UNIVERSIDADE FEDERAL DA FRONTEIRA SUL
CAMPUS DE CHAPECÓ
CURSO DE CIÊNCIA DA COMPUTAÇÃO

RODRIGO MIGUEL TOMAZI

ANÁLISE DO IMPACTO DA REPROGRAMAÇÃO DO MICROCONTROLADOR ATmega328P NA PREVISIBILIDADE DO FreeRTOS NO ARDUINO UNO

CHAPECÓ
2019

RODRIGO MIGUEL TOMAZI

ANÁLISE DO IMPACTO DA REPROGRAMAÇÃO DO MICROCONTROLADOR ATmega328P NA PREVISIBILIDADE DO FreeRTOS NO ARDUINO UNO

Trabalho de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.
Orientador: Marco Aurélio Spohn

CHAPECÓ
2019

Tomazi, Rodrigo Miguel

Análise do impacto da reprogramação do microcontrolador ATmega328P na previsibilidade do FreeRTOS no Arduino Uno / Rodrigo Miguel Tomazi. – 2019.
66 f.: il.

Orientador: Marco Aurélio Spohn.
Trabalho de conclusão de curso (graduação) – Universidade Federal da Fronteira Sul, curso de Ciência da Computação, Chapecó, SC, 2019.

1. FreeRTOS. 2. Medir atraso. 3. Arduino Uno. 4. *Clock*. 5. Economia de energia. I. Spohn, Marco Aurélio, orientador. II. Universidade Federal da Fronteira Sul. III. Título.

© 2019
Todos os direitos autorais reservados a Rodrigo Miguel Tomazi. A reprodução de partes ou do todo deste trabalho só poderá ser feita mediante a citação da fonte.
E-mail: rodrigo.tomazi@outlook.com

RODRIGO MIGUEL TOMAZI

ANÁLISE DO IMPACTO DA REPROGRAMAÇÃO DO MICROCONTROLADOR ATmega328P NA PREVISIBILIDADE DO FreeRTOS NO ARDUINO UNO

Trabalho de conclusão de curso apresentado como requisito para obtenção do grau de Bacharel em Ciência da Computação da Universidade Federal da Fronteira Sul.

Orientador: Marco Aurélio Spohn

Este trabalho de conclusão de curso foi defendido e aprovado pela banca avaliadora em: 02/12/2019.

BANCA AVALIADORA

Marco Aurélio Spohn – UFFS

Adriano Sanick Padilha – UFFS

Emilio Wuerges – UFFS

AGRADECIMENTOS

Gostaria de agradecer à minha família, por tornar possível a realização desse sonho e por sempre demonstrarem apoio e compreensão durante a trajetória. Aos professores, por revelarem-se mais do que grandes profissionais, mas também amigos, que não mediram esforços para estarmos aqui hoje. Ao orientador, Prof. Marco Aurélio Spohn, por estar sempre disposto a colabolar na realização deste trabalho. Aos colegas e amigos, pela doação e por cada momento compartilhado. E por fim, mas não menos importante, agradeço à UFFS, por ter sido não apenas um espaço de ensino, mas também de troca de experiências e novas convivências.

RESUMO

Esse trabalho tem como objetivo analisar como mudanças nas configurações do microcontrolador afetam a previsibilidade do sistema operacional de tempo real nele embarcado. Para atingir esse objetivo, o *clock* do sistema operacional foi alvejado, onde foram implementados testes que tornaram possível a medição do atraso dos principais recursos do sistema operacional, em diferentes frequências. A troca de contexto e a criação e manipulação dos recursos foram devidamente mensuradas, mostrando um aumento exponencial do atraso. Os temporizadores de *software* também foram analisados e apresentaram algumas anormalidades em alguns cenários. De maneira geral, o atraso dos recursos do FreeRTOS comportou-se de maneira esperada, dobrando a cada vez que a frequência é diminuída pela metade, mostrando proveitoso utilizar a diminuição da frequência, de acordo com a tolerância do sistema, para fins de economia de energia.

Palavras-chave: FreeRTOS. Medir atraso. Arduino Uno. *Clock*. Economia de energia.

ABSTRACT

The purpose of this paper is to analyze how changes in microcontroller settings affect the predictability of the real-time operating system embedded in it. To achieve this objective, the operating system clock was targeted, where tests were implemented that made it possible to measure the delay of key operating system features at different frequencies. Context switching and features creation and manipulation were measured, showing an exponential increase in delay. Software timers were also analyzed and showed some abnormalities in some scenarios. In general, the delay in FreeRTOS features behaved as expected, doubling each time the frequency is halved, and it is advantageous to use the frequency reduction, according to system tolerance, to save energy.

Keywords:FreeRTOS. Measure delay. Arduino Uno. Clock. Energy saving.

LISTA DE ILUSTRAÇÕES

Figura 1 – Modelo de implementação dos testes de criação e exclusão dos recursos . . 32
Figura 2 – Modelo de implementação dos testes de comunicação entre tarefas 33
Figura 3 – Modelo de implementação dos testes de sinalização de uma ISR à uma tarefa 34
Figura 4 – Implementação do teste de sincronização de tarefas 36
Figura 5 – Modelo de implementação dos testes de aquisição e entrega do *mutex* e dos semáforos 37
Figura 6 – Implementação do teste da troca de contexto 37
Figura 7 – Implementação do teste dos temporizadores de *software* 38
Figura 8 – Tempo de criação estática e exclusão das filas 41
Figura 9 – Tempo de criação estática das filas, de acordo com o tamanho da pilha com o processador operando à 16 MHz 43
Figura 10 – Tempo de criação estática e exclusão das tarefas 43
Figura 11 – Tempo para sincronização de tarefas 48
Figura 12 – Esquema de execução das tarefas 50

LISTA DE TABELAS

Tabela 1 – Especificações básicas do ATmega328P 28
Tabela 2 – Fator de divisão do *clock* do ATmega328p 29
Tabela 3 – Consumo para algumas taxas de *clock* do ATmega328p 29
Tabela 4 – Tempo de criação estática e exclusão das filas 40
Tabela 5 – Tempo de criação estática e exclusão do *mutex* e dos semáforos 41
Tabela 6 – Tempo de criação estática e exclusão dos grupos de eventos 42
Tabela 7 – Tempo de criação estática e exclusão das tarefas 42
Tabela 8 – Tempo de envio de uma mensagem entre tarefas a partir de uma fila 44
Tabela 9 – Tempo de envio de uma mensagem entre tarefas a partir de um grupo de eventos 44
Tabela 10 – Tempo de envio de uma mensagem entre tarefas, usando notificações de tarefas 45
Tabela 11 – Tempo de envio de uma mensagem da ISR para a tarefa, usando filas 46
Tabela 12 – Tempo de envio de uma mensagem da ISR para a tarefa, usando grupo de eventos 46
Tabela 13 – Tempo para ISR enviar uma notificação de tarefas 47
Tabela 14 – Tempo para ISR sinalizar à tarefa, usando semáforos 47
Tabela 15 – Tempo para sincronização de tarefas, usando grupo de eventos 48
Tabela 16 – Tempo para aquisição e entrega do *mutex* e dos semáforos 49
Tabela 17 – Tempo para troca de contexto 49

LISTA DE ABREVIATURAS E SIGLAS

CLKPR	*Clock Prescale Register*
CPU	*Central Processing Unit*
IDE	*Integrated Development Environment*
EEPROM	*Electrically-Erasable Programmable Read-Only Memory*
ROM	*Read-Only Memory*
RTOS	*Real Time Operating Systems*
SO	Sistema Operacional
SRAM	*Static Random-Access Memory*
ISR	*Interrupt Service Routine*

SUMÁRIO

1 INTRODUÇÃO 19
1.1 OBJETIVO GERAL 19
1.2 OBJETIVOS ESPECÍFICOS 20
1.3 JUSTIFICATIVA 20
2 REVISÃO BIBLIOGRÁFICA 21
2.1 SISTEMAS OPERACIONAIS 21
2.1.1 Sistemas Operacionais Embarcados 21
2.1.2 Sistemas Operacionais de Tempo Real 22
2.2 FreeRTOS 23
2.2.1 Recursos 23
2.2.1.1 Gerenciamento de memória 23
2.2.1.2 Tarefas 24
2.2.1.3 Filas 25
2.2.1.4 Temporizadores de *software* 25
2.2.1.5 Gerenciamento de interrupções 25
2.2.1.5.1 Semáforos binários 26
2.2.1.5.2 Semáforos de contagem 26
2.2.1.6 Gerenciamento de recursos 26
2.2.1.7 Grupos de eventos 27
2.2.1.8 Notificações de tarefas 28
2.3 ARDUINO UNO 28
2.3.1 IDE e Monitor Serial 28
2.3.2 CLKPR e Consumo de Energia 29
3 TESTES PARA ANÁLISE DE DESEMPENHO 31
3.1 CRIAÇÃO E EXCLUSÃO DOS RECURSOS 31
3.2 MANIPULÇÃO DOS RECURSOS 32
3.2.1 Comunicação entre tarefas 32
3.2.2 Comunicação entre ISR e tarefas 34
3.2.3 Sincronização de tarefas 35
3.2.4 Aquisição e entrega de *mutex* e semáforos 35
3.3 TROCA DE CONTEXTO 36
3.4 TEMPORIZADORES DE *SOFTWARE* 38
4 RESULTADOS E ANÁLISE 39
4.1 CRIAÇÃO E EXCLUSÃO DOS RECURSOS 39
4.1.1 Filas 39
4.1.2 *Mutexes* e Semáforos 40
4.1.3 Grupos de Eventos 41

4.1.4 Tarefas 42
4.2 MANIPULAÇÃO DOS RECURSOS 43
4.2.1 Comunicação entre tarefas 43
4.2.1.1 Filas 43
4.2.1.2 Grupos de eventos 44
4.2.1.3 Notificação de tarefas 44
4.2.2 Comunicação entre ISR e tarefas 45
4.2.2.1 Filas 45
4.2.2.2 Grupos de eventos 46
4.2.2.3 Notificação de tarefas 46
4.2.2.4 Semáforos 47
4.2.3 Sincronização de tarefas 47
4.2.4 Aquisição e entrega de *mutex* e semáforos 48
4.3 TROCA DE CONTEXTO 48
4.4 TEMPORIZADORES DE *SOFTWARE* 49
5 CONCLUSÃO 51
5.1 TRABALHOS FUTUROS 51
REFERÊNCIAS 53
APÊNDICE A – RESULTADO DOS TEMPORIZADORES DE *SOFTWARE* 55
A.1 GRÁFICOS 55
A.2 TABELAS 61

1 INTRODUÇÃO

Um sistema operacional (SO) é um *software* que geralmente opera diretamente sobre o *hardware* e é responsável por gerenciar o mesmo e facilitar o desenvolvimento de programas aplicativos. Existem hoje uma gama de SO's, com as mais variadas finalidades e com recursos bastante robustos para gerenciamento do *hardware*.

Um modelo de SO são os sistemas operacionais de tempo real ou RTOS (Real Time Operating Systems), que tem como principal característica a previsibilidade, ou seja, a garantia de execução de tarefas dentro do prazo estipulado. Devido suas garantias, sistemas críticos podem estar recorrendo ao uso desse tipo de SO. Alguns exemplos de uso, segundo Fassini (2018, p. 19) são sistemas de controle de temperatura de uma usina nuclear e controladores de bordo de avião.

Um exemplo de RTOS é o FreeRTOS, que é um sistema multitarefas e que possui alguns recursos para possibilitar a comunicação e sincronização entre as mesmas e tratamento de eventos externos, além de possuir mecanismos de tempo úteis para execução periódica de funções. O FreeRTOS é um sistema de código aberto mantido atualmente pela *Amazon*, que pode ser modificado e compilado para qualquer dispositivo, como é o caso do *port* para o Arduino Uno.

A maioria dos RTOS que operam sobre sistemas embarcados devem lidar com *hardware* bastante limitado, inclusive a bateria, uma vez que segundo Urriza et al. (2004, p. 01) o avanço das baterias tem sido lento em relação à outros componentes de *hardware*, restando como opção o uso de grandes baterias ou a realização de um gerenciamento de energia eficiente.

Porém a maioria dos RTOS não possuem gerenciamento de energia bem projetado ou não fornece suporte para os modos de baixo consumo, como é o caso do FreeRTOS, que devido a dependência do *hardware*, não fornece recursos de gerenciamento de energia (SIMONOVIĆ; SARANOVAC, 2013, p. 199). Apesar disso, há algumas técnicas que permitem mudar as configurações de alguns microcontroladores, por exemplo, a redução da frequência do *clock* e que acarretam em um menor consumo de energia.

Mas ao utilizar essas técnicas, deve-se ter muito cuidado, pois as alterações do microcontrolador podem tornar o sistema inútil, pois prejudicam o desempenho do microcontrolador e a previsibilidade do RTOS, uma vez que "atrasos para economizar energia podem prejudicar a resposta esperada no tempo predeterminado"(SANTOS, 2017, p. 32). Por isso, esse trabalho pretende analisar como as mudanças nas configurações do microcontrolador afetam a previsivilidade do RTOS.

1.1 OBJETIVO GERAL

Analisar como mudanças nas configurações do microcontrolador, que tem como propósito diminuir o seu consumo de energia, afetam a previsibilidade do sistema operacional de

tempo real nele embarcado.

1.2 OBJETIVOS ESPECÍFICOS

- Analisar como as mudanças no microcontrolador afetam a criação de tarefas, filas, *mutex*, semáforos e grupos de eventos no *port* do FreeRTOS para o Arduino Uno;
- Verificar o atraso na manipulação de filas, grupos de eventos, aquisição e entrega do *mutex* e dos semáforos e envio e recebimento de notificações de tarefas, nas diferentes configurações do microcontrolador;
- Examinar o comportamento dos temporizadores de *software* a partir das alterações no microcontrolador;
- Verificar como as alterações do microcontrolador afetam a latência da troca de contexto.

1.3 JUSTIFICATIVA

Como mudanças no microcontrolador que visam economizar energia acabam afetando o seu desempenho, é de extrema importância ao programador saber as limitações do sistema sob o qual está trabalhando. Por isso, ter conhecimento de atrasos e comportamento dos principais recursos de um RTOS se torna muito importante no desenvolvimento de aplicações para esse tipo de sistema, pois como um RTOS deve atender todas as tarefas em tempo hábil, sistemas mal projetados podem vir a se tornar inúteis futuramente, principalmente sistemas críticos. Apesar disso, a documentação é bastante precária no quesito configurações do microcontrolador e seus impactos, especialmente em placas que não são originalmente suportadas pelo FreeRTOS, como é o caso do Arduino Uno.

2 REVISÃO BIBLIOGRÁFICA

2.1 SISTEMAS OPERACIONAIS

Um SO, é um sistema responsável por gerenciar o *hardware* e permitir a comunicação do *software* aplicativo com o mesmo, criando assim um isolamento entre o *hardware* e o *software*. "Este isolamento permite que a aplicação não sofra alteração quando há mudança no *hardware*." (SANTOS, 2017, p. 3).

Para (TANENBAUM; BOS, 2016, p. 3) os SO's

> realizam duas funções essencialmente não relacionadas: fornecer a programadores de aplicativos (e programas aplicativos, claro) um conjunto de recursos abstratos limpo em vez de recursos confusos de *hardware*, e gerenciar esses recursos de *hardware*.

Enquanto Maziero (2017, p. 3) ressalta que os sistemas operacionais têm como principais objetivos realizar a comunicação entre os programas aplicativos e os dispositivos, tornar os aplicativos independentes do *hardware*, facilitar a interface entre dispositivos de tecnologia distinta e gerenciar os recursos de *hardware*, como processador, memória e dispositivos de entrada e saída.

A falta de um sistema operacional pode ocasionar algumas etapas a mais no desenvolvimento de aplicativos para o sistema, uma vez que, segundo (SANTOS, 2017, p. 3)

> toda a responsabilidade de organizar o andamento dos processos, os acessos ao *hardware* e o gerenciamento da memória é do programador. Este aumento de responsabilidade, a baixa capacidade de reutilização de código, e a consequente necessidade de recriar os códigos e rotinas, podem ser causadores de erros nos programas.

Existem vários tipos de SO's, com os mais diversos propósitos. Nas próximas seções serão apresentados dois tipos, os sistemas operacionais embarcados e os sistemas operacionais de tempo real.

