UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO DE JOINVILLE
ENGENHARIA MECATRÔNICA

VITOR JOSÉ DUARTE QUINTANS

COMPARAÇÃO ENTRE RTOS CONSTRUÍDO EM C E RUST

Joinville
2024

VITOR JOSÉ DUARTE QUINTANS

COMPARAÇÃO ENTRE RTOS CONSTRUÍDO EM C E RUST

Trabalho apresentado como requisito parcial para obtenção do título de bacharel em Engenharia Mecatrônica, no Curso de Engenharia Mecatrônica, do Centro Tecnológico de Joinville, da Universidade Federal de Santa Catarina.

Orientador: Gian Ricardo Berkenbrock, Dr.

Joinville
2024

Este trabalho é dedicado aos meus colegas de classe e aos meus queridos pais.

Comparação entre RTOS construído em C e Rust

AGRADECIMENTOS

Em primeiro lugar, gostaria de expressar minha gratidão aos meus pais, Adson e Michella. Sem o apoio incondicional e os sábios conselhos que me proporcionaram ao longo da vida, não teria conseguido chegar até este momento. Seus exemplos de dedicação foram fundamentais para o meu desenvolvimento pessoal e acadêmico.

Gostaria também de agradecer ao meu orientador, Dr. Gian Ricardo Berkenbrock, pelo conhecimento compartilhado e pela orientação prestada durante a elaboração deste trabalho. Sua paciência e comprometimento foram essenciais para a concretização deste projeto.

Estendo meus agradecimentos aos meus colegas do Laboratório de Sistemas Embarcados (LSE) pela colaboração e apoio ao longo deste percurso. A convivência e as trocas de experiências com cada um de vocês enriqueceram este trabalho. Os momentos compartilhados no laboratório foram fundamentais para superar os desafios enfrentados.

Por fim, agradeço sinceramente aos amigos que estiveram ao meu lado ao longo de toda a minha jornada acadêmica. Os momentos de descontração e apoio emocional proporcionados por vocês foram essenciais para que eu pudesse manter o equilíbrio e seguir em frente nos momentos mais difíceis.

“ A mudança é a lei da vida. Aqueles que olham apenas para o passado ou para o presente serão esquecidos no futuro ”

John Kennedy

RESUMO

O uso de linguagens sem segurança de memória em sistemas operacionais de tempo real embarcados pode ocasionar diversas falhas críticas, comprometendo a confiabilidade e a segurança do sistema. Como alternativa, linguagens atuais oferecem garantias de segurança de memória, uma delas é Rust. Contudo, aplicações em sistemas embarcados enfrentam desafios devido a restrições temporais e limitações de hardware. Nesse contexto, esse trabalho apresenta uma análise comparativa do desempenho de sistemas embarcados de tempo real implementados em Rust, utilizando o framework Embassy, e em C, utilizando o FreeRTOS. O objetivo é avaliar o impacto da escolha da linguagem de programação e do sistema operacional de tempo real nas métricas de latência, *jitter* e uso de recursos em aplicações críticas. Foram realizados experimentos em dois cenários, medindo-se latência e *jitter* em sistemas com diferentes cargas de trabalho. Os resultados indicam que, para os cenários desenvolvidos e com ambos os RTOS na configuração padrão, o sistema em Rust com Embassy apresentou menor latência e *jitter*, além de utilizar menos recursos de memória e armazenamento, em comparação com o sistema em C com FreeRTOS.

Palavra-chave: Sistemas embarcados. Sistemas de tempo real. Rust. C. Latência.

ABSTRACT

The use of languages without memory safety in real-time operating systems for embedded systems can lead to various critical failures, compromising the reliability and safety of the system. As an alternative, modern languages offer memory safety guarantees, one of which is Rust. However, applications in embedded systems face challenges due to time constraints and hardware limitations. In this context, this work presents a comparative analysis of the performance of real-time embedded systems implemented in Rust, using the Embassy framework, and in C, using FreeRTOS. The objective is to evaluate the impact of the choice of programming language and real-time operating system on latency, jitter, and resource usage metrics in critical applications. Experiments were conducted in two scenarios, measuring latency and jitter in systems under different workloads. The results indicate that, in the proposed scenarios and with both systems in their default configurations, the Rust system with Embassy demonstrated lower latency and jitter, while also consuming less memory and storage compared to the C system with FreeRTOS.

Keywords: Embedded systems. Real-time systems. Rust. C. Latency.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE CÓDIGOS

SUMÁRIO

1 INTRODUÇÃO

A representatividade do software em sistemas mecatrônicos tem se intensificado nos últimos anos, à medida que a complexidade desses sistemas cresce em setores críticos. Um exemplo é o Ford GT, um veículo moderno que utiliza aproximadamente dez milhões de linhas de código em seus sistemas embarcados, integrando desde o controle do motor até funções de entretenimento e segurança (MOTOR AUTHORITY, 2023). Esse aumento na quantidade e sofisticação do código eleva as exigências sobre a confiabilidade e segurança do software, especialmente em indústrias como a automotiva, onde a integração de múltiplos sistemas é essencial.

Nessa classe de equipamentos, uma falha pode ter consequências graves, tornando imperativa a adoção de técnicas robustas de programação e controle para mitigar riscos, justificando a nomenclatura de tais como sistemas críticos (WANG et al., 2022). Em sistemas críticos, como veículos autônomos, equipamentos médicos e plantas industriais, não basta apenas garantir a eficiência no uso de recursos; é indispensável assegurar previsibilidade e segurança. Nesses contextos, uma ferramenta utilizada são sistemas operacionais de tempo real, do inglês *Real Time Operation Systems* (RTOS) (BUTTAZZO, 1997).

Os RTOS são ferramentas projetadas para gerenciar recursos e garantir que tarefas críticas sejam completadas em prazos estritos, dado que a validade dessas operações depende não apenas do valor do resultado, mas também do momento em que ele é produzido (BUTTAZZO, 1997). Marwedel (2021) enfatiza que RTOS são fundamentais em aplicações onde a previsibilidade e a pontualidade são essenciais, como em sistemas embarcados críticos, automação industrial e controle de processos. Os RTOS, portanto, desempenham um papel central na gestão dessas tarefas, fornecendo mecanismos que garantem a execução de atividades nos prazos específicos exigidos em sistemas embarcados.

Mesmo que os algoritmos que compõem os RTOS sejam rigorosamente validados, os blocos construtivos básicos desses sistemas ainda são as linguagens de programação. Para essa aplicação, utilizam-se as linguagens de sistemas, caracterizadas por produzir programas que interage diretamente com hardware (NAHAS; MAAITA, 2012). Adicionalmente, outras propriedades desejáveis para linguagens em sistemas de tempo real incluem eficiência para atender a limitações de recursos, suporte à criação de bibliotecas reutilizáveis e portabilidade entre diferentes processadores (PONT, 2003).

Alguns exemplos de linguagens a que atendem a tais requisitos são: Assembly, C, C++, Ada, Java e Rust, dentre as quais, segundo Nahas e Maaita (2012), o C destaca-se historicamente por sua ampla adoção nesses sistemas. Contudo, essa

linguagem apresenta operações primitivas, como o gerenciamento manual de memória, que podem levar a falhas críticas, como estouros de buffer e condições de corrida. Esse contexto evidencia a necessidade de explorar alternativas que ofereçam maior segurança e previsibilidade sem comprometer o desempenho essencial para tarefas em tempo real.

> Para novos produtos, escolher desenvolver em uma linguagem de programação com segurança de memória é uma decisão arquitetural inicial que pode trazer benefícios significativos de segurança. Mesmo para bases de código existentes, onde uma reescrita completa do código é mais desafiadora, ainda existem caminhos para adotar linguagens de programação com segurança de memória por meio de uma abordagem híbrida. (OFFICE OF THE NATIONAL CYBER DIRECTOR, 2024, tradução nossa)

A linguagem Rust, por exemplo, tem sido apontada como uma alternativa promissora, ao conciliar a eficiência típica de linguagens de baixo nível com garantias de segurança em tempo de compilação, minimizando falhas comuns como condições de corrida e vazamentos de memória (RUST ON EMBEDDED DEVICES WORKING GROUP et al., 2024). Todavia, embora Rust ofereça vantagens em termos de segurança, é necessário avaliar os impactos que a adoção de novas tecnologias pode ter sobre o desempenho do sistema, especialmente em sistemas embarcados onde recursos são limitados e a eficiência é fundamental.

Nesse contexto, este trabalho propõe uma análise comparativa de propriedades temporais e uso de recursos entre dois RTOS implementados em C (FreeRTOS) e Rust (Embassy RTOS), com foco em aplicações embarcadas. Será apresentada a construção e implementação de um sistema de controle de tempo real, utilizado para analisar o comportamento dos RTOS em condições próximas ao uso real. Posteriormente, métricas como latência, jitter, tempo de resposta de pior caso, e uso de memória serão mensuradas para comparar ambas as implementações.

1.1 OBJETIVO GERAL

Comparar propriedades temporais e uso de recursos entre RTOS para sistemas embarcados desenvolvidos em linguagem de programação C e Rust.

1.2 OBJETIVOS ESPECÍFICOS

- Estudar as propriedades de sistemas de tempo real embarcados;
- Definir sistema de teste para comparação entre sistemas de tempo real embarcados;
- Construir cenários de teste;
- Analisar o desempenho temporal e use de recurso de cada RTOS.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, é realizada uma revisão teórica dos conceitos essenciais para a condução deste estudo. Primeiramente, apresentam-se as linguagens de programação para sistemas embarcados, com ênfase na linguagem Rust. Em seguida, discutem-se os sistemas operacionais de tempo real (RTOS), abordando seus princípios, requisitos e diferenças em relação a sistemas convencionais. Também são introduzidas algumas métricas para avaliação de tais sistemas, utilizadas para analisar o desempenho dos RTOS. Além disso, explora-se o controle em malha fechada, elemento-chave para aplicações críticas em ambiente embarcado. Por fim, apresenta-se a literatura relacionada, consolidando a base teórica necessária para o desenvolvimento do trabalho.

2.1 LINGUAGENS DE PROGRAMAÇÃO PARA SISTEMAS EMBARCADOS

Sistemas embarcados são sistemas integrados que interagem diretamente com processos físicos, desempenhando papéis fundamentais em uma variedade de aplicações, desde eletrodomésticos e automóveis até dispositivos médicos e sistemas industriais (MARWEDEL, 2021). Devido à natureza desses processos, tais sistemas frequentemente apresentam requisitos de tempo e desempenho, os quais demandam uma escolha da linguagem de programação (MARWEDEL, 2021). A linguagem selecionada influencia diretamente a eficiência, a confiabilidade e a capacidade de resposta do sistema, sendo, portanto, um fator crítico no desenvolvimento de sistemas embarcados (NAHAS; MAAITA, 2012).

Historicamente, *Assembly* foi a primeira linguagem frequentemente aplicada a sistemas embarcados, fornecendo controle direto sobre o hardware. Contudo, seu uso apresentou limitações, uma vez que ambientes de desenvolvimento baseados em *Assembly* careciam de ferramentas robustas para depuração e testes, o que dificultava o desenvolvimento e a manutenção de sistemas complexos (HALANG; STOYENKO, 1990). Nos anos 1960, essa limitação levou à introdução de linguagens de alto nível para sistemas embarcados, trazendo vantagens como maior facilidade de aprendizado, programação, entendimento, depuração, manutenção e portabilidade de código (BOULTON; REID, 1969).

Para que uma linguagem de alto nível atendesse aos requisitos dos sistemas em tempo real, alguns critérios essenciais foram definidos. De acordo com Boulton e Reid (1969), esses critérios incluem métodos para lidar com sinais e interrupções em tempo real, além de mecanismos para o escalonamento de tarefas. Essas características são indispensáveis para garantir a previsibilidade e o controle dos tempos

de resposta, aspectos fundamentais para a operação segura e confiável de sistemas embarcados em contextos críticos.

Com o avanço das linguagens de programação ao longo das décadas, novas linguagens surgiram com características que visam melhorar a segurança, a confiabilidade e a eficiência em sistemas de tempo real. O C, por exemplo, emergiu como a principal linguagem para sistemas embarcados devido à sua eficiência e acesso direto ao hardware. No entanto, suas limitações, como a ausência de mecanismos de segurança automáticos, evidenciam a necessidade de explorar alternativas que conciliem controle de baixo nível com segurança e previsibilidade, a exemplo de Ada e Rust (NAHAS; MAAITA, 2012).

2.1.1 Rust

O Rust é uma linguagem de programação lançada oficialmente em 2015, desenvolvida para conciliar eficiência em tempo de execução com um extenso suporte ao desenvolvedor (KLABNIK; NICHOLS, 2023). Essa foi projetada para superar as limitações de linguagens como C e C++, oferecendo controle de baixo nível sobre o hardware, sem comprometer a segurança e a facilidade de uso (KLABNIK; NICHOLS, 2023). Neste contexto, serão abordadas algumas das características que fazem do Rust uma escolha promissora para o desenvolvimento de sistemas de baixo nível, especialmente em aplicações onde a segurança e a eficiência são cruciais.

Uma das principais características do Rust é sua capacidade de garantir a segurança de memória sem comprometer o desempenho. O gerenciamento de memória é um elemento fundamental de qualquer linguagem de programação que permita alocação dinâmica. Algumas linguagens utilizam um mecanismo denominado garbage collector, que verifica as regiões de memória não utilizadas para liberá-las. No entanto, essa abordagem pode resultar em consumo excessivo de recursos e imprevisibilidade de tempo. Outras linguagens exigem que o programador libere explicitamente as regiões de memória que não serão mais utilizadas, o que pode levar a erros de acesso e vazamentos de memória.

Rust, por sua vez, implementa uma terceira abordagem: um sistema denominado ownership (posse), fundamentado em três regras básicas (KLABNIK; NICHOLS, 2023):

1. Cada valor em Rust tem um proprietário.
2. Pode haver apenas um proprietário por vez.
3. Quando o proprietário sai do escopo, a memória associada ao valor é liberada.

Outra característica fundamental do ownership é a transferência de posse. Quando uma variável que possui um valor é atribuída a outra variável, a posse do valor é transferida para a nova variável, e a variável original deixa de ser válida. Isso

significa que o valor não pode ser acessado pela variável original após a transferência, evitando acessos simultâneos ou ilegais a um mesmo espaço de memória. Esses recursos permitem que os desenvolvedores escrevam código seguro, minimizando o risco de falhas críticas.

Para a implementação de sistemas complexos, a separação de responsabilidades em módulos é essencial (COSTA et al., 2019). Nesse sentido, Rust permite a separação dos programas em subelementos denominados crates, que apresentam diferentes níveis de organização e escopo. À medida que um projeto cresce, a organização do código se torna cada vez mais importante, e a separação em módulos e arquivos distintos é uma prática recomendada para facilitar a manutenção e a escalabilidade.

Os crates são árvores de módulos que produzem uma biblioteca ou executável, permitindo que o código seja agrupado de acordo com funcionalidades relacionadas e características distintas. Dessa forma, a localização do código que implementa uma funcionalidade específica é simplificada. Além disso, Rust oferece recursos que permitem encapsular os detalhes da implementação, possibilitando o reuso de código em alto nível, limitando a quantidade de detalhes que os desenvolvedores precisam ter em mente ao trabalhar no projeto.

O conceito de escopo também é fundamental em Rust. Cada contexto em que o código é escrito possui um conjunto de nomes definidos como "em escopo". Durante a leitura, escrita e compilação do código, é crucial saber se um nome em particular refere-se a uma variável, função, estrutura, enumeração, módulo, constante ou outro item, e qual é o significado desse item. Rust oferece ferramentas para criar escopos e gerenciar quais nomes estão ou não em escopo, resolvendo possíveis conflitos.

Outra característica do Rust são as traits. Traits são uma forma de definir funcionalidades compartilhadas entre diferentes tipos, permitindo que eles compartilhem comportamento comum. Elas são semelhantes ao conceito de interfaces em outras linguagens, por especificarem um conjunto de métodos que um tipo deve implementar, promovendo a reutilização de código e facilitando a criação de abstrações. Assim, permitem que funções e métodos sejam genéricos e flexíveis, desde que os tipos implementem as funcionalidades necessárias.

