Pontifícia Universidade Católica do Rio Grande do Sul

Faculdade de Engenharia

Programa de Pós-Graduação Em Engenharia Elétrica

Desenvolvimento de um I-IP para o Monitoramento da Atividade do Sistema Operacional em Processadores Multinúcleos

Chrístofer Caetano de Oliveira

Orientador: Prof. Dr. Fabian Luis Vargas

Co-orientador: Prof. Dr. Ariel Lutenberg

Porto Alegre

2014

Pontifícia Universidade Católica do Rio Grande do Sul

Faculdade de Engenharia

Programa de Pós-Graduação Em Engenharia Elétrica

Desenvolvimento de um I-IP para o Monitoramento da Atividade do Sistema Operacional em Processadores Multinúcleos

Chrístofer Caetano de Oliveira

Orientador: Prof. Dr. Fabian Luis Vargas

Co-orientador: Prof. Dr. Ariel Lutenberg

Dissertação apresentada ao Programa de Mestrado em Engenharia Elétrica, da Faculdade de Engenharia da Pontifícia Universidade Católica do Rio Grande do Sul, como requisito parcial à obtenção do título de Mestre em Engenharia Elétrica.

Porto Alegre

2014

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

DESENVOLVIMENTO DE UM I-IP PARA O MONITORAMENTO DA ATIVIDADE DO SISTEMA OPERACIONAL EM PROCESSADORES MULTINÚCLEOS

CANDIDATO: CHRÍSTOFER CAETANO DE OLIVEIRA

Esta Dissertação de Mestrado foi julgada para obtenção do título de MESTRE EM ENGENHARIA ELÉTRICA e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Pontifícia Universidade Católica do Rio Grande do Sul.

DR. FABIAN LUIS VARGAS - ORIENTADOR

DR. ARIEL LUTENBERG - CO-ORIENTADOR

BANCA EXAMINADORA

DR. FRANK SILL TORRES - DEPARTAMENTO DE ENGENHARIA ELETRÔNICA - DELT - UFMG

DRA. LETÍCIA MARIA BOLZANI POEHLS - PPGEE - FENG - PUCRS

PUCRS
Campus Central
Av. Ipiranga, 6681 - Prédio 30 - Sala 103 - CEP: 90619-900
Telefone: (51) 3320.3540 - Fax: (51) 3320.3625
E-mail: engenharia.pg.eletrica@pucrs.br
www.pucrs.br/feng

Agradecimentos

Agradeço meus familiares por todo o apoio prestado até hoje.

Agradeço ao grupo SiSC pelo apoio e disponibilização da plataforma e equipamentos utilizados para o desenvolvimento deste trabalho.

Agradecimentos ao Professor Dr. Fabian Luis Vargas, meu orientador, e ao Professor Dr. Ariel Lutenberg, meu co-orientador, pela oportunidade dada para a realização deste trabalho.

Agradecimentos aos demais professores do grupo Professores Dra. Letícia B. Poehls, Juliano D. Benfica e Marcos A. Stemmer pela disponibilidade e colaboração ao longo destes anos.

Agradecimentos ao Sr. Federico G. Zacchigna, desenvolvedor do Plasma Multicore, utilizado como estudo de caso pelo suporte rápido e preciso fornecido.

Ao Programa CAPES/PROSUP pela cessão da bolsa a qual tornou possível a realização deste trabalho.

E aos demais que contribuíram de alguma forma para o desenvolvimento deste trabalho, o meu mais sincero MUITO OBRIGADO!

Resumo

O uso de sistemas operacionais de tempo real (Real-Time Operating Systems, RTOS), tornou-se uma solução atrativa para o projeto de sistemas embarcados críticos de tempo real. Ao mesmo tempo, observamos com entusiasmo o amplo uso de processadores *multicores* em uma lista interminável de nossas aplicações diárias. É também um acordo comum a crescente pressão do mercado para reduzir o consumo de energia em que estes sistemas portáteis embarcados necessitam para operar. A principal consequência é que estes sistemas estão se tornando cada vez mais suscetíveis à falhas transientes originadas por um amplo espectro de fontes de ruídos como Interferência Eletromagnética (Electromagnetic Interference, EMI) conduzida e irradiada e *radiação ionizante* (single-event transient: SET e total-ionizing dose: TID). Portanto, a confiabilidade destes sistemas é degradada. Nesta dissertação, discute-se o desenvolvimento e validação de um I-IP (Infrastructure-Intellectual Property) capaz de monitorar a atividade do RTOS em um processador *multicore*. O objetivo final é detectar falhas que corrompem o processo de escalonamento de tarefas em sistema sistemas embarcados baseados em RTOS preemptivos. Como exemplo destas falhas podem ser aquelas que impedem o processador de atender uma interrupção de alta prioridade, tarefas alocadas para serem executadas por um determinado núcleo, mas que são executadas por outro núcleo, ou até a execução de tarefas de baixa prioridade enquanto houver tarefas de alta prioridade na lista de tarefas prontas atualizada dinamicamente pelo RTOS. Este I-IP, chamado RTOS–Watchdog, foi descrito em VHDL e é conectado ao Barramento de Endereços da CPU em cada núcleo do processador. O RTOS–Watchdog possui uma interface parametrizável de modo a facilitar a adaptação a qualquer processador.

Um estudo de caso baseado em um processador multicore executando diferentes benchmarks sob o controle de um RTOS preemptivo típico foi desenvolvido. O estudo de caso foi prototipado em uma FPGA Xilinx Virtex4 montada em uma plataforma dedicada (placa mais software de controle) totalmente desenvolvida no Grupo *Computing Signals & Systems (SiSC)* [1] da *Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)*. Para a validação, todo o sistema foi exposto aos efeitos combinados de EMI e TID. Estes experimentos foram realizados em diversos passos, parte deles foram realizados na *PUCRS*, Brasil e parte no *Instituto Nacional de Tecnologia Industrial (INTI)* e *Centro Atómico*, ambos na cidade de Buenos Aires, Argentina. Os resultados

demonstram que a abordagem proposta fornece uma maior cobertura de falhas e latência de falhas reduzida quando comparados aos mecanismos de detecção de falhas nativos embarcados no *kernel* do RTOS.

Abstract

The use of Real-Time Operating System (RTOS) became an attractive solution to design safety-critical real-time embedded systems. At the same time, we enthusiastically observe the widespread use of multicore processors in an endless list of our daily applications. It is also a common agreement the increasing market pressure to reduce power consumption under which these embedded, portable systems have to operate. As the major consequence, these systems are becoming more and more sensitive to transient faults originated from a large spectrum of noisy sources such as conducted and radiated Electromagnetic Interference (EMI) and ionizing radiation (single-event effect: SEE and total-ionizing dose: TID). Therefore, the system's reliability degrades. In this work, we discuss the development and validation of an Infrastructure-Intellectual Property (I-IP) able to monitor the RTOS' activity in a multicore processor system-on-chip. The final goal is to detect faults that corrupt the task scheduling process in embedded systems based on preemptive RTOS. Examples of such faults could be those that prevent the processor from attending an interruption of higher priority, tasks that are strictly allocated to run on a given core, but are running on another one, or even the execution of low-priority tasks that are passed over high-priority ones in the ready-task list maintained on-the-fly by the RTOS. This I-IP, namely RTOS-Watchdog, was described in VHDL and is connected to each of the processor CPU-Addresses busses. The RTOS–Watchdog has a parameterizable interface to easily fit any processor bus.

A case-study based on a multicore processor running different test programs under the control of a typical preemptive RTOS was implemented. The case-study was prototyped in a Xilinx Virtex4 FPGA mounted on a dedicated platform (board plus control software) fully developed at the *Computing Signals & Systems' Group (SiSC)* [1] of the *Catholic University (PUCRS)*. For validation, the whole system was exposed to combined effects of EMI and TID. Such experiments were performed in several steps, part of them carried out at PUCRS, Brazil, and part at the *Instituto Nacional de Tecnología Industrial (INTI)* and *Centro Atómico*, both located in the city of Buenos Aires, Argentina. The obtained results demonstrate that the proposed approach provides higher fault coverage and reduced fault latency when compared to the native fault detection mechanisms embedded in the kernel of the RTOS.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

CAM	Content-Addressable Memory
CPU	Central Processing Unit
DDR SDRAM	Double-Data-Rate Synchronous Dynamic Random Access Memory
EMI	Electromagnetic Interference
FPGA	Field Programmable Gate Array
FSM	Finite State Machine
GPIO	General Purpose Input/Output
GPOS	General-Purpose Operating System
I\O	Input-Output
IEC	International Electrotechnical Commission
I-IP	Infrastructure-Intellectual Property
IP	Intellectual Property
MAC	Media Access Control
MEE	Monitor de Eventos de Escalonamento

PC	Personal Computer
RTOS	Real-Time Operating Systems
RTOS-WD	RTOS-Watchdog
SET	Single-Event Transient
SiSC	Signals & Systems for Computing
SMP	Symmetric Multi-Processing
SoC	System On Chip
SP	Stack-Pointer
SRAM	Static Random Access Memory
TCP\IP	Transmission Control Protocol \ Internet Protocol
TID	Total-Ionizing Dose
UART	Universal Asynchronous Receiver/Transmitter
UC	Unidade de Controle
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuits
WD	Watchdog

Sumário

PARTE I – INTRODUÇÃO

1 Introdução

Atualmente, sistemas em chip (Systems-on-Chip, SoCs) críticos suportam aplicações de tempo real que devem respeitar restrições temporais estritas. Em outras palavras, estes sistemas têm que fornecer não apenas resultados logicamente corretos, mas também resultados temporalmente precisos [2], uma falha nestes sistemas críticos pode resultar em danos catastróficos, desde prejuízos materiais, financeiros e até risco para a sociedade.

Neste contexto, a alta complexidade dos SoCs para aplicações críticas têm aumentado a necessidade de se empregar sistemas operacionais de tempo real (Real-Time Operating Systems, RTOSs) a fim de acelerar e minimizar a complexidade em seus projetos, de modo que o projetista do SoC se preocupe somente com o desenvolvimento da aplicação uma vez que o RTOS já possui as rotinas de baixo nível para o gerenciamento do hardware. Desta forma, SoCs baseados em RTOS podem explorar importantes recursos associados a mecanismos intrínsecos do sistema operacional tais como gerenciamento de tarefas, concorrência entre processos, coerência de memória e interrupções. Em mais detalhe, RTOSs atuam como uma interface entre software e hardware.

Com o crescente uso de sistemas wireless, como telefonia celular, equipamentos com tecnologia *Bluetooth* e redes Wi-Fi. O nível de ruído eletromagnético do meio está aumentando. Além disso, a preocupação com o consumo de energia faz com que os novos sistemas operem com margens de segurança cada vez menores, representando assim um desafio para a confiabilidade de sistemas embarcados de tempo real [3].

Neste cenário, sistemas eletrônicos baseados em RTOS devem operar em ambientes com alta exposição ao ruído eletromagnético. Em consequência disto, estes sistemas estão permanentemente expostos a falhas transientes. Estas falhas podem afetar não somente a execução da aplicação propriamente dita, mas também a execução do próprio RTOS. Afetando o RTOS, este tipo de falha pode se propagar para a aplicação e por consequência, levar a um comportamento incorreto do sistema.

Através de experimentos constatou-se que 34% das falhas transientes conduziram o RTOS a mau funcionamento no escalonador de tarefas e um adicional de 17% resultaram em *system crash*es [4]. Das falhas que se propagam para as tarefas da aplicação, cerca de 21% levam a falhas da aplicação [2]. Devido a essa alta taxa de falhas que se propagam para o RTOS e a aplicação resultando em erros é valida o esforço para o

aumento de robustez do sistema, detectando falhas que ocorram no escalonador do RTOS.

Ainda em muitos casos, tipicamente naqueles sistemas embarcados de alto desempenho baseados em processadores *multicore*, a maior complexidade do hardware a ser gerenciado pelo sistema operacional é outro fator agravante que expõe ainda mais as deficiências das técnicas existentes.

