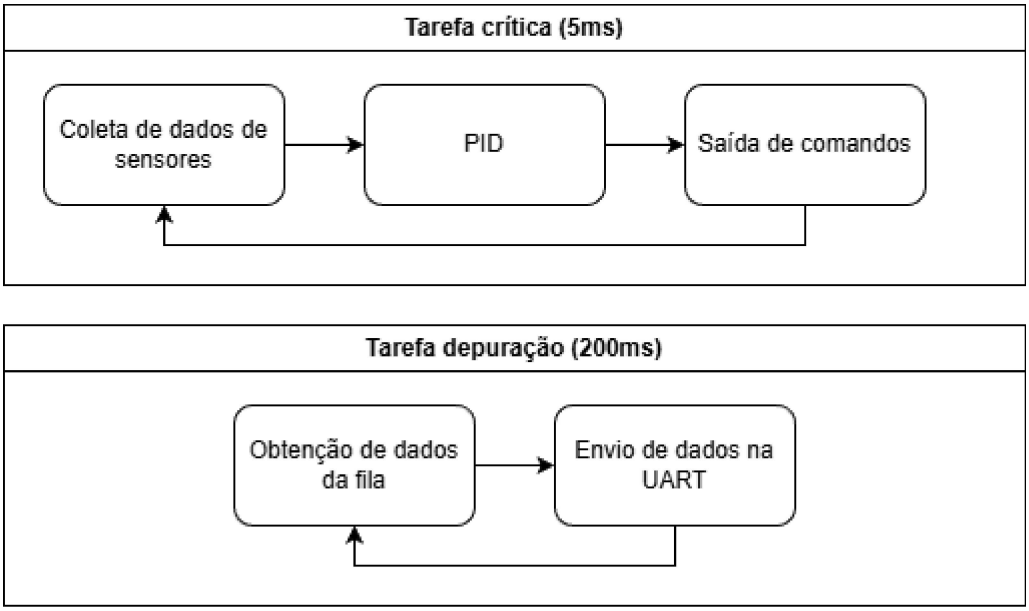
Além disso, os diferentes métodos podem ser selecionados a partir de diretivas de compilador, permitindo que o software seja avaliado quanto ao consumo de memória RAM e ROM, compilando somente as seções selecionadas pelas diretivas iniciais do sistema.

A tarefa secundária não está presente em todas as versões testadas, como detalhado em cada um dos métodos ensaiados.

O envio de dados entre a tarefa de maior prioridade e a de menor prioridade é feita a partir de fila própria do sistema operacional, garantindo que a transição de informação entre tarefas não gere impactos no funcionamento do sistema.

Segue um diagrama detalhando a arquitetura do procedimento experimental:

Figura 8 – Arquitetura de tarefas do software



Fonte: Elaborado pelo autor.

5.1.6 Procedimento de teste

Para cada método de depuração, serão realizadas 30 medições consecutivas sob condições controladas idênticas, garantindo a reprodutibilidade dos resultados.

5.1.7 Coleta de dados

A coleta de dados será realizada através de um timer configurado para medição precisa do tempo de execução da ordem de 10 us além do contador de ticks do sistema operacional, os dados de consumo de memória serão obtidos a partir do linker gerado.

6 Resultados

A partir dos ensaios propostos na seção anterior, foram realizadas 4 sequências de ensaios, coletando os dados e os organizando para demonstração.

Os dados obtidos em conjunto com os dados da própria tarefa de controle permitem avaliar o impacto da introdução do software de registro de dados de controle no sistema.

6.1 Resultados do Método 1: Logger Uart bloqueante na tarefa de controle

A tabela abaixo mostra os valores médios dos resultados obtidos nos ensaios do primeiro método:

RTOS (ms)		Timer (us)		Uso de memória (KB)	
Tempo entre execuções	Consumo de CPU	Tempo entre execuções	Consumo de CPU	Memória RAM	Memória Flash
13 ± 0	8 ± 0	13000 ± 0	8761 ± 3	21,32	51,89

Observa-se que neste caso a tarefa de controle não está conseguindo ser executada na frequência desejada, esse comportamento é esperado devido ao tempo de execução da própria task, que excede o tempo definido entre execuções, dessa forma o escalonador do sistema operacional não pode chamar uma próxima execução da tarefa, pois depende da finalização da primeira execução.

Neste caso, apesar do uso de um sistema operacional de tempo real, o método de depuração selecionado impede que ele opere em condições de tempo real, afetando drasticamente o resultado esperado pela tarefa.