2.1.1 Sistemas Operacionais Embarcados

Um SO embarcado é "construído para operar sobre um *hardware* com poucos recursos de processamento, armazenamento e energia" (MAZIERO, 2017, p. 5). Alguns exemplos de equipamentos que possuem esses sistemas são celulares, micro-ondas, carros e sistemas de alarme.

Esses sistemas embarcados geralmente são projetados para executar funções específicas, privando o usuário de alterar o *software* carregado na ROM. Segundo Tanenbaum (2016, p. 26) a principal propriedade dos SO's embarcados é a certeza de que nenhum outro *software*, a não ser o originalmente carregado na memória, será executado nesses dispositivos, o que acarreta em uma menor necessidade de proteção do sistema e simplificações no *design*.

Como os recursos do *hardware* geralmente são bastante limitados, os maiores desafios para o desenvolvimento de sistemas embarcados é a gerência adequada dos recursos fornecidos e do consumo de energia que, segundo Simonovic (2013, p. 1), tem se tornado uma grande preocupação em sistemas embarcados.

2.1.2 Sistemas Operacionais de Tempo Real

Um RTOS é caracterizado "pela habilidade de atender a eventos com pouco ou nenhum atraso" (SANTOS, 2017, p. 7). Por isso, para Rammig (RAMMIG et al., 2009 apud CHABATURA NETO, 2018, p. 21) a principal característica de um RTOS é a previsibilidade, pois todos os prazos devem ser cumpridos independente das circunstância, onde a corretude do sistema não depende somente dos seus resultados, mas também do tempo em que esses resultados são produzidos.

Geralmente um RTOS nada mais é do “que uma biblioteca conectada com os programas aplicativos, com todas as partes do sistema estritamente acopladas e sem nenhuma proteção entre si” (TANENBAUM; BOS, 2016, p. 26).

Os sitemas RTOS podem ser divididos em categorias. Tanenbaum (2016, p. 26) divide-os em duas categorias, o sistema de tempo real crítico, que deve garantir que a ação irá ocorrer no tempo determinado ou em uma certa faixa de tempo e o sistema de tempo real não crítico, onde uma certa taxa de atrasos ou perda de prazos é aceitável, uma vez que isso não causará danos permanentes, apesar de não ser o desejável.

Já Christofferson (2013, p. 5–6), divide os RTOS em três categorias:

- *Hard real-time applications*: prazos são cruciais e se não forem alcançados inutilizam totalmente o sistema e podem causar consequências catastróficas. Por exemplo: sistemas de controle de motores de carros;

- *Firm real-time applications*: nessa categoria de RTOS, caso o prazo não seja alcançado, não há comprometimento total do sistema, apenas produção de resultados inúteis. Por exemplo: sistemas de processamento de sinais de rádio;

- *Soft real-time applications*: nessa categoria, se os prazos não forem alcançados, não há danos graves, apenas a diminuição da performance. Por exemplo: servidores de rede, onde a perda parcial do dados vai apenas diminuir a qualidade do serviço, mas a aplicação vai continuar utilizável até certa taxa de perdas.

Para (TANENBAUM; BOS, 2016, p. 26), "as categorias de sistemas portáteis, embarcados e de temepo real se sobrepõem consideravelmente".

2.2 FreeRTOS

As informações dessa seção foram extraídas de *"Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide"*, de Richard Barry (2016), exceto os trechos devidamente referenciados.

O FreeRTOS é um RTOS de código aberto desenvolvido originalmente pela *Real Time Engineers Ltd.* mas mantido atualmente pela *Amazon* e que foi desenvolvido com foco em microcontroladores ou pequenos microprocessadores.

Esse RTOS é um sistema multitarefas, ou seja, ele permite a criação de várias tarefas independentes e com diferentes prioridades. A escolha de qual tarefa será executada é tomada por uma ferrramenta chamada escalonador, que levará em conta a prioridade de cada tarefa, onde as tarefas com maior prioridade serão executadas antes do que as tarefas de prioridade menor. A priorização das tarefas pode ser bastante útil se um sistema possuir tarefas de tempo real crítico e não crítico ao mesmo tempo, onde as tarefas de tempo real crítico receberão prioridades maiores do que as tarefas de tempo real não crítico. As tarefas serão melhor apresentadas na subseção 2.2.1.2.

O FreeRTOS é fornecido como um conjunto de códigos fonte em C e está disponível para ser compilado em algumas plataformas, porém, por ser código aberto, o mesmo pode ser ajustado conforme houver necessidade e incorporados em novos dispositivos, como é o caso do o *port* não oficial, ajustado por (STEVENS, 2019) e disponibilizado na plataforma *GitHub*, para o Arduino Uno.

2.2.1 Recursos

O FreeRTOS conta com vários recursos que permitem simplificar o trabalho do programador de aplicativos. Nessa seção serão detalhados seus principais recursos dentro das limitações do *port* para o Arduino Uno.

2.2.1.1 Gerenciamento de memória

Para versões do FreeRTOS posteriores à 9.0.0 os objetos do *kernel* podem ser alocados dinamicamente durante a execução do sistema ou estaticamente durante sua compilação, antes disso a memória era pré-alocada em grupos de blocos com tamanhos diferentes durante a compilação. Esse modo de alocação foi abandonado, pois o gerenciamento de memória não era eficiente, principalmente para os sistemas com *hardware* bastante limitado.

Originalmente, as novas versões trazem 5 implementações para alocação dinâmica de memória, para que os desenvolvedores possam escolher a que melhor se adapta ao seu sistema. No caso do Arduino Uno, na versão disponibilizada por (STEVENS, 2019), somente a variante *"Heap_3"* é utilizada. Essa variante possui duas funções, a ***pvPortMalloc***, para alocar um

espaço na memória e a ***vPortFree***, para liberar um espaço na memória. Essas funções são versões seguras, pois suspende temporariamente o escalonador de tarefas, evitando que outras tarefas assumam a CPU durante a alocação ou liberação da memória.

2.2.1.2 Tarefas

No FreeRTOS as tarefas são implementadas em forma de funções do tipo *void*, ou seja, não possuem retorno e recebem como parâmetro um ponteiro[1] também do tipo *void*. Uma mesma função pode ser usada para criar inúmeras tarefas.

Como já mencionado, uma tarefa não pode conter instruções de retorno e não podem ser executadas até o fim, por isso, geralmente, uma tarefa é implementada dentro de um *loop*[2]. Quando uma tarefa não é mais útil, ela deve ser excluida.

Para criar uma tarefa exitem duas funções, a ***xTaskCreate*** e a ***xTaskCreateStatic***, que aloca a memória para a tarefa estaticamente durante a compilação.

Geralmente um SO é um sistema que suporta a execução de multiplas tarefas e, na maioria das vezes, a quantidade de tarefas é maior do que a quantidade de núcleos de processamento, por isso algumas tarefas devem aguardar para serem executadas. Para que seja possível fazer um gerenciamento de quais tarefas serão executadas, cada tarefa possui um estado, que pode ser:

- Bloqueado: as tarefas entram nesse estado quando estão aguardando por um acontecimento. Pode ser a passagem de um período de tempo ou algum evento externo;
- Pronto: as tarefas estão aguardando para serem executadas pelo processador;
- Em execução: as tarefas estão sendo executadas pelo processador;
- Suspenso: as tarefas só podem ser suspensas pelo usuário, com chamadas de funções específicas e também só podem sair desse estado com chamadas de funções a partir de outras tarefas.

Os SO's possuem um mecanismo conhecido como escalonador, que selecionam qual tarefa será executada a partir das condições definidas. No caso do FreeRTOS, o escalonador seleciona primeiro as tarefas que possuem prioridades mais elevadas. Deve haver muito cuidado na hora da implementação das tarefas e da definição de prioridades, pois tarefas com prioridades altas e que estão sempre em estado pronto não permitirão a execução de tarefas com prioridades mais baixas. A troca de tarefas na CPU é chamada de troca de contexto. A troca de contexto salva a instrução atual da tarefa e a condição da CPU naquele instante e carrega essas mesmas informações de outra tarefa que está no estado pronto.

[1] Tipo de dado que remete à um endereço de memória.
[2] Repetição de um conjunto de instruções.

2.2.1.3 Filas

Uma fila é uma estrutura de armazenamento de dados, onde tarefas ou rotinas de serviço de interrupção[3] (ISR) inserem dados e/ou obtêm dados. As filas, como o nome já sugere, tem como característica principal a ordem de entrada e saída dos dados, onde o primeiro a entrar na fila será o primeiro a ser lido e os novos dados irão entrar sempre no final da fila, porém, o FreeRTOS também implementa uma função onde pode-se inserir um dado no início da fila.

O FreeRTOS permite também a criação de conjuntos de filas, onde é possível integrar várias filas ou semáforos (os semáforos serão apresentados na subseção 2.2.1.5). Isso permite que uma tarefa possa receber dados de diferentes filas ou semáforos sem ter que ficar sondando cada um, pois ele retorna um identificador para acessar diretamente a fila ou semáforo que contém dados.

O FreeRTOS implementa funções para criação de filas, para inserção de elementos, tanto no final da fila quanto no início, para obtenção de um elemento, para ler um elemento sem retirá-lo da fila e para contar o número de itens presentes na fila. Caso uma tarefa tentar retirar um elemento da fila e a mesma estiver vazia, a tarefa pode entrar em estado bloqueado por um tempo determinado, o mesmo pode ocorrer para a tarefa que desejar inserir um elemento na fila, mas a mesma já estiver cheia.

2.2.1.4 Temporizadores de *software*

Temporizadores de *software* é um recurso do FreeRTOS que executa uma função, chamada de "função de retorno de chamada do temporizador", após um tempo definido ou de forma periódica. Geralmente uma função de retorno é uma função curta, que executa até o seu fim e não entra em estado bloqueado, para isso, dentro da função de retorno devem ser usadas as funções seguras[4].

Existem dois tipos de temporizadores, os temporizadores únicos, que executam a função de retorno uma única vez e não reiniciam automaticamente, mas podem ser reiniciados manualmente e os temporizadores de recarregamento automático, que se reiniciam automaticamente a cada vez que expiram, ou seja, executam a função de retorno de forma periódica.

2.2.1.5 Gerenciamento de interrupções

O FreeRTOS utiliza de um recurso chamado de gerenciamento de interrupções para tratar eventos originados a partir do ambiente, como o aperto de um botão ou o acionamento de algum sensor. Para fazer o tratamento de interrupções uma ISR será executada. É recomendado

[3] Uma rotina de serviço de interrupção é um processo de *software* que interrompe a tarefa em execução para poder tratar uma interrupção do sistema.

[4] Funções seguras são funções que não entram no estado bloqueado. A maioria dos recursos do FreeRTOS implementam funções seguras que terminam com *FromISR*.

que uma ISR seja a mais curta possível e, caso seja necessário, a continuação do tratamento da interrupção deverá ser enviada para uma tarefa. Vale ressaltar que dentro do código de uma ISR deverão ser usadas as funções seguras do FreeRTOS, para evitar que ela entre em estado bloqueado.

As interrupções são controladas por *hardware* e, assim como as tarefas, também possuem prioridades, porém, uma tarefa, por mais alta que seja sua prioridade, nunca irá ser executada enquanto houver interrupções para tratar, por mais baixas que sejam suas prioridades. Portanto, se houver muitas interrupções consecutivas e/ou o trecho de código responsável por tratá-las for muito extenso, pode ser que tarefas que tratam interrupções mais importantes e que aconteceram primeiro não sejam executadas no tempo certo. Para que seja possível a transferência do tratamento das interrupções para as tarefas, o FreeRTOS implementa alguns recursos para sincronização, como os semáforos.

2.2.1.5.1 Semáforos binários

Um semáforo binário é um recurso que pode ser usado para sincronização e que é semelhante à uma fila, mas que pode conter no máximo um elemento, portanto ou ela está vazia ou cheia. Para os semáforos não importa o que tem na fila, o que importa é se tem ou não um elemento nela.

Um semáforo binário pode ser usado para que uma rotina de interrupção informe a uma tarefa que determinada interrupção ocorreu e que ela precisa ser tratada, onde a rotina de interrupção é responsável por carregar o semáforo e a tarefa fica aguardando, em estado bloqueado, para obter o semáforo.

Ele é um recurso relativamente simples e é capaz de sinalizar um único acontecimento, ou seja, caso uma interrupção aconteça consecutivamente e a tarefa ainda não recebeu a notificação anterior, ela não saberá que uma nova interrupção aconteceu, pois o semáforo já está cheio. Isso pode ou não ser um problema dependendo da aplicação, mas caso seja, pode ser resolvido usando os semáforos de contagem.

2.2.1.5.2 Semáforos de contagem

Os semáforos de contagem são semelhantes aos semáforos binários, a única diferença é que eles tem comprimento maior que um. Com ele é possível sinalizar à tarefa que processa uma interrupção a quantidade de vezes que ela ocorreu após a última aquisição do semáforo pela tarefa.

2.2.1.6 Gerenciamento de recursos

Em um sistema multitarefas, onde várias tarefas compartilham um mesmo recurso, como os periféricos e endereços de memória, é extremamente importante a implementação de

gerenciadores de recursos, para que não sejam gerados resultados inconsistentes. Pois se, por exemplo, existir um determinado endereço de memória que está sendo compartilhado entre duas tarefas, onde uma delas lê o dado lá presente, realiza operações sobre ele, mas antes de gravar o novo valor ela perde o processador e a outra tarefa que assume o processador manipula esse mesmo dado, então quando a tarefa anterior for gravar o seu novo valor, ela irá substituir o dado por um valor desatualizado. Para resolver esse problema o FreeRTOS oferece implementações que possibilitam exclusão mútua.

A exclusão mútua é uma técnica que tem como objetivo garantir que, enquanto uma tarefa está usando um recurso compartilhado do sistema, nenhuma outra tarefa terá acesso a esse recurso, até que ele seja liberado. O código que acessa os recursos compartilhados é conhecido como região crítica. Para que somente uma tarefa por vez tenha acesso à região crítica, o FreeRTOS disponibiliza dois recursos:

- Suspensão do escalonador: Impede que haja troca de contexto e outra tarefa assuma o processador, ou seja, essa será a única tarefa em execução até que ela mesma chame a função para retomar o escalonador. Vale ressaltar que esse recurso não desativa as interrupções, ou seja, ISR ainda poderão acessar a região crítica.

- *Mutex*: é como se fosse uma permissão para poder acessar a região crítica. Para que uma tarefa possa acessar a região crítica, é necessário primeiramente adquirir o *mutex* e caso o *mutex* já tenha sido adquirido por outra tarefa, a tarefa corrente irá aguardar. Uma tarefa deve sempre entregar o *mutex* ao sair da região crítica, para que outra tarefa possar obtê-lo. Caso a tarefa não entregue, outras tarefas que estão aguardando poderão entrar em *deadlock*[5] e caso ela mesma tentar adquirir novamente o *mutex*, também entrará em *deadlock*.

2.2.1.7 Grupos de eventos

Um grupo de eventos é um conjunto de *bits*, que ficam armazenados dentro de uma mesma variável e cujos valores podem ser 1 para indicar que um evento ocorreu ou 0 para indicar que um evento ainda não ocorreu. Um grupo de eventos também pode ser usado para sincronização e comunicação entre tarefas e ISR.

Uma tarefa ou ISR pode notificar que um evento ocorreu atribuindo 1 aos bits definidos, enquanto outras tarefas ficam aguardando o evento para serem executadas. Se uma tarefa lê o grupo de eventos e um evento ainda não ocorreu ela pode aguardar em estado bloqueado até que o evento ocorra ou por um tempo determinado.

[5] Quando dois ou mais processos ficam em estado bloqueado aguardando uns pelos outros.

2.2.1.8 Notificações de tarefas

Diferente dos recursos vistos até agora, que permitem comunicação entre tarefas através de objetos intermediários, as notificações de tarefas permitem a comunicação direta entre as tarefas ou entre ISR e tarefas. As notificações de tarefas utilizam de uma espécie de variável na tarefa de recebimento, que pode ser modificada pela tarefa de envio.

Uma notificação de tarefa é um recurso mais rápido do que os outros, porém, algumas limitações podem surgir, pois, diferentemente dos anteriores, não é possível enviar uma notificação de uma tarefa para uma ISR, apenas uma tarefa pode ser notificada por vez e diferentemente das filas, a tarefa de envio não poderá entrar em estado bloqueado caso haja notificação pendente na tarefa de recebimento.