2 Motivação

Com o passar dos anos e devido à crescente complexidade das aplicações, e a demanda por dispositivos de baixa potência, o simples aumento da frequência dos processadores tornou-se inapropriado. Embora a evolução na tecnologia de fabricação tenha melhorado, chegou-se perto do limite físico dos semicondutores [5]. Assim se aumentou muito a complexidade dos projetos para processadores de alta frequência, a dissipação de calor tornou-se uma constante preocupação dos projetistas. Então a começou-se a explorar o paralelismo, o que permitiu um avanço em desempenho com uma frequência mais baixa de operação, não gerando aumento do consumo de energia, uma vez que estes dispositivos podem operar com uma tensão de alimentação mais baixa.

Os processadores *multicores* são bastante empregados hoje em dia em aplicações como: computadores pessoais, telefones celulares, consoles de jogos, televisores e *tablets*.

Após a realização de uma pesquisa bibliográfica em importantes bibliotecas digitais como o IEEE Xplore e ACM, de congressos e workshops, encontrou-se uma grande quantidade de técnicas que detectam falhas em tempo de execução de processadores multinúcleos. Estas técnicas são geralmente baseadas em software. As soluções para detecções de falhas propostas na literatura, para sistemas multicores, proporcionam tolerância a falhas somente para o nível de aplicativo e não consideram falhas que afetam o RTOS e então, se propagam para as tarefas da aplicação.

As técnicas existentes baseadas em software fazem uso instruções redundantes, controle de fluxo através de checkpoints ou até ambos. A redução o desempenho fica em torno de 40% por detecção de falhas [6]. Pois estas técnicas são baseadas na inserção de instruções, assim acarretam atrasos, este efeito pode ser diminuído através de *profiling*, ou seja, o algoritmo é protegido somente nas regiões mais críticas. Porém o atraso não pode ser eliminado.

Uma técnica que faz uso de redundância de instruções é a *Error Detection by Duplicated Instructions* (EDDI) [7], baseada em *software*, seu conceito é fundamentado na duplicação instruções e comparação de resultados entre a instrução original e a sua duplicata. Esta técnica deve ser aplicada em processadores superescalares, porém requer modificações no compilador, para alocar registradores para as instruções duplicadas e automatizar a aplicação da técnica em nível de instruções.

Outra técnica baseada em software propõe redundância em nível de processos *Process-Level Redundancy* (PLR) [6]. Esta técnica ocorre em nível de processo (ao contrario da EDDI que prevê redundância em nível de instruções) assim o sistema operacional fica a cargo de utilizar recursos ociosos do sistema para realizar a verificação de erros. Segundo o autor esta técnica gerou um *overhead* de desempenho de 16,9 % e este *overhead* é menor que as outras soluções, justamente por se aproveitar do tempo ocioso do processador.

Ainda, outra técnica encontrada na literatura a mSWAT [8], é uma técnica baseada em hardware. Esta técnica é capaz de diferenciar falhas transientes de falhas permanentes, através de reprocessamento e comparação. É baseada na técnica SWAT, acrônimo para SoftWare Anomaly Treatment. Para a reexecução é utilizado uma técnica de *checkpoint* e *logging*. As falhas são detectadas através da geração de exceções como: Divisão por Zero.

Cabe lembrar que todas estas técnicas têm um objetivo comum que é a detecção de falhas transientes ou permanentes durante a execução da aplicação propriamente dita. Até o momento não foi encontrada nenhuma técnica na literatura que tenha por finalidade a detecção de falhas durante a execução do sistema operacional, mais especificamente da execução do processo de escalonamento de tarefas para sistemas multinúcleos. As técnicas existentes demonstram resultados promissores, porém a um alto custo, pois geram perda de desempenho devido à redundância em nível de processos ou até mesmo instruções. Para sistemas críticos de tempo real a perda de desempenho deve ser evitada a todo custo, pois uma falha de *timing* pode comprometer o sistema ou gerar graves danos.

O Grupo SiSC [1] já propôs técnicas para detectar falhas no funcionamento do RTOS baseadas em hardware, na forma de I-IPs (Infrastructure-Intellectual Properties). Porém ambas as soluções são para sistemas *single-cores*.

A primeira delas proposta por J. Tarrilho [9] foi desenvolvida para sistemas operacionais com o algoritmo *Round-Robin* (ver em 5.4 Algoritmos de Escalonamento) e não tem suporte à interrupções. Para solucionar este problema foi proposto por Dhiego Sant'Anna da Silva [10] um I-IP capaz de manipular interrupções e com suporte para sistemas operacionais preemptivos (ver em 5.4 Algoritmos de Escalonamento).

Devido ao crescente emprego da tecnologia *multicore* nos sistemas modernos, a principal motivação para este trabalho é o desenvolvimento de uma técnica baseada

em hardware capaz de detectar mais falhas que a proteção nativa dos sistemas operacionais consegue detectar e com menor latência. Visando cobrir esta lacuna de pesquisa para sistemas multinúcleos esta dissertação propõe o desenvolvimento de um *watchdog* on-chip para monitorar a atividade do sistema operacional, mais especificamente o processo de escalonamento de tarefas.

3 Objetivos

Desenvolver um método genérico para o monitoramento de uma função muito importante realizada pelo sistema operacional: o escalonamento de tarefas. Aumentando assim sua robustez.

Para se atingir este objetivo será desenvolvido um *watchdog* on-chip (I-IP), o RTOS–Watchdog, que será instanciado e conectado ao barramento de endereços de CPU do processador. Este *watchdog* será descrito em VHDL e prototipado em lógica reconfigurável do tipo FPGA. Neste sentido utilizar-se-á o ambiente de desenvolvimento de sistema Xilinx® ISE Design Suite e o simulador ModelSim da Menthor Graphics®. Este mesmo ambiente de desenvolvimento será utilizado para validar e avaliar o RTOS–Watchdog. Mais especificamente será utilizada uma plataforma para ensaios de interferência eletromagnética conduzida segundo o standard IEC 61000-4-29 [11].

O RTOS–Watchdog deverá realizar o monitoramento passivo do processo de escalonamento de tarefas do sistema operacional. Entende-se por monitoramento "passivo" como sendo aquele que não gera nenhum tipo de interferência no funcionamento normal do processador, ou seja, não apresentará redução no desempenho do sistema. Esta técnica deverá ser aplicada a processadores multinúcleos. Como o estudo-de-caso será utilizado a versão *multicore* do Microprocessador Plasma [12] descrito em 6 Microprocessador Plasma.

Serão desenvolvidos benchmarks, para a validação e avaliação da proposta, estes benchmarks utilizarão os mais diversos recursos do sistema operacional, como semáforos, *mutexes* e filas de mensagens.

A solução a ser proposta deve gerar um hardware com a menor área possível frente à área do processador, com o objetivo de reduzir a probabilidade de falhas transientes atingirem o próprio RTOS–Watchdog.

PARTE II – FUNDAMENTOS

4 Robustez de Sistema

A definição original de robus tez diz que o sistema deve entregar um serviço de forma razoavelmente confiável. Uma definição alternativa que fornece o critério para decidir se o serviço é robusto diz: A robustez de um sistema é a capacidade de evitar falhas de serviço que são mais frequentes e mais graves do que o aceitável [13].

A robustez nada mais é que a qualidade de serviço fornecido por um sistema particular [14].

A robustez de sistema é um conceito que engloba os seguintes atributos [13] [14]:

- **Disponibilidade:** Prontidão para o serviço correto. É definida como a probabilidade que um sistema estar em operação e pronto para realizar suas funções num instante de tempo qualquer;
- **Confiabilidade:** Continuidade do serviço correto. É uma probabilidade condicional de o sistema operar corretamente num intervalo de tempo $[t_0, t]$ dado que o sistema opera corretamente no instante t_0;
- **Sistema seguro:** Ausência de consequências catastróficas para o usuário ou no ambiente. É a probabilidade de o sistema realizar suas funções corretamente ou descontinuá-las de maneira segura, sem afetar os seus usuários ou outros sistemas;
- **Integridade:** Ausência de alterações inapropriadas do sistema. Isto é o sistema deve estar sempre de acordo com as especificações originais ou com alterações certificadas;
- **Manutenibilidade:** Capacidade de ser submetido a modificações e reparos. Ou seja, é uma métrica que visa medir o quão fácil e a efetuação do reparo de um sistema uma vez que este apresente um defeito. Pode ser interpretada como a probabilidade de um sistema ser restaurado a um estado operacional num intervalo de tempo *t*;
- ***Performability*:** Existe a possibilidade de um sistema desempenhar corretamente as suas funções após a ocorrência de falhas de hardware ou software, entretanto, este sistema pode sofrer redução de desempenho. A *performability* pode ser definida como a probabilidade de um sistema ter um desempenho em um determinado nível ou acima num instante *t*.

4.1 Tipos de Falhas

Todas as falhas que podem afetar um sistema durante sua vida são classificadas conforme oito pontos de vistas [13], demonstrados na Figura 4.1.

Estas falhas podem ser agrupadas em três grandes grupos, parcialmente sobrepostos [13]:

- **Falhas de desenvolvimento**: Incluem todas as classes de falhas ocorridas durante o desenvolvimento;
- **Falhas Físicas**: Incluem todas as classes de falhas que afetam o hardware;
- **Falhas de interação**: Incluem todas as falhas externas.

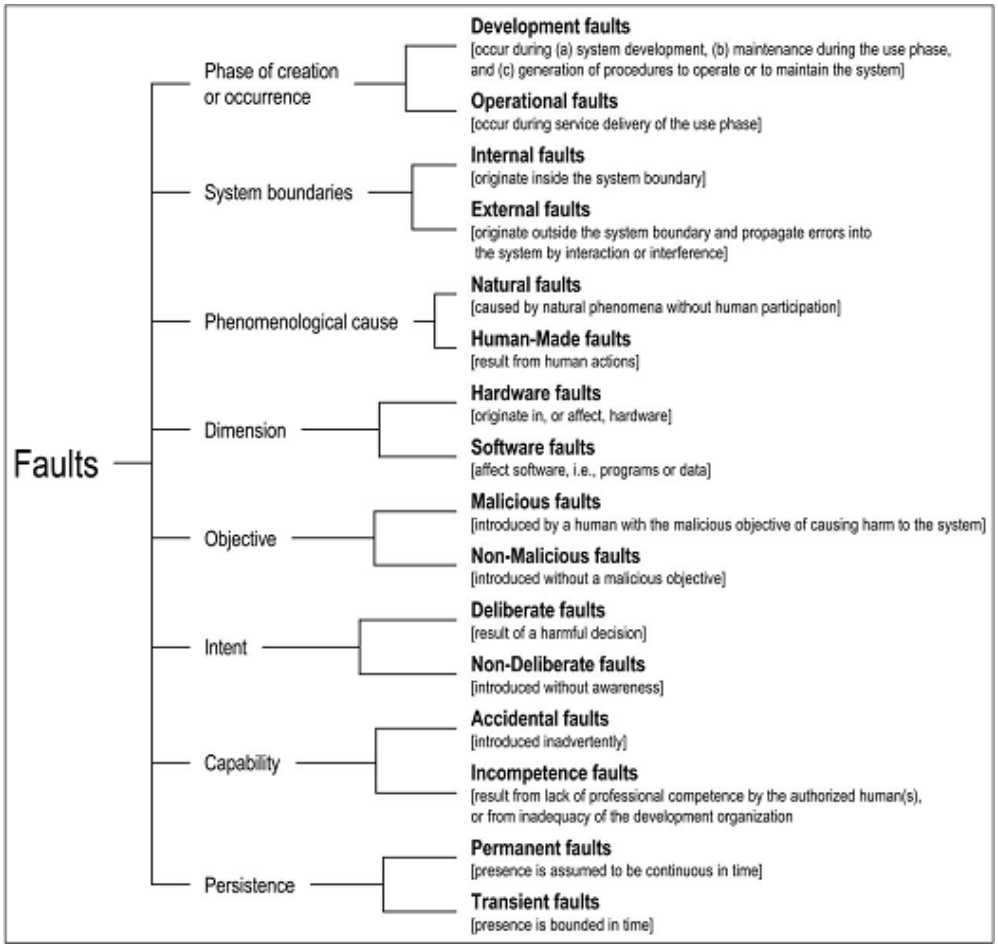


Figura 4.1: Classes elementares de falhas [13]

As falhas podem ter origem natural, ou serem produzidas por humanos. Falhas naturais são geradas por fenômenos naturais sem interferência humana. As falhas produzidas por humanos podem ser classificadas como: Maliciosas ou não maliciosas [13].