6.2 Resultados do Método 2: Logger Uart bloqueante em tarefa de menor prioridade:

A tabela abaixo mostra os valores médios dos resultados obtidos nos ensaios do primeiro método:

RTOS (ms)		Timer (us)		Uso de memória (KB)	
Tempo entre execuções	Consumo de CPU	Tempo entre execuções	Consumo de CPU	Memória RAM	Memória Flash
5 ± 0	0,0 ± 0,0	4992 ± 4	80 ± 0	21,33	52,61

Observa-se que neste caso o tempo de execução da tarefa foi mantido dentro do projetado para o sistema operacional, garantindo a operação de tempo real.

Esse método de depuração, ao utilizar uma tarefa de menor prioridade, permite que o sistema possa ser observado sem impactar a tarefa crítica do sistema de mais alta prioridade.

O impacto desse método está na necessidade da criação de uma tarefa específica para debug, aumentando a complexidade do software e exigindo que a depuração seja implementada ainda em fase de projeto, não como alternativa a uma detecção de falha, além disso onera o sistema com maior consumo de RAM e FLASH.

Como apresentado no capítulo 3, sistemas embarcados possuem uma característica comum relacionada a escassez de recursos, e em alguns casos o aumento e consumo de RAM e FLASH, ou necessidade de novas tasks pode inviabilizar o método de depuração.

6.3 Resultados do Método 3: Logger Uart DMA na tarefa de controle

A tabela abaixo mostra os valores médios dos resultados obtidos nos ensaios do primeiro método:

RTOS (ms)		Timer (us)		Uso de memória (KB)	
Tempo entre execuções	Consumo de CPU	Tempo entre execuções	Consumo de CPU	Memória RAM	Memória Flash
5 ± 0	0,0 ± 0,0	4999 ± 2	368 ± 3	21,58	52,67

Observa-se que neste caso, apesar de ser executado na própria tarefa de controle, o sistema operacional mantém o determinismo de tempo, devido ao uso do coprocessador para envio de dados de depuração em paralelo a execução da tarefa.

Apesar disso, nota-se que o consumo de CPU da tarefa é afetado, portanto em sistemas que possuem restrição de tempo de execução de tarefas, esse método pode interferir no comportamento do sistema, podendo interferir na análise do sistema.

6.4 Resultados do Método 4: Logger Uart DMA em tarefa de menor prioridade

A tabela abaixo mostra os valores médios dos resultados obtidos nos ensaios do primeiro método:

RTOS (ms)		Timer (us)		Uso de memória (KB)	
Tempo entre execuções	Consumo de CPU	Tempo entre execuções	Consumo de CPU	Memória RAM	Memória Flash
5 ± 0	0,0 ± 0,0	4986 ± 5	80 ± 0	21,59	53,89

Observa-se que neste caso, tanto o tempo entre execuções das tarefas, quanto o tempo de execução da tarefa não são afetados, permitindo que a tarefa crítica possua determinismo sem sobrecarga de processamento com a depuração.

Apesar disso, assim como citado na seção 6.2, há a necessidade da criação de uma segunda tarefa, focada em depuração, que gera aumento de complexidade, memória e alternância do escalonador. Em sistemas que operam com maior sobrecarga esses fatores podem ser limitantes para a operação desse método.

6.5 Análise comparativa e validação da hipótese

A partir dos resultados, podemos observar que os métodos com menor intrusão ao sistema, apresentados como proposta no capítulo 4 demonstram ter menor impacto nas métricas de desempenho do sistema operacional.

Porém, a adição de códigos de depuração podem gerar outros impactos, como a necessidade de desenvolvimento adicional para a obtenção de dados, consumo de memória ROM no software desenvolvido, maior consumo de memória RAM para a execução das tarefas e maior tempo de execução da tarefa que se deseja monitorar, portanto sendo necessária a análise para cada contexto específico, considerando fatores comuns em contexto de aplicações de software embarcado, como as apresentadas na seção 3 deste projeto, como a restrição de memória, criticidade de tempo de execução da tarefa e tempo de resposta.

