UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO - CTC
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA DE AUTOMAÇÃO E SISTEMAS

Bruno Dourado Miranda

Análise de Tempo de Resposta de Tarefas no Sistema Operacional FreeRTOS

Florianópolis
2021

Bruno Dourado Miranda

Análise de Tempo de Resposta de Tarefas no Sistema Operacional FreeRTOS

Dissertação submetida ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina para a obtenção do título de mestre em Engenharia de Automação e Sistemas.
Orientador: Prof. Rômulo Silva de Oliveira, Dr.
Coorientador: Prof. Andreu Carminati, Dr.

Florianópolis
2021

Ficha de identificação da obra elaborada pelo autor,
através do Programa de Geração Automática da Biblioteca Universitária da UFSC.

Miranda, Bruno Dourado
Análise de Tempo de Resposta de Tarefas no Sistema Operacional FreeRTOS / Bruno Dourado Miranda ; orientador, Rômulo Silva de Oliveira, coorientador, Andreu Carminati, 2021.
146 p.

Dissertação (mestrado) - Universidade Federal de Santa Catarina, Centro Tecnológico, Programa de Pós-Graduação em Engenharia de Automação e Sistemas, Florianópolis, 2021.

Inclui referências.

1. Engenharia de Automação e Sistemas. 2. FreeRTOS. 3. Sistemas de Tempo Real. 4. Sistemas Operacionais de Tempo Real. 5. Tempos de Resposta. I. Oliveira, Rômulo Silva de. II. Carminati, Andreu. III. Universidade Federal de Santa Catarina. Programa de Pós-Graduação em Engenharia de Automação e Sistemas. IV. Título.

Bruno Dourado Miranda

Análise de Tempo de Resposta de Tarefas no Sistema Operacional FreeRTOS

O presente trabalho em nível de mestrado foi avaliado e aprovado por banca examinadora composta pelos seguintes membros:

Luís Fernando Arcaro, Dr.
Embraer S.A.

Prof. Mauro Marcelo Mattos, Dr.
Universidade Regional de Blumenau

Prof. Rômulo Silva de Oliveira, Dr.
Universidade Federal de Santa Catarina

Certificamos que esta é a **versão original e final** do trabalho de conclusão que foi julgado adequado para obtenção do título de mestre em Engenharia de Automação e Sistemas.

Documento assinado digitalmente
Werner Kraus Junior
Data: 05/08/2021 06:11:49-0300
CPF: 531.085.239-53
Verifique as assinaturas em https://v.ufsc.br

Coordenação do Programa de
Pós-Graduação

Documento assinado digitalmente
Romulo Silva de Oliveira
Data: 04/08/2021 19:16:10-0300
CPF: 407.617.200-06
Verifique as assinaturas em https://v.ufsc.br

Prof. Rômulo Silva de Oliveira, Dr.
Orientador

Florianópolis, 2021.

Este trabalho é dedicado a toda criança que é curiosa e
é sonhadora. O mundo pertence a vocês.

AGRADECIMENTOS

Aos meus orientadores, Dr. Rômulo Silva de Oliveira, professor e pesquisador da Universidade Federal de Santa Catarina (UFSC) e Dr. Andreu Carminati, professor e pesquisador do Instituto Federal de Santa Catarina (IFSC). O tempo dedicado a orientação, as rodadas de discussão e profundas revisões de texto durante a elaboração desta pesquisa foram fundamentais para o andamento deste trabalho. Adquirir experiência e conhecimento com dois pesquisadores de relevância na área de Sistemas de Tempo Real é motivo de grande orgulho.

Aos meus pais, Rosane e Júlio César, a quem lhes devo o carinho, o compartilhar de sonhos, projetos e desafios. O suporte financeiro durante desafiantes seis meses sem financiamento para pesquisa em tempo integral foi essencial. Nunca lhes faltou disposição de ajudar-me em meus momentos de necessidade.

A Larissa Lourençon, que está comigo desde o início desta incrível jornada. Obrigado por compreender os momentos em que não pude estar contigo para elaborar esta pesquisa e pelo prazer de sua companhia no LTIC/DAS para os cafés sem açúcar aos sábados e domingos.

Ao Programa de Pós-Graduação em Engenharia de Automação e Sistemas da Universidade Federal de Santa Catarina (PPGEAS/UFSC), na figura de seus docentes, discentes e funcionários pelo conhecimento ministrado, pela ajuda e suporte para a pesquisa.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

“O que gaba de ser perfeito, só é perfeito em tolice.”
(Charles H. Spurgeon, 1834-1892)

RESUMO

Os sistemas operacionais de tempo real (SOTR) são usados pela indústria para construir aplicações que possuem requisitos de temporização suave (*soft real-time*). Um sistema de tempo real é dividido em tarefas, que são fragmentos de código que possuem restrições temporais. Cada tarefa de um sistema de tempo real possui um tempo de execução e um tempo mínimo entre ativações sucessivas. Idealmente, um SOTR não deve ter impacto temporal na execução de sistemas de tempo real. No entanto, os algoritmos e estruturas de controle de um SOTR tem influência nos aspectos temporais de uma tarefa. Em um cenário idealista, os módulos de um SOTR deveriam ser determinísticos para que sua influência no sistema fosse visível e previsível, porém, isto é algo fora da realidade. As influências temporais de um SOTR são chamadas de *overheads*, que são execuções de rotinas internas de *microkernel* para gerenciar tarefas em execução em um microprocessador. Dessa forma, o objetivo dos projetistas de um SOTR é minimizar os *overheads* impostos pelo *microkernel* nas tarefas de aplicação em tempo real. O *Worst-Case Execution Time* (WCET) de uma tarefa é o tempo de processador que a tarefa leva do início ao fim de sua própria execução em seu pior cenário. O *Worst-Case Response Time* (WCRT) é o tempo que a tarefa leva da chegada à conclusão, considerando as interferências, *release jitters* e bloqueios que recebe de outras tarefas do sistema e do próprio *microkernel*. Quando um SOTR é usado, o *overhead* causa influência no tempo de resposta de cada tarefa, por isso é importante conhecer o comportamento temporal dos *overheads* e como eles podem influenciar o tempo de resposta do sistema. O *microkernel FreeRTOS* é de código aberto e distribuído com uma licença MIT. É também um SOTR flexível e adaptável em vários modelos de sistemas, tais como: Executivo cíclico com ou sem interrupções, tarefas aperiódicas ou periódicas, criação de tarefas em tempo de execução e prioridades fixas ou dinâmicas. O objetivo deste trabalho é apresentar uma análise de WCRT de tarefas no *FreeRTOS* quando executado na arquitetura *ARM Cortex-M*4. Os modelos algébricos criados foram utilizados em comparação aos testes realizados na plataforma ARM. Os resultados obtidos pelos modelos algébricos quando comparados com os valores temporais coletados nos testes fornecem evidências de que os modelos criados refletem o comportamento temporal de tarefas no *microkernel*.

Palavras-chave: Tempo de Reposta no Pior Caso. Sistema Operacional de Tempo Real. Sistemas de Tempo Real. ARM. *FreeRTOS*.

ABSTRACT

Real-Time Operating Systems (RTOS) are used by the industry to implement applications which have soft timing requirements (soft real-time). A Real-Time application is divided in tasks, that are code snippets that have temporal constraints. Each task of a real-time application usually has a execution time and a period. Ideally, a RTOS should have no temporal impact on the execution of real-time applications. However, the algorithms and control structures of a RTOS have influence on temporal aspects of a task. In a idealistic scenario, the RTOS modules should be deterministic, so their temporal influence on applications would be visible and predictable, however that is not our reality. The temporal influences of a RTOS are called overheads, which are executions of internal microkernel routines for managing tasks running on a microprocessor. In that way, the objective of the designers of a RTOS is to minimize the overheads imposed by the microkernel on the real-time application tasks. The Worst-Case Execution Time of a task is the time that task takes from start to finish its own execution in its worst scenario. The Worst-Case Response Time is the time that task takes from arrival to conclusion considering the interference, release jitters and blocking it receives from other system tasks and the microkernel itself. When a RTOS is used, the overhead cause influences in the response time of each task, so it is important to know the temporal behavior of the overheads and how they can influence application response time. FreeRTOS is an open-source microkernel project distributed with a MIT license. FreeRTOS is a flexible and adaptable RTOS in many system models, such as: Cyclic executive with or without interruptions, aperiodic or periodic tasks, creation of tasks at run time and fixed or dynamic priorities. The goal of this work is to present a Worst-Case Response Time analysis of FreeRTOS tasks when it runs on the ARM Cortex-M4 architecture. The algebraic models created were used in comparison to tests performed on the ARM platform. The results obtained by algebraic models when compared with the temporal values collected in the tests provide evidence that the created algebraic models reflect the tasks temporal behavior in the microkernel.

Keywords: Worst Case Response Time. Real-Time Operating System. Real-Time System. ARM. FreeRTOS.

LISTA DE FIGURAS

LISTA DE QUADROS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ADC	*Analog-to-Digital Converter*
API	*Application Programming Interface*
CAN	*Controller Area Network*
CPU	*Control Process Unit*
CSBC	Congresso da Sociedade Brasileira de Computação
CSV	*Comma-Separated Values*
DMA	*Direct Access Memory*
EDF	*Earliest Deadline First*
FAT	*File Allocation Table*
FIFO	*First In, First Out*
GFC	Grafo de Fluxo de Controle
GPIO	*General-Purpose Input/Output*
HLP	*Highest Locker Priority Protocol*
HPI	*High Priority Interrupts*
HWM	*High Water Mark*
IDE	*Integrated Development Environment*
LR	*Link Register*
LTS	*Long-Term Support*
MMU	*Memory Management Unit*
MPU	*Memory Protection Unit*
MSP	*Main Stack Pointer*
NPP	*Non-Preemptive Protocol*
NVIC	*Nested Vectored Interrupt Controller*
PC	*Program Counter*
PCB	*Process Control Block*
PCP	*Priority Ceiling Protocol*
PIP	*Priority Inheritance Protocol*
PSP	*Process Stack Pointer*
RM	*Rate Monotonic*
RTA	*Response Time Analysis*
SDIO	*Secure Digital Input Output*
SO	Sistema Operacional
SOPG	Sistema Operacional de Propósito Geral
SOTR	Sistema Operacional de Tempo Real
SPI	*Serial Peripheral Interface*
SRAM	*Static Random-Access Memory*
SRP	*Stack Resource Policy*
TCB	*Task Control Block*

TLB	*Translation Lookaside Buffer*
USART	*Universal Synchronous Asynchronous Receiver Transmitter*
UTC	*Coordinated Universal Time*
WCET	*Worst Case Execution Time*
WCRT	*Worst Case Response Time*

LISTA DE SÍMBOLOS

τ	Tarefa
T	Período de ativação de uma tarefa
C	Tempo de computação de uma tarefa
R	Tempo de resposta de uma tarefa
U	Utilização de CPU
B	Tempo de bloqueio de uma tarefa
J	*Release Jitter* de uma tarefa
S	Semáforo compartilhado entre tarefas
Π	Teto do sistema
Ω	Prioridade mínima do sistema
σ	Prioridade de uma tarefa
γ	Tarefa do tipo *Task*
θ	Tarefa do tipo *Interrupt*
L	Latência temporal da arquitetura
W	Janela de execução
I	Interrupção
Γ	Conjunto de todas as tarefas do tipo *Task*
Θ	Conjunto de todas as tarefas do tipo *Interrupt*
δ	Tarefa do tipo *Timer*
Δ	Conjunto de todas as tarefas do tipo *Timer*

SUMÁRIO

1 INTRODUÇÃO

Quaisquer sistemas que se caracterizam como de tempo real devem atender a uma série de requisitos, dentre eles, o estrito cumprimento de prazos, mantendo a sua corretude lógica e completa integridade de dados. Com a evolução da tecnologia de processamento e sofisticação de periféricos acoplados, foram construídos Sistemas Operacionais para projetos em que modularidade, menor complexidade de manutenção de aplicações programadas e respeito a propriedades lógico-temporais fossem necessários.

É fundamental separar a ideia do que é um Sistema Operacional de Propósito Geral (SOPG) do que é um Sistema Operacional de Tempo Real (SOTR). A função principal de um Sistema Operacional (SO), segundo Oliveira (2018), é abstrair as complexidades de gerenciamento de módulos de hardware aos projetistas e oferecer gerenciamento de recursos em um contexto de programação concorrente. O que diferencia um SOPG de um SOTR é a distinção nas políticas de coordenação na alocação de recursos. Um SOPG prioriza a justiça, onde, o objetivo é reduzir ou anular o efeito temporal de *starvation* aos processos que estão na fila de aptos. Já um SOTR têm uma política de alocação de recursos priorizando o cumprimento de requisitos temporais das aplicações, obedecendo sempre a política de prioridades adotada para o sistema.

Sistemas Operacionais de Tempo Real aplicados a dispositivos embarcados tendem a ser pequenos e acoplados a malhas de controle, buscando sempre diminuir a sua interferência no tempo de resposta de uma tarefa. Todavia, como influencia o comportamento temporal da aplicação, uma das abordagens em análise de uma tarefa de tempo real é a avaliação de *Worst Case Response Time* (WCRT) (OLIVEIRA, 2018). Essa é uma variável a ser levada em consideração ao se estudar a viabilidade de adoção de um SOTR em um projeto, onde o sistema operacional a ser adotado interfere na aplicação programada pelo usuário e o algoritmo de escalonamento e de gerenciamento de seções críticas pode fazer com que o deadline de uma tarefa seja perdido.

Aliado a influência temporal causada pelo SO em uma tarefa existe a influência temporal causada pelo hardware. Microprocessadores da década de 80, como o Intel 8051, que é considerado como um dos mais populares do mundo, possuem tempos de computação de instrução definidos em ciclos de *clock* que são constantes, como definidos em seu manual de instruções de máquina. No entanto, com o passar dos anos, fabricantes de processadores e microprocessadores foram adicionando mecanismos de aceleração de processamento como *pipelines*, *branch predictors*, *Direct Access Memory* (DMA), *out-of-order execution* e memórias *cache*. O efeito temporal dessas adições ampliam a velocidade média com que instruções enviadas ao microprocessador são executadas e ampliam a variabilidade de tempo na execução de instruções.

O ARM Cortex-M4 é um microprocessador de 32 bits *single core* com um *pipeline* de três estágios, suporte a ponto flutuante, dezessete registradores e que possui suporte ao SOTR *FreeRTOS*.

O *FreeRTOS* (AWS, 2021) é um *microkernel* de tempo real que pode ser usado em projetos aplicados a dispositivos embarcados. Ele possui APIs para gerenciamento de semáforos, filas, notificação de tarefas e suporte a sistemas mono-processados ou bi-processados. Ele possui código aberto, pode ser utilizado em projetos comerciais e é suportado por mais de quarenta microprocessadores diferentes. Com amplo suporte da comunidade, já chegou na sua versão 10 e possui uma versão *Long-Term Support* (LTS) (AWS, 2021).

1.1 OBJETIVOS E MOTIVAÇÃO

O objetivo deste trabalho é, à luz da teoria de sistemas de tempo real, medir e analisar matematicamente o comportamento dos tempos de resposta de tarefas periódicas ou esporádicas com prioridade fixa obtidos a partir de sua execução no *FreeRTOS*, um sistema operacional de tempo real. O *FreeRTOS* (AWS, 2021) é um SOTR que permite a criação de um sistema com laço contínuo com ou sem interrupções, tarefas aperiódicas ou periódicas, criação de tarefas em tempo de execução e prioridades variáveis ou não. Além do *Worst Case Execution Time* (WCET) há a medida WCRT que considera o perfil da tarefa e outras variáveis de sistema, como bloqueios, interrupções e possíveis atrasos de outras tarefas. Tudo isto sofre influência temporal da arquitetura-alvo na qual o *FreeRTOS* estará executando, como memória *cache*, *pipelines*, *branch-predictors* e o próprio Grafo de Fluxo de Controle (GFC) gerado pelo compilador.

Este trabalho concentra-se especificamente no contexto ao qual o *FreeRTOS* está executando e o comportamento do sistema à luz do modelo matemático de avaliação do WCRT. Portanto, este projeto se limita em tarefas executando no *FreeRTOS* e será baseado na análise do tempo de resposta do trabalho de Audsley *et al*. (1993) propondo-se assim sua adaptação para o SOTR analisado.

O *microkernel* de tempo real é alvo de estudos pela comunidade científica e como contribuição espera-se modelo(s) algébrico(s) adaptado(s) de Audsley *et al.* (1993) para o contexto de sistemas em execução no *FreeRTOS*.

Com o desenvolvimento deste trabalho espera-se contribuir com uma análise temporal de tarefas executando em um sistema embarcado com o SOTR *FreeRTOS* tendo em vista os seus tempos de resposta. Assim como pretende-se descrever o comportamento de módulos do *microkernel* para a execução de tarefas, lógicas de escalonamento e bloqueios.

1.1.1 Objetivo Geral

Construir um modelo de WCRT de tarefas executando no *FreeRTOS* no microprocessador ARM Cortex-M4, de forma a mapear interferências e bloqueios temporais de outras tarefas e do *microkernel*.

O trabalho de pesquisa descrito nesta dissertação de mestrado pode ser classificado (SILVA; MENEZES, 2005), do ponto de vista da sua natureza, como pesquisa aplicada. Do ponto de vista da forma de abordagem do problema pode ser descrito como pesquisa quantitativa. Com respeito aos seus objetivos trata-se de uma pesquisa explicativa cujos procedimentos técnicos são próprios de uma pesquisa experimental.

1.1.2 Objetivos Específicos

São os objetivos específicos:

(a) Levantamento de características do sistema-alvo;

(b) Caracterização do processador da plataforma-alvo;

(c) Levantamento de técnicas utilizadas para medições de tempos de resposta;

(d) Adaptar o modelo de análise de tempo de resposta como descrita na literatura de tempo real para a sua aplicação no contexto do *FreeRTOS*;

(e) Aplicação de modelo(s) matemático(s) em cenário(s) de teste(s).

1.2 ESTRUTURA DO TEXTO

O trabalho está organizado em oito capítulos, sendo este capítulo incluso, que está composto por conceitos básicos, motivação e objetivos.

No Capítulo 2 são apresentadas as bases teóricas de análise de tarefas em um sistema de tempo real para sistemas mono-processados, sendo conceitos e definições gerais de tarefas, escalonamento de prioridade fixa e protocolos de sincronização entre tarefas.

No Capítulo 3 são apresentadas as características de hardware do microprocessador utilizado no trabalho que podem influenciar os tempos de computação de tarefas no *microkernel*, assim como estruturas utilizadas por ele para construção do SOTR, como os tratadores de interrupção do microprocessador.

No Capítulo 4 são apresentados os conceitos, estruturas, modos de orgnização de tarefas e escalonamento do SOTR alvo de análise deste trabalho.

No Capítulo 5 é apresentadas a plataforma experimental do trabalho, com características e estratégias de coleta de dados gerados pelas tarefas do SOTR. É apresentada a bancada de testes para o ARM Cortex-M4.

No Capítulo 6 é apresentada uma análise algébrica de tarefas sob gestão do *microkernel*, assim como os *overheads* do SOTR que podem influenciar o WCRT de uma tarefa. No capítulo se encontra a principal contribuição deste trabalho.

No Capítulo 7 são apresentados testes realizados na plataforma e uso dos modelos algébricos criados e apresentados no capítulo anterior.

No Capítulo 8 são apresentadas as considerações finais da pesquisa, as contribuições e possibilidades de trabalhos futuros.

2 FUNDAMENTOS DE SISTEMAS DE TEMPO REAL

Neste capítulo serão apresentados os fundamentos e conceitos de um sistema de tempo real para sistemas mono-processados. Um sistema mono-processado é um sistema que possui um único processador (também chamado *single-core* ou *mono-core*). O processador é um recurso a ser disputado quando duas ou mais tarefas (ou *threads*) precisam se executadas neste processador, seja esse sistema de tempo real ou não.

2.1 CONCEITOS BÁSICOS

Em um SOTR, uma tarefa pode ser considerada como qualquer trecho de código que possua um requisito temporal, uma periodicidade e uma consequência adversa, seja ao sistema ou seja ao meio externo, de não cumprimento de seu *deadline*. Um *deadline* é o tempo que a tarefa tem, desde sua ativação, para concluir sua computação. Para administrar o comportamento lógico-temporal de um conjunto de tarefas que são executadas sobre um SOTR é necessária a ação de um escalonador. O escalonador nada mais é do que um algoritmo que lida com questões gerenciais de recursos de processamento, sendo que existem dezenas de algoritmos implementados e alguns são voltados para tempo real (OLIVEIRA, 2018).

Para caracterizar o comportamento de execução de tarefas em um sistema de tempo real, é necessário adotar algum tipo de modelo que sirva como referência para cálculos de escalonabilidade e utilização de um processador. Em um modelo inicial é suposto que o algoritmo de escalonamento seja preemptivo. No modelo desenvolvido por Liu e Layand (1973) uma tarefa possui um período de ativação e possui um tempo de computação, já para o escalonamento é adotada a análise a partir do escalonamento com lógica *Earliest Deadline First* (EDF) ou *Rate Monotonic* (RM).

As tarefas presentes nesse modelo podem sofrer interferências, isto é, ter sua execução interrompida por uma ou mais tarefas de mais alta prioridade ativada(s). É assumido que as tarefas são independentes entre si, portanto, não há compartilhamento de seções críticas com tarefas com mais baixa prioridade, logo, a presença de algum bloqueio, onde uma tarefa de baixa prioridade que obtém acesso a um semáforo compartilhado impede a execução de uma tarefa de mais alta prioridade, não existe.

A partir do modelo inicial desenvolvido por Liu e Layland (1973) é possível estendê-lo para lidar com aspectos de bloqueio e atrasos de liberação. Tarefas executadas sobre um SOTR por vezes são flexíveis e o modelo de Liu e Layland (1973) é restrito por conta de requisitos exigidos pela análise apresentada. No modelo desenvolvido por Audsley *et al*. (1993) é possível a adoção de um modelo mais flexível de análise a partir da adição da possibilidade da tarefa sofrer bloqueio(s) e um atraso de liberação denominado *release jitter*.

2.2 CARACTERÍSTICAS DE UMA TAREFA

Conforme Oliveira (2018), em um SOTR, uma tarefa (τ) é considerada como qualquer trecho de código que esteja diretamente relacionado a alguma funcionalidade com restrição temporal. Toda tarefa τ que pertence a algum conjunto analisável de tarefas possui uma série de características e definições comuns. São listadas a seguir as características comuns entre elas (FARINES; FRAGA; OLIVEIRA, 2000):

- **Prioridade**: A prioridade de uma tarefa é um escolha que é interpretada através dos requisitos temporais de um projeto. Uma tarefa de alta prioridade possui preferência no seu atendimento em relação a uma tarefa de baixa prioridade e também é possível uma tarefa possuir a mesma prioridade que outra tarefa. Em implementação é possível atribuir um número para a prioridade onde a ordem de preferência de atendimento pode ser crescente ou decrescente a depender do sistema. A prioridade de uma tarefa pode ser fixa ou dinâmica.
- **Recorrência**: A recorrência ou *T* é um aspecto a ser levado em consideração na previsibilidade temporal de uma tarefa e pode impactar na análise temporal das demais. Oliveira (2018) elenca três tipos de comportamentos distintos de recorrências de uma tarefa:
  - Periódica: A tarefa possui um comportamento em que é ativada para execução em intervalos regulares;
  - Esporádica: A tarefa possui um comportamento em que há um intervalo mínimo entre cada liberação para execução;
  - Aperiódica: A tarefa não possui um comportamento periódico e nem sequer esporádico, logo, nenhum padrão de recorrência é conhecido sobre ela. Isso pode gerar rajadas de τ no sistema, o que pode levar a um certo descontrole no seu atendimento.
- ***Deadline***: O *deadline* de uma tarefa é o tempo que a tarefa tem para que seu processamento seja concluído sem que haja algum tipo de ônus lógico ou alguma falha ao sistema. Ainda segundo Oliveira (2018) há dois tipos distintos de *deadline*, o relativo e o absoluto. O *deadline* relativo é definido a partir do momento de chegada da tarefa, já o *deadline* absoluto é relacionado ao tempo *Coordinated Universal Time* (UTC). Quando um *deadline* é igual ao seu *T*, diz-se que o *deadline* de τ é implícito.

A classificação de um *deadline* em uma tarefa é sempre vinculada às consequências de sua perda. Em Oliveira (2018) são apresentados três tipos distintos de *deadline*:

- ***Deadline* crítico**: O não cumprimento gera um resultado catastrófico ao sistema no momento de sua perda;
- ***Deadline* firme**: O não cumprimento gera um resultado lógico-temporal que se torna inútil ao sistema no momento de sua perda;
- ***Deadline* brando**: O não cumprimento faz com o resultado lógico-temporal vá perdendo valor conforme a tarefa não chega em sua conclusão, até que seu resultado lógico-temporal se torna inútil ao sistema.

Em relação ao comportamento temporal de uma tarefa Farines, Fraga e Oliveira (2000), elencam um padrão de requisitos, que são:

- **Tempo de Chegada**: É o momento em que o sistema recebe a informação que a tarefa deve ser despertada. Em tarefas periódicas esse momento de chegada sempre coincide com T;
- **Tempo de Liberação**: É o momento em que o sistema aloca a tarefa que foi despertada em uma fila de prontos para a concorrência de recursos de processamento;
- **Tempo de Início**: É o momento em que a tarefa inicia sua execução;
- **Tempo de Término**: É o momento em que a tarefa completa sua execução;
- **Tempo de Computação**: Tempo de computação ou C é o tempo que a tarefa necessita para sua completa execução. É também conhecido como *job*. O WCET ou pior tempo de computação é quando a tarefa atinge um cenário em que aspectos lógicos e de hardware fazem com que a tarefa atinja o seu pior cenário possível de execução;
- **Tempo de Resposta**: Tempo de resposta ou R é o tempo total entre o tempo de chegada e o tempo em que realiza sua completa execução. O WCRT ou pior tempo de resposta é quando a tarefa atinge um cenário em que aspectos lógicos e de hardware fazem com que a tarefa atinja o seu pior cenário possível.

A coordenação lógico-temporal de tarefas em um sistema de tempo real fica a cargo de um escalonador. Um escalonador (*scheduler*) nada mais é que um algoritmo que gerencia os recursos de processamento de um sistema. Com vistas a escalonamento, três tipos de algoritmos são utilizados (OLIVEIRA, 2018):

- **Preemptivo**: Uma tarefa pode ter seu direito à utilização de recursos de processamento interrompido pela chegada de uma tarefa de alta prioridade.

- **Não-preemptivo**: Uma tarefa de alta prioridade, no momento de sua chegada, não tem o direito de retirar os recursos de processamento de uma tarefa de baixa prioridade.
- **Colaborativo**: O direito aos recursos de processamento é retirado de alguma tarefa a partir de algum ponto especificado por ela mesma.

É apresentada na seção a seguir a teoria de escalonamento de prioridade fixa voltada a sistemas de tempo real.

2.3 ESCALONAMENTO DE PRIORIDADE FIXA EM TEMPO REAL

Em tempo real, o escalonamento é voltado de acordo com o tipo de prioridade adotada pelo sistema. Como tipo de prioridade, entenda-se prioridades **fixas** ou **dinâmicas**. Normalmente, quando se faz uma análise de escalonabilidade, isto é, uma análise matemática temporal de uma tarefa inserida no sistema, é necessário que o projetista conheça os tempos de computação de cada tarefa que irá disputar os recursos de processamento. Para tal, em modelos de análise sempre se assume C como sendo o WCET. Para um sistema de tempo real, testes são classificados em função de sua escalonabilidade. Classifica-se os tipos de teste como sendo (FARINES; FRAGA; OLIVEIRA, 2000):

- **Suficientes**: Garantem a escalonabilidade de um certo subconjunto de tarefas analisado a partir de uma série de requisitos, mas, outros subconjuntos de tarefas que não atendem os requisitos do teste suficiente também podem ser escalonadas, portanto, nesse tipo de teste nada pode ser dito sobre o subconjunto reprovado mas sim sobre o subconjunto aprovado;
- **Necessários**: Não conseguem afirmar nada sobre um subconjunto aprovado, no entanto, garantem que os subconjuntos reprovados de fato não são escalonáveis;
- **Exatos**: Permite identificar subconjuntos escalonáveis e não escalonáveis. Um teste exato é ao mesmo tempo suficiente e necessário.

2.3.1 Escalonabilidade de tarefas periódicas e esporádicas

Considerado pela comunidade científica como o trabalho inicial de análise comportamental de tarefas em um sistema de tempo real, o modelo de Liu e Layland (1973) adota o conceito de utilização de uma *Control Process Unit* (CPU) para tarefas periódicas ou esporádicas. Como um teste suficiente, eles garantem que tarefas aprovadas no teste são de fato escalonáveis, no entanto, em caso de reprovação, nada pode ser dito sobre a escalonabilidade do sistema. O modelo foi construído para tarefas que

executam um algoritmo de escalonamento de prioridade fixa e variável. Para prioridade fixa, o algoritmo utilizado é o *Rate Monotonic* (RM) e para prioridades variáveis, o algoritmo adotado é o EDF.

O algoritmo de RM é um algoritmo em que a prioridade é diretamente relacionada ao período da tarefa. A prioridade mais alta é proporcionada à tarefa com menor período, a tarefa com o segundo menor período recebe a segunda mais alta prioridade e assim por diante. É também um algoritmo preemptivo, isto é, quando ocorre a chegada de uma tarefa de mais alta prioridade que a tarefa que está sendo processada no momento, o escalonador efetua um chaveamento de contexto, em outras palavras, armazena as informações dos registradores da tarefa em memória e fornece os recursos de processamento para a tarefa de alta prioridade que chegou.

O início da análise se baseia no instante crítico de uma tarefa. O artigo de Liu e Layland (1973) define o instante crítico como o instante em que uma tarefa é liberada em conjunto com todas as tarefas. Logo, para tal análise Liu e Layland (1973) utilizam as seguintes premissas:

a) Todas as tarefas são independentes entre si;

b) Todas as tarefas são periódicas;

c) C é considerado como WCET;

d) Todas as tarefas possuem um *deadline* implícito;

e) O efeito temporal de trocas de contexto é desprezado;

f) As tarefas chegam no instante crítico.

Para tarefas esporádicas, o T pode ser considerado como o tempo mínimo entre as chegadas de τ. Portanto, a utilização de uma tarefa no processador é designado pela Equação (2.1):

$$U_i = \frac{C_i}{T_i} \tag{2.1}$$

A fórmula de utilização do subconjunto de tarefas disputando a CPU é dado pela Equação (2.2):

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1 \tag{2.2}$$

Como a análise de escalonabilidade é feita em tempo de projeto para RM, o teste suficiente proposto por Liu e Layland (1973) se baseia no cálculo de utilização de CPU obtida pela Equação (2.3), onde n representa o número de tarefas presentes no sistema:

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1) \tag{2.3}$$

Se as tarefas obedecem ao limiar traçado pela Equação (2.3) o teste garante que nenhuma delas jamais perderá um *deadline*. Supondo que n tende ao infinito, é possível traçar que a equação converge para um limiar de 0,693, como apresentado pela Equação (2.4):

$$\lim_{n\to\infty} n(2^{\frac{1}{n}} - 1) = 0,693 \tag{2.4}$$

Caso o limiar de 69,3% de utilização de uma CPU seja obedecido pelo subconjunto analisado, é possível afirmar, não importando a quantidade de tarefas ou seus períodos, que os *deadlines* de todas as tarefas serão cumpridos.

2.3.2 Extensão de análise em um modelo preemptivo

No trabalho de Joseph e Pandya (1986) é estendido o modelo de escalonamento de prioridade fixa proposto em Liu e Layland (1973) para ser um teste exato ao invés de ser apenas suficiente. A análise feita parte da busca do pior tempo de resposta de uma tarefa periódica a partir do instante crítico, que é definido como o instante de chegada de τ e de todas as tarefas de mais alta prioridade que ela. O modelo algébrico definido pela Equação (2.5):

$$R_i = C_i + \sum_{j \in HP(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j \tag{2.5}$$

Em que o conjunto *HP* contém tarefas de mais alta prioridade que a tarefa i.

Para tarefas esporádicas, nas quais Tj_{min} representa o intervalo mínimo entre chegadas de uma tarefa j no sistema, o mesmo raciocínio se aplica, conforme o modelo algébrico apresentado na Equação (2.6):

$$R_i = C_i + \sum_{j \in HP(i)} \left\lceil \frac{R_i}{T_{j_{min}}} \right\rceil C_j \tag{2.6}$$

Em que o conjunto *HP* contém tarefas de mais alta prioridade que a tarefa i.

Como na Equação (2.6) não é possível isolar R_i, o tempo de resposta é obtido através das iterações de R_i. A partir do momento em que o modelo converge, em outras palavras, que $R_i^y = R_i^{y+1}$, onde y representa o número de iterações de R, o pior tempo de resposta é encontrado. No modelo algébrico é assumido que $R^0 = C_i$. Deste modo, se o pior tempo de resposta é menor ou igual ao *deadline* a tarefa é escalonável. No entanto, em contexto de tempo real, é muito comum que as tarefas não sejam independentes e que ocorram atrasos em sua liberação para disputa de recursos. Para tanto, Audsley *et al.* (1993) propõem uma abordagem de escalonamento que leva em consideração as consequências desses efeitos temporais.

Um **bloqueio** (B) decorre de interações entre tarefas de alta prioridade e baixa prioridade. Quando uma tarefa de baixa prioridade se utiliza de recursos comparti-

lhados com uma tarefa de mais alta prioridade, que também pretende no momento utilizar a seção crítica, a tarefa de mais alta prioridade pode ter sua execução atrasada quando a tarefa de baixa prioridade está com o acesso garantido por um semáforo. De acordo com Oliveira (2018) a forma como é obtido o máximo tempo de bloqueio de uma tarefa B_{max} depende do tipo de protocolo adotado pelo sistema. Para sistemas mono-processados e que possuem escalonamento preemptivo os protocolos são discutidos em Sha, Rajkumar e Lehoczky (1990).

Já um **atraso de liberação**, J, é o máximo atraso temporal entre o tempo de chegada e o tempo de liberação para disputa de recursos de processamento com outras tarefas do sistema. Isso ocorre por causa de uma espera por mensagem externa à tarefa, por interrupções desativadas, ou por conta de um escalonador movê-la de uma fila de tarefas que não disputam o processador para a fila de prontos.

Para tal, é necessária apenas a inclusão desses tempos de bloqueio e de atraso máximo de liberação na Equação (2.6), como demonstrado por Audsley *et al.* (1993) na Equação (2.7).

$$R_i = J_i + W_i \tag{2.7}$$

em que,

$$W_i = C_i + B_i + \sum_{j \in HP(i)} \left\lceil \frac{W_i + J_j}{T_j} \right\rceil C_j$$

Em que o conjunto HP contém tarefas de mais alta prioridade que a tarefa i.

2.4 PROTOCOLOS DE SINCRONIZAÇÃO

No WCRT de uma tarefa, o protocolo utilizado na sincronização de tarefas pode ter uma influência significativa no tempo B. Na disputa de recursos compartilhados, principalmente em sistemas mono-processados, bloqueios podem ser mais evidentes e se tornarem um problema na viabilidade de tarefas implementadas em um sistema de tempo real, com vista a estimativa de WCRT de tarefas potencialmente bloqueáveis.

Em aspectos de coordenação de bloqueio, os protocolos abordados podem alterar dinamicamente as prioridades das tarefas que outrora eram fixas. Para tanto, uma nova prioridade é atribuída à tarefa, uma **prioridade dinâmica** ou **prioridade efetiva**, p_i, onde $p_i \geq P_i$, através do qual P_i é o valor de prioridade fixa, ou nominal, da tarefa.

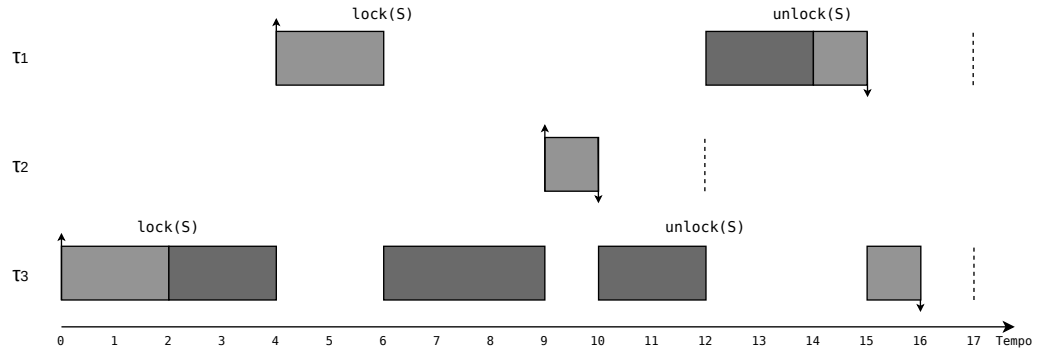
2.4.1 Inversão descontrolada de prioridades

É caracterizada em Sha, Rajkumar e Lehoczky (1990) uma inversão de prioridade como um fenômeno de bloqueio de uma tarefa de alta prioridade por tarefas de

baixa prioridade. Sem algum tipo de protocolo que coordene a política de acesso aos recursos compartilhados do sistema, ocorre um fenômeno denominado pela literatura como inversão descontrolada de prioridades, em que a política de prioridades do sistema é desrespeitada. Uma inversão descontrolada de prioridades se define quando o tempo de bloqueio *B* é inconclusivo e pode ser protelado infinitamente.

É apresentado na Figura 1 um exemplo do fenômeno. As tarefas τ_1,τ_2 e τ_3 são de alta, média e baixa prioridade, respectivamente. As tarefas τ_1 e τ_3 compartilham uma seção crítica, denominada *S*, que é protegida por um *mutex*. As operações de *lock* e *unlock* são representadas por *lock(S)* e *unlock(S)*, respectivamente.

Figura 1 – Inversão descontrolada de prioridades em mono-processador



Fonte – Autor

O instante de chegada de uma tarefa é representado pela seta direcionada para cima e o instante de término de uma tarefa é representado pela seta direcionada para baixo, no final de cada execução, que é representada pelo retângulo. O *deadline* de cada tarefa é representada pela linha tracejada.

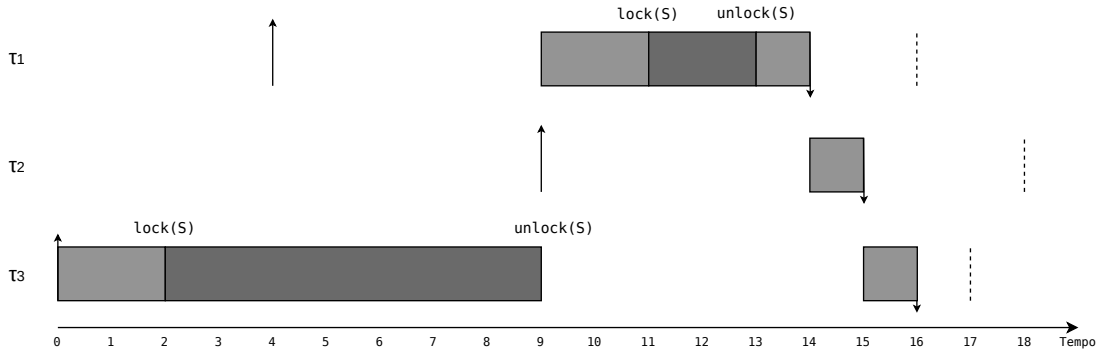
Ainda no cenário da Figura 1, tarefa τ_3 obtém o acesso a *S* e bloqueia τ_1 no instante 6, impondo ao sistema uma inversão de prioridade. Enquanto executava a seção crítica, τ_3 foi interrompida por τ_2 e, até o fim da execução de τ_2, τ_1 se comporta na prática como a tarefa com mais baixa prioridade do sistema. Após o término de τ_2 e τ_3, a tarefa τ_1 é desbloqueada e encerra, antes de chegar ao seu respectivo *deadline*, no instante 16.

Não é possível evitar a inversão de prioridades em um sistema de tempo real, entretanto, é possível impedir a inversão descontrolada de prioridades com protocolos adequados para lidar com tarefas em situação de bloqueio. Buttazzo (2011) e Oliveira (2018) elencam alguns protocolos para coordenação de seções críticas, que serão discutidos nas subseções a seguir.

2.4.2 *Non-Preemptive Protocol (NPP)*

Uma abordagem que pode evitar uma inversão descontrolada de prioridades segundo Buttazzo (2011) é o *Non-Preemptive Protocol* (NPP). A partir do uso desse protocolo, a tarefa que compartilha uma seção crítica efetua uma operação de *lock* em um *mutex* e obtém o acesso ao recurso compartilhado. Nesse instante as interrupções são desativadas pelo sistema até a operação de *unlock* pela tarefa. É apresentado na Figura 2 o cenário visualizado na Figura 1 com execução do protocolo NPP.

Figura 2 – Funcionamento do protocolo *NPP* em mono-processador



Fonte – Autor

Uma característica negativa desse protocolo é que de forma desnecessária a tarefa que tem acesso ao recurso compartilhado que gerou o efeito de interrupções desabilitadas no processador acaba bloqueando ou impondo um atraso de liberação a uma tarefa de alta prioridade (OLIVEIRA, 2018). Esse fenômeno causa também um desrespeito à política de prioridades do sistema, no entanto, o problema de inversão descontrolada de prioridades é evitado.

Neste protocolo, uma tarefa de alta prioridade pode ser bloqueada apenas uma vez durante um instante crítico por conta das interrupções desativadas na operação de *lock* em um *mutex* por uma tarefa de mais baixa prioridade, logo, de forma genérica, o máximo tempo de bloqueio de uma tarefa utilizando o protocolo NPP pode ser definido pelo maior tempo utilizado de uma seção crítica por uma tarefa de mais baixa prioridade, de acordo com Buttazzo (2011).

2.4.3 *Priority Inheritance Protocol (PIP)*

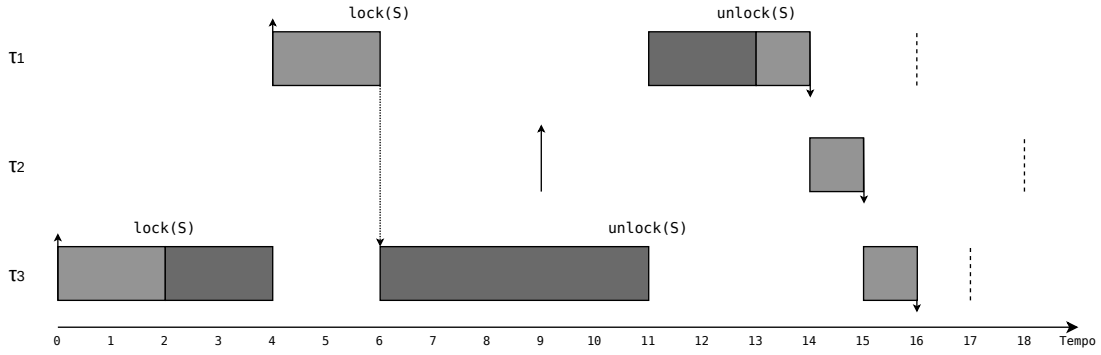
No protocolo *Priority Inheritance Protocol* (PIP), Sha, Rajkumar e Lehoczky (1990), é definido que quando bloqueada, a tarefa de mais alta prioridade repasse a sua prioridade efetiva para a tarefa que a bloqueou. Duas formas de bloqueio podem ocorrer nessa abordagem:

- **Bloqueio direto**: Quando uma tarefa de baixa prioridade obtém acesso a um *mutex S* e uma tarefa de mais alta prioridade é bloqueada diretamente ao realizar a operação de *lock* no mesmo *mutex S*;
- **Bloqueio por herança**: Quando uma tarefa de média prioridade é bloqueada por uma tarefa de baixa prioridade que herdou a prioridade nominal de uma tarefa de alta prioridade.

Para exemplificar a lógica, assume-se H_p como uma tarefa de alta prioridade e L_p como uma tarefa de baixa prioridade. A prioridade efetiva de L_p é alterada apenas quando H_p está utilizando algum mecanismo de sincronização e realiza uma operação de *lock* em um *mutex* ocupado por L_p. A partir desse instante, a prioridade efetiva de L_p é alterada para o valor de prioridade de H_p. Ao realizar a operação de *unlock*, a prioridade efetiva de L_p retorna ao seu valor original, isto é, igual ao valor da prioridade fixa nominal e H_p obtém acesso à seção crítica.

Retornando ao exemplo da Figura 1, o problema da inversão descontrolada de prioridades demonstrado é dirimido com o uso do protocolo de herança de prioridades, como exposto na Figura 3. A tarefa mais prioritária τ_1 repassa sua prioridade para τ_3 na falha da tentativa de *lock* no *mutex*. Mesmo com a chegada da tarefa τ_2, τ_3 possui uma prioridade efetiva mais alta que τ_2, portanto, ela continua sua execução até concluir a seção crítica e na operação de *unlock* sua prioridade efetiva retorna ao mesmo valor de sua prioridade nominal. Com isso, τ_2 executa após o término de τ_1, voltando a respeitar assim a política de prioridades.

Figura 3 – Funcionamento do protocolo PIP em mono-processador



Fonte – Autor

Esse protocolo é simples em que o *overhead* é considerado baixo, no entanto, não impede *deadlocks* em um uso de várias seções críticas aninhadas, ficando a cargo do projetista evitá-los em código ou prever sua ocorrência.