4.2 Falha, Erro e Defeito

Segundo Pradhan [14], existe relação entre falhas, erro e defeito, suas definições são apresentadas abaixo:

- **Falha:** Estas são causadas por fenômenos naturais de origem interna e externa e ações humanas acidentais ou intencionais. Existem tanto em nível de software e hardware. Podem ser geradas devido a interferências ou componentes envelhecidos [15]. A falha é considerada ativa quando produz um erro. Caso contrário a falha é considerada dormente ou silenciosa [13];
- **Erro:** É um estado no qual o sistema se encontra. Neste estado o processamento posterior conduz o sistema a um defeito [15]. O erro é propagável, ou seja, um erro é transformado em outro erro. É gerado por um processo computacional [13];
- **Defeito:** Ocorre quando há um desvio das especificações do projeto [15]. O defeito ocorre quando um erro é propagado para a interface de serviço gerando o comportamento fora das especificações de projeto do sistema [13].

Estes conceitos estão relacionados conforme o universo ao qual estão associados. As falhas estão no universo físico, enquanto erros se dão no universo da informação e os defeitos no universo do usuário [14]. Esta relação está demonstrada na Figura 4.2.

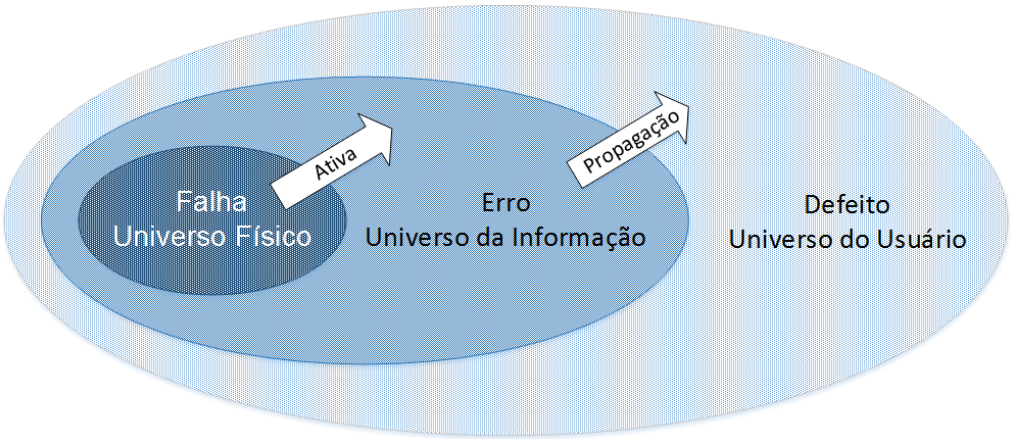


Figura 4.2: Relação entre Falha, Erro e Defeito

5 Real-Time Operating Systems

Um RTOS é um programa que escalona a execução de uma aplicação no tempo correto, gerencia recursos do sistema, e fornece um alicerce para o desenvolvimento do código da aplicação [16], diferentemente de um sistema operacional de proposito geral (General-Purpose Operating System, GPOS), um RTOS é caracterizado por terem tempo de conclusão de uma tarefa como um parâmetro fundamental. Por exemplo, em determinadas aplicações existe prazos rígidos para a conclusão (Hard Deadline), por exemplo, em linhas de montagem uma solda realizada muito cedo ou muito tarde pode resultar na perda do produto. Neste caso utiliza-se um sistema operacional de tempo crítico. Em outras aplicações onde o descumprimento do prazo é aceitável (Soft Deadline), por exemplo, reprodução de áudio, neste caso um atraso gera uma distorção no áudio reproduzido e seu efeito logo desaparece. Para estes casos são empregados os sistemas operacionais de tempo não críticos [17].

Todos os sistemas operacionais possuem um *kernel*. O *kernel* fornece os serviços básicos para as outras partes do sistema operacional. Tipicamente estes serviços incluem gerenciamento de memória, processos e arquivos. O conteúdo de um *kernel* varia muito entre os sistemas operacionais, mas estes incluem tipicamente: Escalonador e controlador de interrupções [18].

Hoje em dia o RTOS é a chave para muitos sistemas embarcados, pois fornece uma plataforma na qual o projetista pode se basear para construir aplicações.

5.1 Escalonador

O escalonador é um componente presente em qualquer *kernel* este é o responsável pela escolha da tarefa e a troca de contexto do processador. O escalonador é composto por objetos e serviços (Figura 5.1 adaptada do livro Real-Time Concepts [16]). Os objetos são ferramentas que ajudam o projetista a desenvolverem suas aplicações. Os serviços são operações que o *kernel* realiza nos objetos, dentre estas operações estão à temporização, gerenciamento de recursos e atendimento de interrupções [16]. Sua eficiência não depende somente do algoritmo empregado como também da natureza da aplicação, por exemplo, quanto ao quesito de *tempo real*, ou seja, um evento deve ser iniciado e concluído em um tempo predefinido.

Figura 5.1: Componentes comuns em um escalonador

5.2 Objetos

Os objetos ajudam o projetista a desenvolver a aplicação para sistemas operacionais de tempo real [16]. Os objetos mais comuns nos RTOS são as Tarefas, Semáforos e Filas de mensagens. Porém podem existir outros objetos como Timers e Pipes, a critério do projetista do sistema.

5.2.1 Tarefas

As tarefas são as subdivisões de uma aplicação, estas concorrem pelo tempo de execução do processador [16]. Cada tarefa possui sua própria pilha independente das outras tarefas e um bloco de controle. As tarefas são escalonáveis, para o controle do escalonador foram criados estados que são atribuídos as estas tarefas no decorrer da execução da aplicação. A Figura 5.2 adaptada do livro Real-Time Concepts [16] demonstra estes estados e as possíveis transições.

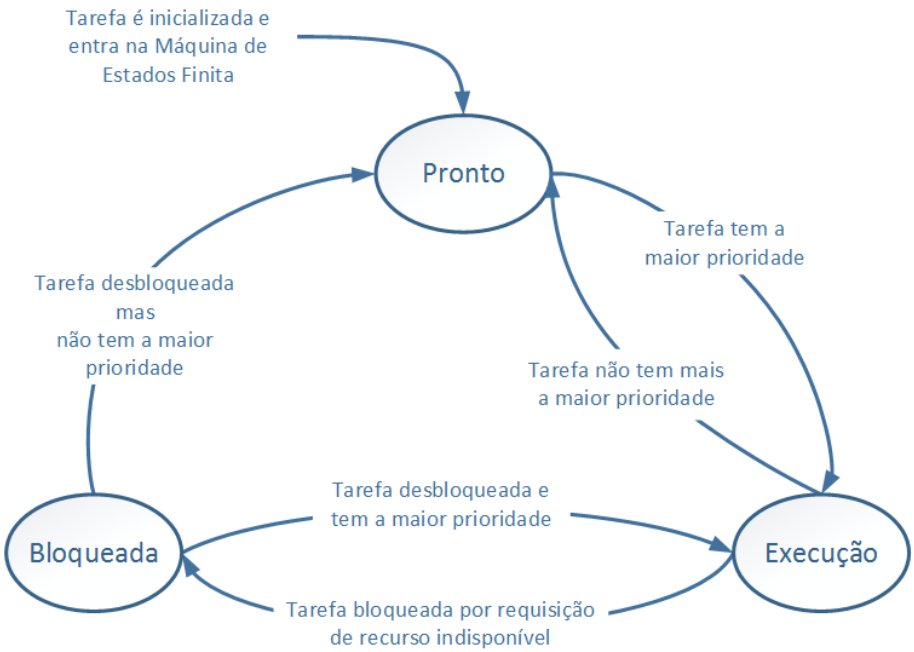


Figura 5.2: Estados típicos da execução de uma tarefa.

As tarefas são iniciadas no estado pronto (*ready*), neste estado à tarefa esta aguardando o escalonador executá-la. O estado bloqueado (*blocked*) é quando uma tarefa faz alguma requisição de algum recurso indisponível, ou esta aguardando algum semáforo, este estado serve para liberar o tempo de processamento para outras tarefas sejam escalonadas. Finalmente quando a tarefa possui a maior prioridade e não está bloqueada é atribuído a ela o estado 'Execução' (*running*) e esta passa a ocupar o tempo de processamento.

Estes são os estados típicos de uma tarefa, porém os projetistas de sistemas operacionais podem desenvolver outros estados mais complexos conforme conveniência ou necessidade. Alguns *kernels* comerciais, como VxWorks, definem outros estados mais granulares, ou seja, subdivisões desses estados [16].

É necessária a existência de uma tarefa para ser executadas enquanto não há tarefas prontas aguardando o tempo de execução. Esta tarefa é designada como *Idle Thread* ou *Idle Task*. Esta tarefa pode simplesmente manter o sistema em loop ou até mesmo pode induzir o CPU a um estado de baixo consumo de energia, caso a processador tenha suporte. Existem casos onde esta tarefa é utilizada para realizar algumas funções de baixa prioridade, ficando a critério do projetista. A única restrição é que esta tarefa não deve ser bloqueada.

5.2.2 Semáforos

O semáforo tem como função o controle de acesso a recursos compartilhados em um ambiente multitarefa. Sua invenção é atribuída a Edsger Dijkstra [19]. O semáforo é utilizado para sincronização e exclusão mútua [16].

O semáforo é basicamente um contador inteiro. O semáforo possui duas operações, de incremento (UP) e de decremento (DOWN), quando essa variável atinge o valor ‘0’ a tarefa que tentar adquiri-lo (DOWN) será bloqueada até que outra tarefa o libere (UP), caso contrário à execução da tarefa continua normalmente [20]. Atualmente são utilizadas outras nomenclaturas para operações, existem outras designações para estas operações como *Pend*, *Wait* e *Take* equivalente ao DOWN e *Post*, *Signal* ou *Release* para o UP.

Existem duas classificações uma delas “Counting Semaphores” esses semáforos são utilizados para recursos múltiplos. Uma analogia para esse tipo de semáforo seria o sistema de reserva de salas em uma biblioteca, conhecida como *Library Analogy*. Onde há um número limitado de salas que podem ser reservadas pelos estudantes. Neste cenário as salas seriam os recursos e o funcionário responsável pelas reservas seria o semáforo. No momento que uma sala for reservada (PEND) o contador é decrementado, se o valor resultante for maior que zero significa que há sala(s) disponível(eis), caso contrário todas as salas estão ocupadas então o usuário deve aguardar a liberação de alguma sala. Quando uma sala é liberada ocorre a operação (POST). A outra classificação “Binary Semaphore” bloqueia um recurso somente, seu valor é restrito a ‘0’ (locked, indisponível) e ‘1’ (unlocked, disponível) [21].

5.2.3 Mutexes

São similares aos Binary Semaphores (5.2.2 Semáforos), porém este é utilizado para fins de exclusão mútua, pois SOMENTE a tarefa que o adquiriu pode liberá-lo.

5.2.4 Filas de mensagem

As filas de mensagem, ou *message queues*, têm a função de intercomunicação entre as tarefas e mutua exclusão. É um *buffer* do tipo FIFO (First-In First-Out) com capacidade limitada. Ela retém a mensagem do emissor até que o receptor tenha condições de recebê-la. De modo que as tarefas não precisem enviar e receber mensagens simultaneamente, evitando assim o bloqueio do emissor.