A tabela abaixo compara as métricas levantadas para os diferentes métodos:

	RTOS (ms)		Timer (us)		Uso de memória (KB)	
Método	Tempo entre execuções	Consumo de CPU	Tempo entre execuções	Consumo de CPU	Memória RAM	Memória Flash
1	13 ± 0	8 ± 0	13000 ± 0	8761 ± 3	21,32	51,89
2	5 ± 0	0,0 ± 0,0	4992 ± 4	80 ± 0	21,33	52,61
3	5 ± 0	0,0 ± 0,0	4999 ± 2	368 ± 3	21,58	52,67
4	5 ± 0	0,0 ± 0,0	4986 ± 5	80 ± 0	21,59	53,89

Nota-se que todos os métodos de depuração de menor intrusão testados possuem pouco impacto no determinismo da tarefa crítica do sistema. Reforçando a hipótese da menor interferência na aplicação desses métodos, porém também possuem aumento no consumo de memória RAM e FLASH, assim como tempo de execução da tarefa no caso 3, essas métricas podem ser determinantes na seleção do método que melhor performa para cada especificidade de projeto.

Comparando-se os métodos 1 com os restantes, observa-se que o sistema perde o determinismo de tempo, neste caso, apesar do uso de um sistema de tempo real, o resultado não garante o determinismo.

Para os métodos 2, 3 e 4, que possuem menor intrusão no sistema, observa-se que todos mantiveram o determinismo de tempo real, garantindo a execução das tarefas a cada 5 ms, reforçando a necessidade de aplicar métodos de depuração de menor intrusão.

A comparação entre os métodos 2 e 3 ou 3 e 4 demonstram que o método 3 possui maior interferência no sistema, gerando um aumento no tempo de execução da tarefa crítica do sistema, essa métrica pode ser crítica em sistemas que exigem

resposta rápida na execução de tarefas, ainda assim, neste exemplo, o tempo de execução da tarefa foi inferior ao tick do escalonador do sistema operacional, portanto não impede a transição de prioridades.

Apesar da aparente desvantagem do método 3, referente ao tempo de execução da tarefa crítica, é importante ressaltar que esse método não exige a adição de uma nova tarefa no escalonador do sistema operacional, garantindo que esse sistema tenha um número menor de transição entre tarefas, reduzindo o risco de falhas de concorrência e sincronização, como detalhadas no capítulo 3.

Para todos os casos analisados, os dados obtidos permitem a avaliação do sistema a partir dos logs obtidos, como uma análise post-mortem (ou fora de linha), como apresentadas no capítulo 3, sendo necessário avaliar os requisitos de cada projeto para a definição do método que melhor se adequa para obtenção desses dados. Além disso, o método 1 recomenda-se somente para projetos que não são críticos de tempo real, como detalhado no capítulo 2.

O resultado obtido reforça a hipótese de que métodos não intrusivos possuem vantagens na depuração de sistemas embarcados de tempo real, principalmente no determinismo das tarefas, porém deve-se atentar para os impactos resultantes destes métodos. Os demais métodos apresentados na seção 4 deste trabalho possuem cada um as suas vantagens e desvantagens, como custo, disponibilidade, e outros impactos que decorrem das implementações necessárias para as suas aplicações, assim como demonstrados neste trabalho, abrindo espaço para que futuros projetos realizem comparativos com demais ferramentas ou no desenvolvimento de ferramentas que reduzam os impactos observados neste ensaio.

7 CONCLUSÃO

A depuração de sistemas embarcados em tempo real representa uma atividade essencial, porém desafiadora, na engenharia de sistemas críticos. Esses sistemas operam sob restrições severas de tempo, consumo de energia, memória e processamento, exigindo precisão absoluta em suas respostas e previsibilidade em sua execução.

Diante dessas exigências, o uso de métodos tradicionais de depuração, como instruções de printf(), breakpoints e depuradores em tempo real, mostra-se frequentemente inadequado, pois introduz interferências no sistema que podem alterar seu comportamento, comprometendo a validade das análises.

Neste trabalho, foram apresentados conceitos fundamentais sobre sistemas embarcados e sistemas de tempo real, com destaque para suas principais características: reatividade, determinismo, confiabilidade e desempenho.

Com base nesse panorama, discutiram-se os principais desafios enfrentados na depuração desses sistemas, como a intrusividade dos métodos convencionais, a dificuldade de reprodutibilidade dos erros, as limitações de recursos nos dispositivos embarcados e a escassez de ferramentas compatíveis com aplicações críticas.

Em resposta a esses desafios, o trabalho realizou a comparação entre diferentes métodos, definidos como intrusivos a métodos não intrusivos, permitindo uma análise prática do *trade-off* entre as diferentes aplicações.

A análise comparativa entre métodos intrusivos e não intrusivos demonstra que, embora os métodos tradicionais ainda tenham valor em cenários de desenvolvimento iniciais ou prototipagem, sua aplicabilidade em ambientes críticos e sistemas de produção é limitada. A adoção de técnicas não intrusivas é, portanto, um passo essencial para garantir a confiabilidade, segurança e qualidade dos produtos embarcados.