Rust também suporta programação assíncrona. A programação assíncrona em Rust é baseada no uso da palavra-chave async para definir funções assíncronas e da await para aguardar a conclusão dessas funções. Esse modelo permite que tarefas sejam realizadas concorrentemente sem bloquear a execução do programa, otimizando o uso de recursos. A biblioteca *async-std* e o *Tokio* são exemplos de bibliotecas que oferecem suporte para programação assíncrona, facilitando a implementação de servidores web, sistemas distribuídos, sistemas multitarefas e outras aplicações onde o processamento simultâneo é essencial.

No contexto de programação embarcada, existem dois tipos principais de ambientes: os ambientes hospedados, que se assemelham a um computador de alto nível, com uma interface de sistema e suporte a bibliotecas padrão; e os ambientes bare metal, onde não há sistema operacional e o código interage diretamente com o hardware (RUST ON EMBEDDED DEVICES WORKING GROUP et al., 2024). Em ambientes bare metal, o uso de Rust sem a biblioteca padrão (*no_std*) é comumente empregado, utilizando-se a *libcore*, uma versão mínima da biblioteca padrão que oferece suporte a funcionalidades básicas sem depender de integração com o sistema.

A Tabela 1 compara os recursos disponíveis nos ambientes *no_std* e *std* em Rust.

Tabela 1 – Comparação de ambientes std e no_std

Feature	**no_std**	**std**
Heap (memória dinâmica)	não	sim
Collections (Vec, BTreeMap, etc)	não	sim
Proteção contra estouro de pilha	não	sim
Código de inicialização	não	sim
libstd disponível	não	sim
libcore disponível	sim	sim

Fonte: Adaptado de RUST ON EMBEDDED DEVICES WORKING GROUP et al. (2024).

No ambiente *no_std*, o suporte a memória dinâmica (*heap*) e coleções como *Vec* e *BTreeMap* é possível apenas com o uso de *crates* específicos, como *alloc* para alocação de memória e *collections* para estruturas de dados, quando houver um alocador adequado. Em contraste, o ambiente *std* fornece suporte a esses recursos por padrão, além de proteção contra estouro de pilha e execução de código de inicialização antes da função *main*. A biblioteca *libcore* está disponível em ambos os ambientes, enquanto a *libstd* está disponível apenas no ambiente *std*.

Sistemas de tempo real são uma aplicação comum de sistemas embarcados. Definem-se como sistemas nos quais a correção não depende apenas dos resultados computacionais, mas também do tempo em que esses resultados são gerados, sendo essenciais em aplicações que exigem respostas a eventos externos em prazos limitados, onde a validade dos resultados está intrinsecamente ligada ao momento de sua produção (BUTTAZZO, 1997). A importância de tais cresce com o aumento de sistemas complexos controlados por computadores, como plantas industriais, sistemas de transporte, dispositivos médicos, automação industrial e sistemas de defesa.

Em muitos casos, esses sistemas são embarcados, variando desde dispositivos portáteis, como celulares, até sistemas maiores, como veículos e aeronaves, demonstrando a ampla aplicabilidade e relevância da computação em tempo real na sociedade moderna (BUTTAZZO, 1997). Tais sistemas não se destacam pela velocidade, mas pela capacidade de prever o comportamento, onde cada tarefa possui um

deadline associado. Nesse contexto, a área de sistemas de tempo real foca no desenvolvimento de técnicas que garantem e validam essa previsibilidade (BUTTAZZO, 1997).

Os sistemas de tempo real são divididos em três categorias principais, com base nas consequências de uma resposta tardia. A primeira categoria é composta pelas tarefas de tempo real rígido (*hard real-time tasks*), nas quais a produção de resultados após o deadline pode resultar em consequências catastróficas. Um exemplo comum desse tipo de tarefa encontra-se em sistemas críticos de segurança, como o controle de reatores nucleares ou a navegação de aeronaves, onde um atraso pode comprometer a segurança de todo o sistema (BUTTAZZO, 1997).

A segunda categoria inclui as tarefas de tempo real firme (*firm real-time tasks*), nas quais os resultados, se entregues após o prazo, tornam-se inúteis para o sistema, mas não acarretam danos catastróficos. Esse tipo de tarefa pode ser encontrado em sistemas multimídia ou em redes de transmissão de dados, onde o atraso na entrega de pacotes ou quadros de vídeo pode prejudicar a qualidade do serviço, mas sem impactar diretamente na integridade do sistema.

Por fim, as tarefas de tempo real suave (*soft real-time tasks*) são aquelas em que um atraso pode causar uma degradação no desempenho, mas os resultados ainda mantêm alguma utilidade. Exemplos desse tipo de tarefa incluem a interação com usuários em interfaces gráficas, como a manipulação de entrada de teclado ou a exibição de informações na tela (BUTTAZZO, 1997). A abordagem para lidar com essas tarefas envolve estratégias que minimizam o tempo médio de resposta, sem comprometer o funcionamento global do sistema.

A previsibilidade em sistemas de tempo real é alcançada por meio de mudanças nos paradigmas de projeto. Nesses sistemas, o código de cada tarefa é conhecido antecipadamente, permitindo uma análise detalhada de suas características, como tempo de computação, uso de recursos e relações de precedência com outras tarefas. Essa previsibilidade possibilita que o escalonador do sistema garanta o cumprimento de todas as tarefas críticas nos prazos especificados, ao invés de apenas minimizar o tempo médio de resposta, como ocorre em sistemas tradicionais (BUTTAZZO, 1997).

Além disso, em aplicações típicas de tempo real, as atividades de controle podem ser consideradas módulos colaborativos que operam em conjunto para alcançar um objetivo comum, a exemplo do controle de uma usina nuclear ou a navegação de uma aeronave. Nesse cenário, as tarefas não são completamente independentes, o que significa que não é necessário suportar espaços de endereço independentes para cada tarefa. As principais características desejáveis em um sistema de tempo real incluem pontualidade, previsibilidade, eficiência, robustez, tolerância a falhas e facilidade de manutenção (MARWEDEL, 2021).

A pontualidade é crucial, pois os resultados não devem ser apenas corretos

em valor, mas também no tempo em que são produzidos. Isso exige que o sistema operacional forneça mecanismos específicos de gerenciamento de tempo e suporte a tarefas com restrições temporais explícitas. A previsibilidade, por sua vez, requer que o sistema preveja as consequências de cada decisão de escalonamento. Em sistemas críticos de segurança, todas as tarefas com restrições temporais rígidas devem ser garantidas antes da operação do sistema, e se alguma tarefa não puder ser garantida, o sistema deve notificar essa falha antecipadamente, permitindo a implementação de ações alternativas.

A eficiência também é fundamental, especialmente em sistemas embarcados com restrições severas de espaço, peso, energia, memória e poder computacional. Nesses sistemas, uma gestão eficiente dos recursos disponíveis é essencial para garantir o desempenho desejado. A robustez é outra característica essencial, garantindo que o sistema não entre em colapso sob condições de carga extrema. Além disso, os sistemas de tempo real devem ser tolerantes a falhas, de modo que falhas pontuais de hardware ou software não causem a interrupção completa do sistema (BUTTAZZO, 1997).

2.2 SISTEMAS OPERACIONAIS DE TEMPO REAL (RTOS)

Sistemas de tempo real, ou sistemas computacionais com requisitos de tempo real, são aqueles que devem atender a restrições temporais rigorosas para garantir o comportamento correto. Essas restrições incluem, por exemplo, prazos máximos para execução de tarefas, periodicidade obrigatória de execução, intervalos máximos ou mínimos entre ações, e validade temporal de dados. Tais sistemas são projetados para operar dentro de limites temporais estritos, onde falhas no cumprimento dos requisitos podem comprometer a funcionalidade ou segurança do sistema (OLIVEIRA, 2020).

Para lidar com essas diferentes categorias de tarefas, os sistemas operacionais de tempo real (RTOS) precisam ser projetados com estratégias que garantam a execução nos prazos estabelecidos. Nos sistemas de tempo real rígido, é fundamental que todas as tarefas críticas sejam garantidas a priori, através de uma análise de escalonamento que assegure que cada tarefa será executada dentro de seu limite de tempo. Já para as tarefas firmes, a garantia pode ser feita dinamicamente, abortando as tarefas que não podem ser concluídas no prazo, enquanto as tarefas suaves devem ser tratadas para otimizar o tempo médio de resposta (BUTTAZZO, 1997).

A capacidade de um sistema de tempo real responder a eventos externos, como interrupções de hardware, é outro aspecto essencial. Em sistemas críticos, a previsibilidade e a eficiência no manuseio de interrupções são fundamentais para garantir que as tarefas de maior prioridade sejam atendidas primeiro, sem comprometer o desempenho de outras tarefas em execução. No entanto, é importante destacar

que a simples prioridade não é suficiente para assegurar o cumprimento de deadlines rigorosos.

Dessa forma, diversas abordagens de escalonamento são utilizadas para garantir a previsibilidade do sistema em tempo real, sendo categorizadas em quatro paradigmas principais: escalonamento estático baseado em tabela, escalonamento preemptivo com prioridade estática, escalonamento dinâmico baseado em planejamento e escalonamento dinâmico de melhor esforço (RAMAMRITHAM; STANKOVIC, 1994). O escalonamento estático baseado em tabela oferece previsibilidade ao determinar antecipadamente os tempos de execução de cada tarefa, mas sua inflexibilidade o torna menos adequado para sistemas dinâmicos.

Já o escalonamento preemptivo com prioridade estática é usado para garantir que tarefas com prazos mais curtos tenham prioridade. Contudo, quando há incerteza sobre o comportamento das tarefas, o escalonamento dinâmico baseado em planejamento e o escalonamento dinâmico de melhor esforço surgem como alternativas para lidar com situações imprevisíveis, embora apresentem trade-offs em termos de complexidade e garantias formais (RAMAMRITHAM; STANKOVIC, 1994).

2.3 MÉTRICAS PARA AVALIAÇÃO DE SISTEMAS DE TEMPO REAL

Nesta seção, apresentamos métricas utilizadas para avaliar o desempenho e a eficiência de RTOS. Compreender esses indicadores é fundamental para analisar aspectos críticos de sistemas embarcados, especialmente em contextos com recursos limitados e onde a previsibilidade temporal é vital. As métricas abordadas dividem-se em análise temporal, incluindo latência e jitter, e uso de recursos, representado pelo consumo de memória e número de símbolos.

2.3.1 Latência

A latência é definida como o intervalo de tempo entre o momento em que um evento externo ocorre (por exemplo, uma interrupção) e o momento em que o sistema começa a processar esse evento. Em sistemas de tempo real, minimizar a latência é crucial para garantir respostas a eventos críticos nos prazos estabelecidos (AROCA et al., 2009).

2.3.2 Jitter

O jitter refere-se à variação não determinística nos tempos de latência entre eventos consecutivos (AROCA et al., 2009). Em outras palavras, é a inconsistência no tempo de resposta do sistema a eventos que ocorrem periodicamente. Em aplicações sensíveis ao tempo, como controle de motores ou sistemas de comunicação, um jitter elevado pode comprometer a precisão e a estabilidade do sistema.

2.3.3 Tamanho do Programa

O tamanho do programa refere-se à quantidade de memória ocupada pelo código executável e pelos dados necessários para a sua operação. Em sistemas embarcados, onde a memória disponível é um recurso escasso, essa métrica é fundamental para avaliar a eficiência do código gerado pelo compilador. Programas menores tendem a permitir o uso de microcontroladores com menos capacidade, reduzindo custos.

2.3.4 Uso de Memória da Pilha

A memória da pilha é utilizada para o armazenamento de variáveis locais, parâmetros de funções e controle de fluxo durante a execução das tarefas. O uso eficiente dessa memória é vital em sistemas embarcados de tempo real, pois o esgotamento da pilha pode resultar em falhas críticas ou comportamentos imprevisíveis. A gestão da pilha é especialmente importante em sistemas com múltiplas tarefas concorrentes.

2.3.5 Quantidade de Símbolos

A quantidade de símbolos no código compilado, incluindo nomes de funções, variáveis e outros identificadores, é uma métrica relevante para a análise da complexidade do programa. Um número elevado de símbolos pode indicar maior complexidade e potencial sobrecarga nos processos de compilação e execução, o que pode ser indesejável em sistemas críticos de tempo real.

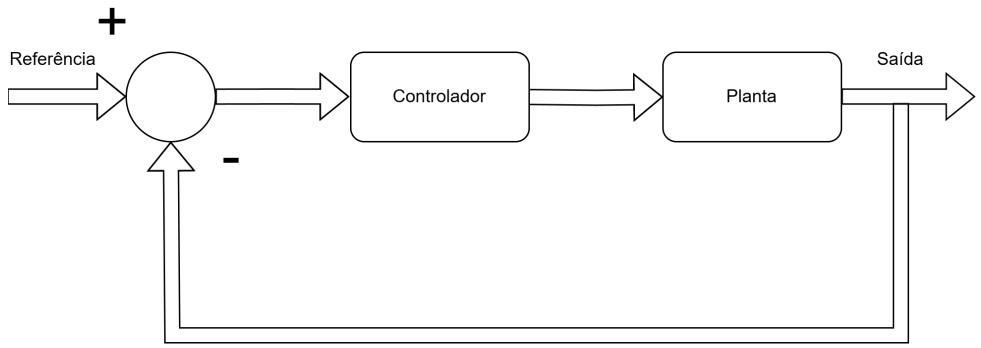
2.4 CONTROLE EM MALHA FECHADA

Uma das aplicações de sistemas de tempo real é o controle de malha fechada, o qual é uma abordagem para o desenvolvimento de sistemas que requerem precisão e estabilidade em resposta a variações e distúrbios externos. Nesse tipo de controle, o sistema utiliza informações do processo, obtidas por sensores, para ajustar continuamente a variável de controle e garantir que a saída esteja conforme um valor de referência. Dessa forma, os controladores em malha fechada são fundamentais em aplicações que necessitam de alto desempenho e robustez, uma vez que corrigem desvios e reduzem o impacto de incertezas e perturbações sobre o sistema (OGATA et al., 2009).

A Figura 1 ilustra um exemplo típico de um sistema de controle em malha fechada. Neste diagrama, pode-se observar o fluxo de informações entre os principais componentes do sistema: o sensor, que mede a saída e fornece a retroalimentação; o controlador, que ajusta a variável de controle com base no erro; e a planta, sendo o sistema a ser controlado. O objetivo do controlador é minimizar a diferença entre a

saída medida e o valor de referência, garantindo que o sistema atinja o comportamento desejado.

Figura 1 – Diagrama de blocos de controle em malha fechada



Fonte: Elaborado pelo autor (2024)

Dentre os controladores utilizados em malha fechada, destaca-se o controle Proporcional-Integral-Derivativo (PID), empregado em sistemas industriais e embarcados devido à sua simplicidade e eficiência. O controlador PID ajusta a variável de controle com base em três componentes: uma ação proporcional ao erro instantâneo, uma ação integral que considera o histórico de erros acumulados e uma ação derivativa que antecipa mudanças futuras ao analisar a taxa de variação do erro. Esse modelo proporciona uma resposta adaptativa, melhorando a precisão e estabilidade do sistema ao longo do tempo.

Conforme descrito por Ogata et al. (2009), o controle PID é particularmente eficaz em sistemas onde é necessário corrigir desvios, mantendo a saída do sistema próxima ao valor de referência, mesmo em condições de distúrbios. A simplicidade de implementação e a robustez do controle PID tornam-no adequado para uma ampla gama de aplicações, desde processos industriais até sistemas embarcados, onde o gerenciamento de variáveis físicas como velocidade, temperatura e posição é crítico.