Com relação ao tempo máximo de bloqueio de uma tarefa utilizando o protocolo PIP, tudo depende de como são realizadas as operações de *lock* e *unlock* a partir de um instante crítico. Para cenários simples, Buttazzo (2011) apresenta um algoritmo que calcula o máximo tempo de bloqueio de uma tarefa, contudo, esse algoritmo considera como premissa que não há aninhamento de *mutexes*. A implementação completa do algoritmo quando não há bloqueios aninhados se encontra nos anexos deste trabalho, na Seção A.1. Já para um comportamento complexo de bloqueios, há um algoritmo demonstrado em Rajkumar (1991) que faz uma busca exaustiva, com tempo exponencial do maior tempo possível de bloqueio de uma tarefa.

2.4.4 *Priority Ceiling Protocol* (PCP)

O *Priority Ceiling Protocol* (PCP), Sha, Rajkumar e Lehoczky (1990), é classificado como uma alternativa ao PIP por conta de sua prevenção a *deadlocks* no sistema e redução de tempo de bloqueio no pior caso (OLIVEIRA, R. S., 2018). Cada seção crítica no sistema possui um valor de teto de prioridade (*ceiling*) do recurso, normalmente simbolizada por $\prod$, que nada mais é do que a mais alta prioridade nominal entre todas as tarefas. O sistema também possui um teto $\prod$. Caso nenhum recurso esteja sendo utilizado, o teto do sistema retorna uma prioridade mais baixa do que todas as outras tarefas do sistema. Esta prioridade é denotada por Ω. As seguintes regras de alocação em um recurso compartilhado são adotadas (OLIVEIRA,2018):

- Se uma seção crítica *S* está alocada para uma outra tarefa, então, a tarefa que requisita o acesso à seção crítica é bloqueada.
- Se uma seção crítica *S* está livre:
  - O recurso é alocado para a tarefa solicitante τ se a prioridade efetiva de τ é maior que o teto do sistema;
  - Caso a prioridade efetiva seja menor ou igual, a tarefa solicitante é bloqueada.

Além do bloqueio direto e por herança, já discutidos na subseção anterior, o PCP pode sofrer mais um tipo de bloqueio, o *bloqueio por teto*. Um bloqueio por teto é um bloqueio quando um recurso está livre no sistema, no entanto, uma tarefa de alta prioridade não pode utilizá-lo, pois o teto do sistema é mais alto e impossibilita o acesso ao recurso. Esse comportamento rigoroso em relação a tarefa de mais alta prioridade impede o *deadlock* no sistema.

2.4.5 *Highest Locker Priority Protocol* (HLP)

O *Highest Locker Priority Protocol* (HLP), (KLEIN; RALYA, 1990) , também conhecido como *Immediate Priority Ceiling*, modifica os dois protocolos discutidos anteri-

ormente, o NPP e o PCP. Ao adentrar em uma seção crítica, a tarefa τ recebe a mais alta prioridade nominal entre as tarefas que compartilham o recurso, se e somente se tal prioridade for mais alta que a própria prioridade corrente de τ.

Entre as vantagens desse protocolo, é visível a maior facilidade de implementação e menor frequência de chaveamentos de contexto em comparação ao protocolo PCP (OLIVEIRA, 2018).

2.5 RESUMO

Neste capítulo foi introduzido o conceito de tarefa, suas principais características e a ideia de *deadline*. Além disso, neste capítulo, para tarefas com prioridade fixa foram discutidas as principais concepções sobre a teoria de escalonamento em sistemas mono-processados. Foram expostos modelos de previsibilidade de ocupação de CPU por tarefas periódicas e esporádicas, assim como foram apresentados modelos algébricos de pior tempo de resposta. Para sistemas mono-processados foram apresentados protocolos de sincronização de tarefas periódicas NPP, PIP, PCP e HLP.

Para tarefas com prioridade fixa, a teoria de escalonamento pode ser aplicada para prioridades variáveis, como o EDF, *deadlines* arbitrários e para sistemas multiprocessados (RAJKUMAR; SHA; LEHOCZKY, 1988). Há também várias abordagens de gerenciamento de recursos compartilhados para prioridades variáveis, onde é possível destacar o *Stack Resource Policy* (SRP), protocolo este que não foi discutido neste capítulo.

3 CARACTERÍSTICAS RELEVANTES DO ARM CORTEX-M4

Neste capítulo são consideradas as características do microprocessador ARM Cortex-M4, um processador *single-core*, com vistas aos fatores relevantes à construção de um Sistema de Tempo Real e possíveis influências em seu tempo de resposta. A linha de processadores ARM Cortex-M é preparada pela fabricante para que um projetista, se desejar, possa construir seu próprio SOTR, logo, conhecer as características de hardware de um microprocessador ajuda o projetista a entender aspectos de construção de um sistema, bem como identificar possíveis módulos que podem causar variação temporal e também para poder proporcionar uma melhor depuração em busca de erros lógico-temporais.

Em uma visão puramente modular, o microprocessador é simples em comparação aos processadores *multi-core*. O microprocessador possui dezessete registradores, dos quais treze são para uso geral. Também, possui um *pipeline* de três estágios, única estrutura com tempos de execução probabilista. Para gerenciamento de exceções, possui um módulo específico chamado *Nested Vectored Interrupt Controller* (NVIC). A frequência máxima de *clock* é de 180 *MHz*, isso significa que nesta frequência, uma instrução de máquina demora aproximadamente cinco nanosegundos. Não possui memória *cache*, que por consumir muita área dentro de um *chip* é reservado aos processadores mais sofisticados.

As seções a seguir apresentam as características relevantes ao comportamento de hardware que influenciam uma tarefa embarcada. Protocolos de comunicação em barramento e periféricos não serão abordados neste capítulo.

3.1 INTERRUPÇÕES

Para atender uma interrupção de forma eficiente, há um módulo de gerenciamento de interrupções na linha *Cortex-M*, denominado pela fabricante como NVIC, nele é possível agrupar uma série de interrupções por prioridade e permitir seu aninhamento. Também é possível agendar interrupções se o módulo de tratamento se encontra desabilitado, logo, as interrupções podem ficar pendentes. Uma interrupção no contexto de processadores ARM é tratada como uma exceção que permite ao projetista adicionar código em um *handler* específico.

3.1.1 Modos e características de exceção

Na arquitetura ARM Cortex-M, o tratamento de uma interrupção pode ser síncrono ou assíncrono. O tratamento de uma exceção síncrona é executada no momento em que uma instrução de máquina ou uma alteração em um pino lógico dispara a interrupção. A interrupção assíncrona dispara o tratamento depois de uma série de

comandos do microprocessador após o comando de disparo. É descrito no Quadro 1 o estado de execução de tratamento de uma exceção disparada por uma interrupção.

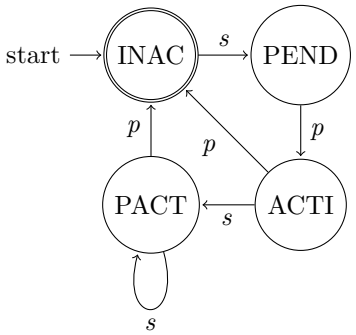
Quadro 1 – Estados de uma exceção do ARM Cortex-M4

Estado	**Descrição**
Inativo	A interrupção não está ativa, ou seja, não está em execução tampouco pendente.
Pendente	A interrupção solicitou atendimento, mas não iniciou execução, ou seja, o tratamento não foi iniciado.
Ativo	A interrupção executa o código de tratamento, mas não finalizou o atendimento. Devido ao tratamento de interrupções de alta prioridade é possível a exceção ser interrompida neste estado e aninhamento de exceções.
Ativo e Pendente	Possível em interrupções assíncronas. Ocorre quando uma interrupção é disparada de uma mesma fonte enquanto a exceção está ativa.

Fonte – ARM Holdings (2011)

É apresentado na Figura 4 o comportamento lógico dos estados de uma exceção em um autômato . A transição *s* simboliza o sinal de interrupção que o processador recebe, já a transição *p* simboliza o final de tratamento do estado, ou seja, o momento em que a CPU atende uma exceção ou finaliza a execução. Os estados, como descritos anteriormente são representados por *ACTI* (Ativo), *INAC* (Inativo), *PACT* (Ativo e Pendente) e *PEND* (Pendente).

Figura 4 – Autômato de estados da exceção e suas transições



Fonte – Autor

Cada exceção possui um número único e imutável que significa uma posição no vetor NVIC e cada interrupção possui uma prioridade, que pode ser modificada ou não a depender do tipo de exceção a ser tratada. A política de atribuição de prioridades é definida pela ordem decrescente, isso significa que uma interrupção que tenha prioridade 0 é mais prioritária que uma interrupção que tenha prioridade 1 e assim por diante.

O Quadro 2 detalha os tipos de exceção, em ordem de prioridade. As interrupções descritas no Quadro 2 são referentes a erros de lógica, semântica, proteção de memória ou *reset* no decorrer de execução de um programa na CPU.

Quadro 2 – Exceções do ARM Cortex-M não vinculadas a um *SOTR*

Exceção	Descrição
Reset	Esta exceção aponta o 'entry point' do programa.
NMI	Uma interrupção que não pode ser mascarada, utilizada para lidar com funções críticas ao microprocessador.
HardFault	Interrompe a execução na existência de um erro genérico que não é tratado por outra exceção.
MemManage	Ocorre quando uma falta é relatada pelo mecanismo de proteção de memória.
BusFault	Ocorre na falha na busca de uma instrução de processador ou algum dado corrompido.
UsageFault	Ocorre na falha lógica ou semântica no microprocessador, como uma divisão por zero, uma instrução indefinida, etc.
SVCall	Uma exceção que é disparada pela instrução SVC do microprocessador. É utilizada para acessar funções de *kernel* e dispositivos.
PendSV	Programado especificamente para uso em trocas de contexto entre tarefas na aplicação de usuário.
SysTick	Um timer de hardware específico que, ao chegar em zero, dispara esta exceção. É programado para ser utilizado como marcação de tempo em um SOTR, ou seja, um *tick*.
***Interrupts* (IRQ)**	Uma ou mais exceções que podem ser disparada(s) por periféricos ou sinais enviados ao microprocessador, tendo sua ativação feita de forma assíncrona.

Fonte – ARM (2011)

Sobre as exceções discutidas nos quadros anterior, é possível inserí-las em dois tipos de categorias (ARM Holdings, 2011):

- ***Fault Handlers***: Todas as exceções que lidam com erros ou falhas, sendo elas *HardFault*, *MemManage*, *BusFault* e *UsageFault*;
- ***System Handlers***: Todas as exceções que são ativadas via software, sendo elas: *NMI*, *PendSV*, *SVCall* e *SysTick*, assim como todas as *Fault Handlers*.

É apresentado na Tabela 1 o número, a posição e a prioridade no NVIC que cada exceção possui. Cada exceção possui um endereço de *offset*, que é o implementado em memória e as reservadas são utilizadas por outras funções do microprocessador e que não estão visíveis ao projetista.

O microprocessador tem suporte a 256 interrupções, das quais 240 interrupções independentes programáveis, no entanto, é possível agrupar interrupções por prioridade. O microprocessador possui 8 bits para essa finalidade, que podem ser divididos em duas partes: Os bits mais significativos representam a prioridade e os menos significativos a sub-prioridade. Na ocorrência de duas interrupções de mesma prioridade, os bits de sub-prioridade definem qual interrupção será tratada primeiro, nesse caso,

Tabela 1 – Detalhamento de exceções do *Cortex-M*

Número de exceção	**Exceção**	**Prioridade**	**Ativação**	**Endereço de *offset***
1	*Reset*	-3	Assíncrono	0x00000004
2	NMI	-2	Assíncrono	0x00000008
3	*HardFault*	-1	Assíncrono	0x0000000C
4	*MemManage*	Configurável	Síncrono	0x00000010
5	*BusFault*	Configurável	Assíncrono/Síncrono	0x00000014
6	*UsageFault*	Configurável	Síncrono	0x00000018
7-10	Reservado	-	-	-
11	*SVCall*	Configurável	Síncrono	0x0000002C
12-13	Reservado	-	-	-
14	*PendSV*	Configurável	Assíncrono	0x00000038
15	*SysTick*	Configurável	Assíncrono	0x0000003C
16	*Interrupts IRQ*	Configurável	Assíncrono	0x00000040

Fonte – ARM (2011)

os bits de menor valor. Na linha *Cortex-M*4, os 4 bits mais significativos são os bits de prioridade e os 4 menos significativos são relacionados à sub-prioridade.

Em exceções, o microprocessador proporciona ao SOTR uma forma de ter acesso a recursos privilegiados. Todas as vezes em que uma exceção está tratando um evento disparado por uma interrupção, o microprocessador entra em *Handler Mode*, ou seja, possui acesso privilegiado a recursos do microprocessador. Esse acesso privilegiado proporciona ao projetista a manipulação de registradores especiais que somente podem ter seu valor alterado nesse modo de operação, ao contrário do *Thread Mode*, que por padrão, não pode ter acesso a esse tipo de privilégio. Quando executando uma tarefa programada pelo usuário o SOTR deve executar em *Thread Mode*.

3.1.2 Latência de interrupção

Em um contexto ideal o desejado seria que a interrupção disparasse uma exceção que fosse tratada imediatamente, respeitando a sua prioridade entre as demais pela CPU. Na prática isso não ocorre devido ao processo de *stacking* e *unstacking* na pilha de registradores, que é um fator a ser considerado em uma análise temporal de uma exceção, o que temporalmente é chamado de latência. Para compreender os motivos de seu efeito, é necessário compreender a finalidade dos registradores utilizados pela linha ARM Cortex-M.

Com um banco de dezessete registradores com 32 *bits* cada, o microprocessador destina treze deles a uso geral pela aplicação (**R0**-**R12**), os oito primeiros, de zero a sete são chamados de *low registers* e os cinco últimos, de oito a doze, são

chamados de *high registers*, os três próximos registradores da pilha (R13-R15) são utilizados para controle de aplicação e os quatro últimos, os registradores especiais (PSR, PRIMASK, FAULTMASK, BASEPRI, CONTROL) são de uso privilegiado.

O registrador treze (**R13**), também nomeado de *Stack Pointer* tem a finalidade de lidar com ponteiros de memória de forma privilegiada ou não, portanto, é dividido em dois: *Main Stack Pointer* (MSP), de acesso privilegiado, utilizado depois da ocorrência de uma exceção do tipo *Reset* ou quando o processador se encontra em *Handler Mode* e o *Process Stack Pointer* (PSP), utilizado apenas em *Thread Mode*. Em sistemas simples com a ausência de um SO embarcado, não é necessário a utilização do MSP, geralmente, R13 é usada para separar a pilha de memória da tarefa e a pilha de memória do *microkernel*.

O registrador catorze (**R14**), também denominado como *Link Register* (LR) tem a função de armazenar o endereço de memória quando há um desvio de fluxo por chamada de função ou sub-rotina na aplicação, e ao fim da execução da função ou sub-rotina o LR repassa o endereço armazenado ao registrador quinze.

O registrador quinze (**R15**) é o *Program Counter* (PC), que armazena o endereço de instrução que será executada pelo microprocessador.

Os *Program Status Registers* (PSR) são uma combinação de três registradores que armazenam os estados de interrupção, aplicação e execução. O registrador *Priority Mask Register* (PRIMASK) armazema uma máscara lógica que previne a ativação de todas as interrupções, excetuando a NMI. Também o *Fault Mask Register* (FAULTMASK) previne a ativação de *Fault Handlers* adicionando uma máscara lógica, excetuando a NMI. O *Base Priority Mask Register* (BASEPRI) serve para prevenção de ativação de interrupções com prioridade menor que as configuradas. Finalmente o registrador *CONTROL Register* (CONTROL) gerencia os privilégios de aplicação e o acesso às pilhas de memória.

O ARM Cortex-M4 possui uma latência de doze ciclos de *clock* quando não ocorre um *zero wait state memory* (YIU, 2014). Portanto, com essa premissa é possível considerar um *overhead* nos cálculos de tempo de resposta quando uma tarefa se encontra implementada em uma exceção.

3.2 MEMÓRIA

Em um SOPG, devido à complexidade de manipulação de memória por processos é comum ao uso de um módulo de gerência de memória, a *Memory Management Unit* (MMU). Sua função é traduzir os endereços lógicos para físicos utilizando uma memória *cache* embutida, denominada *Translation Lookaside Buffer* (TLB). Para sistemas menores e SOTR que buscam menor variação temporal ou até mesmo maior simplicidade, apenas um módulo da MMU é utilizado que é a *Memory Protection Unit* (MPU), sendo o seu uso opcional na linha ARM Cortex-M. Sua função é proteger a memória

de acessos em regiões de memória que excedam os limites pré-estabelecidos.

Em aspectos de capacidade de memória a maior possível é de 4GB que é resultante do tamanho da palavra suportada pela arquitetura, que é de 32 *bits*. A subseção a seguir apresenta um maior detalhamento da estrutura de memória de um microprocessador ARM Cortex-M4.

3.2.1 Mapeamento e estrutura de memória

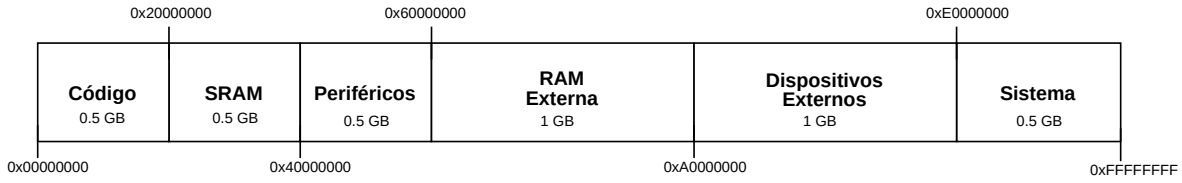
Para discutir sobre aspectos de memória é sempre necessário conhecer como é o seu mapeamento lógico, haja vista que ela fisicamente é um bloco monolítico. O microprocessador divide a memória em seis regiões lógicas distintas e não-estáticas (ARM Holdings, 2011), sendo elas, em ordem de endereçamento:

- **Código**: Espaço reservado para código executável presente no microprocessador, permitindo também acesso de dados;
- ***SRAM***: Espaço reservado para conexão com uma *Static Random-Access Memory* (SRAM), podendo também executar código executável a partir da expansão da região de código;
- **Periféricos**: Espaço reservado para conexão com periféricos acoplados ao microprocessador;
- ***RAM* Externa**: Espaço reservado para código executável e alocação de pilhas de memória de tarefas;
- **Dispositivos Externos**: Espaço reservado para periféricos não acoplados por padrão ao microprocessador;
- **Sistema**: Espaço reservado para controle de sistema, acesso a periféricos como o próprio NVIC e MPU e áreas de depuração.

É apresentado na Figura 5 as regiões de código e seus endereçamentos de memória, a partir do manual ARM Holdings (2011), supondo um uso de memória de 4GB.

Para uso das regiões de memória disponíveis pelo microprocessador, os dados devem seguir um padrão compreensível e uma estrutura que seja suportada por ele é necessária. Em muitos dispositivos os dados podem estar em *little endian* ou *big endian*, o microprocessador suporta as duas formas de ordenação dos *bits* sendo que a ordenação padrão segue o formato *little endian*. No entanto, é necessário especificar o padrão desejado ao evento de *reset* do microprocessador (YIU, 2014). Isso significa que se algum periférico ou módulo do sistema utilizar alguma outra forma de ordenação distinta da que o sistema iniciou é necessária uma conversão de dados para o padrão

Figura 5 – Mapeamento de memória ARM Cortex-M4



Fonte – (ARM, 2011)

de ordem que o microprocessador utiliza, não sendo suportada a variação deles no decorrer da execução do sistema.

Alguns microprocessadores, assim como o ARM Cortex-M4, suportam o particionamento de uma palavra de 32 *bits* (4 *bytes*) em blocos de tamanho de 1 *byte* ou de 2 *bytes* em transferência no barramento de memória. Tais blocos podem estar separados por toda a região da memória, a isto é chamado **modo desalinhado**. Já para uma forma em que o endereço de memória de um determinado dado é múltiplo de seu tamanho (em *bytes*) é dado o nome de **modo alinhado** de transferência. Neste modo, uma palavra por exemplo pode estar presente no endereço 0x00000000, 0x00000004, já uma meia-palavra (*half word*), pode ocupar 0x00000000, 0x00000002, 0x00000004 e assim por diante (YIU, 2014). O ARM Cortex-M4 suporta apenas o modo desalinhado para acesso em *Thread Mode*.

Em algumas operações em um dado presente na memória é necessária apenas a alteração de um *bit* em uma palavra. Em um modo tradicional e pouco eficiente, o projetista pode carregar o dado presente da memória em um registrador de propósito geral, alterar o *bit* presente na palavra e realizar uma operação de *store* no antigo endereço. Para tanto, o microprocessador implementa uma estrutura de manipulação de dados em memória denominada *bit-band* em que para algumas regiões de memória (o primeiro 1 MB) SRAM e Periféricos é possível realizar operações de *load* e *store* em endereços para apenas um *bit* de um dado em um endereço. Isso possibilita implementações de protocolos seriais para *General-Purpose Input/Output* (GPIO) (YIU, 2014).

3.3 RESUMO

Neste capítulo foram discutidos aspectos de hardware relevantes ao comportamento temporal de um SOTR, inclusive, foram discutidas as principais características de um processador ARM Cortex-M4 como por exemplo a sua política de interrupções através do NVIC. No capítulo também foi discutido como o microprocessador trabalha com suas unidades de memória e a sua limitação em 4GB por conta da palavra de

32 *bits*. Foi mostrado o mapeamento, com as seis regiões fixas de memória (Código, SRAM, Periféricos, RAM Externa, Dispositivos Externos e Sistema) e a estrutura de dados em memória.

Sobre aspectos de arquitetura que são relevantes na influência para os tempos de resposta de uma tarefa, tudo o que foi discutido no capítulo pode-se resumir na Tabela 2.

Tabela 2 – Características de um microprocessador ARM Cortex-M4

Estrutura de hardware	**Característica**
Pipeline	Com três estágios, possuindo um *branch predictor* acoplado.
Memory Protection Unit	Implementa proteção de memória opcional em oito regiões distintas.
Taxa de frequência	Variando de 1MHz até 180MHz.
Memória	Suporte a até 4 GB. A fabricante não implementa no modelo mecanismo de memória *cache*.
Interrupções	*Suporte a até 240 interrupções, com 256 níveis de prioridade e opcionalmente implementação de sub-prioridades.*
Interrupções para uso em um SOTR	Implementados em *SysTick* para marcação de tempo (*tick*) e *PendSV*, para troca de contexto.
Ponto Flutuante	Opcional, seguindo a norma (ANSI/IEEE, 1985).
Registradores	Dezessete registradores, sendo treze de uso geral.

Fonte – ARM (2011)

4 *FREERTOS*

Neste capítulo serão discutidos os principais conceitos sobre o *FreeRTOS* em sua versão 10. O *FreeRTOS* é um SOTR de código aberto com licença MIT, programado em linguagem C e em *assembly*.

4.1 ORGANIZAÇÃO DE TAREFAS

Para controle de uma tarefa em qualquer SO é preciso que uma série de informações sobre ela estejam disponíveis para acesso do escalonador em tempo de execução. Em um SOPG é implementado um bloco descritor de processo, chamado de *Process Control Block* (PCB), que embora seja diferente entre os Sistemas Operacionais de Propósito Geral existentes possui informações em comum referentes ao estado de uma *thread*, como prioridade, tempo de execução, endereço de pilhas de memória, valores de registradores, conexões com dispositivos de Entrada/Saída e ponteiros para arquivos. O escalonador do SO utiliza dessas informações de uma *thread* para o escalonamento seguindo a política desejada pelo sistema. Em um SOTR o conceito de descrição de tarefas, chamado de *Task Control Block* (TCB), é semelhante, porém, como os módulos do *microkernel* são mais simples é necessária uma menor quantidade de informações que um SOPG tradicional.

Em um *microkernel* de tempo real cada tarefa possui um TCB distinto e que pode ser movido entre listas utilizadas pelo escalonador para armazenar as tarefas no decorrer da execução do sistema, movimentos esses que dependem de estado e de prioridade individual de cada tarefa no instante em que ocorre algum evento interno ou externo ao sistema.

Na criação de uma tarefa no *FreeRTOS* é utilizada uma *Application Programming Interface* (API) chamada de *xTaskCreate*. Uma variante dela pode ser encontrada na documentação como *xTaskCreateStatic* em que o projetista especifica aspectos relacionados a tamanho de memória da tarefa e a sua pilha. Em ambos os casos, é necessário que o usuário informe o tamanho da pilha de memória da tarefa e sua prioridade, com essas informações em mãos, o SOTR consegue organizar a estrutura do TCB de uma tarefa.

4.1.1 Estados de tarefa e *Task Control Block* no *FreeRTOS*

Assim como um processo presente em um SOPG, no *FreeRTOS* é possível que as tarefas também tenham estados, sendo eles o estado pronto (*ready*), estado executando (*running*), estado suspenso (*suspended*) e estado bloqueado (*blocked*). No decorrer da execução de um sistema de tempo real em um microprocessador *single-core*, não é possível que duas tarefas estejam no estado executando, mas é

possível que nenhuma ou mais tarefas estejam em qualquer um dos outros estados (pronto, suspenso e bloqueado). As alterações de um estado de uma tarefa decorrem de eventos na execução do sistema, seja eles por chamadas de função que as fazem se locomover de um estado para outro, sejam pela ocorrência de uma interrupção ou pela ação do escalonador. É apresentada no Quadro 3 a descrição de cada um dos estados pelos quais uma tarefa pode transitar.

Quadro 3 – Estados de uma tarefa no *FreeRTOS*

Estado	Descrição
Ready	O estado indica que a tarefa está apta a ser executada e está concorrendo por recursos de processamento.
Running	Estado em que a tarefa está ocupando os recursos de processamento, este momento indica que ela é a tarefa mais prioritária entre as aptas na disputa pelo microprocessador, ocasião em que deixa o estado *Ready*.
Blocked	Indica que a tarefa está aguardando uma mensagem ou acesso exclusivo a um recurso compartilhado por um *mutex*.
Suspended	Um estado pouco usual em que a tarefa pode se auto-suspender ou suspender outras a partir de chamadas de *vTaskSuspend* e retornar à fila de aptos através de *vTaskResume*.

Fonte – Amazon (2017)

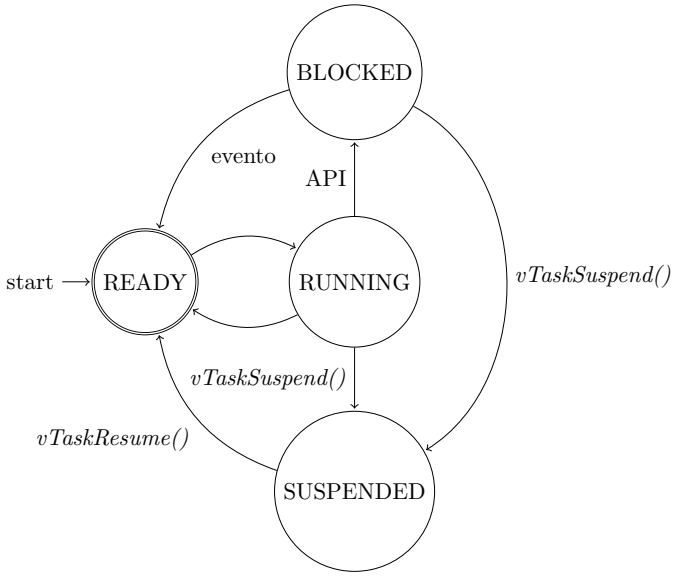
É apresentado no autômato da Figura 6 a lógica de transição entre estados de uma tarefa no *FreeRTOS*. A transição entre o estado pronto (*ready*) e o estado executando (*running*) se dá por ação direta e exclusiva do escalonador, sendo que uma ação de transição entre elas depende da prioridade efetiva da tarefa no momento. Para acesso ao estado bloqueado é necessária uma chamada a alguma API que faça com que a tarefa aguarde uma mensagem com ou sem *timeout* definido ou tenha acesso impedido a um recurso alocado a outra tarefa. Já para deixar o estado bloqueado (*blocked*) é necessário ocorrer um evento, seja ele a liberação de um recurso desejado por uma tarefa de baixa prioridade, seja pela chegada de uma mensagem ou a ocorrência de um *timeout*. O estado suspenso ocorre única e exclusivamente por ação das tarefas presente no sistema através da função *vTaskSuspend()* e sua saída do estado suspenso ocorre pela função *vTaskResume()*.

O TCB de uma tarefa no *FreeRTOS*, representado pela estrutura *TCB_t* é composto por conexões com a sua pilha de memória, com objetos de sincronização de uso de recursos pelas tarefas. Algumas propriedades da tarefa são opcionais e definidas pelas *flags* relacionadas a:

a) **Arquitetura**: Atrelados a aspectos de gerência de memória e arquitetura do SOTR, *portUSING_MPU_WRAPPERS*, *portSTACK_GROWTH*, *tskSTATIC_AND_DYNAMIC_ALLOCATION_POSSIBLE*, assim como outras *flags* representadas por *configNUM_THREAD_LOCAL_STORAGE_POINTERS*, *configUSE_NEWLIB_REENTRANT* e *portCRITICAL_NESTING_IN_TCB*;

b) **Recursos**: *configUSE_MUTEXES*, ligado a recursos compartilhados;

Figura 6 – Autômato de estados da tarefa no *FreeRTOS* e suas transições



Fonte – AWS (2017)

c) **Histórico**: Relacionado à identificação de uma tarefa e seu comportamento no decorrer da execução da tarefa. *configUSE_TRACE_FACILITY* e a *flag configGENERATE_RUN_TIME_STATS* que armazena o tempo no estado *running* da tarefa;

d) **Erro**: Ligada à ocorrência de algum erro na tarefa, representada pela *flag configUSE_POSIX_ERRNO*;

e) **Comportamento**: Uma tarefa pode deixar o estado bloqueado de maneira forçada, representada pela *flag INCLUDE_xTaskAbortDelay*;

f) **Mensagens**: Representado pela *configUSE_TASK_NOTIFICATIONS* para envio de mensagens entre tarefas.

É apresentada na Listagem 4.1 a estrutura mais básica de uma tarefa, com o acréscimo de estruturas de compartilhamento de recursos, devido ao uso em *Response Time Analysis* (RTA).

```
1  typedef struct tskTaskControlBlock {
2      volatile StackType_t  *pxTopOfStack; // Topo da pilha de memoria
3      ListItem_t       xStateListItem; // Pronto, Bloqueado ou Suspenso
4      ListItem_t       xEventListItem; // Evento(s) que esteja relacionado
5      UBaseType_t      uxPriority; // Prioridade da tarefa
6      StackType_t      *pxStack; // Base da pilha de memoria
7      char             pcTaskName[configMAX_TASK_NAME_LEN]; //Nome da tarefa
8
9      #if ( configUSE_MUTEXES == 1 )
10         UBaseType_t    uxBasePriority; // Prioridade original da tarefa
11         UBaseType_t    uxMutexesHeld; // Mutexes utilizados pela tarefa
12     #endif
```

```
13
14      // Outras opcoes...
15      // ...
16  } tskTCB
17
18  typedef tskTCB TCB_t;
```

Listagem 4.1 – TCB básico de uma tarefa implementada no *FreeRTOS*

Em relação à política de prioridades, o *FreeRTOS* é projetado para adotar um número inteiro distinto e não-exclusivo que representa a prioridade da tarefa, de, quanto mais próximo de zero, mais baixa é a sua prioridade e consequentemente quanto mais distante de zero, mais alta a sua prioridade. Não é possível adotar uma prioridade negativa para uma tarefa e a quantidade máxima de prioridades disponíveis no SO é definida em *configMAX_PRIORITIES*.

4.1.2 Gerência de listas de tarefas no *microkernel*

Sistemas Operacionais dispõem de diversas listas para gerenciamento do sistema, um *microkernel* de tempo real como o *FreeRTOS* utiliza estas listas para ordenação de tarefas por ordem de prioridade e de tarefas que aguardam eventos ou mensagens de outras tarefas. A construção desse tipo de estrutura possibilita ao *microkernel* a adoção de políticas de criação, modificação (através de mudança de prioridade) e exclusão de novas tarefas em tempo de execução. Tal flexibilidade impõe ao SOTR que tal estrutura seja implementada como uma lista duplamente encadeada, onde é possível movimentar-se com maior rapidez entre os nós. A Listagem 4.2 apresenta um item de uma lista.

```
1  struct xLIST_ITEM {
2      configLIST_VOLATILE TickType_t xItemValue;
3      struct xLIST_ITEM * configLIST_VOLATILE pxNext;
4      struct xLIST_ITEM * configLIST_VOLATILE pxPrevious;
5      void * pvOwner;
6      struct xLIST * configLIST_VOLATILE pxContainer;
7  }
```

Listagem 4.2 – Item de lista no *FreeRTOS*

Na estrutura, *xItemValue* pode armazenar (ou ficar vazio, *0xFF*) o valor de prioridade ou de *timeout* na espera de um evento. A referência *pxNext* é um ponteiro para o próximo item na lista, já *pxPrevious* aponta para o item anterior. Em *pvOwner*, é acomodado o TCB da tarefa e *pxContainer* é armazenado o endereço da lista ao qual *xLIST_ITEM* pertence. É possível também na estrutura verificar a integridade dos dados inseridos no nó por duas macros inseridas no início e no fim da estrutura, entretanto, é algo opcional e não influencia na lógica de manipulação das listas, sendo, respectivamente, *listFIRST_LIST_ITEM_INTEGRITY_CHECK_VALUE* e também *list-*

SECOND_LIST_ITEM_INTEGRITY_CHECK_VALUE. É apresentada na Listagem 4.3 uma lista implementada através da estrutura *xLIST*.

```
1 typedef struct xLIST {
2   volatile UBaseType_t uxNumberOfItems;
3   ListItem_t * configLIST_VOLATILE pxIndex;
4   MiniListItem_t xListEnd;
5 } List_t;
```

Listagem 4.3 – Lista no *FreeRTOS*

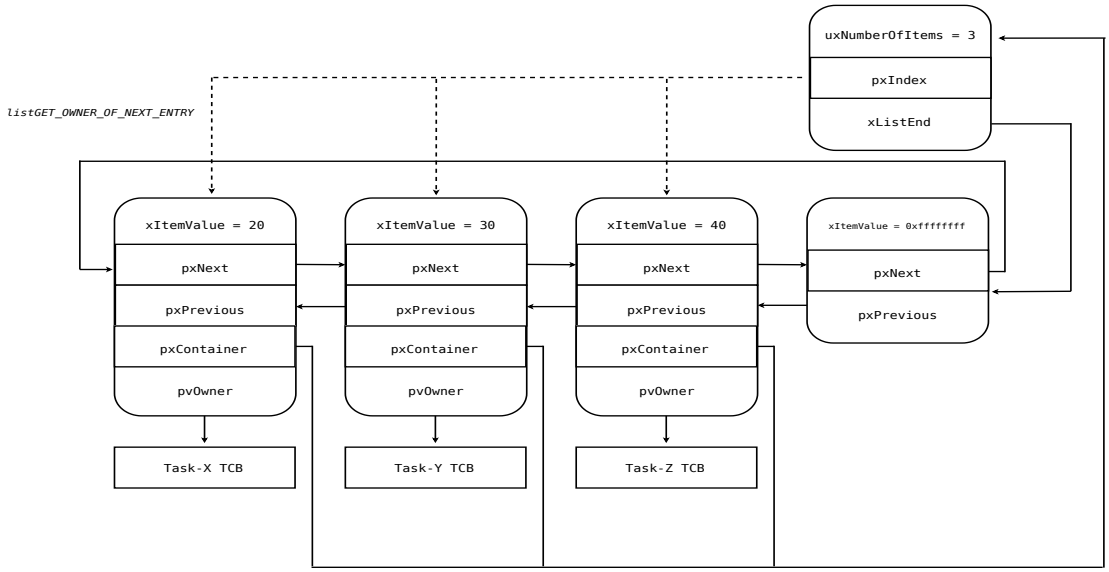
Uma característica das listas no *FreeRTOS* é que elas são feitas de forma lógica circular. Isso significa que a estrutura além de armazenar uma referência para o primeiro nó de uma lista, deve armazenar estruturas que possam controlar e verificar que uma volta foi realizada, para tanto, as variáveis *pxIndex* e *xListEnd* são utilizadas. Em *pxIndex* é localizada a referência da última chamada da função *listGET_OWNER_OF_NEXT_ENTRY*, uma função utilizada para percorrer a lista. Em *xListEnd* é localizado o último nó da lista, sendo utilizado também pelo *microkernel* para indicar que a fila chegou ao fim e o próximo elemento a ser chamado pela macro *listGET_OWNER_OF_NEXT_ENTRY* é o início da lista. Já *uxNumberOfItems* armazena a quantidade de nós presentes em uma lista. O tipo *MiniListItem_t* de *xListEnd* contém a mesma estrutura de *xLIST_ITEM*, excetuando a presença de uma referência a *pvOwner*.

A partir dos estados das tarefas o *FreeRTOS* trabalha com três funções distintas de listas: A lista dos prontos, para as tarefas aptas a disputarem os recursos de processamento no instante de verificação dos elementos presentes nessa lista, as listas de tarefas em atraso, para tarefas que aguardam o seu tempo de liberação ou espera por mensagem (com *timeout*) e uma terceira lista que trabalha com as tarefas que se tornaram aptas no instante em que o escalonador estava suspenso. Quando o escalonador é suspenso, significa que uma tarefa que está utilizando os recursos de processamento não pode ser interrompida por estar lidando com seções críticas. Esse mecanismo é diferente de desativar as interrupções, afinal, quando apenas o escalonador é suspenso, podem ocorrer interrupções programadas pelo hardware. Para que o instrumento de suspensão de escalonamento possa funcionar, é necessária a chamada de função *vTaskSuspendAll* e o retorno ao funcionamento do escalonador se deve a chamada de *vTaskResumeAll*.

Para tarefas em espera de liberação, existem duas listas distintas de armazenamento. No instante de inserção na fila de tarefas em atraso, se a tarefa não perdeu o seu *deadline*, ela será armazenada em *pxDelayedTaskList*, caso contrário, ela será armazenada em *pxOverflowDelayedTaskList*. Isso se dá a partir da chamada de alguma API de controle de tarefas, como a espera de uma mensagem, funções de *delay* ou de um *lock* em um *mutex* ocupado por uma tarefa. As listas são ordenadas, do menor valor para o maior valor de *xItemValue* , em *xItemValue* é armazenado o tempo

do instante de liberação da tarefa. Em casos de tempo de liberação infinito, como a espera de liberação de um *mutex* por uma tarefa de mais baixa prioridade, é possível também inserir um tempo indefinido, representado pela macro *portMAX_DELAY*. É apresentado na Figura 7 o cenário de três tarefas distintas aguardando sua liberação a partir de *pxDelayedTaskList*, com períodos de liberação de *Task-X, Task-Y, Task-Z*, com 20, 30 e 40 unidades de tempo respectivamente.

Figura 7 – Estrutura lógica de *pxDelayedTasksList*



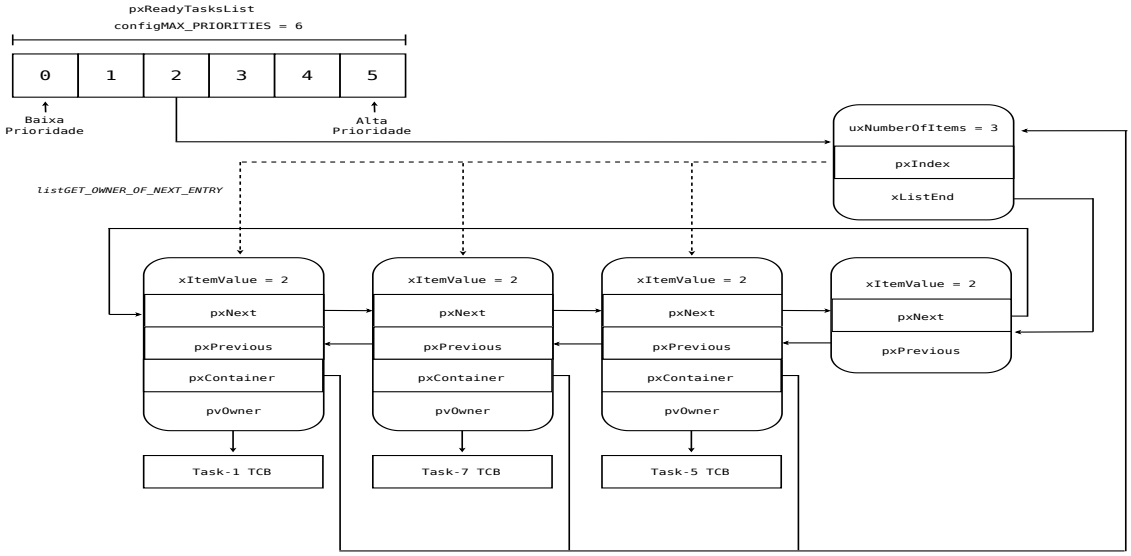
Fonte – Autor

A lista dos prontos trabalha com um vetor de *configMAX_PRIORITIES* posições. Cada posição no vetor representa uma lista de tarefas com mesma prioridade que estão aptas a disputar os recursos de processamento. Esse mecanismo permite que o *microkernel* faça uma varredura mais veloz ao invés de comparar as tarefas uma a uma ou implementar uma fila única que deve ser organizada a cada chegada de uma nova tarefa apta. O primeiro elemento da lista em uma posição é sempre a com maior preferência dentre as tarefas de mesma prioridade. No SOTR a implementação da lista de tarefas prontas é representada pelo vetor *pxReadyTasksList[configMAX_PRIORITIES]*.

É apresentada na Figura 8 a estrutura lógica da lista de tarefas prontas para uma prioridade.

Para o caso de *uxNumberOfItems* maior que 1 numa posição do vetor de listas de prioridade, o *microkernel* adota uma política de justiça em tempo de ocupação, utilizando para tanto o algoritmo *round-robin*, como no caso apresentado pela Figura 8. O *quantum* ou fatia de tempo padrão fornecida pelo algoritmo no SOTR é de dois *ticks*. Como o período de cada *job* de um *Tick* é configurável através das macros

Figura 8 – Estrutura lógica de tarefas prontas



Fonte – Autor

portTICK_PERIOD_MS e *configTICK_RATE_HZ*, é possível alterá-lo de acordo com os requisitos de projeto.

As funções de manipulação das tarefas do *microkernel* estão disponíveis no arquivo *list.h*, de uso privilegiado ao SOTR no uso de uma MPU. Além das funções de inicialização, mais duas entram em destaque, a função *vListInsert* e a função *vListInsertEnd*. A Listagem 4.4 mostra todas as funções presentes no arquivo, que são privilegiadas ao SOTR.

```
1  #define listLIST_IS_INITIALISED(pxList)  // Verifica se lista esta inicializada
2  #define listSET_LIST_ITEM_OWNER(pxListItem,pxOwner) // Coloca o TCB em uma posicao de lista
3  #define listGET_LIST_ITEM_OWNER(pxListItem) // Obtem o TCB em uma posicao de lista
4  #define listSET_LIST_ITEM_VALUE(pxListItem, xValue) //Coloca valor em item na lista
5  #define listGET_LIST_ITEM_VALUE(pxListItem) //Obtem um valor em item na lista
6  #define listGET_ITEM_VALUE_OF_HEAD_ENTRY(pxList) //Obtem o primeiro valor em item na lista
7  #define listGET_OWNER_OF_NEXT_ENTRY(pxTCB, pxList) //Obtem a proxima tarefa de lista
8  #define listGET_OWNER_OF_HEAD_ENTRY(pxList) //Obtem a primeira tarefa de uma lista
9  #define listIS_CONTAINED_WITHIN(pxList, pxListItem) //Verifica se a lista contem um item
10 #define listGET_HEAD_ENTRY(pxList) //Retorna o primeiro elemento da lista
11 #define listGET_NEXT(pxListItem) //Obtem o proximo item de um inicio de lista
12 #define listGET_END_MARKER(pxList) //Obtem o ultimo item de uma lista
13 #define listLIST_IS_EMPTY(pxList) //Verifica se lista esta vazia
14 #define listCURRENT_LIST_LENGTH(pxList) //Obtem o tamanho da lista
15
16 // Inicializa uma lista
17 void vListInitialise(List_t *const pxList) PRIVILEGED_FUNCTION;
18
19 // Inicializa um item de uma lista
20 void vListInitialiseItem(ListItem_t *const pxItem) PRIVILEGED_FUNCTION;
21
22 // Insere um item em uma lista (em ordem crescente)
23 void vListInsert(List_t *const pxList,ListItem_t *const pxNewListItem) PRIVILEGED_FUNCTION;
```

```
24
25 // Insere um item ao final uma lista
26 void vListInsertEnd(List_t *const pxList,ListItem_t *const pxNewListItem)
       PRIVILEGED_FUNCTION;
27
28 // Remove um item de uma lista
29 UBaseType_t uxListRemove(ListItem_t *const pxItemToRemove) PRIVILEGED_FUNCTION;
```

Listagem 4.4 – Funções de manipulação de lista presentes em *list.h*

4.1.3 Tarefas periódicas

Para o cenário básico de execução de uma tarefa periódica do FreeRTOS é importante conhecer quais funções do *microkernel* estão envolvidas. No desenvolvimento de uma tarefa periódica, a forma mais intuitiva de se implementar seria adicionar, dentro de um laço infinito, um *delay* até a próxima ativação, como demonstrado na Listagem 4.5.

```
1 for(;;){
2     funcaoA();
3     funcaoB();
4     delay(1000); // Delay de 1 segundo
5 }
```

Listagem 4.5 – Lógica de uma tarefa periódica

Assumindo-se um cenário hipotético de uma tarefa qualquer que execute uma função que tem um WCET de 2 milissegundos, que possui um período de 3 milissegundos e partindo do instante zero em tempo real. O *FreeRTOS* possui duas API's de atraso de uma tarefa: *vTaskDelay* e *vTaskDelayUntil*. A função *vTaskDelay* possui como argumento em sua função a variável *xTicksToDelay* em que o valor inserido depende da granularidade temporal definida nas macros do *FreeRTOS*. Considerando também a granularidade como 1 milissegundo, qualquer valor inserido é o próprio número em milissegundos. É apresentada na Listagem 4.6 a implementação da lógica no contexto do *FreeRTOS*.

```
1 void v1Task1_Handler(void *params){
2     for(;;){
3         funcaoA();
4         vTaskDelay(3);
5     }
6 }
```

Listagem 4.6 – Tarefa com *delay* no *FreeRTOS*

A tarefa executa um *delay* de 3 milissegundos, no entanto, em termos absolutos a tarefa será ativada apenas após o término da função. Considerando um cenário de concorrência em que essa tarefa divide recursos de processamento com outras tarefas no SOTR e até mesmo da própria variação temporal que é fruto da execução da função A, do ponto de vista temporal o momento da ativação de *v1Task1_Handler*

será imprevisível. Considere P(τ) como o instante de ativação de uma tarefa. Utilizando um WCET da tarefa implementada como 2 milissegundos, tem-se o seguinte registro de ativações:

- P(τ): 0 milissegundos. A função A executa por 2 milissegundos e a tarefa é suspensa.
- P(τ): 5 milissegundos. A função A executa por 2 milissegundos e a tarefa é suspensa.
- P(τ): 10 milissegundos. A função A executa por 2 milissegundos e a tarefa é suspensa.