2.3 ARDUINO UNO

Segundo o site oficial da placa (AG, 2019), o Arduino Uno é uma placa microcontrolodora de *design* aberto, baseada no microcontrolador ATmega328P. Ela possui 14 pinos de entrada e saída digitais e 6 pinos de entrada analógica, possui uma conexão USB, que permite a comunicação entre o computador e a placa e é usada para gravar o *software* na memória do microcontrolador. Possui também um conector de energia[6] e um botão de reset. Por ter o *hardware* bastante simples, é muito usada para prototipação

O microcontrolador Atmega328p, presente no Arduino Uno, é um microcontrolador programável, produzido pela Atmel. A Tabela 1 mostra as principais especificações do microcontrolador.

Memória *Flash*	32 KB (sendo que 0,5 KB estão reservados para o *bootloader*) [7]
Memória SRAM	2 KB
Memória EEPROM	1 KB
Clock	16 MHz
Tensão de operação	2.7V à 5.5V

Fonte: Elaborado pelo autor com dados de (ATMEGA328P. . . , 2015)

Tabela 1 – Especificações básicas do ATmega328P

2.3.1 IDE e Monitor Serial

O arduino disponibiliza seu próprio ambiente de desenvolvimento integrado ou IDE para programação das placas microcontroladoras. A linguagem utilizada é um conjunto de comandos da linguagem C, C++ e alguns comandos específicos. A IDE está disponível tanto para *desktop*

[6] A placa pode ser alimentada tanto pelo conector de energia quanto pela conexão USB.
[7] Código responsável por fazer a inicialização do sistema

quanto *online* e também oferece a possibilidade de compilar o programa e fazer o *upload* direto para a placa. A IDE *online* permite salvar os códigos na nuvem e compartilhá-los, porém, para fazer o *upload* para a placa, é necessário instalar um *plugin* específico.

As duas IDE's contém praticamente os mesmos recursos e permitem a instalação de bibliotecas externas, como o *port* do FreeRTOS para o Arduino Uno. Além disso, possuem um monitor serial, que é uma ferramenta utilizada para realizar a comunicação entre o computador e a placa.

2.3.2 CLKPR e Consumo de Energia

O microprocessador ATMega328P possui um registrador chamado de *Clock Prescale Register* (ou CLKPR), que pode ser usado para alterar o *clock* da CPU e consequentemente o seu consumo de energia. O CLKPR divide o *clock* por valores específicos chamados fatores de divisão, que podem ser vistos na Tabela 2.

Fator de divisão	*Clock* da CPU
1	16 MHz
2	8 MHz
4	4 MHz
8	2 MHz
16	1 MHz
32	500 kHz
64	250 kHz
128	125 kHz
256	62,5 kHz

Fonte: Elaborado pelo autor com dados de (ATMEGA328P. . . , 2015)

Tabela 2 – Fator de divisão do *clock* do ATmega328p

O *datasheet* não trás muitas informações sobre o consumo de energia para as todas as frequências de *clock* apresentadas na Tabela 2, porém ele apresenta alguns consumos para frequências iguais à 4, 8 e 16 MHz, que são apresentados na Tabela 3, onde o consumo médio é obtido com a placa operando à uma temperatura de 25°C.

Clock	Tensão de operação	Consumo médio(25°C)	Consumo máximo
4 MHz	3V	1,5 mA	2,4 mA
8 MHz	5V	5,2 mA	10 mA
16 MHz	5V	9,2 mA	14 mA

Fonte: Elaborado pelo autor com dados de (ATMEGA328P. . . , 2015)

Tabela 3 – Consumo para algumas taxas de *clock* do ATmega328p

A biblioteca *avr-libc* contém uma biblioteca intena que pode ser incluída via <*avr/power.h*> e tem suporte ao Arduino Uno. Essa biblioteca, disponível em (LIBC, 2018),

permite a alteração do *clock* a partir da função *void clock_prescale_set (clock_div_t __x)*, que recebe como parâmetro *__x* o fator de divisão, onde *__x* pode assumir:

```
typedef enum
{
    clock_div_1 = 1,
    clock_div_2 = 2,
    clock_div_4 = 4,
    clock_div_8 = 8,
    clock_div_16 = 16,
    clock_div_32 = 32,
    clock_div_64 = 64,
    clock_div_128 = 128
} clock_div_t;
```

3 TESTES PARA ANÁLISE DE DESEMPENHO

Para Sacha (1995, p. 1), os requisitos essenciais em SO's de tempo real são a pontualidade e a confiabilidade, que podem ser atingidos usando os princípios de multitarefas, comunicação e sincronização eficiente de tarefas e manipulação eficiente de tempo e eventos. Mas ao usar o CLKPR com o propósito de diminuir o consumo de energia, o atraso do FreeRTOS pode sofrer alguns impactos e afetar esses requisitos essenciais. Por isso, foram testados tempos de criação e manipulação dos recursos do FreeRTOS, da troca de contexto e temporizadores de *software*. Em cada teste exposto nas subseções seguintes, nenhuma tarefa ou rotina alheia às apresentadas estavam em execução.

Os testes estão disponíveis na plataforma Zenodo, que podem ser acessados através de (TOMAZI, 2019)

3.1 CRIAÇÃO E EXCLUSÃO DOS RECURSOS

Os testes de criação e exclusão seguiram o modelo da Figura 1, que apresenta a tarefa principal, responsável por medir o tempo de criação e exclusão dos recursos. O tipo da variável *xRecurso* é relativo a cada recurso testado e representa o identificador do mesmo. Para realizar as medições de tempo, a tarefa entra em um laço que cria as variáveis – cujos ponteiros devem ser passados como parâmetro para a função de criação–, armazena o tempo do sistema na variável *início*, executa a função de criação do recurso e armazena o tempo do sistema na variável *fim*. Caso o recurso tenha sido efetivamente criado, a tarefa soma na variável *mediaTempoCriacao* a diferença das variáveis *fim* e *início* – que representa o tempo que a tarefa levou para criar o recurso – , armazena novamente o tempo do sistema na variável *início*, exclui o recurso, armazena o tempo do sistema na variável *fim* e soma na variável *mediaTempoExclusao* a diferença das variáveis *fim* e *início* – que representa o tempo que a tarefa levou para excluir o recurso. Se a tarefa não foi criada, essa iteração deverá ser executada novamente, para isso, incrementa-se 1 na variável que conta quantas iterações ainda restam. Por fim, o tempo médio de criação e exclusão é calculado e enviado para o monitor serial. O cálculo se dá dividindo ambas as variáveis pelo número total de iterações.

Um outro modelo de teste pode incluir a criação das variáveis que devem ser passadas como parâmetro para a função de criação depois da primeira medição de tempo, para que possa-se medir o tempo da criação das variáveis e da chamada da função de criação do recurso como uma única operação. O resultado para ambos os modelos é o mesmo, uma vez que a memória é alocada durante a compilação e portanto, não altera o tempo de execução.

Figura 1 – Modelo de implementação dos testes de criação e exclusão dos recursos

```
void vTarefaPrincipal(void *){
  unsigned int inicio = 0, fim = 0, i = ITERACOES;
  float mediaTempoCriacao = 0.0, mediaTempoExclusao = 0.0;

  do{
    //Declaração das variáveis necessárias de acordo com cada recurso
    inicio = micros();
    //Espaço para executar a função de criação do recurso
    fim = micros();

    if(xRecurso != NULL){
      mediaTempoCriacao += (float)(fim - inicio);
      inicio = micros();
      //Espaço para executar a função de exclusão do recurso
      fim = micros();
      mediaTempoExclusao += (float)(fim - inicio);
      xRecurso = NULL;
    }else
      i++;
  }while(i--);

  mediaTempoCriacao /= (float)ITERACOES;

  mediaTempoExclusao /= (float)ITERACOES;

  Serial.print("Tempo de criação: ");
  Serial.println(mediaTempoCriacao);
  Serial.print("Tempo de exclusão: ");
  Serial.println(mediaTempoExclusao);
  vTaskDelete(NULL);
}
```

3.2 MANIPULÇÃO DOS RECURSOS

3.2.1 Comunicação entre tarefas

O FreeRTOS oferece alguns recursos que possibilitam a comunicação entre as tarefas, como filas, grupos de eventos e notificação de tarefas.

Para Sacha (1995, p. 2), duas ações devem ser executadas entre o início do envio da mensagem e o final do recebimento, que são a entrega da mensagem pelo SO e a alternância das tarefas. Por isso, ele sugere que devemos medir o tempo total da operação, pois é a menor parte que pode ser observada na vida real e a sua divisão é irrelevante para um programador.

Os testes para analisar a comunicação entre tarefas seguem o modelo de Sacha (1995, p. 3) e podem ser vistos na Figura 2. Eles consistem em duas tarefas com a mesma prioridade. A tarefa principal tem como objetivo calcular o tempo do envio de uma mensagem. Para isso ela captura o tempo do sistema em microssegundos e entra em um laço. O laço envia a mensagem para a tarefa secundária e aguarda em estado bloqueado por uma mensagem idêntica originada na tarefa secundária. A tarefa secundária, por sua vez, ficará em um laço por tempo indeterminado, aguardando, em estado bloqueado, pela mensagem da tarefa principal e ao receber, respondendo-a com uma mensagem idêntica. Ao receber a mensagem, a tarefa principal volta para o início do laço ou sai dele caso atingir a quantidade de iterações definidar. Ao sair do laço, ela captura

o tempo novamente e calcula o tempo médio para a troca de uma única mensagem. O cálculo se dá pegando o tempo total, dividindo-o pelo número de iterações e divindo-o por 2, uma vez que para cada mensagem enviada pela tarefa principal, uma idêntica foi enviada pela tarefa secundária.

Figura 2 – Modelo de implementação dos testes de comunicação entre tarefas

```
void vTarefaPrincipal(void *){
  volatile uint32_t i = ITERACOES;
  uint32_t inicio = 0, fim = 0;
  float mediaTempo = 0.0;

  inicio = micros();
  do{
    //Envia a mensagem para a tarefa secundária
    //Aguarda pela mensagem da tarefa secundária
  }while(i--);
  fim = micros();

  mediaTempo = float(fim - inicio) / (float)ITERACOES;
  mediaTempo /= 2.;

  Serial.println(mediaTempo);
  vTaskDelete(NULL);
}
```

```
void vTarefaSecundaria(void *){
  do{
    //Aguarda pela mensagem da tarefa primária
    //Envia a mensagem para a tarefa primária
  }while(1);
}
```

Os testes englobaram os seguintes recursos:

- **Filas**. Os testes aconteceram usando duas filas. A *xFila1*, usada para mandar a mensagem à tarefa secundária pela tarefa primária e a *xFila2*, usada para mandar a mensagem à tarefa primária pela tarefa secundária.
- **Grupo de eventos**. Dois grupos de eventos foram usados. No mesmo esquema das filas, o *xGrupoDeEventos1* é manipulado pela tarefa principal e aguardado pela tarefa secundária, enquanto o *xGrupoDeEventos2* é manipulado pela tarefa secundária e aguardado pela tarefa primária, realizando assim a comunicação entre as tarefas para informar que um evento aconteceu.
- **Notificação de tarefas**. Diferente dos recursos anteriores, não há necessidade de criar os recursos a serem usados, apenas salvar os identificadores das tarefas no momento de criação das mesmas. Existem dois modelos de notificação de tarefas:
  - O modelo básico apenas notifica determinada tarefa que um evento ocorreu. Não é possível passar valores para a tarefa. Esse modelo é implementado usando a função *xTaskNotifyGive* para enviar a notificação e *ulTaskNotifyTake* para receber a notificação.
  - O modelo configurável é implementado usando a função *xTaskNotify* para enviar a notificação e a função *xTaskNotifyWait* para receber a notificação. Esse modelo possui vários modos de ação:
    - *eNoAction* - A mesma função do modelo mais simples de notificação de tarefas, apenas informa à tarefa que há uma notificação, mas não altera o valor de notificação da mesma. O parâmetro *ulValue* da função não é usado.

* *eSetBits* - Realiza um *OR bit* a *bit* entre o valor atual de notificação da tarefa e o valor passado no parâmetro *ulValue*.
* *eIncrement* - O valor de notificação da tarefa de recebimento é incrementado em 1. O parâmetro *ulValue* não é usado.
* *eSetValueWithoutOverwrite* - Caso a tarefa de recebimento está com uma notificação pendente nenhuma ação será executada, caso contrário, o valor de notificação da tarefa de recebimento receberá o valor passado em *ulValue*.
* *eSetValueWithOverwrite* - O valor de notificação da tarefa recebe o valor passado em *ulValue*, independente de a tarefa de recebimento ter ou não uma notificação pendente.

3.2.2 Comunicação entre ISR e tarefas

Uma interrupção de *software* executa uma rotina que faz o tratamento da interrupção. Caso uma rotina se torne muito complexa, a continuação do tratamento deve ser continuada em uma tarefa. Para que a ISR sinalize à tarefa que deve continuar com o tratamento, alguns recursos podem ser usados, que podem incluir os mesmos usados na comunicação entre tarefas e também os semáforos.

Para Sacha (1995), a operação de sinalização é constituída pelas instruções de geração de sinal, entrega de sinal, captura de sinal e alternância de tarefas.

Os testes para analisar a sinalização de uma ISR à uma tarefa seguem o modelo da Figura 2. Eles consistem em uma tarefa primária, uma secundária, que irá gerar a interrupção de *software* e uma ISR, que repassa o sinal à tarefa principal, onde esta estará aguardando em estado bloqueado. A ISR irá armazenar o tempo antes de enviar o sinal à tarefa, que por sua vez, medirá o tempo após o recebimento do sinal. Após receber a quantidade de sinais definidas, a tarefa principal sairá do laço e calculará o tempo médio de cada sinal. Esse cálculo se dá pegando o tempo total e dividindo-o pelo número de iterações. A tarefa secundária tem prioridade menor do que a tarefa principal e é usada para gerar as interrupções, sendo executada somente quando a tarefa principal e a ISR não estão sendo executadas ou em estado pronto, no caso da tarefa.

Figura 3 – Modelo de implementação dos testes de sinalização de uma ISR à uma tarefa

```
uint32_t somaTempo = 0;
void vTarefaPrincipal(void *){
  volatile uint32_t i = ITERACOES;

  do{
    //Aguarda pelo sinal da ISR
    somaTempo += micros();
  }while(i--);

  float mediaTempo = float(somaTempo) / (float)ITERACOES;

  Serial.println(mediaTempo);
  vTaskDelete(NULL);
}
```

```
void vTarefaSecundaria(void *){
  do{
    digitalWrite(pinoInterrupcao, LOW);
    digitalWrite(pinoInterrupcao, HIGH);
    taskYIELD();
  }while(1);
}
```

```
static void ulTrataInterrupcaoISR(void){
    BaseType_t xHigherPriorityTaskWoken = pdTRUE;
    somaTempo -= micros();
    //Envia o sinal para a tarefa primária
}
```

Os testes se deram usando os seguintes recursos:

- **Filas**. Os testes aconteceram usando uma única fila, que manda um sinal da ISR à tarefa primária.
- **Grupo de eventos**. Um grupo de eventos foi usado onde, semelhante às filas, o grupo de eventos manda um sinal da ISR à tarefa primária.
- **Notificação de tarefas**. Os dois mesmos modos da notificação de tarefa usados na comunicação entre tarefas foram testados, onde a tarefa que vai continuar o tratamento recebe uma notificação emitida pela ISR.
- **Semáforos**. Como o semáforo é uma espécie de autorização, ele também pode ser usado como forma de sinalizar à tarefa que determinado evento ocorreu. Então, de maneira semelhante aos demais, a tarefa principal aguarda em estado bloqueado até que a ISR autorize à tarefa a continuar sua execução. Foram testados os semáforos binários e contadores.

3.2.3 Sincronização de tarefas

A sincronização de tarefas foi testada usando a função *xEventGroupSync*, que usa grupo de eventos. Essa função diz ao grupo de eventos que ela está pronta e entra em estado bloqueado até que todas estejam prontas ou até que o tempo de espera expire.