5.3 Serviços

Os serviços auxiliam no desenvolvimento da aplicação, estes serviços compreendem um conjunto de funções, como uma biblioteca evitando-se que o projetista tenha que se preocupar com o desenvolvimento dessas sub-rotinas. Estes serviços geralmente são para a interface entre a aplicação e o *hardware*, como dispositivos de entrada e saída (Input / Output, I/O), ou para atuar nos objetos do *kernel*. Além de facilitar gerenciamento de tempo e atendimento de interrupções [16].

5.4 Algoritmos de Escalonamento

Os primeiros sistemas operacionais utilizavam aplicações que cediam voluntariamente o tempo de execução para outra. Essa abordagem foi eventualmente suportada por muitos sistemas operacionais é conhecida como *cooperative multitasking*. Hoje em dia este algoritmo é pouco utilizado, pois este algoritmo depende da regularidade de cada processo ao ceder o tempo de execução, um programa mal desenvolvido ocuparia todo o tempo de CPU, podendo fazer todo o sistema pendurar. Entretanto se corretamente utilizado permite uma alta previsibilidade [22].

Os algoritmos atuais relegam o controle do tempo de execução ao sistema operacional. Atualmente dois algoritmos de escalonamento são utilizados. Embora o desenvolvedor tenha liberdade para criar e definir seus próprios algoritmos de escalonamento conforme a aplicação seja de proposito geral ou tempo-real.

O algoritmo mais simples é o Round Robin (Figura 5.3). Este algoritmo cede iguais fatias de tempo para cada tarefa. Não há classificação das tarefas por prioridade.

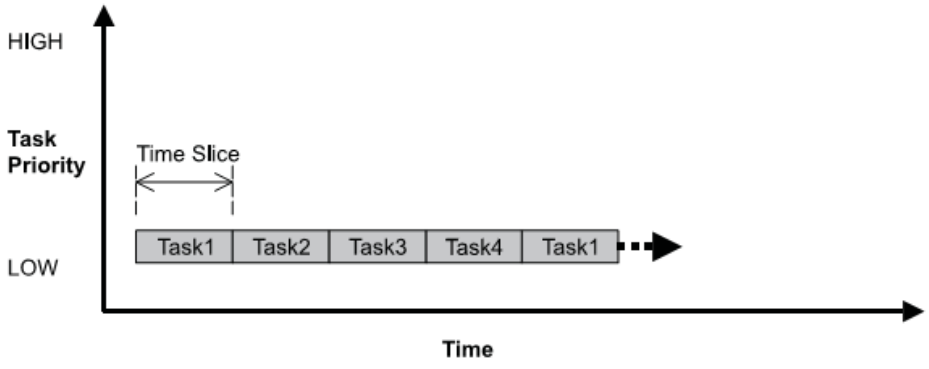


Figura 5.3: Algoritmo de escalonamento Round Robin

No entanto este algoritmo não é eficiente para sistemas de tempo real, pois não há compromisso com restrição temporal. Este algoritmo é mais utilizado em sistema operacional de propósito geral, como os utilizados em computadores pessoais (Personal

Computer, PC). Em GPOSs mais complexos, este algoritmo possuí fatias de tempo ajustadas dinamicamente para cada tarefa conforme uma série de critérios, por exemplo, players de mídia.

Em sistemas de tempo real as tarefas devem ser iniciadas e concluídas em tempos determinados. Por isso desenvolveu-se o algoritmo Preemptivo, neste algoritmo as tarefas são executadas conforme um sistema de prioridades e não mais por fatias de tempo.

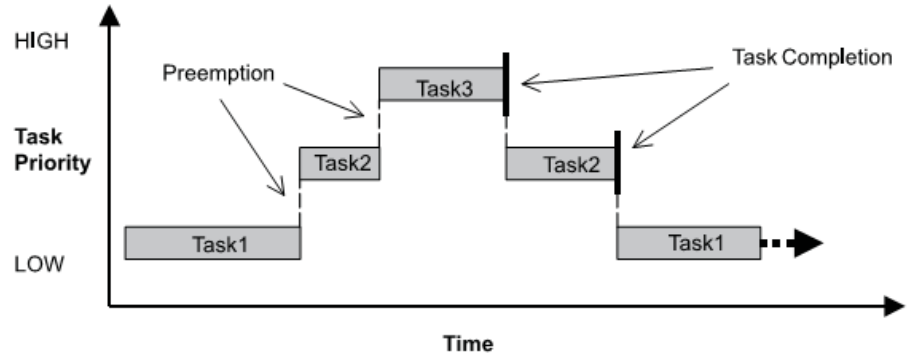


Figura 5.4: Algoritmo de escalonamento Preemptivo [16].

Este algoritmo é mais adequado para aplicações de tempo real. Porém só há troca de tarefas como consequência de um bloqueio pela requisição de recurso indisponível, devido a liberação de recurso aguardado por uma tarefa de alta prioridade, intencionalmente pelo projetista para forçar o escalonamento (*sleep*) ou por fim de tempo de espera por algum evento ou recurso (*timeout*). Surgiu então um algoritmo híbrido que une o melhor de Round Robin e o Preemptivo.

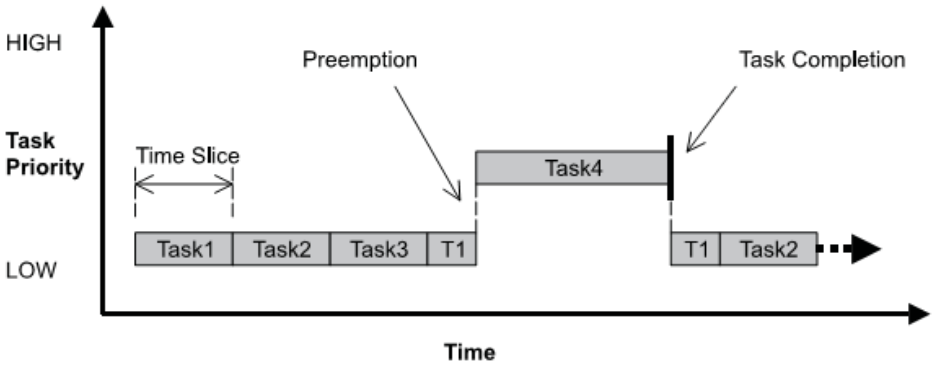


Figura 5.5: Algoritmo de escalonamento híbrido Premptivo-RR [16].

Este algoritmo é capaz de escalonar tarefas de igual prioridade por fatias de tempo e possui a capacidade de escalonar uma tarefa de alta prioridade na decorrência de algum evento de alta prioridade conforme os citados no caso anterior.

6 Microprocessador Plasma

Baseado no microprocessador Plasma [23] um *softcore* compatível com a arquitetura MIPS I ™ de 32 bits. O Plasma é capaz de executar quase todas as instruções de modo de usuário desta arquitetura com exceção das *load* e *store* desalinhadas, pois estas estavam patenteadas durante seu desenvolvimento.

Desenvolvido por Steve Rhoads em 2001 descrito em VHDL, e disponibilizado como *open source* no site *Opencores* [24]. O Plasma vem acompanhado por um RTOS próprio, desenvolvido em Linguagem C, chamado PlasmaRTOS, também desenvolvido por Rhoads e disponibilizado igualmente como *open source*.

O Plasma é modular, a CPU (Central Processing Unity) pode ser sintetizada com um pipeline de dois ou três estágios com um estágio opcional para leitura ou escrita de memória. Há opção de ser sintetizado como *Big* ou *Little Endian*.

E seus periféricos podem ser incluídos ou descartados na síntese. O Plasma possui controlador Ethernet MAC, UART (Universal Asynchronous Receiver / Transmitter), interface para memórias Flash, SRAM (Static Random Access Memory) e DDR SDRAM (Double-Data-Rate Synchronous Dynamic Random Access Memory), controlador de interrupção. Além dos periféricos o Plasma possui multiplicador e divisor em hardware opcional.

Na Figura 6.1, encontra-se o diagrama do Microprocessador Plasma, seu funcionamento está explicado na página de seu projeto em *opencores.org* [25].

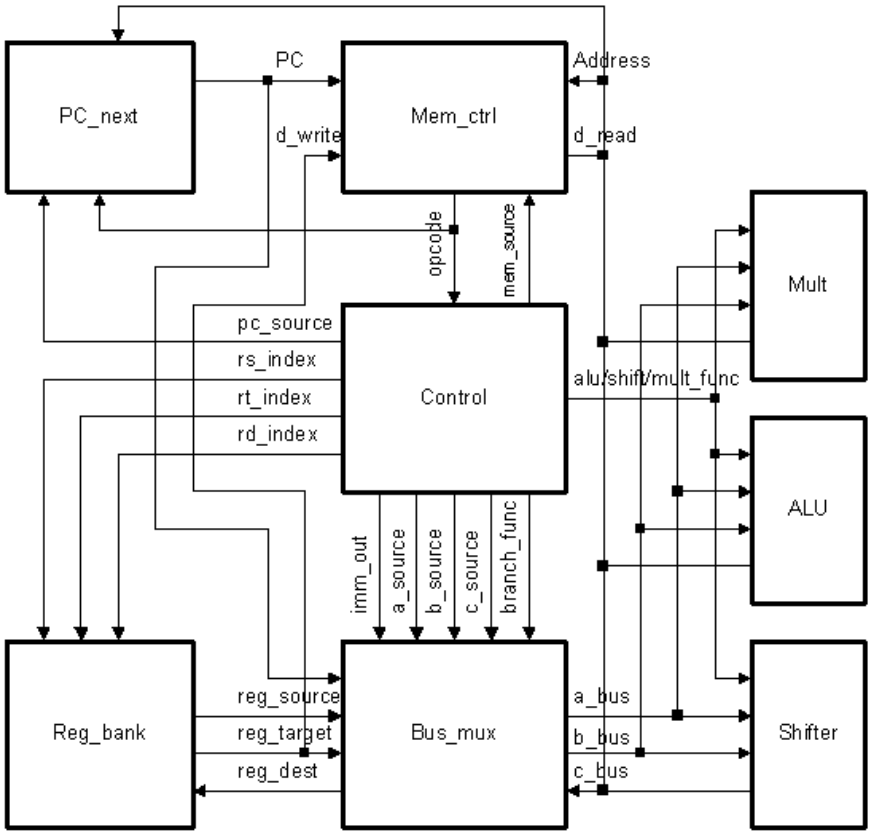


Figura 6.1: Diagrama de blocos processador Plasma [23]

O Plasma é bastante recorrente em pesquisas acadêmicas por possuir código aberto e livre além de ser relativamente simples o seu entendimento. É compatível com diversas FPGAs (Field Programmable Gate Array). Segundo o autor, seu funcionamento foi verificado em FPGAs da Altera® e Xilinx®.

Com o objetivo de pesquisa sobre sistemas *multicores* foi desenvolvida uma versão *multicore* do Plasma [12], por Federico Giordano Zacchigna. O número de núcleos do Plasma é parametrizado, podendo ser determinado no momento da síntese. Porém há uma restrição, o número de núcleos deve ser potência de dois (2^n).

O Plasma Multicore foi desenvolvido para trabalhar a 25 MHz, possui *cache* de 512 kB, com política de escrita *Write-Through Write-Allocate*, mapeamento direto, e protocolo de *snooping* para coerência de *cache*. Memória RAM de 16 MB para instruções e dados. Memória interna de 1kB para cada núcleo e conexão UART; 1 porta GPIO (General Purpose Input / Output) de 32b de saída, 1 porta GPIO 32b de entrada [12].

A Figura 6.2 demonstra a estrutura hierárquica do Plasma Multicore, com os blocos que a compõe.