E assim sucessivamente. Portanto, *vTaskDelay* possui um comportamento de atraso relativo, ou seja, a tarefa depende do instante de chamada da função *vTaskDelay* para o cálculo do próximo instante de ativação. Este cenário que não é desejado para análise, devido à sua imprevisibilidade de comportamento temporal, dado que a tarefa pode sofrer interferências e bloqueios no decorrer da sua execução no sistema.

A função *vTaskDelayUntil* possui dois argumentos: *pxPreviousWakeTime* e *xTimeIncrement*, onde o primeiro se refere ao tempo relativo à ativação anterior no qual se deseja que a tarefa seja ativada. Esse valor é dado pelo próprio *microkernel* ao se ativar a função, e é atualizado conforme *vTaskDelayUntil* é invocado. Já *xTimeIncrement* é o tempo absoluto em que se deseja que a tarefa seja ativada, sendo que essa variável é utilizada para a atualização de *pxPreviousWakeTime*. É apresentada na Listagem 4.7 a implementação para o *FreeRTOS*, considerando a lógica anterior e alterando o cenário para a correta execução da função de atraso.

```
1 void v1Task1_Handler(void *params){
2     TickType_t xLastWakeTime;
3     const TickType_t xFrequency = pdMS_TO_TICKS(3);
4     xLastWakeTime = xTaskGetTickCount();
5     for(;;){
6         vTaskDelayUntil(&xLastWakeTime, xFrequency);
7         funcaoA();
8     }
9 }
```

Listagem 4.7 – Tarefa periódica no *FreeRTOS*

Utilizando *vTaskDelayUntil*, o comportamento temporal observado, ao contrário de *vTaskDelay*, é uma ativação respeitando requisito temporal de T igual a 3 milissegundos, independente da variação temporal da função A:

- P(τ): 0 milissegundos. A função A executa por 1,5 milissegundos e a tarefa é suspensa por 3 milissegundos;
- P(τ): 3 milissegundos. A função A executa por 0,6 milissegundos e a tarefa é suspensa por 3ms;

- P(τ): 6 milissegundos. A função A executa por 1,68 ms e a tarefa é bloqueada por 3ms.

Como o WCET em hipótese é 2 milissegundos, qualquer valor que varie abaixo disso não influenciará no tempo de liberação da tarefa, *v1Task1_Handler*, ao contrário do que ocorre no uso de *vTaskDelay*. Portanto, a função *vTaskDelayUntil* é uma função que lida com atrasos em tempo absoluto e é a função de atraso adequada para ser utilizada na programação de tarefas periódicas.

No aspecto lógico, em *vTaskDelayUntil*, o escalonador é suspenso antes de executar o código da função, isso se deve porque ela lida com variáveis compartilhadas e listas de tarefas, em que na ocorrência de uma preempção em qualquer momento para um outro *TCB* dentro do *microkernel* a integridade das informações de controle ou o conteúdo das listas poderia ficar comprometida. Embora o escalonador seja suspenso, as interrupções externas não são desativadas. Após a suspensão, o ponteiro que aponta para tarefa é removido da fila de aptos e colocado na lista de tarefas atrasadas, *xDelayedTaskList1* ou *xDelayedTaskList2*, a depender da ocorrência ou não de um *overflow* temporal. Após a *TCB* da tarefa ser colocada nessa lista, o escalonador é reativado.

Neste momento, as interrupções são desativadas, uma seção crítica do *microkernel* é executada e, logo em seguida, as interrupções são novamente religadas. Após essa seção crítica, uma solicitação para que ocorra uma troca de contexto para a tarefa mais prioritária na fila de aptos é efetuada e o escalonador retira os recursos de processamento da tarefa. É apresentado na Listagem 4.8 o código de *vTaskDelayUntil*.

```
1  xTaskDelayUntil(TickType_t * const pxPreviousWakeTime, const TickType_t xTimeIncrement) {
2      TickType_t xTimeToWake;
3      BaseType_t xAlreadyYielded, xShouldDelay = pdFALSE;
4      configASSERT( pxPreviousWakeTime );
5      configASSERT( ( xTimeIncrement > 0U ) );
6      configASSERT( uxSchedulerSuspended == 0 );
7      vTaskSuspendAll(); {
8          const TickType_t xConstTickCount = xTickCount;
9          xTimeToWake = *pxPreviousWakeTime + xTimeIncrement;
10         if( xConstTickCount < *pxPreviousWakeTime ){
11             if((xTimeToWake < *pxPreviousWakeTime) && (xTimeToWake > xConstTickCount)){
12                 xShouldDelay = pdTRUE;
13             } else mtCOVERAGE_TEST_MARKER();
14         } else{
15             if((xTimeToWake < *pxPreviousWakeTime) || (xTimeToWake > xConstTickCount)){
16                     xShouldDelay = pdTRUE;
17                 } else mtCOVERAGE_TEST_MARKER();
18         }
19         *pxPreviousWakeTime = xTimeToWake;
20         if( xShouldDelay != pdFALSE ){
21             traceTASK_DELAY_UNTIL( xTimeToWake );
22             prvAddCurrentTaskToDelayedList( xTimeToWake - xConstTickCount, pdFALSE );
23         }else mtCOVERAGE_TEST_MARKER();
24     }
25     xAlreadyYielded = xTaskResumeAll();
```

```
26      if( xAlreadyYielded == pdFALSE ) portYIELD_WITHIN_API();
27      else mtCOVERAGE_TEST_MARKER();
28      return xShouldDelay;
29  }
```

Listagem 4.8 – Lógica de *vTaskDelayUntil*

Em um cenário de RTA é possível incluir o tempo de execução da função *vTaskDelayUntil* no WCET da tarefa que a usa, de tal forma em que a sua inserção resulta em considerar a manipulação da seção crítica do *microkernel* em *vTaskDelayUntil* que desabilita interrupções, o que pode gerar *release jitters* em interrupções e em outras tarefas.

4.1.4 Tarefa ociosa

A tarefa ociosa (*IDLE*) é criada automaticamente pelo SOTR ao se iniciar a execução do sistema (BARRY, 2016). A causa da criação dessa tarefa para execução no *microkernel* se deve à política de prevenção de falhas do SO, em que é necessário para o correto funcionamento do escalonador que ao menos uma tarefa esteja executando no sistema. Em caso de um limiar de utilização de CPU menor que 100%, em um instante qualquer quando nenhuma tarefa programada pelo projetista estiver executando, a tarefa *IDLE* entra em execução, afinal, possui a mais baixa prioridade possível para alocação de tarefas do sistema (prioridade 0). Ela também é útil para o sistema em controle e gestão de memória, pois, há uma função implementada na tarefa ociosa que deixa livre para novas alocações regiões de memória que estavam alocadas a tarefas que foram excluídas no sistema, contudo, isso só é feito na presença de um algoritmo de gerência de memória que permite uma deleção de blocos e pilhas de memória. É apresentado na Listagem 4.9 o código de implementação da tarefa ociosa para escalonamento preemptivo.

```
1  static portTASK_FUNCTION(prvIdleTask, pvParameters) {
2      ( void ) pvParameters;
3      portALLOCATE_SECURE_CONTEXT( configMINIMAL_SECURE_STACK_SIZE );
4
5      for(;;) {
6          prvCheckTasksWaitingTermination();
7          #if ((configUSE_PREEMPTION == 1) && (configIDLE_SHOULD_YIELD == 1)){
8              if(listCURRENT_LIST_LENGTH(&( pxReadyTasksLists[ tskIDLE_PRIORITY ]))
9              > (UBaseType_t) 1){
10                 taskYIELD();
11             }
12         }
13         #endif
14         #if ( configUSE_IDLE_HOOK == 1 ){
15                 extern void vApplicationIdleHook( void );
16                 vApplicationIdleHook();
17         }
18         #endif
19     }
20 }
```

Listagem 4.9 – Tarefa *idle* implementada no *FreeRTOS*

Como explícito na lógica do pseudocódigo da Listagem 4.9, não é necessário que a tarefa ociosa se preocupe em vistoriar a lista de tarefas prontas em busca de tarefas de mais alta prioridade que ela, uma vez que, na chegada de qualquer outra tarefa prioritária, são retirados da tarefa ociosa os recursos de processamento pelo escalonador. Logo após a verificação da lista de tarefas prontas, a função gancho da tarefa *IDLE* é chamada para execução, se ela tiver sido implementada pelo projetista.

Como a implementação de um tratamento de interrupção, a função de gancho de *IDLE* não deve conter elementos que gerem algum tipo de bloqueio ou suspensão na tarefa, porquanto ela também lida com um aspecto sensível ao gerenciamento de memória do *microkernel* que são desalocações de blocos de memória não utilizados por tarefas que já foram excluídas. Um outro aspecto relevante é que a tarefa ociosa, através da função gancho *vApplicationIdleHook*, não foi projetada para implementar funções de usuário que contenham requisitos periódicos. Por isso, não é possível garantir a confiabilidade de que *jobs* serão ativados ou até mesmo chegarão nos tempos definidos por requisitos de projeto justamente pelo fato de que não é possível utilizar nenhuma estrutura de controle, como um *delay*, de uma tarefa comum (criada por *xTaskCreate*). Deste modo, sob a ótica de análise puramente matemática do modelo RTA e considerando suas premissas, não é possível utilizar a função gancho de *IDLE* para implementar tarefas relevantes ao sistema e que venham a ser consideradas para análise.

O uso dessa tarefa é indicado por Barry (2016) para executar códigos em *background* independentes do sistema, medição de tempo de ociosidade ou colocar a CPU em modo *low-power* para economia de bateria em malhas de controle.

4.1.5 *Timers*

O *microkernel* se utiliza de uma tarefa específica que gerencia todos os *timers* programados por software, chamada de *Tmr Svc*. Os *timers* programados pelo SO são totalmente independentes do hardware e implementados de duas formas: *One-Shot* ou *Auto-Reload*. Em uma forma *One-shot*, o *job* de um *timer* é ativado de forma aperiódica, por sua vez, em um modo de *auto-reload*, um *job* de *Tmr Svc* é ativado periodicamente.

A tarefa *Tmr Svc* utiliza-se de uma fila de serviços enviados para ela por outras tarefas do *microkernel*.

Como comando, entenda-se qualquer API ou função que foi projetada para manipulação de *timers*, por exemplo, *xTimerStart* ou *xTimerStop*, implementados no arquivo *timers.c*. O envio de comandos para a fila implementada em *Tmr Svc* é feito

de forma automática a cada chamada de função, ficando a cargo do projetista adequar o tamanho da fila para os requisitos de projeto. A omissão no cuidado à quantidade de comandos enviados pode gerar um *overflow* na fila, e um comportamento inesperado do sistema pode ocorrer. São apresentados na Listagem 4.10 o código da tarefa *Tmr Svc*.

```
1  static portTASK_FUNCTION( prvTimerTask, pvParameters ){
2      TickType_t xNextExpireTime;
3      BaseType_t xListWasEmpty;
4
5      ( void ) pvParameters;
6
7      #if ( configUSE_DAEMON_TASK_STARTUP_HOOK == 1 ) {
8          extern void vApplicationDaemonTaskStartupHook( void );
9          vApplicationDaemonTaskStartupHook();
10     }
11     #endif
12
13     for(;;){
14         xNextExpireTime = prvGetNextExpireTime( &xListWasEmpty );
15         prvProcessTimerOrBlockTask( xNextExpireTime, xListWasEmpty );
16         prvProcessReceivedCommands();
17     }
18 }
```

Listagem 4.10 – Tarefa *Tmr Svc* implementada no *FreeRTOS*

Uma característica para todos os *timers* implementados no sistema é o compartilhamento de prioridade, em outras palavras, a prioridade é a mesma para todos os *timers* implementados no sistema, isso supondo que a função de *callback* de algum *timer* jamais altere a prioridade de *Tmr Svc* através da API *vTaskPrioritySet*. É apresentado na Tabela 3 os comandos de gerenciamento de *timers* no *FreeRTOS*.

4.2 FILAS

Em Sistemas Operacionais as filas (*queues*) são uma importante estrutura lógica que fornece aos processos e tarefas uma estrutura de comunicação e armazenamento de informações em um espaço reservado, sendo utilizadas inclusive em problemas clássicos da computação como o problema do produtor-consumidor (DIJKSTRA, 1972). Para um SOPG uma fila pode funcionar, por exemplo, como um auxiliar na recepção de informações de tratadores de interrupção originadas por dispositivos de entrada e saída e demais aplicações em que se necessite um armazenamento de dados sequenciais. Já em um *microkernel*, a cada período de tempo pode ser necessário armazenar o conteúdo recebido por um sensor de temperatura para que uma tarefa do sistema possa atuar em uma determinada planta. No *FreeRTOS* uma fila funciona como uma comunicação entre tarefas ou entre uma tarefa e um tratador de interrupção e vice-versa (BARRY, 2016). Para uma fila a ordem de chegada de informações é importante, logo, é intrínseco ao seu funcionamento que os dados sejam sequenciais,

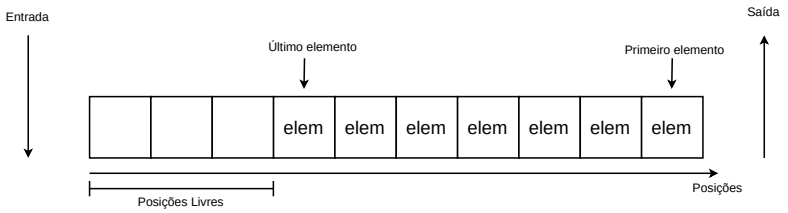
Tabela 3 – Serviços de gerenciamento de *timers* no *FreeRTOS*

Função	**Razão de uso**
xTimerCreate	Cria um novo *timer* de software.
xTimerCreateStatic	Cria um novo *timer* de software com *buffer* estático.
xTimerIsTimerActive	Retorna um booleano *true* se *timer* está ativo.
pvTimerGetTimerID	Retorna um identificador (ID) de um *timer*.
pcTimerGetName	Retorna o nome de um *timer*.
vTimerSetReloadMode	Modifica o *timer* para *auto-reload* ou *one-shot*.
xTimerStart	Inicia a contagem de um *timer*.
xTimerStop	Pausa a contagem de um *timer*.
xTimerChangePeriod	Modifica a periodicidade de um *timer auto-reload*.
xTimerDelete	Remove um *timer*.
xTimerReset	Reinicia um *timer*.
vTimerSetTimerID	Define um identificador (ID) para um *timer*.
xTimerGetTimerDaemonTaskHandler	Retorna *vApplicationDaemonTaskStartupHook*.

Fonte – Amazon (2017)

não sendo possível inserir novos dados entre as posições de início e fim da fila, mantendo assim uma consistência de dados. É apresentada na Figura 9 a estrutura lógica de uma fila com um tamanho de dez posições.

Figura 9 – Estrutura lógica de uma fila



Fonte – Autor

4.2.1 Organização de filas

O *FreeRTOS* não permite mudanças no tamanho das filas no decorrer da execução, mas permite a criação e exclusão de filas por uma tarefa. Em sistemas operacionais os valores para uma fila podem ser enviados por cópia ou por referência. Para o primeiro caso, os valores enviados para uma fila são copiados para uma posição

da fila, já para a segunda, um ponteiro referenciando o valor é repassado para uma posição da fila.

O *microkernel* adota a primeira estratégia, ou seja, envia os dados para a fila através de cópia de valor. Barry (2016) argumenta como razões de escolha dessa estratégia uma maior simplicidade de implementação e maior robustez do mecanismo, prevenindo assim uma referência nula ou implementações de *buffer* auxiliares. Assim, como qualquer SO, um elemento de fila no *FreeRTOS* não possui um tipo definido por padrão, ficando a critério de escolha do projetista criar alguma estrutura (*struct*) em linguagem C que se encaixe no que é requisito para a comunicação de tarefas, logo, o *microkernel* possui apenas controle sobre a estrutura que trabalha com o enfileiramento lógico, sendo representado no código-fonte como *QueueHandle_t*. É apresentada na Listagem 4.11 a estrutura de uma fila, *QueueDefinition*.

```
1  typedef struct QueueDefinition {
2      int8_t *pcHead; // Aponta para o inicio do armazenamento da fila.
3      int8_t *pcWriteTo;// Aponta para o proxima area livre de armazenamento.
4      union { // Reune ...
5          QueuePointers_t xQueue;// Ponteiros de fila (pcTail e pcReadFrom).
6          SemaphoreData_t xSemaphore; // Dados de um mutex xMutexHolder e
       uxRecursiveCallCount
7      }u;
8      // Tarefas que estao aguardando para enviar dados na fila (quando fila cheia).
9      List_t xTasksWaitingToSend;
10     // Tarefas que estao aguardando para receber dados na fila (quando fila vazia).
11     List_t xTasksWaitingToReceive;
12
13     volatile UBaseType_t uxMessagesWaiting; //Quantidade de mensagens na fila.
14     UBaseType_t uxLength; // Tamanho da fila.
15     UBaseType_t uxItemSize;// Quantidade de bytes por posicao.
16     volatile int8_t cRxLock; // Quantidade retirada da fila quando ela estava bloqueada.
17     volatile int8_t cTxLock; // Quantidade inserida da fila quando ela estava bloqueada.
18
19     #if ((configSUPPORT_STATIC_ALLOCATION == 1) && (configSUPPORT_DYNAMIC_ALLOCATION == 1))
20         uint8_t ucStaticallyAllocated; // Retorna se a fila e estaticamente alocada.
21     #endif
22    // Habilita a fila para participar de um conjunto de filas (queue sets).
23     #if (configUSE_QUEUE_SETS == 1)
24         struct QueueDefinition *pxQueueSetContainer;
25     #endif
26     #if ( configUSE_TRACE_FACILITY == 1 )
27         UBaseType_t uxQueueNumber; //Numero de fila (para depuracao).
28         uint8_t ucQueueType;// Tipo de fila (mutex ou fila).
29     #endif
30 }
```

Listagem 4.11 – Estrutura de fila no *FreeRTOS*

Como mostrado na Listagem 4.11, o SOTR acrescenta um mecanismo de *timeout* na inserção e remoção de dados em uma fila, assim como também é possível agrupar filas em conjuntos ou *sets*. Com a adição desses mecanismos na gerência de uma fila é possível utilizá-la para implementação de atividades no *microkernel*. Todavia, consequentemente ao adotar *timeouts* em operações com filas, há a ocorrência de

release jitter nas liberações de tarefas.

Não é possível porém trabalhar com filas que possuem *timeout* em tratamentos de interrupção, sendo construídas funções específicas para se trabalhar com filas (com final *fromISR*), retirando possíveis causas de bloqueio. A Tabela 4 apresenta as funções de gerenciamento de uma fila no *FreeRTOS*.

Tabela 4 – Funções de gerenciamento de fila no *FreeRTOS*

Função	**Razão de uso**
xQueueCreate	Criação de uma fila.
xQueueCreateStatic	Criação de uma fila com memória estática.
xQueueDelete	Exclusão de uma fila.
xQueueSend (fromISR)	Envio de mensagem para o fim de fila.
xQueueSendToBack (fromISR)	Envio explícito de mensagem para o fim de fila.
xQueueSendToFront (fromISR)	Envio explícito de mensagem para o início de fila.
xQueueReceive (fromISR)	Recebimento de mensagem do início da fila.
xQueueMessagesWaiting (fromISR)	Quantidade de mensagens armazenadas em uma fila.
xQueueSpacesAvaliable	Quantidade de espaços disponíveis em uma fila.
xQueueReset	Eliminar todos os elementos da fila.
xQueuePeek (fromISR)	Recebimento de item do início da fila, sem removê-la.
vQueueAddToRegistry	Adicionar um nome para uma fila, para depuração.
pcQueueGetName	Retorna o nome da fila.
vQueueUnregisterQueue	Remove o nome de uma fila.
xQueueOverwrite	Sobrescreve o último item da fila.
xQueueIsQueueEmptyFromISR	Verifica se fila está vazia (exclusivo para ISR).
xQueueIsQueueFullFromISR	Verifica se fila está cheia (exclusivo para ISR).

Fonte – Amazon (2017)

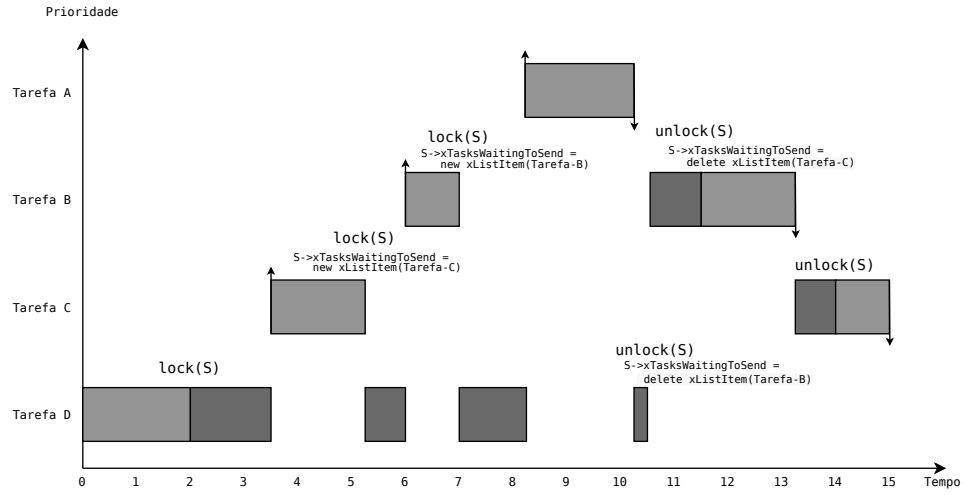
4.2.2 Recursos compartilhados

O *FreeRTOS* trabalha com filas para a construção de mecanismos que implementam recursos compartilhados. Isso é evidenciado pelo tipo criado para recursos compartilhados, pois, *SemaphoreHandle_t* é apenas um invólucro da função gestora de uma fila, *QueueHandle_t*, pois, ela implementa uma lista interna de tarefas que são bloqueadas ao executar um *lock* no semáforo e necessitam refazer novamente uma tentativa de *lock* a partir da liberação do recurso. Em uma seção crítica, é comum em linguagem de máquina utilizar programação com instruções atômicas para implementação de semáforos, como para processadores *multi-core*. Para sistemas embarcados, é comum em regiões críticas desabilitar interrupções para evitar alguma possível corrupção de dados devido à simplicidade de implementação em mono-processadores.

Há uma série de mecanismos que auxiliam na construção de seções compartilhadas, como soluções puramente baseadas em software com o algoritmo de Peterson (1981) e a mescla de soluções de software e instruções atômicas de hardware, como o *spin-lock*. Ambos possuem suas vantagens e desvantagens, por exemplo, o primeiro possui como vantagem a portabilidade e como desvantagens o *busy-waiting* e a complexidade de implementação, já o segundo possui como vantagem a simplicidade e como desvantagem o *busy-waiting*.

Em recursos compartilhados, além da estratégia de desativar interrupções, como evidenciado na Listagem 4.9, é utilizado *SemaphoreHandle_t* para gestão de seções críticas no *microkernel* para sincronização de tarefas. É apresentado na Figura 10 um diagrama temporal com quatro tarefas que disputam um semáforo (*S*) que controla acesso a uma região compartilhada.

Figura 10 – Diagrama de tempo de *mutexes* no *FreeRTOS*



Fonte – Autor

Neste cenário, a *Tarefa-D* executa um *lock* no recurso *S* que está livre, em seguida a *Tarefa-C* realiza a mesma operação de *lock* em *S*, que já está ocupado. Nesse caso, a *Tarefa-C* é inserida na lista de *xTasksWaitingToSend* e a *Tarefa-D* herda a prioridade de *Tarefa-C*. Isso significa que o semáforo programado no *microkernel* utiliza-se do protocolo PIP como protocolo padrão de gerenciamento de recursos compartilhados. Em seguida, a *Tarefa-B* realiza o mesmo *lock* em *S*, repassando sua prioridade para a *Tarefa-D*. Terminando sua execução, *Tarefa-D* devolve o recurso, desbloqueando a *Tarefa-B* e em seguida, após o uso da seção crítica pela *Tarefa-B*, a *Tarefa-C* conclui sua execução.

Além dos *mutexes* implementados e do mecanismo de desabilitação de interrupções, o *microkernel* também fornece opções ao projetista de proteger seções críticas com semáforos binários e contadores, semáforos recursivos.

A opção de suspender o escalonador, isto é, permitir que apenas interrupções sejam atendidas e não pedidos de troca de contexto entre tarefas, que no caso de ocorrerem ficam pendentes (*pend*), afinal, é sempre recomendado em um SO que o tempo que as interrupções ficam desabilitadas seja o menor possível (OLIVEIRA, CARISSIMI e TOSCANI, 2010). São apresentadas na Tabela 5 as APIs que são utilizadas para implementação de recursos compartilhados.

Tabela 5 – Funções de gerenciamento de seções críticas no *FreeRTOS*

Função	**Razão de uso**
xSemaphoreCreateBinary	Cria um semáforo binário.
xSemaphoreCreateBinaryStatic	Cria um semáforo binário com memória estática.
xSemaphoreCreateCounting	Cria de um semáforo contador.
xSemaphoreCreateCountingStatic	Cria um semáforo contador com memória estática.
xSemaphoreCreateMutex	Cria um *mutex*.
xSemaphoreCreateMutexStatic	Cria um *mutex* com memória estática.
xSemaphoreCreateRecursiveMutex	Cria um *mutex* recursivo.
xSemaphoreCreateRecursiveMutexStatic	Cria um *mutex* recursivo com memória estática.
vSemaphoreDelete	Apaga um semáforo.
xSemaphoreGetMutexHolder	Indica se a tarefa está com a posse de um *mutex*.
xSemaphoreTake (*FromISR*)	Realiza um *lock* em um *mutex*.
xSemaphoreTakeRecursive	Realiza um *lock* em um *mutex* recursivo.
xSemaphoreGive (FromISR)	Devolve a posse de um *mutex*.
uxSemaphoreGetCount	Retorna o valor de um semáforo contador.
taskENTER_CRITICAL	Desabilita interrupções em uma tarefa.
taskEXIT_CRITICAL	Habilita interrupções em uma tarefa.
vTaskSuspendAll	Pausa o escalonador.
vTaskResumeAll	Inicia o escalonador.

Fonte – Amazon (2017)

4.3 *OVERHEADS* DE SISTEMA

Como discutido no Capítulo 3, o *FreeRTOS* faz uso de algumas exceções para implementar tarefas e módulos de gerenciamento de tarefas, mais especificamente as do grupo *System Handlers*. Esta seção tem como o objetivo a discussão dos módulos envolvidos no escalonamento de tarefas, medição da passagem do tempo e suas influências temporais em uma tarefa.

4.3.1 Chaveamento de contexto

Ao efetuar um procedimento de troca de contexto, o *microkernel* ativa uma interrupção tratada na exceção *PendSV* do ARM Cortex-M4. Em um tipo de escalonamento colaborativo, o projetista pode ativar a exceção a partir de uma tarefa programada por ele no SOTR através das macros *portYIELD*, *portYIELD_WITHIN_API* ou em um contexto de tratamento de interrupção pela macro *portYIELD_FROM_ISR*. Já no cenário de escalonamento preemptivo baseado em prioridades, essa ativação é um privilégio exclusivo do escalonador do sistema.

A forma de implementação da lógica de troca de contexto é uma mescla de comandos específicos de hardware do microprocessador e de uma função otimizada de escolha da tarefa mais prioritária a ser liberada. Isso se deve ao *stacking* e *unstacking* da pilha de registradores da tarefa programada no *microkernel*, ou seja, o seu contexto. Quando a interrupção *PendSV* é ativada pela tarefa *Tick* um *release jitter* é observado devido às interrupções estarem desativadas e adicionado à latência da própria interrupção. Já em *vDelayTaskUntil*, o único atraso observado se deve à latência de interrupção. A implementação de troca de contexto é tratada pela função *PendSV_Handler* que, por sua vez para a escolha da tarefa mais prioritária na fila dos prontos utiliza a função *vTaskSwitchContext*.

É apresentada na Listagem 4.12 a função de troca de contexto. Como não é implementada por uma seção crítica, a execução da troca de contexto pode ser interrompida por interrupções configuradas pelo projetista, entretanto, não é interrompida pela tarefa *Tick*, possuindo uma prioridade maior que ela e menor que as outras programadas. Em uma análise RTA o valor numérico temporal do chaveamento de contexto é adicionado ao WCET da tarefa (OLIVEIRA, R. S., 2018).

```
1  void vTaskSwitchContext( void ){
2      if( uxSchedulerSuspended != ( UBaseType_t )pdFALSE){
3          xYieldPending = pdTRUE;
4      } else {
5          xYieldPending = pdFALSE;
6          traceTASK_SWITCHED_OUT();
7          taskCHECK_FOR_STACK_OVERFLOW();
8          taskSELECT_HIGHEST_PRIORITY_TASK();
9          traceTASK_SWITCHED_IN();
10     }
11 }
```

Listagem 4.12 – Lógica de troca de contexto

4.3.2 *Tick*

O *Tick* do *FreeRTOS*, além de contagem da passagem do tempo, possui a função de auxiliar o escalonamento de tarefas de usuário, inclusive, esse é o único tratador de interrupções periódicas que se encontra no SOTR. Em aspectos de me-

dição, o valor de tempo é um inteiro, nomeado no *microkernel* como *xTickCount* em que é acrescido em uma unidade quando uma interrupção do tipo *SysTick* ocorre. Na liberação de alguma tarefa de usuário em que a tarefa a ser liberada possui mais alta prioridade que a tarefa em execução, uma troca de contexto é disparada através do registrador *portNVIC_INT_CTRL_REG*. É importante ressaltar também que a implementação de *Tick* é inteiramente uma seção crítica. É apresentada na Listagem 4.13 a implementação da tarefa *Tick*, presente no arquivo *port.c* para o processador ARM Cortex-M4.

```
1 void xPortSysTickHandler(void){
2     portDISABLE_INTERRUPTS();
3     {
4         if( xTaskIncrementTick() != pdFALSE ){
5             portNVIC_INT_CTRL_REG = portNVIC_PENDSVSET_BIT;
6         }
7     }
8     portENABLE_INTERRUPTS();
9 }
```

Listagem 4.13 – Tarefa *Tick* implementada no *FreeRTOS*

É dada ao *SysTick* a mais baixa prioridade entre as prioridades de interrupções do microprocessador, ou seja, a tarefa que implementa o *Tick* possui baixa prioridade em relação aos demais tratadores de interrupção criados pelo projetista do sistema. Para isso, o *microkernel* utiliza de uma macro no arquivo *port.c*, sendo ela *portNVIC_SYSTICK_PRI*.

A periodicidade *T* de ativação do *job* de *Tick* é controlada pelo *microkernel* através de duas macros presentes no arquivo de configuração: *portTICK_PERIOD_MS* e *configTICK_RATE_HZ*. A primeira é configurável e a segunda é uma constante fracionária definida como $\frac{1000}{configTICK\_RATE\_HZ}$, logo, se o projetista necessita de uma granularidade de 1 milissegundo, basta que a macro *configTICK_RATE_HZ* seja atribuída com o valor 1000, contudo, se deseja uma granularidade maior ou menor, basta diminuir ou aumentar o *rate*, respectivamente.

4.4 EVENTOS E MENSAGENS ENTRE TAREFAS

Foram apresentadas na Seção 4.1, na Seção 4.2 e na Seção 4.3 as estruturas básicas do *FreeRTOS* para implementação de um sistema com requisitos de tempo real. As estruturas e mecanismos apresentados bastariam para que o projetista de maneira pontual, implementasse mecanismos customizados de comunicação entre tarefas. No entanto, o *microkernel* suporta mais dois tipos de mecanismos de troca de mensagens entre tarefas ou entre tarefas e tratadores de interrupção, as notificações de tarefa (*Task Notify*) e os grupos de Eventos (*Event Groups*). A diferença entre os dois mecanismos se dá na quantidade de tarefas que podem receber uma mensagem através de uma chamada de API: Enquanto em uma *Task Notify* apenas uma tarefa

pode receber uma mensagem, em *Event Groups* é possível implementar um modelo de *broadcasting* entre tarefas, se necessário.

4.4.1 *Task Notify*

Uma das vantagens de se utilizar notificações de tarefas segundo Barry (2016) aponta, é o baixo uso de memória em comparação com certas estruturas, como uma fila, pois, a estrutura de notificação de uma tarefa remete a alterações no seu próprio TCB. Ao se habilitar a macro *configUSE_TASK_NOTIFICATIONS* são inseridos dois novos atributos em seu TCB, que são: *ulNotifyState* e *ulNotifiedValue*. O primeiro é um inteiro de 8 *bits* que representa três estados da tarefa que aguarda uma mensagem, *NOT_WAITING*, *WAITING* e *RECEIVED* e o segundo um inteiro de 32 *bits*, que é um valor de mensagem a se passar diretamente para a tarefa.

É possível também utilizar notificações de tarefa para implementação de semáforos binários ou contadores. Barry (2016) também cita algumas limitações, como a não possibilidade de envio de mensagens para mais de uma tarefa, a impossibilidade de envio de notificações para um tratador de interrupção, a impossibilidade de envio de mensagens do tipo estrutura (*struct*) e a quantidade de mensagens que pode ser enviada, que não é maior que um.

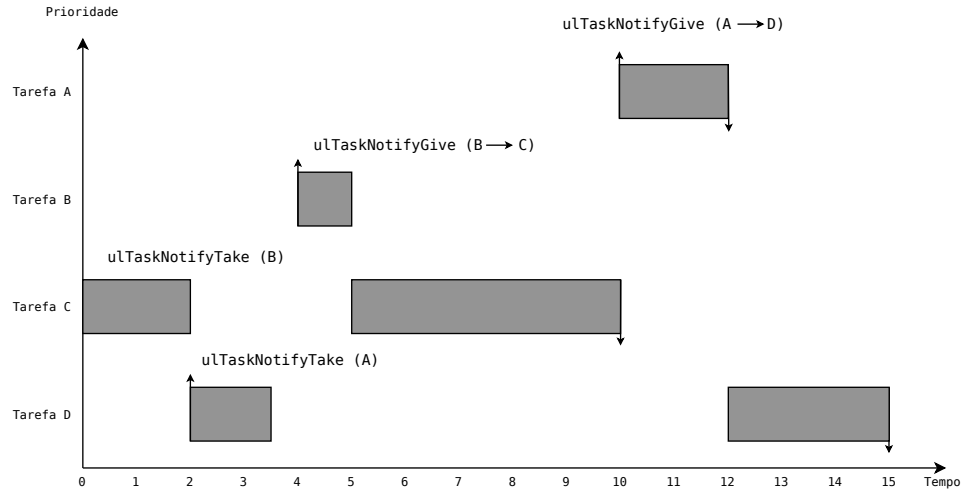
É apresentado na Figura 11 um diagrama temporal em que duas tarefas aguardam uma notificação (*C* e *D*) e tarefas que enviam uma notificação (*A* e *B*). O conteúdo da mensagem não é relevante para o exemplo, logo, em um dado momento, a tarefa *D* necessita aguardar uma mensagem de *A* e fica na fila *pxDelayedTaskList* até que isso aconteça. Quando a tarefa *A* envia a notificação, a tarefa *D* é colocada na fila de prontos e pode concorrer aos recursos de processamento.

Notificações, portanto, são estruturas úteis, porém, mais simples. Há uma série de API's que implementam a estrutura de comunicação de tarefas, todas elas baseadas nas primitivas de *send* e *receive*, como: As que notificam uma tarefa sem enviar uma mensagem (*xTaskNotify* e *xTaskNotifyGive*), as que enviam uma notificação com uma mensagem (*ulTaskNotifyTake* e *ulTaskNotifyGive*), e as que armazenam valores de notificação anteriores a chamada da próxima API de envio de notificação (*xTaskNotify* e *xTaskNotifyAndQuery*). De forma similar às demais APIs, estão disponíveis e suas variantes para uso em tratadores de interrupção (com final *fromISR*).

4.4.2 *Event Groups*

Grupos de Eventos são uma estrutura opcional do *FreeRTOS* que permite sincronização de tarefas. Da mesma maneira que a estrutura de *Task Notify* e a estrutura de *Event Groups* requer pouca memória para implementação. Um evento nessa estrutura é a alteração de um *bit* (chamado de *Event Flag*) em um vetor de 32 *bits* (chamado de *Event Group*). Os eventos são independentes entre si, sendo permitido criar uma

Figura 11 – Diagrama temporal de notificação entre quatro tarefas



Fonte – Autor

série de combinações de *Event Flags* que fazem com que uma tarefa seja liberada para execução apenas na ocorrência de uma combinação específica ou de um evento isolado, de acordo com a necessidade da tarefa.

Para registro de um evento no SOTR, a *Event Flag* associada a um *Event Group* tem o valor alterado de 0 para 1. Isso provê flexibilidade ao tratamento de eventos, sendo assim uma alternativa viável para construção de *broadcasting*. A abordagem de *Event Groups* fornece uma primitiva de comunicação de um a *n* tarefas, em um modelo semelhante aos de *publish/subscribe*.

A API de espera de um determinado evento é a *xEventGroupWaitBits*, já a API de registro de ocorrência para um evento é *xEventGroupSetBits*. Em ocasiões em que é necessário retornar todos os eventos para o estado zero, basta que a tarefa chame a função *xEventGroupClearBits*.

4.4.3 Algoritmos de gerência de memória do *FreeRTOS*

A memória é um item fundamental na implementação de um SOTR, pois, a forma como sua lógica de gerenciamento é construída influencia nos tempos de resposta de uma tarefa. É liberado ao projetista implementar sua própria lógica de gerenciamento, no entanto, com a evolução do *microkernel*, até a sua versão de número 10, cinco diferentes algoritmos foram construídos para os mais de 40 tipos de arquitetura que o SO suporta, logo, por ser um recurso necessário e finito, principalmente em sistemas embarcados, a gerência de memória é um item importante a ser discutido em um projeto que possua requisitos temporais. Um problema sensível em sistemas embarcados são as operações de *malloc* e *free*, respectivamente destinadas à alocação e liberação

de memória, substituídas no SOTR por *pvPortMalloc* e *vPortFree*.

O *FreeRTOS* separa a pilha de memória do sistema das pilhas de memória das tarefas, e implementa alguns algoritmos simples de gerência. As cinco estratégias de gerenciamento são detalhadas abaixo (BARRY, 2016):

a) ***heap_1.c***: Sendo o mais simples das cinco estratégias de gerência de memória, preparado para pra sistemas em que memória é um recurso extremamente crítico, dividindo a memória em blocos menores. Não é possível nessa estratégia de gerenciamento liberar espaços de memória, isso significa que operações como deletar uma fila, uma tarefa ou um *mutex* não estão disponíveis. É indicado para sistemas em que requisitos proíbem o gerenciamento dinâmico de memória.

b) ***heap_2.c***: Ao contrário de *heap_1*, o gerenciamento proporcionado pelo algoritmo permite liberação de memória de forma dinâmica. Utilizando o algoritmo *best-fit*, ele não combina os espaços lógicos vazios em um bloco maior. Em aspectos temporais é não-determinístico, dependendo de blocos livres, blocos ocupados e do tamanho de cada requisição de espaço em memória.

c) ***heap_3.c***: Encapsula as operações *malloc* e *free* padrão da linguagem C, com o tamanho de pilha de memória definido pela configuração do compilador. Ao *FreeRTOS* cabe deixá-las *thread safe*, desativando interrupções durante a chamada das funções.

d) ***heap_4.c***: Utiliza o algoritmo *first-fit* e divide os blocos em espaços menores como *heap_1* e *heap_2*. Ao contrário de *heap_2*, combina blocos adjacentes de memória em blocos maiores, de forma a tentar mitigar efeitos de fragmentação. Em aspectos temporais, também é não-determinístico.

e) ***heap_5.c***: O algoritmo em questão equivale a *heap_4.c*, mas permite a alocação dinâmica de dados em múltiplas regiões de memória. Para tanto, o projetista deve fornecer ao *microkernel* todas as regiões disponíveis através da API *vPortDefineHeapRegions*.

4.5 FORMAS DE *TRACING*

Para aferição temporal de um módulo implementado pelo SOTR é necessário conhecer o seu comportamento lógico-temporal e efetuar medições, afinal, a influência do SO não é desprezível no contexto de tempo real. As Funções Gancho (*Hook*) são funções que são ativadas a partir de uma determinada tarefa nos módulos do *FreeRTOS*. Elas fornecem suporte a partir de um evento para que o sistema possa atuar e são configuráveis a partir do arquivo de configuração do *microkernel*. São apresentadas no Quadro 4 as funções de apoio.

Quadro 4 – Funções *Hook* no *FreeRTOS*

Função	Descrição
Idle Hook	Uma função invocada na execução da tarefa ociosa do *microkernel*. O uso é pela alteração da *flag configUSE _IDLE _HOOK* e implementada no protótipo *vApplicationIdleHook*.
Tick Hook	Invocado ao final da execução de um *tick*. O uso é pela alteração da *flag* em *configUSE _TICK _HOOK* e é implementada em *vApplicationTickHook*
Malloc Failed Hook	É possível a invocação a partir da alteração da *flag configUSE _MALLOC _FAILED _HOOK* e implementado em *vApplicationMallocFailedHook*. É chamado ao final de um erro de alocação de memória.
Stack Overflow Hook	Invocado quando há a alteração da *flag configCHECK _FOR _STACK _OVERFLOW*. Há dois modos que podem ser ativados e a implementação pode ser realizada no protótipo *vApplicationStackOverflowHook*
Daemon Task Startup Hook	Chamada ao começo da ocorrência de um *timer* de software pela primeira vez. Para implementação no protótipo *vApplicationDaemonTaskStartupHook*, é necessário a alteração da *flag configUSE _DAEMON _TASK _STARTUP _HOOK*

Fonte – Amazon (2017)

O *microkernel* também oferece funções que auxiliam o projetista na depuração comportamental do sistema, que são chamadas de *trace macros*. As macros definidas pelo *FreeRTOS* se encontram distribuídas por todo o código do *microkernel* e proporciona ao projeto depuração e indicar, por exemplo, o início e o término de execução de uma tarefa, um valor enviado para uma posição em uma fila ou a ocorrência de um bloqueio.

Com as macros definidas pelo usuário ao final do arquivo de configuração (*FreeRTOSConfig.h*) é possível atuar sobre o microprocessador ou depurar o projeto, acionando um pino GPIO e enviando dados via barramento serial ou armazenando-os em memória. Para não comprometer o funcionamento temporal do sistema, é indicado que o código inserido seja sucinto e não-bloqueante, evitando-se assim utilizar módulos do *FreeRTOS* como filas, semáforos e *mutexes*. São mais de quarenta *trace macros* disponíveis para aferição. São apresentadas na Tabela 6 as principais macros que podem ser utilizadas na aferição de comportamento de uma tarefa.

O isolamento temporal de uma tarefa é um desafio, e técnicas de medição que buscam minimizar possíveis efeitos ocasionados por elas são alvos de estudo pela academia. Empiricamente, a forma menos invasiva de medição é a alteração de um pino lógico na chamada de cada *trace macro* desejada, acoplada a um osciloscópio. Com os dados lógico-temporais do sistema em mãos, é possível observar o comportamento da tarefa e as influências a que está submetida.