A Figura 4 mostra a tarefa que testa a sincronização. Essa tarefa atribui à variável *uxBitDaTarefa* o valor recebido por parâmetro, que representa o *bit* da tarefa no grupo de eventos. Após, ela cria uma variável chamada *uxTodosBitsDeSicronizacao*, que contém os *bits* de todas as tarefas que irão aguardar para sincronização. Criado todas as variáveis necessárias, a tarefa irá bloquear até que todas as outras também tenham criado as variáveis, ou seja, ela vai sincronizar com as demais, para depois iniciar a medição do tempo. O tempo é armazenado antes da tarefa entrar no laço. Ao entrar no laço, a tarefa irá sincronizar com as demais por determinadas vezes. Ao sair do laço, a tarefa irá ler o tempo novamente e entrar em estado bloqueado até que todas tenham armazenado o tempo. Apos, é realizado o cálculo do tempo médio de cada sincronização da tarefa corrente, que se dá diminuindo o tempo final do tempo inicial e dividindo-o pela quantidade de iterações. Por fim, cada tarefa irá suspender o escalonador, para evitar que outra tarefa assuma a CPU enquanto ela estiver escrevendo no monitor serial o seu tempo médio de sincronização.

O tempo médio final é dado fazendo uma média aritmética simples entre os resultados de cada tarefa.

3.2.4 Aquisição e entrega de *mutex* e semáforos

Foram feitos testes para medir o tempo de aquisição e entrega do *mutex*, semáforo binário e semáforo contador. A Figura 5 mostra a tarefa que mede esses recursos. Para isso, a tarefa

Figura 4 – Implementação do teste de sincronização de tarefas

```
EventGroupHandle_t xGrupoDeEventos;
#define BIT_TAREFA (1UL << b)

void vTarefaSincronizacao(void *pvParameters){
  volatile uint32_t i = ITERACOES;
  uint32_t inicio = 0, fim = 0;
  float mediaTempo = 0.0;

  EventBits_t uxBitDaTarefa = (EventBits_t) pvParameters;

  EventBits_t uxTodosBitsDeSicronizacao;
  for(unsigned long b = 0; b < QNT_TAREFAS; b++)
    uxTodosBitsDeSicronizacao |= BIT_TAREFA;

  xEventGroupSync(xGrupoDeEventos, uxBitDaTarefa, uxTodosBitsDeSicronizacao, portMAX_DELAY);

  inicio = micros();
  do{
    xEventGroupSync(xGrupoDeEventos, uxBitDaTarefa, uxTodosBitsDeSicronizacao, portMAX_DELAY);
  }while(i--);
  fim = micros();

  xEventGroupSync(xGrupoDeEventos, uxBitDaTarefa, uxTodosBitsDeSicronizacao, portMAX_DELAY);

  mediaTempo = float(fim - inicio) / (float)ITERACOES;

  vTaskSuspendAll();
  Serial.println(mediaTempo);
  xTaskResumeAll();
  vTaskDelete(NULL);
}
```

entra em um laço que executa a instrução de aquisição e entrega por uma determinada quantidade de vezes. Para cada aquisição e entrega, o tempo é medido e somado a respectiva variável. Por fim, ao sair do laço, os tempos são calculados, dividindo a soma de cada um pela quantidade de iterações.

3.3 TROCA DE CONTEXTO

Segundo Sacha (1995, p. 6), o tempo consumido pela alternância de tarefas representa uma sobrecarga pura em qualquer SO multitarefas portanto, a duração desse tempo é um recurso importante que caracteriza a eficiência geral do SO. Para medição desse tempo, foi utilizado o modelo proposto por Sacha (1995, p. 7), que pode ser visto na Figura 6.

O teste possui duas tarefas com a mesma prioridade. A tarefa principal é responsável por medir o tempo. Primeiramente ela mede o tempo do laço vazio e calcula o tempo médio de cada iteração. O segundo passo é medir o tempo da troca de contexto. Para isso, um laço é responsável por executar a instrução que entrega a CPU por uma certa quantidade de vezes, onde tempo é medido antes e depois da execução do laço. Para cada iteração do laço, a tarefa principal entrega a CPU, forçando a CPU a alternar para a terefa secundária, que por sua vez, tem como único objetivo devolver a CPU para a tarefa principal, ou seja, para cada troca de contexto realizada entre a tarefa principal e a secundária, outra é realizada de maneira oposta. O cálculo final é dado dividindo o tempo total, obtido subtraindo-se o tempo inicial do tempo

Figura 5 – Modelo de implementação dos testes de aquisição e entrega do *mutex* e dos semáforos

```
void vTarefaPrincipal(void *){
  volatile uint32_t i = ITERACOES;
  uint32_t inicio = 0, fim = 0;
  float mediaTempoAquisicao = 0.0, mediaTempoEntrega = 0.0;

  do{
    inicio = micros();
    //Adquire o recurso
    fim = micros();
    mediaTempoAquisicao += float(fim - inicio);
    inicio = micros();
    //Entrega o recurso
    fim = micros();
    mediaTempoEntrega += float(fim - inicio);
  }while(i--);

  mediaTempoAquisicao /= (float)ITERACOES;

  mediaTempoEntrega /= (float)ITERACOES;

  Serial.print("Aquisicao: ");
  Serial.print(mediaTempoAquisicao);
  Serial.print(" - Entrega: ");
  Serial.println(mediaTempoEntrega);
  vTaskDelete(NULL);
}
```

final, pela quantidade de iterações e dividindo esse resultado por 2. Como já mencionado, a divisão por 2 é necessária pois, a cada iteração, duas trocas de contexto são realizadas, uma da tarefa principal para a secundária e outra ao oposto. Para finalizar o cálculo do tempo médio da troca de contexto, o tempo médio de cada iteração do laço é descontado.

Figura 6 – Implementação do teste da troca de contexto

```
void vTarefaPrincipal(void *){
  volatile uint32_t i = ITERACOES;
  uint32_t inicio = 0, fim = 0;
  float mediaTempo = 0.0, mediaTempoLaco = 0.0;

  inicio = micros();
  do{
  }while(i--);
  fim = micros();

  mediaTempoLaco = float(fim - inicio) / float(ITERACOES);

  i = ITERACOES;

  inicio = micros();
  do{
    taskYIELD();
  }while(i--);
  fim = micros();

  mediaTempo = ((float(fim - inicio) / (float)ITERACOES)
               / 2.0);
  mediaTempo = mediaTempo - mediaTempoLaco;

  Serial.println(mediaTempo);
  vTaskDelete(NULL);
}
```

```
void vTarefaSecundaria(void *){
  do{
    taskYIELD();
  }while(1);
}
```

3.4 TEMPORIZADORES DE *SOFTWARE*

Os temporizadores também foram testados. A Figura 7 mostra a implementação da função que é executada pelo temporizador a cada vez que ele expirar. A variável *TEMPO_TICKS* armazena o tempo do temporizador, em *ticks*[8]. O temporizador usado nesse teste foi um temporizador de recarregamento automático que executa a função de retorno *FuncaoRetorno*. A função de retorno tem como objetivo mandar para o monitor serial o tempo interno do sistema em microssegundos, para que possa-se verificar a diferença de tempo entre as execuções do temporizador. Após 51 execuções da função (50 diferenças de tempo entre as execuções do temporizador), o temporizador é parado, seu tempo é incrementado em 1 *tick* e o temporizador é reiniciado.

Também foram usadas tarefas chamadas de *vTarefaX*, que tem como objetivo disputar a CPU com a *daemon*[9], para verificar como os temporizadores se comportam ao não conseguir a CPU quando expirarem.

Figura 7 – Implementação do teste dos temporizadores de *software*

```
volatile uint16_t TEMPO_TICKS, i;
TimerHandle_t xTimer;

void FuncaoRetorno(TimerHandle_t xTimer){
  Serial.println(micros());
  i++;
  if(i >= 51){
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    xTimerStopFromISR(xTimer, &xHigherPriorityTaskWoken);
    i = 0;
    TEMPO_TICKS++;
    Serial.print("Tempo : ");
    Serial.print(TEMPO_TICKS * portTICK_PERIOD_MS);
    Serial.println(" ms");
    xTimerChangePeriodFromISR(xTimer, TEMPO_TICKS, &xHigherPriorityTaskWoken);
  }

}
```

```
void vTarefaX(void *){
  for( ;; );
}
```

[8] Interrupção de *hardware* que ocorre de forma periódica. Todas as funções que envolvem tempo no FreeRTOS são expressadas em *ticks*.

[9] Tarefa responsável por controlar a expiração dos temporizadores de *software* e a execução da função de retorno

4 RESULTADOS E ANÁLISE

Os testes foram realizados usando a versão 10.2.0-3 do FreeRTOS, com suas configurações padrões, rodando sobre o Arduino Uno Rev3 e para frequências da CPU iguais à 16 MHz (*clock_div_1*), 8 MHz (*clock_div_2*), 4 MHz (*clock_div_4*), 2 MHz (*clock_div_8*) e 1 MHz (*clock_div_16*). As frequências mais baixas, alcançadas por *clock_div_32*, *clock_div_64* e *clock_div_128*, apesar de estarem disponíveis, não são suportadas pelo Arduino Uno pois, existe uma constante interna identificada como *clockCyclesPerMicrosecond*, que representa a quantidade de ciclos de *clock* por microssegundos e é o resultado da frequência da CPU dividido por 1.000.000 (quantidade de microssegundos em um segundo). Como a divisão é realizada entre dois inteiros, frequências menores que 1.000.000 arredondam a constante para 0 e sendo que essa constante é utilizada como divisor em outras definições, ela acaba fornecendo uma divisão por 0, que é matematicamente impossível.

Todos os testes de tempo de criação e manipulação dos recursos foram realizados dentro de tarefas para verificar o atraso do sistema para reagir a um comando durante sua execução. Tanto os recursos quanto a troca de contexto foram testados com o número de iterações igual à 1000.

Os resultados também estão disponíveis na plataforma Zenodo, que podem ser acessados através de (TOMAZI, 2019)

4.1 CRIAÇÃO E EXCLUSÃO DOS RECURSOS

Os testes de criação dos recursos foram realizados com as funções de criação estática, funções essas que terminam com o termo *Static*. As funções que utilizam a alocação dinâmica não foram executadas pelo FreeRTOS, apontando erro de memória. Isso se deu devido à *heap* disponibilizada para esse *port* utilizar a função de *malloc* do próprio arduino para alocar a memória, porém essa função não executa dentro das tarefas.

Vale ressaltar que as funções de exclusão referente a cada recurso foram usadas e tiveram seu tempo de execução medido, porém não se sabe se o espaço de memória foi de fato liberado.

4.1.1 Filas

A criação e exclusão das filas foram testadas usando uma quantidade de itens iguais à 1, 8 e 16 e com tamanhos iguais à 1, 8 e 16 bytes. O resultado dos testes, presentes na Tabela 4, mostram que como era de se esperar, o número de itens e o tamanho deles não influencia no tempo de criação e exclusão das filas, pois a memória para os itens já havia sido alocada no momento de compilação e é repassada como ponteiro para a função de criação.

O gráfico da Figura 8 mostra a relação do tempo com a frequência. Pode-se ver que a frequência da CPU tem relação direta ao tempo de criação e exclusão dos itens. Para a criação,

CRIAÇÃO DAS FILAS						
Número de itens	**Tamanho dos itens**	**Tempo médio (microssegundos)**				
		16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
1	1 byte	12,33	27,18	54,70	108,67	218,56
	8 bytes	12,25	27,30	54,94	109,73	221,50
	16 bytes	12,28	27,50	55,06	109,57	221,18
8	1 byte	12,28	27,32	54,94	110,11	221,66
	8 bytes	12,28	27,50	55,06	109,38	220,86
	16 bytes	12,30	26,95	55,42	109,06	221,31
16	1 byte	12,28	27,32	55,06	109,38	220,86
	8 bytes	12,30	26,97	55,41	109,09	221,31
	16 bytes	12,27	27,39	55,2	109,41	222,08
EXCLUSÃO DAS FILAS						
Número de itens	**Tamanho dos itens**	**Tempo médio (microssegundos)**				
		16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
1	1 byte	2,34	7,68	17,01	38,27	74,37
	8 bytes	2,37	7,68	17,39	36,93	72,70
	16 bytes	2,32	7,68	17,41	37,38	74,24
8	1 byte	2,35	7,70	17,39	37,92	72,83
	8 bytes	2,32	7,68	17,41	37,63	74,62
	16 bytes	2,79	7,27	17,04	38,24	73,15
16	1 byte	2,32	7,70	17,41	37,63	74,62
	8 bytes	2,79	7,30	16,86	38,75	73,15
	16 bytes	2,39	7,66	17,2	37,57	73,41

Fonte: Elaborado pelo autor

Tabela 4 – Tempo de criação estática e exclusão das filas

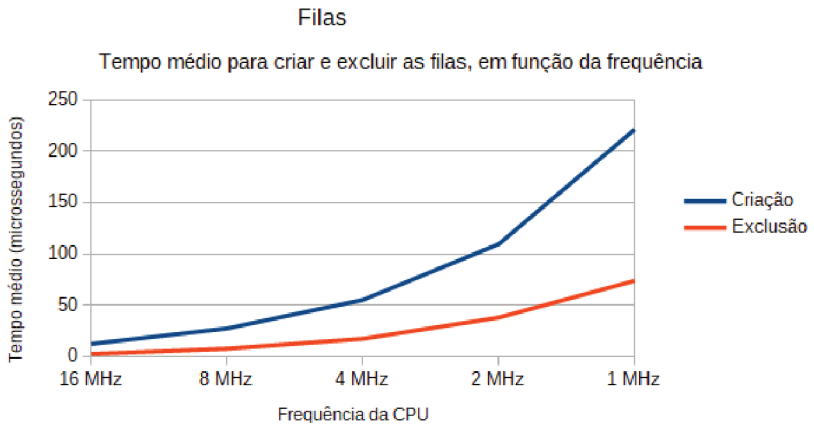
o tempo dobra a cada divisão da frequência, apresentando um aumento exponencial do tempo, exceto aos 8 MHz, que tem um aumento um pouco maior do que o dobro em relação à frequência de 16 MHz.

Já o tempo de exclusão, também exponencial, tem mostrado um aumento mais rápido na mudança entre as frequências mais altas, uma vez que o tempo na frequência de 8 MHz foi 3 vezes maior em relação ao tempo na frequência de 16 MHz, porém tem diminuido seu aumento de tempo à medida que a frequência diminui, chegando a um aumento um pouco menor de 2 vezes na frequência de 1 MHz em relação à de 2 MHz, ou seja, a taxa de aumento do tempo cai com a diminuição da frequência.

4.1.2 *Mutexes* e Semáforos

A Tabela 5 mostra os tempos de criação e exclusão dos *mutexes*, semáforos binários e semáforos contadores. Como esses recursos são implementados usando filas, os tempos são muito parecidos com os das mesmas. Todas as funções de criação desses recursos criam uma fila genérica, a única diferença é que o *mutex*, quando é criado também é inicializado, ou seja, um

Figura 8 – Tempo de criação estática e exclusão das filas



item é adicionado na fila que representa-o, por isso seu tempo de criação é maior que os demais. A função de exclusão é a mesma para os 3 recursos aqui testados e executa internamente a função de exclusão das filas, tornando os tempos de exclusão praticamente iguais aos das mesmas.

Operação	**Recurso**	**Tempo médio (microssegundos)**				
		16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
Criação	*Mutex*	21,01	44,23	88,60	181,73	379,07
	Semáforo binário	12,30	26,74	55,28	108,99	220,80
	Semáforo contador	12,24	27,34	55,17	109,22	221,57
Exclusão	*Mutex*	2,60	7,56	16,99	37,98	84,99
	Semáforo binário	2,89	7,76	17,10	38,43	76,99
	Semáforo contador	2,36	7,62	17,25	37,41	74,24

Fonte: Elaborado pelo autor

Tabela 5 – Tempo de criação estática e exclusão do *mutex* e dos semáforos

4.1.3 Grupos de Eventos

Os resultados para a criação e exclusão dos grupos de eventos estão presentes na Tabela 6. Percebe-se também um aumento de tempo exponencial tanto na criação quanto na exclusão dos grupos de eventos. Na criação, o aumento foi de aproximadamente 2 vezes à medida que a frequência era diminuida, a maior diferença se deu entre as frequências de 16 e 8 MHz, com um aumento de quase 2,5 vezes. Já na exclusão, o tempo passou do dobro em todas as frequências, em relação com a anterior medida. Dentre os recursos testados nesse trabalho, os grupos de eventos foram os únicos que tiveram um tempo de exclusão maior que o tempo de criação, isso ocorre pois durante a exclusão, diferente das filas, o grupo de eventos desbloqueia todas as tarefas que estavam aguardando por algum *bit*.