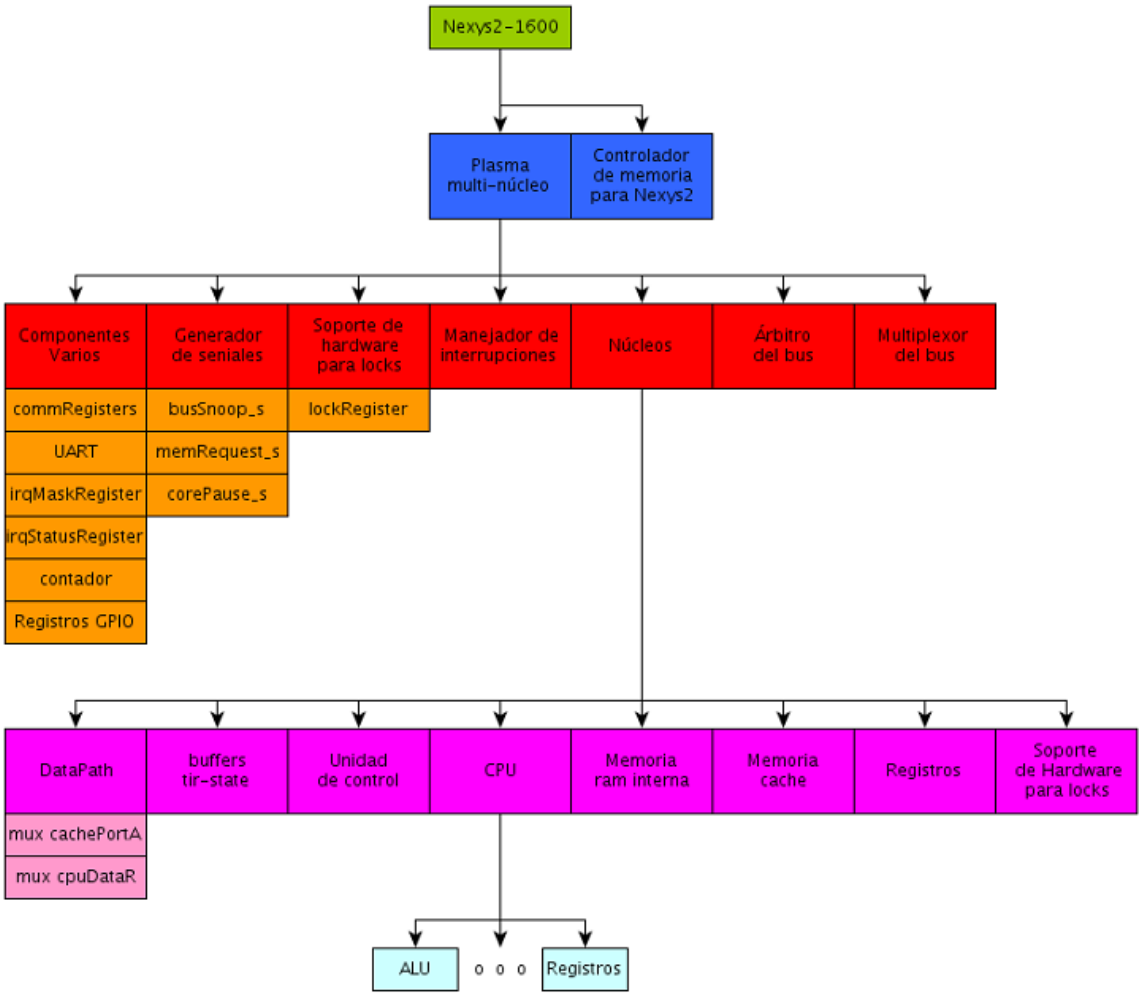


Figura 6.2: Estrutura Hierárquica do Plasma Multicore [12].

7 Sistema Operacional de Tempo Real – PlasmaRTOS

O sistema operacional PlasmaRTOS desenvolvido por Steve Rhoads já possuí suporte para Multiprocessamento Simétrico (Symmetric Multi-Processing, SMP), apesar do microprocessador Plasma ser originalmente *single core*. O sistema operacional do Plasma o PlasmaRTOS, é preemptivo tem suporte a *threads*, semáforos, *mutexes*, *timers*, *heaps* e filas de mensagens. Vem com uma biblioteca ANSI C com ponto flutuante de precisão simples, pilha TCP/IP e Servidor WEB.

Para o Plasma Multicore foram realizadas modificações no PlasmaRTOS por Federico G. Zacchigna, principalmente no suporte ao *lock* por hardware.

8 Interferência Eletromagnética

Interferência Eletromagnética (ou EMI) é um distúrbio que afeta um circuito elétrico devido à condução eletromagnética ou radiação eletromagnética emitida por uma fonte externa. EMI conduzidas ou irradiadas são diferenciadas pelo modo com seus campos eletromagnéticos são propagados. EMI conduzida é causada por contato físico de condutores, EMI radiada é gerada por indução, sem contato físico. Seu efeito pode variar de uma simples degradação de um dado até sua perda total.

O comportamento da EMI é dependente da frequência de operação, e não podem ser controladas em altas frequências. Para baixas frequências a EMI é gerada por condução, para altas frequências por radiação [26].

8.1.1 EMI Irradiada

Este tipo de EMI é transmitida através na forma de ondas eletromagnéticas. A fonte de EMI irradiada pode ser tanto artificial (emissões espúrias de rádio, ou por equipamentos de uso militar com este proposito) como natural (Auroras Boreais) [27] [26].

8.1.2 EMI Conduzida

Este tipo de EMI é transmitida por condutores, tanto de alimentação quanto de Entrada e Saída de equipamentos.

A EMI conduzida pode ser gerada por fontes chaveadas estas são geradores naturais de espectros de banda estreita com componentes em sua frequência fundamental e harmônicas de ordens mais altas [28]. Motores, relés, e lâmpadas fluorescentes também são geradores EMI conduzida bastante comuns ambientes industriais e domésticos. Este tipo de interferência pode ser combatido geralmente com a adição de filtros no circuito.

A EMI conduzida causa um distúrbio no campo eletromagnético ao redor do condutor evitando que este permaneça igualmente distribuído e causa o *skin effect*, perdas por histerese, transientes, quedas de tensão, distúrbios eletromagnéticos dentre outros efeitos [26].

9 Norma IEC 61.000-4-29

Tanto a operação correta quanto o desempenho, de equipamentos e/ou dispositivos elétricos e eletrônicos, podem sofrer degradação quando estes estão sujeitos a distúrbios nas suas linhas de alimentação. Neste sentido, a norma IEC 61.000-4-29 objetiva estabelecer um método básico para teste de imunidade de equipamentos e/ou dispositivos alimentados por fontes de corrente contínua externas de baixa tensão [15].

Esta norma técnica prevê três tipos de distúrbios:

- **Queda de tensão:** é uma queda abrupta de tensão de curto período, na alimentação do dispositivo e/ou equipamento.
- **Pequena Interrupção:** é a interrupção da alimentação do equipamento por um curto período.
- **Variação de tensão:** É a variação gradual da tensão de alimentação, esta pode ser para um valor superior ou inferior, e ainda pode ser de curta ou longa duração.

A seguir estão seguem as tabelas com valores recomendados para a variação de tensão e duração para a aplicação dos distúrbios. Estes níveis estão descritos percentualmente conforme o nível nominal (U_T).

Tabela 9.1: Valores recomendados para queda de tensão [11].

Nível de Variação (%U_T)	Duração (s)
40% a 70%, ou conforme especificação do produto.	0,01, 0,03, 0,1, 0,3, ou conforme especificação do produto

Tabela 9.2: Valores recomendados para pequena interrupção [11].

Duração (s)
0,001, 0,003, 0,01, 0,03, 0,1, 0,3, 1, ou conforme especificação do produto

Tabela 9.3: Valores recomendados para variação de tensão [11].

Nível de Variação (%U_T)	Duração (s)
85% a 120%, 80% a 120% ou conforme especificação do produto.	0,01, 0,03, 0,1, 0,3, 1, ou conforme especificação do produto

PARTE III – METODOLOGIA

10 Proposta

A técnica proposta é baseada em hardware, e implementada através de um I–IP do tipo *watchdog* e é denominado: RTOS–Watchdog (RTOS–WD). Desta forma é possível sua inclusão em um processador sem que sejam necessárias modificações em sua arquitetura e sem gerar impacto em seu desempenho. São apresentados os passos de seu desenvolvimento, assim como sua arquitetura no nível de sistema e a sua arquitetura interna com seus respectivos blocos.

A técnica é baseada em sistemas invariantes, ou seja, com um conjunto conhecido de tarefas e prioridades fixas, visando o menor hardware possível para o RTOS–Watchdog.

É utilizada uma abordagem genérica em seu desenvolvimento, de modo que este possa ser incluso em qualquer processador *multicore*. Para tanto, é considerado na sua concepção que os sinais utilizados para monitoramento são restritos aos mais prováveis de serem encontrados nos diversos processadores multinúcleos existentes.

Evidentemente é necessário que o projetista tenha conhecimento da arquitetura e organização do processador além do conhecimento do *kernel* do sistema operacional pelo menos no que diz respeito as suas rotinas de escalonamento.

Além do conhecimento do sistema operacional, o projetista deverá conhecer os endereços nos quais as funções de escalonamento foram alocadas pelo compilador possibilitando a detecção de suas chamadas pelos monitores.

É necessário conhecer as prioridades das tarefas, assim como as restrições de núcleo de execução (*core lock*) se houver.

Devido às várias CPUs é necessário um controle para a sincronia, para garantir que o RTOS já tenha sido iniciado em ambos os núcleos evitando-se a indicação de falsos erros. Para o controle de início do RTOS, foi cogitado monitorar a o endereço da função *"OS_START"*, porém quando é realizado o *download* do RTOS pelo Plasma (Capítulo 6) na fase de inicialização do processador, esse endereço é requisitado pela CPU para a escrita do RTOS na RAM o que indicaria um falso início durante o *download* do sistema operacional. A solução proposta para sanar este problema é que o RTOS sinalize ao RTOS–Watchdog sua inicialização acessando-o como um periférico, através de um endereço reservado no barramento de periféricos. Quando todos os núcleos realizarem este procedimento é gerado o sinal de *Start.* Neste endereço há um registrador com '*n' bits* (sendo '*n'* o número de núcleos do processador) inicializado com o valor

‘0’ em todos os seus bits. O hardware permite somente a escrita de valores ‘1’ pelo processador, evitando-se a escrita do valor de inicialização deste registrador. Durante a função *“OS_START”*, o RTOS deve realizar esta escrita em um dos bits desse registrador conforme seu índice na arquitetura do processador. No momento em que todos os bits deste registrador são ‘1’ o sinal *Start* é gerado.

Na Figura 10.1 segue um diagrama simplificado da arquitetura de um processador multicore e as conexões de seus blocos com o RTOS–Watchdog (bloco preto).