Tabela 6 – *Trace Macros* do *Cortex-M4*

Macro	**Ativação**
traceTASK_INCREMENT_TICK	Após a ocorrência de um *Tick*.
traceTASK_SWITCHED_IN	Após a tarefa receber o direito aos recursos de CPU.
traceTASK_SWITCHED_OUT	Após a tarefa ter retirado o direito aos recursos de CPU.
traceMOVED_TASK_TO_READY_STATE	Após a tarefa entrar na fila de prontos.
traceBLOCKING_ON_QUEUE_RECEIVE	Após a tarefa ser bloqueada por fila vazia ou *mutex* ocupado.
traceBLOCKING_ON_QUEUE_SEND	Após a tarefa ser bloqueada por fila cheia.
traceTASK_DELAY_UNTIL	Após a tarefa chegar ao fim de sua execução.
traceTASK_PRIORITY_INHERIT	Após a tarefa repassar sua prioridade para outra tarefa.
traceTASK_PRIORITY_DISINHERIT	Após a tarefa retornar a sua prioridade original.
traceCREATE_MUTEX	Após a criação de um *mutex*.
traceCREATE_MUTEX_FAILED	Após falha na criação de um *mutex*.
traceQUEUE_SEND_FROM_ISR	Após envio de dado para uma fila (em interrupção).
traceQUEUE_SEND	Após envio de dado para uma fila.
traceQUEUE_RECEIVE_FROM_ISR	Após receber de dado para uma fila (em interrupção).
traceQUEUE_RECEIVE	Após receber um dado em uma fila.
traceTASK_NOTIFY	Após a tarefa receber um mensagem de outra tarefa.
traceTASK_NOTIFY_FROM_ISR	Após a tarefa receber um mensagem de uma interrupção.
traceTASK_NOTIFY_WAIT	No ato da tarefa aguardar uma mensagem de outra tarefa.

Fonte – AWS (2017)

4.6 RESUMO

Neste capítulo foram discutidos os principais módulos que compõem o *microkernel FreeRTOS* na sua versão 10. Foram discutidos os conceito de tarefas, representada no *microkernel* pela estrutura *TCB_t*.

As estruturas de fila são utilizadas para gerenciamento de recursos compartilhados. O *FreeRTOS* adota o protocolo PIP para evitar o descontrole na inversão de prioridades, causada pela disputa dos recursos compartilhados, implementando-o em *mutexes* e semáforos, assim como disponibiliza funções que desabilitam interrupções, que são utilizadas pelo SOTR para lidar com suas próprias seções críticas.

O sistema também possui suas próprias tarefas: A *Tmr Svc* que tem o propósito de gerenciar todos os *timers* de software do sistema. A tarefa ociosa, que tem a função de reagrupar blocos livres adjacentes de memória *heap* na ausência de execução de tarefas programadas pelo usuário. A troca de contexto, que tem o propósito de varrer a lista de tarefas prontas e selecionar a primeira tarefa mais prioritária da lista.

Uma tarefa periódica que marca a passagem de tempo, a tarefa *Tick*, que possui sua periodicidade atrelada à frequência de *clock* do microprocessador.

Para traçar o comportamento do *microkernel*, o *FreeRTOS* disponibiliza dezenas de *trace macros* que são úteis para depuração em busca de erros ou como pontos de medição, assim como funções gancho, que são invocadas a partir da conclusão de um evento, como por exemplo um *Tick* ou uma falha em alocação de memória.

Foram também discutidas as estruturas de troca de mensagem e sincronização entre tarefas: *Task Notify* e *Event Groups*. Há também outras estruturas que não foram abordadas como *Message Buffers* e *Stream Buffers* que são construídas a partir de *Task Notify* e as co-rotinas que não são mais suportadas pela versão 10 do *FreeRTOS*.

Em aspectos de memória, foram apresentados os cinco algoritmos de gerenciamento, *heap_1.c*, *heap_2.c*, *heap_3.c*, *heap_4.c* e *heap_5.c*.

5 PLATAFORMA EXPERIMENTAL

Para coletar dados de um sistema, em especial sistemas considerados de tempo real, em que a medição de tempos de execução de módulos distintos é importante para uma análise temporal precisa, o desafio é buscar a forma menos intrusiva e que não influencie em cálculos de WCRT ou no sistema. Diversas abordagens estão presentes na literatura, como por exemplo Nicodemos *et al.*(2016) que cria uma abordagem baseada em hardware e outros, como Oliveira e Oliveira (2014) que apresentam alternativas de tecnologias já existentes de medição para o *Preempt-RT Linux*, que são adotadas por Oliveira e Oliveira (2016) para análise temporal de tarefas no próprio sistema.

Os experimentos foram realizados com a placa de desenvolvimento NUCLEO-F446RE, da marca *STMicrolectronics* [1], que possui um microprocessador ARM Cortex-M4. A frequência máxima de *clock* possível para o sistema é de 180*MHz* e os testes foram realizados com uma frequência fixada em 84*MHz*, frequência padrão na *Integrated Development Environment* (IDE) *STM32CubeIDE* [2]. Neste capítulo são apresentadas ferramentas para testes do *microkernel* e estratégias de coleta de dados temporais gerados pelas tarefas.

5.1 ESTRATÉGIAS DE COLETA DE DADOS

As subseções a seguir tratam de três estratégias de coleta de dados em um microprocessador. Diferentemente de um processador com um SOPG embarcado como o *Linux*, que possui ferramentas de *trace* próprias em que são apresentados os tempos de execução de módulos do Sistema Operacional a partir de chamadas de função específicas, o *FreeRTOS* não possui ferramentas avançadas de medição e coleta. A falta de ferramentas de coleta temporal disponibilizadas pelo fabricante de um microcontrolador ou de um microprocessador induzem o projetista que deseja uma aferição temporal do sistema a construir seus próprios módulos de coleta ou utilizar software de terceiros.

O *FreeRTOS* disponibiliza uma ferramenta de estatísticas sobre execução de tarefas que é implementada na função *vTaskGetRunTimeStats*, todavia, é apenas fornecido o tempo de execução de cada *Task* no microprocessador e sua utilização de CPU, que pode ser importante para outras aplicações, mas que fornece dados insuficientes para a realização de análises temporais. Para o fim de análise lógico-temporal, o *FreeRTOS* utiliza implementações de softwares de terceiros que são acopladas às *trace macros*, já discutidas no Capítulo 4.

Como parceiro oficial o *FreeRTOS* adota o software *Tracealyzer*, que é uma ferramenta paga. Como alternativa gratuita existem uma série de softwares, entre os

[1] Disponível em `https://www.st.com/en/evaluation-tools/nucleo-f446re.html`
[2] *Download* gratuito em `https://www.st.com/en/development-tools/stm32cubeide.html`

quais o *Segger System View*, que permite a gravação contínua ou por um tempo pré-determinado, a depender do espaço do *buffer* disponibilizado em memória para armazenamento dos eventos do *microkernel*. Para a primeira alternativa, o *firmware* da *Segger* é carregado no microprocessador, o que pode influenciar os tempos de execução em razão de ser um *firmware* de terceiros e não o *firmware* original que executará o SOTR. Para a segunda alternativa a memória é um obstáculo, pois em sistemas embarcados alocar um *buffer* com centenas ou dezenas de milhares de eventos pode ser inviável.

5.1.1 Coleta de dados via barramento serial

Para uma coleta de dados em que há a intenção do analista em utilizar um barramento serial de comunicação entre o microprocessador e um computador, é essencial a escrita de um programa que gere dados das tarefas. Independentemente de linguagem de programação, é fundamental que o programa receptor no computador possua suporte a conexões seriais através do SOPG. Há duas estratégias distintas em que é possível a transmissão de dados por barramento serial: Por armazenamento ou por envio imediato.

Na primeira estratégia, por armazenamento, assim que ocorre um evento interno ao *microkernel* como um bloqueio, uma troca de contexto ou um *Tick*, os dados que caracterizam o evento (um nome, um *timestamp* e uma *string*) são enviados para um *buffer*, que por sua vez são enviados para o computador através do barramento serial por uma tarefa de baixa prioridade, que é implementada exclusivamente para realizar a escrita de dados no barramento serial para posterior depuração de comportamento do sistema. É apresentada na Listagem 6.1 a lógica de armazenamento de um evento do *microkernel* no *buffer* a partir de sua ocorrência.

```
1 traceMOVED_TASK_TO_READY_STATE(xTask){
2     char nome_tarefa[] = xTask->pcTaskName; // Extrai nome da tarefa
3     // Adiciona evento em buffer
4     adicionaEventoBuffer(nome_tarefa, get_time(), TASK_READY);
5 }
```

Listagem 5.1 – Armazenamento de evento em *buffer*

A Listagem 6.2 apresenta a lógica da tarefa implementada para envio de dados pelo barramento serial. A sua prioridade deve ser a menor dentre as tarefas úteis ao sistema, ou seja, tarefas que implementam os requisitos do sistema de tempo real. A ordem dos eventos está garantida se a política de inserção de dados for do tipo *First In, First Out* (FIFO).

```
1  void taskSerial(void *params) {
2      struct Evento ev;
3      for (;;) {
4          if(naoVazio(&buffer)){ //Se o buffer nao esta vazio
5              removerEventoBuffer(ev, &buffer); // Remove evento do buffer
6              enviarEventoSerial(ev); // Envia dados pela serial
7          }else{
8              // Efetua pedido de troca de contexto para escalonador
9              efetuar_troca_contexto();
10         }
11     }
12 }
```

Listagem 5.2 – Lógica da tarefa auxiliar

Do ponto de vista do microcontrolador, como é implícito nesse tipo de abordagem, o uso de memória é um aspecto sensível e uma desvantagem, considerando a sua escassez em um sistema embarcado. Por delegar a retirada de eventos do *buffer* para uma tarefa implementada no próprio SOTR analisado, que está suscetível a concorrência pela CPU, é possível a perda de eventos em tempo de execução. Ainda que uma lista dinâmica seja utilizada como uma lista encadeada, como a memória é finita, o problema de espaço em *buffer* permanece, trazendo assim uma nova desvantagem de utilização dessa abordagem.

Uma vantagem é a rapidez com que a informação é inserida no *buffer*, trazendo um baixo *overhead* nos módulos internos ao *microkernel*, visto que os dados são inseridos diretamente na primeira posição livre do *buffer*. Portanto, é possível afirmar que dado esse comportamento, não são todos os casos em que tal abordagem de medição deve ser utilizada, sendo possível sua implementação em sistemas com poucas tarefas ou com uma certeza de comportamento lógico-temporal da tarefa que envia as informações para o barramento serial, de forma a garantir que nenhum evento será perdido por falta de espaço em *buffer*.

Do ponto de vista do computador que receberá os dados, é importante que não haja perda de informações, isso significa que a política de FIFO para armazenagem de eventos implementada no microprocessador também deve ser implementada no programa de recepção de dados.

A estratégia de envio imediato elimina a necessidade de implementação de um *buffer* auxiliar em que dados de eventos anteriores sejam armazenados, assim como logicamente elimina a necessidade de uma tarefa que descarregue todos os dados dos eventos em um barramento. Na ocorrência de um evento do *microkernel*, os dados referentes a ele são enviados imediatamente ao barramento serial, como exemplificado na Listagem 6.3.

```
1 traceMOVED_TASK_TO_READY_STATE(xTask){
2     // Cria estrutura de evento
3     struct Evento ev = { xTask->pcTaskName, get_time(), TASK_READY };
4     enviarEventoSerial(ev); // Envia e aguarda termino de envio
5 }
```

Listagem 5.3 – Armazenamento de evento em *buffer*

O uso dessa estratégia gera como desvantagem o alto *overhead* associado ao envio de um evento pelo barramento serial. É importante também ressaltar que em algumas situações, onde interrupções se encontram desabilitadas, não é possível utilizá-la, devido a sua inserção em pontos críticos ao *microkernel*, gerando assim possíveis perdas de eventos. Isso se deve a questões arquiteturais, em que protocolos seriais utilizam de tratadores de interrupção do microprocessador para o envio e recebimento de *bytes* através do barramento serial, não sendo obrigatório seu uso. Uma vantagem é o determinismo temporal, afinal o *baud rate* da aplicação é fixo.

5.1.2 Coleta armazenada em memória não-volátil

Em uma coleta de dados em que o projetista deseja armazenar informações em um cartão de memória ou outro tipo de memória não-volátil, é possível adotar as mesmas estratégias apresentadas, modificando as funções de envio de dados pelo barramento serial para armazenamento de dados em cartão de memória. Para o microcontrolador escolhido é possível adotar o protocolo *Serial Peripheral Interface* (SPI) ou *Secure Digital Input Output* (SDIO) para implementação de comunicação em baixo nível, acoplado a um sistema de gerenciamento de arquivos.

Na plataforma utilizada neste trabalho, é possível adotar o *File Allocation Table* (FAT) para sistemas de arquivos, implementado pela biblioteca ELM Chan [3]. A principal desvantagem da estratégia é a alta variação temporal no armazenamento, que depende da posição do arquivo em disco, supondo que o projetista opte por salvar todos os *logs* gerados em apenas um arquivo texto, para o caso de envio imediato. Isso gera uma alta variação no *overhead* do *microkernel*, algo indesejado. Para inserção de eventos em *buffer* para posterior envio por uma tarefa, a desvantagem de falta de espaço permanece.

5.1.3 Coleta por GPIO

Essa estratégia realiza apenas a mudança de estado em um ou mais pinos do microprocessador. Para isso atribui-se determinada ocorrência de algum evento a um determinado pino presente do microprocessador. Também é necessário um programa que faça a coleta lógico-temporal de cada mudança de estado do pino. Isso

[3] Disponível para *download* em `http://elm-chan.org/fsw/ff/00index_e.html`

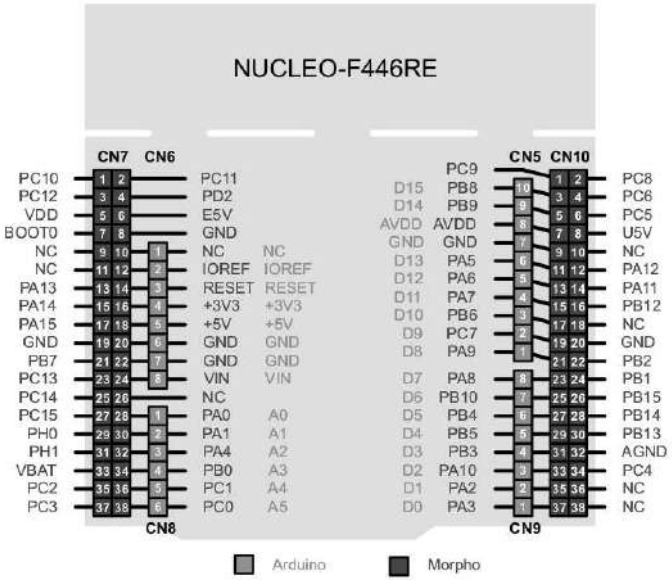
elimina a necessidade de implementação de um *buffer* que armazena os estados e por consequência elimina uma tarefa auxiliar que envia informações. A vantagem dessa estratégia é um baixo *overhead* , com apenas a execução de uma função em linguagem C. É apresentada na Listagem 6.4 a lógica de implementação.

```
1 traceMOVED_TASK_TO_READY_STATE(xTask){
2       if(strcmp(xTask->pcName, "Tarefa_Teste")){
3           HAL_GPIO_WritePin(GPIOC, GPIO_PIN_0, GPIO_PIN_SET);
4       }
5 }
```

Listagem 5.4 – Alteração de pino em mudança de estado para pronto

O microcontrolador possui quatro portas com 16 pinos cada para implementação de protocolos e também conectores para *Arduino*. Os valores são aferidos por um analisador lógico, com uma taxa de amostragem de 24.000.000 amostras por segundo, para armazenamento lógico-temporal e conferidos em um osciloscópio. É apresentado na Figura 12 o esquema lógico dos pinos no microcontrolador.

Figura 12 – Pinos do NUCLEO-F446RE



Fonte – UM1724 (2020)

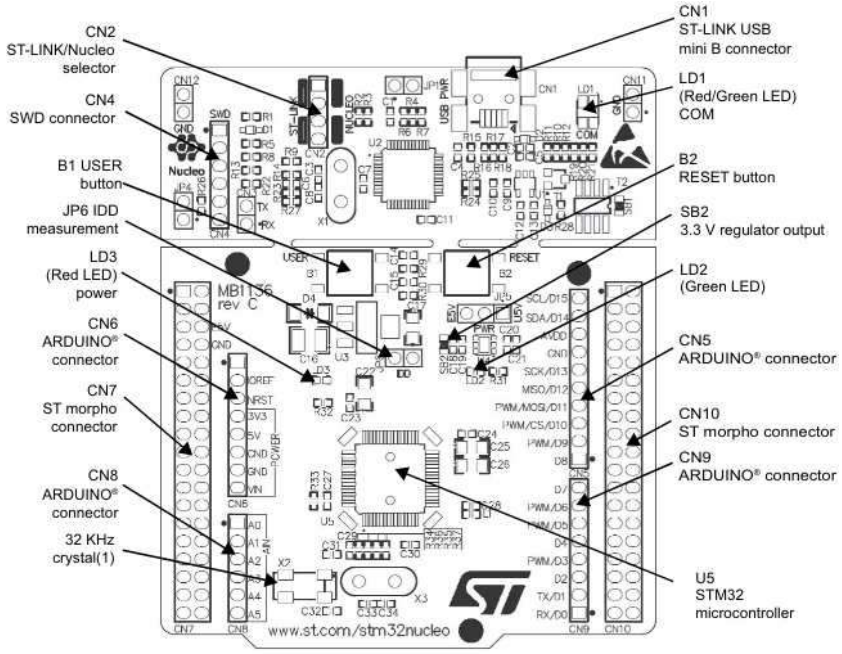
5.1.4 Forma de coleta

A forma de coleta de informações temporais adotada é o de coleta por GPIO, dada a baixa intrusão no *microkernel* e simplicidade de implementação. Com isso é possível a aferição de tempos de execução em osciloscópio e armazenamento dos estados lógicos dos pinos utilizando um analisador lógico acoplado a um computador.

5.2 MATERIAL UTILIZADO

É apresentado na Figura 13 o *layout* da placa de desenvolvimento NUCLEO-F446RE.

Figura 13 – *Layout* de NUCLEO-F446RE



Fonte – (UM1724. . . , 2020)

A placa possui conexões voltadas para módulos da plataforma *Arduino* através das Conexões *CN5*, *CN6*, *CN8* e *CN9*. A placa também dispõe de dois botões de teste *B1* e *B2* juntamente com dois *leds LD1* e *LD2* ligados no microcontrolador. Além das conexões para a plataforma *Arduino*, possui conexões *Morpho CN7* e *CN10* para demais conexões com outros periféricos. Conexões de depuração estão representados em *CN1*, *CN2* e *CN4*.

A placa contém um microcontrolador embarcado do modelo STM32F446E, que dispõe de um núcleo microprocessador ARM Cortex-M4. Segundo o manual DS10693(2020) o microcontrolador possui uma memória *flash* de *512 KB* e *128 KB* de memória *RAM*. Além disso possui três módulos *Analog-to-Digital Converter* (ADC), dezessete *timers* implementados em hardware e vinte tipos distintos de protocolos de comunicação como *Controller Area Network* (CAN), SPI e *Universal Synchronous Asynchronous Receiver Transmitter* (USART).

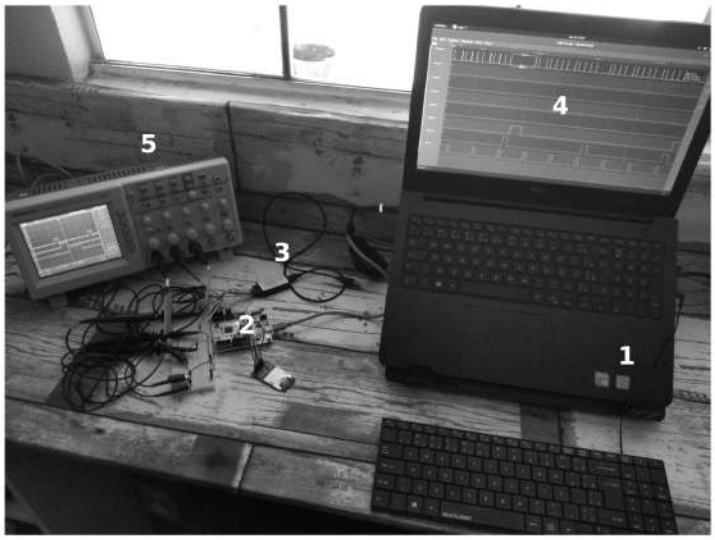
A placa também possui um controlador de DMA e suporte a *low power mode*, para menor consumo de energia. Por se tratar de um microcontrolador, as bibliotecas em linguagem C são limitadas.

Para coleta de dados foi utilizado um analisador lógico de oito canais *Logic Saleae 8* com uma frequência de 24*MHz*. O analisador possui um software gratuito de captura de dados, o *Saleae Logic* [4]. É possível salvar os dados coletados em formato *Comma-Separated Values* (CSV) e utilizar um software para análise de dados. Para isso, foi utilizado o *MATLAB*. O computador utilizado para execução da coleta de dados utilizando o software *Saleae Logic* foi um *Intel Core i5* com quatro núcleos de processamento e *8GB* de memória *RAM* executando uma distribuição *Fedora Linux* 33.

5.3 BANCADA DE TESTES

É apresentada na Figura 14 a bancada de testes construída para experimentação e coleta de dados temporais resultante de eventos internos ao *microkernel*, *Tasks*, *Timers* e *Interrupts*.

Figura 14 – Bancada de testes utilizada para experimentação



Fonte – Autor

Na Figura 14, os elementos estão dispostos na seguinte ordem:

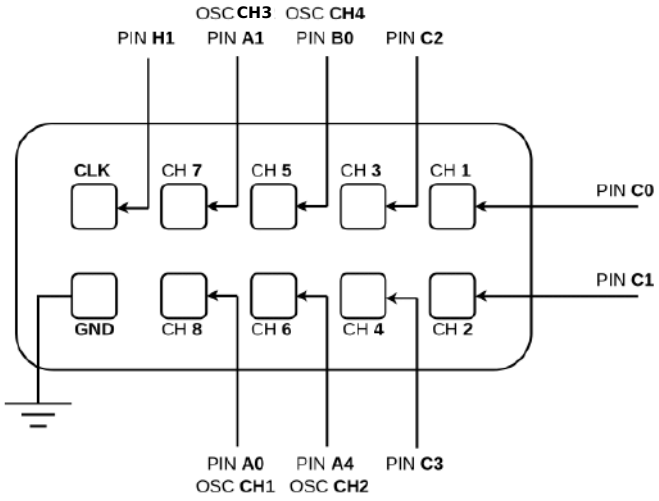
- Em **1**, um *notebook* executando o *Fedora Linux* em sua versão 33, com um processador *Intel Core i5* com quatro núcleos;
- Em **2**, o *kit* de desenvolvimento NUCLEO-F446RE com um microcontrolador equipado com um núcleo ARM Cortex-M4, utilizando como SOTR o *FreeRTOS* na versão 10.2.1 e que é programado usando o software *STMCubeIDE*;

[4] *Download* gratuito em `https://www.saleae.com/pt/downloads/`

- Em **3**, um analisador lógico de 8 *bits Saleae* com capacidade de amostragem de 24 milhões de amostras por segundo que são utilizadas por **4**, um software *Saleae Logic* para coleta de dados e exportação de dados para formatos como o CSV;
- Em **5**, um osciloscópio digital de quatro canais utilizado para fins de comparação de dados coletados por **3**.

Os sinais digitais do sistema são definidos pelas saídas *Morpho* do microcontrolador. Como o analisador lógico possui apenas oito *bits* de entrada, é possível apenas o uso de oito saídas digitais do microprocessador. São apresentadas saídas digitais utilizadas são **C0**, **C1**, **C2**, **C3**, **B0**, **A0**, **A1** e **A4**. Os outros dois pinos são o aterramento (**GND**) e de sincronização de *clock* (**CLK**). As pontas de provas do osciloscópio, sendo elas, *OSC CH1*, *OSC CH2*, *OSC CH3* e *OSC CH4*, são mapeados para a porta A e porta B. É apresentado na Figura 15 as saídas digitais e pontas de prova do osciloscópio digital mapeadas para o analisador lógico.

Figura 15 – Mapeamento de saídas digitais para analisador lógico



Fonte – Autor

Para as pontas de prova do osciloscópio digital, devido ao número limitado de quatro canais, são mapeados os sinais digitais mais relevantes para análise, como tempos de computação de *Tasks*, *Interrupts* ou *Timers* que podem influenciar no aumento de WCRT da tarefa analisada.

5.4 RESUMO

O objetivo deste capítulo foi apresentar o material utilizado para testes temporais sobre o *microkernel FreeRTOS*.

Como o microprocessador *Cortex-M*4 e o *microkernel FreeRTOS* não possuem ferramentas completas de *tracing*, são necessárias estratégias de coletas de dados temporais. Para isso, foi apresentada a coleta por barramento serial, que é intrusiva, porém com alto determinismo. Também foi apresentada a coleta por armazenamento em memória não-volátil, com alta capacidade de armazenamento em um cartão de memória, mas baixo determinismo causada pelo uso do sistema de arquivos FAT, que podem causar uma alta intrusão temporal no sistema. Finalmente, foi apresentada a coleta por GPIO, que possui um alto determinismo temporal e baixo *overhead* em comparação aos demais, porém com necessidade de aferição de dados em osciloscópio.

Neste capítulo também foram apresentados a bancada de testes temporais do sistema, os materiais utilizados e os recursos da placa NUCLEO-F446RE, que é equipada com o microcontrolador do modelo *STM32F446E*. Foi apresentado, para coleta por GPIO, o analisador lógico de oito canais e 24*MHz* de frequência.

São apresentados no Quadro 5 os possíveis métodos de coleta temporal de dados, com suas vantagens e desvantagens de aplicação em um sistema embarcado. A abordagem de coleta por GPIO é utilizada para coleta de dados temporais gerados pelas tarefas no SOTR para esta pesquisa. Com os dados temporais coletados em mãos, é possível exportar os dados do software de coleta para o formato CSV, que pode ser utilizado para análise de dados. Para esta pesquisa, o software *MATLAB* foi utilizado para análise de dados.

Quadro 5 – Formas possíveis de coleta de dados temporais na plataforma

Coleta	**Vantagens**	**Desvantagens**
Serial	Alto determinismo.	Alto *overhead*, interrupções de barramento, escrita de tarefa, uso excessivo de memória e escrita de programa em computador para armazenamento de dados.
Disco	Alto armazenamento.	Baixo determinismo, interrupções de barramento e escrita de programa em computador para armazenamento de dados.
GPIO	Alto determinismo, baixo *overhead* e programas prontos para coleta de dados	Aferição de dados em osciloscópio de forma a comparar dados obtidos com um analisador lógico.

6 ANÁLISE TEMPORAL DE TAREFAS

Nos capítulos anteriores foram discutidos aspectos teóricos relevantes da análise de sistemas de tempo real, as características de hardware do microprocessador, o ARM Cortex-M4, uma discussão a respeito do *microkernel* de tempo real *FreeRTOS* e finalmente uma descrição da plataforma experimental. As análises de tempo de resposta não se preocupam com aspectos de variação do tempo de execução, de forma a dirimi-los ou identificar possíveis fontes de variação, preocupando-se somente em obter uma previsibilidade matemática de comportamento de uma tarefa de forma a confirmar ou não que a mesma nunca perderá o seu *deadline*. Podem ser necessárias suposições temporais nos cálculos de WCRT de tarefas do sistema, como usar o *High Water Mark* (HWM), que é o pior tempo de execução observado de uma tarefa pelo analista, mas que possivelmente não é o seu verdadeiro WCET.

Caracterizar algebricamente uma tarefa em um SOTR pode ajudar o projetista a entender melhor o funcionamento de um sistema programado no *microkernel*. Neste capítulo será apresentada uma caracterização de tarefas programadas no *FreeRTOS*, com foco em tempos de resposta e caracterização matemática de influências do *overhead* do *microkernel* em tarefas programadas. As tarefas programadas pelo usuário no microkernel podem ser de três tipos distintos: *Timers*, *Tasks* e *Interrupts*. Através dessas três tarefas um usuário pode oferecer um serviço ou atuar sobre uma malha de controle através do *microkernel*.

6.1 PREMISSAS

É importante para análise dos sistemas de tempo real previsibilidade de execução, isso significa que certas premissas de um modelo de análise devem ser adotadas. No modelo de RTA, por exemplo, é fundamental em um conjunto analisável de tarefas que possuam um período fixo ou esporádico e uma prioridade única e distinta das demais tarefas do conjunto em execução. Os tópicos a seguir detalham premissas de um modelo criado para análise de tempos de resposta para tarefas implementadas pelo usuário no *FreeRTOS*.

- **Tarefa**: No modelo de análise, uma tarefa é criada exclusivamente antes da execução do escalonador, não sendo possível sua criação ou remoção no decorrer da execução do sistema;
- **Prioridade**: Ao contrário do que é proporcionado pelo *FreeRTOS* pela função *vTaskPrioritySet*, uma tarefa programada pelo usuário possuirá uma prioridade fixa e única, excluindo-se a possibilidade de execução do algoritmo *round-robin* do escalonador entre tarefas de usuário, impondo assim ao sistema que o escalonamento seja totalmente preemptivo e baseado em prioridade fixa;

- **Recorrência**: Todas as tarefas possuirão um tempo de recorrência mínima T_{min}, não sendo possível sua alteração no decorrer da execução do sistema;
- **Termos Algébricos**: Para estimativas de pior tempo de resposta de uma tarefa, serão adotados os termos algébricos do trabalho de Audsley *et al.*(1993). Os fundamentos de sistemas de tempo real foram apresentados no Capítulo 2.

6.2 TERMINOLOGIA

É necessária, para compreensão deste capítulo, a introdução de terminologias adotadas pelo *microkernel* na sua documentação. Termos adotados na literatura de tempo real quando aplicados ao contexto do *FreeRTOS* possuem algumas diferenças semânticas, portanto, a terminologia adotada para este capítulo busca manter uma proximidade com os termos utilizados pelo SOTR em inglês. Os termos adotados neste capítulo são apresentados a seguir:

- **Interrupção**: Evento associado ao mecanismo de interrupções quando, por hardware ou software, externo ou interno, sendo ou não para o tratamento de exceções, sinaliza uma interrupção. Interrupções são geradas por periféricos, por exceções, por software, pelo temporizador em hardware, etc. Uma interrupção pode ser desabilitada por funções específicas da arquitetura e NMIs.
- **Tratador de interrupção**: Trecho de código executado em resposta a uma interrupção, após a interrupção ter sido reconhecida e o fluxo de execução desviado para o respectivo tratador. Como interrupções são tipadas, cada tipo de interrupção aciona um tratador específico.
- ***Interrupt***: Tarefa de usuário implementada na forma de um tratador de interrupção em hardware e que está fora do escopo de gerenciamento do *microkernel*. Um *Interrupt* está diretamente vinculado à arquitetura, possuindo a sua própria prioridade fixa e ativação periódica ou esporádica programada pelos temporizadores de hardware, barramentos seriais, sensores, etc.
- ***Task***: Tarefa de usuário implementada que está sob gestão do *microkernel*, possuindo uma prioridade fixa, um período fixo e que é criada através da função *"xTaskCreate"*, disponível na biblioteca *"task.h"* do *FreeRTOS*.
- ***Timer***: Tarefa de usuário que está sob gestão do *microkernel* através de uma *Task* criada automaticamente pelo SOTR denominada *Tmr Svc*, possuindo uma prioridade fixa herdada de *Tmr Svc*, um período fixo e que é criada através da função *"xTimerCreate"*, disponível na biblioteca *"timer.h"* do *FreeRTOS*.

6.3 *TICK*

O *Tick*, como exposto no Capítulo 4, é um código implementado sob um tratador de interrupção do ARM Cortex-M4 denominado *SysTick*. Além de marcar a passagem de tempo, o *Tick* possui o propósito de auxiliar no escalonamento de tarefas implementadas no *microkernel*. Tendo em vista uma análise voltada para caracterização de tempos de resposta é necessário identificar os efeitos temporais que tal tarefa do *microkernel* impõe às tarefas programadas pelo usuário, portanto, esta seção tem como objetivo uma caracterização algébrica do comportamento do *Tick*.

6.3.1 Fontes de Atraso

O código presente em *Tick* executa com interrupções desabilitadas, como mostrado na Listagem 4.13. Logo, uma vez iniciado o processamento da função *xTaskIncrementTick* é impossível ao *microkernel* ou mesmo ao usuário parar sua execução com alguma função específica. Na busca de um maior tempo observável de *release jitter* é possível no modelo apontar duas causas que podem atrasar a ativação desse *job*.

A primeira causa é a habilidade de uma tarefa de usuário ou mesmo alguma *system-call* conseguir, assim como *Tick*, desabilitar interrupções para gerenciamento de alguma seção crítica. Dado o tempo t_i de ativação de uma tarefa, em que uma chamada de sistema é realizada com interrupções desabilitadas, um efeito temporal retardante é imposto a *Tick*, causando assim um *release jitter*.

A segunda causa são *Interrupts*. Como foi exposto no Capítulo 4, a prioridade de *Tick* é a mais alta dentre as tarefas programadas pelo projetista no *microkernel* e é a mais baixa em relação a todas as interrupções de hardware programadas. No momento de ativação t_i de *Tick*, é possível que uma *Interrupt* programada pelo usuário esteja executando, isso significa que *Tick* deverá aguardar o término de execução da *Interrupt* para iniciar sua execução.

Além disso, por conta da flexibilidade e disponibilidade, o microprocessador permite aninhamento de *Interrupts*, logo, em um cenário temporal ainda mais descontrolado, é possível que *Tick* aguarde a execução de todas as *Interrupts* aninhadas à *Interrupt* que impediu sua liberação, por também serem mais prioritárias que *Tick*. Em casos onde uma interrupção do tipo *SysTick* é liberada e começa a ser atendida, é possível que antes que as instruções de máquina que desabilitam as interrupções sejam processadas, um *Interrupt* de mais alta prioridade pode chegar e ser atendida.

6.3.2 Tempo de computação

A variação temporal na execução de *Tick* é diretamente influenciada pela quantidade de tarefas periódicas implementadas sobre o *microkernel*. É apresentada na

Listagem 5.1 a implementação da função *xTaskIncrementTick*. A variação temporal decorre das manipulações de listas do *microkernel*, em que *Tick* vai retirando tarefas do estado bloqueado nos casos em que o *timeout* de um *job* expirou e necessita naquele instante novamente ser ativado.

```
1  BaseType_t xTaskIncrementTick( void ) {
2  TCB_t * pxTCB;
3  TickType_t xItemValue;
4  BaseType_t xSwitchRequired = pdFALSE;
5
6  traceTASK_INCREMENT_TICK( xTickCount );
7  if( uxSchedulerSuspended == ( UBaseType_t ) pdFALSE ){
8    const TickType_t xConstTickCount = xTickCount + ( TickType_t ) 1;
9    xTickCount = xConstTickCount;
10
11   if( xConstTickCount == ( TickType_t ) 0U ) {
12     taskSWITCH_DELAYED_LISTS();
13   }else mtCOVERAGE_TEST_MARKER();
14
15   if( xConstTickCount >= xNextTaskUnblockTime ){
16     for( ;; ){
17       if( listLIST_IS_EMPTY( pxDelayedTaskList ) != pdFALSE ){
18         xNextTaskUnblockTime = portMAX_DELAY;
19         break;
20       } else{
21         xItemValue = listGET_LIST_ITEM_VALUE( &( pxTCB->xStateListItem ) );
22         if( xConstTickCount < xItemValue ){
23           xNextTaskUnblockTime = xItemValue;
24           break;
25         } else mtCOVERAGE_TEST_MARKER();
26
27         ( void ) uxListRemove( &( pxTCB->xStateListItem ) );
28
29         if(listLIST_ITEM_CONTAINER(&(pxTCB->xEventListItem))!= NULL ){
30           ( void ) uxListRemove( &( pxTCB->xEventListItem ) );
31         }else mtCOVERAGE_TEST_MARKER();
32
33         prvAddTaskToReadyList( pxTCB );
34
35         #if (configUSE_PREEMPTION == 1){
36           if( pxTCB->uxPriority >= pxCurrentTCB->uxPriority ){
37             xSwitchRequired = pdTRUE;
38           } else mtCOVERAGE_TEST_MARKER();
39         }
40         #endif // configUSE_PREEMPTION
41         }
42     }
43   }
44
45   #if ((configUSE_PREEMPTION == 1) && (configUSE_TIME_SLICING == 1)){
46     if( listCURRENT_LIST_LENGTH(&(pxReadyTasksLists[pxCurrentTCB->uxPriority])) > (
       UBaseType_t) 1 ){
47       xSwitchRequired = pdTRUE;
48     }
49     else mtCOVERAGE_TEST_MARKER();
50   }
51   #endif
52
53   #if (configUSE_TICK_HOOK == 1){
```

```
54      if(uxPendedTicks == ( UBaseType_t ) 0U){
55        vApplicationTickHook();
56      } else mtCOVERAGE_TEST_MARKER();
57    }
58    #endif
59  } else {
60    ++uxPendedTicks;
61
62    #if (configUSE_TICK_HOOK == 1){
63      vApplicationTickHook(); // The tick hook gets called at regular intervals, even if the
        scheduler is locked.
64    }
65    #endif
66  }
67
68  #if (configUSE_PREEMPTION == 1) {
69    if(xYieldPending != pdFALSE) {
70      xSwitchRequired = pdTRUE;
71    } else mtCOVERAGE_TEST_MARKER();
72  }
73  #endif
74
75  return xSwitchRequired;
76  }
```

Listagem 6.1 – xTaskIncrementTick Function

Para cada *Task* desbloqueada em seu respectivo período, sua retirada de *pxDelayedTaskList* (através da implementação de *uxListRemove* para eventos e para tarefas aguardando ativação) e sua posterior inserção na respectiva lista de *pxReadyTasksLists[σ]* através da função *prvAddTaskToReadyList* ($\sigma \equiv$ prioridade da *Task*), gera uma nova iteração dentro da função *xTaskIncrementTick*. A ocorrência desse fenômeno em uma análise pessimista pode fazer com que o WCET de *Tick* seja considerado o instante crítico de Liu e Layland (1973) onde todas as tarefas implementadas são liberadas ao mesmo tempo, gerando *n* iterações, onde *n* é a quantidade de tarefas implementadas sob gestão do *microkernel*, excetuando tarefas implementadas em tratadores de interrupção e *Timers*.

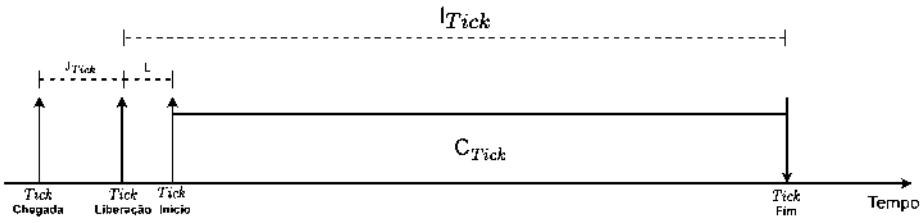
6.3.3 Modelo algébrico temporal

De forma a evitar uma análise pessimista, é possível realizar a partição de comportamento temporal de *Tick* em *pseudo-Tasks*. Cada *pseudo-Task* γ^i_{Tick} tem um tempo de computação correspondente ao tempo de remover uma *Task* γ de *pxDelayedTaskList* e sua subsequente inserção em *pxReadyTaskLists*[σ], onde σ denota sua prioridade. Isso logicamente faz com que *n Tasks* criem *n pseudo-Tasks* resultantes da remoção e da inserção de *Tasks* em listas. Como o comportamento temporal de γ^i_{Tick} é periódico, é possível assumir que T(γ^i_{Tick}) é igual T(γ). Em relação à prioridade de γ^i_{Tick} é possível assumir que essa *pseudo-Task* possui uma prioridade menor que *Tick* e mais alta prioridade que outras tarefas sob gestão do *FreeRTOS*, dado que esta

execução é feita no contexto de interrupções desabilitadas.

É possível construir um modelo algébrico temporal de *Tick*. Dado que *Tick* executa com interrupções desabilitadas, não é possível a preempção por *Interrupts* com mais alta prioridade que *Tick*. Também, *Tick* não compartilha recursos com outra *Task* ou outra *Interrupt*, portanto, nenhum ponto de bloqueio é observado. É apresentado pela Figura 16 o comportamento de *Tick* sobre *Tasks* e *Timers*.

Figura 16 – Comportamento temporal de *Tick*



Fonte – Autor

No diagrama temporal apresentado pela Figura 16, o impacto da latência arquitetural, que no ARM Cortex-M4 é de 12 ciclos de *clock*, representado por *L*, corresponde ao intervalo de tempo necessário para o banco de registradores armazenar e recarregar o contexto de execução de *Tick*. A influência temporal de *Tick* é representado por C_{tick}. É apresentada na Equação (6.1) a influência temporal de *Tick* sobre uma tarefa programada pelo usuário.

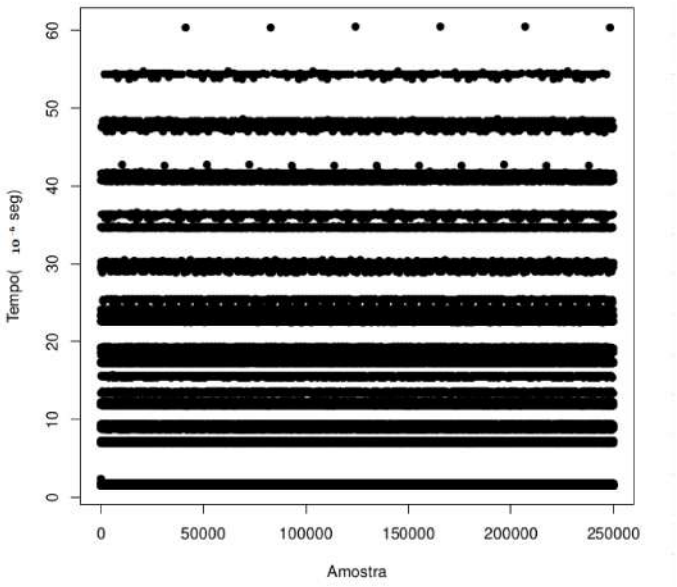
$$I_{Tick} = L + C_{tick} \tag{6.1}$$

6.3.4 Análise temporal

São apresentados na Figura 17 os tempos de computação de *Tick* ao decorrer de 250 segundos no microprocessador ARM Cortex-M4. O sistema possui 10 *Tasks* periódicas. A prioridade de cada *Task* é irrelevante para a análise do comportamento temporal de *Tick*. Os períodos de cada *Task* são, em ordem crescente: 05, 10, 12, 15, 20, 23, 30, 40, 45 e 50 milissegundos.

O gráfico apresenta seis picos causados pela chegada simultânea de todas as *Tasks* no sistema, também chamado de instante crítico por Liu e Layland (1973). O WCET observado de *Tick* para liberação de dez tarefas é aproximadamente de 60,75µs e o melhor caso de execução medido é de 1,5µs. Esse tempo de execução é observado quando não há manipulação de *Tasks* em listas do *microkernel*, um cenário otimista para *Tick*.

Figura 17 – Tempos de execução de *Tick*



Fonte – Autor

Com o objetivo de reduzir o pessimismo, neste modelo, *Tick* pode ser particionado em dez novas *pseudo-Tasks* com mesma prioridade que *Tick*, que possui uma prioridade menor que a prioridade de *Interrupts*. Uma vez que cada chegada de uma *Task* influencia o tempo de execução do *Tick*, logo, são criadas *pseudo-Tasks* γ^{i}_{Tick} para cada *Task* γ_i. Para definir o tempo de execução das *pseudo-Tasks* é necessário medir os tempos de iteração em *xTaskIncrementTick*. O tempo de inserção e remoção de cada *Task* possui uma variação temporal causada pelos mecanismos de aceleração do hardware. Portanto, o tempo de execução de γ^{i}_{Tick} é assumido como o pior tempo de iteração medido quando é inserida qualquer *Task* em *pxReadyTaskLists* e removida de *pxDelayedTaskList*.

No entanto, o tempo de *Tick* medido em pior caso é 60,75μs, o que significa que o *overhead* estimado difere 22,07μs do tempo medido de *Tick* no pior caso. Isso acontece porque as instruções antes da linha 17 da função *xTaskIncrementTick* na Listagem 5.1 não são medidas. Esse bloco de código não é variante de tempo no algoritmo, portanto, é aceitável dividir esse tempo restante entre pseudo-tarefas. Considerando isso, se somarmos o tempo para mover a *Task* de *pxDelayedList* (consulte linha 21) para a lista pronta (consulte a linha 33) para o tempo de execução de *Tick* restante, cada pseudo-tarefa leva 5,9 μs para ser executada. A Tabela 7 apresenta a modelagem das pseudo-tarefas resultantes da divisão do *Tick*, bem como seus respectivos períodos.