Operação	Tempo médio (microssegundos)				
	16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
Criação	5,31	13,03	25,09	52,22	110,40
Exclusão	10,41	23,27	48,11	98,69	204,10

Fonte: Elaborado pelo autor

Tabela 6 – Tempo de criação estática e exclusão dos grupos de eventos

4.1.4 Tarefas

Os testes de criação e exclusão das tarefas se deram para pilhas iguais à 32, 64, 128, 256 e 512 bytes. Os resultados desse teste estão dispostos na Tabela 7. Eles mostraram que o tamanho da pilha influencia diretamente o tempo de criação das tarefas, pois apesar da memória da pilha já estar alocada, a função *memset* é chamada durante a criação e reescreve todos os bytes da memória da mesma com uma espécie de marca d'água do próprio FreeRTOS. Quanto a exclusão das tarefas, os testes mostraram que o tamanho da pilha não tem influência. O gráfico da Figura 9 mostra o crescimento do tempo em relação ao tamanho da pilha, para o processador operando à 16 MHz.

Operação	Tamanho da pilha	Tempo médio (microssegundos)				
		16 MHz	8 MHz	4 MHz	2 MHz	1 MHz
Criação	32 bytes	49,48	100,14	202,90	409,54	830,85
	64 bytes	61,38	125,86	255,02	512,38	1038,34
	128 bytes	86,34	174,09	350,78	704,77	1428,03
	256 bytes	134,45	271,10	544,03	1092,93	2210,37
	512 bytes	231,69	465,03	932,18	1873,38	3779,52
Exclusão	32 bytes	14,91	30,76	64,14	131,42	274,69
	64 bytes	14,84	31,14	63,39	130,75	267,97
	128 bytes	14,73	31,10	63,81	132,06	274,24
	256 bytes	14,63	30,48	64,32	131,52	274,18
	512 bytes	15,06	30,92	63,58	132,29	272,7

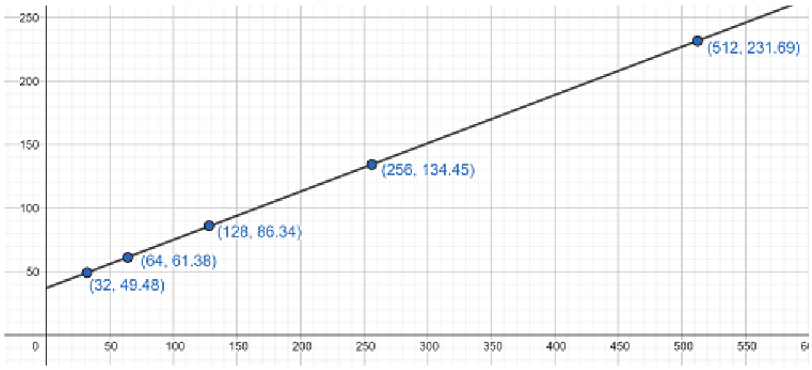
Fonte: Elaborado pelo autor

Tabela 7 – Tempo de criação estática e exclusão das tarefas

Para que possa-se melhor observar a exponencialidade no atraso da criação e exclusão das tarefas, a Figura 10 mostra esse atraso para tarefas com o tamanho da pilha igual à 32 *bytes*.

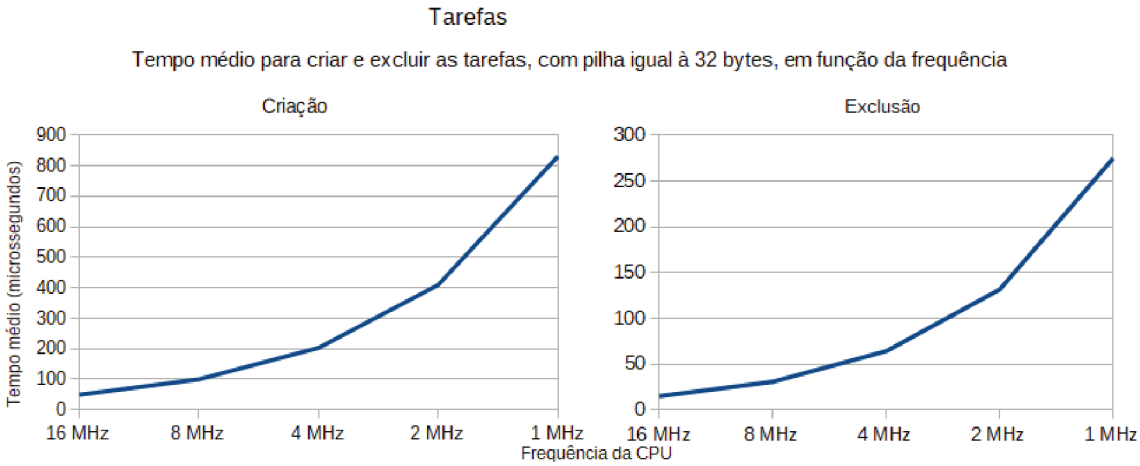
Fonte: Elaborado pelo autor no site do Geogebra

Figura 9 – Tempo de criação estática das filas, de acordo com o tamanho da pilha com o processador operando à 16 MHz



Fonte: Elaborado pelo autor usando o site do Geogebra

Figura 10 – Tempo de criação estática e exclusão das tarefas



4.2 MANIPULAÇÃO DOS RECURSOS

4.2.1 Comunicação entre tarefas

Para realizar a comunicação entre as tarefas foram usados filas, grupo de eventos e notificação de tarefas. Os resultados serão apresentados nas próximas subseções.

4.2.1.1 Filas

Foram testadas mensagens com tamanhos de 1, 2, 4, 8, 16, 32, 64 e 128 *bytes*. As mensagens enviadas pela tarefa primária e secundária sempre possuíam o mesmo tamanho. Os resultados podem ser vistos na Tabela 8. O aumento de tempo se deu de forma exponencial à medida que a frequência era diminuída.

Tamanho da mensagem	Tempo médio (microssegundos)				
	16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
1 byte	95,33	190,45	380,57	760,06	1460,70
2 bytes	96,38	192,21	381,02	765,09	1510,53
4 bytes	98,08	195,75	390,56	785,49	1587,97
8 bytes	102,11	204,61	407,98	809,20	1640,29
16 bytes	110,39	220,43	440,08	874,80	1746,72
32 bytes	126,38	252,21	504,19	1008,08	1996,26
64 bytes	158,52	316,58	635,95	1249,92	2495,68
128 bytes	223,06	445,46	891,33	1749,76	3494,08

Fonte: Elaborado pelo autor

Tabela 8 – Tempo de envio de uma mensagem entre tarefas a partir de uma fila

O tamanho das mensagens também teve influência crescente no tempo, uma vez que uma operação envolveu a cópia da mensagem duas vezes pelo SO, uma no envio, da tarefa para a pilha e uma na entrega, em sentido contrário.

4.2.1.2 Grupos de eventos

A Tabela 9 mostra o tempo de comunicação entre tarefas a partir de um grupo de eventos. A configuração padrão do FreeRTOS disponibiliza um grupo de eventos com 8 *bits*, ou seja, pode-se mandar uma mensagem de até 1 *byte* entre as tarefas, uma vez que é possível obter o valor do grupo de eventos. Podemos ver que o aumento do tempo desse recurso também se deu de forma exponencial, porém com uma média de entrega da mensagem 16% mais rápido do que as filas, se comparadas às mensagens de 1 *byte*.

Tempo médio (microssegundos)				
16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
79,30	158,65	318,03	629,81	1250,05

Fonte: Elaborado pelo autor

Tabela 9 – Tempo de envio de uma mensagem entre tarefas a partir de um grupo de eventos

4.2.1.3 Notificação de tarefas

Como já apresentado na subseção 3.2.1, as notificações de tarefas apresentam dois modelos, o modelo básico e o modelo configurável. A Tabela 10 contém os resultados dos testes das notificações de tarefas para comunicação entre sí, onde o aumento do tempo se deu de forma exponencial à medida que a frequência é diminuída, dobrando-o a cada vez que a frequência era diminuída pela metade. A notificação é o recurso mais rápido para fazer a comunicação entre

as tarefas. Para ambos os modelos, o SO utiliza a mesma função para enviar a notificação, onde o modelo básico utiliza o *eIncrement* como *eNotifyAction*. A única diferença real entre eles é a função de recebimento da notificação, por isso a pequena diferença no tempo. Já entre o modelo configurável, como era de se esperar, o modo *eNoAction* é o mais rápido, pois diferente dos demais, ele não altera o valor de notificação da tarefa.

MODELO BÁSICO					
	Tempo médio (microssegundos)				
	16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
	48,77	97,44	194,94	390,62	783,10
MODELO CONFIGURÁVEL					
eNotifyAction	**Tempo médio (microssegundos)**				
	16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
eNoAction	50,76	101,53	203,48	407,06	806,94
eSetBits	52,05	104,08	208,45	416,8	832,9
eIncrement	52,05	104,12	208,52	416,53	833,18
eSetValueWithoutOverwrite	51,47	102,92	206,05	410,5	831,74
eSetValueWithOverwrite	52,18	102,5	205,02	409,06	830,08

Fonte: Elaborado pelo autor

Tabela 10 – Tempo de envio de uma mensagem entre tarefas, usando notificações de tarefas

4.2.2 Comunicação entre ISR e tarefas

Para que uma ISR possa comunicar um evento à uma tarefa, foram testados os mesmos recursos da comunicação entre tarefas e também os semáforos binários e os semáforos contadores.

4.2.2.1 Filas

As filas se comportaram da mesma maneira do que na comunicação entre as tarefas, porém de forma mais rápida, como pode ser visto na Tabela 11 uma vez que as funções de inserção na fila utilizadas pelas ISR's não precisam se preocupar em desabilitar as interrupções e o escalonador, diferente das funções usadas pelas tarefas, que precisam ter esse cuidado para que nenhuma outra tarefa ou interrupção tente manipular a fila ao mesmo tempo.

A diferença de tempo para a entrega entre as tarefas e de uma ISR para a tarefa, não altera de acordo com o tamanho da mensagem, apenas de acordo com a frequência da CPU, levando aproximadamente 38, 75, 148, 284 e 545 microssegundos a menos, nas respectivas frequências de 16, 8, 4, 2 e 1 MHz, para a enviar uma mensagem da ISR para a tarefa.

Tamanho da mensagem	Tempo médio (microssegundos)				
	16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
1 byte	56,83	114,50	231,12	467,04	944,19
2 bytes	57,77	116,56	235,04	474,94	961,25
4 bytes	59,82	120,61	243,28	490,78	994,37
8 bytes	63,89	128,67	259,49	523,26	1055,62
16 bytes	71,92	144,82	291,6	588,48	1188,67
32 bytes	88,07	177,11	356,34	716,96	1448,77
64 bytes	120,3	241,64	485,31	976,32	1970,3
128 bytes	184,75	370,54	744,83	1494,08	3007,30

Fonte: Elaborado pelo autor

Tabela 11 – Tempo de envio de uma mensagem da ISR para a tarefa, usando filas

4.2.2.2 Grupos de eventos

Analisando a Tabela 12, podemos ver que o tempo para entregar uma mensagem de uma ISR para uma tarefa apresentou o mesmo crescimento observado na maioria dos outros recursos, dobrando a cada divisão da frequência da CPU. Apesar da função de definição dos *bits*, chamada pela ISR, utilizar internamente a mesma função que a tarefa, o atraso obtido na entrega da mensagem pela ISR foi aproximadamente o dobro do que o atraso de entrega da mensagem a partir das tarefas, apresentados na Tabela 9. Não foi identificado nenhum fator que levasse a esse aumento.

Tempo médio (microssegundos)				
16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
154,46	309,86	621,71	1250,85	2510,91

Fonte: Elaborado pelo autor

Tabela 12 – Tempo de envio de uma mensagem da ISR para a tarefa, usando grupo de eventos

4.2.2.3 Notificação de tarefas

Da mesma maneira que na comunicação entre tarefas, a notificação de tarefas também foi o recurso com o menor atraso para comunicação de uma ISR e uma tarefa, como apresentado na Tabela 13 Diferente das funções chamadas pelas tarefas, que internamete são as mesmas, nas funções chamadas pela ISR, as funções do modelo básico e do modelo configurável são implementadas separadamente. Apesar disso, os tempos de ambas as funções foram muito próximos. Assim como nas filas, a transferência das mensagens entre a ISR e a tarefa foi mais rápido do que a transferência entre as tarefas, pois nas funções chamadas pela ISR não há necessidade de controlar a entrada na região crítica.

MODELO BÁSICO					
	Tempo médio (microssegundos)				
	16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
	39,60	80,11	162,10	328,45	667,39
MODELO CONFIGURÁVEL					
eNotifyAction	**Tempo médio (microssegundos)**				
	16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
eNoAction	39,10	78,98	160,14	323,36	657,60
eSetBits	40,23	81,31	164,83	333,57	677,5
eIncrement	40,33	81,81	164,93	334,5	680,32
eSetValueWithoutOverwrite	39,99	80,86	163,78	331,39	672,83
eSetValueWithOverwrite	39,86	80,58	162,98	330,53	671,42

Fonte: Elaborado pelo autor

Tabela 13 – Tempo para ISR enviar uma notificação de tarefas

4.2.2.4 Semáforos

Barry (2016, p. 191–211) sugere o uso de semáforos binários e contadores para sincronização da ISR com a tarefa por isso, foram testados os tempos para os semáforos binários e contadores notificarem a tarefa que o evento ocorreu. O resultado está disposto na Tabela 14. Como a função para aquisição e entrega do semáforo é a mesma para ambos os modelos, era de se esperar que o atraso para eles também fosse o mesmo.

Recurso	**Tempo médio (microssegundos)**				
	16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
Semáforos binários	44,97	90,84	183,76	371,49	752,64
Semáforos contadores	44,98	90,84	183,60	371,49	752,58

Fonte: Elaborado pelo autor

Tabela 14 – Tempo para ISR sinalizar à tarefa, usando semáforos

4.2.3 Sincronização de tarefas

Os grupos de eventos fornecem uma função que permite realizar a sincronização das tarefas. Essa função foi testada usando 2, 3, 4, 5, 6 e 7 tarefas que desejavam se sincronizar. Todas as tarefas possuem a mesma prioridade e receberam como parâmetro o *bit* que a representa no grupo de eventos. O atraso para a sincronização das tarefas teve um crescimento exponencial de acordo com a frequência, como é possível observar na Tabela 15. A quantidade de tarefas também proporcionou um aumento significativo no tempo de sincronização. O gráfico da Figura 4 mostra esse tempo para a CPU operando à uma frequência de 16 MHz onde, para cada

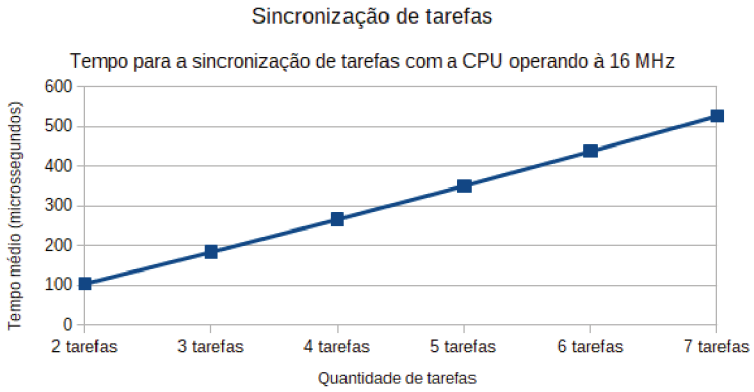
tarefa adicionada, um aumento de um pouco mais de 80 microssegundos foi fornecido, sendo que esse tempo adicional também aumenta de acordo com a quantidade de tarefas.

Quantidade de tarefas	**Tempo médio (microssegundos)**				
	16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
2	103,52	207,28	416,22	833,36	1688,26
3	183,53	367,49	736,94	1480,64	2988,84
4	265,85	532,42	1067,31	2146,42	4359,59
5	350,39	702,09	1406,99	2828,42	5730,07
6	437,25	875,64	1754,93	3526,57	7127,36
7	526,34	1054,20	2113,29	4249,00	8610,22

Fonte: Elaborado pelo autor

Tabela 15 – Tempo para sincronização de tarefas, usando grupo de eventos

Figura 11 – Tempo para sincronização de tarefas



4.2.4 Aquisição e entrega de *mutex* e semáforos

O último teste de manipulação dos recursos consistiu em verificar o atraso das funções de aquisição e entrega do *mutex*, do semáforo binário e do semáforo contador. Os resultados da Tabela 16 mostram o aumento exponencial do tempo, de acordo com a frequência da CPU. Os 3 recursos aqui testados utilizam a mesma função para aquisição e entrega porém, podemos verificar que o atraso do *mutex* é levemente maior do que o atraso dos semáforos, onde esse último não sofreu alteração de acordo com o seu tipo.