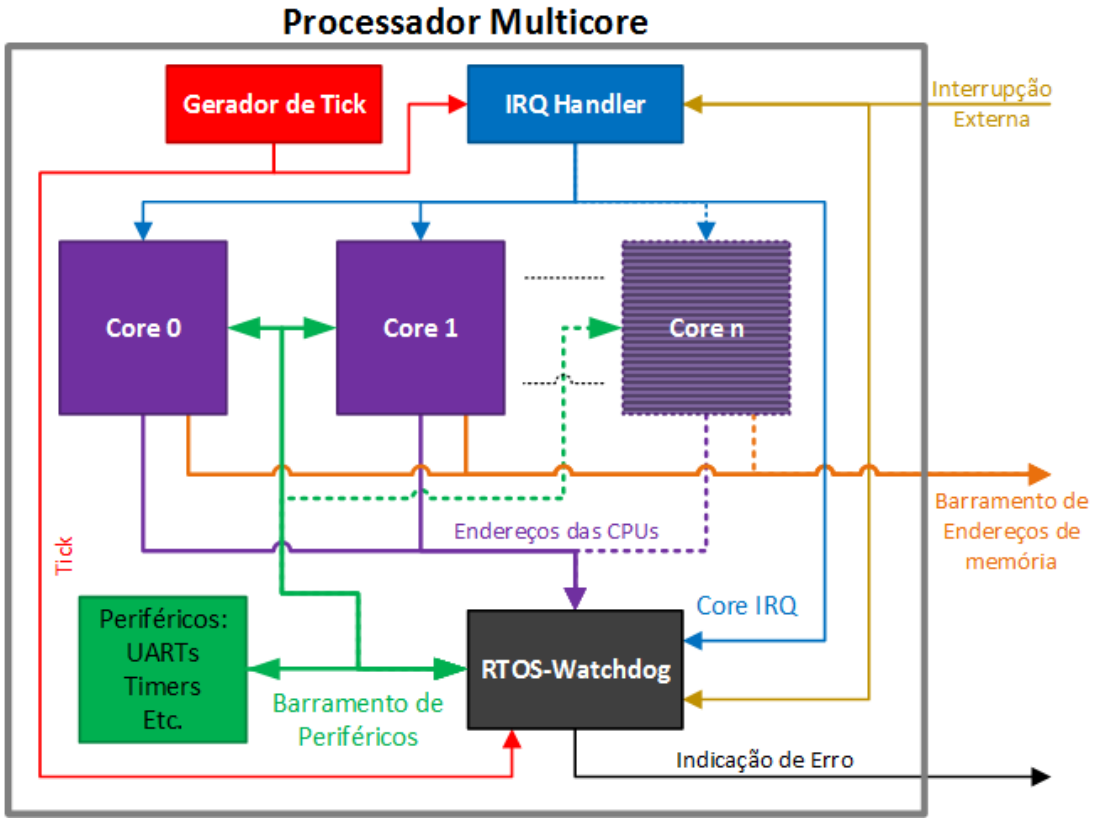


Figura 10.1: Arquitetura externa do RTOS-Watchdog

O Tick Generator é o gerador de *tick*, este é um sinal periódico, uma subdivisão arbitrária do *clock* principal do processador, geralmente na ordem dos milissegundos. Este sinal é utilizado pelo RTOS para controle de tempo em eventos de *timeout*. Em processadores comerciais este sinal pode ser gerador por qualquer timer, desde que este possa gerar interrupção.

O IRQ Handler é o árbitro de interrupção, este módulo é responsável por selecionar o núcleo que atenderá a interrupção ocorrida, no caso do Plasma ele funciona simplesmente alternando-os de forma cíclica.

O Barramento de periféricos é o barramento onde os periféricos do processador são conectados e são acessados através de um ou mais endereços reservados, por exemplo: UART, Timers e GPIOs.

A arquitetura detalhada do RTOS-WD será apresentada nas seções seguintes.

10.1 Arquitetura Interna

A arquitetura do RTOS-WD é composta basicamente por dois tipos de blocos. O primeiro tipo é o Monitor de Eventos de Escalonamento (MEE) (ver 10.1.2), para cada núcleo de processamento é instanciado um MEE e estes atuam de forma independente. O outro bloco é a Unidade de Controle (UC) (ver 10.1.3).

Na Figura 10.2 é demonstrada a arquitetura interna do RTOS-Watchdog.

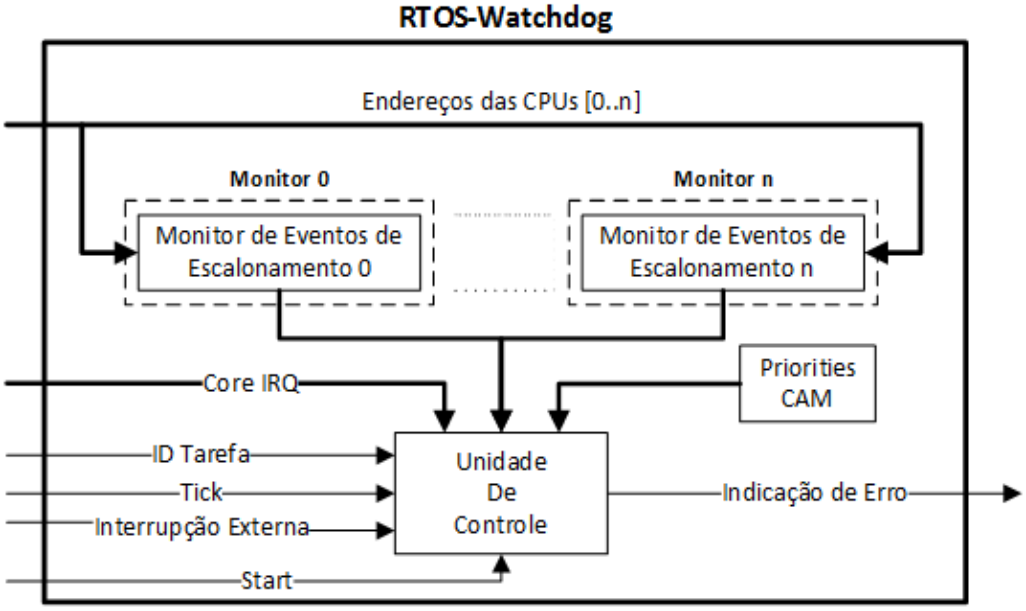


Figura 10.2: Arquitetura interna do RTOS-Watchdog

Os blocos Monitor de Eventos de Escalonamento são os blocos que realizam o monitoramento do RTOS. Identificando os eventos de escalonamento ocorridos no núcleo monitorado, sinalizando para o controle a ocorrência destes eventos.

Finalmente o Controle, é responsável por concentrar os dados oriundos dos demais blocos e processá-los verificando a ocorrência de erros e sinalizá-los quando for o caso.

10.1.1 Monitoramento de Tarefa

Cogitou-se fazer o monitoramento através do endereço de CPU por um monitor específico, porém em alguns casos algumas tarefas interrompidas por alguma interrupção não eram detectadas (exemplo na Figura 10.3), antes de ser gerado outro evento de escalonamento, resultando em uma falsa detecção de erro. E este método possui duas limitações cada tarefa deve ter uma função de entrada única e esta deve ser recorrente durante a execução desta tarefa caso contrário esta acaba não sendo detectada. Tarefas que desempenhassem a mesma função deveriam ser implementadas em diferentes locais do código-fonte implicando num aumento deste com trechos repetidos.

Com o objetivo de se resolver este problema foi reservado outro endereço no barramento do Plasma para o RTOS sinalizar ao RTOS-WD qual tarefa será escalonada. Outra solução para a detecção da tarefa univocamente seria identifica-la através de seu *stack pointer* (SP), porém isso iria requerer uma intrusão maior e uma modificação na arquitetura do processador para ter acesso a esse registrador. Além disso, seria necessário o conhecimento do intervalo para o SP de cada tarefa. Essas modificações fogem a proposta deste trabalho, além de ser impraticável para processadores de código fechado. Com essa solução poder-se-ia detectar inclusive estouro de pilha, uma falha grave que leva o sistema a um estado indeterminado.

As tarefas são identificadas através de um número identificador único. Existem duas tarefas reservadas.

- **Idle Thread**: Tarefa de baixa prioridade escalonada quando não há outras tarefas prontas (5.2.1 Tarefas). Esta tarefa é identificada como ‘0’.
- **Interruption Task**: Tarefa virtual, uma vez que é executada sob a pilha (*stack*) da tarefa interrompida, de alta prioridade escalonada por um evento externo. Esta tarefa é sinalizada por um número do formato $(2^n-1)_{10}$ ou $(11...1)_2$ que seja maior que o número de tarefas configuradas no WD. Por exemplo, quando houver 6 (110_2) tarefas no sistema, a tarefa de interrupção é identificada pelo número 7 (111_2). Ou se houver 10

(1010_2) tarefas no sistema, a tarefa de interrupção é identificada pelo número 15 (1111_2).

Na Figura 10.3 é comparado através de simulação o RTOS-WD com o monitor de tarefas proposto inicialmente (U0), no qual a identificação de tarefas é realizada pelo endereço de CPU e a nova abordagem (U1), com as modificações no monitoramento de tarefa conforme explicado anteriormente, onde a tarefa é informada pelo RTOS no momento da comutação.

Há três cursores nesta figura cada um destacando algum evento:

- O primeiro demonstra o momento em que a tarefa ‘4’ é suspensa em decorrência do evento *rsrcTimeout* gerado no núcleo ‘1’ (SchMon1 – *sched_o*) e decorrente escalonamento da tarefa ‘5’.
- O segundo cursor mostra o momento em que a tarefa ‘4’ é escalonada pelo RTOS e não é detectada pela versão inicial (U0) do RTOS-WD, pois há um segundo escalonamento sem o retorno ao endereço monitorado pelo por este Monitor de Tarefas.
- O terceiro cursor demonstra a falsa detecção de erro sinalizada pela variação no sinal *test_o* de U0 após um evento de escalonamento gerado pela tarefa ‘4’. Os códigos de erros estão explicados em 10.3 Verificações do RTOS.

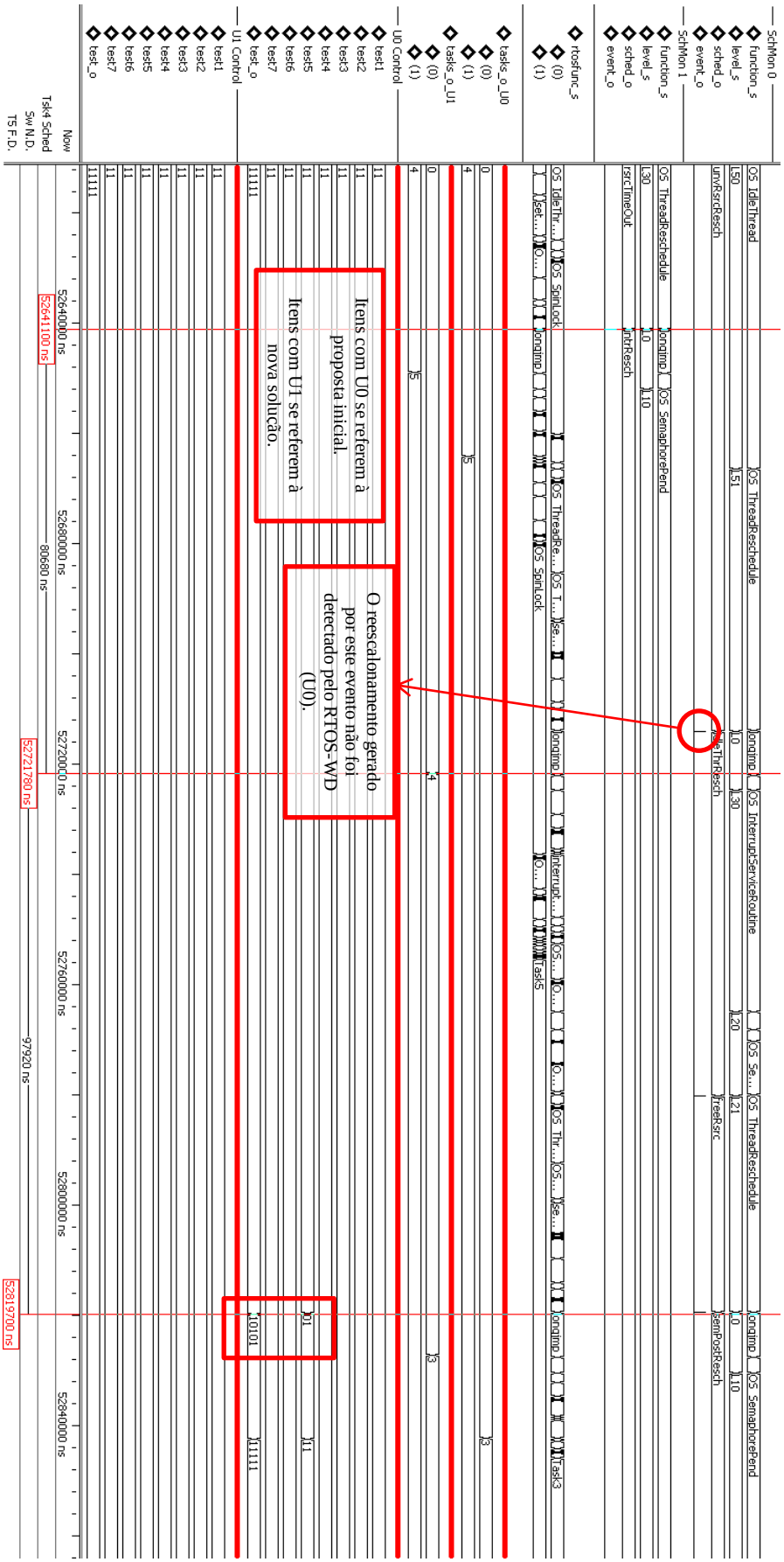


Figura 10.3: Falsa detecção de erro

10.1.2 Monitor de Eventos de Escalonamento

O Monitor de Eventos de Escalonamento (MEE) tem a finalidade de informar à Unidade de Controle do RTOS-WD a ocorrência dos eventos de escalonamentos ocorridos.