Por fim, ressalta-se que a evolução das ferramentas de desenvolvimento embarcado, aliada ao avanço dos microcontroladores modernos com recursos integrados de rastreamento e depuração, vem tornando essas abordagens cada vez mais acessíveis.

Ainda assim, há espaço para pesquisa e inovação, especialmente no desenvolvimento de soluções de baixo custo, abertas e compatíveis com

ecossistemas amplamente utilizados como o ARM Cortex-M e plataformas como Arduino e STM32, mitigando os efeitos de complexidade de desenvolvimento de soluções mais robustas de depuração e permitindo menor impacto na integração desses sistemas em projetos complexos.

REFERÊNCIAS

BARROS, E.; CAVALCANTE, S. Introdução aos sistemas embarcados. Universidade Federal de Pernambuco, 2010. Centro de estudos em informática. Disponível em: http://www.maxpezzin.com.br/aulas/6_EAC_Sistemas_Embarcados/apoio_SE_teorias_conceitos.pdf. Acesso em: 3 maio 2025.

BARR, M.; MASSA, A. Programming Embedded Systems: with C and GNU Development Tools. 2. ed. Sebastopol: O'Reilly Media, 2006. Disponível em: https://www.oreilly.com/library/view/programming-embedded-systems/0596009836/. Acesso em: 3 maio 2025.

BLACK MAGIC PROBE. Black Magic Probe Documentation. GitHub, 2024. Disponível em: https://github.com/blacksphere/blackmagic. Acesso em: 3 maio 2025.

CASTRO, F. M.; PACHECO, M. A. F.; PINHEIRO, E. C. N. M. Indústria 4.0 e internet industrial das coisas – (IIOT): sistema IoT para monitoramento e controle de sensores e atuadores mapeados. Ciências Exatas e da Terra, v. 27, n. 122, 2023.

CRISTIANO MARÇAL TONIOLO. Sistemas embarcados. 2018. Disponível em: http://cm-kls-content.s3.amazonaws.com/201801/INTERATIVAS_2_0/SISTEMAS_EMBARCADOS/U1/LIVRO_UNICO.pdf. Acesso em: 3 maio 2025.

DEMICHELI, G. Hardware/software co-design: application domains and design technologies. In: DEMICHELI, G.; SAMI, M. (Ed.). *Hardware/software co-design*. Dordrecht: Kluwer Academic Publishers, 1996. p. 1–28.

GANSSLE, J. The Art of Designing Embedded Systems. 2. ed. Burlington: Newnes, 2008.

HEATH, S. Embedded Systems Design. 2. ed. Burlington: Newnes, 2002. Disponível em:

https://www.sciencedirect.com/book/9780750655460/embedded-systems-design. Acesso em: 3 maio 2025.

LIASCH, J. Os sistemas de comunicação de uma aeronave moderna. 2025. Disponível em: http://culturaaeronautica.blogspot.com/2014/06/os-sistemas-de-comunicacao-de-uma.html. Acesso em: 5 maio 2025.

LABROSSE, J. J. MicroC/OS-II: The Real-Time Kernel. CMP Books, 2002.

MARWEDEL, P. Embedded System Design: Embedded Systems Foundations of Cyber-Physical Systems. 2. ed. Heidelberg: Springer, 2010.

OPENOCD. Open On-Chip Debugger. 2024. Disponível em: http://openocd.org/. Acesso em: 3 maio 2025.

KIM, J. et al. Performance Evaluation of Zone-Based In-Vehicle Network Architecture for Autonomous Vehicles. Sensors, [S. l.], v. 23, n. 2, p. 669, 2023. DOI: [10.3390/s23020669](https://doi.org/10.3390/s23020669). Acesso em: 09 jul. 2025.

PERCEPIO AB. Tracealyzer for FreeRTOS, Zephyr and other RTOS. Västerås, Suécia, 2024. Disponível em: https://percepio.com/tracealyzer/. Acesso em: 1 maio 2025.

PROLIFE. Quais são os parâmetros básicos de um monitor multiparâmetro? 2025. Disponível em: https://prolife.com.br/quais-sao-os-parametros-basicos-de-um-monitor-multiparametro/. Acesso em: 3 maio 2025.

REIS, F. Introdução aos sistemas embarcados. 2015. Disponível em: https://www.bosontreinamentos.com.br/eletronica/eletronica-geral/introducao-aos-sistemas-embarcados/. Acesso em: 6 maio 2025.

SALEAE. Saleae Logic Analyzers. 2024. Disponível em: https://www.saleae.com/. Acesso em: 3 maio 2025.