4.3 TROCA DE CONTEXTO

O tempo consumido pela troca de contexto também teve crescimento exponencial à medida que a frequência é reduzida. A Tabela 17 mostra os resultados desse atraso. É importante salientar que independente da frequência da CPU, um *tick* ocorre a cada 16 ms no

Operação	Recurso	Tempo médio (microssegundos)				
		16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
Aquisição	Mutex	7,71	16,32	34,45	78,50	164,35
	Semáforos binários	6,25	12,66	27,04	60,38	128,64
	Semáforos contadores	6,17	12,70	26,83	60,19	128,90
Entrega	Mutex	12,43	26,61	53,34	110,18	230,85
	Semáforos binários	10,61	22,88	48,06	98,18	212,03
	Semáforos contadores	10,68	23,15	48,06	98,75	208,96

Fonte: Elaborado pelo autor

Tabela 16 – Tempo para aquisição e entrega do *mutex* e dos semáforos

FreeRTOS então, em um sistema operando à 1 MHz, 1,63% do tempo de execução das tarefas é gasto com troca de contexto. Já com o sistema operando à 16 MHz esse tempo se torna praticamente insignificante, com o consumo de apenas 0,1% do tempo.

Tempo médio (microssegundos)				
16 MHz	**8 MHz**	**4 MHz**	**2 MHz**	**1 MHz**
15,92	31,99	63,99	128,56	260,80

Fonte: Elaborado pelo autor

Tabela 17 – Tempo para troca de contexto

4.4 TEMPORIZADORES DE *SOFTWARE*

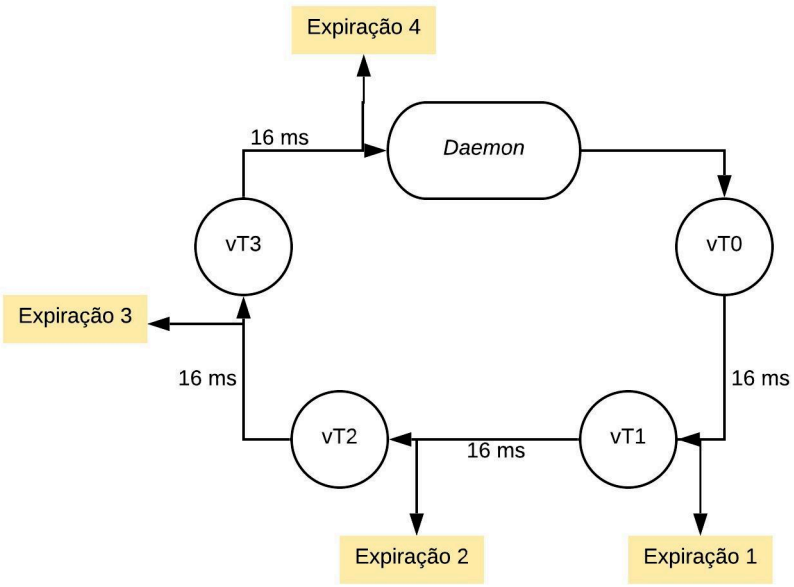
O Apêndice A traz o resultado dos testes dos temporizadores, onde os mesmos foram testados com tempo de 16, 32, 48, 64, ..., 400 ms e com 0, 1, 2, 4 e 8 tarefas do tipo *vTarefaX*, que tem como objetivo disputar a CPU com a tarefa *Daemom*, responsável pela execução dos temporizadores. Como os tempos dos temporizadores são definidos em *ticks* e cada *tick* equivale a 16 ms, independente da frequência da CPU, só é possível obter temporizadores com tempos múltiplos de 16 ms. Para qualquer tempo que não seja múltiplo de 16 ms será executado um temporizador para o múltiplo logo abaixo desse tempo, por exemplo: para o tempo igual à 18, 25 ou 31 ms, o temporizador irá executar com o tempo de 16 ms.

Os temporizadores mostraram um atraso em sua execução, que é dependente da frequência da CPU e da quantidade de tarefas. Para a CPU operando à 16 MHz e sem *vTarefaX*, os temporizadores apresentaram um atraso de aproximadamente 9,9%, diminuindo à medida que a frequência também era diminuída, chegando a um atraso de 8,75% à frequência de 1 MHz, apresentado uma diferença de atraso de aproximadamente 1,15% entre essas frequências. Essa diferença não foi notada quando existia *vTarefaX* rodando que, independente da frequência e da quantidade de tarefas rodando, o atraso aproximou-se sempre de 9%. Um fator importante de se observar é que o desvio padrão, apesar sofrer um pequeno aumento à medida que a frequência

é diminuída, foi muito baixo, o que revela que o atraso não sofreu grandes variações entre as execuções, exceto em alguns casos especiais que serão comentados a seguir.

Os temporizadores de *software* apresentaram um desvio padrão elevado para a execução com 2 *vTarefaX* e com tempo de 16 ms, 4 *vTarefaX* e com tempos de 16, 32 e 48 ms e com 8 *vTarefaX* e tempo iguais à 16, 32, 48, 64, 80, 96 e 112 ms. Isso se deu devido à tarefa *Daemon*, responsável pela execução dos temporizadores não ter conseguido a CPU a tempo. A Figura 12 mostra um esquema que representa a execução das tarefas na CPU (não necessáriamente nessa ordem), onde cada *vTi* representa uma tarefa *vTarefaX*.

Figura 12 – Esquema de execução das tarefas



Como cada *tick* do sistema ocorre a cada 16 ms e a troca de contexto ocorre a cada *tick*, a tarefa *Daemon* vai ser executada a cada 64 ms aproximadamente, nesse esquema com 4 *vTarefaX*. Então, em 64 ms, um temporizador de 16 ms foi expirado 4 vezes, por exemplo, fazendo com que, quando a tarefa *Daemon* assume a CPU, o temporizador seja executado 4 vezes seguidas, ou até a tarefa perder a CPU, aumentando o desvio padrão, uma vez que o tempo entre essas 4 execuções do temporizador é extremamente baixo.

Por algum motivo desconhecido, o sistema mostrou-se incapaz de executar os temporizadores nos seguintes casos:

- Tempo igual à 16 ms, com o processador operando a 1 MHz e com 4 *vTarefaX* em execução;
- Tempo igual à 16 ms, com o processador operando a 2 MHz e com 8 *vTarefaX* em execução;
- Tempos iguais à 16, 32 e 48 ms, com o processador operando a 1 MHz e com 8 *vTarefaX* em execução.

5 CONCLUSÃO

Com o estudo do FreeRTOS realizado nesse trabalho foi possível realizar medições de tempo de seus principais recursos e entender melhor seu funcionamento. Apesar da alocação dinâmica não ter executado dentro das tarefas, o objetivo proposto para medir como as mudanças do microcontrolador afetam a criação dos recursos foi parcialmente atingido através da medição do tempo de criação estática dos recursos. Já a verificação do atraso na manipulação dos recursos e da troca de contexto foram realizadas nos moldes propostos. A partir desses testes podemos concluir que, de modo geral, o aumento dos atrasos do SO foi exponencial onde, a cada vez que a frequência da CPU é dividida pela metade, o tempo é no mínimo dobrado, já que a CPU executa a metade das instruções no mesmo período de tempo, mostrando que o SO se comportou da maneira esperada.

As tarefas foram o recurso com o maior atraso de criação e exclusão. Dentre os demais recursos, o *mutex* foi o recurso com o maior atraso de criação e o grupo de eventos com o maior atraso de exclusão. Já na comunicação entre tarefas, a fila é o recurso mais demorado para entregar uma mensagem, em contraposição com a notificação de tarefas, que foi o recurso mais rápido. Para a comunicação da ISR com uma tarefa, o grupo de eventos foi o recurso com o maior atraso, mantendo a continuidade da notificação de tarefas como o recurso mais eficiente.

Os temporizadores também foram devidamente analisados e verificou-se que eles são executados a cada *tick*, limitando a precisão dos mesmos em 16 ms, independente da frequência. Se o tempo do temporizador, em *ticks* for maior ou igual ao número de tarefas que tem a mesma prioridade da *Daemon*, nenhuma anormalidade foi identificada. Abaixo disso, algumas frequências podem coibir ou tornar irregular a execução dos temporizadores. As tabelas do Apêndice A podem servir de apoio para o uso dos temporizadores em futuros projetos.

Por fim, pode-se concluir que caso o propósito do sistema não seja prejudicado pela diminuição da frequência da CPU, a economia de energia apresentada na Tabela 3 é extremamente vantajosa para frequência igual à 4 MHz pois, apesar de o atraso do sistema for, de modo geral, 4 vezes maior em relação ao atraso na frequência de 16 MHz, o consumo de energia é aproximadamente 6 vezes menor. Já para o processador operando em 8 MHz, a economia não é tão significativa, pois o atraso é o dobro do que o atraso com o processador operando à 16 MHz, mas a economia de energia, apesar de próxima, é menor do que a metade.

Espera-se que esse trabalho possa servir como base para desenvolvedores de sistemas que operam sobre o FreeRTOS ou outros SO's de tempo real que tenham características similares a este.

5.1 TRABALHOS FUTUROS

No anseio de que esse trabalho sirva como inspiração para trabalhos futuros, deixa-se nesta subseção algumas sugestões. Como a criação dinâmica não foi possível nesse *port* e o

FreeRTOS é de código aberto, torna-se possível o estudo dessa limitação e também a sugestão de alguma correção para a mesma, caso houver. Em relação à criação estática, pode-se estudar se a exclusão dos recursos libera a memória para que ela possa ser alocada por outros recursos. Outro ponto relevante, é que o FreeRTOS é um sistema que possui muitas funções e nem todas foram testadas nesse trabalho, por isso, para que esse trabalho possa ser considerado completo e para que os desenvolvedores possam obter conhecimento de tempo de todo o sistema, seria de extrema valia que essas funções fossem testadas e tivessem seu tempo medido. Além do mais, esse trabalho pode ser estendido para outros SO's, de tempo real ou não, assim como também, para outros microcontroladores e arquiteturas.

REFERÊNCIAS

AG, Arduino. **ARDUINO UNO REV3**. 2019. Disponível em: <https://store.arduino.cc/arduino-uno-rev3>. Acesso em: 3 jun. 2019.

ATMEL CORPORATION. **ATmega328P**. San Jose, CA 95110 USA, 2015. Rev.: 7810D–AVR–01/15.

BARRY, Richard. **Mastering the FreeRTOS Real Time Kernel: A Hands-On Tutorial Guide**. [S.l.]: Real Time Engineers Ltd, 2016.

CHABATURA NETO, Felipe. O sistema operacional em tempo real Freertos na plataforma de hardware Arduino Uno: uma análise dos recursos de gerenciamento de memória, tarefas, filas e do desempenho geral do sistema, 2018. Disponível em: <https://rd.uffs.edu.br/handle/prefix/2709>. Acesso em: 20 abr. 2019.

CHRISTOFFERSON, Michael. ways to improve performance in embedded Linux systems. In: KOREA Linux Forum. [S.l.: s.n.], 2013.

FASSINI, Leonardo Tironi. Sistema operacional em tempo real Freertos: análise dos componentes de timers, interrupções e gerenciamento de recursos na plataforma de hardware minimalista Arduino Uno, 2018. Disponível em: <https://rd.uffs.edu.br/handle/prefix/2708>. Acesso em: 20 abr. 2019.

LIBC, AVR. **Power Reduction Management**. 2018. Disponível em: <https://www.nongnu.org/avr-libc/user-manual/group__avr__power.html>. Acesso em: 9 out. 2019.

MAZIERO, Carlos A. Sistemas operacionais: Conceitos e mecanismos. **Livro aberto. Acessível em: http://www.lcvdata.com/ads/so-livro.pdf**, 2017.

RAMMIG, Franz et al. Basic concepts of real time operating systems. In: HARDWAREDEPENDENT Software. [S.l.]: Springer, 2009. p. 15–45.

SACHA, Krzysztof M. Measuring the real-time operating system performance. In: IEEE. REAL-TIME Systems, 1995. Proceedings., Seventh Euromicro Workshop on. [S.l.: s.n.], 1995. p. 34–40.

SANTOS, César Augusto Marcelino dos. Sistema Dinâmico de Economia de Energia em RTOS, 2017. Disponível em: <http://repositorio.unifei.edu.br/xmlui/handle/123456789/699>. Acesso em: 15 mai. 2019.

SIMONOVIĆ, Mirela; SARANOVAC, Lazar. Power management implementation in FreeRTOS on LM3S3748. **Serbian Journal of Electrical Engineering**, v. 10, n. 1, p. 199–208, 2013. Disponível em: <http://dx.doi.org/10.2298/sjee1301199s>. Acesso em: 29 abr. 2019.

STEVENS, Phillip. **Arduino_FreeRTOS_Library**. 2019. Disponível em: <https://github.com/feilipu/Arduino_FreeRTOS_Library>. Acesso em: 15 mai. 2019.

TANENBAUM, Andrew S; BOS, Herbert. **Sistemas operacionais modernos**. 4. ed. São Paulo: Pearson Education do Brasil, 2016. tradução Jorge Ritter; revisão técnica Raphael Y. de Camargo.

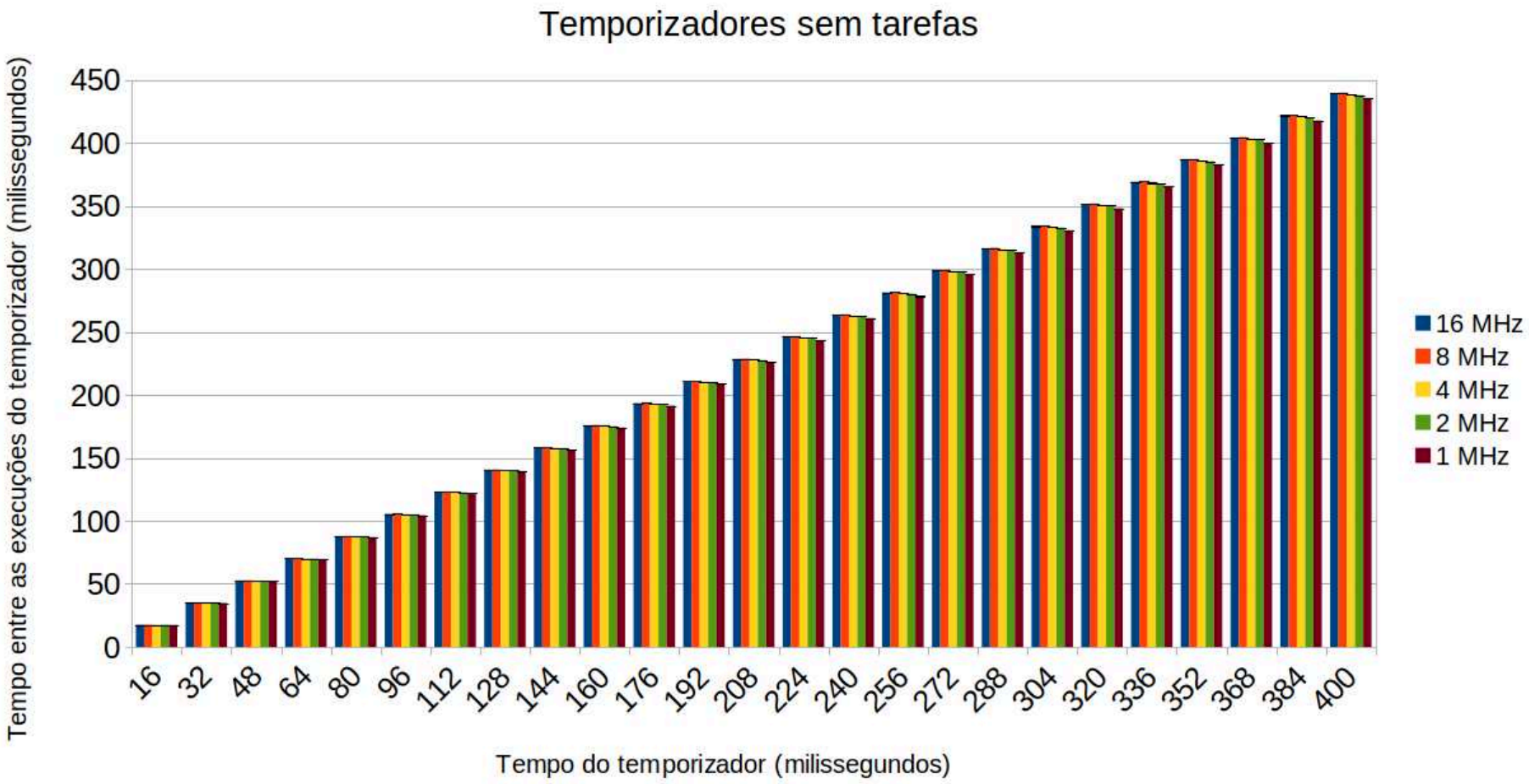
TOMAZI, Rodrigo Miguel. **ANÁLISE DO IMPACTO CAUSADO NO FreeRTOS DEVIDO A ALTERAÇÃO DO CLOCK DO MICROCONTROLADOR ATmega328P NO ARDUINO UNO**. [S.l.]: Zenodo, nov. 2019. Orientador: Prof. Dr. Marco Aurélio Spohn. DOI: 10.5281/zenodo.3575216. Disponível em: <https://doi.org/10.5281/zenodo.3575216>.