Para se realizar este monitoramento foi proposta esta arquitetura para o MEE. Este componente é constituído por um decodificador, cuja entrada é o Endereço de CPU do núcleo monitorado e sua saída é um identificador para alguma das funções monitoradas. O outro componente uma máquina de estados sensível à função detectada pelo decodificador, e a sua finalidade é a detecção dos eventos ocorridos conforme a sequência de funções.

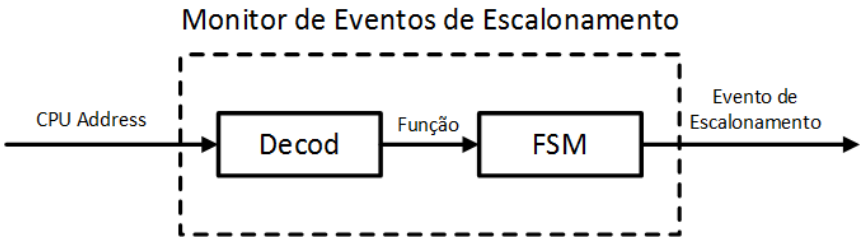


Figura 10.4: Arquitetura Monitor de Eventos de Escalonamento

São identificadas as funções de escalonamento do RTOS, então após a compilação do *kernel* seus endereços devem ser conhecidos. Para a identificação destas funções foram encontradas as funções de onde partem as chamadas para a função OS_ThreadReschedule, pois é esta função que verifica se a tarefa atual deve ser comutada.

Foi desenvolvida uma ferramenta em linguagem C, um *parser*, responsável pela interpretação do mapa de endereços e criação de um *package* com os endereços destas funções, utilizado na configuração no RTOS–WD. No Apêndice A consta um exemplo de *package* gerado pela ferramenta citada. As funções monitoradas pelo MEE estão descritas na Tabela 10.1.

Como cada evento possui uma sequência definida de funções e existem funções comuns a mais de um evento, assim foi necessária a criação de uma FSM (Finite State Machine) para se identificar corretamente cada um dos eventos de escalonamento.

Quando a ocorrência de um evento é detectada, é gerado um sinal para o controle, indicando sua ocorrência e informando qual o evento ocorrido.

Após estudo do Plasma RTOS obteve-se o conjunto mínimo de funções relevantes para o monitoramento. Estas funções estão descritas na Tabela 10.1 são monitoradas oito funções de escalonamento e a *"Idle Thread"* (mencionada em 5.2.1 Tarefas), pois esta pode gerar escalonamento, por exemplo, quando ocorre um evento em outra CPU de liberação de recurso. Então o núcleo ocioso pode executar uma tarefa que esteja pronta.

Tabela 10.1: Funções de escalonamento monitoradas pelo RTOS-Watchdog

Função	**Propósito**
longjmp	Realiza a troca de contexto do processador, e posiciona o *stack pointer* para a pilha da tarefa selecionada.
OS_ThreadTimeoutRemove	Remove uma tarefa da lista de *timeout*, assim esta pode ser escalonada.
OS_ThreadReschedule	Determina qual tarefa deve ser executada, obedecendo a restrições de núcleo, prioridade e estados.
OS_SemaphorePost *	Realiza a operação de incremento de um semáforo. Liberando tarefas pendentes quando estas existirem.
OS_SemaphorePend *	Realiza a operação de decremento de um semáforo, suspendendo a tarefa requerente até sua liberação por outra tarefa.
OS_ThreadExit *	Indica ao RTOS que a tarefa foi finalizada e deve ser retirada da lista de tarefas.
OS_InterruptServiceRoutine *	Esta função é chamada para o atendimento de qualquer interrupção como Interrupção Externa ou *Tick.* Seu proposito é chamar a função que atenderá a interrupção ocorrida.
OS_ThreadTick	Seu objetivo é atender a interrupção de *tick*, liberando tarefas com *timeout*.
OS_IdleThread *	É a tarefa de baixa prioridade. Serve para quando não há tarefas em estado "pronto". Tem como objetivo secundário gerar um evento de escalonamento, caso tenha sido liberado um recurso em alguma

Função	Propósito
	tarefa em execução outro núcleo.

* Funções primárias: Geradoras de eventos de escalonamento.

Com esse conjunto de funções é possível considerar oito eventos de escalonamento *(Scheduling Event)* vide Tabela 10.2. Conforme mencionado o sistema deve ser invariante, há ainda mais eventos de escalonamento relativos à troca de prioridade de uma tarefa através da função (*OS_ThreadPrioritySet*), e ao criar uma tarefa após a inicialização do RTOS (*OS_ThreadCreate*), a adaptação requereria o aumento da interface com o processador resultando em aumento do hardware.

Há somente funções relativas a semáforos, pois neste RTOS, os outros objetos do escalonador, *mutexes* e *message queues*, são implementados como uma camada adicional no semáforo e o RTOS-WD consegue detectar os eventos de escalonamento gerado por estes objetos normalmente. Simplificando-se assim o hardware do monitor.

Os eventos identificados preveem: liberação ou aquisição de recursos e reescalonamento de tarefa.

A Tabela 10.2 descreve os eventos identificados no Plasma-RTOS e o seu respectivo código para referência nas Figuras 10.5-‘a’, ‘b’, ‘c’, ‘d’, ‘e’ e ‘f’:

Tabela 10.2: Eventos de escalonamento detectados pelo RTOS-Watchdog

Nome	Código
Escalonamento por recurso indisponível	unvRsrcResch
Liberação de Recurso (não gera escalonamento de tarefas)	freeRsrc
Reescalonamento por liberação de semáforo	semPostResch
Evento de Tick	tickEvent
Reescalonamento após interrupção	intrResch
Liberação de recurso por fim de tempo de espera	rsrcTimeOut
Fim de tarefa	thrExitResch
Reescalonamento por Idle Thread	idleThrResch

Segue a tabela de estados do MEE, onde cada desvio é realizado com base em uma função de escalonamento executada pelo RTOS no núcleo monitorado, conforme a Tabela 10.1, e os eventos detectados são os descritos na Tabela 10.2.

A Tabela 10.3 é a Tabela de Transição de Estados do MEE. É exposto nesta tabela todas as transições possíveis deste componente, e quando for o caso o evento gerado nesta transição.

Tabela 10.3: Tabela de Transição de Estados do MEE

Estado Atual	Função Chamada	Estado Futuro	Evento Gerado
S0	OS_SemaphorePend	S10	
	OS_SemaphorePost	S20	
	OS_InterruptServiceRoutine	S30	
	OS_ThreadExit	S40	
	OS_IdleThread	S50	
S10	longjmp	S0	unvRsrcResch
	OS_SemaphorePost	S20	
	OS_InterruptServiceRoutine	S30	
	OS_ThreadExit	S40	
	OS_IdleThread	S50	
S20	OS_ThreadReschedule	S21	freeRsrc
	OS_SemaphorePend	S10	
	OS_InterruptServiceRoutine	S30	
	OS_ThreadExit	S40	
	OS_IdleThread	S50	
S21	longjmp	S0	semPostResch
	OS_SemaphorePend	S10	
	OS_SemaphorePost	S20	
	OS_InterruptServiceRoutine	S30	
	OS_ThreadExit	S40	
	OS_IdleThread	S50	
S30	longjmp	S0	intrResch
	OS_ThreadTick	S31	TickEvent
	OS_SemaphorePend	S10	
	OS_SemaphorePost	S20	
	OS_ThreadExit	S40	

Estado Atual	Função Chamada	Estado Futuro	Evento Gerado
	OS_IdleThread	S50	
S31	OS_ThreadTimeoutRemove	S31	rsrcTimeout
	OS_SemaphorePend	S10	
	OS_SemaphorePost	S20	
	OS_InterruptServiceRoutine	S30	
	OS_ThreadExit	S40	
	OS_IdleThread	S50	
S40	longjmp	S0	thrExitResch
	OS_SemaphorePend	S10	
	OS_SemaphorePost	S20	
	OS_InterruptServiceRoutine	S30	
	OS_IdleThread	S50	
S50	longjmp	S0	idleThrResch
	OS_SemaphorePend	S10	
	OS_SemaphorePost	S20	
	OS_InterruptServiceRoutine	S30	
	OS_ThreadExit	S40	

Abaixo seguem os fluxogramas de cada estado cuja ideia é representar o explicado na Tabela 10.3. Nos fluxogramas abaixo podem aparecer funções com o nome alterado por questões de espaço e simplificação. Se a função atual for a escrita dentro dos losangos o fluxo é desviado para o terminal ‘S’ caso contrário vai ao terminal ‘N’. Onde aparecer uma caixa com o texto: “schedEvent =” significa que é gerado um evento pelo MEE, o círculo significa que há uma troca de estado.

O estado S0 é o estado inicial, o MEE é mantido neste estado até a ocorrência de uma função primária de escalonamento e retorna a este estado após a conclusão de algum evento.

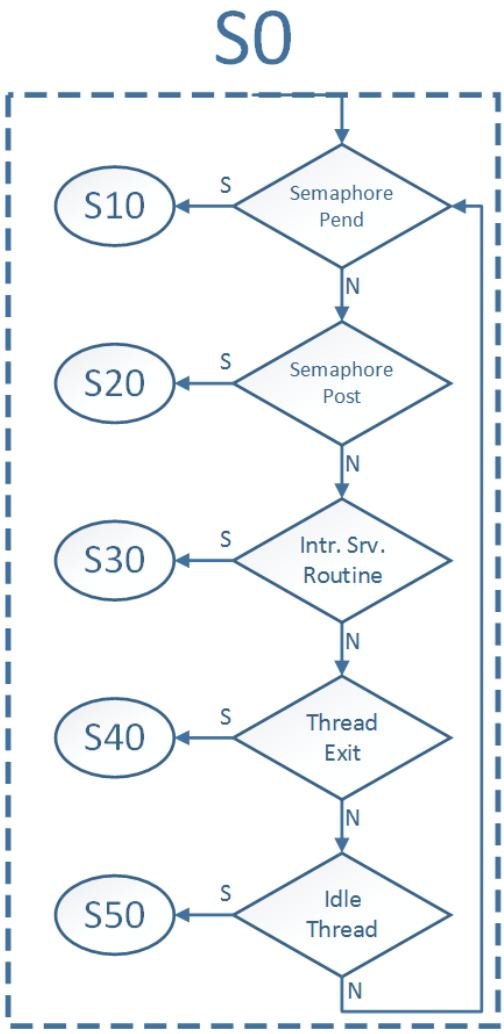


Figura 10.5-a: Fluxograma do Monitor de Eventos de Escalonamento

No momento em que uma tarefa requer acesso à um recurso protegido é chamada a função *"OS_Semaphore_Pend"*. Quando esta função é chamada o estado do monitor é alterado para o S10, este estado pode gerar o evento *unvRsrcResch* caso o recurso requisitado esteja indisponível, ao ser detectada a chamada para a função *"longjmp"*.