SEGGER MICROCONTROLLER GMBH. SystemView – Real-time recording and visualization. Monheim am Rhein, 2024. Disponível em: https://www.segger.com/products/development-tools/systemview/. Acesso em: 3 maio 2025.

SILVA JÚNIOR, E. A internet das coisas e a plataforma Arduino como computação embarcada em mapas táteis: uma avaliação dessa tecnologia assistiva para o ensino das pessoas ouvintes com deficiência visual. 2018. Disponível em: https://www.professores.uff.br/screspo/wp-content/uploads/sites/127/2023/09/Elias-Disserta%C3%A7%C3%A3o-de-Mestrado.pdf. Acesso em: 13 maio 2025.

TANENBAUM, A. S.; BOS, H. Modern Operating Systems. 4. ed. Boston: Pearson, 2015.

UNIPAR. Internet das coisas. 2025. Disponível em: https://moodle.ead.unipar.br/materiais/webflow/topicos-especiais-em-sistemas-de-informacao/unidade-ii.html. Acesso em: 23 maio 2025.

VARGAS, R. F. Sistemas embarcados: acoplamento do soft-core plasma ao barramento opb de um PowerPC 405. 2007. 49 f. Trabalho de Conclusão de Curso (Bacharelado em Engenharia Elétrica) – Universidade Federal de Santa Catarina, Florianópolis.

WOLF, W. Computers as Components: Principles of Embedded Computing System Design. San Francisco: Morgan Kaufmann, 2001.

YIU, J. The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors. 3. ed. Oxford: Newnes, 2015.

KOOPMAN, Philip. Better Embedded System Software. Versão 1.1. [S. l.]: Amazon KDP, 2021. eBook. ISBN-13: 979-8596008050. Disponível em: <https://www.amazon.com/Better-Embedded-System-Software-Koopman-ebook/dp/B09NQZJQ6S>. Acesso em: 08 jul. 2025.

AMIANTI, Giovani. Arquitetura de software aviônico de um VANT com requisitos de homologação. 2008. Dissertação (Mestrado em Engenharia de Controle e Automação Mecânica) - Escola Politécnica, Universidade de São Paulo, São Paulo, 2008. doi:10.11606/D.3.2017.tde-30052008-125557. Acesso em: 2025-10-24.

Hong W. E.; et Al. RTLinux based Hard Real-Time Software Architecture for Unmanned Autonomous Helicopters. In: International Conference on Embedded and Real-Time Computing Systems and Applications, n.11, 2005.

BORGES, Rodrigo Weissmann. Aplicabilidade de sistemas operacionais de tempo real (RTOS) para sistemas embarcados de baixo custo e pequeno porte. 2011. Dissertação (Mestrado em Processamento de Sinais e Instrumentação) - Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2011. doi:10.11606/D.18.2011.tde-09082011-081631. Acesso em: 2025-10-24.