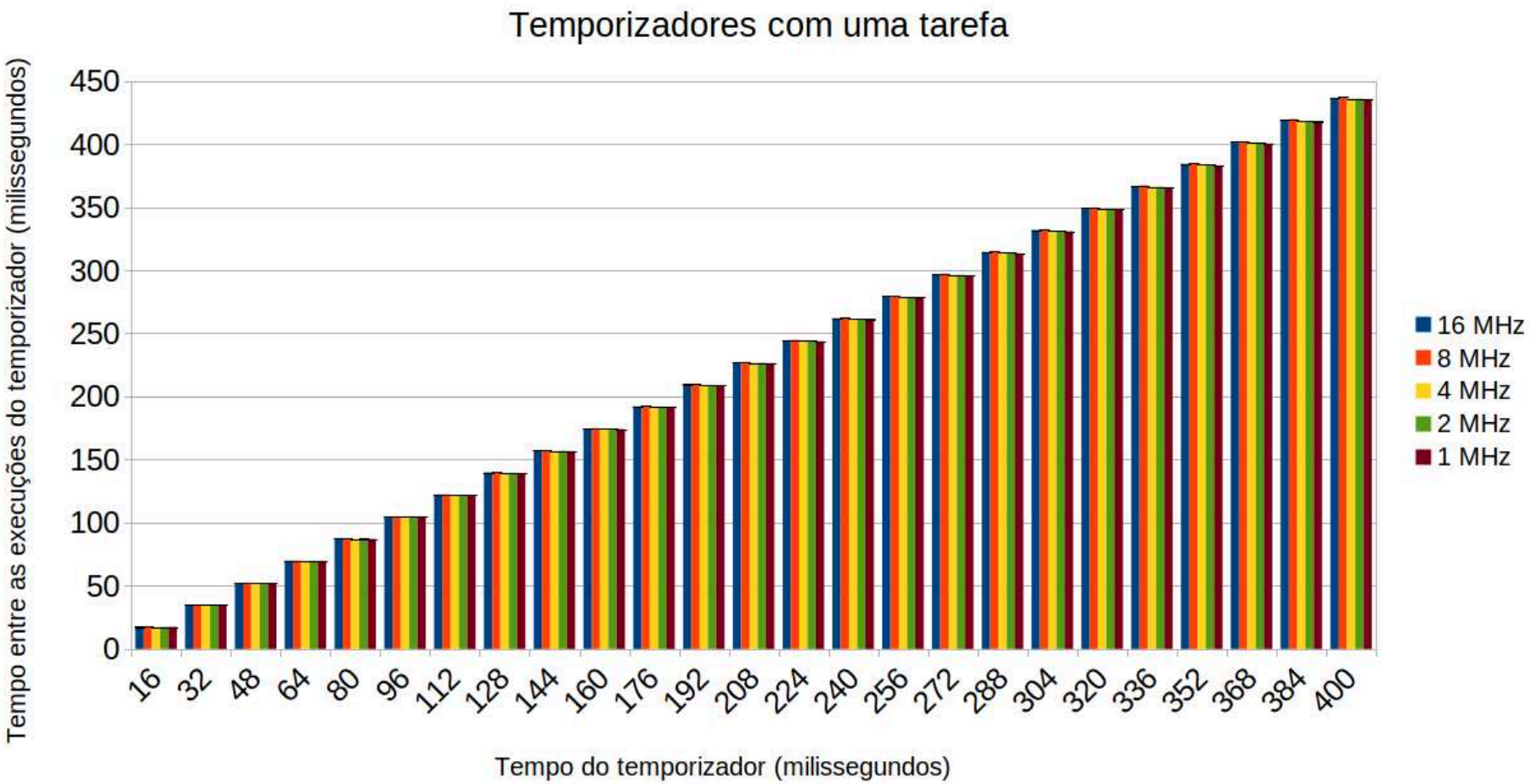
URRIZA, José M et al. Economia de energia em dispositivos móveis. In: VI Workshop de Comunicação sem Fio e Computação Móvel. [S.l.: s.n.], 2004. p. 48–56.

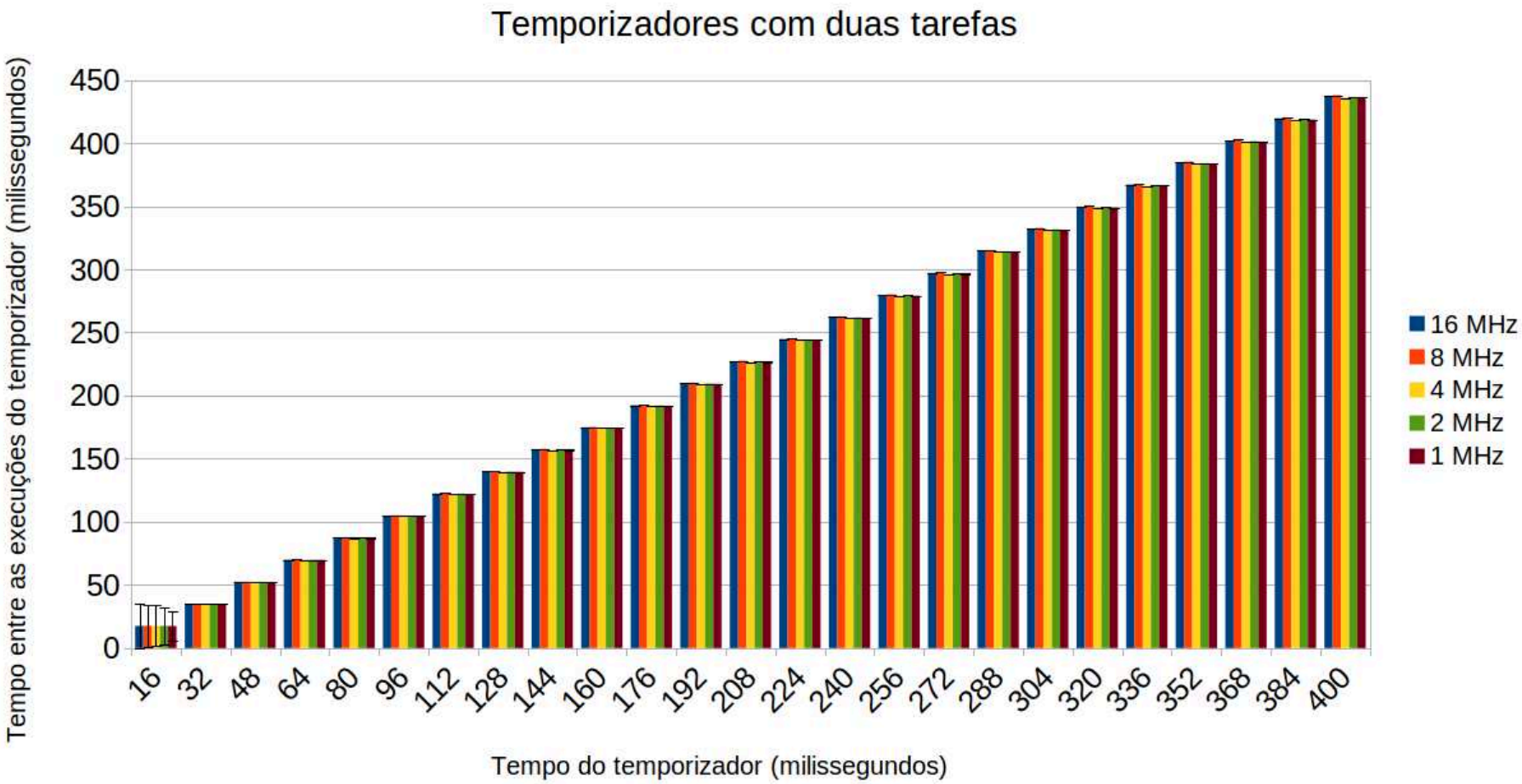
APÊNDICE A – RESULTADO DOS TEMPORIZADORES DE *SOFTWARE*

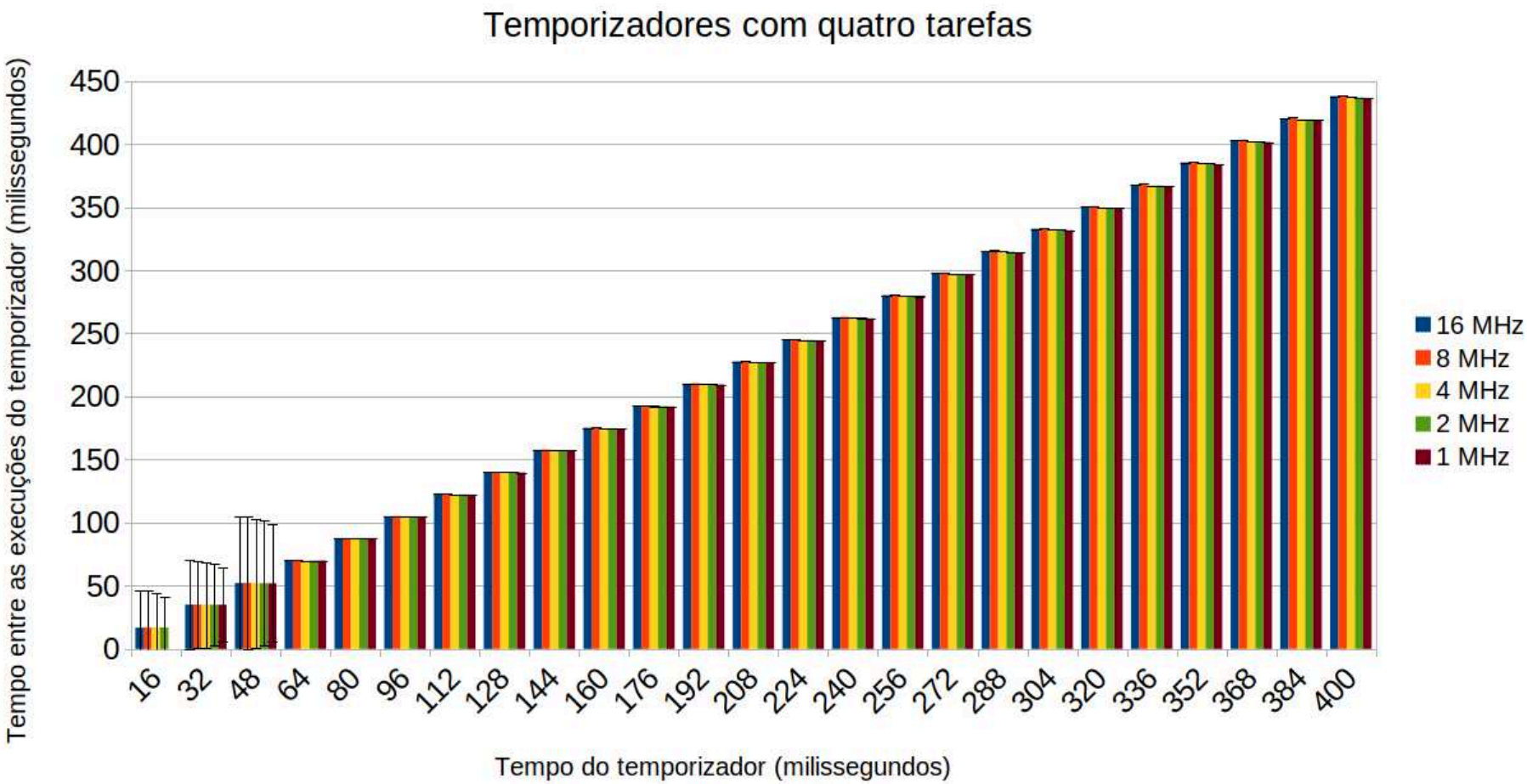
Esse apêndice contém os resultados dos testes dos temporizadores, onde os valores de tempo estão expressos em milissegundos. Na seção A.1, os resultados estão expressos em forma de gráfico, para uma rápida visualização e entendimento do comportamento dos temporizadores. Os gráficos trazem o tempo médio para a execução dos temporizadores e seu desvio padrão. A seção A.2 trás os resultados em forma de tabelas, que podem ser consultados para obter uma maior precisão dos tempos.

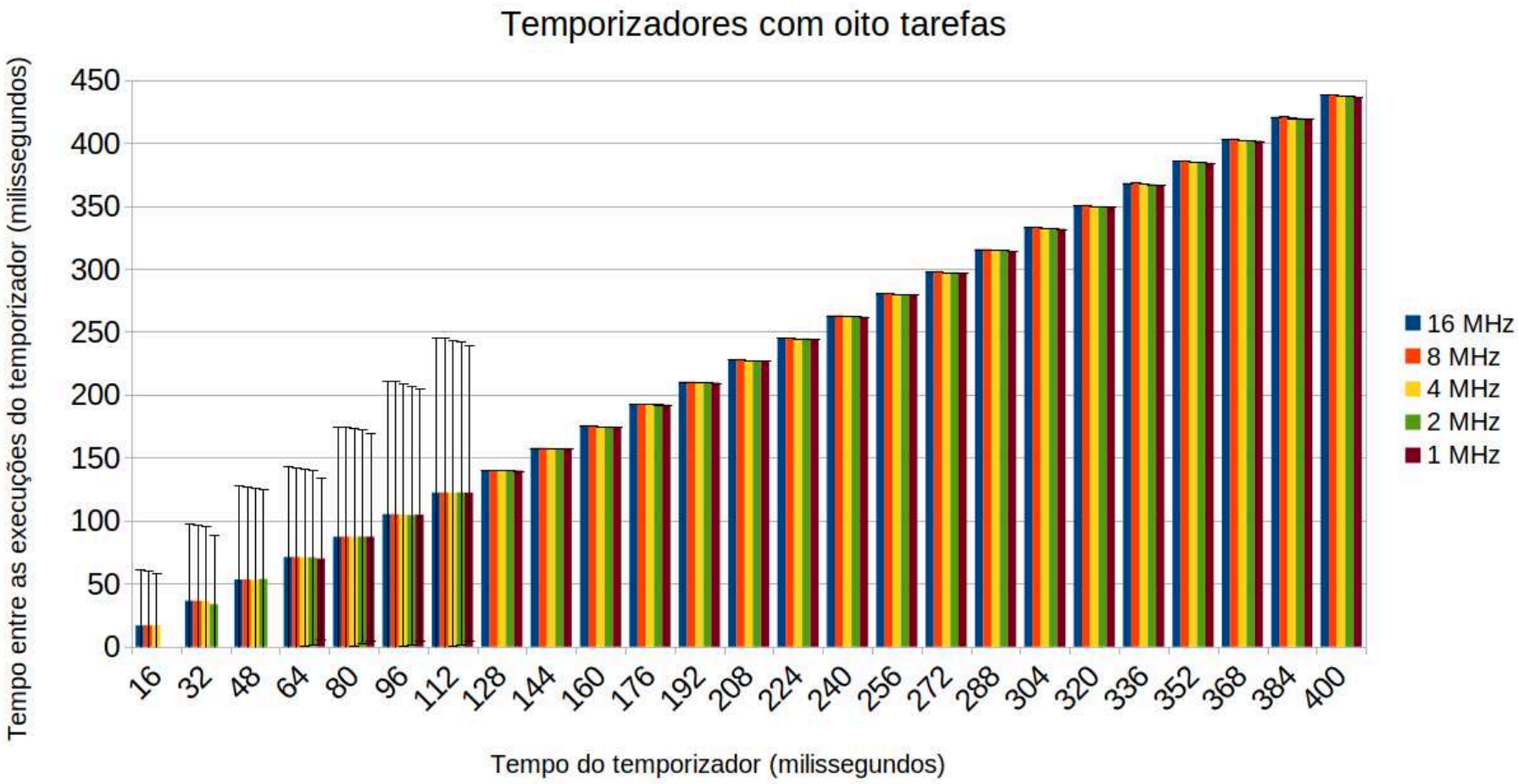
A.1 GRÁFICOS


Temporizadores sem tarefas
Tempo entre as execuções do temporizador (milissegundos)
450
400
350
300
250
200
150
100
50
0
16
32
48
64
80
96
112
128
144
160
176
192
208
224
240
256
272
288
304
320
336
352
368
384
400
Tempo do temporizador (milissegundos)
16 MHz
8 MHz
4 MHz
2 MHz
1 MHz