Para aprimorar a eficiência do controle em malha fechada, técnicas de ajuste de parâmetros, como o método de Ziegler-Nichols, são frequentemente empregadas. Esses métodos permitem determinar os ganhos para os componentes proporcional, integral e derivativo, otimizando a resposta do sistema em termos de tempo de resposta, superação e estabilidade. Além disso, avanços modernos, como o uso de algoritmos adaptativos e controle preditivo, incorporam-se para lidar com sistemas mais complexos e sujeitos a variações dinâmicas, ampliando as capacidades do controle PID tradicional.

2.5 TRABALHOS RELACIONADOS

O trabalho proposto por Beningo (2024) realiza uma comparação entre RTOS utilizando *ThreadX Metric Benchmark*, um conjunto de testes aberto para análise do desempenho de sistemas de tempo real (ECLIPSE FOUNDATION, 2024). Tal abordagem, vantagens de simplicidade, porém exige uma adaptação do sistema em questão. Além disso, segundo (BENEDEN, 2002), não é possivel mensurar as caracteristicas de um sistema de tempo real com precisão sem o uso de dispositivos externos.

Já os trabalhos apresentados por Barbalace et al. (2008), Aroca et al. (2009) desenvolve uma comparação utilizando a cópia de um sinal com um gerador de frequência e osciloscópio. Tal método permite o isolamento das propriedades do sistema de tempo real sem adicionar carga ao sistema em testes.

Quanto ao uso de recursos, os trabalhos Raza et al. (2016), Shafagh et al. (2015) analisam o uso de memória do sistema usando o programa *arm-none-eabi-size*, identificando o uso de memória exigido para o código e variáveis. Assim, este trabalho aproxima-se mais da abordagem de análise temporal apresentada por Aroca et al. (2009), enfatizando a medição e avaliação de latência e jitter em sistemas de tempo real, porém diferenciando-se por se concentrar em aplicações embarcadas. Simultaneamente, reflete o foco no uso de recursos explorado por Raza et al. (2016), especialmente no que concerne ao tamanho do programa, consumo de memória da pilha no código compilado. Adicionalmente, será adicionada uma métrica relativa ao número de símbolos presente no programa.

3 MATERIAIS E MÉTODOS

Este capítulo apresenta os aspectos fundamentais da execução deste projeto. Inicialmente, serão descritos os sistemas operacionais de tempo real (RTOS) selecionados, seguidos pela configuração experimental, detalhamento do hardware e software utilizados, e os casos de teste aplicados. O objetivo é comparar o desempenho dos RTOS em ambientes embarcados, considerando métricas temporais como tempo de resposta, e utilização de recursos, como utilização de memória.

3.1 RTOS SELECIONADOS PARA COMPARAÇÃO

Neste trabalho, foram selecionados dois sistemas operacionais de tempo real (RTOS) de código aberto para análise comparativa: o *FreeRTOS*, amplamente utilizado em aplicações de internet das coisas (MUSADDIQ et al., 2018) e implementado em C, e o *Embassy*, um RTOS emergente desenvolvido em Rust. A escolha desses sistemas baseou-se em critérios como suporte a tarefas periódicas e aperiódicas, mecanismos de sincronização entre tarefas, e a disponibilidade de código aberto, facilitando tanto o uso quanto a personalização.

3.1.1 FreeRTOS (C)

O *FreeRTOS* (AWS, 2024), na versão $11.1.0$, é um sistema operacional de tempo real desenvolvido na linguagem C amplamente adotado em sistemas embarcados. Oferecido sob a licença de código aberto MIT, torna-se uma escolha atrativa para uma ampla variedade de projetos, desde dispositivos simples até sistemas industriais complexos (BARRY, 2016).

Esse foi selecionado para este estudo devido à sua extensa documentação, comunidade ativa e suporte a múltiplos processadores e arquiteturas. Além disso, permite integração com diversas plataformas de hardware, proporcionando um ambiente favorável para o desenvolvimento de aplicações de tempo real (BARRY, 2016).

3.1.1.1 Características Relevantes

- **Suporte a tarefas periódicas e aperiódicas**: O FreeRTOS permite a criação de tarefas periódicas, controladas por temporizadores, e tarefas aperiódicas, disparadas por eventos externos ou condições internas do sistema.
- **Sincronização entre tarefas**: O sistema oferece diversos mecanismos de sincronização, como semáforos, mutexes e filas.
- **Preempção e prioridade de tarefas**: O FreeRTOS utiliza uma política de escalonamento preemptivo com prioridade fixa e compartilhamento de tempo entre

tarefas de mesma prioridade. Isso significa que o escalonador sempre executa a tarefa de maior prioridade que está pronta para ser executada. Se uma interrupção (ISR) modificar o estado de uma tarefa de maior prioridade, o escalonador preemptivamente interrompe a tarefa de menor prioridade, mesmo que esta esteja em execução, e inicia a tarefa de maior prioridade. Para tarefas de mesma prioridade, o escalonador utiliza o método *round-robin*, no qual as tarefas se alternam no estado de execução com base em fatias de tempo definidas pelo *tick* do sistema.

- **Popularidade e suporte**: A ampla adoção do FreeRTOS resulta em uma vasta gama de exemplos e integrações prontas com diversos hardwares e ferramentas de desenvolvimento, facilitando o trabalho de engenheiros e desenvolvedores.

3.1.2 Embassy (Rust)

Embassy (EMBASSY PROJECT CONTRIBUTORS, 2024) RTOS na versão $0.1.0$ é um projeto de código aberto licenciada sob a MIT, que visa simplificar o desenvolvimento de sistemas embarcados utilizando a linguagem de programação Rust (EMBASSY, 2024). Um dos diferenciais do *Embassy* é o suporte nativo a programação assíncrona, permitindo a implementação de sistemas que gerenciam múltiplas tarefas concorrentes de forma eficiente e segura. O design modular é composto por diversos *crates* (módulos), como o executor, HALs (camadas de abstração de hardware), pilha de rede, suporte a Bluetooth, USB, e outras funcionalidades específicas para sistemas embarcados (EMBASSY, 2024).

Rust é considerada uma linguagem promissora para o desenvolvimento de sistemas embarcados, devido à sua ênfase em segurança e desempenho (QIN et al., 2024). O Embassy foi selecionado por sua capacidade de gerenciar sistemas embarcados sem a sobrecarga de um RTOS tradicional, mantendo os benefícios de um modelo de programação assíncrono.

3.1.2.1 Características Relevantes

- **Suporte a tarefas periódicas e aperiódicas**: O Embassy, assim como o FreeRTOS, oferece suporte para a criação de tarefas periódicas, controladas por temporizadores ou eventos internos, além de tarefas aperiódicas, executadas em resposta a eventos externos.
- **Sincronização entre tarefas**: O Embassy implementa mecanismos de sincronização utilizando futures e await, conceitos nativos de Rust para programação assíncrona. Isso permite a execução concorrente de múltiplas tarefas sem a necessidade de preempção, resultando em maior eficiência no uso de CPU e energia.

- **Segurança de memória**: O Embassy se beneficia do sistema de tipos e do controle de memória de Rust, garantindo a ausência de problemas como condições de corrida ou estouro de buffer, aspectos críticos em sistemas embarcados.
- **Modelo assíncrono**: Utilizando um modelo assíncrono, o Embassy elimina a necessidade de um escalonador tradicional, o que permite ao sistema operar com consumo mínimo de recursos, especialmente em dispositivos de baixo consumo de energia.
- **Escalonamento cooperativo**: O modelo de escalonamento cooperativo do Embassy, baseado em futures e chamadas assíncronas, promove a execução de tarefas de maneira não preemptiva. Diferente de sistemas preemptivos tradicionais, onde o kernel pode interromper tarefas, no Embassy as tarefas devolvem o controle ao escalonador voluntariamente, geralmente ao encontrar operações assíncronas como leitura de sensores ou comunicação serial. Isso reduz a sobrecarga de troca de contexto e aumenta a previsibilidade.
- **Pseudo-prioridades e controle de execução**: Embora o Embassy não implemente diretamente um sistema de escalonamento por prioridade, ele permite o controle da ordem de execução de tarefas por meio de notificações e filas de eventos, criando um mecanismo de pseudo-prioridades. Adicionalmente, é possível definir um comportamento hierárquico entre tarefas utilizando múltiplos executores com diferentes níveis de prioridade.

A análise dos sistemas FreeRTOS e Embassy revelou abordagens distintas na maneira como gerenciam a execução de tarefas, a sincronização e a segurança de memória. O FreeRTOS, amplamente utilizado, adota um modelo tradicional preemptivo baseado em prioridades, enquanto o Embassy, escrito em Rust, propõe um modelo assíncrono, focado na segurança e eficiência de recursos. Essas diferenças estruturais destacam os pontos fortes de cada RTOS em diferentes contextos de aplicação. A seguir, uma comparação qualitativa explora essas distinções em maior profundidade, analisando como cada sistema lida com aspectos críticos do desenvolvimento embarcado, como escalonamento, sincronização, segurança e consumo de recursos.

3.1.3 Comparação Qualitativa

A comparação entre o FreeRTOS e o Embassy oferece uma perspectiva sobre as abordagens tradicionais e modernas para sistemas embarcados:

- **Modelo de escalonamento**: o FreeRTOS utiliza escalonamento preemptivo baseado em prioridades, enquanto o Embassy adota um modelo assíncrono, gerenciando a execução de tarefas por meio de futures e await, sem preempção.

- **Sincronização de tarefas**: ambos os sistemas fornecem mecanismos robustos de sincronização, embora o Embassy use abstrações modernas e seguras fornecidas por Rust.
- **Segurança de memória**: O Embassy se destaca pela segurança de memória garantida por Rust, minimizando o risco de falhas comuns em sistemas embarcados, como vazamentos de memória e condições de corrida.
- **Consumo de recursos**: O modelo assíncrono do Embassy tende a ser mais eficiente em termos de consumo de recursos em aplicações de baixo consumo de energia, enquanto o FreeRTOS, por permitir um modelo preemptivo, pode oferecer vantagens em sistemas que exigem respostas imediatas e preempção de tarefas.

Visando avaliar objetivamente as diferenças entre os dois sistemas, foi definido um conjunto de métricas que considera tanto as características estáticas quanto as dinâmicas de ambos os RTOS. Tais métricas foram escolhidas para fornecer uma análise abrangente, considerando aspectos de desempenho, uso de recursos e eficiência no gerenciamento de tarefas, permitindo uma comparação entre os sistemas.

3.2 MÉTRICAS

As métricas utilizadas foram especificadas na Seção 2.3. Nesse tópico, é detalhado como tais indicadores foram quantificados nos sistemas implementados em C com FreeRTOS e em Rust com Embassy. Para esse fim, foram utilizadas ferramentas específicas para cada linguagem e RTOS, visando uma avaliação precisa e comparável entre os sistemas.

3.2.1 Latência e do Jitter

As medições de latência e jitter foram realizadas seguindo os métodos descritos por (AROCA et al., 2009), utilizando estímulos externos e instrumentos de medição.

- Foi utilizado um gerador de sinais para produzir eventos externos (interrupções) nos sistemas em teste.
- Um osciloscópio foi empregado para medir o tempo entre o sinal de entrada (evento externo) e a resposta do sistema (ação do RTOS), capturando assim a latência.
- Foram coletadas 60 amostras independentes.

Os dados coletados foram organizados e analisados estatisticamente. A latência média, mínima, máxima, o desvio padrão e o jitter foram calculados para cada sistema e cenário, conforme as definições apresentadas anteriormente.

3.2.2 Tamanho do Programa

O tamanho do programa foi obtido a partir dos binários compilados, focando na seção `.text`, que contém o código executável.

- **C**: Utilizou-se o comando `arm-none-eabi-size`, que reporta os tamanhos das seções `.text`, `.data` e `.bss`, além do tamanho total em formato decimal e hexadecimal. A seção `.text` armazena o código compilado gerado.
- **Rust**: Utilizou-se o comando `cargo size`, que fornece informações similares, reportando os tamanhos das seções `.text`, `.data` e `.bss` do binário compilado. O foco principal foi a análise da seção `.text` para comparar a eficiência do código executável gerado.

3.2.3 Uso de Memória da Pilha

Para medir o uso de memória da pilha, foram analisadas as seções `.data` e `.bss`, que representam o armazenamento de variáveis globais e estáticas.

- **C**: O comando `arm-none-eabi-size` foi utilizado novamente para obter os tamanhos das seções `.data` e `.bss`. A soma desses valores fornece uma estimativa do uso total de memória relacionado às variáveis globais.
- **Rust**: De forma análoga, o comando `cargo size` forneceu os tamanhos das seções `.data` e `.bss` no binário compilado em Rust.

Essa análise permitiu avaliar o uso de memória da pilha em cada sistema.

3.2.4 Quantidade de Símbolos

A contagem de símbolos no código compilado foi realizada para analisar a complexidade e possíveis sobrecargas introduzidas por cada RTOS.

- **C**: Utilizou-se o comando `arm-none-eabi-nm` para listar todos os símbolos presentes no binário compilado. A quantidade total foi obtida contando-se o número de linhas retornadas pelo comando.
- **Rust**: O comando `cargo nm` foi empregado para listar os símbolos no binário compilado em Rust, permitindo uma comparação direta com o sistema em C.

3.3 REQUISITOS DO SISTEMA

Para o desenvolvimento e avaliação dos RTOS neste trabalho, foram definidos os seguintes requisitos, baseados em funções comuns de sistemas embarcados (MARWEDEL, 2021) e suas respectivas implementações em um sistema de tempo real.

- **Leitura periódica de sensores**: A leitura de sensores é necessária para a coleta de dados do ambiente. Essa função será implementada como uma **tarefa periódica**, com prioridade alta, uma vez que a coleta de dados precisa ocorrer em intervalos regulares para garantir a precisão e a previsibilidade do sistema. A implementação desta tarefa no RTOS deve assegurar que a leitura dos sensores ocorra sem perturbações, respeitando os prazos definidos, uma vez que atrasos podem comprometer o desempenho geral do sistema.
- **Acionamento de atuadores**: A resposta a eventos detectados pelos sensores se dá através do acionamento de atuadores, como motores, válvulas ou displays. Esta função será implementada como uma **tarefa periódica de menor prioridade**. Apesar de ser periódica, a tarefa de acionamento de atuadores deve ocorrer após o processamento das leituras de sensores. Portanto, o RTOS deve garantir que, embora com menor prioridade, essa tarefa seja executada, respeitando os prazos estabelecidos para o funcionamento dos dispositivos físicos controlados.
- **Processamento dos dados**: O processamento das informações coletadas pelos sensores é essencial para determinar as ações a serem tomadas. Este requisito será implementado como parte das tarefas de **computação** no sistema, responsável por calcular, filtrar e analisar os dados. O RTOS deve conseguir gerenciar a execução eficiente desse processamento, evitando interferências nas tarefas de alta prioridade, como as leituras periódicas.
- **Processamento de eventos**: Além das tarefas periódicas, sistemas que interagem com fenômenos físicos frequentemente precisam reagir a eventos inesperados, como mudanças bruscas no ambiente ou comandos externos. Esse tipo de função será implementado como uma **tarefa aperiódica**. A execução dessas tarefas deve ser tratada de maneira eficiente pelo RTOS, sendo acionada apenas quando um evento relevante ocorre, sem afetar a previsibilidade das tarefas periódicas.