ANEXOS

Anexo I: Software desenvolvido para a realização dos ensaios

```
/* USER CODE BEGIN Header */

/* USER CODE END Header */
/* Includes
------------------------------------------------------------------
----*/
#include "main.h"
#include "cmsis_os.h"
#include "adc.h"
#include "dma.h"
#include "tim.h"
#include "usart.h"
#include "gpio.h"

/* Private includes
----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "stdio.h"
#include "string.h"
#include "stm32f4xx_hal_uart.h"
/* USER CODE END Includes */

/* Private typedef
-----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define
------------------------------------------------------------*/
/* USER CODE BEGIN PD */
```

```
/* USER CODE END PD */

/* Private macro
-----------------------------------------------------------*/
/* USER CODE BEGIN PM */
#define DELTADEBUG
#define BLOCKINGDEBUG
#define DMADEBUG
/* USER CODE END PM */

/* Private variables
---------------------------------------------------------*/

/* USER CODE BEGIN PV */
osThreadId_t controlTaskHandle, buttonTaskHandle,
UARTTaskHandle;
osMessageQueueId_t debugQueue;
typedef struct {
     uint32_t adc_data;
     uint32_t pwm_value;
     float pid_value;
     float error;
#ifdef DELTADEBUG
     uint32_t delta_ms;
     uint32_t execution_ms;
     uint32_t delta_us;
     uint32_t execution_us;
#endif
} DebugData;

typedef struct {
     float Kp;
     float Ki;
```

```
    float Kd;
    float setpoint;
    float integral;
    float prev_error;
}PID_Controller;

PID_Controller pid = {
    .Kp = 0.5f,
    .Ki = 0.1f,
    .Kd = 0.05f,
    .setpoint = 22.0f,  // Setpoint para ADC de 12 bits
(0-4095)
    .integral = 0.0f,
    .prev_error = 0.0f
};
osMessageQueueId_t debugQueue;

volatile uint64_t microSecondCounter =0;

#ifdef DMADEBUG
#define UART_TX_BUFFER_SIZE 256
uint8_t uartTxBuffer[UART_TX_BUFFER_SIZE];
volatile uint8_t uartTxBusy = 0;
#endif

/* USER CODE END PV */

/* Private function prototypes
-----------------------------------------------*/
void SystemClock_Config(void);
void MX_FREERTOS_Init(void);
/* USER CODE BEGIN PFP */

void control_task(void *argument);
```

```
#ifndef BLOCKINGDEBUG
void UART_task(void *argument);
#endif
float calculate_pid(uint32_t adc_value);

uint32_t GetMicroseconds(void);

#ifdef DMADEBUG
void UART_Send_DMA(UART_HandleTypeDef huart, uint8_t *data,
uint16_t len);
#endif

/* USER CODE END PFP */

/* Private user code
---------------------------------------------------------*/
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/**
  * @brief  The application entry point.
  * @retval int
  */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */

  /* MCU Configuration--------------------------------*/

  /* Reset of all peripherals, Initializes the Flash interface
and the Systick. */
```

```
HAL_Init();

/* USER CODE BEGIN Init */

/* USER CODE END Init */

/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_DMA_Init();
MX_USART2_UART_Init();
MX_TIM2_Init();
MX_ADC1_Init();
MX_TIM4_Init();
/* USER CODE BEGIN 2 */

HAL_TIM_Base_Start_IT(&htim4);

   //Queue de mensagens entre tasks
   const osMessageQueueAttr_t debugQueue_attributes = {
             .name = "debugQueue",
             .attr_bits = 0,
             .cb_mem = NULL,
             .cb_size = 0,
             .mq_mem = NULL,
             .mq_size = 0
   };
```

```
    debugQueue = osMessageQueueNew(10, sizeof(DebugData),
&debugQueue_attributes);

    //Task de controle de temperatura
    const osThreadAttr_t controlTask_attributes = {
            .name = "controlTask",
            .stack_size = 512 * 4,  // 512 words = 2048
bytes
            .priority = (osPriority_t) osPriorityHigh,
    };
    controlTaskHandle = osThreadNew(control_task, NULL,
&controlTask_attributes);

#ifndef BLOCKINGDEBUG
    //Task de depuração de dados
    const osThreadAttr_t UARTTask_attributes = {
            .name = "UARTTask",
            .stack_size = 256 * 4,
            .priority = (osPriority_t) osPriorityNormal,
    };

    UARTTaskHandle = osThreadNew(UART_task, NULL,
&UARTTask_attributes);
#endif

  /* USER CODE END 2 */

  /* Init scheduler */
  osKernelInitialize();  /* Call init function for freertos
objects (in freertos.c) */
  MX_FREERTOS_Init();

  /* Start scheduler */
  osKernelStart();
```

```
  /* We should never get here as control is now taken by the
scheduler */
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
     while (1)
     {
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
     }
  /* USER CODE END 3 */
}

/**
  * @brief System Clock Configuration
  * @retval None
  */
void SystemClock_Config(void)
{
  RCC_OscInitTypeDef RCC_OscInitStruct = {0};
  RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

  /** Configure the main internal regulator output voltage