Como discutido e apresentado na Tabela 7, as medições de *Tick* comprovam

Tabela 7 – Tarefas de sistema resultantes do particionamento de *Tick*

Tarefa	**Período**	**Tempo de Computação**
Tick	01 ms	1,5 μs
γ^{A}_{Tick}	05 ms	(3,7 + 2,207) μs
γ^{B}_{Tick}	10 ms	(3,7 + 2,207) μs
γ^{C}_{Tick}	12 ms	(3,7 + 2,207) μs
γ^{D}_{Tick}	15 ms	(3,7 + 2,207) μs
γ^{E}_{Tick}	20 ms	(3,7 + 2,207) μs
γ^{F}_{Tick}	23 ms	(3,7 + 2,207) μs
γ^{G}_{Tick}	30 ms	(3,7 + 2,207) μs
γ^{H}_{Tick}	40 ms	(3,7 + 2,207) μs
γ^{I}_{Tick}	45 ms	(3,7 + 2,207) μs
γ^{J}_{Tick}	50 ms	(3,7 + 2,207) μs

o modelo teórico baseado em pseudo-tarefas de maneira a reduzir o pessimismo na estimativa de pior tempo de computação de *Tick*.

6.4 CHAVEAMENTO DE CONTEXTO

Operações de chaveamento de contexto (*context switch*) são consideradas normalmente como uma caixa-preta em que um projetista leigo em aspectos arquiteturais de uma determinada tecnologia de implementação de um Sistema Operacional não possui certeza do que acontece, mas, apenas observa o efeito lógico esperado, que é a troca do monopólio de processamento entre uma tarefa e outra. Alguns projetistas mais atentos à questão temporal podem medir os tempos de execução de chaveamento de contexto, todavia, não se preocupam com fontes de atraso ou motivos de variabilidade temporal.

Em uma perspectiva de preocupação com o comportamento temporal de um chaveamento de contexto em qualquer SOTR, o ideal seria que tal efeito lógico possuísse um tempo de computação igual a zero, não interferindo em outra tarefa ou não acrescentando fontes de variação para a própria tarefa que deseja entregar os recursos de processamento. No entanto, o que pode ser feito na realidade é minimizar os efeitos temporais de tal operação de escalonamento, acelerando ou otimizando seus algoritmos, haja vista que é geralmente impossível fazê-lo de forma instantânea.

Como discutido no Capítulo 4, o *FreeRTOS* possui uma estrutura de chaveamento de contexto que é programada na exceção *PendSV* da arquitetura ARM Cortex-M4. Nela, a função *vTaskSwitchContext* implementada em linguagem C, é executada de forma mesclada a instruções próprias da arquitetura *Cortex-M* em *assembly*. Nesta seção será discutido o comportamento temporal algébrico da operação de chavea-

mento de contexto, com vista às suas fontes de atraso e fontes de variação temporal.

6.4.1 Fontes de atraso

Release jitter para execução de chaveamento de contexto pode acontecer quando *Tick* desabilita interrupções. Como evidenciado pela Listagem 5.1 da seção anterior, a função *xTaskIncrementTick* possui a função de incrementar a variável que conta a passagem de tempo e alterar estados das tarefas de bloqueado ou suspenso para pronto, assim como movimentar o ponteiro da tarefa que monopolizará o processador entre as listas de *pxDelayedTaskList* e *pxReadyTasksLists*[σ], em que σ é a sua prioridade. Ao se detectar que é necessário um chaveamento de contexto, isto é, uma tarefa de mais alta prioridade ficou pronta em um instante t_i qualquer, o *microkernel* ativa uma interrupção para tal, que é a *PendSV* para chaveamento de contexto, como evidenciado na Listagem 4.12. Contudo, como as interrupções se encontram desabilitadas, a execução de chaveamento de contexto não será imediata, levará um tempo para o chaveamento de contexto e ainda haverá latência para que seja atendida a operação que implementa o chaveamento de contexto propriamente dito. Atrasos de liberação de chaveamento de contexto por causa de interrupções desabilitadas por outras tarefas dependem do *design* do sistema.

6.4.2 Tempo de Computação

De forma a viabilizar o chaveamento de contexto o *microkernel* executa instruções escritas diretamente em *assembly*, que estão apresentadas na Listagem 5.2.

```
1   void xPortPendSVHandler( void ) {
2  __asm volatile
3  (
4  "   mrs r0, psp                         \n" // Pilha de memoria da tarefa
5  "   isb                                 \n"
6  "                                       \n"
7  "   ldr r3, pxCurrentTCBConst           \n"
8  "   ldr r2, [r3]                        \n"
9  "                                       \n"
10 "   tst r14, #0x10                      \n" // Condicional ponto flutuante
11 "   it eq                               \n" // Armazena Registradores VFP
12 "   vstmdbeq r0!, {s16-s31}             \n"
13 "                                       \n"
14 "   stmdb r0!, {r4-r11, r14}            \n" // Armazena contexto.
15 "   str r0, [r2]                        \n"
16 "                                       \n"
17 "   stmdb sp!, {r0, r3}                 \n"
18 "   mov r0, %0                          \n"
19 "   msr basepri, r0                     \n" // Desabilita interrupcoes
20 "   dsb                                 \n"
21 "   isb                                 \n" // Seleciona...
22 "   bl vTaskSwitchContext               \n" // Tarefa + alta prioridade.
23 "   mov r0, #0                          \n"
24 "   msr basepri, r0                     \n" // Habilita interrupcoes.
25 "   ldmia sp!, {r0, r3}                 \n"
```

```
26    "                                       \n"
27    "    ldr r1, [r3]                       \n"
28    "    ldr r0, [r1]                       \n"
29    "                                       \n"
30    "    ldmia r0!, {r4-r11, r14}           \n" // Restaura novo contexto
31    "                                       \n"
32    "    tst r14, #0x10                     \n" // Condicional ponto flutuante
33    "    it eq                              \n" // Restaura Registradores VFP
34    "    vldmiaeq r0!, {s16-s31}            \n"
35    "                                       \n"
36    "    msr psp, r0                        \n"
37    "    isb                                \n"
38    "                                       \n"
39    #ifdef WORKAROUND_PMU_CM001 // XMC4000 specific errata workaround.
40      #if WORKAROUND_PMU_CM001 == 1
41    "            push { r14 }               \n"
42    "            pop { pc }                 \n"
43      #endif
44    #endif
45    "                                       \n"
46    "    bx r14                             \n" // Retorna a proxima instrucao...
47    "                                       \n" // posterior ao chaveamento de contexto
48    "    .align 4                           \n"
49    "pxCurrentTCBConst: .word pxCurrentTCB  \n"
50    ::"i"(configMAX_SYSCALL_INTERRUPT_PRIORITY)     // representa %0
51      );
52  }
```

Listagem 6.2 – Implementação de chaveamento de contexto

Como visto na Listagem 5.2, o chaveamento de contexto, que está em *Handler Mode*, descarrega os registradores de propósito geral (ou seja, seu contexto) na pilha de memória da tarefa que deixará *pxReadyTasksLists*[σ] e será movida para *pxDelayedTaskList*. Logo após esse carregamento, as interrupções são desabilitadas (excetuando as mais prioritárias que *configMAX_SYSCALL_INTERRUPT_PRIORITY*) e então, a função *vTaskSwitchContext* é executada.

A maior fonte de variação temporal ocasionada pelo algoritmo de chaveamento de contexto se dá através da função presente em *vTaskSwitchContext* que seleciona a tarefa de mais alta prioridade, *taskSELECT_HIGHEST_PRIORITY_TASK*. Essa variação é causada pela busca em *pxReadyTasksLists*[σ] de uma lista que contenha uma tarefa pronta, a partir de uma variável que armazena a mais alta prioridade de uma tarefa pronta naquele instante, *uxTopReadyPriority*. A implementação dessa função é apresentada na Listagem 5.3. O pior caso de tempo de execução para essa função é quando uma tarefa de mais alta prioridade realiza um chaveamento de contexto para a tarefa ociosa. O tempo de execução mais baixo possível é o inverso, quando a tarefa ociosa deixa o processador e a tarefa que receberá recursos de processamento é a de mais alta prioridade.

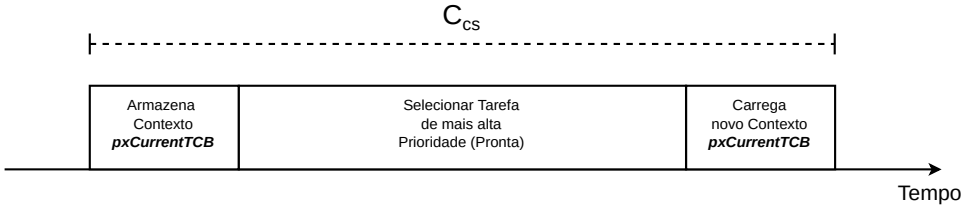
```
1  #define taskSELECT_HIGHEST_PRIORITY_TASK() {
2      UBaseType_t uxTopPriority = uxTopReadyPriority;
3
4      // Encontra a fila de maior prioridade que contem tarefas prontas.
```

```
5       while( listLIST_IS_EMPTY( &( pxReadyTasksLists[ uxTopPriority ] ))){
6           configASSERT( uxTopPriority );
7           uxTopPriority = uxTopPriority - 1; // no codigo original --uxPriority
8       }
9
10      // listGET_OWNER_OF_NEXT_ENTRY indexa atraves da lista,
11      // de modo que as tarefas de mesma prioridade obtem uma
12      // parcela igual do tempo do processador.
13      listGET_OWNER_OF_NEXT_ENTRY(pxCurrentTCB, &(pxReadyTasksLists[uxTopPriority]));
14      uxTopReadyPriority = uxTopPriority;
15  }
```

Listagem 6.3 – Implementação de seleção de tarefa de mais alta prioridade

Obviamente, a execução do algoritmo também está sujeita à variabilidade temporal causada pelos mecanismos de aceleração do microprocessador. No entanto, uma variação temporal mais sensível é percebida, mesmo que por vezes baixa, nos tempos da função de seleção da tarefa, que são dependentes do valor de prioridade da tarefa de mais alta prioridade do momento. O diagrama de tempo C_{cs} de um chaveamento de contexto é apresentado na Figura 18, em que a função *vTaskSwitchContext* é representada pela seleção de tarefa de mais alta prioridade.

Figura 18 – Comportamento do chaveamento de contexto



Fonte – Autor

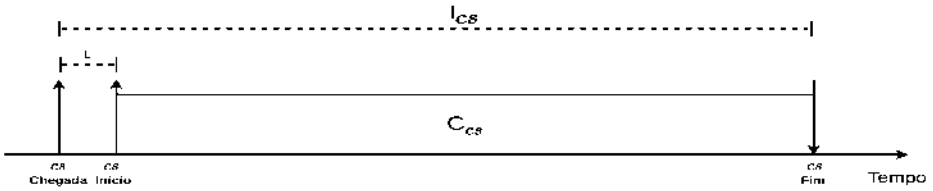
6.4.3 Modelo algébrico temporal

É apresentada na Figura 19 a influência I_{cs} de um chaveamento de contexto em uma tarefa no decorrer da execução do sistema. Em L é representado o tempo de atendimento da interrupção. Já C_{cs} representa o tempo de computação, apresentado pela Listagem 5.2, engloba a execução da função *vTaskSwitchContext*.

É apresentado na Equação (6.2) a influência temporal de chaveamento de contexto I_{cs} sobre uma tarefa programada pelo usuário.

$$I_{cs} = L + C_{cs} \tag{6.2}$$

Figura 19 – Comportamento temporal de chaveamento de contexto



Fonte – Autor

6.4.4 Análise temporal

Para a análise do tempo do *Tick*, é irrelevante definir as prioridades das tarefas. Como mostramos com o experimento do *Tick*, as chegadas de tarefas são fundamentais para a variabilidade do tempo de execução do *Tick*. No entanto, a variação do tempo de execução não é observada para a troca de contexto se utilizado o mesmo teste do experimento anterior, dado que as prioridades das tarefas do sistema são as mesmas. Logo, ao contrário do *Tick*, a variação do tempo de mudança de contexto está ligada às prioridades das tarefas.

Considerando um cenário em que quatro tarefas de usuário implementadas no *FreeRTOS* disputam recursos de processamento, são necessárias medições de trocas de contexto entre a tarefa analisada e as demais tarefas. Como o tempo de execução de *taskSELECT_HIGHEST_PRIORITY_TASK* é variável, é fundamental a distinção entre as trocas de contexto da tarefa analisada com as demais tarefas, sejam elas de alta prioridade ou não. São apresentadas na Tabela 8 as tarefas e seus respectivos tempos de recorrência e prioridade.

Tabela 8 – Tarefas para teste de chaveamento de contexto

Tarefa	**Período**	**Prioridade**
Tarefa A	5 ms	6 (Alta)
Tarefa B	6 ms	5 (Média-Alta)
Tarefa C	7 ms	4 (Média)
Tarefa D	8 ms	3 (Média-Baixa)
Tarefa IDLE	-	0 (Baixa)

Considerando que $CS(X, Y)$ representa o tempo de chaveamento de contexto de tarefa X para tarefa Y, para o cálculo de WCRT da tarefa A, o tempo de chaveamento de contexto para recebimento de recursos de processamento (*in*) entre $CS(B, A)$, $CS(C, A)$, $CS(D, A)$ e $CS(IDLE, A)$ são iguais, de um ponto de vista puramente lógico do *microkernel*, sendo que a variação em tempo de execução é sujeita apenas à

variação temporal imposta pelos mecanismos de aceleração da arquitetura. Isso é consequência da atribuição de *uxTopReadyPriority* que é a maior do sistema, causando apenas uma iteração no laço (*while*) presente na Listagem 5.7, independente da tarefa de mais baixa prioridade que terá os seus recursos de processamento retirados. Para ceder os recursos de processamento (*out*), a tarefa A possuirá quatro tempos de chaveamento de contexto distintos, o chaveamento de contexto de $CS(A, B)$, $CS(A, C)$, $CS(A, D)$ e $CS(A, IDLE)$.

No cálculo de WCRT da tarefa B, há dois tempos de chaveamento de contexto distintos para recebimento de recursos de processamento (*in*), $CS(A, B)$ e o conjunto de $CS(C, B)$, $CS(D, B)$ e $CS(IDLE, B)$ que são iguais. Para ceder os recursos de processamento (*out*), a tarefa B possuirá quatro tempos de chaveamento de contexto distintos, $CS(B, A)$, $CS(B, C)$, $CS(B, D)$ e $CS(B, IDLE)$. Para cálculo de WCRT da tarefa C, há três tempos de chaveamento de contexto distintos para recebimento de recursos de processamento (*in*), $CS(A, C)$, $CS(B, C)$ e o conjunto de $CS(D, C)$ e $CS(IDLE, C)$ que são iguais. Para ceder os recursos de processamento (*out*), a tarefa C possuirá três tempos de chaveamento de contexto distintos, o conjunto $CS(C, A)$, $CS(C, B)$, que são iguais e $CS(C, D)$ e $CS(C, IDLE)$.

Na tarefa D, para recebimento de recursos de processamento (*in*) há quatro tempos de chaveamento de contexto distintos, $CS(A, D)$, $CS(B, D)$, $CS(C, D)$ e $CS(IDLE, D)$. Para ceder os recursos de processamento (*out*), a tarefa D possuirá dois tempos de chaveamento de contexto distintos, o conjunto $CS(D, A)$, $CS(D, B)$ e $CS(D, C)$, que são iguais e $CS(C, IDLE)$. Finalmente, para a tarefa IDLE, para recebimento de recursos de processamento (*in*) há quatro tempos de chaveamento de contexto distintos, $CS(A, IDLE)$, $CS(B, IDLE)$, $SC(C, IDLE)$ e $CS(D, IDLE)$. Para ceder os recursos de processamento (*out*), a tarefa D possuirá quatro tempos iguais, o tempo de chaveamento de contexto do conjunto $CS(IDLE, A)$, $CS(IDLE, B)$, $CS(IDLE, C)$ e $CS(IDLE, D)$.

É apresentado no Quadro 6 o número de iterações no *loop* da função *taskSELECT_HIGHEST_PRIORITY_TASK* e o HWM das trocas de contexto entre tarefas da Tabela 8 após mil segundos de execução. São apresentadas no quadro as iterações que o laço da função *taskSELECT_HIGHEST_PRIORITY_TASK* realiza na busca da posição na lista em que *Y* se encontra em *pxReadyTasksLists*, vinculada ao valor de sua prioridade numérica. O número de iterações realizadas na seleção de uma tarefa de mais alta prioridade em um chaveamento de contexto (apresentado na Listagem 5.3) é resultado da subtração da prioridade da tarefa *X* de mais alta prioridade, que deixa os recursos de processamento, para a tarefa de baixa prioridade *Y*, que irá receber os recursos de processamento. Quando ocorre o inverso, em que uma tarefa de baixa prioridade tem seus recursos de processamento alocados para uma tarefa de alta prioridade, o número de iterações é 1, pois *uxTopReadyPriority* recebe o va-

lor de prioridade da tarefa *X*, que é a tarefa de mais alta prioridade no momento do chaveamento de contexto.

Quadro 6 – Iterações para trocas de contexto entre quatro tarefas e IDLE

Iterações	**Trocas de contexto**	**Tempo**
1 iteração	*CS(A,B), CS(B,A), CS(B,C), CS(C,A), CS(C,B), CS(C,D), CS(D,A), CS(D,B), CS(D,C), CS(IDLE,A), CS(IDLE,B), CS(IDLE,C)* e *CS(IDLE,D).*	$\approx 1{,}5us$
2 iterações	*CS*(*A*, *C*) e *CS*(*B*, *D*).	$\approx 2{,}875us$
3 iterações	*CS*(*A*, *D*) e *CS*(*D*, *IDLE*).	$\approx 3{,}255us$
4 iterações	*CS*(*B*, *IDLE*) e *CS*(*C*, *IDLE*).	$\approx 3{,}875us$
5 iterações	*CS*(*A*, *IDLE*).	$\approx 4{,}375us$

6.5 *INTERRUPTS*

Como discutido anteriormente, mecanismos de interrupção são independentes do *microkernel* para qualquer arquitetura que o *FreeRTOS* suporta. Essa decisão de projeto por parte dos projetistas do SOTR gera maior flexibilidade no uso de periféricos em um sistema embarcado, assim como retira os efeitos temporais do uso de mecanismos de escalonamento em uma hipotética disputa de recursos com outras tarefas de usuário programadas em *Timers* ou *Tasks*.

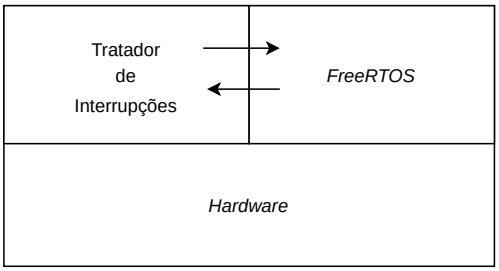
Em uma análise RTA é necessário que os tratadores de interrupção sejam considerados como uma tarefa e o seu comportamento deve ser previsível, tanto no trato de recorrência, onde eventos de interrupção devem possuir um intervalo mínimo entre chegadas, como no trato de sua prioridade em relação às demais.

Em uma camada puramente lógica, *Interrupts* estão no mesmo nível hierárquico de acesso ao hardware que o *microkernel*, representada na Figura 20. O *FreeRTOS* fornece APIs de comunicação entre *Interrupts* e o microkernel, como inserções de dados em filas compartilhadas entre tarefas e *Interrupts*, notificações de tarefas, grupos de eventos, e também mecanismos de sincronização como o ato de desabilitar interrupções, representada na Figura 20 pelas setas horizontais.

Por serem camadas distintas, ambas possuem sua própria política de prioridades entre tarefas. Na arquitetura *Cortex-M*4 a política de prioridades nas interrupções é decrescente numericamente conforme discutido no Capítulo 3, enquanto no *microkernel*, conforme discutido no Capítulo 4, a política de prioridades adotada é crescente numericamente.

Essa modelagem de tratamento de interrupções pelo *FreeRTOS* gera uma série de conclusões quanto ao comportamento temporal de uma tarefa programada em um *Interrupt*. A primeira conclusão é de que não há interferência de uma tarefa programada

Figura 20 – Camada hierárquica entre interrupção e hardware



Fonte – Autor

no *microkernel* sobre uma tarefa programada em um *Interrupt* no seu momento de chegada, exceto por tarefas de mais alta prioridade programadas no próprio *Interrupt*.

A segunda conclusão é que não há fontes de bloqueio entre tarefas programadas no *microkernel* e *Interrupts*, pois todas as funções de manipulação de estruturas (de final *fromISR*) que ligam a camada de tratador de interrupções ao *microkernel* têm suas fontes de bloqueio retiradas pelo projetista do SOTR. Não é possível, no contexto de execução e uso de ferramentas oferecidas pelo *FreeRTOS*, bloqueios entre tarefas do tipo *Interrupt*.

Release Jitters podem ocorrer em um *Interrupt* ocasionados por interrupções desabilitadas, seja pelo código interno ao *microkernel*, seja por *Tasks* implementadas ou por interrupções que utilizam seções críticas no decorrer da execução do sistema. Logicamente, por ser programada sobre um tratador de interrupção, uma tarefa sofre a latência do disparo da interrupção, que é um valor temporal dependente da arquitetura e da frequência de *clock*.

O conjunto *HPI* de *Interrupts j* de alta prioridade em que $j \in HPI$ que podem interferir na execução de um *Interrupt* são apenas as tarefas implementadas em outros *Interrupts*. O *overhead* causado pelo *microkernel* em *Interrupts* se limitam apenas ao *release jitter*, afinal, *Tick* e trocas de contexto são de mais baixa prioridade que qualquer *Interrupt* programada pelo projetista. *Tasks* também não interferem em tratadores de interrupção, pois são programados em *Thread Mode*, dentro do espaço gerenciado pelo SOTR.

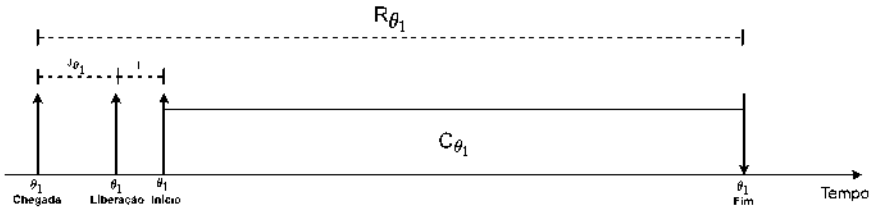
6.5.1 Um *Interrupt*

Neste cenário, considerando a execução de apenas uma tarefa θ_1 do tipo *Interrupt* sobre o sistema, é possível inferir seu tempo de resposta. Como discutido, *Interrupts* não sofrem influências temporais do SOTR exceto quando interrupções se encontram desabilitadas por ele. Isso pode ocorrer numa eventual chamada da arquitetura de *SysTick* para execução de *Tick* ou de *PendSV* para chaveamento de contexto antes da

chamada de um *Interrupt* θ_1. Também, através da macro *task_ENTER_CRITICAL* ou também através da macro *task_ENTER_CRITICAL_FROM_ISR* executado por qualquer tarefa pode fazer 0_1 sofrer um atraso.

É apresentado na Figura 21 um possível cenário do pior tempo de resposta de um *Interrupt*. Na Figura 21, um *Interrupt* inicia e conclui sua execução sem interferências, mas sofre um *release jitter* e a latência de atendimento de tratadores de interrupção da arquitetura.

Figura 21 – Tempo de Resposta de 0_1



Fonte – Autor

É apresentado na Equação (6.3) o tempo de resposta de uma única *Interrupt* executando no sistema. Em C_{θ_1} é representado o tempo de computação de um *Interrupt* θ, onde o pior caso $C_{\theta_1} = WCET_{\theta_1}$. O valor J_{θ_1} representa o maior tempo de *release jitter* causado por interrupções desabilitadas e L representa a latência temporal da arquitetura para atendimento da *Interrupt*.

$$R_{\theta_1} = J_{\theta_1} + L + C_{\theta_1} \tag{6.3}$$

6.5.2 Dois *Interrupts*

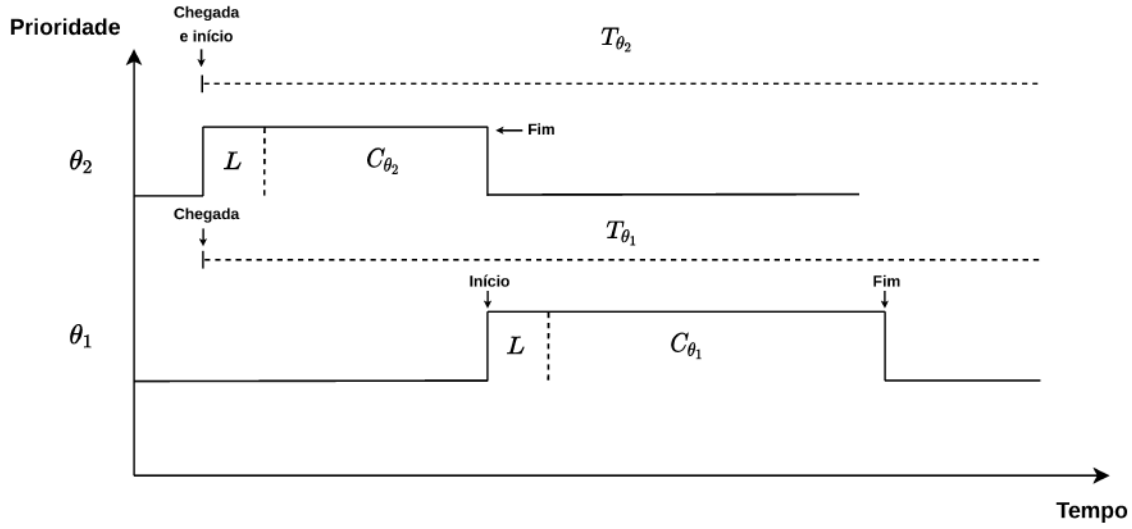
Neste cenário, dois *Interrupts* θ_1 e θ_2 de períodos e tempos de computação distintos executam no *FreeRTOS*, assim como *Tasks* programadas pelo usuário. Neste cenário, é também considerado que a prioridade de θ_1 é mais baixa em relação a θ_2. Durante toda a sua execução, o *Interrupt* θ_1 executa com interrupções habilitadas, padrão para a arquitetura.

O impacto da chegada de θ_1 no sistema nessas condições impõe a esse *Interrupt* um *release jitter* J_{0_1} que possui origem em interrupções desabilitadas. Também deve ser considerada a latência arquitetural de atendimento de interrupções, que está diretamente associada com a frequência de *clock* do sistema. O tempo de latência da arquitetura para atender *Interrupts* é representado por L, para a arquitetura *Cortex-M*.

É apresentado na Figura 22 um possível cenário de tempo de resposta de θ_1. Neste cenário, a *Interrupt* θ_1 sofre uma interferência de θ_2. A linha tracejada

verticalmente representa o fim do *overhead* temporal de latência (L) da arquitetura nos *Interrupts*, a partir de sua chegada no sistema.

Figura 22 – Tempo de Resposta de θ_1 sob interferência de θ_2



Fonte – Autor

O WCET de θ_2 é representado por C_{θ_2}, que por ter prioridade mais alta que θ_1 executa primeiro e finalmente, o WCET de θ_1 é representado por C_{θ_1}. Neste sentido, θ_1 pode ser interrompida por θ_2 a cada T_{θ_2}. É apresentado na Equação (6.4) o pior tempo de resposta de θ_1 neste cenário.

$$R_{\theta_1} = J_{\theta_1} + W_{\theta_1} \tag{6.4}$$

em que,

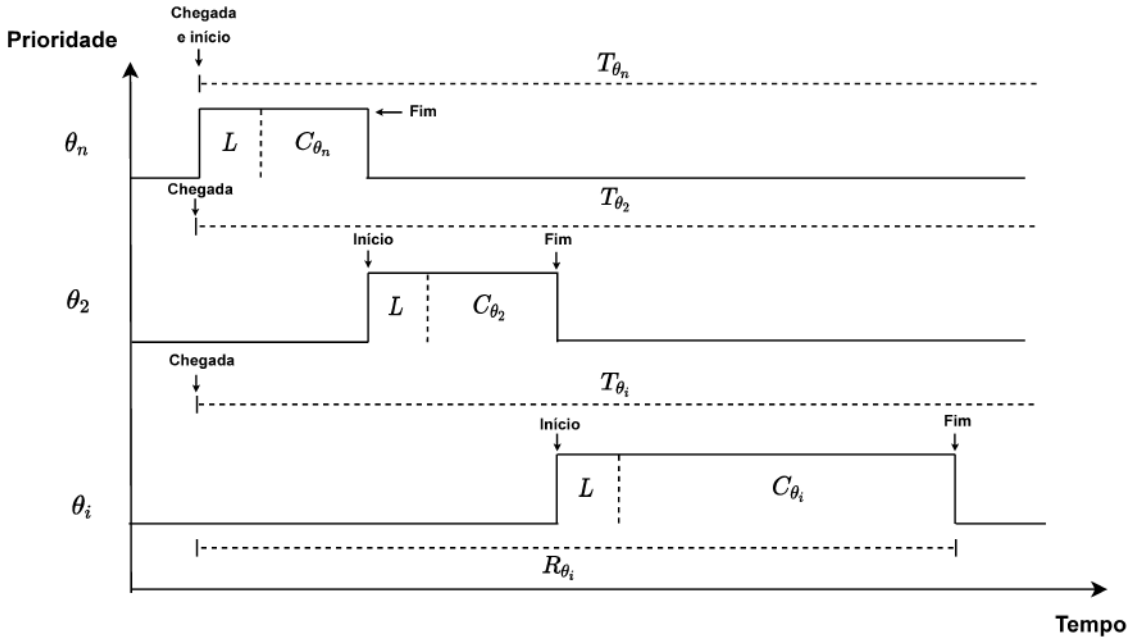
$$W_{\theta_1} = L + C_{\theta_1} + \left\lceil \frac{W_{\theta_1} + J_{\theta_2}}{T_{\theta_2}} \right\rceil \times (L + C_{\theta_2})$$

6.5.3 *N Interrupts*

Neste cenário, *n-1* tarefas do tipo *Interrupt*, θ_2 a θ_n pertencem a um conjunto de *Interrupts* denominado *High Priority Interrupts* (HPI). $HPI(\theta_i)$ denota o conjunto de *Interrupts* com prioridade mais alta do que θ_i. Considere θ_j, $j \in HPI(\theta_i)$ com distintos tempos de computação e recorrência que podem interferir na tarefa θ_i. No pior cenário possível para θ_i todas as tarefas pertencentes a $HPI(\theta_i)$ chegam ao mesmo tempo, aumentando o seu tempo de resposta. Neste cenário, é considerado também que a chegada das tarefas de mais alta prioridade que θ_i possuem seu tempo de computação igual ao seu WCET.

É apresentado um possível cenário de pior tempo de resposta de θ_i na Figura 23. Neste cenário, θ_i sofre interferências de todas as *Interrupts* de mais alta prioridade do sistema. *L* representa a latência arquitetural e a linha tracejada horizontalmente representa o fim do *overhead*, com o início da computação do *Interrupt*.

Figura 23 – Tempo de Resposta de θ_i sob *n* interferências de *Interrupts*



Fonte – Autor

É apresentado na Equação (6.5) um possível cenário de tempo de resposta R_{θ_i} para θ_i executando em uma arquitetura embarcada com o *FreeRTOS*. Para esta modelagem algébrica é considerado que θ_i executa com interrupções habilitadas, padrão para a arquitetura *Cortex-M*.

$$R_{\theta_i} = J_{\theta_i} + W_{\theta_i} \tag{6.5}$$

em que,

$$W_{\theta_i} = L + C_{\theta_i} + \sum_{j \in HPI(\theta_i)} \left\lceil \frac{W_{\theta_i} + J_j}{T_j} \right\rceil \times (L + C_j)$$

6.6 *TASK*

Uma *Task* é toda a tarefa criada no *FreeRTOS* a partir da função *xTaskCreate*. A quantidade de tarefas programadas em um sistema embarcado está atrelada aos requisitos de projeto e à quantidade de espaço disponível em memória, portanto, é

necessário mensurar o impacto de outras *Tasks* e *Interrupts* sobre uma *Task* analisada. O *FreeRTOS* em uma *Task* permite a utilização de todos os recursos de sistema antes apresentados, de forma distinta das tarefas de usuário criadas em *Timers* ou *Interrupts* em que uma operação de *lock* em um *mutex* não pode ser bloqueante ou mesmo funções de atraso não podem ser chamadas. Isso acarreta uma maior complexidade de análise temporal, em que há a possibilidade de eventos de interrupção, distintas fontes de bloqueio e *release jitters*. As subseções seguintes tratam dessas fontes de variação temporal em uma tarefa do tipo *Task*. As análises abordadas por esta seção consideram que *Tasks* executam com interrupções habilitadas, padrão para a arquitetura *Cortex-M*.

6.6.1 Fontes de atraso

Em *Tasks* um *release jitter* pode se dar de duas formas. A primeira, é o desabilitar de interrupções por outras *Tasks, Timers, Interrupts* ou pelo *microkernel*. No *microkernel*, durante o *Tick*, são desabilitas as interrupções para analisar *pxDelayedTaskList* em busca de *Tasks* a serem alocadas na fila de prontos. A ordem que os *jobs* estão em *pxDelayedTaskList* em *Tick* se baseia o tempo (em ordem crescente) no qual elas devem ser ativadas. Isso significa que uma *Task* de baixa prioridade pode ser alocada na fila dos prontos antes de uma *Task* de alta prioridade, ainda que as duas possuam o mesmo período. Por exemplo, considerando um cenário em que duas *Tasks*, uma de alta prioridade (γ_2) e outra de baixa prioridade (γ_1) possuam períodos distintos. Em um determinado ponto de execução, γ_1 faz uma chamada de *vTaskDelay*, que a coloca na fila *pxDelayedTaskList*. Supondo que o tempo de *delay* coincida com a chegada de γ_2 é possível que γ_1 preceda γ_2 em *pxDelayedTaskList* e γ_1 seja liberada antes, aumentando o *release jitter* da *Task* de alta prioridade. No pior cenário possível, γ_2 se encontrará na última posição da lista de *pxDelayedTaskList*.

É apresentado em Farines, Fraga e Oliveira (2000) o conceito de atividade, que é um encapsulamento lógico de tarefas que se comunicam através de mensagens ou se sincronizam em uma relação de precedência. Em outras palavras, uma tarefa pode ficar bloqueada aguardando uma mensagem ou a ocorrência de um evento por outra tarefa, logo, o tempo que a tarefa espera a mensagem pode ser considerado como o seu *release jitter*. O *release jitter* da tarefa de uma determinada atividade que aguarda o evento ou mensagem no pior caso é o WCRT da tarefa sincronizante ou emissora da mensagem. Isto posto, uma outra fonte de *release jitter* em uma tarefa de usuário pode ser a adoção de estruturas de notificação, como *ulTaskNotifyTake*, recepção de dados em fila sem *timeout* ou espera por eventos no decorrer da execução através de *xEventGroupWaitBits*.

O *FreeRTOS*, além de um mecanismo para desabilitar interrupções, proporciona para *Tasks* semáforos binários e semáforos contadores para gerenciamento de seções

críticas. De forma a evitar a inversão descontrolada de prioridades, que gera atrasos maiores para *Tasks* de alta prioridade, o *microkernel* proporciona também a implementação de um *mutex* a partir da função *xSemaphoreTake* com protocolo PIP (ver Seção 2.3.3). Para cenários recursivos, um tipo de *mutex* recursivo é disponibilizado.

Podem ocorrer interferências de tarefas de alta prioridade, sejam elas programadas em *Interrupts* ou *Tasks* com prioridade superior a tarefa em análise, como ocorre com *Tick*, uma tarefa do *microkernel* programada sob um tratador de interrupção de forma periódica e que pode interromper toda e qualquer *Task* em tempo de execução. *Tasks* periódicas são programadas pelo projetista através da função de atraso *vTaskDelayUntil*, como apresentado na Listagem 4.7. Ela influencia no tempo de resposta de uma *Task* ao inserir a tarefa em *pxDelayedTaskList* e desabilitar interrupções por alguns ciclos de *clock*.

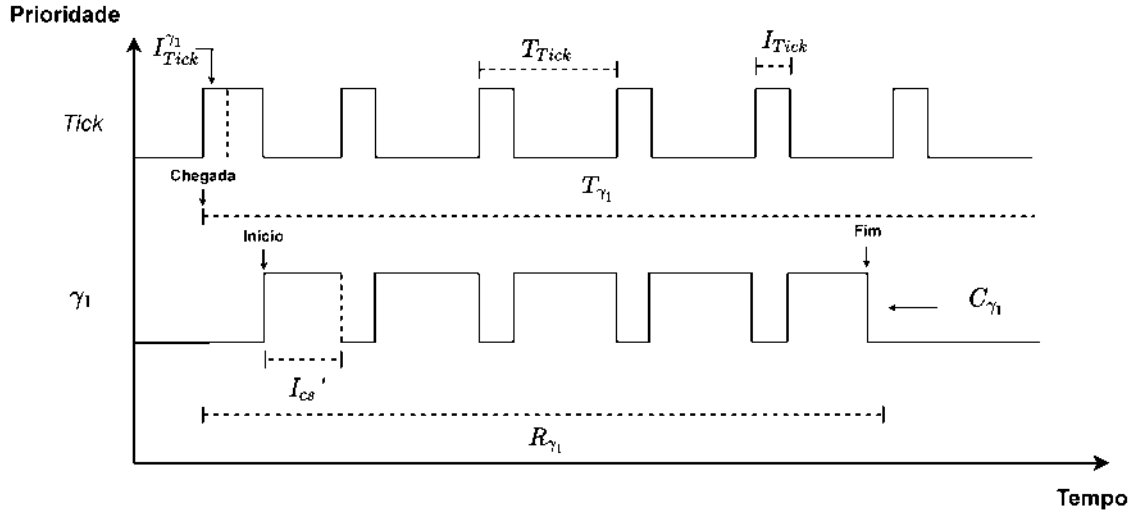
6.6.2 Uma *Task*

Nesse cenário, uma *Task* γ_1 executando no sistema possui uma periodicidade T_1. Após a liberação de γ_1, um chaveamento de contexto é necessário. Os tempos de chaveamento de contexto I'_{cs}, como discutido na seção anterior, dependem da prioridade da *Task* anteriormente executada. No cenário analisado o tempo de computação de chaveamento de contexto é constante em que, se o *deadline* nunca é perdido, então I'_{cs} é o tempo total de troca de monopólio de processador da tarefa ociosa para a *Task* γ_1, que para este cenário é a mais prioritária do sistema.

A *Task* γ_1 possui um tempo de computação C_{γ_1}, que certamente também pode sofrer interferências de *Tick*, representado por C_{Tick}, se o tempo de resposta de γ_1 for maior que T_{Tick}. Logo, é necessário em uma análise de WCRT incluir o tempo de computação de *Tick*, representado pela soma de seu tempo de computação acrescido da latência para atendimento do tratador de interrupção de *Tick*, $L + C_{Tick}$. É necessário acrescentar na janela de execução de γ_1 os tempos de computação de C_{Tick} no decorrer de suas ativações. O total de interferências de *Tick* em γ_1 é denotado por $\left\lceil \frac{W_{\gamma_1}+J_{Tick}}{T_{Tick}} \right\rceil$. Portanto, para obtenção do tempo total de atraso imposto por *Tick* sobre γ_1 basta multiplicar o WCET de *Tick* pelo número de interferências causadas sobre ela.

É apresentado um possível cenário de tempo de resposta de uma *Task* γ_1 na Figura 24. Neste cenário, γ_1 sofre interferências de *Tick* a cada T_{Tick}. É apresentada também a interferência da pseudo-tarefa resultante do particionamento de *Tick*, que aumenta seu tempo de resposta. A linha tracejada verticalmente representa o final do *overhead* de chaveamento de contexto para γ_1.

Figura 24 – Tempo de Resposta de γ_1



Fonte – Autor

A interferência temporal de *Tick* é expressa por:

$$\left\lceil \frac{W_{\gamma_1} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

I_{Tick} é o tempo total de interferência de cada *Tick*. J_{Tick} é o seu tempo de *release jitter*. A execução de *Tick* ocorre periodicamente a cada T_{Tick}. A interferência temporal da pseudo-tarefa imposta ao sistema por *Tick* é dada por $I^{\gamma_1}_{tick}$.

O WCRT de γ_1 é apresentado na Equação (6.6).

$$R_{\gamma_1} = J_{\gamma_1} + W_{\gamma_1} \tag{6.6}$$

em que,

$$W_{\gamma_1} = I^{\gamma_1}_{Tick} + (I'_{cs} + C_{\gamma_1}) + \left\lceil \frac{W_{\gamma_1} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

6.6.3 Duas *Tasks* sem seção crítica compartilhada

A subseção anterior apresentou o WCRT de γ_1 quando nenhuma outra *Task* disputa o monopólio de processamento com ela. Neste novo cenário, duas *Tasks* γ_1 e γ_2 possuem prioridade e periodicidade distintas. A *Task* γ_1 possui a prioridade mais baixa do sistema e γ_2 possui a prioridade mais alta do sistema. A *Task* γ_1 possuirá um comportamento distinto do observado anteriormente após a inserção de γ_2 no sistema.

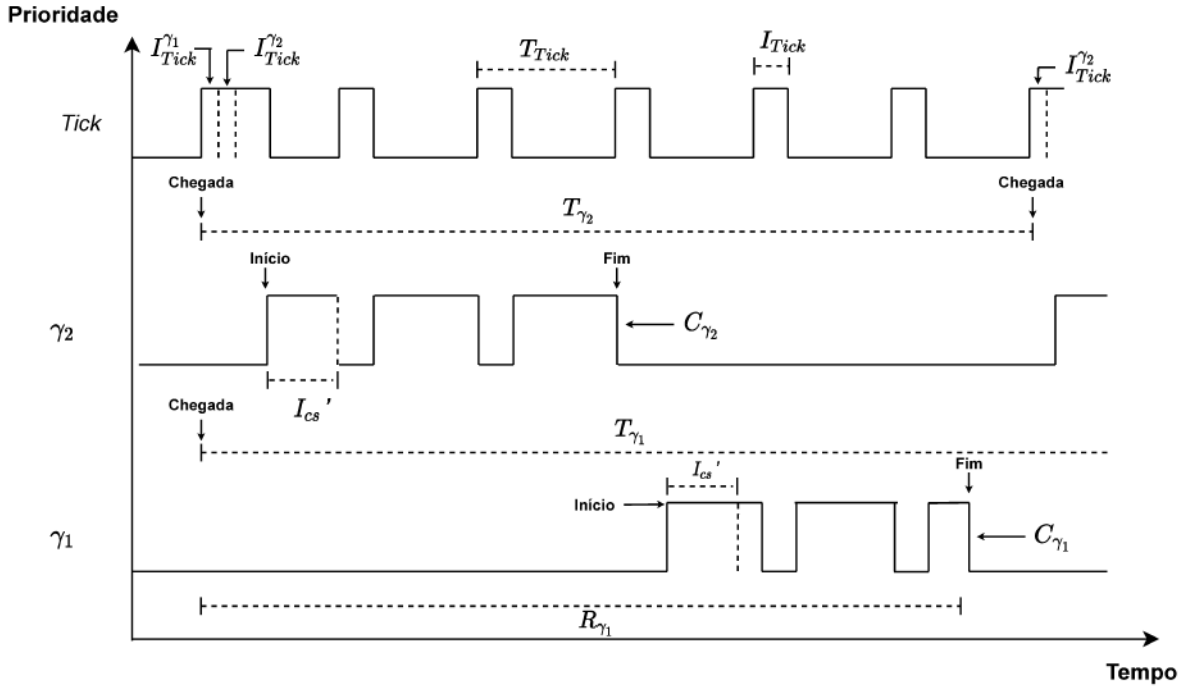
O instante crítico de análise se torna o momento do instante de chegada simultânea de γ_1 e γ_2. No momento da execução das *Tasks*, pelo microprocessador estar

com interrupções desabilitadas por causa da execução de *Tick*, é imposto um *release jitter* J_{γ_2} na *Task* γ_2. Um *release jitter* J_{γ_1} também pode ocorrer em γ_1 quando interrupções são desabilitadas por γ_2. Uma outra mudança é o tempo entre chaveamentos de contexto, que depende da tarefa que anteriormente monopolizava os recursos de processamento. Caso o *deadline* de γ_2 nunca seja perdido, considerando que o *deadline* seja implícito, a *Task* que antes do instante crítico monopolizava os recursos de processamento é a tarefa ociosa.

O tempo total de troca de contexto da tarefa ociosa para γ_2 é representado por I'_{cs}. Após a execução de γ_2, o tempo de chaveamento de contexto para γ_1 é alterado em relação a I'_{cs}, gerando um novo I''_{cs}. Durante a execução de γ_1 se $R_{\gamma_1} \geq T_{Tick}$, certamente haverá interferências de *Tick* sobre γ_1 a cada período de ativação de *Tick*.

É apresentado na Figura 25 um possível cenário de tempo de resposta de γ_1 a partir de influências temporais de γ_2 e de *overheads* do *microkernel*. No instante crítico, em que as duas *Tasks* e *Tick* chegam no sistema, γ_1 sofre interferências de *Tick* a cada T_{Tick} e sofre interferências de γ_2 a cada T_{γ_2}. As linhas tracejadas verticais representam o fim dos *overheads* sobre as *Tasks*, oriundas de chaveamento de contexto e *Tick*, oriundas das pseudo-tarefas do sistema.

Figura 25 – Tempo de resposta de γ_1 sob interferência de γ_2



Fonte – Autor

A interferência temporal de *Tick* é expressa por:

$$\left\lceil \frac{W_{\gamma_1} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

I_{Tick} é o tempo total de interferência de cada *Tick* no sistema. J_{Tick} é o seu tempo de *release jitter*. A execução de *Tick* ocorre periodicamente a cada T_{Tick}. A interferência temporal das pseudo-tarefas impostas ao sistema por *Tick* é dada por $I_{tick}^{\gamma_1}$ e $I_{tick}^{\gamma_2}$.