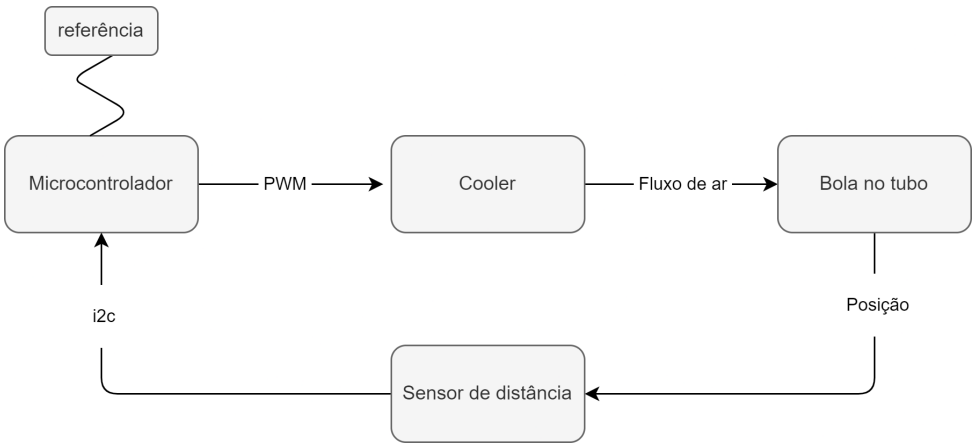
3.4 MODELAGEM DO SISTEMA

A fim de atender tais requerimentos, o sistema escolhido para este estudo foi o caso de um controle de altura de uma bola em um tubo. Tal planta visa ajustar a velocidade de um ventilador que sopra ar em um tubo, de modo a manter uma pequena bola suspensa em um nível determinado no interior do tubo (PEREIRA; BOWLES, 1996).

A posição da bola é determinada por um sensor de distância, que mede a altura no tubo e envia esses dados ao microcontrolador por meio da interface *Inter-Integrated Circuit* (I2C). O microcontrolador processa essa informação e compara com uma referência de posição pré-determinada. Com base nessa comparação, o micro-

controlador ajusta a velocidade do ventilador via um sinal de *Pulse-Width Modulation* (PWM). O ventilador sopra ar para dentro do tubo, criando um fluxo que mantém a bola suspensa na altura desejada; tal funcionamento é ilustrado na Figura 2.

Figura 2 – Diagrama de blocos do sistema de testes



Fonte: Elaborado pelo autor (2024)

Esse modelo simplificado ajuda a identificar os componentes necessários e suas relações. Nas próximas seções, serão discutidos os subsistemas de hardware e software responsáveis por tais funções.

3.5 CONSTRUÇÃO DA PLANTA

Para delimitar a região de operação, optou-se por um tubo de acrílico com diâmetro externo de 50 mm, diâmetro interno de 44 mm e altura de 950 mm. O acrílico foi escolhido devido à sua transparência, permitindo a observação direta dos fenômenos internos, e à sua resistência mecânica adequada para aplicações estruturais. Além disso, o material apresenta resistência a impactos e é relativamente leve, facilitando o manuseio e a instalação.

O sensor de tempo de voo VL53L0X (STMICROELETRONICS, 2024) foi utilizado para medir as distâncias no sistema. A tecnologia de tempo de voo permite medições independentes das condições de luz ambiente, tornando-o adequado para aplicações internas e externas. A escolha deste sensor foi motivada pelo baixo custo e pela facilidade de integração com microcontroladores, graças à interface de comunicação I2C. A disponibilidade imediata e a compatibilidade com a infraestrutura existente também influenciaram na decisão.

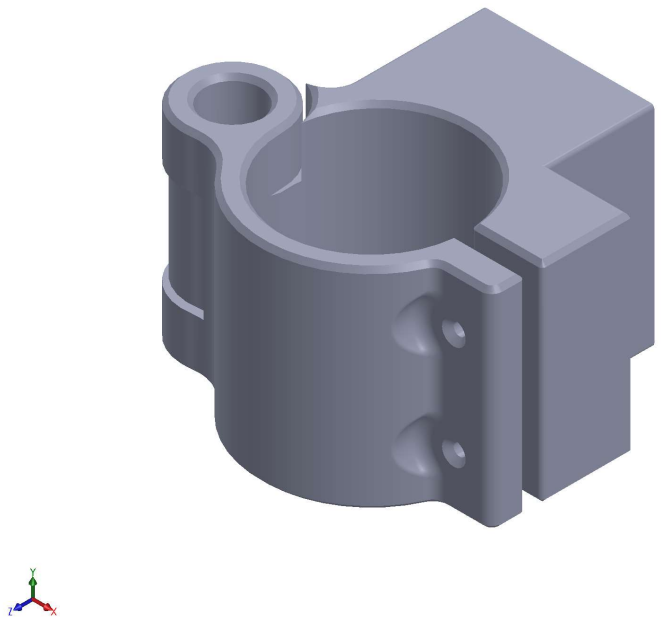
O sistema de ventilação conta com um ventilador AUB0912VH-CX09 (DELTA ELECTRONICS, 2024). Este componente é responsável por gerar o fluxo de ar no tubo de acrílico, permitindo o controle das condições experimentais. A escolha deste modelo se deu pela compatibilidade com os requisitos do projeto, em especial, por

incluir um controle de velocidade por PWM integrado, removendo a necessidade de um circuito adicional.

A fim de sustentar o conjunto, um suporte metálico construído em perfil de alumínio foi desenvolvido. O alumínio foi escolhido por sua leveza, resistência e facilidade de manipulação. Este suporte é essencial para manter o alinhamento correto do sistema e permitir ajustes finos durante os experimentos. A fabricação personalizada do suporte permitiu a adequação às dimensões do tubo e demais componentes.

Para fixar o tubo de acrílico ao suporte metálico, foram desenvolvidas peças sob medida por meio de modelagem e impressão 3D, o que otimizou o encaixe e minimizou vibrações ou desalinhamentos. Essa solução personalizada permitiu prototipagem e ajustes, estando o resultado representado na Figura 3.

Figura 3 – Modelo de suporte para o tubo de acrílico



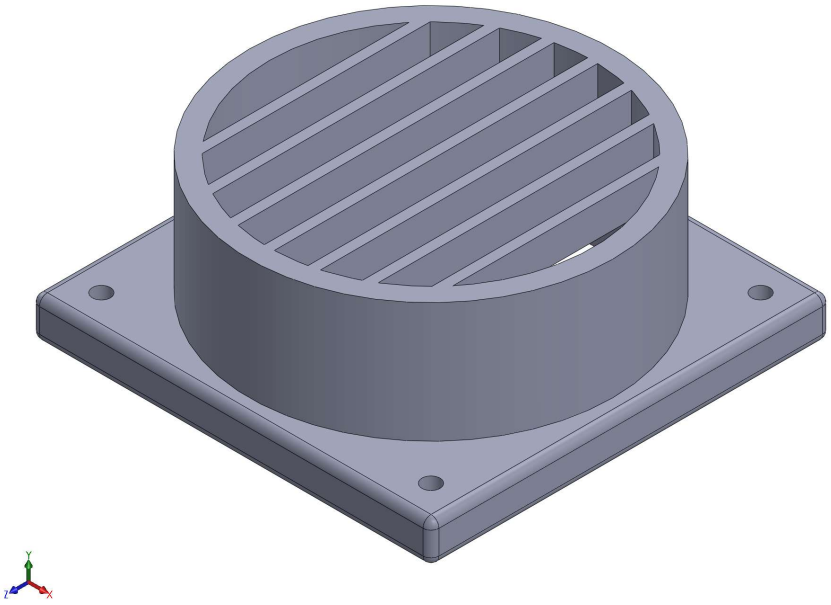
Fonte: Elaborado pelo autor (2024)

Da mesma forma, a fixação do ventilador foi projetada utilizando modelagem 3D, criando um suporte específico que auxilia no posicionamento deste componente. A Figura 4 apresenta o modelo do suporte para o ventilador.

Além disso, para o sensor, foi criado um suporte específico por meio de modelagem 3D, garantindo o posicionamento adequado do componente. O modelo deste suporte é ilustrado na Figura 5.

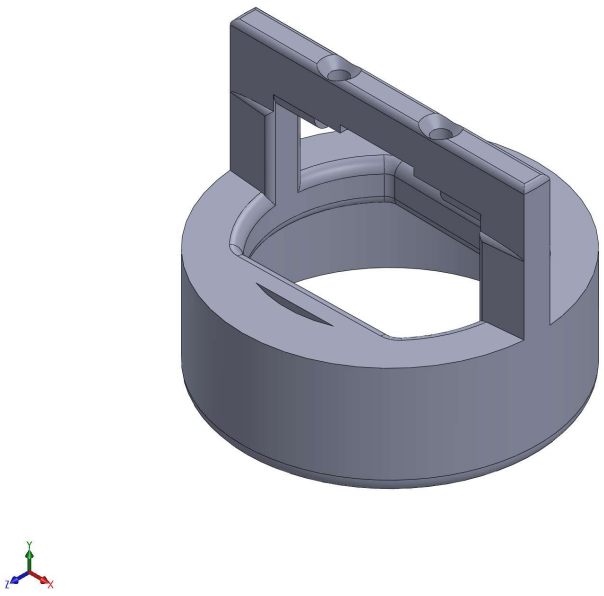
Uma bola de tênis de mesa, com diâmetro de 40 mm, foi selecionada como objeto de teste no tubo de acrílico. Este item apresenta características desejáveis para os experimentos propostos, como baixa massa e superfície lisa, permitindo que

Figura 4 – Modelo de suporte para ventilador



Fonte: Elaborado pelo autor (2024)

Figura 5 – Modelo de suporte para o sensor



Fonte: Elaborado pelo autor (2024)

responda de maneira previsível ao fluxo de ar. Ademais, a escolha desse objeto padronizado facilita a reprodução dos experimentos por outros pesquisadores, contribuindo para a validade científica dos resultados obtidos.

O microcontrolador escolhido foi o STM32F103C8T6 da STMicroelectronics (2024). Este microcontrolador, da família STM32, foi selecionado por sua compatibilidade com os requisitos de hardware do projeto, oferecendo uma gama de funcionalidades que atendem diretamente às necessidades do sistema. Um dos principais motivos para sua escolha foi o suporte nativo à comunicação I2C e PWM, essenciais para a leitura do sensor de distância VL53L0XV2 e o controle da velocidade do ventilador, respectivamente.

Além disso, o STM32F103C8T6 conta com um processador ARM Cortex-M3 de 32 bits, que oferece um baixo consumo de energia, uma característica importante para sistemas embarcados que requerem eficiência. Com 64 kB de memória Flash e 20 kB de RAM, este microcontrolador possui recursos suficientes para o armazenamento e execução das tarefas do sistema, incluindo as rotinas de controle de sensores e atuadores, bem como o processamento de dados em tempo real.

Outro fator determinante para a escolha desse microcontrolador foi seu suporte aos dois RTOS utilizados no projeto: FreeRTOS e Embassy (Rust). Ambos os sistemas operacionais de tempo real possuem bibliotecas e exemplos de uso documentados para a plataforma STM32, facilitando a implementação e integração do sistema de controle. A disponibilidade de documentação, comunidade ativa e ferramentas de desenvolvimento compatíveis, como o *STM32CubeIDE* e o compilador *rustc*, também foi considerada na decisão, garantindo maior flexibilidade e suporte ao longo do desenvolvimento do projeto. Essa placa de desenvolvimento pode ser visualizada na Figura 6.

O Quadro 1 apresenta as conexões utilizadas no sistema, detalhando os pinos do microcontrolador e as respectivas funções.

Quadro 1 – Conexões dos Pinos do Microcontrolador STM32F103C8T6

Pino	**Função**	**Descrição**
PA8	PWM	Controla o ventilador para ajustar o fluxo de ar.
PB6	I2C SCL	Linha de clock de comunicação com o sensor.
PB7	I2C SDA	Linha de dados com o sensor.
PA0	Entrada Digital	Interrupção para alterar a referência de controle.
PB1	Saída Digital	Cópia do sinal no pino A0

Fonte: Elaborado pelo Author (2024)

A entrada digital visa alterar a referência do sistema de controle, permitindo a incorporação de uma perturbação assíncrona que modifica o estado do sistema. Essa alteração de referência emula uma mudança inesperada no ambiente, permitindo testar a resposta do sistema a eventos externos. Já a saída digital, conforme

especificado no Quadro 1, atua como uma replicação do estado da entrada, facilitando o monitoramento desse comportamento. Essa funcionalidade será utilizada na coleta de métricas do sistema, permitindo a análise da resposta temporal e desempenho diante de alterações na referência. A construção completa da planta se encontra na Figura 7.

3.6 DESCRIÇÃO DAS TAREFAS

Dado todos os componentes especificados, as tarefas que os utilizam serão descritas. De modo geral, a tarefa de leitura do sensor requisita periodicamente a distância atual da bola no tubo, transmitindo essa informação para o controlador. Com base na nova distância lida e na referência atualizada, o controlador calcula a saída adequada para ajustar o fluxo de ar, para manter a bola na altura desejada. Ademais, em qualquer momento durante o processo, o usuário pode pressionar um botão, gerando uma interrupção que altera o valor de referência do sistema; tal comportamento é ilustrado no diagrama de sequência da Figura 8.

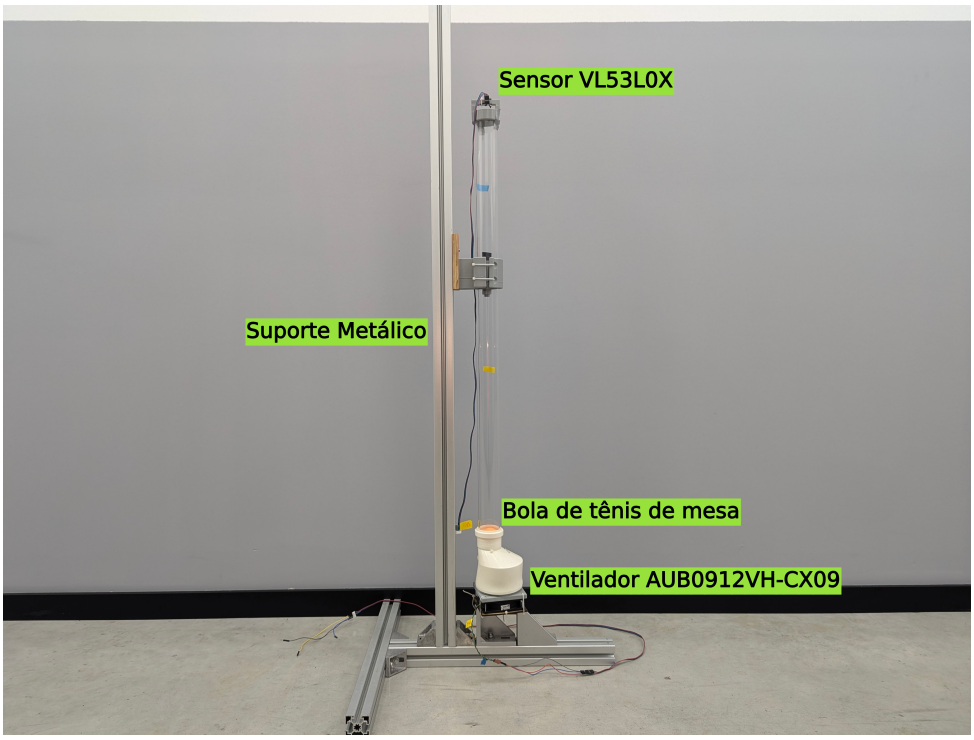
3.6.1 Leitura Periódica de Sensores

A tarefa responsável pela leitura do sensor de distância é implementada como uma tarefa periódica de alta prioridade. Esta tarefa garante que as medições ocorram em intervalos regulares, conforme definido por um período estabelecido. A precisão

Figura 6 – Placa de desenvolvimento STM32F103C8T6

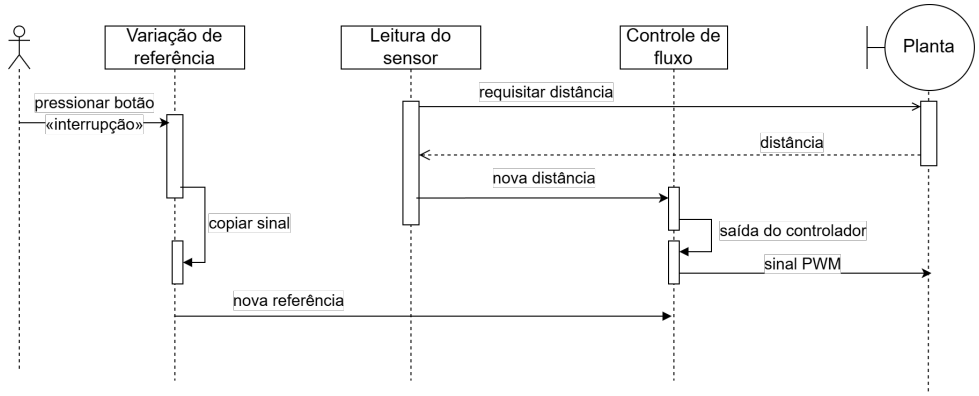
Fonte: STM32-base (2024)

Figura 7 – Planta do sistema bola no tubo



Fonte: Elaborado pelo autor (2024)

Figura 8 – Diagrama de sequência da interação entre tarefas



Fonte: Elaborado pelo autor (2024)

temporal é essencial para a confiabilidade dos dados coletados, pois atrasos ou leituras irregulares podem afetar o desempenho do controle do sistema.