  */
  __HAL_RCC_PWR_CLK_ENABLE();

__HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE2);

  /** Initializes the RCC Oscillators according to the
specified parameters
  * in the RCC_OscInitTypeDef structure.
  */
  RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSE;
```

```
  RCC_OscInitStruct.HSEState = RCC_HSE_BYPASS;
  RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
  RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSE;
  RCC_OscInitStruct.PLL.PLLM = 4;
  RCC_OscInitStruct.PLL.PLLN = 84;
  RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV2;
  RCC_OscInitStruct.PLL.PLLQ = 7;
  if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
  {
    Error_Handler();
  }

  /** Initializes the CPU, AHB and APB buses clocks
  */
  RCC_ClkInitStruct.ClockType =
RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK

|RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
  RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
  RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
  RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV2;
  RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

  if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_2)
!= HAL_OK)
  {
    Error_Handler();
  }
}

/* USER CODE BEGIN 4 */
DebugData debug_msg;
void control_task(void *argument)
{
```

```
      (void) argument;

      // Variáveis locais
      uint32_t adc_value = 0;
      float adc_mv = 0.0f;
      float temperature_c = 0.0f;
      uint32_t pwm_value = 0;
      float pid_output = 0.0f;
      static DebugData debug_data;
      pid.setpoint = 25.0f;

      // inicialização de hardware
      HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);
      HAL_ADC_Start(&hadc1);

#ifdef DELTADEBUG
    // Variáveis para medição de tempo usando ticks do RTOS
    static uint32_t last_start_tick = 0;
    static uint32_t last_start_us = 0;
    uint32_t start_tick, end_tick, delta_tick, execution_tick
= 0;
    uint32_t start_us, end_us, delta_us, execution_us = 0;
    uint32_t delta_ms, execution_ms = 0;

    last_start_tick = osKernelGetTickCount();
    last_start_us = GetMicroseconds();
#endif

      for (;;) {

#ifdef DELTADEBUG
        // Marcar o INICIO da execução
        start_tick = osKernelGetTickCount();
        start_us = GetMicroseconds();
```

```
        // Calcular o delta desde o último INICIO
        delta_tick = start_tick - last_start_tick;
        delta_ms = delta_tick;  // Cada tick = 1ms

        delta_us = start_us - last_start_us;
#endif

          // 1. Leitura do ADC
          //for (volatile uint16_t i = 0; i < 1000; i++);  //
Simulação de carga de processamento
          HAL_ADC_Start(&hadc1);
          if(HAL_ADC_PollForConversion(&hadc1, 10) == HAL_OK){
               adc_value = HAL_ADC_GetValue(&hadc1);
          }
          HAL_ADC_Stop(&hadc1);

          //VDDA = 3.3V
          adc_mv = (adc_value * 3300.0f) / 4095.0f;

          //Temperatura = ((V_sense-V_25) / Avg_slope) + 25
          //V_25 = 760mV
          //Avg_slope = 2.5mV/°C
          temperature_c = ((adc_mv - 760.0f) / 2.5f) + 25.0f;

          // 2. Cálculo do PID
          pid_output = calculate_pid(temperature_c);

          // 3. Ajuste do PWM
          pwm_value = (uint32_t)(pid_output * 10);
          if (pwm_value > 1000) pwm_value = 1000;   // Limitar
valor máximo
          __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1,
pwm_value);
```

```
#ifdef BLOCKINGDEBUG
          char buffer[256];
          int len = snprintf(buffer, sizeof(buffer),
                "Temp:%luC|PWM:%lu|PID:%.2f|Error:%.2fC|"
                "RTOS:Delta=%lums Exec:%lums|
Timer:Delta=%luus Exec=%luus\r\n",
                debug_data.adc_data,
                debug_data.pwm_value,
                debug_data.pid_value,
                debug_data.error,
                debug_data.delta_ms,
                debug_data.execution_ms,
                    debug_data.delta_us,
                    debug_data.execution_us);
#ifdef DMADEBUG
          UART_Send_DMA(huart2, (uint8_t*)buffer, len);
#else
          HAL_UART_Transmit(&huart2, (uint8_t*)buffer, len,
HAL_MAX_DELAY);
#endif
#endif

#ifdef DELTADEBUG
        // Marcar o FIM da execução
        end_tick = osKernelGetTickCount();
        end_us = GetMicroseconds();

        execution_tick = end_tick - start_tick;
        execution_ms = execution_tick;  // Cada tick = 1ms
        execution_us = end_us - start_us;
#endif
```

```
            // 4. Preparar dados de depuracao
            debug_data.adc_data = temperature_c;
            debug_data.pwm_value = pwm_value;
            debug_data.pid_value = pid_output;
            debug_data.error = pid.setpoint - temperature_c;

#ifdef DELTADEBUG
            debug_data.delta_ms = delta_ms;