O WCRT de γ_1 para este cenário é apresentado na Equação (6.7).

$$R_{\gamma_1} = J_{\gamma_1} + W_{\gamma_1} \tag{6.7}$$

em que,

$$W_{\gamma_1} = I_{Tick}^{\gamma_1} + (I'_{cs} + C_{\gamma_1}) + \left\lceil \frac{W_{\gamma_1} + J_{\gamma_2}}{T_{\gamma_2}} \right\rceil \times (I'_{cs} + C_{\gamma_2} + I_{Tick}^{\gamma_2}) + \left\lceil \frac{W_{\gamma_1} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

6.6.4 *N Tasks* sem seção crítica compartilhada

Neste cenário, γ_i possui a prioridade mais baixa que todas as *Tasks* com mais alta prioridade executando no sistema. É considerado neste cenário que todas as *Tasks* são independentes entre si, isto é, não possuem seções críticas entre elas, portanto, γ_i não impõe bloqueio para *Tasks* de mais alta prioridade. Além das fontes de interferência causadas por *Tick*, as interferências de *Tasks* sobre γ_i ocorrem a partir do instante crítico. O número de interferências de γ_j de mais alta prioridade do sistema sobre γ_i é dado por $\left\lceil \frac{W_{\gamma_i}+J_{\gamma_j}}{T_{\gamma_j}} \right\rceil$ em que J_{γ_j} representa o *release jitter* de γ_j pelo desabilitar de interrupções por outras *Tasks* no decorrer da execução do sistema.

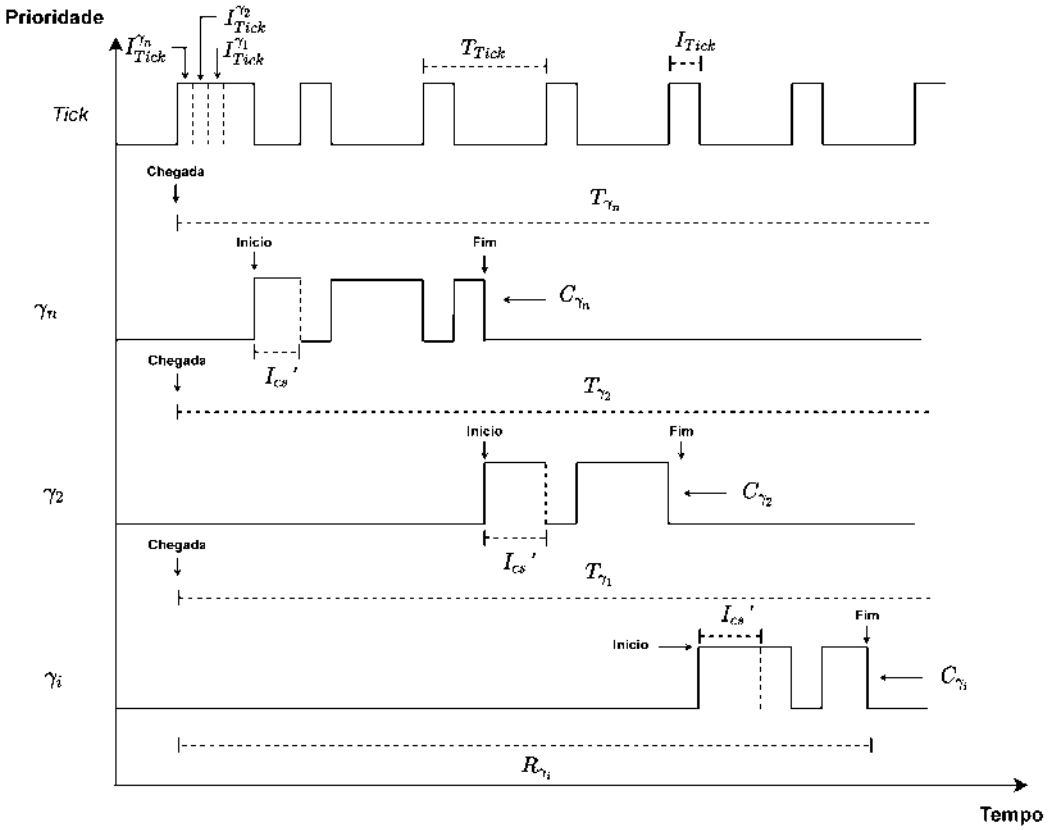
Para cada interferência de γ_j é imposta a γ_i a execução por duas vezes do tratador de interrupção *PendSV*, ou seja, o chaveamento de contexto de saída de γ_i para ceder recursos de processamento para γ_j e sua posterior devolução para γ_i.

É apresentado na Figura 26 um possível cenário de tempo de resposta de uma *Task* γ_i. No instante crítico, todas as *Tasks* e *Tick* chegam ao mesmo tempo, atrasando o tempo de resposta de γ_i. No cenário, as linhas tracejadas em *Tick* representam o fim da execução das pseudo-tarefas resultantes de seu particionamento e nas *Tasks* representam o fim dos *overheads* resultantes do chaveamento de contexto do *micro-kernel*.

A interferência temporal do conjunto de *Tasks* γ de mais alta prioridade que γ_i é expressa por:

$$\sum_{j \in HPT(\gamma_i)} \left\lceil \frac{W_{\gamma_i} + J_j}{T_j} \right\rceil \times [I_{Tick}^{j} + (I'_{cs_j} + C_j + I''_{cs_j})]$$

Figura 26 – Tempo de resposta de γ_i sob interferências de *n Tasks*



Fonte – Autor

C_j é o tempo de computação de cada *Task*, I'_{cs_j} o tempo total de troca de contexto de acesso ao processador e I''_{cs_j} o tempo total de troca de contexto de saída do processador. J_j representa o *release jitter* da *Task*. I^j_{tick} é o tempo de computação da pseudo-tarefa resultante do particionamento de *Tick*. A *Task* que cria essa nova pseudo-tarefa é de mais alta prioridade que γ_i. A interferência temporal de *Tick* é expressa por:

$$\left\lceil \frac{W_{\gamma_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

I_{Tick} é o tempo total de interferência de cada *Tick*. J_{Tick} é o seu tempo de *release jitter*. A execução de *Tick* ocorre periodicamente a cada T_{Tick}. A interferência temporal das demais pseudo-tarefas impostas ao sistema por *Tick* é dada por:

$$\sum_{f \in (\Gamma - HPT(\gamma_i))} (I^f_{Tick})$$

O conjunto Γ representa todas as *Tasks* do sistema. As *Tasks* que geram as pseudo-tarefas em *Tick* no somatório são γ_i e as demais *Tasks* de mais baixa pri-

oridade que γ_i. I^f_{Tick} representa o tempo de computação de cada pseudo-tarefa. É apresentado na Equação (6.8) o WCRT de γ_i sob interferência de *n Tasks*.

$$R_{\gamma_i} = J_{\gamma_i} + W_{\gamma_i} \tag{6.8}$$

em que,

$$W_{\gamma_i} = \sum_{f \in (\Gamma - HPT(\gamma_i))} (I^f_{Tick}) + (I'_{cs} + C_{\gamma_i}) + \left\lceil \frac{W_{\gamma_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick} + \sum_{j \in HPT(\gamma_i)} \left\lceil \frac{W_{\gamma_i} + J_j}{T_j} \right\rceil \times [I^j_{Tick} + (I'_{cs_j} + C_j + I''_{cs_j})]$$

6.6.5 *Tasks* sem seção crítica compartilhada com *Interrupts*

Expandindo o cenário da subseção anterior, considera-se nessa seção um cenário com *n Interrupts* de mais alta prioridade que *Tasks* γ, no qual deseja-se analisar o comportamento de γ_i. O conjunto Θ representa todas as *Interrupts* do sistema. O número de interferências do *Interrupt k*, sendo *k* a *Interrupt* de mais alta prioridade, sobre a *Task* γ_i é dado por $\left\lceil \frac{W_{\gamma_i} + J_k}{T_k} \right\rceil$.

É apresentado na Figura 27 um possível cenário de tempo de resposta de uma *Task* γ_i que sofre interferência de *Interrupts*, outras *Tasks* e do *Tick*. No instante crítico, todas elas chegam ao mesmo tempo, aumentando o tempo de resposta de γ_i. As linhas verticais tracejadas representam, a partir da chegada, o fim do *overhead* arquitetural para *Interrupts*, o fim da execução das pseudo-tarefas para *Tick*, e o fim do chaveamento de contexto para *Tasks*.

Como um *Interrupt* é programado sobre um tratador de interrupção, é necessário considerar a latência de arquitetura, *L*, e seu tempo de computação, e o *release jitter* J_k. A interferência temporal do conjunto de *Interrupts* Θ de mais alta prioridade que γ_i do sistema é expressa por:

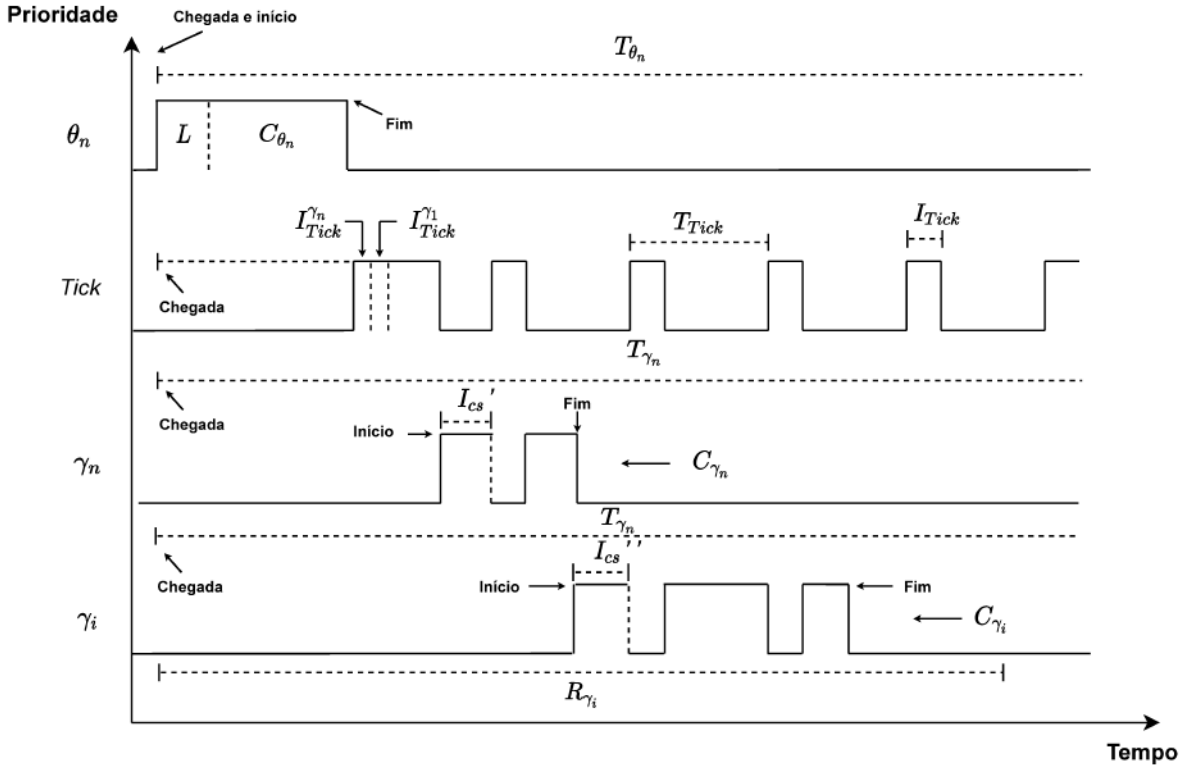
$$\sum_{k \in \Theta} \left\lceil \frac{W_{\gamma_i} + J_k}{T_k} \right\rceil \times (L + C_k)$$

A interferência temporal do conjunto de *Tasks* γ de mais alta prioridade que γ_i é expressa por:

$$\sum_{j \in HPT(\gamma_i)} \left\lceil \frac{W_{\gamma_i} + J_j}{T_j} \right\rceil \times [I^j_{Tick} + (I'_{cs_j} + C_j + I''_{cs_j})]$$

C_j é o tempo de computação de cada *Task*, I'_{cs_j} o tempo total de troca de contexto de acesso ao processador e I''_{cs_j} o tempo total de troca de contexto de saída do processador. J_j representa o *release jitter* da *Task*. Já I^j_{Tick} é o tempo de computação

Figura 27 – Tempo de resposta de γ_i sob múltiplas interferências



Fonte – Autor

da pseudo-tarefa resultante do particionamento de *Tick*. A *Task* que cria essa nova pseudo-tarefa é de mais alta prioridade que γ_i. A interferência temporal de *Tick* é expressa por:

$$\left\lceil \frac{W_{\gamma_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

I_{Tick} é o tempo total de interferência de cada *Tick*. J_{Tick} é o seu tempo de *release jitter*. A execução de *Tick* ocorre periodicamente a cada T_{Tick}. A interferência temporal das demais pseudo-tarefas resultantes do particionamento de *Tick* é dada por:

$$\sum_{f \in (\Gamma - HPT(\gamma_i))} (I_{Tick}^{f})$$

O conjunto Γ representa todas as *Tasks* do sistema. As *Tasks* que geram as pseudo-tarefas em *Tick* no somatório são γ_i e as demais *Tasks* de mais baixa prioridade que γ_i. I_{Tick}^{f} representa o tempo de computação de cada pseudo-tarefa. É apresentado na Equação (6.9) o WCRT de γ_i.

$$R_{\gamma_i} = J_{\gamma_i} + W_{\gamma_i} \tag{6.9}$$

em que,

$$W_{\gamma_i} = \sum_{f \in (\Gamma - HPT(\gamma_i))} (I^f_{Tick}) + (I'_{cs} + C_{\gamma_i}) + \left\lceil \frac{W_{\gamma_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick} +$$
$$\sum_{k \in \Theta} \left\lceil \frac{W_{\gamma_i} + J_k}{T_k} \right\rceil \times (L + C_k) + \sum_{j \in HPT(\gamma_i)} \left\lceil \frac{W_{\gamma_i} + J_j}{T_j} \right\rceil \times [I^j_{Tick} + (I'_{cs_j} + C_j + I''_{cs_j})]$$

6.6.6 *Tasks* com seção crítica compartilhada com *Interrupts*

Tasks γ_0 de mais baixa prioridade que compartilham algum *mutex* com *Tasks* γ_i de mais alta prioridade podem possuir um período de ativação distinto. O conjunto Θ representa todas as *Interrupts* do sistema. No decorrer da execução de um sistema sobre um SOTR, uma *Task* γ_i pode ser bloqueada por γ_0 de mais baixa prioridade. Expandindo o modelo construído na subseção anterior, será tratado o efeito de *Tasks* de mais baixa prioridade que possuem seções compartilhadas com *Tasks* de mais alta prioridade.

Supondo um instante que γ_0 de mais baixa prioridade faz uma operação de *lock* com sucesso em um *mutex* compartilhado e γ_i também realiza a operação de *lock* no mesmo *mutex*, ocorrerá um bloqueio B em γ_i. Na presença de um ou mais *mutexes* entre as *Tasks*, o maior tempo de bloqueio B para estimativa de WCRT de γ_i é complexo, dado que o algoritmo de herança de prioridade é adotado pelo *microkernel*.

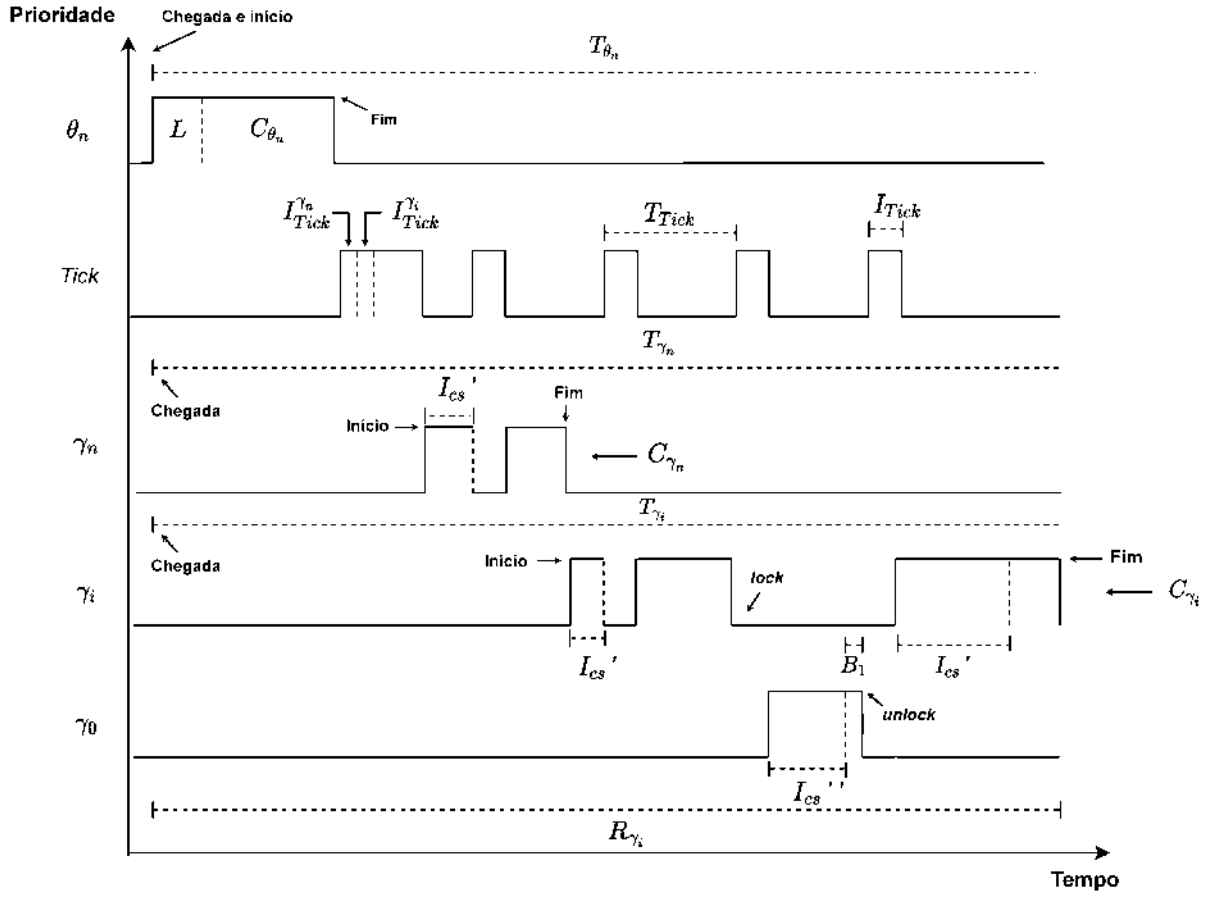
O algoritmo *Computing B_i For The Basic Inheritance Protocol* (RAJKUMAR, 1991) utiliza um conceito de árvore de busca para estimativa do tempo de bloqueio. Existem simplificações, como o algoritmo *Block Time Computation* (BUTTAZZO, 2011) que desconsidera aninhamentos de *mutexes*. Há também como desabilitar interrupções para implementação de seções críticas, com o pior tempo de bloqueio já determinado por Sha, Rajkumar e Lehoczky (1990).

Independente da forma de organização das seções críticas, B será considerado no modelo como o pior tempo de bloqueio de γ_i. Na presença de bloqueio há o tempo de chaveamento de contexto para a tarefa de mais baixa prioridade. Caso a *Task* não sofra bloqueio mas imponha bloqueio a uma outra de mais alta prioridade, um chaveamento de contexto é realizado pelo *microkernel* entregando o recurso a γ_i.

É apresentado na Figura 28 um possível cenário de tempo de resposta para a *Task* γ_i com influências temporais de *Tasks* de mais alta prioridade, de *Interrupts* e do *Tick*. Também é considerada a influência de uma seção crítica compartilhada com uma *Task* γ_0 de mais baixa prioridade, que impõe um bloqueio B_i a γ_i. No instante crítico, *Interrupts, Tasks* de mais alta prioridade e o *Tick* chegam ao mesmo tempo, aumentando o tempo de resposta de γ_i. As linhas verticais tracejadas representam o fim dos *overheads*, a partir da chegada no sistema. Em *Interrupts*, é representado

o fim da latência arquitetural, em *Tick*, o fim da execução do particionamento das pseudo-tarefas e nas *Tasks*, o fim do chaveamento de contexto do *microkernel*.

Figura 28 – Tempo de resposta de γ_i sob múltiplas interferências com bloqueio



Fonte – Autor

A interferência temporal do conjunto de *Interrupts* Θ de mais alta prioridade que γ_i do sistema é expressa por:

$$\sum_{k \in \Theta} \left\lceil \frac{W_{\gamma_i} + J_k}{T_k} \right\rceil \times (L + C_k)$$

A interferência temporal do conjunto de *Tasks* de mais alta prioridade que γ_i é expressa por:

$$\sum_{j \in HPT(\gamma_i)} \left\lceil \frac{W_{\gamma_i} + J_j}{T_j} \right\rceil \times [I^j_{Tick} + (I'_{cs_j} + C_j + I''_{cs_j})]$$

C_j é o tempo de computação de cada *Task*, I'_{cs_j} o tempo total de troca de contexto de acesso ao processador e I''_{cs_j} o tempo total de troca de contexto de saída do processador. J_j representa o *release jitter* da *Task* de mais alta prioridade. Já I^j_{Tick} representa o tempo de computação da pseudo-tarefa resultante do particionamento

de *Tick*. A *Task* que cria essa nova pseudo-tarefa é de mais alta prioridade que γ_i. A interferência temporal de *Tick* é expressa por:

$$\left\lceil \frac{W_{\gamma_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

I_{Tick} é o tempo de total de interferência de cada *Tick*. J_{Tick} é o seu tempo de *release jitter*. A execução de *Tick* ocorre periodicamente a cada T_{Tick}. A interferência temporal das demais pseudo-tarefas impostas ao sistema por *Tick* é dada por:

$$\sum_{f \in (\Gamma - HPT(\gamma_i))} (I^f_{Tick})$$

O conjunto Γ representa todas as *Tasks* do sistema. As *Tasks* que criam as pseudo-tarefas em *Tick* no somatório são γ_i e as demais *Tasks* de mais baixa prioridade que γ_i. I^f_{Tick} representa o tempo de computação de cada pseudo-tarefa. É apresentado na Equação (6.10) o WCRT de γ_i.

$$R_{\gamma_i} = J_{\gamma_i} + W_{\gamma_i} \tag{6.10}$$

em que,

$$W_{\gamma_i} = \sum_{f \in (\Gamma - HPT(\gamma_i))} (I^f_{Tick}) + (I'_{cs} + C_{\gamma_i}) + \omega + \left\lceil \frac{W_{\gamma_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$
$$+ \sum_{j \in HPT(\gamma_i)} \left\lceil \frac{W_{\gamma_i} + J_j}{T_j} \right\rceil \times [I^j_{Tick} + (I'_{cs} + C_j + I''_{cs})] + \sum_{k \in \Theta} \left\lceil \frac{W_{\gamma_i} + J_k}{T_k} \right\rceil \times (L + C_k)$$

$\omega = I'_{cs} + I''_{cs}$ se a *Task* de baixa prioridade impõe bloqueio a uma *Task* de alta prioridade e $\omega = 2I'_{cs} + B_{\gamma_i} + I''_{cs}$ se a *Task* sofre bloqueio de uma tarefa de baixa prioridade.

6.7 *TIMERS*

Assim como outros sistemas operacionais voltados para tempo real, o *FreeRTOS* também fornece *Timers* (denominados também temporizadores) independentes de hardware. Ao contrário do que pode ser intuitivo em que cada *Timer* programado cria uma tarefa distinta, o *microkernel* utiliza uma *Task* denominada *Tmr Svc* (ou *Daemon Task*) que, no período de ativação de uma tarefa implementada em um temporizador, esta é selecionada na lista de *Timers* programados para ser executada por *Tmr Svc*. Essa escolha de implementação de *Timers* pelos projetistas do *microkernel* causa impacto na própria *Task*, assim como nas outras *Tasks* implementadas no sistema.

Por padrão, *Timers* possuem prioridade igual a *configTIMER_TASK_PRIORITY*, que geralmente é mais baixa que *Tasks* programadas no sistema, logo, para análise

será considerado que *Tmr Svc* sempre possuirá prioridade mais baixa que as *Tasks* do sistema. É apresentado na Listagem 4.11 do Capítulo 4 o código de *Tmr Svc*, que é implementada na função *portTASK_FUNCTION*.

6.7.1 Fontes de atraso

A *Task Tmr Svc* apresentada na Seção A.2, pode sofrer *release jitter* por interrupções desabilitadas pelas demais *Tasks* do sistema no seu período de ativação. Como demonstrado na Listagem 4.11, a definição de qual temporizador deve executar é feita pela função *prvGetNextExpireTime*, que também impõe a um *Timer* qualquer criado pelo projetista um atraso na computação de código presente em seu *handler* de função, que busca na lista *pxCurrentTimerList* o *Timer* que necessita ser ativado no instante. Como a lista *pxCurrentTimerList* é ordenada ascendentemente por tempo de expiração, o temporizador a ser executado em *Tmr Svc* está sempre presente na primeira posição da lista, logo, isso significa que *prvGetNextExpireTime* possui baixa variação temporal.

A função *prvProcessTimerOrBlockTask* na Listagem 4.11 é também uma fonte de atraso na computação de um *Timer*. O atraso que a função causa em um temporizador programado é devido a algumas poucas comparações entre o tempo atual e o tempo que o *Timer* presente na primeira posição de *pxCurrentTimerList*, que vai definir se o *Timer* expirou ou não. Caso o *Timer* expire, ocorre um atraso por *prvProcessTimerOrBlockTask*. Ao final da execução do temporizador, o *Timer* δ_i é novamente "suspenso" através do pedido de chaveamento de contexto por essa função.

Em um código programado pelo usuário que está contido em um temporizador não é possível acrescentar trechos de código bloqueantes ao *Timer*. Isso significa que é possível utilizar um *mutex* no temporizador somente se o tempo de bloqueio não for maior que zero, o que na prática elimina a possibilidade de gerenciamento de seções críticas em um *Timer*.

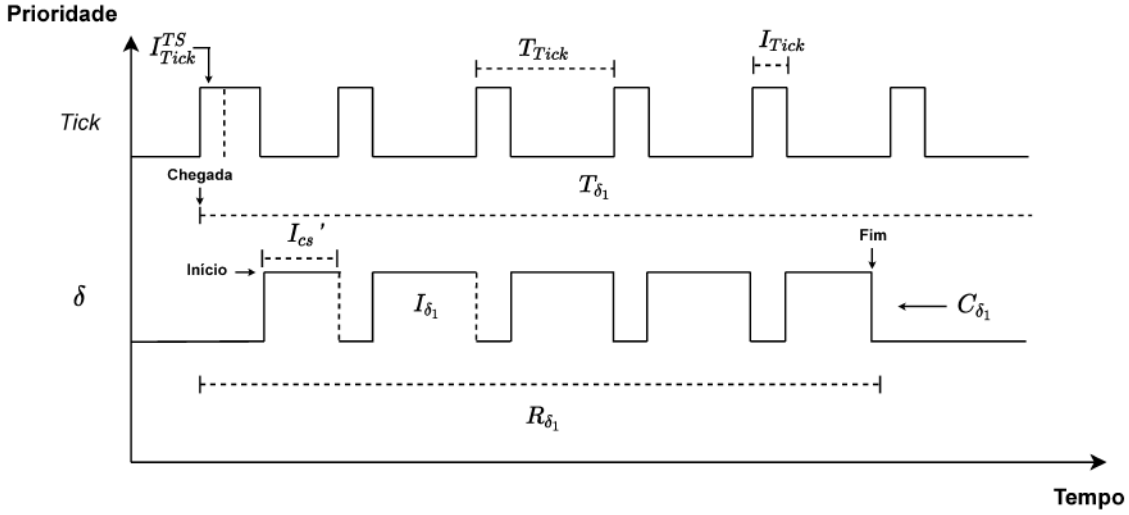
6.7.2 Um *Timer*

Neste cenário, é considerada a execução de um *Timer* δ_1 no sistema, com ativação periódica. No cálculo de WCRT de δ_1 são considerados os efeitos de *Tick*, de trocas de contexto e controles de *Tmr Svc* sobre o temporizador. Como *Tmr Svc* é programado sobre uma *Task*, no momento em que um *Timer* deve ser ativado, em seu instante crítico, pode ocorrer um atraso J_{δ_1}. Um chaveamento de contexto com um tempo total de troca de contexto I_{cs}, também é necessário. Após o chaveamento de contexto para *Tmr Svc*, o *Timer* demora um tempo I_{δ_1} na manipulação da lista de *Timers* do sistema para encontrar o respectivo *handler* (ou função) do temporizador.

É apresentado na Figura 29 um possível cenário de tempo de resposta de um *Timer* δ_1 em seu instante crítico. No cenário, *Tick* interfere em *Timers* a cada T_{Tick},

o que aumenta seu tempo de resposta. As linhas tracejadas verticais representam o fim de *overheads* sobre *Timers*, para chaveamento de contexto e manipulação da lista *pxCurrentTimerList*. Já para *Tick*, a linha tracejada representa o fim da execução da pseudo-tarefa resultante do particionamento de *Tick*, por conta da liberação de *Tmr Svc* (I_{Tick}^{TS}).

Figura 29 – Tempo de resposta de δ_1



Fonte – Autor

A interferência temporal de *Tick* é expressa por:

$$\left\lceil \frac{W_{\delta_1} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

I_{Tick} é o tempo total de interferência de cada *Tick*. J_{Tick} é o seu tempo de *release jitter*. A execução de *Tick* ocorre periodicamente a cada T_{Tick}. A interferência temporal da pseudo-tarefa imposta ao sistema por *Tick*, por conta da liberação de *Tmr Svc*, é dada por I_{tick}^{TS}.

É apresentado na Equação (6.11) o WCRT de δ_1.

$$R_{\delta_1} = J_{\delta_1} + W_{\delta_1} \tag{6.11}$$

em que,

$$W_{\delta_1} = I_{Tick}^{TS} + (I'_{cs} + I_{\delta_1} + C_{\delta_1}) + \left\lceil \frac{W_{\delta_1} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

6.7.3 Dois *Timers*

Neste cenário, um *Timer* δ_2 com o tempo de computação C_{δ_2} e recorrência T_{δ_2} é programado no *microkernel*. O instante crítico de δ_1 se dá no mesmo instante

temporal de ativação de δ_2. Como são programados sob a mesma *Task*, não há trocas de contexto ativadas pela interrupção *PendSV* da arquitetura como observado em outras *Tasks* programadas pelo usuário.

No entanto, há *overhead* por parte de *Tmr Svc* na organização da lista de *Timers* ativos, em que se encontram δ_1 e δ_2, antes da chamada dos *handlers* de cada *Timer*. O pior caso para δ_1 neste cenário é a execução completa de δ_2 antes de sua própria execução pelo microprocessador. Como a fila de *Timers* trabalha com lógica FIFO, isso pode ocorrer caso, por exemplo, δ_1 tenha sido executado antes da próxima ativação de δ_2, que coincidentemente, será ativado junto com δ_1. Nesse caso, *Tmr Svc* executará primeiro δ_2, e somente depois de δ_1.

No instante crítico de δ_1 as interrupções estarão desabilitadas, impondo a δ_1 um *release jitter* J_{δ_1}. Após o término de *Tick*, um chaveamento de contexto com um tempo total de I_{cs} é realizado para *Tmr Svc*. Após receber o monopólio de processamento, *Tmr Svc* avalia a fila de comandos e seleciona o *Timer* que se encontra no início da fila, que no instante crítico de δ_1, é δ_2, gerando um *overhead* temporal I_{δ_2}. Ao término da execução de δ_2, é imposto um custo temporal a δ_1 de C_{δ_2}. Após a execução de δ_2, *Tmr Svc* manipula novamente a lista de *Timers*, de forma a executar o *handler* de δ_1, gerando um *overhead* temporal I_{δ_1}.

É apresentado na Figura 30 um possível cenário de tempo de resposta de um *Timer* δ_1 em seu instante crítico. No cenário, *Tick* interfere em *Timers* a cada T_{Tick}, o que aumenta seu tempo de resposta. As linhas tracejadas verticais representam o fim de *overheads* sobre *Timers*, para chaveamento de contexto e manipulação da lista *pxCurrentTimerList*. Já para *Tick*, a linha tracejada representa o fim de execução da pseudo-tarefa resultante do particionamento de *Tick*, causada pela liberação de *Tmr Svc* (I^{TS}_{Tick}).

É possível no decorrer da execução de δ_1 interferências de *Tick* a cada T_{Tick} e um *release jitter* a cada J_{Tick}. A interferência temporal é expressa por:

$$\left\lceil \frac{W_{\delta_1} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

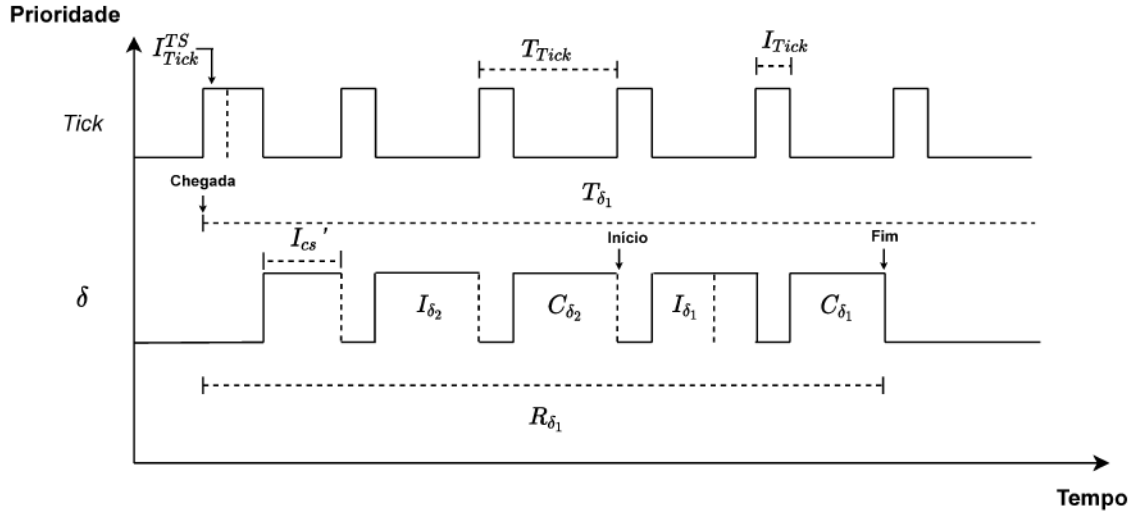
I_{Tick} é o tempo total de interferência de cada *Tick*. J_{Tick} é o seu tempo de *release jitter*. A execução de *Tick* ocorre periodicamente a cada T_{Tick}. A interferência temporal da pseudo-tarefa imposta ao sistema em *Tick*, por conta da liberação de *Tmr Svc*, é dada por I^{TS}_{tick}. É apresentado na Equação (6.12) o WCRT do *Timer* δ_1.

$$R_{\delta_1} = J_{\delta_1} + W_{\delta_1} \tag{6.12}$$

em que,

$$W_{\delta_1} = I^{TS}_{Tick} + (I'_{cs} + [I_{\delta_2} + C_{\delta_2}] + I_{\delta_1} + C_{\delta_1}) + \left\lceil \frac{W_{\delta_1} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

Figura 30 – Tempo de resposta de δ_1 sob influência de δ_2



Fonte – Autor

6.7.4 Vários *Timers*

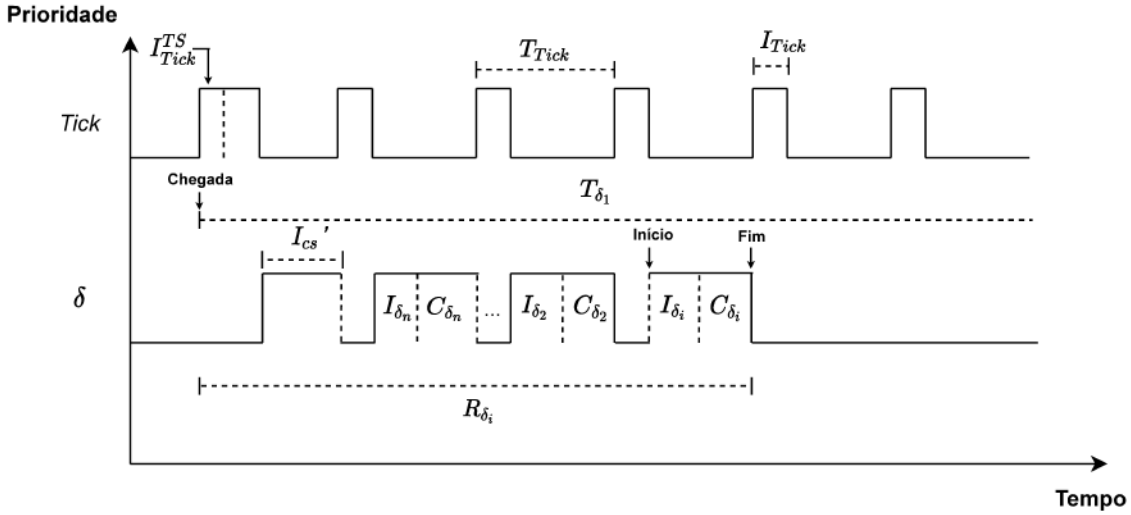
Neste cenário é considerado um conjunto de *n Timers* que executam no sistema e influenciam o comportamento temporal de δ_i. No instante crítico de δ_i, a partir da ativação de δ_i, todos os *Timers* do sistema são ativados em conjunto com ele, gerando interferências e atrasando o início de sua execução. Como o δ_i depende de sua posição na lista de *Timers*, a ordem dos *Timers* que atrasam a liberação de execução de δ_i é aleatória. Todavia, o instante crítico é considerado quando todos os *Timers* são ativados e executam antes de δ_i.

Como fontes de atraso há a influência de interferências de *Tick* a cada T_{Tick}, e a partir da liberação de δ_i em seu instante crítico, no instante de liberação da *Task Tmr Svc*, as interrupções estão desabilitadas, impondo a δ_i um *release jitter* J_{δ_i}. Após a liberação de *Tmr Svc*, um chaveamento de contexto com tempo I_{cs} é realizado. Nesse instante, δ_i estará no fim da fila e todos os *Timers* do sistema são executados antes dele, por causa do padrão FIFO de atendimento do *microkernel*.

É apresentado na Figura 31 um possível cenário de tempo de resposta de um *Timer* δ_i em seu instante crítico. No cenário, *Tick* interfere em *Timers* a cada T_{Tick}, o que aumenta seu tempo de resposta. As linhas tracejadas verticais representam o fim de *overheads* sobre *Timers*, para chaveamento de contexto e manipulação da lista *pxCurrentTimerList*. Já para *Tick*, a linha tracejada representa o fim de execução da pseudo-tarefa resultante do particionamento de *Tick*, causado pela liberação de *Tmr Svc* (I^{TS}_{Tick}). Outros *Timers* executam antes de δ_i, aumentando o seu tempo de resposta.

No momento de execução de δ_i, há o *overhead* I_{δ_i} causado por *Tmr Svc*, que

Figura 31 – Tempo de resposta de δ_i sob influência de vários *Timers*



Fonte – Autor

executará o *handler* de δ_i. Interferências de *Tick* também ocorrem a cada T_{Tick}. A interferência temporal é expressa por:

$$\left\lceil \frac{W_{\delta_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

I_{Tick} é o tempo total de interferência de cada *Tick*. J_{Tick} é o seu tempo de *release jitter*. A execução de *Tick* ocorre periodicamente a cada T_{Tick}. A interferência temporal da pseudo-tarefa imposta ao sistema por *Tick*, por conta da liberação de *Tmr Svc*, é dada por I^{TS}_{Tick}. A interferência temporal do conjunto Δ de todos os *Timers* do sistema, excetuando δ_i é dada por:

$$\sum_{j \in \Delta \& j \neq i} (I_{\delta_j} + C_{\delta_j})$$

I_{δ_j} é o tempo que *Tmr Svc* demora manipulando a lista de *Timers* e C_{δ_j} o tempo de computação do *Timer*.

É apresentado na Equação (6.13) o WCRT de δ_i.

$$R_{\delta_i} = J_{\delta_i} + W_{\delta_i} \tag{6.13}$$

em que,

$$W_{\delta_i} = I^{TS}_{Tick} + \sum_{j \in \Delta \& j \neq i} (I_{\delta_j} + C_{\delta_j}) + (I'_{cs} + I_{\delta_i} + C_{\delta_i}) + \left\lceil \frac{W_{\delta_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

6.7.5 Vários *Timers* e vários *Interrupts*

Expandindo o cenário da subseção anterior e introduzindo no sistema a presença de *Interrupts*, o comportamento temporal de δ_i é alterado. O conjunto Θ representa todas as *Interrupts* do sistema. No instante crítico, todos os *Interrupts* do sistema aumentam o tempo de resposta do *Timer*. Como cada *Interrupt* θ_i é periódico, δ_i sofrerá a imposição dos tempos de latência e computação de cada um dos *Interrupts* programados no sistema, afinal, *Interrupts* possuem prioridade mais alta em relação a um *Timer*, que é gerenciado pela *Task Tmr Svc* sob gestão pelo *microkernel*.

No pior cenário possível para δ_i, um *release jitter* J_{δ_i} ocorre no momento de ativação de *Tmr Svc*, que executará após o chaveamento de contexto I_{cs}. Todas as *Interrupts* do sistema serão executadas e δ_i estará no fim da lista de *Timers* a serem ativados, logo, todos os *Timers* do sistema também serão executados antes de δ_i.

É apresentado na Figura 32 um possível cenário de tempo de resposta de um *Timer* δ_i em seu instante crítico. No cenário, *Interrupts* chegam ao mesmo tempo de um *Timer*. Também, *Tick* interfere em *Timers* a cada T_{Tick}, o que aumenta seu tempo de resposta. As linhas tracejadas verticais representam o fim de *overheads* sobre *Timers*, para chaveamento de contexto e manipulação da lista *pxCurrentTimerList*. Já para *Tick*, a linha tracejada representa o fim de execução da pseudo-tarefa resultante do particionamento de *Tick*, causada pela liberação de *Tmr Svc* (I^{TS}_{Tick}). Em *Interrupts*, a linha tracejada representa o fim da execução da latência arquitetural sobre um *Interrupt*. Outros *Timers* executam antes de δ_i, aumentando o seu tempo de resposta.

A interferência temporal do conjunto de *Interrupts* θ de mais alta prioridade que os *Timers* do sistema é expressa por:

$$\sum_{k \in \Theta} \left\lceil \frac{W_{\delta_i} + J_k}{T_k} \right\rceil \times (L + C_k)$$

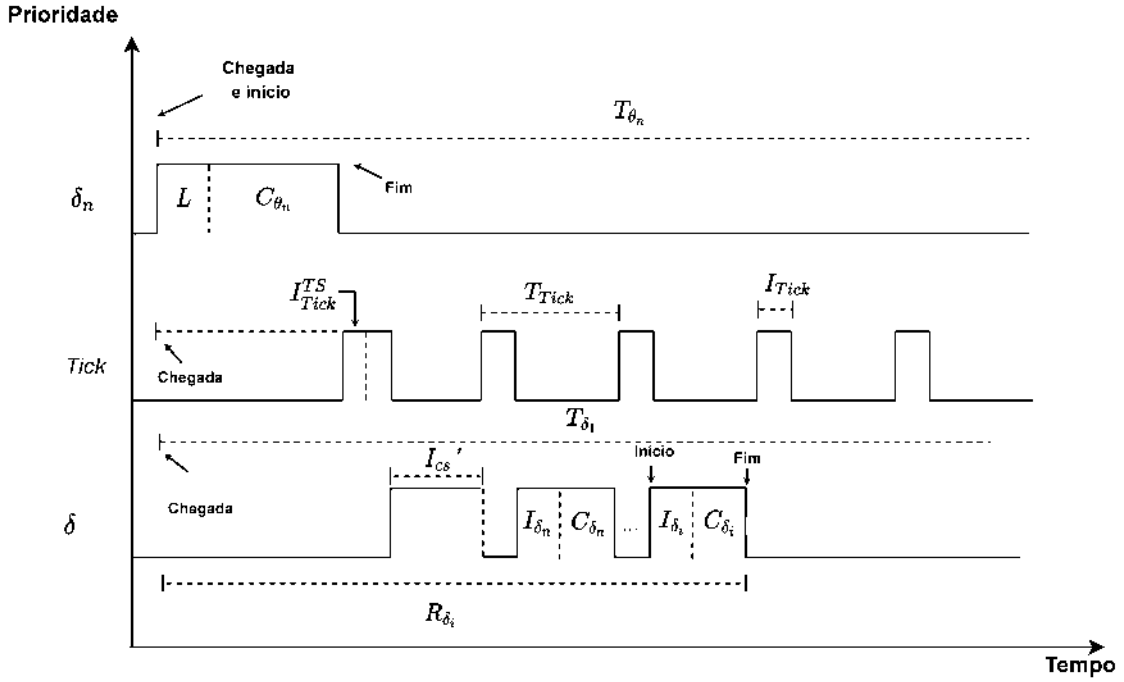
C_k é o tempo de computação de cada *Interrupt*, L a latência da arquitetura e J_k o *release jitter*. A cada T_k também ocorrerá uma execução de um *Interrupt*, atrasando o término de execução do *Timer*. Já a interferência temporal de *Tick* é expressa por:

$$\left\lceil \frac{W_{\delta_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

I_{Tick} é o tempo total de interferência de cada *Tick*. J_{Tick} é o seu tempo de *release jitter*. A execução de *Tick* ocorre periodicamente a cada T_{Tick}. A interferência temporal da pseudo-tarefa imposta ao sistema por *Tick* é dada por I^{TS}_{Tick}. A interferência temporal do conjunto Δ de todos os *Timers* do sistema, excetuando δ_i, é dada por:

$$\sum_{j \in \Delta \& j \neq i} (I_{\delta_j} + C_{\delta_j})$$

Figura 32 – Tempo de resposta de δ_i sob influência de *n Timers* e *Interrupts*



Fonte – Autor

I_{δ_j} é o tempo que *Tmr Svc* demora manipulando a lista de *Timers* e C_{δ_j} o tempo de computação do *Timer*. É apresentado na Equação (6.14) o WCRT de δ_i.

$$R_{\delta_i} = J_{\delta_i} + W_{\delta_i} \tag{6.14}$$

em que,

$$W_{\delta_i} = (I_{Tick}^{TS}) + (I'_{cs} + I_{\delta_i} + C_{\delta_i}) + \sum_{j \in \Lambda \& j \neq i} (I_{\delta_j} + C_{\delta_j}) + \left\lceil \frac{W_{\delta_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick} + \sum_{k \in \Theta} \left\lceil \frac{W_{\delta_i} + J_k}{T_k} \right\rceil \times (L + C_k)$$

6.7.6 Vários *Timers*, vários *Interrupts* e várias *Tasks*

Expandido o modelo da subseção anterior com a presença de *Tasks* executando em um sistema em conjunto com *Timers* e *Interrupts*, o comportamento temporal de δ_i é alterado. O conjunto Γ representa todas as *Tasks* do sistema, já o conjunto Θ representa todas as *Interrupts* do sistema. No instante crítico, uma *Task* ou todas as *Tasks* chegando no sistema em conjunto com *Interrupts* aumentam o tempo de resposta do *Timer*. Como cada *Task* γ é periódica, δ_i sofrerá a imposição dos tempos de trocas de contexto e computação de cada um das *Tasks* programadas no sistema,

afinal, neste cenário considera-se que *Tasks* possuem prioridade mais alta em relação a um *Timer*, que é gerenciado pela *Task Tmr Svc* sob gestão pelo *microkernel*.