A implementação utiliza uma interface de comunicação para interagir com o sensor, iniciando medições contínuas e lendo os valores de distância. Os dados coletados são transmitidos por meio de um canal assíncrono, permitindo que outras tarefas acessem as informações atualizadas sem bloqueios ou interferências.

3.6.2 Controle de fluxo

A tarefa que controla o atuador (ventilador) é implementada como uma tarefa periódica de menor prioridade. Ela ajusta o sinal de controle enviado ao atuador com base nos dados recebidos do sensor e no algoritmo de controle implementado, um controlador PID.

O sistema de controle ajusta a velocidade do atuador conforme a saída do controlador, permitindo manipular a posição da esfera no tubo e responder às variações detectadas pelo sensor. A tarefa considera a possibilidade de atualização do ponto de referência por um mecanismo de sinalização ou comunicação entre tarefas, permitindo a mudança dinâmica da posição desejada da esfera.

3.6.3 Variação de referência

A tarefa responsável pelo processamento de eventos inesperados é implementada como uma tarefa aperiódica que responde a interrupções ou sinais externos. Essa monitora entradas configuradas para detecção de mudanças de estado, permitindo que o sistema reaja a eventos externos, como um botão acionado pelo usuário.

Ao detectar uma mudança de estado, a tarefa pode alterar o ponto de referência do controlador entre valores predefinidos ou executar outras ações necessárias, demonstrando a capacidade do sistema de responder a comandos externos. Esta implementação objetiva que eventos assíncronos não comprometam a previsibilidade das tarefas periódicas, mantendo a integridade e o desempenho do sistema.

Assim, um resumo dos parâmetros de cada tarefa está presente no Quadro 2. Observa-se que o tempo de execução utilizado foi estimado com base nas operações presentes e especificações dos componentes.

Quadro 2 – Modelagem das Tarefas

Tarefa	**Período (ms)**	**Duração aproximada (ms)**
Leitura de Sensor	50	30
Controle de fluxo	500	1
Variação de referência	-	0,03

Fonte: Elaborado pelo Author (2024)

Na próxima seção, será abordada a implementação de tais tarefas em ambas as linguagens utilizadas, bem como as particularidades relativas a cada RTOS.

3.7 IMPLEMENTAÇÃO DO SOFTWARE

Como mencionado anteriormente, o sistema de controle foi implementado em duas abordagens distintas: uma utilizando a linguagem Rust com o RTOS Embassy, e

outra utilizando C com o FreeRTOS. Ambas visam controlar a posição de uma esfera em um tubo, ajustando a velocidade de um ventilador com base nas medições de um sensor de distância. A seguir, detalham-se os principais aspectos de cada implementação e destacam-se as diferenças entre elas.

3.7.1 Leitura do Sensor ToF

A implementação em Rust utiliza o *Embassy*. O código é estruturado em tarefas assíncronas que interagem por meio de canais e sinais, permitindo um gerenciamento das operações de entrada e saída, bem como da concorrência.

A tarefa *read_tof_sensor* realiza a leitura periódica do sensor de tempo de voo *VL53L0X*. Utiliza a interface I2C para comunicação e envia as medições para outras tarefas via um canal assíncrono, conforme mostrado no trecho de Código 3.1.

Código 3.1 – Tarefa de leitura do sensor ToF em Rust

```
#[embassy_executor::task]
async fn read_tof_sensor(...) {
    let mut tof = VL53L0x::new(i2c).expect("tof");
    tof.start_continuous(0).expect("start");
    loop {
        let millimeters = tof.read_range_continuous_millimeters_blocking
            ()?;
        SENSOR_MILLIMETERS.send(millimeters).await;
        Timer::after_millis(MEASUREMENT_PERIOD_MS).await;
    }
}
```

Fonte: Elaborado pelo autor (2024)

Já a implementação em C utiliza o *FreeRTOS* para gerenciar as tarefas. A estrutura é semelhante à implementação em Rust, mas adapta-se às características da linguagem C e do RTOS utilizado; tal implementação é visível no trecho de Código 3.2.

Código 3.2 – Tarefa de leitura do sensor ToF em C

```
static void read_tof_sensor(void *parameters) {
    VL53L0X sensor = init_sensor();
    while (1) {
        current_sensor_reading = VL53L0X_readRangeSingleMillimeters(&
            sensor);
        notify_pwm_task();
        vTaskDelay(MEASUREMENT_PERIOD_MS / portTICK_PERIOD_MS);
    }
}
```

Fonte: Elaborado pelo autor (2024)

3.7.1.1 Controle PWM do Ventilador

A tarefa *set_pwm* ajusta a razão cíclica do PWM para controlar a velocidade do ventilador. Recebe as medições do sensor, processa-as por meio de um controlador PID e atualiza o sinal de controle. Um sinal é responsável por realizar a transmissão de dados entre tarefas, garantindo o sincronismo, conforme apresentado no trecho de Código 3.3.

Código 3.3 – Tarefa de controle PWM do ventilador em Rust

```
#[embassy_executor::task]
async fn set_pwm(...) {
let mut pwm = SimplePwm::new(...);
    let max = pwm.get_max_duty() as f32;
    pwm.enable(Channel::Ch1);
    let min = max * 0.61;
    pwm.set_duty(Channel::Ch1, min as u16);
    Timer::after_millis(5000).await;
    let mut controller = create_pid_controller();
    let mut setpoint = 500.00;
    loop {
        match SENSOR_MILLIMETERS.receive().await {
            sensor_distance => {
                let measurement = sensor_distance as f32;
                if SETPOINT.signaled() {
                    let new_setpoint = SETPOINT.wait().await;
                    setpoint = new_setpoint;
                }
                let control_out: ControllerOutput =
                    controller.controller_output(&(measurement), &
                        setpoint);

                let new_duty_cycle =
                    (max * transpose_control_out(control_out.full_output
                        , 0.61)) as u16;
                pwm.set_duty(Channel::Ch1, new_duty_cycle);
            }
        };
    }
}
```

Fonte: Elaborado pelo autor (2024)

Na implementação em C, o comportamento é equivalente, entretanto, a sincronização é obtida por via de uma notificação entre tarefas, como representado no trecho de Código 3.4.

Código 3.4 – Tarefa de controle PWM do ventilador em C

```c
void set_pwm(void *parameters) {
    Init_Pwm();
    Change_PWM_Duty_Cycle_Percentage(0.61f);
    TickType_t xLastWakeTime = xTaskGetTickCount();
    xTaskDelayUntil(&xLastWakeTime, 5000);
    PID controller = create_pid_controller();
    const TickType_t maxBlockTime = pdMS_TO_TICKS(10 *
       MEASUREMENT_PERIOD_MS);
    uint32_t sensorReadingNotifications;
    while (1) {
        sensorReadingNotifications = ulTaskNotifyTakeIndexed(0, pdTRUE,
           portMAX_DELAY);
        if (sensorReadingNotifications == 0) {
            continue;
        }
        sensorReadingNotifications = 0;
        ControllerOutput output = PID_controller_output(&controller,
           current_sensor_reading, setpoint);
        float new_duty_cycle = transpose_control_out(output.full_output,
            0.61f);
        Change_PWM_Duty_Cycle_Percentage(new_duty_cycle);
    }
}
```

Fonte: Elaborado pelo autor (2024)

3.7.1.2 Processamento de Eventos Externos

A tarefa *interrupt_task* lida com eventos externos, como a mudança de referência via interrupção. Monitorando um pino de entrada e alterando a referência do controlador quando um evento é detectado, tal comportamento se encontra no trecho de Código 3.5.

Código 3.5 – Tarefa de processamento de eventos externos em Rust

```rust
#[embassy_executor::task]
async fn interrupt_task(...) {
    let mut input = ExtiInput::new(Input::new(input_pin, Pull::Up),
       interrupt);
    let mut output = Output::new(output_pin, Level::Low, Speed::Low);
    let mut current_setpoint = SETPOINTS[0];
    loop {
        input.wait_for_rising_edge().await;
        output.set_high();
        input.wait_for_falling_edge().await;
```

```
        output.set_low();
        current_setpoint = if current_setpoint == SETPOINTS[0] {
            SETPOINTS[1] } else { SETPOINTS[0] };
        SETPOINT.signal(current_setpoint);
    }
}
```

Fonte: Elaborado pelo autor (2024)

O trecho de Código 3.6 descreve o equivalente em C para essa tarefa.

Código 3.6 – Tarefa de processamento de eventos externos em C

```
static void interrupt_task(void *parameters) {
  volatile uint32_t notificationValue;
  while (1) {
    notificationValue = ulTaskNotifyTakeIndexed(0,pdTRUE, portMAX_DELAY)
        ;
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_SET);
    notificationValue = ulTaskNotifyTakeIndexed(1,pdTRUE, portMAX_DELAY)
        ;
    HAL_GPIO_WritePin(GPIOB, GPIO_PIN_1, GPIO_PIN_RESET);
    setpoint = SETPOITS[setpoint == SETPOITS[0]];
    notify_pwm_task();
  }
}
```

Fonte: Elaborado pelo autor (2024)

3.7.2 Considerações

As duas implementações alcançam o objetivo de controlar a posição de uma esfera utilizando um controlador PID, sensor de distância e atuador. A implementação em Rust com Embassy RTOS destaca-se pela programação assíncrona e segurança oferecida pela linguagem, enquanto a implementação em C com FreeRTOS é uma abordagem consolidada, mas requer maior cuidado no gerenciamento de recursos. Dessa forma, tomando como base uma análise qualitativa, a escolha entre as duas abordagens depende dos requisitos do projeto, experiência da equipe e considerações sobre segurança, desempenho e manutenção do código.

3.8 CONFIGURAÇÕES E OPTIMIZAÇÕES

Além da implementação das tarefas, é necessário realizar configurações nos RTOS e compiladores; essas serão tratadas a seguir.

Ambos os sistemas foram executados no microcontrolador configurado com um clock de 72 MHz, o valor máximo suportado pelo hardware utilizado. Essa configuração assegura que ambos os RTOS operem nas mesmas condições de processamento, permitindo uma comparação dos resultados.

Para o sistema implementado em C com FreeRTOS, foi utilizada a opção de otimização `-O2` do compilador. Essa opção equilibra a otimização do código gerado com o tempo de compilação e o tamanho do executável, proporcionando melhorias de desempenho sem comprometer a estabilidade.

No sistema implementado em Rust com Embassy, foi definido o nível de otimização `opt-level = 2` no arquivo `Cargo.toml`. Assim como a opção `-O2` no C, essa configuração visa otimizar o código gerado pelo compilador Rust, melhorando a eficiência do executável resultante.

3.8.1 Ambiente de Desenvolvimento

Para assegurar a reprodutibilidade dos experimentos e a confiabilidade dos resultados obtidos, detalha-se a seguir o ambiente de desenvolvimento utilizado neste estudo. A especificação das versões dos sistemas operacionais de tempo real (RTOS), compiladores e ferramentas auxiliares é fundamental para garantir a consistência entre os testes realizados em diferentes plataformas.

Os RTOS empregados foram:

- **Embassy**: versão `0.1.0`, utilizado em conjunto com a linguagem Rust para a implementação das aplicações assíncronas e concorrentes.
- **FreeRTOS**: versão `11.1.0`, selecionado por sua ampla adoção em sistemas embarcados e por oferecer uma base consolidada para comparações de desempenho.

As ferramentas de compilação e linguagens de programação utilizadas foram:

- **Compilador Rust (`rustc`)**: versão `1.81.0`, responsável pela compilação do código escrito em Rust para a plataforma de hardware alvo.
- **Compilador GCC para ARM (`arm-none-eabi-gcc`)**: versão `10.3.1`, empregado na compilação do código em C para a arquitetura ARM Cortex-M, garantindo a compatibilidade com o FreeRTOS.

O sistema operacional utilizado durante o desenvolvimento e testes foi o Ubuntu 20.04, executado no *Windows Subsystem for Linux* (WSL) versão 2. Essa configuração permitiu a integração de ferramentas típicas do ambiente Linux com a conveniência de execução em um sistema Windows.

Essas configurações padronizadas permitem que os resultados obtidos sejam atribuídos às características inerentes das implementações, RTOS e das linguagens

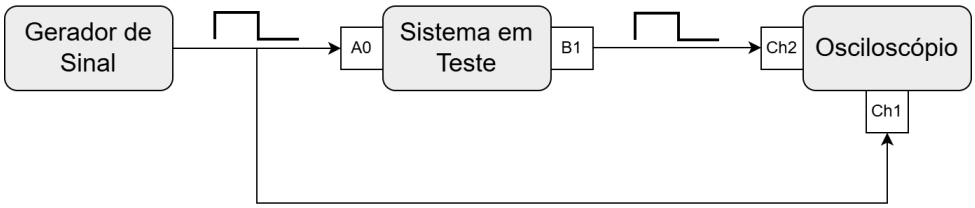
de programação utilizadas. Com a definição dos sistemas concluída, a próxima seção abordará os testes desenvolvidos a fim de comparar o desempenho desses.

3.9 CENÁRIOS DE TESTES

Para avaliar o desempenho e a eficiência do sistema desenvolvido, foram definidos dois cenários de teste que permitem uma análise de suas respostas e características temporais. Esses cenários foram escolhidos para isolar o comportamento de uma tarefa simples e, posteriormente, avaliar o sistema completo, com todas as suas tarefas e interrupções.

Em ambos os casos, um osciloscópio foi utilizado para monitorar a entrada, conectada ao primeiro canal e a saída do sistema, conectada ao segundo canal, como representado na Figura 9. Sessenta amostras foram coletadas, com uma frequência de amostragem de 1GS/s, registrando os tempos de resposta para cada pulso de entrada. Esses dados permitiram a análise estatística da latência e do jitter em cada cenário, possibilitando a comparação do desempenho entre os diferentes RTOS utilizados.

Figura 9 – Diagrama de Blocos dos Experimentos



Fonte: Elaborado pelo Author (2024)

3.9.1 Cenário 1: Tarefa de Cópia de Sinal de Entrada

O primeiro cenário de teste considera exclusivamente a tarefa de cópia de sinal de entrada, se aproximando do experimento proposto por Aroca et al. (2009). Nesse, a entrada é uma onda quadrada de frequência 1 Hz, aplicada diretamente à interrupção, a fim de simular o acionamento de um botão em intervalos constantes. A tarefa de cópia replica o estado da entrada para uma saída digital, permitindo observar o tempo de resposta do sistema sem a interferência de outras tarefas.

Esse cenário é fundamental para avaliar o comportamento do sistema em uma configuração simplificada, onde apenas uma tarefa aperiódica está em execução. Ao isolar essa tarefa, é possível medir a latência e o jitter da resposta, fornecendo uma linha de base para a análise comparativa com o sistema completo.

3.9.2 Cenário 2: Sistema Completo com Todas as Tarefas

O segundo cenário de teste avalia o sistema em sua totalidade, executando todas as tarefas previstas, incluindo a leitura periódica do sensor, o acionamento do atuador e o processamento de dados, em conjunto com a tarefa de cópia de sinal de entrada. Nesse cenário, a entrada de 1 Hz é novamente aplicada na interrupção, proporcionando uma carga mais realista e permitindo observar a interação entre as diferentes tarefas do sistema.