            debug_data.execution_ms = execution_ms;

            debug_data.delta_us = delta_us;
            debug_data.execution_us = execution_us;
#endif

#ifndef BLOCKINGDEBUG
            // 5. Enviar dados para a task UART
            if (osMessageQueuePut(debugQueue, &debug_data, 0, 0)
!= osOK) {
                  //Timeout
            }
#endif

#ifdef DELTADEBUG
            last_start_tick = start_tick;
            last_start_us = start_us;
#endif
            osDelay(5);
      }
}

// Função de cálculo PID
float calculate_pid(uint32_t temperature) {
    float error = pid.setpoint - temperature;
    float p_term = pid.Kp * error;
```

```
    // Termo integral limitado
    if(pid.integral > 100.0f || pid.integral < -100.0f){
     pid.integral =100.0f;
    }else{
     pid.integral += error;
    }
    //for (volatile uint16_t i = 0; i < 500; i++);  //
Simulação de atraso
    float i_term = pid.Ki * pid.integral;

    // Termo derivativo
    float d_term = pid.Kd * (error - pid.prev_error);
    //for (volatile uint16_t i = 0; i < 300; i++);  //
Simulação de atraso

    pid.prev_error = error;

    // Saída do PID com limites
    float output = p_term + i_term + d_term;
    if (output > 100.0f) output = 100.0f;
    if (output < 0.0f) output = 0.0f;

    return output;
}
#ifndef BLOCKINGDEBUG
void UART_task(void *argument)
{
     (void) argument;

     DebugData debug_data;
     char buffer[256];

     for(;;)
```

```
    {
        if(osMessageQueueGet(debugQueue, &debug_data, NULL,
osWaitForever) == osOK){
#ifdef DELTADEBUG
            int len = snprintf(buffer, sizeof(buffer),
                "Temp:%luC|PWM:%lu|PID:%.2f|Error:%.2fC|"
                "RTOS:Delta=%lums Exec:%lums|
Timer:Delta=%luus Exec=%luus\r\n",
                debug_data.adc_data,
                debug_data.pwm_value,
                debug_data.pid_value,
                debug_data.error,
                debug_data.delta_ms,
                debug_data.execution_ms,
                    debug_data.delta_us,
                    debug_data.execution_us);
#else
            int len = snprintf(buffer, sizeof(buffer),
                "Temp:%luC|PWM:%lu|PID:%.2f|Error:%.2fC|"
                "\r\n",
                debug_data.adc_data,
                debug_data.pwm_value,
                debug_data.pid_value,
                debug_data.error);
#endif

#ifdef DMADEBUG
            UART_Send_DMA(huart2, (uint8_t*)buffer, len);
#else
            HAL_UART_Transmit(&huart2, (uint8_t*)buffer,
len, HAL_MAX_DELAY);
#endif

        }else{
```

```
                sprintf(buffer, "Queue free\r\n");
                HAL_UART_Transmit(&huart2, (uint8_t*)buffer,
strlen(buffer), 100);
          }

          osDelay(200);
     }
}
#endif

#ifdef DELTADEBUG
uint32_t GetMicroseconds(void){
     return microSecondCounter;
}
#endif

#ifdef DMADEBUG
volatile uint32_t debug_DMA_Busy = 0;
void UART_Send_DMA(UART_HandleTypeDef huart, uint8_t *data,
uint16_t len)
{
    // Copiar os dados para o buffer de TX
    if(len > UART_TX_BUFFER_SIZE) {
        len = UART_TX_BUFFER_SIZE;
    }
    memcpy(uartTxBuffer, data, len);

    // Iniciar transferência por DMA
    if(HAL_UART_Transmit_DMA(&huart, uartTxBuffer, len) ==
HAL_BUSY){
     debug_DMA_Busy++;
    }

}
```

```
#endif

/* USER CODE END 4 */

/**
  * @brief  Period elapsed callback in non blocking mode
  * @note   This function is called  when TIM3 interrupt took
place, inside
  * HAL_TIM_IRQHandler(). It makes a direct call to
HAL_IncTick() to increment
  * a global variable "uwTick" used as application time base.
  * @param  htim : TIM handle
  * @retval None
  */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  /* USER CODE BEGIN Callback 0 */

  /* USER CODE END Callback 0 */
  if (htim->Instance == TIM3) {
    HAL_IncTick();
  }
  /* USER CODE BEGIN Callback 1 */

  /* USER CODE END Callback 1 */
}

/**
  * @brief  This function is executed in case of error
occurrence.
  * @retval None
  */
void Error_Handler(void)
```

```
{
  /* USER CODE BEGIN Error_Handler_Debug */
     /* User can add his own implementation to report the HAL
error return state */
     __disable_irq();
     while (1)
     {
     }
  /* USER CODE END Error_Handler_Debug */
}

#ifdef  USE_FULL_ASSERT
/**
  * @brief  Reports the name of the source file and the source
line number
  *         where the assert_param error has occurred.
  * @param  file: pointer to the source file name
  * @param  line: assert_param error line source number
  * @retval None
  */
void assert_failed(uint8_t *file, uint32_t line)
{
  /* USER CODE BEGIN 6 */
     /* User can add his own implementation to report the file
name and line number,
     ex: printf("Wrong parameters value: file %s on line
%d\r\n", file, line) */
  /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```