Portanto, no pior cenário possível para δ_i, um *release jitter* J_{δ_i} ocorre no momento de ativação de *Tmr Svc*, que executará após o chaveamento de contexto I'_{cs}. Todas as *Interrupts* serão executadas, assim como as *Tasks*. O *Timer* δ_i estará no fim da lista de *Timers* a serem ativados, logo, todos os *Timers* do sistema também serão executados antes de δ_i.

É apresentado na Figura 33 um possível cenário de tempo de resposta de um *Timer* δ_i em seu instante crítico. No cenário, *Interrupts* e *Tasks* chegam ao mesmo tempo de um *Timer*. O *Tick* também interfere em *Timers* a cada T_{Tick}, o que aumenta seu tempo de resposta. As linhas tracejadas verticais representam o fim de *overheads* sobre *Timers* e *Tasks*, para chaveamento de contexto, e também o *overhead* em *Timers*, I_δ, para manipulação da lista *pxCurrentTimerList*. Já para *Tick*, a linha tracejada representa o fim da execução da pseudo-tarefa resultante do particionamento de *Tick*, causada pela liberação de *Tmr Svc* (I^{TS}_{Tick}). Em *Interrupts*, a linha tracejada representa o fim da execução da latência arquitetural sobre um *Interrupt*. Outros *Timers* executam antes de δ_i, aumentando o seu tempo de resposta.

A interferência temporal do conjunto de *Interrupts* θ de mais alta prioridade que os *Timers* do sistema é expressa por:

$$\sum_{k \in \Theta} \left\lceil \frac{W_{\delta_i} + J_k}{T_k} \right\rceil \times (L + C_k)$$

C_k é o tempo de computação de cada *Interrupt*, L a latência da arquitetura e J_k o *release jitter*. A cada T_k também ocorrerá uma execução de um *Interrupt*, atrasando o término de execução do *Timer*.

A interferência temporal do conjunto de *Tasks* γ do sistema é expressa por:

$$\sum_{l \in HPT(\gamma_{TS})} \left\lceil \frac{W_{\delta_i} + J_l}{T_l} \right\rceil \times [I^l_{Tick} + (I'_{cs_l} + C_l + I''_{cs_l})]$$

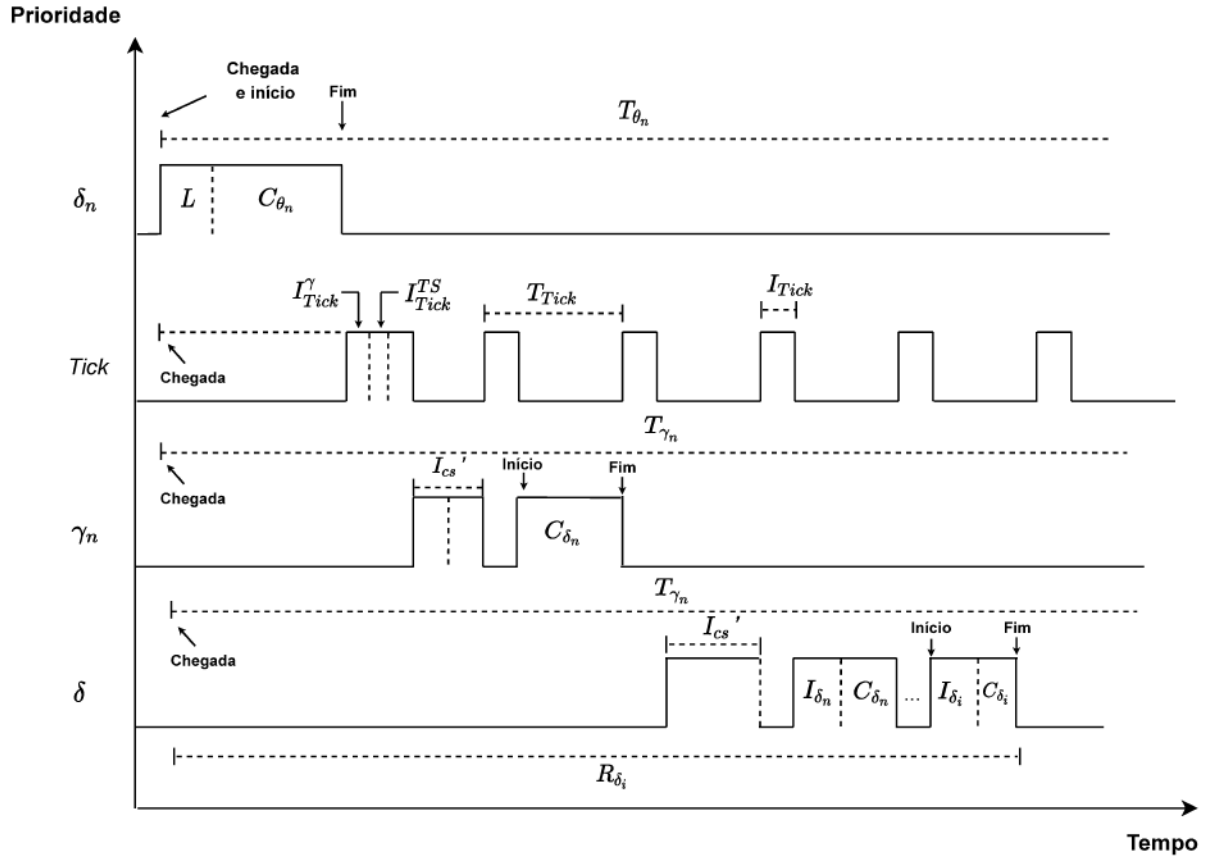
C_l é o tempo de computação de cada *Task*, I'_{cs_l} o tempo total de troca de contexto de acesso ao processador e I''_{cs_l} o tempo total de troca de contexto de saída do processador. J_l representa o *release jitter* da *Task* de mais alta prioridade. γ_{TS} representa a *Task Tmr Svc*. Já I^l_{Tick} representa o tempo de computação da pseudo-tarefa resultante do particionamento de *Tick*. A *Task* que cria essa nova pseudo-tarefa é de mais alta prioridade que γ_i.

A interferência temporal de *Tick* é expressa por:

$$\left\lceil \frac{W_{\delta_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

I_{Tick} é o tempo total de interferência de cada *Tick*. J_{Tick} é o seu tempo de *release jitter*. A execução de *Tick* ocorre periodicamente a cada T_{Tick}.

Figura 33 – Tempo de resposta de δ_i com diversas interferências



Fonte – Autor

A interferência temporal do conjunto de todos *Timers* Δ do sistema, excetuando δ_i, é dada por:

$$\sum_{j \in \Delta \& j \neq i} (I_{\delta_j} + C_{\delta_j})$$

I_{δ_j} é o tempo que *Tmr Svc* demora manipulando a lista de *Timers* e C_{δ_j} o tempo de computação do *Timer* de maior período.

É apresentado na Equação (6.15) o WCRT de δ_i.

$$R_{\delta_i} = J_{\delta_i} + W_{\delta_i} \tag{6.15}$$

em que,

$$W_{\delta_i} = I^{TS}_{Tick} + (I'_{cs} + I_{\delta_i} + C_{\delta_i}) + \sum_{j \in \Delta \& j \neq i} (I_{\delta_j} + C_{\delta_j}) + \left\lceil \frac{W_{\delta_i} + J_{Tick}}{T_{Tick}} \right\rceil \times I_{Tick}$$

$$+ \sum_{k \in \Theta} \left\lceil \frac{W_{\delta_i} + J_k}{T_k} \right\rceil \times (L + C_k) + \sum_{l \in HPT(\gamma_{TS})} \left\lceil \frac{W_{\delta_i} + J_l}{T_l} \right\rceil \times [I^{l}_{Tick} + (I'_{cs_l} + C_l + I''_{cs_l})]$$

6.8 RESUMO

Neste capítulo foram apresentadas as premissas e respectivas terminologias adotadas para análise temporal de tarefas executando sobre o SOTR *FreeRTOS*. Foi discutido o comportamento temporal de *Tick*, que possui suas fontes de atraso devido a interrupções desabilitadas e execução de outras *Tasks* ou *Interrupts*, assim como foi discutida a possibilidade de *Tick* ser dividida em duas ou mais pseudo-tarefas. Foi discutida a variação temporal em operações de chaveamento de contexto no *microkernel*, que geralmente é considerado na literatura como uma constante acrescentada ao tempo de computação de uma tarefa (OLIVEIRA, 2018). Na prática, no entanto, o tempo de execução de um chaveamento de contexto no *FreeRTOS* varia conforme a prioridade da *Task* de destino no sistema.

Interrupts têm prioridade mais alta que todas as *Tasks* sob gestão do *microkernel*. *Interrupts* possuem atraso também por conta de interrupções desabilitadas e podem sofrer interferências por outros *Interrupts* com prioridade mais alta. Assim como *Timers*, *Interrupts* não podem implementar códigos que possuam possibilidade de bloqueio, como *mutex* com espera indefinida. *Interrupts* se comunicam com tarefas do *microkernel* com funções específicas e todos os trechos de computação dessas funções específicas de sistema possuem bloqueio igual a zero.

Tasks sob gestão do *microkernel* possuem fontes de atraso como *release jitter* por conta de interrupções desabilitadas, assim como atrasos também podem ocorrer em uma *Task* de mais baixa prioridade por tarefas implementadas em *Interrupts* ou *Tasks* de mais alta prioridade. Bloqueios em *Tasks* podem ocorrer por conta de operações de *lock* em tarefas de baixa prioridade, impondo atraso temporal considerável, a depender de como a seção crítica está programada. O protocolo ao qual o *mutex* é implementado é o PIP, sendo possível também utilizar semáforos binários, contadores ou *mutexes* recursivos.

Timers nativos do *FreeRTOS* são programados sob a tarefa *Tmr Svc*, que possui fontes de atraso como interrupções desabilitadas e processamento de funções de gerenciamento de *Timers* no *microkernel*. É possível também a interferência de execução de um *Timer* por *Tasks* e *Interrupts*, sendo que um *Timer* nunca interrompe a execução de outro *Timer*, contudo, seu tempo de computação pode atrasar a execução de um *Timer* no fim de uma lista FIFO criada por *Tmr Svc*. *Timers* não podem programar funções que coloquem a tarefa sob possibilidade de bloqueio, como *lock* em *mutex* com espera indefinida ou um *timeout* de função maior que zero.

Na Seção 6.5, na Seção 6.6 e na Seção 6.7 foram apresentadas as equações referentes aos tempos de resposta de tarefas implementadas pelo usuário, sejam elas programadas em *Interrupts*, *Tasks* ou *Timers*, adaptando o modelo algébrico temporal de WCRT demonstrado em Audsley *et al.* (1993) para o contexto de execução de tarefas no *FreeRTOS*.

São apresentados na Tabela 9 as equações de WCRT para tarefas no *FreeRTOS*. Seja *i* uma *Task*, *Interrupt* ou *Timer*. A Tabela é composta de duas colunas: A primeira mostra o conjunto de tarefas de alta prioridade que podem influenciar o tempo de resposta de *i*. Na segunda coluna, é apresentada a respectiva equação de WCRT de *i*.

Tabela 9 – Equações para WCRT de tarefas no *FreeRTOS*

Tarefa *i* do tipo *Interrupt*	
Conjunto de tarefas $HP_{(i)}$	**Equação**
Nenhuma	Equação 6.3
Um *Interrupt*	Equação 6.4
Dois ou mais *Interrupts*	Equação 6.5
Tarefa *i* do tipo *Task*	
Conjunto de tarefas $HP_{(i)}$	**Equação**
Nenhuma	Equação 6.6
Uma *Task*	Equação 6.7
Duas ou mais *Tasks* (sem seção crítica)	Equação 6.8
Duas ou mais *Tasks* e *Interrupts* (sem seção crítica)	Equação 6.9
Duas ou mais *Tasks* e *Interrupts* (com seção crítica)	Equação 6.10
Tarefa *i* do tipo *Timers*	
Conjunto de tarefas $HP_{(i)}$	**Equação**
Nenhuma	Equação 6.11
Um *Timer*	Equação 6.12
Dois ou mais *Timers*	Equação 6.13
Dois ou mais *Timers* e *Interrupts*	Equação 6.14
Dois ou mais *Timers*, *Interrupts* e *Tasks*	Equação 6.15

7 TESTES TEMPORAIS NA PLATAFORMA EXPERIMENTAL

Neste capítulo serão apresentados testes de execução de *Interrupts*, *Tasks* e *Timers* executando no SOTR *FreeRTOS*. O objetivo deste capítulo é também avaliar algebricamente o pior tempo de resposta de tarefas executando no *microkernel* sobre um microprocessador *single core*, baseando-se nas fórmulas de WCRT adaptadas de Audsley *et al.*(1993), apresentadas no Capítulo 6.

Os testes foram realizados em um microcontrolador NUCLEO-F446RE da *ST-Microelectronics*. O microcontrolador, apresentado no Capítulo 5, possui um ARM Cortex-M4 de 32 *bits* e executa o *FreeRTOS* em sua versão 10.2.1, e possui suas funcionalidades apresentadas no Capítulo 4. Para os eventos do *microkernel*, foram utilizados *trace macros* oferecidas pelo Sistema Operacional. Na ocorrência de eventos do *microkernel*, como *Tasks* liberadas para disputa de monopólio de processamento, saídas de sinais digitais pré-configurados no ARM Cortex-M4 serão acionados, coletados pelo analisador lógico e aferidos por um osciloscópio digital de quatro canais. O uso dessa estratégia foi detalhada no Capítulo 5.

O WCET de cada uma das tarefas apresentadas é considerado equivalente ao seu HWM e os períodos foram pré-configurados no microprocessador para *Interrupts* e no *microkernel* para *Tasks*. São apresentadas nas próximas seções um panorama geral de *benchmark*.

7.1 TESTE 1 - *INTERRUPTS*

Neste teste são analisadas três tarefas do tipo *Interrupt*. Para execução deste teste na plataforma, foi escolhido um algoritmo de ordenação vetorial *Bubble Sort* (ASTRACHAN, 2003) que possui seu melhor e seu pior comportamento lógico conhecido.

Todos os *Interrupts* executam o mesmo algoritmo de ordenação sobre três vetores distintos, sendo a diferença apenas o tamanho entre eles. O *Interrupt* θ_1 executa o algoritmo de ordenação para um vetor de 50 posições, o *Interrupt* θ_2 executa o algoritmo de ordenação para um vetor de 75 posições, e o *Interrupt* θ_3 executa o algoritmo de ordenação para um vetor de 100 posições.

Seja um vetor numérico v com n elementos. Em uma análise de complexidade, o melhor caso de execução para o algoritmo *Bubble Sort*, quando adota a ordenação crescente, ocorre quando os elementos do vetor encontram-se ordenados. Nessa situação o algoritmo fará i iterações sobre v, sendo $i = n$, e não realizará qualquer troca na ordem dos elementos. Já o pior caso para execução do algoritmo ocorre quando o vetor está ordenado de maneira decrescente, situação na qual o algoritmo fará i iterações sobre v, sendo $i = n^2$.

Por estar executando em conjunto com *Interrupts*, o *FreeRTOS* também introduz no cenário a execução de *Tick* a cada 1 milissegundo, com um HWM conhecido de

1,5 microssegundos, quando o microprocessador opera a uma frequência de 84 *MHz*. Através de medições e aferição em osciloscópio, o WCET medido do algoritmo *Bubble Sort* para o *Interrupt* θ_1 tem um tempo total máximo de 1,33 milissegundos. O *Interrupt* θ_2 tem um máximo tempo medido de computação de 3 milissegundos. O *Interrupt* θ_3 possui um tempo máximo medido de 5,34 milissegundos.

São apresentadas na Tabela 10 as propriedades referentes aos *Interrupts* que serão testados no microprocessador.

Tabela 10 – Propriedades de *Interrupts* em teste

Interrupt	**Período**	**Prioridade**	**Preemptivo**	**C**
θ_1	5 ms	5 (Alta)	Sim	1,33 ms
θ_2	15 ms	6 (Média)	Sim	3,00 ms
θ_3	30 ms	7 (Baixa)	Sim	5,34 ms
tick	1 ms	16 (Muito baixa)	Não	1,5 μs

As próximas subseções apresentam uma discussão sobre o comportamento de cada *Interrupt* e seus instantes críticos no sistema.

7.1.1 *Interrupt* de alta prioridade

Como *Interrupt* de mais alta prioridade do sistema, θ_1 não sofre interferências de θ_2 e θ_3. Isso significa que os tempos de computação de ambas as tarefas não influenciarão θ_1 em seu instante crítico, a menos que θ_2 ou θ_3 implementem alguma seção crítica em que interrupções são desabilitadas e atrasem a liberação de θ_1. Entretanto, há a presença de *Tick* executando no sistema a cada 1 milissegundo e, como discutido anteriormente, *Tick* executa com interrupções desabilitadas, logo, o pior caso para θ_1 ocorre quando sua ativação e a ativação de *Tick* acontecem ao mesmo tempo. Para o cenário analisado, $J_{\theta_1} = I_{Tick}$.

Obter o verdadeiro WCET de θ_1 não é trivial. Questões como a influência temporal da arquitetura do microprocessador, e também outros fatores, não estão sendo rigorosamente levados em consideração para análise. Logo, utilizar o vetor em ordem decrescente apenas auxilia na aproximação do WCET de θ_1.

A latência de atendimento de interrupções da arquitetura é de 142 nanossegundos. Numa visão pessimista, porém factível, se leva em consideração o maior tempo encontrado (HWM) e acrescenta-se a esse valor uma "margem segura" . Uma margem adotada na prática pelos projetistas é adicionar ao HWM o valor de 20% do tempo de execução máximo observado (OLIVEIRA, R. S., 2018). Independente do uso de uma opção pessimista de análise, a Equação (6.3) contempla o pior tempo de computação

de θ_1. Então,

$$R_{\theta_1} = 1,5\mu s + 142ns + 1,330ms$$
$$R_{\theta_1} = 1,331642ms$$

Pela análise algébrica, o WCRT do *Interrupt* θ_2 é de 1,331642 milissegundos.

7.1.2 *Interrupt* de média prioridade

Como *Interrupt* de média prioridade no sistema, θ_2 não sofre interferências de θ_3, mas, pode sofrer interferências de θ_1. Isso significa que o tempo de computação de θ_1 influenciará θ_2 em seu instante crítico. Uma influência temporal de θ_3 seria visível se ambos, θ_1 e θ_3 implementassem alguma seção crítica compartilhada, o que não é o caso neste cenário.

Em seu instante crítico, θ_1 e θ_2 são ativados ao mesmo tempo. O *Interrupt* θ_1, por ser de mais alta prioridade, é liberado e passa a ser executado. No instante de ativação de ambos os *Interrupts*, *Tick* pode estar executando e impõe a θ_1 um *release jitter* que corresponde ao seu tempo de computação no pior caso. Para o cenário analisado, $J_{\theta_2} = I_{Tick}$. Antes de liberar θ_2 para execução, o microprocessador também impõe a esse *Interrupt* o tempo de computação de θ_1. Utilizando a Equação (6.4), é possível obter o WCRT de θ_2. Dado que $W^0_{\theta_2} = C_{\theta_2}$, então, é possível fazer a primeira iteração sobre a demanda de tempo de R_{θ_2}. Os dados de $W^1_{\theta_2}$ estão em microssegundos. Logo:

$$W^1_{\theta_2} = 0,142 + 3000 + \left\lceil \frac{3000 + 1,5}{15000} \right\rceil \times (0,142 + 1330)$$

Portanto, $W^1_{\theta_2} = 4,330284ms$. O número de interferências de θ_1 sobre θ_2 é um, dado o seu T que é de 15 milissegundos e seu tempo de computação é próximo a 1,3 milissegundos. Em uma segunda iteração, $W^2_{\theta_2} = 4,330284ms$. Se $W^{i-1}_{\theta_2} = W^i_{\theta_2}$, obtém-se uma demanda de tempo convergente, logo, o WCRT de θ_2 é de:

$$R_{\theta_2} = 1,5\mu s + 4,330284ms$$
$$R_{\theta_2} = 4,331784ms$$

Pela análise algébrica, o WCRT do *Interrupt* θ_2 é de 4,331784 milissegundos.

7.1.3 *Interrupt* de baixa prioridade

Como *Interrupt* de baixa prioridade no sistema, θ_3 sofre interferências de θ_1 e θ_2. No instante crítico, todos os *Interrupts* são ativados em conjunto. O *Interrupt* θ_1, por ser de mais alta prioridade, é liberado e passa a ser executado. Ainda no instante de ativação de todos os *Interrupts*, *Tick* pode estar executando e impõe a θ_1 um *release jitter* que corresponde ao seu tempo de computação no pior caso. Logo, $J_{\theta_1} = I_{Tick}$.

Após o término da execução do *Interrupt* θ_1, o *Interrupt* θ_3 sofre interferência do *Interrupt* θ_2, aumentando ainda mais o seu tempo de resposta. Em seu pior caso, quando o vetor está em ordem decrescente, o *Interrupt* leva um tempo (HWM) de 5,34 milissegundos para completar seu tempo de execução, não sofrendo interferências ou atrasos de liberação de *Tick*. Utilizando a Equação (6.5), é possível obter o WCRT do *Interrupt* θ_3.

Dado que $W^0_{\theta_3} = C_{\theta_3}$, então, é possível fazer a primeira iteração sobre a demanda de tempo de R_{θ_3}. Os dados de $W^1_{\theta_3}$ estão em microssegundos. Logo:

$$W^1_{\theta_3} = 0,142 + 5340 + \left\lceil \frac{5340 + 1,5}{5000} \right\rceil \times (0,142 + 1330) + \left\lceil \frac{5340 + 1,5}{15000} \right\rceil \times (0,142 + 3000)$$

Portanto, $W^1_{\theta_3} = 11,000568ms$. Nota-se que até esse instante, duas interferências de θ_1 e uma interferência de θ_2 ocorrem no *Interrupt* θ_3. Como novas interferências aumentam o tempo de resposta, na próxima iteração de demanda de tempo, haverá uma nova chegada e liberação do *Interrupt* θ_1. Para a segunda iteração, $W^2_{\theta_3} = 12,33071ms$. Para a terceira iteração sobre a demanda de tempo de R_{θ_2}, $W^2_{\theta_3} = 12,33071ms$. Como $W^{i-1}_{\theta_3} = W^i_{\theta_3}$ obtém-se uma demanda de tempo convergente, logo, o WCRT do *Interrupt* θ_3 é de:

$$R_{\theta_3} = 1,5\mu s + 12,33071ms$$
$$R_{\theta_3} = 12,33221ms$$

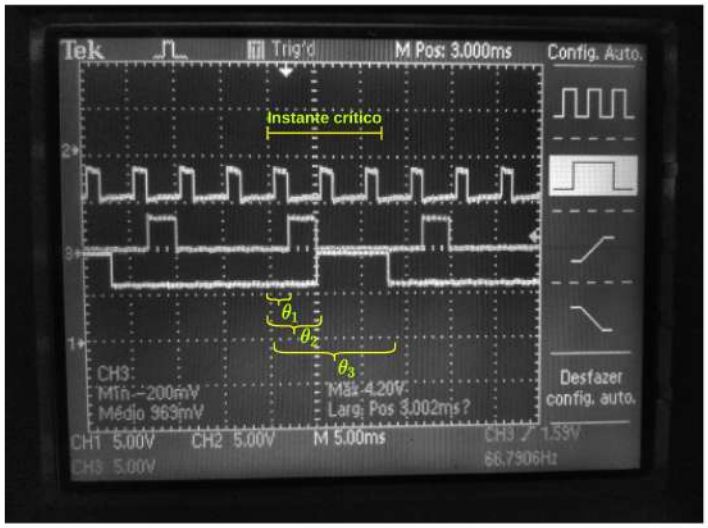
Pela análise algébrica, o WCRT do *Interrupt* θ_3 é de 12,33221 milissegundos.

7.1.4 Medições

Nesta seção são apresentadas as medições temporais realizadas no microprocessador executando um sistema composto por três *Interrupts* apresentados na Tabela 10 com o objetivo de estimar o WCRT da *Interrupt* θ_1, *Interrupt* θ_2 e *Interrupt* θ_3. Dado o valor de WCRT calculado nas subseções anteriores, é importante comparar os resultados teóricos obtidos com as amostras encontradas durante a execução do sistema. Para inserir valores nos vetores a serem ordenados foram utilizados os algoritmos da Listagem A.1 e Listagem A.2 presentes no Apêndice A. É utilizado o algoritmo de ordenação *Bubble Sort* na Listagem A.3, utilizado para os testes no SOTR.

É apresentado na Figura 34 o comportamento do sistema quando todos os *Interrupts* executam o algoritmo de ordenação *Bubble Sort*. Os vetores estão ordenados de forma decrescente. O osciloscópio apresenta o instante crítico do sistema. Sobre o *Interrupt* θ_1 não há interferências dos demais *Interrupts*. Já para o *Interrupt* θ_2, há uma interferência de θ_1. Finalmente, para θ_3, há três interferências de θ_1 e uma interferência de θ_2. A escala no osciloscópio é de 5 milissegundos por divisão.

Figura 34 – Instante crítico de *Interrupts*



Fonte – Autor

São apresentados na Tabela 11 os valores temporais medidos de execução do algoritmo de ordenação *Bubble Sort* no microprocessador ARM Cortex-M4 apresentados na Figura 34. São também apresentados os valores calculados para cada *Interrupt* para fins de comparação, assim como a variação percentual dos valores calculados em relação aos valores medidos.

Tabela 11 – Valores temporais calculados e medidos em *Interrupts*

Interrupt	**WCRT calculado**	**WCRT medido**	**Variação Percentual**
θ_1	1,331642 ms	$\approx$ 1,23 ms	+ 8,26 %
θ_2	4,331784 ms	$\approx$ 4,2 ms	+ 3,14%
θ_3	12,33221 ms	$\approx$ 12,2 ms	+ 1,08 %

Os valores são aproximados e foram coletados por um osciloscópio, que possui alta acurácia temporal. Os valores medidos se aproximam dos valores obtidos pelo uso da Equação (6.3), e da Equação (6.5).

7.2 TESTE 2 - *TASKS*

Neste teste são analisadas duas tarefas do tipo *Interrupt* e três tarefas do tipo *Task* executando no *FreeRTOS*. Para execução deste teste na plataforma, foi escolhido um algoritmo de ordenação vetorial *Insertion Sort* (KNUTH, 1998) que possui seu melhor e seu pior comportamento lógico conhecido.

Todas as *Tasks* executam o mesmo algoritmo de ordenação sobre três vetores distintos, sendo a diferença apenas o tamanho entre eles. A *Task* γ_1 executa o

algoritmo de ordenação para um vetor de 50 posições, a *Task* γ_2 executa o algoritmo de ordenação para um vetor de 100 posições e a *Task* γ_3 executa o algoritmo de ordenação para um vetor de 200 posições. O algoritmo de ordenação *Insertion Sort* para a *Task* γ_2 e *Task* γ_3 são protegidos por um *mutex* de forma a induzir situações de bloqueio na *Task* γ_2.

Seja um vetor numérico *v* com *n* elementos. Em uma análise de complexidade, o melhor caso de execução para o algoritmo *Insertion Sort*, quando adota a ordenação crescente, ocorre quando os elementos de *v* estão ordenados, situação na qual o algoritmo fará *i* iterações sobre *v*, sendo $i = n$. Já o pior caso de execução desse algoritmo ocorre quando o vetor está ordenado de maneira decrescente, situação na qual o algoritmo fará *i* iterações sobre *v*, sendo $i = n^2$. São apresentadas na Tabela 12 as propriedades referentes às *Interrupts* e *Tasks* do sistema. Na Tabela 13 são apresentados os tempos de chaveamento de contexto do sistema.

Tabela 12 – Propriedades de *Interrupts* e *Tasks* em teste

Interrupts				
Tarefa	**Período**	**Prioridade no microprocessador**	**Seção crítica**	***C***
θ_1	20 *ms*	5 (Alta)	Não	$38,25\mu s$
θ_2	15 *ms*	6 (Média)	Não	$76,38\mu s$
Tick	1 *ms*	16 (Baixa)	Não	$1,375\mu s$
Pseudo-tarefas				
Tarefa	**Período**	**Prioridade no microprocessador**	**Seção crítica**	***C***
γ^1_{tick}	20 *ms*	16 (Muito Baixa)	Não	$5,9\mu s$
γ^2_{tick}	22 *ms*	16 (Muito Baixa)	Não	$5,9\mu s$
γ^3_{tick}	21 *ms*	16 (Muito Baixa)	Não	$5,9\mu s$
γ^4_{tick}	150 *ms*	16 (Muito Baixa)	Não	$5,9\mu s$
Tasks				
Tarefa	**Período**	**Prioridade no SOTR**	**Seção crítica**	***C***
γ_1	20 *ms*	6 (Alta)	Não	$606,375\mu s$
γ_2	22 *ms*	5 (Média)	Sim	3,21ms
γ_3	21 *ms*	4 (Baixa)	Sim	9.22 ms
Timers				
Tarefa	**Período**	**Prioridade no SOTR**	**Seção crítica**	***C***
δ_1	150 *ms*	3 (Baixa)	Não	$120\mu s$
δ_2	250 *ms*	3 (Baixa)	Não	$242\mu s$
δ_3	300 *ms*	3 (Baixa)	Não	$397\mu s$
δ_4	300 *ms*	3 (Baixa)	Não	$478\mu s$

Nesta tabela, estão presentes além das *Tasks* e *Interrupts*, quatro *Timers* de

Tabela 13 – Chaveamentos de contexto de *Tasks* em teste

Chaveamentos de Contexto	***C***
$CS(IDLE, \gamma_1)$	$4,733\mu s$
$CS(IDLE, \gamma_2)$	$5,674\mu s$
$CS(IDLE, \gamma_3)$	$6,253\mu s$
$CS(IDLE, \gamma_{TS})$	$6,5\mu s$
$CS(\gamma_1, \gamma_2)$	$3,63\mu s$
$CS(\gamma_1, \gamma_3)$	$3,63\mu s$
$CS(\gamma_1, \gamma_{TS})$	$6\mu s$
$CS(\gamma_2, \gamma_1)$	$2,5\mu s$
$CS(\gamma_2, \gamma_3)$	$2,506\mu s$
$CS(\gamma_2, \gamma_{TS})$	$4,758\mu s$
$CS(\gamma_3, \gamma_1)$	$2,5\mu s$
$CS(\gamma_3, \gamma_2)$	$3,092\mu s$
$CS(\gamma_3, \gamma_{TS})$	$4,372\mu s$
$CS(\gamma_{TS}, \gamma_1)$	$5,15\mu s$
$CS(\gamma_{TS}, \gamma_2)$	$6\mu s$
$CS(\gamma_{TS}, \gamma_3)$	$6\mu s$
L	142ns
I_δ	$12,75\mu s$

mais baixa prioridade que serão analisadas na próxima seção. Por sua presença no sistema, *Tmr Svc* (representada por γ_{TS}) impõe um pequeno atraso em cada *Task* a partir de sua liberação em *Tick*.

Por executar em conjunto com *Interrupts* e *Tasks*, o *FreeRTOS* também introduz no cenário a execução de *Tick* a cada 1 milissegundo, operando a uma frequência de 84 *MHz*. Por quatro *Tasks* estarem presentes no sistema, há variação nos tempos de execução de *Tick* no sistema. Por conta dessa variação, *Tick* pode ser dividida em quatro pseudo-tarefas de mesmo período mínimo de cada *Task*, portanto, em 20, 21, 22 e 150 milissegundos.

Nas próximas subseções são apresentados os comportamentos de *Tasks* no sistema. Neste teste não será analisado o comportamento de *Interrupts*, pois estimativas de WCRT de tarefas desse tipo foram abordadas na Seção 7.1.

7.2.1 *Task* de alta prioridade

Por ser a *Task* de mais alta prioridade do sistema, γ_1 não sofre interferências de γ_2 e γ_3. Isso significa que ambas as *Tasks* não influenciarão γ_1 em seu instante crítico, a menos que γ_2 ou γ_3 implementem alguma seção crítica em que interrupções

sejam desabilitadas e atrasem a liberação de γ_1, o que não é o caso para este cenário. Influenciando o tempo de resposta de γ_1 há a presença de *Tick*, executando no sistema a cada 1 milissegundo. O pior caso de *Tick* é o momento de liberação de todas as três tarefas do sistema, caso no qual as quatro pseudo-tarefas resultantes do particionamento de *Tick* também são incluídos no WCRT de γ_1. Há também a influência temporal de θ_1 e θ_2, que no instante crítico da *Task* chegam no mesmo momento de chegada de γ_1, aumentando seu tempo de resposta.

Em seu instante crítico, θ_1, θ_2 e *Tick* são ativados ao mesmo tempo. Do ponto de vista de análise de γ_1 não importa a ordem de execução dos *Interrupts*, isto é, qual executa primeiro ou se esses *Interrupts* executam antes de γ_1 ou impõem interferência durante a execução de γ_1. O que é relevante na análise para a *Task* γ_1 é a quantidade de interferências e o tempo de computação dos *Interrupts* que é imposto ao pior comportamento temporal de γ_1. Utilizando a Equação (6.9) é possível obter o WCRT de γ_1. São apresentadas os propriedades relevantes do sistema para análise de WCRT de γ_1 na Tabela 12.

Dado que $W^0_{\gamma_1} = C_{\gamma_1}$, então, é possível fazer a primeira iteração sobre a demanda de tempo de W_{γ_1}. Os dados de $W^1_{\gamma_1}$ estão em microssegundos. Logo:

$$W^1_{\gamma_1} = 4 \times (5,9) + [(0,142 + 4,733) + 606,375] + \left\lceil \frac{606,375}{1000} \right\rceil \times (1,5) + \left\lceil \frac{606,375}{20000} \right\rceil \times (0,142 + 38,25) + \left\lceil \frac{606,375}{15000} \right\rceil \times (0,142 + 76,38)$$

Portanto, $W^1_{\gamma_1} = 751,264\mu s$. Nota-se que até esse instante, uma interferência de todos os *Interrupts* e pseudo-tarefas ocorrem em γ_1. Em uma segunda iteração, $W^2_{\gamma_1} = 751,264\mu s$. Como $W^{i-1}_{\gamma_1} = W^i_{\gamma_1}$, obtém-se uma demanda de tempo convergente, logo, o WCRT de γ_1 é de:

$$R_{\gamma_1} = 2,6\mu s + 751,264\mu s$$
$$R_{\gamma_1} = 753,864\mu s$$

Pela análise algébrica, o WCRT da *Task* γ_1 é de 753,864 microssegundos. O *release jitter* da *Task* γ_1 é de $2,6\mu s$, pois o tempo de computação da função *vTaskDelayUntil*, que suspende a execução de *Tasks* periódicas, executa com interrupções desabilitadas.

7.2.2 *Task* de média prioridade

Como *Task* de média prioridade no sistema, γ_2 sofre interferências de γ_1. Isso significa que o tempo de computação de γ_1 irá interferir em γ_2 em seu instante crítico a cada T_{γ_1}. Um bloqueio de γ_3 também é visível, pois, γ_3 implementa uma seção crítica compartilhada com γ_2.

Em seu instante crítico, θ_1, θ_2 e *Tick* são ativados em conjunto, afetando o tempo de computação da *Task*. Também, nesse instante, é possível que γ_3 tenha realizado uma operação de *lock* no *mutex* que impõe a *Task* γ_2 um bloqueio B_{γ_2}. Na liberação de γ_2, ao invés de quatro pseudo-tarefas, há a presença de três pseudo-tarefas resultantes do particionamento de *Tick* (γ^1_{Tick}, γ^2_{Tick} e γ^{TS}_{Tick}) que também influenciam em seu WCRT. Isso decorre de assumirmos que a *Task* γ_3 no instante crítico tenha bloqueado a *Task* em análise, afinal, para que aconteça o bloqueio na *Task*, a liberação de γ_3 precisa já ter ocorrido antes da liberação de γ_2. Utilizando a Equação (6.10) é possível obter o WCRT de γ_2. São apresentadas as propriedades relevantes do sistema para análise de WCRT de γ_2 na Tabela 12.

Dado que $W^0_{\gamma_2} = C_{\gamma_2}$, é possível fazer a primeira iteração sobre a demanda de tempo de R_{γ_2}. Os dados de $W^1_{\gamma_2}$ estão em microssegundos. Logo:

$$W^1_{\gamma_2} = 2 \times (5,9) + [(0,142 + 5,674) + 3210 + (2 \times (2,506) + 8308 + 3,092)] + \left\lceil \frac{3210}{1000} \right\rceil \times (1,5)$$
$$+ \left\lceil \frac{3210}{20000} \right\rceil \times (0,142 + 38,25) + \left\lceil \frac{3210}{15000} \right\rceil \times (0,142 + 76,38) + \left\lceil \frac{3210}{20000} \right\rceil \times (5,9 + [(0,142 + 3,63)$$
$$+606,375 + (0,142 + 2,5)])$$

Portanto, $W^1_{\gamma_2} = 12,283323ms$. Nota-se que até esse instante, uma interferência de todos os *Interrupts*, uma interrupção de *Tick* e pseudo-tarefas ocorrem em γ_2, assim como um bloqueio proveniente de γ_3. Para a segunda iteração, obteve-se $W^2_{\gamma_2} = 12,296823ms$ e para a terceira iteração sobre a demanda de tempo de R_{γ_2}, obteve-se $W^3_{\gamma_2} = 12,296823ms$. Como $W^{i-1}_{\gamma_2} = W^i_{\gamma_2}$, obtém-se uma demanda de tempo convergente, logo, o WCRT de γ_2 é de:

$$R_{\gamma_2} = 2,6\mu s + 12,296823ms$$
$$R_{\gamma_2} = 12,299423ms$$

Pela análise algébrica, o WCRT da *Task* γ_2 é de 12,299423 milissegundos. O *release jitter* da *Task* γ_2 é de $2,6\mu s$, pois o tempo de computação da função *vTaskDelayUntil*, que suspende a execução de *Tasks* periódicas, executa com interrupções desabilitadas.

7.2.3 *Task* de baixa prioridade

Sendo a *Task* de mais baixa prioridade no sistema, γ_3 sofre interferências de γ_1 e γ_2. Isso significa que o tempo de computação de γ_1 e o tempo de computação de γ_2 influenciarão γ_3 em seu instante crítico. A *Task* γ_3 implementa uma seção crítica compartilhada com a *Task* γ_2, mas não sofre bloqueio por ela, pois, a sua prioridade é mais baixa em relação a γ_2. Em seu instante crítico, θ_1, θ_2, γ_1, γ_2 e *Tick* são ativados em conjunto. As quatro pseudo-tarefas resultantes do particionamento de *Tick* também influenciam no WCRT de γ_3. Utilizando a Equação (6.9) é possível obter o WCRT de

γ_1. São apresentadas os propriedades relevantes do sistema para análise de WCRT de γ_1 na Tabela 12.

Dado que $W^0_{\gamma_3} = C_{\gamma_3}$, então, é possível fazer a primeira iteração sobre a demanda de tempo de R_{γ_3}. Os dados de $W^1_{\gamma_3}$ estão em microssegundos. Logo:

$$W^1_{\gamma_3} = 2 \times (5,9) + [(0,142 + 6,253) + 9220 + (2,506 + 3,092)] + \left\lceil \frac{9220}{1000} \right\rceil \times (1,5) + \left\lceil \frac{9220}{20000} \right\rceil \times (0,142 + 38,25) + \left\lceil \frac{9220}{15000} \right\rceil \times (0,142 + 76,38) + \left\lceil \frac{9220}{20000} \right\rceil \times (5,9 + [(0,142 + 2,5) + 606,375 + (0,142 + 3,63)]) + \left\lceil \frac{9220}{22000} \right\rceil \times (5,9 + [(0,142 + 3,091) + 3210 + (0,142 + 2,506)])$$

Portanto, $W^1_{\gamma_3} = 13,214178ms$. Nota-se que até esse instante, uma interferência de todos os *Interrupts*, catorze interferências de *Tick*, uma interferência de todas as *Tasks* e uma interferência das quatro pseudo-tarefas ocorrem em γ_3. Para a segunda iteração, obteve-se $W^2_{\gamma_3} = 13,220178ms$ e para a terceira iteração, $W^3_{\gamma_3} = 13,220178ms$. Como $W^{i-1}_{\gamma_3} = W^i_{\gamma_3}$, obteve-se uma demanda de tempo convergente, logo, o WCRT de γ_3 é de:

$$R_{\gamma_3} = 2,6\mu s + 13,220178ms$$
$$R_{\gamma_3} = 13,222778ms$$

Pela análise algébrica, o WCRT da *Task* γ_3 é de 13,222778 milissegundos. O *release jitter* da *Task* γ_3 é de $2,6\mu s$, pois o tempo de computação da função *vTaskDelayUntil*, que suspende a execução de *Tasks* periódicas, executa com interrupções desabilitadas.

7.2.4 Medições

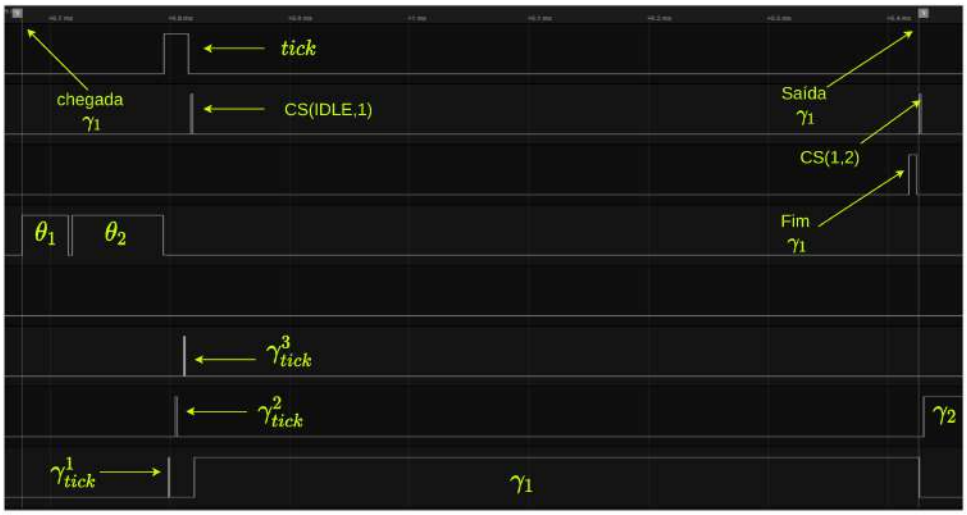
São apresentadas nesta seção as medições temporais realizadas no microprocessador, executando um sistema composto pelas duas *Interrupts* e três *Tasks* apresentadas na Tabela 12, com o objetivo de estimar o WCRT de *Task* γ_1, *Task* γ_2 e *Task* γ_3. Dado o valor de WCRT calculado nas subseções anteriores, é importante comparar os resultados teóricos obtidos com as amostras encontradas durante a execução do sistema. Para inserir valores nos vetores a serem ordenados foram utilizados os algoritmos da Listagem A.1 e Listagem A.2 presentes no Apêndice A. É utilizado o algoritmo de ordenação *Insertion Sort* na Listagem A.4, utilizado para os testes no SOTR.