Este cenário permite avaliar o desempenho do sistema sob carga, analisando como a coexistência de múltiplas tarefas impacta o tempo de resposta. As métricas observadas neste caso, como latência e jitter, ajudam a identificar eventuais limitações no gerenciamento de tarefas e na priorização de eventos em tempo real de cada RTOS, fornecendo informações críticas para a análise do comportamento do sistema em aplicações de controle dinâmico.

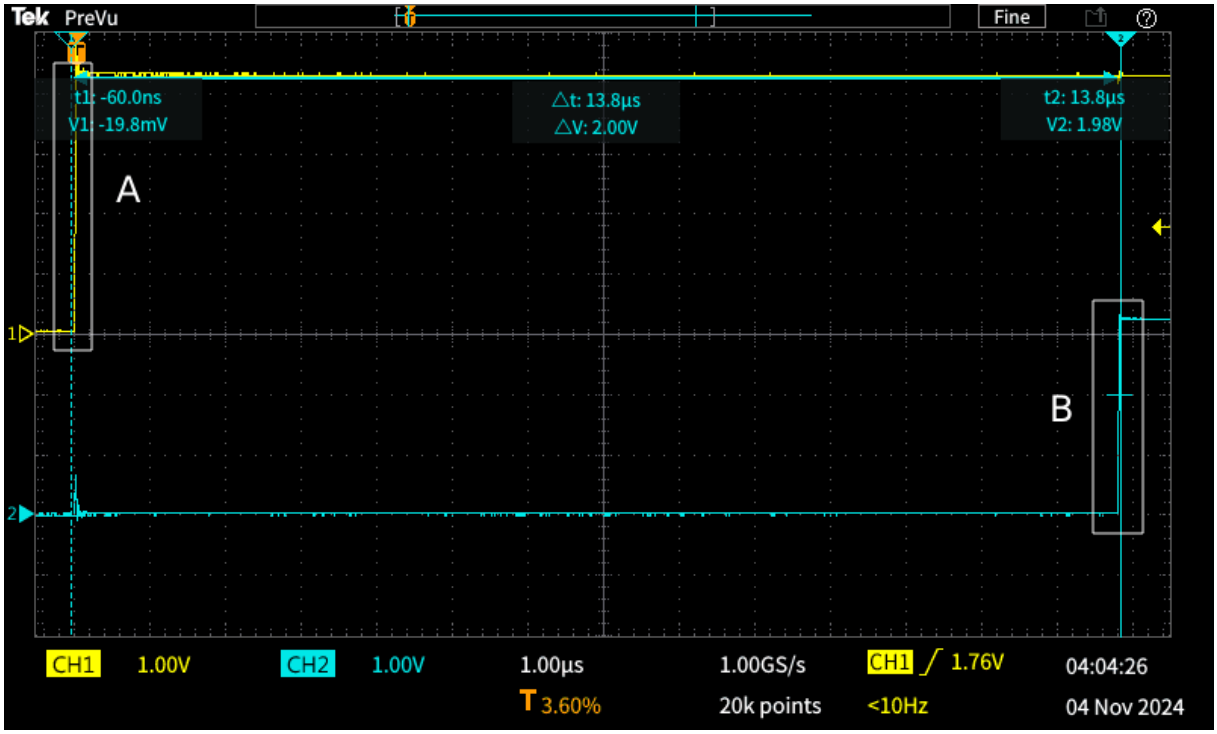
4 APRESENTAÇÃO DOS DADOS

Neste capítulo, apresentam-se os resultados dos experimentos realizados nos dois cenários previamente descritos. O objetivo é analisar o desempenho dos diferentes sistemas operacionais de tempo real (RTOS) e linguagens de programação utilizados, focando nas métricas temporais — latência e jitter — e no uso de recursos, incluindo o tamanho do programa, consumo de memória da pilha e número de símbolos no código compilado. A análise comparativa visa identificar como a coexistência de múltiplas tarefas e a escolha da linguagem impactam o comportamento do sistema em aplicações de tempo real, tanto em termos de desempenho temporal quanto na eficiência no uso dos recursos disponíveis.

4.1 CENÁRIO 1

O primeiro cenário consiste na cópia do sinal de entrada por parte do RTOS. As Figuras 10 e 11 apresentam as formas de onda capturadas para o sistema em FreeRTOS (C) durante a borda de subida e descida do sinal de entrada, respectivamente.

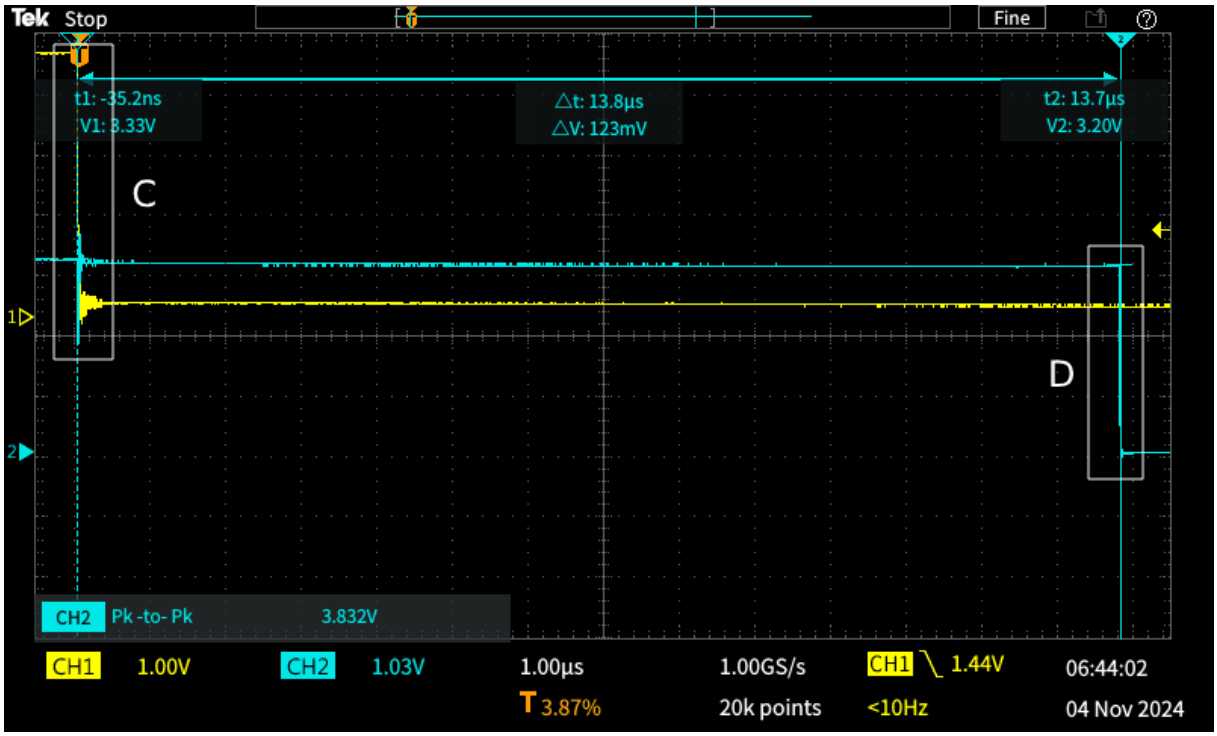
Figura 10 – Borda de subida no FreeRTOS (C) no primeiro cenário



Fonte: Elaborado pelo Author (2024)

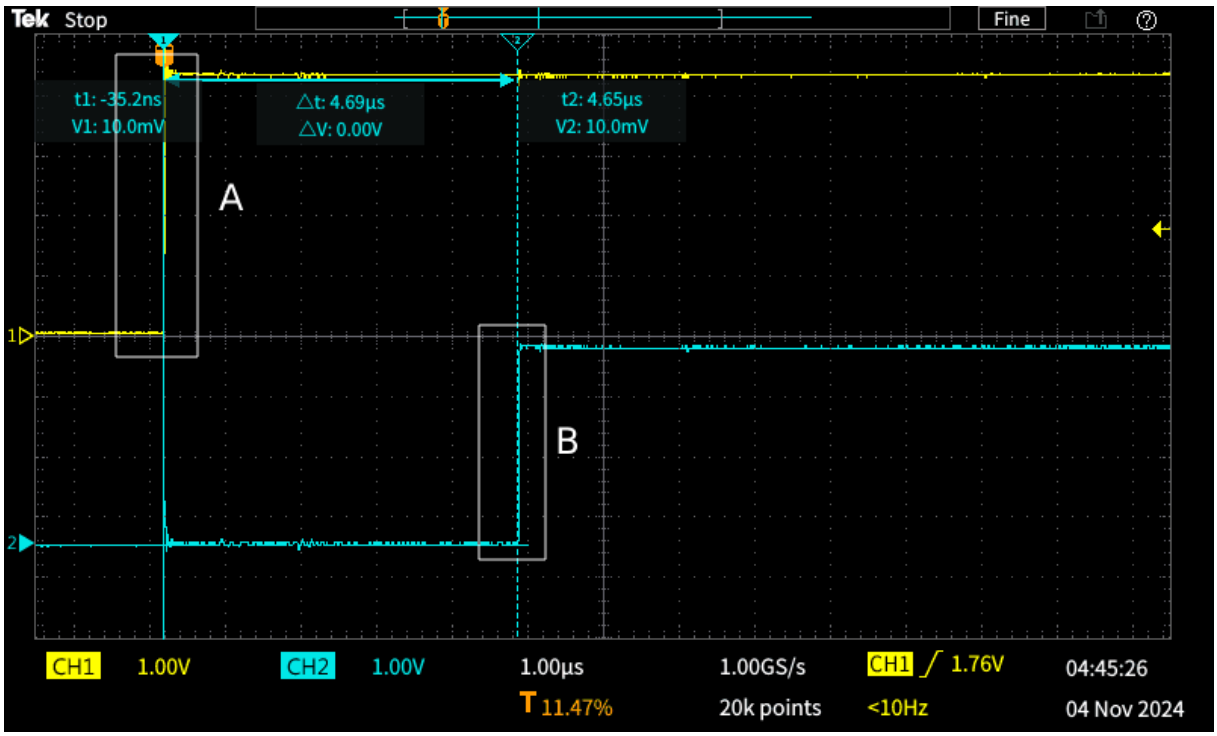
As Figuras 12 e 13 exibem as formas de onda para o sistema em Embassy (Rust) durante a borda de subida e descida, respectivamente.

Figura 11 – Borda de descida no FreeRTOS (C) no primeiro cenário



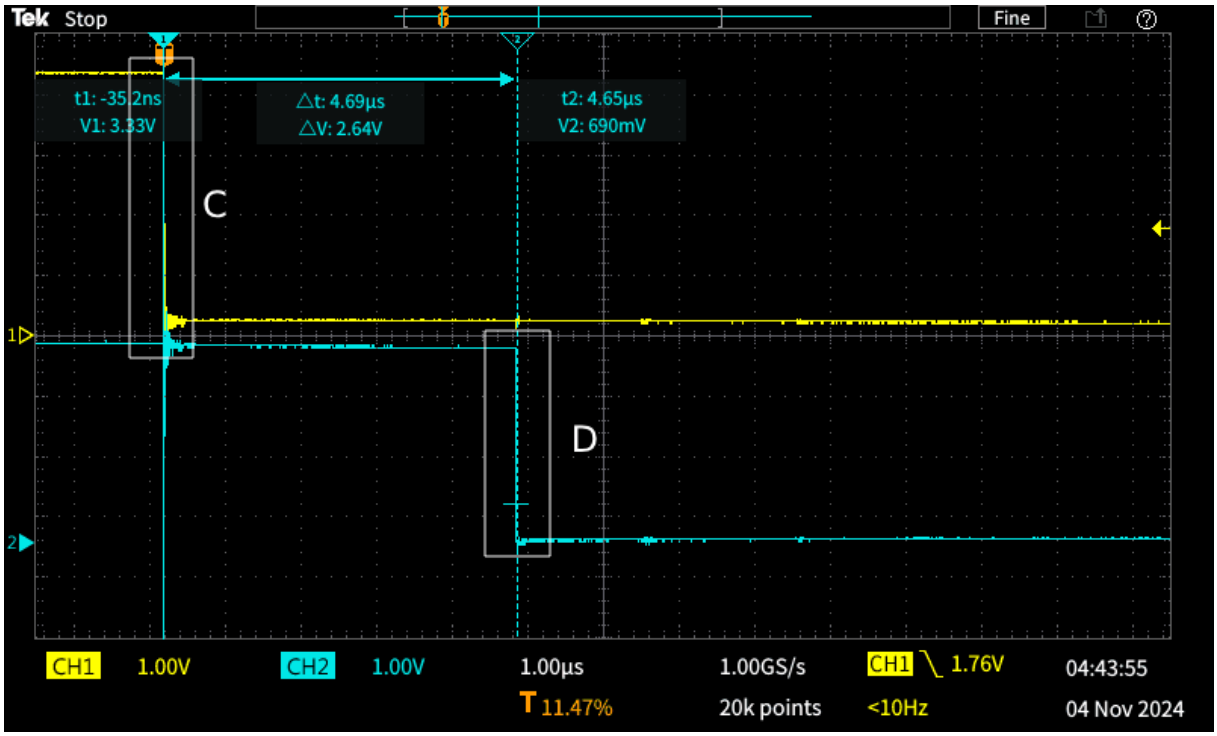
Fonte: Elaborado pelo Author (2024)

Figura 12 – Borda de subida no Embassy (Rust) no primeiro cenário



Fonte: Elaborado pelo Author (2024)

Figura 13 – Borda de descida no Embassy (Rust) no primeiro cenário



Fonte: Elaborado pelo Author (2024)

Nas Figuras apresentadas, a região identificada como "A" representa a borda de subida do sinal de entrada aplicado ao sistema (canal 1, em amarelo), enquanto a marcação "B" se refere à borda de subida do sinal de saída correspondente (canal 2, em azul). Já a área representada com "C" indica a borda de descida da referência de entrada, e "D" aponta a saída correspondente. A diferença temporal entre as bordas dos sinais de entrada e saída corresponde à latência do sistema. Observa-se que, nesses casos, o sistema implementado com Embassy (Rust) apresenta uma latência menor em comparação com o FreeRTOS (C).

Foram coletadas sessenta amostras como essa, e os resultados processados estão presentes na Tabela 2.