Temporizadores com uma tarefa
Tempo entre as execuções do temporizador (milissegundos)
450
400
350
300
250
200
150
100
50
0
16
32
48
64
80
96
112
128
144
160
176
192
208
224
240
256
272
288
304
320
336
352
368
384
400
Tempo do temporizador (milissegundos)
16 MHz
8 MHz
4 MHz
2 MHz
1 MHz


Temporizadores com duas tarefas
Tempo entre as execuções do temporizador (milissegundos)
450
400
350
300
250
200
150
100
50
0
16
32
48
64
80
96
112
128
144
160
176
192
208
224
240
256
272
288
304
320
336
352
368
384
400
Tempo do temporizador (milissegundos)
16 MHz
8 MHz
4 MHz
2 MHz
1 MHz


Temporizadores com quatro tarefas
Tempo entre as execuções do temporizador (milissegundos)
450
400
350
300
250
200
150
100
50
0
16
32
48
64
80
96
112
128
144
160
176
192
208
224
240
256
272
288
304
320
336
352
368
384
400
Tempo do temporizador (milissegundos)
16 MHz
8 MHz
4 MHz
2 MHz
1 MHz


Temporizadores com oito tarefas
Tempo entre as execuções do temporizador (milissegundos)
450
400
350
300
250
200
150
100
50
0
16
32
48
64
80
96
112
128
144
160
176
192
208
224
240
256
272
288
304
320
336
352
368
384
400
Tempo do temporizador (milissegundos)
16 MHz
8 MHz
4 MHz
2 MHz
1 MHz

A.2 TABELAS

SEM TAREFAS										
Tempo (ms)	**16 MHz**		**8 MHz**		**4 MHz**		**2 MHz**		**1 MHz**	
	Tempo Médio	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**
16	17,580	0,002	17,607	0,005	17,544	0,012	17,521	0,024	17,398	0,044
32	35,161	0,003	35,210	0,006	35,082	0,013	35,035	0,018	34,803	0,062
48	52,740	0,003	52,811	0,006	52,618	0,012	52,548	0,024	52,197	0,045
64	70,317	0,002	70,410	0,005	70,152	0,010	70,060	0,021	69,595	0,032
80	87,896	0,003	88,006	0,005	87,686	0,013	87,571	0,019	86,994	0,048
96	105,469	0,005	105,602	0,005	105,222	0,012	105,078	0,020	104,390	0,047
112	123,042	0,004	123,199	0,006	122,759	0,010	122,588	0,019	121,789	0,047
128	140,622	0,003	140,794	0,007	140,296	0,010	140,100	0,021	139,187	0,026
144	158,201	0,003	158,391	0,007	157,830	0,011	157,609	0,021	156,586	0,044
160	175,775	0,004	175,986	0,006	175,367	0,014	175,115	0,017	173,985	0,041
176	193,357	0,004	193,583	0,006	192,901	0,012	192,623	0,028	191,380	0,051
192	210,941	0,004	211,176	0,007	210,440	0,011	210,132	0,022	208,781	0,037
208	228,503	0,009	228,768	0,006	227,972	0,011	227,642	0,026	226,186	0,042
224	246,070	0,006	246,362	0,005	245,508	0,012	245,149	0,021	243,585	0,033
240	263,642	0,006	263,956	0,005	263,043	0,012	262,663	0,021	260,984	0,038
256	281,211	0,008	281,549	0,010	280,579	0,007	280,169	0,032	278,388	0,025
272	298,779	0,006	299,120	0,013	298,115	0,012	297,683	0,026	295,785	0,042
288	316,355	0,004	316,681	0,012	315,655	0,011	315,194	0,021	313,188	0,047
304	333,943	0,008	334,248	0,009	333,198	0,013	332,716	0,026	330,580	0,044
320	351,514	0,011	351,845	0,011	350,748	0,012	350,234	0,014	347,978	0,040
336	369,070	0,006	369,418	0,007	368,285	0,014	367,741	0,018	365,384	0,028
352	386,634	0,007	387,022	0,008	385,794	0,011	385,245	0,020	382,793	0,050
368	404,213	0,006	404,631	0,006	403,342	0,013	402,767	0,016	400,196	0,042
384	421,790	0,006	422,235	0,011	420,868	0,019	420,284	0,026	417,597	0,050
400	439,354	0,015	439,845	0,008	438,415	0,013	437,793	0,025	434,988	0,040

Fonte: Elaborado pelo autor

UMA TAREFA										
Tempo (ms)	**16 MHz**		**8 MHz**		**4 MHz**		**2 MHz**		**1 MHz**	
	Tempo Médio	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**
16	17,459	0,003	17,494	0,005	17,435	0,010	17,434	0,023	17,409	0,040
32	34,918	0,003	34,982	0,006	34,868	0,012	34,867	0,022	34,819	0,013
48	52,378	0,003	52,467	0,006	52,302	0,005	52,300	0,024	52,221	0,045
64	69,838	0,002	69,951	0,006	69,736	0,010	69,730	0,015	69,622	0,075
80	87,300	0,002	87,437	0,006	87,169	0,011	87,160	0,023	87,030	0,035
96	104,763	0,004	104,924	0,006	104,603	0,011	104,590	0,024	104,434	0,052
112	122,226	0,004	122,412	0,008	122,036	0,011	122,017	0,022	121,837	0,043
128	139,685	0,005	139,892	0,007	139,470	0,009	139,440	0,017	139,238	0,032
144	157,147	0,004	157,375	0,006	156,903	0,011	156,866	0,022	156,643	0,041
160	174,614	0,005	174,856	0,009	174,337	0,010	174,292	0,021	174,051	0,041
176	192,083	0,004	192,343	0,006	191,770	0,009	191,725	0,022	191,457	0,043
192	209,545	0,005	209,837	0,007	209,204	0,012	209,155	0,023	208,865	0,032
208	227,015	0,004	227,327	0,004	226,638	0,005	226,588	0,022	226,263	0,042
224	244,467	0,004	244,810	0,007	244,071	0,013	244,015	0,023	243,656	0,038
240	261,929	0,005	262,284	0,009	261,505	0,010	261,439	0,023	261,070	0,054
256	279,400	0,005	279,766	0,007	278,939	0,008	278,867	0,020	278,467	0,026
272	296,875	0,014	297,247	0,007	296,372	0,012	296,301	0,019	295,872	0,045
288	314,354	0,005	314,726	0,006	313,806	0,009	313,727	0,022	313,284	0,037
304	331,800	0,015	332,210	0,007	331,240	0,012	331,164	0,023	330,687	0,035
320	349,228	0,009	349,690	0,005	348,674	0,013	348,585	0,023	348,097	0,038
336	366,699	0,005	367,173	0,006	366,107	0,012	366,020	0,022	365,508	0,044
352	384,180	0,009	384,665	0,007	383,541	0,012	383,450	0,026	382,922	0,044
368	401,659	0,007	402,162	0,008	400,974	0,012	400,904	0,024	400,351	0,045
384	419,121	0,006	419,652	0,008	418,407	0,010	418,343	0,026	417,756	0,032
400	436,579	0,007	437,138	0,007	435,842	0,010	435,777	0,019	435,165	0,045

Fonte: Elaborado pelo autor

DUAS TAREFAS										
Tempo (ms)	**16 MHz**		**8 MHz**		**4 MHz**		**2 MHz**		**1 MHz**	
	Tempo Médio	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**
16	17,473	276,049	17,516	16,969	17,450	16,172	17,466	14,715	17,448	11,643
32	34,946	0,003	35,029	0,005	34,882	0,008	34,926	0,018	34,867	0,168
48	52,420	0,003	52,539	0,005	52,314	0,010	52,381	0,022	52,330	0,044
64	69,897	0,003	70,047	0,007	69,748	0,008	69,839	0,022	69,769	0,029
80	87,374	0,003	87,556	0,004	87,183	0,009	87,295	0,021	87,206	0,045
96	104,852	0,003	105,064	0,007	104,616	0,010	104,760	0,021	104,641	0,027
112	122,333	0,004	122,572	0,007	122,049	0,010	122,223	0,021	122,079	0,045
128	139,814	0,004	140,082	0,004	139,484	0,010	139,675	0,020	139,520	0,053
144	157,300	0,003	157,592	0,005	156,918	0,010	157,130	0,024	156,961	0,037
160	174,788	0,004	175,098	0,007	174,353	0,007	174,584	0,026	174,412	0,040
176	192,274	0,003	192,616	0,007	191,787	0,013	192,041	0,022	191,862	0,044
192	209,754	0,004	210,136	0,008	209,221	0,010	209,498	0,020	209,303	0,042
208	227,238	0,004	227,647	0,005	226,656	0,005	226,959	0,023	226,737	0,048
224	244,723	0,003	245,164	0,006	244,091	0,012	244,415	0,025	244,187	0,041
240	262,206	0,004	262,678	0,006	261,525	0,010	261,866	0,021	261,626	0,032
256	279,690	0,007	280,197	0,006	278,959	0,011	279,322	0,020	279,058	0,039
272	297,180	0,005	297,714	0,007	296,394	0,011	296,769	0,023	296,493	0,045
288	314,671	0,005	315,239	0,008	313,830	0,012	314,226	0,025	313,938	0,050
304	332,151	0,007	332,760	0,008	331,265	0,010	331,693	0,021	331,388	0,042
320	349,651	0,008	350,283	0,006	348,700	0,008	349,164	0,022	348,826	0,045
336	367,135	0,005	367,799	0,006	366,138	0,011	366,635	0,026	366,269	0,036
352	384,622	0,008	385,328	0,007	383,572	0,011	384,093	0,022	383,717	0,045
368	402,116	0,006	402,853	0,009	401,006	0,011	401,549	0,019	401,158	0,037
384	419,609	0,006	420,384	0,007	418,440	0,012	419,020	0,022	418,601	0,048
400	437,106	0,005	437,899	0,008	435,875	0,013	436,500	0,018	436,024	0,048

Fonte: Elaborado pelo autor

QUATRO TAREFAS										
Tempo (ms)	**16 MHz**		**8 MHz**		**4 MHz**		**2 MHz**		**1 MHz**	
	Tempo Médio	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**
16	16,821	29,561	16,873	28,956	16,855	27,512	16,909	24,695	-	-
32	35,012	35,000	35,085	34,704	34,981	33,860	34,961	32,364	34,925	29,236
48	52,164	52,384	52,264	52,117	52,091	51,204	52,044	49,691	51,941	46,499
64	70,028	0,003	70,161	0,005	69,942	0,010	69,910	0,015	69,910	0,042
80	87,538	0,003	87,700	0,006	87,425	0,011	87,382	0,041	87,282	0,039
96	105,050	0,003	105,235	0,007	104,909	0,012	104,856	0,014	104,735	0,045
112	122,564	0,004	122,773	0,005	122,390	0,012	122,332	0,017	122,186	0,037
128	140,079	0,004	140,309	0,006	139,872	0,010	139,800	0,023	139,639	0,034
144	157,596	0,004	157,846	0,006	157,350	0,013	157,274	0,012	157,091	0,045
160	175,110	0,004	175,382	0,006	174,835	0,010	174,748	0,014	174,543	0,044
176	192,629	0,004	192,919	0,007	192,316	0,012	192,222	0,020	191,997	0,034
192	210,143	0,004	210,458	0,006	209,801	0,009	209,693	0,023	209,452	0,044
208	227,664	0,005	227,996	0,006	227,282	0,011	227,165	0,022	226,902	0,040
224	245,183	0,005	245,529	0,007	244,767	0,013	244,639	0,022	244,357	0,036
240	262,696	0,005	263,065	0,006	262,256	0,011	262,115	0,012	261,814	0,030
256	280,214	0,004	280,603	0,005	279,735	0,011	279,595	0,024	279,269	0,041
272	297,726	0,005	298,136	0,007	297,222	0,013	297,069	0,018	296,722	0,039
288	315,241	0,007	315,675	0,005	314,708	0,011	314,540	0,024	314,177	0,027
304	332,768	0,007	333,215	0,009	332,194	0,016	332,011	0,023	331,631	0,034
320	350,272	0,009	350,755	0,011	349,676	0,011	349,491	0,023	349,092	0,045
336	367,804	0,004	368,284	0,007	367,153	0,010	366,957	0,022	366,551	0,040
352	385,318	0,006	385,827	0,006	384,639	0,013	384,435	0,022	384,003	0,043
368	402,835	0,006	403,369	0,007	402,122	0,016	401,924	0,017	401,457	0,064
384	420,357	0,007	420,924	0,008	419,607	0,014	419,396	0,017	418,916	0,049
400	437,873	0,006	438,452	0,007	437,080	0,012	436,872	0,021	436,375	0,044

Fonte: Elaborado pelo autor

OITO TAREFAS										
Tempo (ms)	**16 MHz**		**8 MHz**		**4 MHz**		**2 MHz**		**1 MHz**	
	Tempo Médio	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**	**Tempo Médio**	**Desvio Padrão**
16	16,835	44,998	16,868	44,022	16,861	41,809	-	-	-	-
32	36,414	61,409	36,420	60,758	36,275	59,170	33,705	54,888	-	-
48	53,250	74,452	53,284	73,969	53,133	72,668	53,679	71,188	-	-
64	71,121	71,839	71,168	71,510	70,975	70,546	70,878	68,942	69,870	64,541
80	87,249	87,756	87,313	87,405	87,095	86,355	87,427	85,062	87,325	82,156
96	105,128	105,780	105,207	105,446	104,954	104,354	104,516	102,284	104,785	99,800
112	122,295	123,146	122,378	122,816	122,069	121,675	122,388	120,300	122,243	117,440
128	140,175	0,004	140,275	0,006	139,928	0,010	139,992	0,022	139,698	0,049
144	157,704	0,003	157,808	0,006	157,408	0,009	157,359	0,021	157,142	0,048
160	175,229	0,003	175,345	0,006	174,893	0,010	174,841	0,021	174,592	0,039
176	192,756	0,003	192,888	0,006	192,380	0,010	192,329	0,023	192,047	0,040
192	210,281	0,003	210,423	0,005	209,874	0,012	209,806	0,021	209,500	0,041
208	227,811	0,004	227,956	0,006	227,363	0,011	227,290	0,022	226,956	0,048
224	245,336	0,004	245,496	0,005	244,847	0,009	244,767	0,016	244,411	0,034
240	262,863	0,004	263,041	0,007	262,341	0,011	262,250	0,020	261,865	0,036
256	280,396	0,006	280,577	0,005	279,833	0,011	279,733	0,022	279,320	0,045
272	297,924	0,006	298,122	0,008	297,332	0,011	297,219	0,022	296,781	0,032
288	315,460	0,004	315,666	0,007	314,818	0,009	314,718	0,020	314,232	0,044
304	332,990	0,005	333,213	0,010	332,317	0,012	332,179	0,022	331,686	0,041
320	350,525	0,007	350,767	0,011	349,821	0,016	349,658	0,023	349,137	0,044
336	368,058	0,005	368,311	0,013	367,306	0,013	367,139	0,017	366,587	0,058
352	385,579	0,005	385,836	0,010	384,805	0,021	384,619	0,022	384,049	0,044
368	403,116	0,007	403,390	0,007	402,308	0,023	402,091	0,020	401,499	0,032
384	420,644	0,005	420,935	0,009	419,782	0,012	419,576	0,027	418,959	0,028
400	438,178	0,006	438,488	0,009	437,285	0,009	437,063	0,026	436,402	0,027

Fonte: Elaborado pelo autor