Os valores medidos, assim como os valores calculados de WCRT para as *Task* γ_1, γ_2 e γ_3 são apresentados na Tabela 15, para fins de comparação.

É apresentado na Figura 35 o comportamento de *Task* γ_1 executando no seu instante crítico o algoritmo de ordenação *Insertion Sort*. O software do analisador lógico apresenta o instante crítico da *Task* γ_1. Nesse instante de maior tempo de resposta,

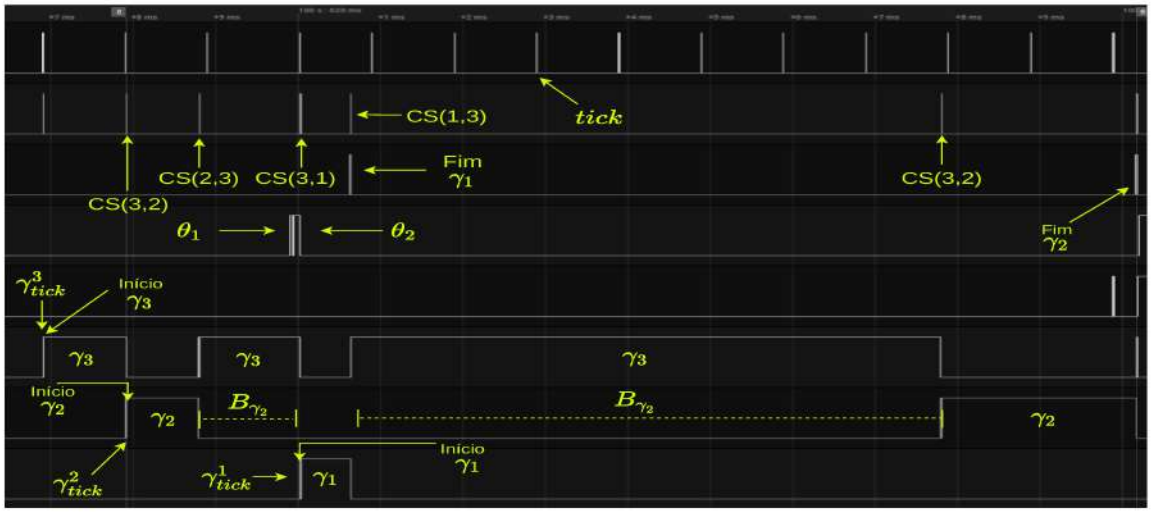
os *Interrupts* θ_1, θ_2, *Tick* e pseudo-tarefas do sistema chegam antes de γ_1 executar. Dois chaveamentos de contexto são realizados, $CS(IDLE, \gamma_1)$ e $CS(\gamma_1, \gamma_2)$. O tempo de resposta de γ_1 apresentado na figura é de aproximadamente 747 microssegundos.

Figura 35 – Instante crítico de γ_1



É apresentado na Figura 36 o comportamento de *Task* γ_2 executando no seu instante crítico o algoritmo de ordenação *Insertion Sort*. O software do analisador lógico apresenta o instante crítico da *Task* γ_2.

Figura 36 – Instante crítico de γ_2

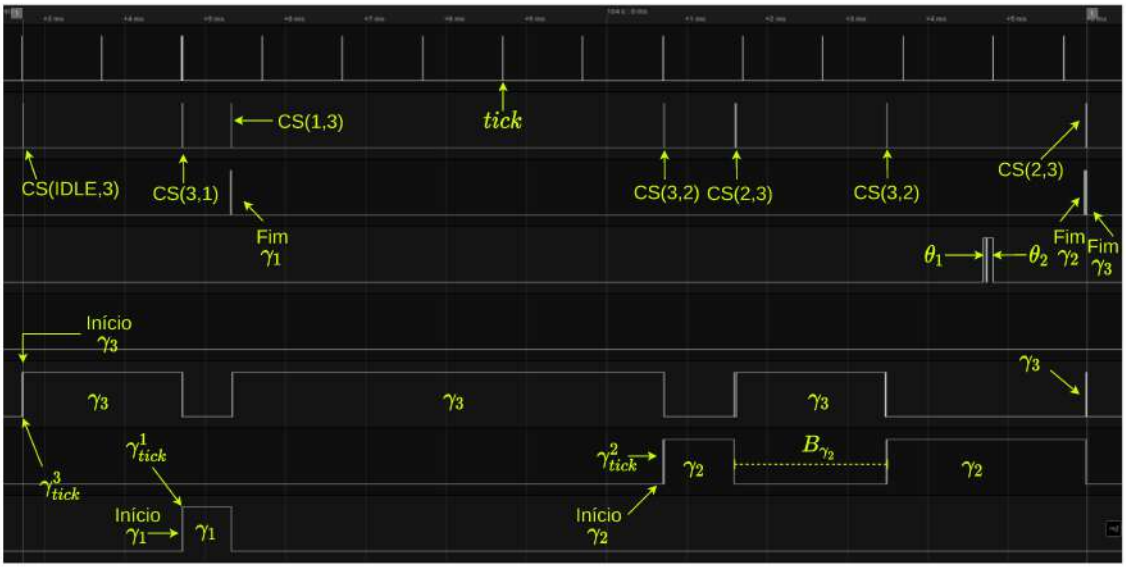


Nesse instante de maior tempo de resposta, os *Interrupts* θ_1, θ_2, *Task* γ_1 e pseudo-tarefas do sistema executam e aumentam o tempo de resposta de γ_2. Cinco chaveamentos de contexto são realizados, $CS(\gamma_3, \gamma_2)$, $CS(\gamma_2, \gamma_3)$, $CS(\gamma_3, \gamma_1)$, $CS(\gamma_1, \gamma_3)$ e $CS(\gamma_3, \gamma_2)$. A *Task* sofre um bloqueio B_{γ_2} e treze interferências de *Tick*.

O tempo de resposta de γ_2 apresentado na figura é de aproximadamente 12,1 milissegundos.

É apresentado na Figura 37 o comportamento da *Task* γ_3 executando no seu instante crítico o algoritmo de ordenação *Insertion Sort*. O software do analisador lógico apresenta o instante crítico da *Task* γ_3.

Figura 37 – Instante crítico de γ_3



Nesse instante medido de maior tempo de resposta, os *Interrupts* θ_1, θ_2, *Task* γ_1, *Task* γ_2 e pseudo-tarefas do sistema executam e aumentam o tempo de resposta de γ_3. Sete chaveamentos de contexto são realizados, $CS(IDLE, \gamma_3)$, $CS(\gamma_3, \gamma_1)$, $CS(\gamma_1, \gamma_3)$, $CS(\gamma_3, \gamma_2)$ e $CS(\gamma_2, \gamma_3)$,$CS(\gamma_3, \gamma_2)$, e $CS(\gamma_2, \gamma_3)$. A *Task* γ_3 impõe um bloqueio B_{γ_2} à *Task* γ_2 e catorze interferências de *Tick*. O *WCRT* de γ_3 apresentado na figura é de aproximadamente 13,1 milissegundos.

São apresentados na Tabela 14 os valores temporais calculados e medidos na plataforma experimental, assim como a variação percentual dos valores temporais calculados em relação aos valores temporais medidos.

Tabela 14 – Valores temporais calculados e medidos em *Tasks*

Tarefa	**WCRT calculado**	**WCRT medido**	**Variação Percentual**
Task γ_1	753,864 μs	$\approx$ 747 μs	+ 0,9189%
Task γ_2	12,299423 ms	$\approx$ 12,1 ms	+ 1,648%
Task γ_3	13,222778 ms	$\approx$ 13,1 ms	+ 0,937%

O software utilizado possui uma ferramenta que identifica o instante das bordas de subida e das bordas de descida de cada sinal eletrônico, o que auxilia na aproximação temporal dos dados coletados.

7.3 TESTE 3 - *TIMERS*

Neste teste foram analisadas duas tarefas do tipo *Interrupt*, três tarefas do tipo *Task* e quatro tarefas do tipo *Timer* executando no *FreeRTOS*. Essa modelagem de sistema é uma expansão do modelo analisado na seção anterior, com dados das *Tasks* e *Interrupts* apresentadas na Tabela 12. Os *Timers* usados no sistema podem ser periódicos ou esporádicos, dado que é possível esse tipo de comportamento em um sistema de tempo real.

Entre *Timers* não há prioridade, todavia, são programados em uma *Task* do sistema denominada *Tmr Svc*, como discutido no Capítulo 6. A *Task Tmr Svc* possui uma prioridade fixa padrão entre as *Tasks* que deve ser considerada para análise, já que os *Timers* herdarão a prioridade de *Tmr Svc*. São apresentados na Tabela 12 os tempos de computação de cada *Timer*.

Como os *Timers* são executados sobre *Tmr Svc*, há a presença de chaveamentos de contexto. São apresentados na Tabela 13 os tempos medidos de chaveamentos de contexto para *Tmr Svc*. Os chaveamentos de contexto $CS(\gamma_{TS}, \gamma_1)$, $CS(\gamma_{TS}, \gamma_2)$ e $CS(\gamma_{TS}, \gamma_3)$, devido à ausência de chaveamentos de contexto para tarefas de alta prioridade nas medições, são presumidos. Isso ocorre porque não foram detectados chaveamentos de contexto entre γ_{TS} com as tarefas γ_1, γ_2 e γ_3 durante os experimentos. De uma forma pessimista, portanto, o tempo de execução é o maior valor encontrado em um chaveamento de contexto para γ_{TS}.

Visto que *Timers* não possuem prioridade entre si, há um mesmo tempo de resposta compartilhado entre eles, variando apenas os tempos de computação de cada um na Equação 6.15. Portanto, apenas um cálculo para todos eles é necessário. Foi escolhido o *Timer* δ_1 para análise algébrica.

7.3.1 Tempo de Resposta

Com as informações presentes na Tabela 12, e utilizando a Equação (6.15), é possível obter o WCRT de δ_1. As interferências de cada *Task*, assim como *Tick*, pseudo-tarefas, *Interrupts* e os *Timers* δ_2, δ_3 e δ_4 influenciam no tempo de execução de δ_1. Dado que:

$$W^0_{\delta_1} = \sum_{j \in \Delta} (I_{\delta_j} + C_{\delta_j})$$

Então, é possível fazer a primeira iteração sobre a demanda de tempo de R_{δ_1}. Na equação, os dados de $W^1_{\delta_1}$ estão em microssegundos. Logo,

$$W^1_{\delta_1} = 5,9 + [(0,142 + 6,5) + 12,75 + 120] + [(12,75 + 242) + (12,75 + 397) + (12,75 + 478)]$$

$$+\left\lceil\frac{1228}{1000}\right\rceil\times(1,5)+\left\lceil\frac{1228}{20000}\right\rceil\times(0,142+38,25)+\left\lceil\frac{1228}{15000}\right\rceil\times(0,142+76,38)+\left\lceil\frac{1228}{20000}\right\rceil\times(5,9$$
$$+[(0,142+6)+606,375+(0,142+5,15)])+\left\lceil\frac{1228}{22000}\right\rceil\times(5,9+[(0,142+6)+3210+(0,142+4,758)])$$
$$+\left\lceil\frac{1228}{21000}\right\rceil\times(5,9+[(0,142+6)+9220+(0,142+4,372)])$$

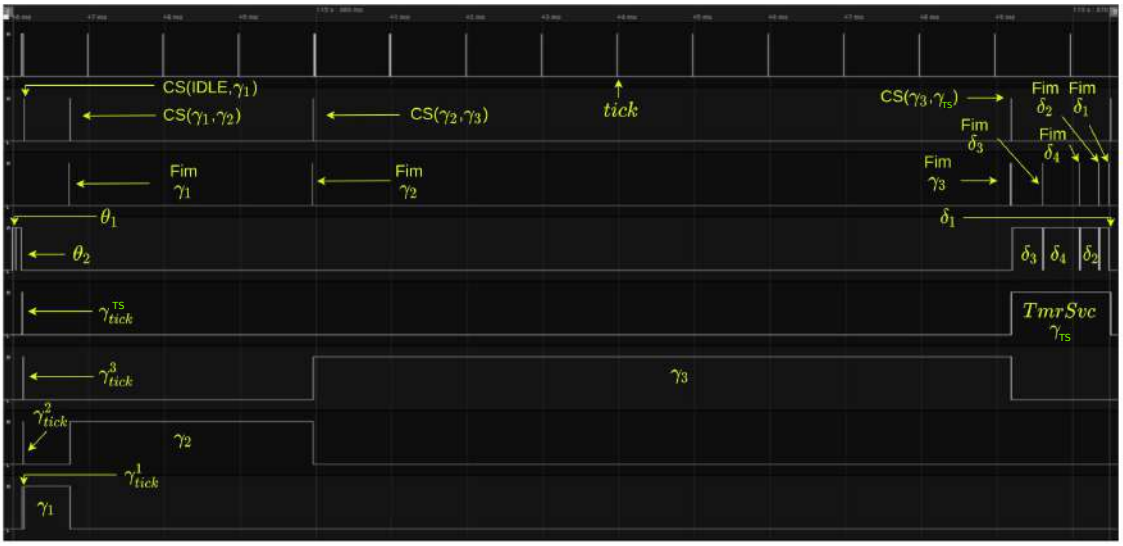
Portanto $W^1_{\delta_1} = 14,505663ms$. Nota-se que até esse instante, uma interferência de todos os *Interrupts*, duas interferências de *Tick*, uma interferência de todas as *Tasks* e uma interferência das quatro pseudo-tarefas ocorrem em δ_1. Para a segunda iteração, obteve-se $W^2_{\delta_1} = 14,525163m$. Nota-se que até esse instante, uma interferência de todos os *Interrupts*, quinze interferências de *Tick*, uma interferência de todas as *Tasks* e uma interferência das quatro pseudo-tarefas ocorrem em δ_1. Para a terceira iteração, obteve-se $W^3_{\delta_1} = 14,525163m$. Como $W^{i-1}_{\delta_1} = W^i_{\delta_1}$, obteve-se uma demanda de tempo convergente, logo, o WCRT de δ_1 é de:

$$R_{\delta_1} = 2,6\mu s + 14,525163ms$$
$$R_{\delta_1} = 14,527763ms$$

Pela análise algébrica, o WCRT do *Timer* δ_1, *Timer* δ_2, *Timer* δ_3 e *Timer* δ_4 é de 14,527763 milissegundos. O *release jitter* de liberação dos *Timers* é de $2,6\mu s$, tempo de computação da função *vTaskDelayUntil*. Essa função, que suspende a execução de *Tasks* periódicas, executa com interrupções desabilitadas.

7.3.2 Medições

Figura 38 – Instante crítico de *Timers* de um sistema



É apresentado na Figura 38 o instante crítico de cada *Timer* executando sobre a *Task Tmr Svc*. Nesse instante, antes de executar, *Tmr Svc* sofre interferência de todas as *Tasks*, *Interrupts*, e de *Tick*. A ordem de ativação dos *Timers* é pseudo-aleatória, dado que o WCRT, igual para todos. O pior tempo de resposta medido de *Tmr Svc* apresentada na figura é de aproximadamente 14,45 milissegundos. Os valores medidos, assim como os valores calculados de WCRT para os *Timers*, são apresentados na Tabela 15, para fins de comparação.

7.4 RESUMO

O objetivo deste capítulo foi aplicar os modelos algébricos de análise de *Tasks*, *Interrupts* e *Timers* apresentados no Capítulo 6 em três testes.

O primeiro teste foi composto de três tarefas periódicas do tipo *Interrupt*, com prioridades e periodicidades distintas. Cada *Interrupt* executou um algoritmo de ordenação *Bubble Sort* sobre um vetor. Foram apresentados seus piores tempos de computação, variação de tempos de resposta e, finalmente, um cálculo de WCRT utilizando a Equação (6.3) e a Equação (6.5), que foram apresentadas no Capítulo 6.

O segundo teste foi composto de três tarefas periódicas do tipo *Task* com prioridades e periodicidades distintas e duas tarefas periódicas do tipo *Interrupt* com prioridades distintas e periodicidades distintas. Cada *Task* executou um algoritmo de ordenação *Insertion Sort* em um vetor. Foram apresentadas seus piores tempos de computação, variação de tempos de resposta e finalmente um cálculo de WCRT utilizando a Equação (6.9) e a Equação (6.10).

O terceiro teste foi feito utilizando uma expansão do segundo teste em que foram adicionados ao sistema quatro *Timers* de períodos distintos executando sobre a tarefa *Tmr Svc*. Foram apresentados seus piores tempos de computação, variação de tempos de resposta e finalmente um cálculo de WCRT de cada *Timer* utilizando a Equação (6.15).

São apresentados na Tabela 15 os tempos de resposta medidos, tempos de resposta calculados e a variação percentual para os *Interrupts*, *Tasks* e *Timers* analisados no Capítulo 7. Como é evidente pelas informações apresentadas, os tempos de resposta calculados são maiores que os medidos, pois existe pessimismo na análise, já que a análise considera eventos cuja probabilidade de ocorrência pode ser baixa.

Nos experimentos realizados no Capítulo 7, os períodos das tarefas são de mesma ordem de grandeza e numericamente próximos. As aplicações utilizadas nos testes são relativamente simples. Com a inserção de *Tasks*, *Interrupts* e *Timers*, as variações percentuais entre os valores calculados e medidos podem aumentar, a partir do momento em que a complexidade do sistema aumenta.

Tabela 15 – Valores temporais calculados e medidos no sistema

Tarefa	**WCRT calculado**	**WCRT medido**	**Variação Percentual**
Interrupt θ_1	1,331642 ms	$\approx$ 1,23 ms	+ 8,26%
Interrupt θ_2	4,331784 ms	$\approx$ 4,2 ms	+ 3,14%
Interrupt θ_3	12,33221 ms	$\approx$ 12,2 ms	+ 1,08%
Task γ_1	753,864 μs	$\approx$ 747 μs	+ 0,92%
Task γ_2	12,299423 ms	$\approx$ 12,1 ms	+ 1,65%
Task γ_3	13,222778 ms	$\approx$ 13,1 ms	+ 0,94%
Timer δ_1	14,527763 ms	$\approx$ 14,45 ms	+ 0,53%
Timer δ_2	14,527763 ms	$\approx$ 14,45 ms	+ 0,53%
Timer δ_3	14,527763 ms	$\approx$ 14,45 ms	+ 0,53%
Timer δ_4	14,527763 ms	$\approx$ 14,45 ms	+ 0,53%

8 CONSIDERAÇÕES FINAIS

As seções seguintes apresentam considerações finais deste trabalho. Este capítulo está dividido na seção Conclusões, que apresenta conclusões a respeito do *microkernel* e da plataforma adotada, Contribuições, que apresenta os objetivos alcançados e Trabalhos Futuros, como sugestões de possíveis linhas de pesquisa.

8.1 CONCLUSÕES

O objetivo inicial deste trabalho foi alcançado. Com vistas a execução do *FreeRTOS* em um microprocessador ARM Cortex-M4, foi possível adaptar o comportamento algébrico descrito em Audsley *et al.*(1993) para execução de tarefas de tempo real suave. Os testes na plataforma, realizados no Capítulo 7, fornecem evidências de que os modelos matemáticos descritos no Capítulo 6 correspondem com o comportamento temporal das tarefas.

Outras considerações deste trabalho podem ser divididas em três tópicos, que englobam características abordadas nesta pesquisa:

- **Dificuldades**, no qual são relatadas as principais dificuldades para a elaboração desta pesquisa;
- ***FreeRTOS***, no qual vantagens e desvantagens do uso desse *microkernel* são abordadas;
- **Tempos de Resposta**, com conclusões a respeito de tempos de resposta de *Interrupts*, *Tasks* e *Timers*;
- **Hardware**, em que é abordado o uso da plataforma *Cortex-M*4 em cenários *soft real-time*.

8.1.1 Dificuldades

Como a elaboração desta pesquisa foi densa, uma das dificuldades encontradas foi a ausência de ferramentas na arquitetura utilizada que possibilitam a análise temporal de tarefas sem grandes interferências temporais no *microkernel*. Até este momento, existem softwares para aferição temporal, como o *TraceAlyzer* e o *Segger System View*, feitos exclusivamente para o uso no *FreeRTOS*. No entanto, além de *TraceAlyzer* ser uma ferramenta paga, há a questão temporal dos módulos de software construídos em ambos os sistemas, em que seu uso no *microkernel* para aferição temporal de *Interrupts*, *Tasks* e *Timers* impactariam os tempos de resposta das tarefas, devendo ser assim consideradas na análise. Caso seja necessário o uso de ambos os softwares, os modelos algébricos apresentados nesta pesquisa devem ser adaptados para o seu uso.

Também foram encontradas dificuldades na modelagem matemática do comportamento temporal das tarefas no *FreeRTOS*. Dado que há fontes distintas de variação temporal na arquitetura e no *microkernel*, para que o modelo não ficasse complexo ou pouco intuitivo, foi necessário o uso de premissas de comportamento de tarefas e generalizações, criando um sistema mais restrito em comparação as possibilidades que o *microkernel* oferece. Isso abre possibilidades de novos trabalhos que considerem cenários que não foram abordados por esta pesquisa.

8.1.2 *FreeRTOS*

O *FreeRTOS* possui um escalonador preemptivo, facilitando a programação de tarefas no *microkernel* e trazendo complexidade no cálculo de tempos de resposta.

Em questões de sincronização, o *FreeRTOS* implementa o PIP para evitar inversões descontroladas de prioridade entre tarefas. Como alternativas ao uso de *mutexes*, mensagens ou grupo de eventos entre tarefas, que têm um custo baixo de memória, podem ser utilizados. Por outro lado, a simplicidade de implementação do protocolo de sincronização coloca uma responsabilidade maior no projetista no sentido de evitar *deadlocks* na lógica de cada tarefa.

O *FreeRTOS* é uma opção de *microkernel* com baixo *overhead*, em que as únicas interferências que são impostas às tarefas periódicas são *Tick* e trocas de contexto. Em seus piores tempos de resposta, *Task*, *Interrupts* e *Timers* sofrem mais com interferências por conta de suas interações e bloqueios em virtude de seções críticas compartilhadas com tarefas de mais baixa prioridade do que por causa do *microkernel*.

8.1.3 Tempos de Resposta no *microkernel*

Interrupts são consideradas as tarefas mais prioritárias pelo sistema. Para a arquitetura, como discutido no Capítulo 3, é possível agrupar *Interrupts* com subprioridades, se assim necessário. O uso do *microkernel* pode inserir *release jitters* em tarefas desse tipo, no entanto, por não possuir habilidade de implementar seções críticas com *Tasks*, *Timers* e outros *Interrupts*, eles são o tipo de tarefa mais privilegiado de uso do microprocessador. O uso de *Interrupts* nesse contexto é indicado para as tarefas que possuem os requisitos temporais mais rígidos no sistema e com prioridade de atendimento, com os menores tempo de resposta do sistema.

Tasks são tarefas sob gestão do *FreeRTOS*. Os *overheads* de chaveamento de contexto, *Tick* e *Interrupts* interferem em sua execução. Nos modelos algébricos apresentados no Capítulo 6, não é possível que *Tasks* sofram interferências de *Timers*. Como alvo principal do escalonador, *Tasks* podem usufruir de todos os recursos disponibilizados pelo *microkernel*. O uso de *Tasks* é indicado para tarefas que necessitam de mecanismos de comunicação e sincronização. O que pode aumentar o tempo

de resposta de *Tasks* são as constantes interferências de *Interrupts* e sincronização com outras *Tasks*, logo, o cuidado com o comportamento de *Tasks* é necessário pelo projetista.

Timers são códigos presentes em módulos programados sobre uma *Task* nomeada pelo *microkernel* como *Tmr Svc*. Os *overheads* de chaveamento de contexto, *Tick* e *Interrupts* também interferem em sua execução, assim como *Tasks*. Nos modelos algébricos apresentados, *Timers* são o tipo de tarefa que mais sofre com interferências de outras tarefas do sistema. No pior caso, todos os *Timers* do sistema compartilham o mesmo WCRT, como apresentado no Capítulo 7. O uso de *Timers* é indicado para tarefas em que os requisitos temporais são brandos.

8.1.4 Hardware

Quanto ao hardware, por possuir poucas estruturas de aceleração, o microprocessador *Cortex-M*4 se mostrou durante os testes um bom microprocessador para análise temporal do *microkernel*. Por ser de baixo custo e rápido para algumas aplicações que trabalham na faixa dos microssegundos, em um projeto com esses requisitos é possível a sua utilização.

8.2 CONTRIBUIÇÕES

O objetivo deste trabalho foi contribuir com uma análise de Pior Tempo de Resposta de tarefas implementadas por um projetista no *FreeRTOS*, mapeando influências temporais de outras tarefas, da arquitetura e do *microkernel*. O modelo de WCRT construído por Audsley *et al.*(1993) foi adaptado para o contexto do SOTR analisado. Como premissa mínima de análise, é necessário que tarefas possuam um pior tempo de computação conhecido, uma periodicidade mínima e uma prioridade fixa. Considerando este objetivo, foram realizadas as seguintes contribuições:

- Um modelo de análise algébrica de WCRT para tarefas do tipo *Interrupt*, executando com interrupções habilitadas, que sofre influências temporais do *microkernel*;
- Um modelo de análise algébrica de WCRT para tarefas do tipo *Task*, que são gerenciadas pelo escalonador do *microkernel* e que pode sofrer interferências de outras *Tasks* e *Interrupts* e bloqueios por *Tasks* de mais baixa prioridade;
- Um modelo de análise algébrica de WCRT para tarefas do tipo *Timer*, que são gerenciadas pelo escalonador do *microkernel* e que pode sofrer interferências de outras *Tasks*, *Interrupts* e também de outros *Timers*;
- De forma a reduzir o pessimismo de estimativas de *Tick* do SOTR, que possui tempos de computação variáveis, foi proposto um modelo de partição de *Tick*, a

partir da criação de pseudo-tarefas que retiram sua variação temporal, fazendo assim com que em uma análise algébrica seja possível considerar *Tick* com um tempo de computação fixo;

- Um artigo completo, Miranda, Oliveira e Carminati (2021) foi aceito e publicado no XLI Congresso da Sociedade Brasileira de Computação (CSBC). O artigo detalha o comportamento de *Tick* e chaveamentos de contexto em tarefas periódicas, assim como as suas causas de variação temporal, que foram discutidas no Capítulo 6.

8.3 TRABALHOS FUTUROS

Dado que o SOTR *FreeRTOS* é de código aberto, o *microkernel* está sempre em evolução por contribuições da comunidade ou por necessidade de seus patrocinadores, seja adicionando novos módulos para uso do *microkernel*, otimizando a lógica de escalonamento ou diminuindo o *overhead* sobre tarefas programadas. Também por possuir uma portabilidade considerável entre diferentes microcontroladores e microprocessadores, são vastas as perspectivas de trabalhos que podem ser realizados utilizando a plataforma como objeto de estudo. São apresentadas a seguir propostas de futuros trabalhos:

- **Alteração lógica de escalonamento e avaliação de resultados**: Trabalhos como o de Oliveira e Lima (2020) buscam alterar códigos de escalonamento e avaliar quais são os resultados, como tempo de resposta e análise de WCET para tarefas inseridas no *FreeRTOS*. A depender da questão de estudo, em possíveis contribuições há a possibilidade de alterar o escalonador ou acoplar as primitivas do *microkernel* à lógica de escalonamento abordado e avaliar resultados, como estimativas do pior tempo de resposta de aplicações no SOTR.
- **Análise de WCRT no *FreeRTOS* de tarefas com igual prioridade**: Em Audsley *et al.*(1993) é considerado que tarefas de igual prioridade podem interferir umas nas outras em um escalonador totalmente preemptivo, no entanto, não é possível aplicar puramente essa característica de modelagem para o *FreeRTOS*, que adota o escalonamento *round-robin* para tarefas de igual prioridade, escalonamento que não é considerado no trabalho dos autores. Há trabalhos, como Racu *et al.*(2007), que abordam problemas de WCRT de tarefas sobre um escalonamento *round-robin* com preempções, então, há possibilidade de contribuição em que é possível fazer adaptações desses modelos algébricos para contexto de execução no *FreeRTOS*.
- **Análise de WCRT no *FreeRTOS* em multi-núcleos**: Em Audsley *et al.*(1993) o modelo algébrico de sistema é mono-processado, essa característica é também

adotada pelos modelos algébricos apresentados neste trabalho. Até o momento, excetuando seus simuladores, o *FreeRTOS* oficialmente possui suporte a microprocessadores *single-core*. Adaptações para processadores *dual-core* foram feitas por empresas patrocinadoras do projeto para seus microcontroladores. Também, trabalhos como o Mistry, Naylor e Woodcock (2014) e Chandrasekaran *et al.* (2014) buscam alterar o código-fonte do *microkernel* para execução de tarefas em cenários *dual-core* e *quad-core*, como o ARM Cortex-A9. Logo, há a possibilidade de contribuição de análise de tempos de resposta no *FreeRTOS* executando em processadores com mais de um núcleo de processamento.

- **Análise de WCRT de *Interrupts* com interrupções desabilitadas**: Em um sistema comercial padrão é comum que projetistas de sistemas embarcados utilizem tarefas que não podem em nenhuma hipótese sofrer interferências quando programados em *Interrupts*. Este trabalho abordou uma modelagem matemática que permite interferências entre *Interrupts*, padrão para a arquitetura utilizada. No entanto, o trabalho não levou em consideração cenários em que todos os *Interrupts* executem com interrupções desabilitadas ou a execução de um sistema híbrido, em que alguns *Interrupts* podem sofrer interferências e outros não. Isso leva a uma adaptação dos modelos apresentados para situações em que bloqueios podem ocorrer.

- **Análise de tempos de resposta em outras plataformas**: Este trabalho levou em consideração o sistema operacional *FreeRTOS* executando sob a plataforma ARM Cortex-M4. Como o *microkernel* em sua décima versão possui suporte para 40 tipos de microcontroladores, é possível a análise de tempos de resposta em plataformas distintas a analisada por este trabalho.

REFERÊNCIAS

American National Standards Institute and Institute of Electrical and Electronics Engineers. **IEEE-754**: *IEEE Standard for Binary Floating-Point Arithmetic*. *New York*: [*s.n.*], out. 1985. P. 20. ISBN 978-0-7381-1165-0.

Arm Holdings. ***Cortex-M4 Devices***: *Generic User Guide*. [*S.l.*], 2011. P. 277. Disponível em: `https://developer.arm.com/documentation`. Acesso em: 31 mai. 2021.

ASTRACHAN, O. *Bubble Sort: An Archaeological Algorithmic Analysis*. ***SIGCSE Bull.***, *Association for Computing Machinery*, New York, NY, USA, v. 35, n. 1, p. 1–5, jan. 2003. ISSN 0097-8418.

AUDSLEY, N.; BURNS, A.; RICHARDSON, M.; TINDELL, K.; WELLINGS, A.J. *Applying new scheduling theory to static priority pre-emptive scheduling*. ***Software Engineering Journal***, v. 8, n. 1, p. 284–292, set. 1993. ISSN 0268-6961.

AWS. **FreeRTOS: Sistema operacional em tempo real para microcontroladores**. [*S.l.*: *s.n.*], 2021. Disponível em: `https://aws.amazon.com/pt/freertos/`. Acesso em: 18 jun. 2021.

AWS. ***The FreeRTOS ™Reference Manual***: *API Functions and Configuration Options*. [*S.l.*: *s.n.*], 2017. P. 400. Disponível em: `https://www.freertos.org/Documentation/RTOS_book.html`. Acesso em: 21 abr. 2020.

BARRY, R. ***Mastering FreeRTOS Real Time Kernel***: A Hands-On Tutorial Guide. *Bristol*, 2016. P. 399. Disponível em: `https://www.freertos.org/Documentation/RTOS_book.html`. Acesso em: 15 mai. 2020.

BUTTAZZO, G.C. ***Hard Real-Time Computing Systems***: *Predictable Scheduling Algorithms and Applications*. 3. ed. Piza: *Springer*, 2011. ISBN 978-1-4614-0675-4.

CHANDRASEKARAN, P.; SHIBU KUMAR, K. B.; MINZ, R. L.; D'SOUZA, D.; MESHRAM, L. *A multi-core version of FreeRTOS verified for datarace and deadlock freedom*. *In*: *2014 TWELFTH ACM/IEEE CONFERENCE ON FORMAL METHODS AND MODELS FOR CODESIGN (MEMOCODE)*. [*S.l.*: *s.n.*], 2014. P. 62–71.

DIJKSTRA, E.W. *Information streams sharing a finite buffer*. ***Information Processing Letters***, *Elsevier*, v. 1, n. 5, p. 179–180, 1972. ISSN 0020-0190.

FARINES, J. M.; FRAGA, J.; OLIVEIRA, R. S. de. **Sistemas de Tempo Real**. 1. ed. Florianópolis: Universidade Federal de Santa Catarina, 2000. ISBN s.n.

JOSEPH, M.; PANDYA, P. *Finding Response Times in a Real-Time System*. ***The Computer Journal***, v. 29, n. 5, p. 390–395, jan. 1986. ISSN 1460-2067.

KLEIN, M. H.; RALYA, T. ***An Analysis of Input/Output Paradigms for Real-Time Systems***. *Pittsburgh*, *Carnegie Mellon University*, jul. 1990. ISBN s.n.

KNUTH, Donald E. ***The art of computer programming***. 3. ed. Boston, MA, USA: Addison-Wesley Longman Pub. Co., 1998. v. 3, p. 80–102. ISBN 0-201-89685-0.

LIU, C.L.; LAYLAND, James W. *Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment*. ***Journal of the Association for Computing Machinery***, v. 20, n. 1, p. 46–61, jan. 1973. ISSN 0004-5411.

MIRANDA, B.; OLIVEIRA, R.; CARMINATI, A. Analysis of FreeRTOS Overheads on Periodic Tasks. *In*: ANAIS do XX Workshop em Desempenho de Sistemas Computacionais e de Comunicação. Evento Online: SBC, 2021. P. 119–130.

MISTRY, J.; NAYLOR, M.; WOODCOCK, J. *Adapting FreeRTOS for Multicores: An Experience Report*. ***Softw. Pract. Exper.***, *John Wiley & Sons, Inc.*, USA, v. 44, n. 9, p. 1129–1154, set. 2014.

NICODEMOS, F. G.; SAOTOME, O.; LIMA, G.; SATO, S. S. *A minimally intrusive method for analysing the timing of RTEMS core characteristics*. ***International Journal of Embedded Systems***, v. 8, n. 5/6, p. 391–411, 2016.

OLIVEIRA, D. B.; OLIVEIRA, R. S. *Comparative Analysis of Trace Tools for Real-Time Linux*. ***IEEE Latin America Transactions***, v. 12, n. 6, p. 1134–1140, 2014.

OLIVEIRA, D. B.; OLIVEIRA, R. S. *Timing analysis of the PREEMPT RT Linux kernel*. ***Software: Practice and Experience***, v. 46, n. 6, p. 789–819, 2016.

OLIVEIRA, G.; LIMA, G. *Evaluation of Scheduling Algorithms for Embedded FreeRTOS-based Systems*. *In*: *2020 X BRAZILIAN SYMPOSIUM ON COMPUTING*

SYSTEMS ENGINEERING (SBESC). Florianópolis, Brasil: *IEE - Institute of Electrical and Electronic Engineers*, 2020. P. 1–8.

OLIVEIRA, R. S. **Fundamentos de Sistemas de Tempo Real**. 1. ed. Florianópolis: Amazon, 2018. ISBN 978-17-2869-404-7.

OLIVEIRA, R. S.; CARISSIMI, A. S.; TOSCANI, S. S. **Sistemas Operacionais**. 4. ed. Porto Alegre: *Bookman*, 2010. ISBN 978-85-7780-521-1.

PETERSON, G.L. *Myths about the Mutual Exclusion Problem*. ***Information Processing Letters***, *Elsevier*, v. 12, n. 3, p. 115–116, 1981. ISSN 0020-0190.

RACU, R.; LI, L.; HENIA, R.; HAMANN, A.; ERNST, R. *Improved Response Time Analysis of Tasks Scheduled under Preemptive Round-Robin*. *In*: *PROCEEDINGS OF THE 5TH IEEE/ACM INTERNATIONAL CONFERENCE ON HARDWARE/SOFTWARE CODESIGN AND SYSTEM SYNTHESIS*. Salzburg, Austria: *Association for Computing Machinery*, 2007. P. 179–184.

RAJKUMAR, R. ***Synchronization in Real-Time Systems***: *A Priority Inheritance Approach*. 1. ed. [*S.l.*]: *Springer*, 1991. ISBN 978-1-4615-4000-7.

RAJKUMAR, R.; SHA, L.; LEHOCZKY, J. P. *Real-time synchronization protocols for multiprocessors*. *In*: *PROCEEDINGS. REAL-TIME SYSTEMS SYMPOSIUM*. [*S.l.*: *s.n.*], 1988. P. 259–269.

SHA, L.; RAJKUMAR, R.; LEHOCZKY, J. P. *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*. ***IEEE Transactions on Computers***, v. 39, n. 9, p. 1175–1185, set. 1990. ISSN 1557-9956.

STMICROELECTRONICS. **DS10693**. 8. ed. Genebra, jul. 2020. Disponível em: `https://st.com/en/evaluation-tools/nucleo-f446re.html`. Acesso em: 16 fev. 2020.

STMICROELECTRONICS. **UM1724**. 14. ed. Genebra, ago. 2020. Disponível em: `https://st.com/en/evaluation-tools/nucleo-f446re.html`. Acesso em: 16 fev. 2020.

YIU, J. ***The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors***. 3. ed. *Massachusetts*: *Elsevier*, 2014. ISBN 978-0-12-408082-9.

APÊNDICE A – CÓDIGO PARA TESTES TEMPORAIS

A primeira função, apresentada na Listagem A.1, preenche um vetor com valores pseudo-aleatórios.

```
1  void random_array(int index, int vector[], int i) {
2    for (i = 0; i < index; i++) {
3      vector[i] = rand() % index;
4    }
5  }
```

Listagem A.1 – Função para inserir valores aleatórios em vetor

A segunda função, apresentada na Listagem A.2, preenche um vetor em ordem decrescente.

```
1  void worst_array(int index, int vector[], int i) {
2    for (i = index-1; i >= 0; i--) {
3      vector[i] = index - i;
4    }
5  }
```

Listagem A.2 – Função para inserir valores em ordem decrescente em vetor

A terceira função, apresentada na Listagem A.3, executa o algoritmo de ordenação *Bubble Sort*.

```
1  void bubble_sort_test(int n, int vector[], int k, int j, int aux) {
2    for (k = 1; k < n; k++) {
3      for (j = 0; j < n - k; j++) {
4        if (vector[j] > vector[j + 1]) {
5          aux = vector[j];
6          vector[j] = vector[j + 1];
7          vector[j + 1] = aux;
8        }
9      }
10   }
11 }
```

Listagem A.3 – Função para ordenar valores com Bubble Sort

A quarta função, apresentada na Listagem A.4, executa o algoritmo de ordenação *Insertion Sort*.

```
1  void insertion_sort_test(int arr[], int n, int i, int keyx, int j) {
2    for (i = 1; i < n; i++) {
3      keyx = arr[i];
4      j = i - 1;
5      while (j >= 0 && arr[j] > keyx) {
6        arr[j + 1] = arr[j];
7        j = j - 1;
8      }
9      arr[j + 1] = keyx;
10   }
11 }
12
```

Listagem A.4 – Função para ordenar valores com Insertion Sort

ANEXO A – *ALGORITMOS*

A.1 ALGORITMO DE BUSCA DE TEMPO MÁXIMO DE BLOQUEIO PIP

Data: Duração de cada $\delta_{i,k}$ para cada tarefa τ_i e cada semáforo S_k
Result: B_i para cada tarefa τ_i

```
// É assumido que as tarefas estão em ordem decrescente de prioridade.
begin
    // Para cada tarefa
    for i ← 1 to n − 1 do
        B_i^l ← 0
        // Para cada tarefa de menor prioridade
        for j ← i + 1 to n do
            D_max ← 0
            //Para cada semáforo
            for k ← 1 to m do
                if C(S) ≥ P_i and δ_{j,k} > D_max then
                    D_max ← δ_{i,k}
                end
            end
            B_i^l ← B_i^l + D_max − 1
        end
        B_i^s ← 0
        //Para cada semáforo
        for k ← 1 to m do
            D_max ← 0
            for j ← i + 1 to n do
                if C(S_k) ≥ P_i and δ_{j,k} > D_max then
                    D_max ← δ_{j,k}
                end
                B_i^s ← B_i^s + D_max − 1
            end
            B_i ← min(B_i^l, B_i^s)
        end
    end
    B_n ← 0
end
```

A.2 TIMER *TASK*

```
1  static portTASK_FUNCTION(prvTimerTask, pvParameters)
2  {
3      TickType_t xNextExpireTime;
4      BaseType_t xListWasEmpty;
5
6      /* Just to avoid compiler warnings. */
7      ( void ) pvParameters;
8
9      #if( configUSE_DAEMON_TASK_STARTUP_HOOK == 1 )
10     {
11         extern void vApplicationDaemonTaskStartupHook( void );
12
13         /* Allow the application writer to execute some code in the context of
14         this task at the point the task starts executing.  This is useful if the
15         application includes initialisation code that would benefit from
16         executing after the scheduler has been started. */
17         vApplicationDaemonTaskStartupHook();
18     }
19     #endif /* configUSE_DAEMON_TASK_STARTUP_HOOK */
20
21     for( ;; )
22     {
23         /* Query the timers list to see if it contains any timers, and if so,
24         obtain the time at which the next timer will expire. */
25         xNextExpireTime = prvGetNextExpireTime( &xListWasEmpty );
26
27         /* If a timer has expired, process it.  Otherwise, block this task
28         until either a timer does expire, or a command is received. */
29         prvProcessTimerOrBlockTask( xNextExpireTime, xListWasEmpty );
30
31         /* Empty the command queue. */
32         prvProcessReceivedCommands();
33     }
34 }
```

```
1  static TickType_t prvGetNextExpireTime( BaseType_t * const pxListWasEmpty )
2  {
3      TickType_t xNextExpireTime;
4
5      /* Timers are listed in expiry time order, with the head of the list
6      referencing the task that will expire first.  Obtain the time at which
7      the timer with the nearest expiry time will expire.  If there are no
8      active timers then just set the next expire time to 0.  That will cause
9      this task to unblock when the tick count overflows, at which point the
10     timer lists will be switched and the next expiry time can be re-assessed.  */
11     *pxListWasEmpty = listLIST_IS_EMPTY( pxCurrentTimerList );
12
13     if( *pxListWasEmpty == pdFALSE )
14     {
15         xNextExpireTime = listGET_ITEM_VALUE_OF_HEAD_ENTRY( pxCurrentTimerList );
16     } else {
17         /* Ensure the task unblocks when the tick count rolls over. */
18         xNextExpireTime = ( TickType_t ) 0U;
19     }
20
21     return xNextExpireTime;
22 }
```

```
1  static void prvProcessTimerOrBlockTask(const TickType_t xNextExpireTime,
2  BaseType_t xListWasEmpty)
3  {
4      TickType_t xTimeNow;
5      BaseType_t xTimerListsWereSwitched;
6      vTaskSuspendAll();
7      {
8          /* Obtain the time now to make an assessment as to whether the timer
9          has expired or not.  If obtaining the time causes the lists to switch
10         then don't process this timer as any timers that remained in the list
11         when the lists were switched will have been processed within the
12         prvSampleTimeNow() function. */
13
14         xTimeNow = prvSampleTimeNow( &xTimerListsWereSwitched );
15         if( xTimerListsWereSwitched == pdFALSE ){
16
17             /* The tick count has not overflowed, has the timer expired? */
18              if( ( xListWasEmpty == pdFALSE ) && ( xNextExpireTime <= xTimeNow ) ) {
19                  ( void ) xTaskResumeAll();
20                  prvProcessExpiredTimer(xNextExpireTime, xTimeNow);
21              } else {
22
23                 /* The tick count has not overflowed, and the next expire
24                 time has not been reached yet.  This task should therefore
25                 block to wait for the next expire time or a command to be
26                 received - whichever comes first.  The following line cannot
27                 be reached unless xNextExpireTime > xTimeNow, except in the
28                 case when the current timer list is empty. */
29
30                  if( xListWasEmpty != pdFALSE ) {
31                     /* The current timer list is empty - is the overflow list also empty? *
    /
32                      xListWasEmpty = listLIST_IS_EMPTY(pxOverflowTimerList);
33                  }
34
35                  vQueueWaitForMessageRestricted(xTimerQueue,
36                                  (xNextExpireTime - xTimeNow), xListWasEmpty);
37
38                  if( xTaskResumeAll() == pdFALSE ) {
39                     /* Yield to wait for either a command to arrive, or the
40                     block time to expire.  If a command arrived between the
41                     critical section being exited and this yield then the yield
42                     will not cause the task to block. */
43                       portYIELD_WITHIN_API();
44                  } else {
45                      mtCOVERAGE_TEST_MARKER();
46                  }
47             }
48         } else{
49             ( void ) xTaskResumeAll();
50         }
51     }
52 }
```