Tabela 2 – Métricas temporais no primeiro cenário

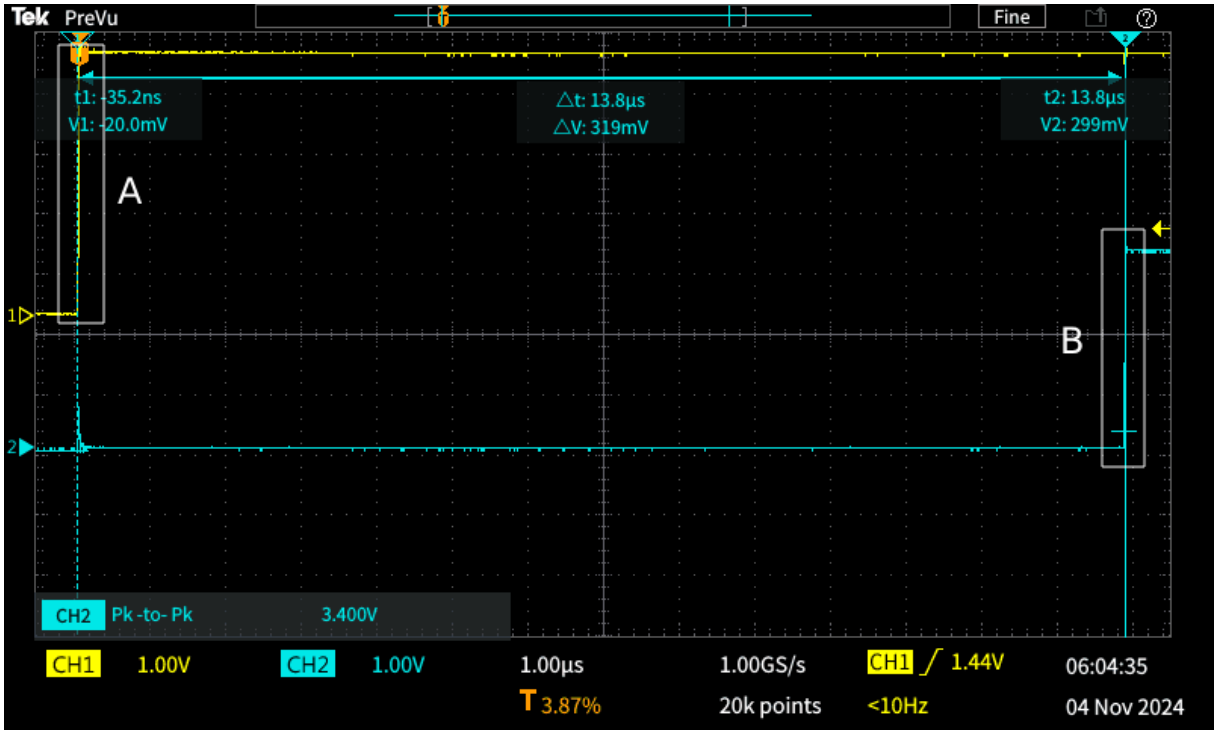
Métrica	**Embassy (Rust)**	**FreeRTOS (C)**
Latência Mínima (ns)	4638	14194
Latência Máxima (ns)	4689	14274
Latência Média (ns)	4659	14234
Desvio Padrão (ns)	20	35
Jitter (ns)	51	80

Fonte: Elaborado pelo Author (2024)

4.2 CENÁRIO 2

Para o segundo cenário, todas as tarefas foram consideradas, adicionando demanda de processamento ao sistema. Nesse caso, a tendência é que as respostas tenham uma latência mais acentuada, enquanto é atribuição do RTOS gerenciar as tarefas para evitar grandes variações. As Figuras 14 e 15 apresentam as formas de onda capturadas para o sistema em FreeRTOS (C) durante a borda de subida e descida, respectivamente.

Figura 14 – Borda de subida no FreeRTOS (C) no segundo cenário



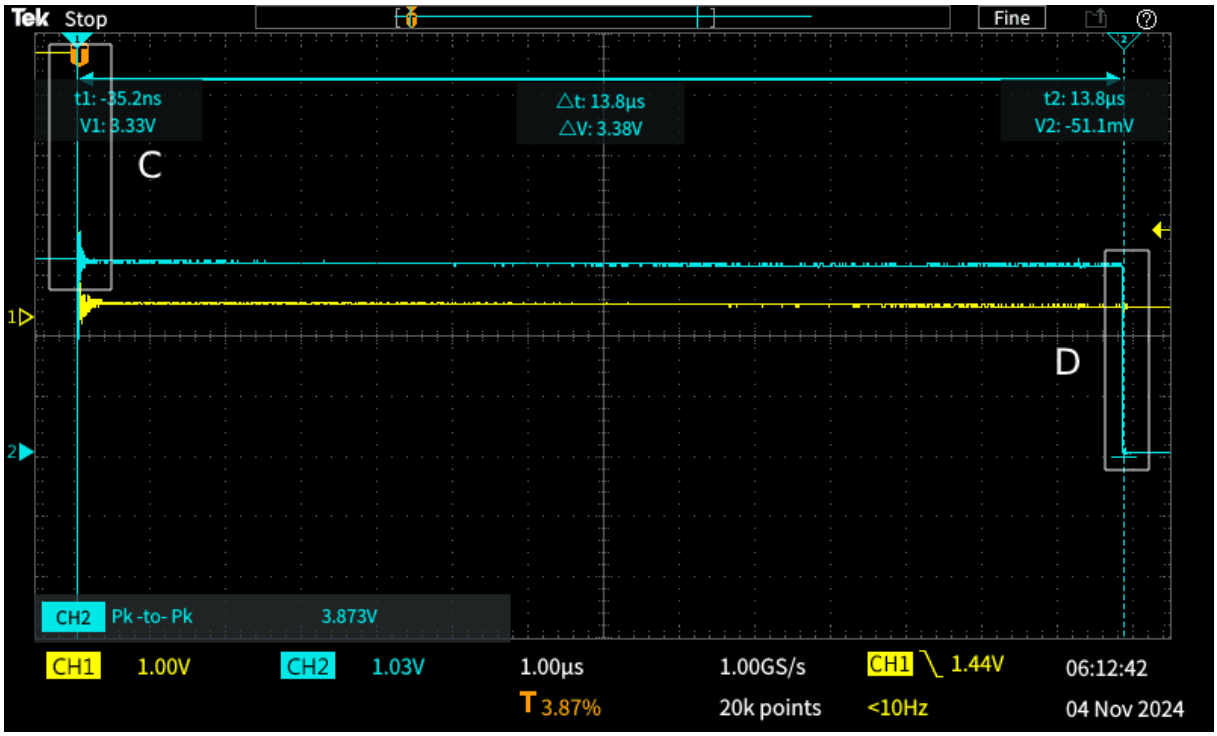
Fonte: Elaborado pelo Author (2024)

As Figuras 16 e 17 exibem as formas de onda para o sistema em Embassy (Rust) durante a borda de subida e descida, respectivamente.

Semelhante ao que foi apresentado no primeiro cenário, a região indicada por “A” representa a transição ascendente do sinal de entrada aplicado ao sistema (canal 1, em amarelo), enquanto “B” denota a transição ascendente do sinal de saída correspondente (canal 2, em azul). Analogamente, a área marcada como “C” refere-se à transição descendente da entrada de referência, e “D” aponta para a respectiva transição descendente na saída.

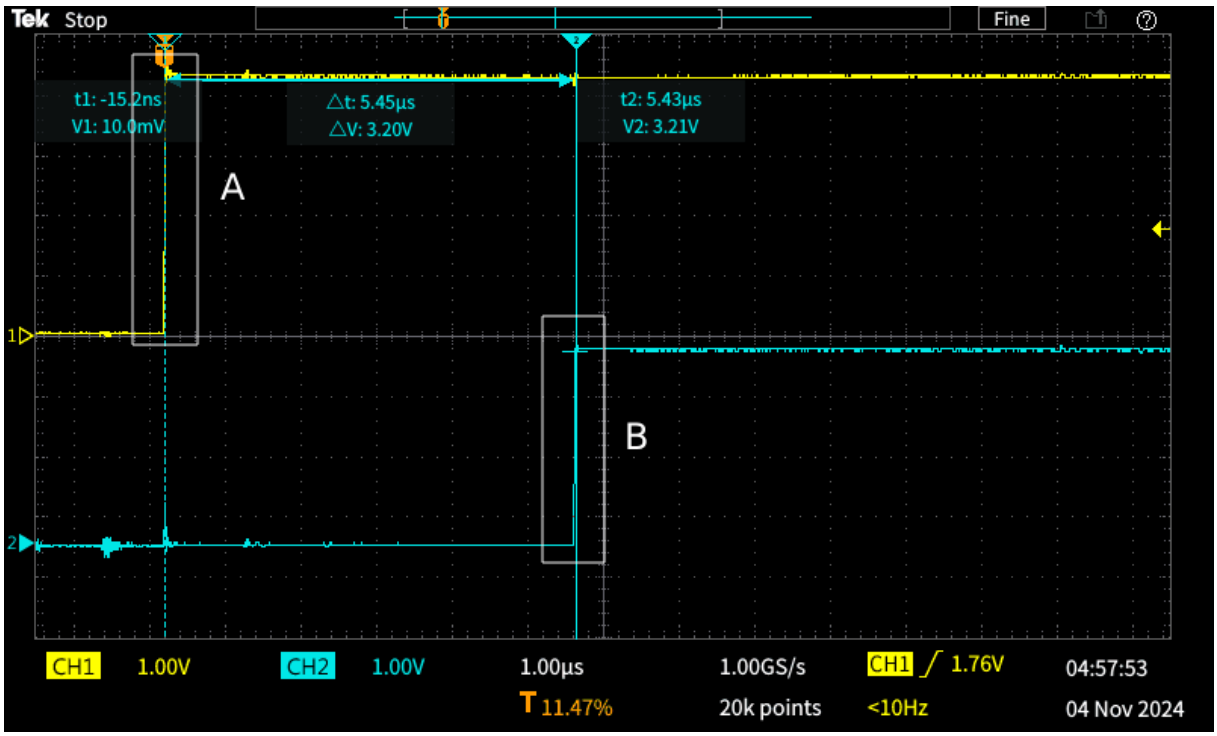
Os resultados obtidos a partir de sessenta amostras para o segundo cenário são apresentados na Tabela 3.

Figura 15 – Borda de descida no FreeRTOS (C) no segundo cenário



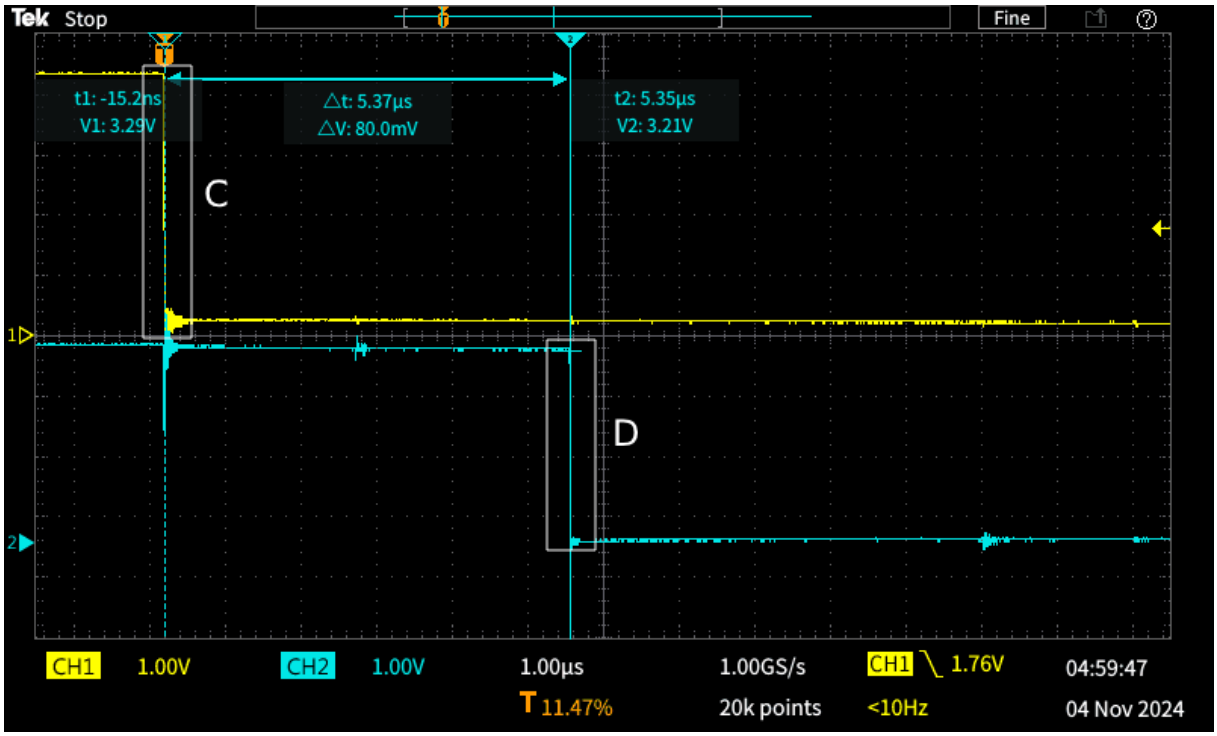
Fonte: Elaborado pelo Author (2024)

Figura 16 – Borda de subida no Embassy (Rust) no segundo cenário



Fonte: Elaborado pelo Author (2024)

Figura 17 – Borda de descida no Embassy (Rust) no segundo cenário



Fonte: Elaborado pelo Author (2024)

Tabela 3 – Métricas temporais no segundo cenário

Métrica	**Embassy (Rust)**	**FreeRTOS (C)**
Latência Mínima (ns)	5413	14320
Latência Máxima (ns)	5427	14438
Latência Média (ns)	5418	14364
Desvio Padrão (ns)	4	24
Jitter (ns)	14	118

Fonte: Elaborado pelo Author (2024)

4.3 COMPARAÇÃO GRÁFICA DAS LATÊNCIAS

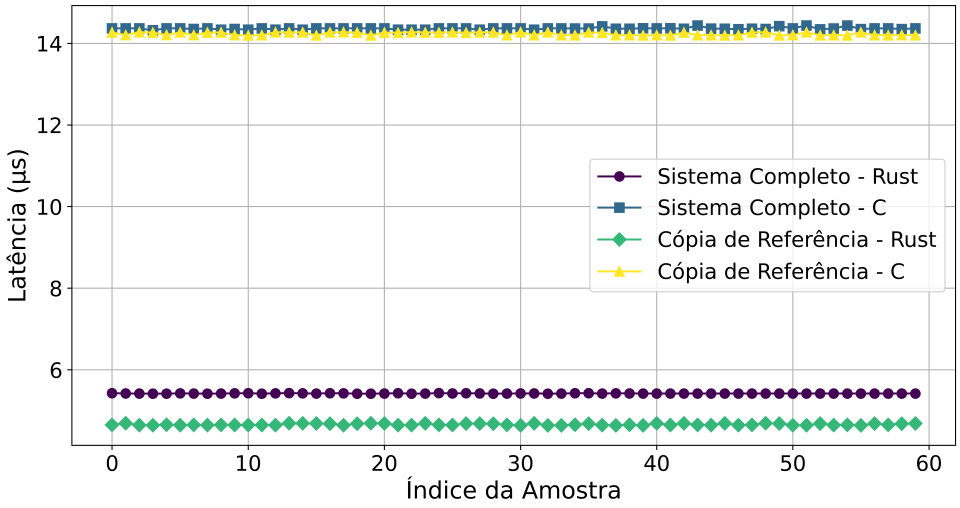
A fim de comparar os quatro experimentos, na Figura 18 é apresentado uma representação gráfica das latências mensuradas para os dois sistemas nos dois cenários.

O gráfico ilustra a diferença de latência entre os sistemas, destacando o desempenho superior do Embassy (Rust) em ambos os cenários.

4.4 USO DE RECURSOS DOS SISTEMAS

Além das métricas de desempenho temporal, o uso de recursos de cada sistema com a aplicação completa também foi avaliado. Nesta seção, são apresentados

Figura 18 – Comparação das latências dos sistemas Embassy (Rust) e FreeRTOS (C) nos dois cenários



Fonte: Elaborado pelo Author (2024)

os resultados referentes ao tamanho dos executáveis, uso de memória da pilha e ao número de símbolos definidos em cada sistema.

4.4.1 Tamanho do Executável

Como mencionado na Seção 2.3, o tamanho do executável de cada sistema foi obtido utilizando ferramentas específicas para cada ambiente. Para o sistema implementado em C com FreeRTOS, foi utilizado o comando `arm-none-eabi-size`, e para o sistema implementado em Rust com Embassy, o comando `cargo size --release`. Os resultados estão apresentados na Tabela 4.

Tabela 4 – Comparação do tamanho dos executáveis

Métrica	**FreeRTOS (C)**	**Embassy (Rust)**
`text` (bytes)	52748	19872
`data` (bytes)	1112	56
`bss` (bytes)	7236	5584
`dec` (bytes)	61096	25512
`data + bss` (bytes)	8348	5640

Fonte: Elaborado pelo Author (2024)

Observa-se que o executável gerado pelo sistema em Rust (Embassy) é significativamente menor em comparação com o executável gerado pelo sistema em C (FreeRTOS). O tamanho da seção `text`, que corresponde ao código executável, é aproximadamente 62% menor no Embassy (Rust). Isso pode ser atribuído a diferença de implementação dos RTOS.

Analisou-se também o uso da memória da pilha, empregada no armazenamento de variáveis locais e no controle de fluxo durante a execução das tarefas. Para essa métrica, considerou-se o uso combinado das seções `.data` e `.bss`. Observou-se que o Rust apresentou um consumo de memória aproximadamente 32% menor em comparação ao C.

4.4.2 Número de Símbolos

O número de símbolos definidos em cada executável reflete a complexidade e a quantidade de funcionalidades implementadas. Para o sistema em C, foi utilizado o comando `arm-none-eabi-nm`, e para o sistema em Rust, o comando `cargo nm -release`. Os resultados são apresentados na Tabela 5.

Tabela 5 – Comparação do número de símbolos do executável

Métrica	**FreeRTOS (C)**	**Embassy (Rust)**
Número de símbolos	791	433

Fonte: Elaborado pelo Author (2024)

O sistema em Rust apresentou um número menor de símbolos em comparação com o sistema em C, indicando uma implementação potencialmente mais enxuta e com menor complexidade.

4.4.3 Análise de resultados

Os resultados evidenciam melhorias nas propriedades temporais do sistema implementado em Rust com Embassy. As métricas de latência e jitter indicam que o sistema em Rust não só responde mais rapidamente aos eventos, como também apresenta menor variabilidade nos tempos de resposta em comparação com o sistema em C com FreeRTOS. No primeiro cenário, o sistema em Rust apresentou uma latência média de 4659 ns, enquanto o sistema em C registrou 14234 ns.

No segundo cenário, com todas as tarefas em execução, a latência média do sistema em Rust aumentou para 5418 ns, representando um incremento de aproximadamente $16,3$%. Já no sistema em C, a latência média aumentou para 14365 ns, um incremento de apenas $0,9$%. Isso indica que, proporcionalmente, o sistema em Rust teve uma variação maior na latência quando todas as tarefas estavam ativas. Uma possibilidade é que isso está relacionado à natureza cooperativa do escalonador do Embassy.

Essa maior variação proporcional no sistema em Rust sugere que ele é mais sensível ao acréscimo de carga de trabalho. No entanto, mesmo com o aumento, a latência absoluta do sistema em Rust permaneceu menor que a do sistema em C. Além disso, o jitter no sistema em Rust reduziu de 51 ns para 14 ns entre os cenários,

enquanto no sistema em C aumentou de 80 ns para 118 ns. Esse comportamento evidencia que o sistema em Rust não apenas manteve tempos de resposta, mas também melhora a consistência temporal nas condições de carga elevada.

Em sistemas de tempo real, é esperado que as respostas apresentem latências mais elevadas com o aumento da carga de trabalho; contudo, é atribuição do RTOS gerenciar as tarefas para evitar grandes variações nos tempos de resposta. O Embassy, aliado ao Rust, demonstrou eficiência nesse gerenciamento, mantendo latências absolutas baixas e jitter reduzido mesmo com todas as tarefas em execução. Essa variação pode ser decorrente de diferenças intrínsecas entre os RTOS ou de variações nas implementações das tarefas em ambos os sistemas.

Passando para o uso de recursos, os resultados demonstram que o sistema implementado em Rust com Embassy utiliza menos recursos em termos de tamanho do executável, uso de memória da pilha e número de símbolos. Em sistemas embarcados, onde a memória flash e RAM disponíveis são limitadas, essa redução é benéfica. Um executável menor e um uso mais eficiente da pilha não só economizam espaço de armazenamento e memória, como também podem contribuir para tempos de inicialização mais rápidos.

A diferença no tamanho do executável, no uso de memória da pilha e no número de símbolos sugere que o Rust, aliado ao Embassy, oferece uma abordagem mais eficiente na gestão de recursos. Isso é possivelmente devido a otimizações em tempo de compilação e ao gerenciamento de memória mais rigoroso proporcionado pela linguagem, que evita alocações desnecessárias e promove o uso consciente dos recursos disponíveis. Além disso, questões relacionadas a implementação do RTOS e tarefas também podem influenciar nesse resultado.

A análise conjunta das propriedades temporais e do uso de recursos reforça os benefícios da utilização da linguagem Rust e do framework Embassy em sistemas embarcados de tempo real, em detrimento do FreeRTOS. Apesar da maior variação proporcional na latência sob carga completa, o Rust mantém latências absolutas inferiores e jitter reduzido em comparação com o C, além de utilizar recursos de forma mais eficiente. Isso indica que o Rust pode ser uma alternativa eficiente e segura para o desenvolvimento de sistemas críticos com restrições de recursos e requisitos rigorosos de desempenho temporal.

No entanto, é importante considerar que a adoção de novas tecnologias deve ser acompanhada de uma avaliação abrangente de seus impactos. Enquanto o Rust oferece benefícios, a familiaridade da equipe de desenvolvimento com a linguagem, o suporte da comunidade e a disponibilidade de bibliotecas e ferramentas também são fatores que influenciam o sucesso de um projeto.

4.5 LIMITAÇÕES

Dado que este trabalho apresentou uma análise das diferenças entre os RTOS FreeRTOS e Embassy em aplicações embarcadas, é fundamental reconhecer as limitações do estudo e possíveis fontes de erro experimental que podem influenciar a interpretação dos resultados.

4.5.1 Limitação de Hardware

A escolha do microcontrolador STM32F103C8T6 foi realizada com base em sua compatibilidade com ambos os RTOS e nos requisitos experimentais. Contudo, esse microcontrolador pertence a uma classe de dispositivos de baixo custo, o que pode limitar a extrapolação dos resultados para sistemas com processadores avançados, que oferecem suporte a sistemas críticos e recursos de hardware mais adequados a esse tipo de sistema.

4.5.2 Cenários Restritos

Os experimentos foram conduzidos em um cenário controlado de controle de fluxo de ar em um tubo, com uma tarefa bem definida de controle de posição da esfera. Este cenário, embora representativo de aplicações embarcadas simples, não contempla sistemas de maior complexidade, como redes distribuídas de sensores ou aplicações com altas taxas de comunicação.

4.5.3 Ambiente de Testes Controlado

Todas as medições foram realizadas em um ambiente de laboratório, com condições de temperatura de $22°C$ e com baixa interferência eletromagnética. Em situações reais, tais como em instalações industriais ou ambientes externos, a presença de ruído e condições adversas pode impactar o desempenho dos RTOS.

4.5.4 Instrumentos de Medição

A precisão do osciloscópio utilizado para medir a latência e o jitter tem uma margem de erro de ± 1 ns, e as medições foram ativadas manualmente, o que pode levar a erros de manuseio. Em sistemas onde esses tempos são críticos, essas imprecisões podem introduzir variações nos resultados e mascarar diferenças reais entre os RTOS.

5 CONCLUSÃO

Em aplicações onde a eficiência, a confiabilidade e o cumprimento de restrições temporais são essenciais, decisões tecnológicas podem influenciar significativamente o comportamento e a segurança do sistema. Diante disso, o objetivo deste estudo foi avaliar o impacto da escolha da linguagem de programação e do sistema operacional de tempo real no desempenho de sistemas embarcados críticos. Para tal, foi realizada uma comparação entre o uso da linguagem Rust, com o framework Embassy, e a linguagem C, com o FreeRTOS, considerando métricas como latência, jitter e uso de memória.

Para investigar essas influências, foi selecionado o sistema de controle de uma planta bola-no-tubo como plataforma de testes. Realizou-se o levantamento dos componentes necessários, a modelagem de hardware e software, bem como a implementação completa do sistema. Essa aplicação permitiu incluir os principais comportamentos de um RTOS: tarefas periódicas com diferentes prioridades, processamento de dados e tarefas aperiódicas. Esse ambiente de teste possibilitou avaliar o desempenho dos sistemas operacionais de tempo real em situações que refletem desafios encontrados em aplicações embarcadas críticas.

Os experimentos foram realizados em dois cenários distintos: um com uma única tarefa de cópia de sinal de entrada e outro com todas as tarefas em execução. Os resultados demonstraram que a implementação em Rust apresentou menor latência e jitter em ambos os cenários. Além disso, o sistema em Rust utilizou menos recursos de memória e armazenamento, evidenciado pelo menor tamanho do executável, uso de memória da pilha e número de símbolos.

Esses resultados indicam que o uso de Rust, aliado ao Embassy, foi alternativa eficiente para o desenvolvimento de sistemas embarcados de tempo real com restrições de recursos. A linguagem proporciona segurança de memória e gerenciamento eficiente de recursos, aspectos críticos para a confiabilidade e desempenho em aplicações de tempo real. Entretanto, é necessário reconhecer como uma limitação desse trabalho que a diferença de desempenho foi validada apenas para os cenários propostos.

Para trabalhos futuros, sugere-se a implementação da automação da coleta de dados, o que aumentaria a eficiência dos experimentos e reduziria possíveis erros humanos. Ademais, realizar comparações em novos cenários de teste ampliaria a abrangência dos resultados e verificaria a consistência das conclusões obtidas. Recomenda-se também um *profiling* do tempo de execução do código, visando identificar quais chamadas são responsáveis pelas diferenças de desempenho e detectar possíveis gargalos no sistema.

Além disso, medir o tempo necessário para a alteração da razão cíclica a partir de uma interrupção pode fornecer percepções sobre a responsividade dos sistemas a eventos externos e avaliar o comportamento em situações de mudança dinâmica. Por fim, ampliar a comparação incluindo outros sistemas operacionais de tempo real possibilitaria uma análise mais abrangente e verificaria se as conclusões obtidas se mantêm em diferentes plataformas.

REFERÊNCIAS

AROCA, R. V.; CAURIN, G.; CARLOS-SP-BRASIL, S. A real time operating systems (rtos) comparison. *In*: **WSO-Workshop de Sistemas Operacionais**. [S.l.: s.n.], 2009. v. 12.

AWS. **FreeRTOS**. USA: [s.n.], 2024. Disponível em: https://www.freertos.org/. Acesso em: oct. 2024.

BARBALACE, A. et al. Performance comparison of vxworks, linux, rtai, and xenomai in a hard real-time application. **IEEE Transactions on Nuclear Science**, IEEE, v. 55, n. 1, p. 435–439, 2008.

BARRY, R. **Mastering the FreeRTOS Real-Time Kernel: A Hands-On Tutorial Guide**. Pre-release 161204 edition. [S.l.]: Real Time Engineers Ltd, 2016. This version covers up to FreeRTOS V8.x.x.

VAN BENEDEN, B. Embedded systems: examining qnx rtos 6.1. **Dr. Dobb's J.**, CMP Media, Inc., USA, v. 27, n. 6, p. 54–60, jun. 2002. ISSN 1044-789X.

BENINGO, J. **How Do You Test RTOS Performance?** 2024. Accessed: 2024-10-23. Disponível em: https://www.embedded.com/how-do-you-test-rtos-performance/.

BOULTON, P. I.; REID, P. A. A process-control language. **IEEE Transactions on Computers**, IEEE, v. 100, n. 11, p. 1049–1053, 1969.

BUTTAZZO, G. C. **Hard real-time computing systems**. [S.l.]: Springer, 1997. v. 356.

COSTA, E. A. da et al. **Projetando com a incerteza: módulos, conexões e sistemas em aberto**. 2019. Tese (Doutorado) — Universidade Federal de Minas Gerais, 2019.

DELTA ELECTRONICS. **Cooler UB0912VH-CX09**. Brasil, 2024. Disponível em: https://www.delta-fan.com/Download/Spec/AUB0912VH-CX09.pdf. Acesso em: jun. 2024.

ECLIPSE FOUNDATION. **ThreadX Utility Benchmarks**. 2024. Accessed: 2024-10-24. Disponível em: https://github.com/eclipse-threadx/threadx/blob/master/utility/benchmarks/thread_metric/thread_metric_readme.txt.

EMBASSY. **Embassy Book**. 2024. Disponível em: https://embassy.dev/book/.

EMBASSY PROJECT CONTRIBUTORS. **Embassy**. 2024. Disponível em: https://embassy.dev/. Acesso em: oct. 2024.

HALANG, W. A.; STOYENKO, A. D. Comparative evaluation of high-level real-time programming languages. **Real-Time Systems**, Springer, v. 2, n. 4, p. 365–382, 1990.

KLABNIK, S.; NICHOLS, C. **The Rust programming language**. [S.l.]: No Starch Press, 2023.

MARWEDEL, P. **Embedded system design: embedded systems foundations of cyber-physical systems, and the internet of things**. [S.l.]: Springer Nature, 2021.

MOTOR AUTHORITY. **The Ford GT Has More Lines of Code than a Boeing Passenger Jet**. 2023. Accessed: 2024-10-23. Disponível em: https://www.motorauthority.com/news/1098308_the-ford-gt-has-more-lines-of-code-than-a-boeing-passenger-jet.

MUSADDIQ, A. et al. A survey on resource management in iot operating systems. **IEEE Access**, IEEE, v. 6, p. 8459–8482, 2018.

NAHAS, M.; MAAITA, A. Choosing appropriate programming language to implement software for real-time resource-constrained embedded systems. **Embedded Systems-Theory and Design Methodology**, InTech, 2012.

OFFICE OF THE NATIONAL CYBER DIRECTOR. **Back to the Building Blocks: A Path Toward Secure and Measurable Software**. [S.l.], 2024. Accessed from National Cybersecurity Strategy resources.

OGATA, K. et al. **Modern control engineering**. [S.l.]: Prentice Hall India, 2009.

OLIVEIRA, R. de. **Fundamentos dos Sistemas de Tempo Real: Segunda Edição**. Amazon Digital Services LLC - Kdp, 2020. ISBN 9798681424635. Disponível em: https://books.google.com.br/books?id=7SjZzQEACAAJ.

PEREIRA, J. S.; BOWLES, J. B. Comparing controllers with the ball in a tube experiment. *In*: IEEE. **Proceedings of IEEE 5th International Fuzzy Systems**. [S.l.], 1996. v. 1, p. 504–510.

PONT, M. J. An object-oriented approach to software development in c for small embedded systems. **Transactions of the Institute of Measurement and Control**, Sage Publications Sage CA: Thousand Oaks, CA, v. 25, n. 3, p. 217–238, 2003.

QIN, B. et al. Understanding and detecting real-world safety issues in rust. **IEEE Transactions on Software Engineering**, IEEE, 2024.

RAMAMRITHAM, K.; STANKOVIC, J. A. Scheduling algorithms and operating systems support for real-time systems. **Proceedings of the IEEE**, IEEE, v. 82, n. 1, p. 55–67, 1994.

RAZA, S.; SEITZ, L.; SITENKOV, D.; SELANDER, G. S3k: Scalable security with symmetric keys—dtls key establishment for the internet of things. **IEEE Transactions on Automation Science and Engineering**, IEEE, v. 13, n. 3, p. 1270–1280, 2016.

RUST ON EMBEDDED DEVICES WORKING GROUP et al. **The Embedded Rust Book**. 2024. Disponível em: https://docs.rust-embedded.org/book/.

SHAFAGH, H.; HITHNAWI, A.; DRÖSCHER, A.; DUQUENNOY, S.; HU, W. Talos: Encrypted query processing for the internet of things. *In*: **Proceedings of the 13th ACM conference on embedded networked sensor systems**. [S.l.: s.n.], 2015. p. 197–210.

STM32-BASE. **STM32-base project**. 2024. Accessed: 2024-10-19. Disponível em: https://stm32-base.org/.

STMICROELECTRONICS. **STMicroelectronics Corporate Information**. 2024. Accessed: 2024-10-19. Disponível em: https://www.st.com.

STMICROELETRONICS. **Sensor de tempo de Voo VL53L0X**. Switzerland: [s.n.], 2024. Disponível em: https://www.st.com/resource/en/datasheet/vl53l0x.pdf. Acesso em: jun. 2024.

WANG, W. et al. Investigation of potential fec schemes for 800g-zr forward error correction. *In*: IEEE. **2022 Optical Fiber Communications Conference and Exhibition (OFC)**. [S.l.], 2022. p. 1–3.