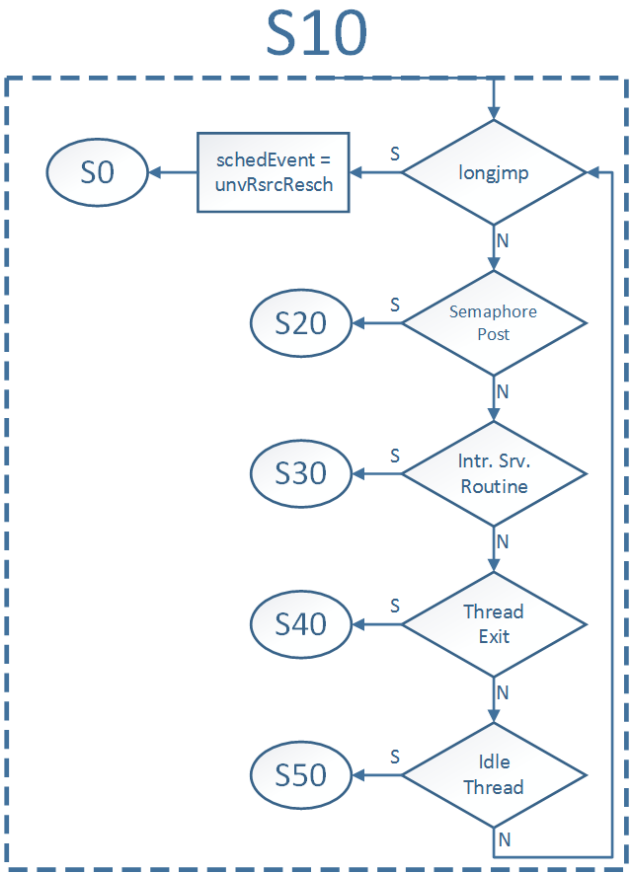


Figura 10.5-b: Fluxograma do Monitor de Eventos de Escalonamento

Ao concluir o acesso a um recurso uma tarefa deve chamar a função *"OS_Semaphore_Post"*. Neste momento o monitor altera o seu estado para o S20. Se houver uma ou mais tarefas aguardando liberação do recurso é então chamada a função *"OS_ThreadReschedule"* neste momento o monitor gera o evento *freeRsrc* e o seu estado é alterado para o S21. Caso alguma tarefa pendente deste recurso tiver a prioridade maior que a atualmente em execução essa é automaticamente escalonada, e neste caso é chamada a função *"longjmp"*. Neste momento é gerado o evento *semPostResch* e o monitor retorna ao estado S0.

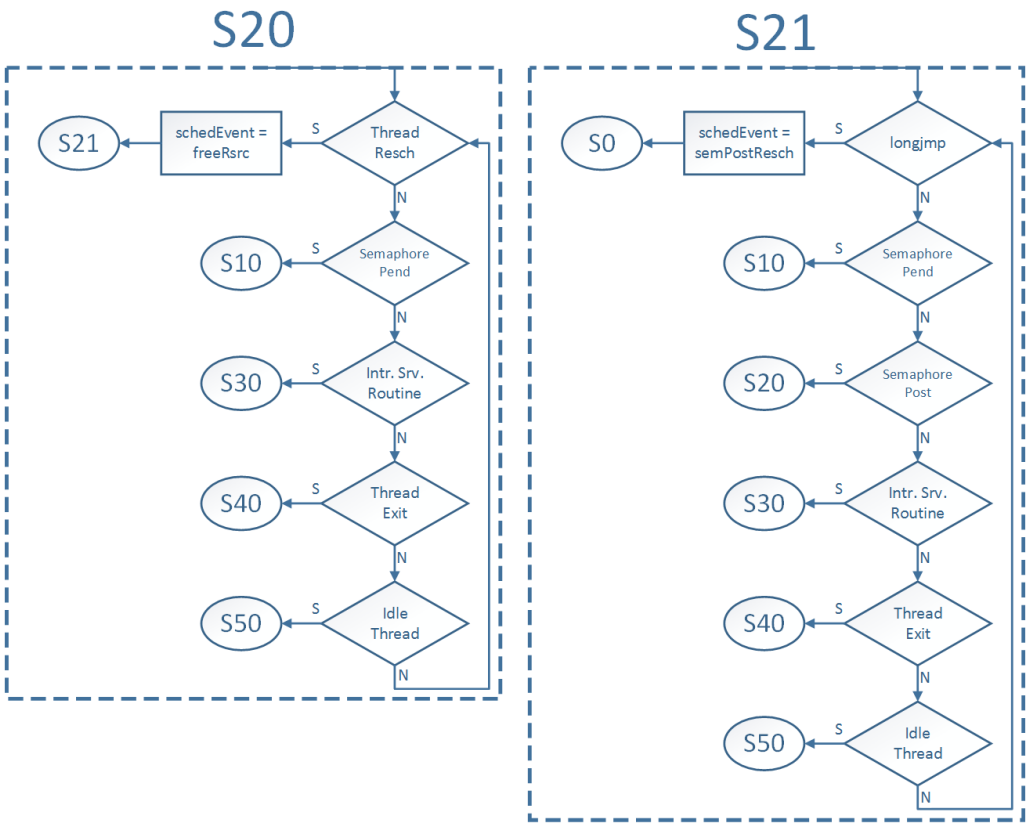


Figura 10.5-c: Fluxograma do Monitor de Eventos de Escalonamento

A função *"OS_InterruptServiceRoutine"* é responsável por verificar qual foi a interrupção e chamar a função adequada para seu atendimento. Ao ser detectada a chamada para esta função o monitor segue ao estado S30.

Após uma interrupção pode ser gerado um evento de escalonamento, este é detectado pela chamada da função *"longjmp"*, este evento é identificado como *intrResch* e faz o monitor retornar ao estado S0.

Caso a interrupção ocorrida seja uma interrupção de *tick* a função *"OS_ThreadTick"* é chamada, então o monitor segue ao estado S31 e é gerado o evento *tickEvent*. No estado S31 são detectadas as liberações de recursos por *time out*, através da chamada da função *"OS_ThreadTimeoutRemove"* este evento é identificado por *rsrcTimeOut*.

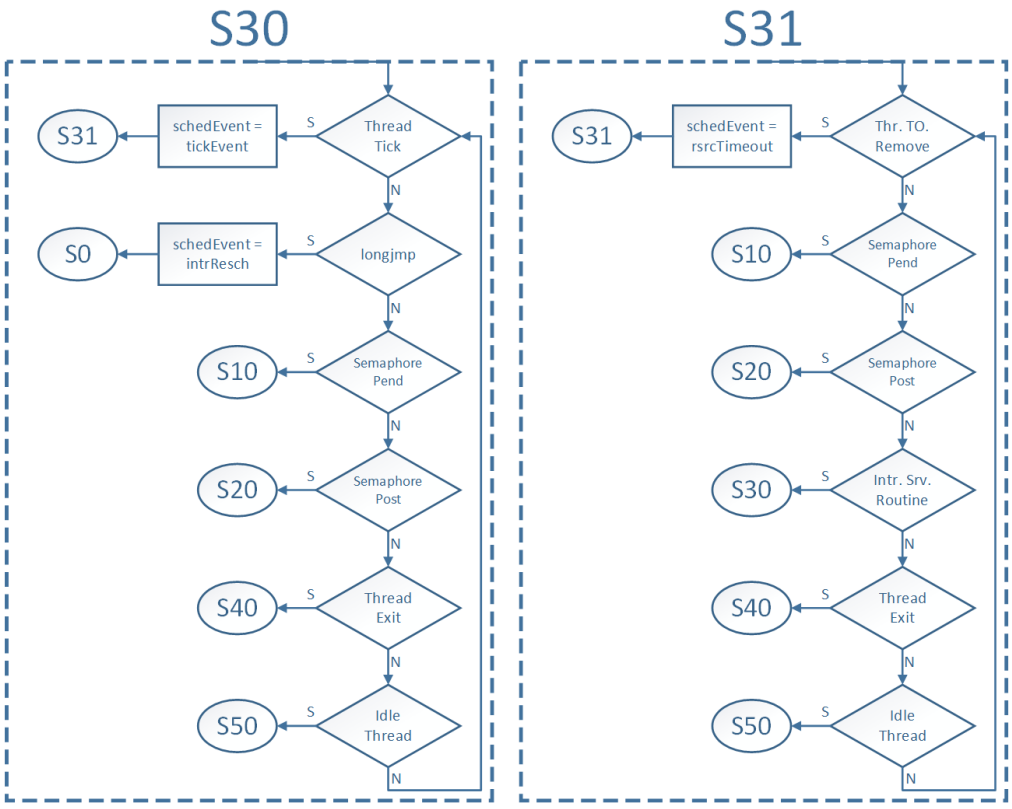


Figura 10.5-d: Fluxograma do Monitor de Eventos de Escalonamento

Ao ser concluída uma tarefa esta deve informar ao escalonador. Este procedimento é identificado através da chamada para a função *"OS_ThreadExit"*. Este procedimento deve gerar um reescalonamento identificado pela chamada da função *"longjmp"* neste momento é gerado o evento *thrExitResch* e o monitor retorna ao estado S0.

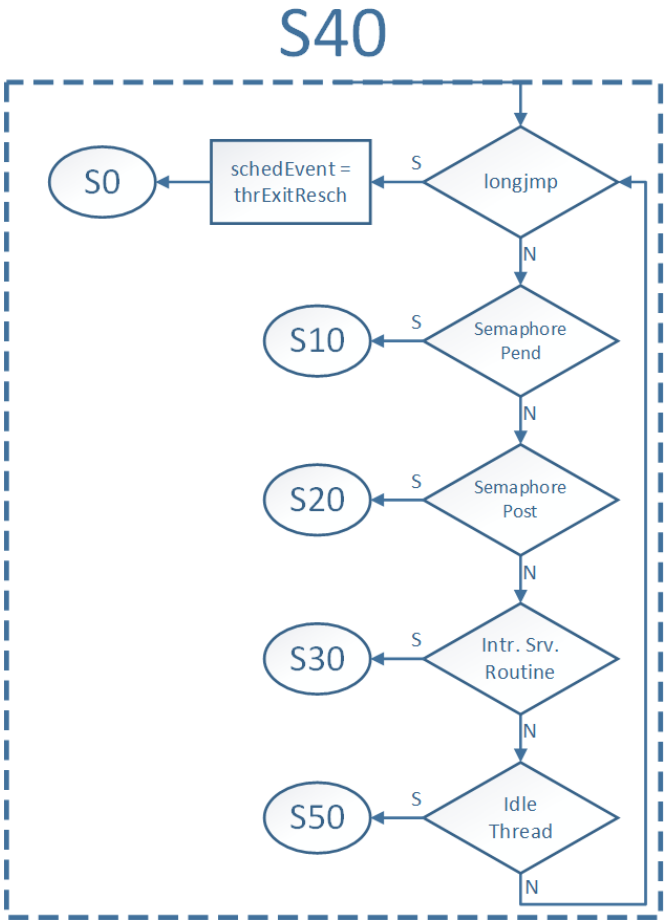


Figura 10.5-e: Fluxograma do Monitor de Eventos de Escalonamento

Quando as demais tarefas estão bloqueadas por algum motivo, o escalonador escalona a *Idle Thread*. Esta tarefa é identificada pela chamada para a função *"OS_IdleThread"*. Neste momento o monitor segue ao estado S50. No PlasmaRTOS a *Idle Thread* periodicamente verifica se há tarefas pendentes. Esta verificação é realizada através da função *"OS_ThreadReschedule"*. Se houver necessidade de reescalonamento a função "*longjmp*" é chamada e a sua detecção gera o evento *idleThrResch*.

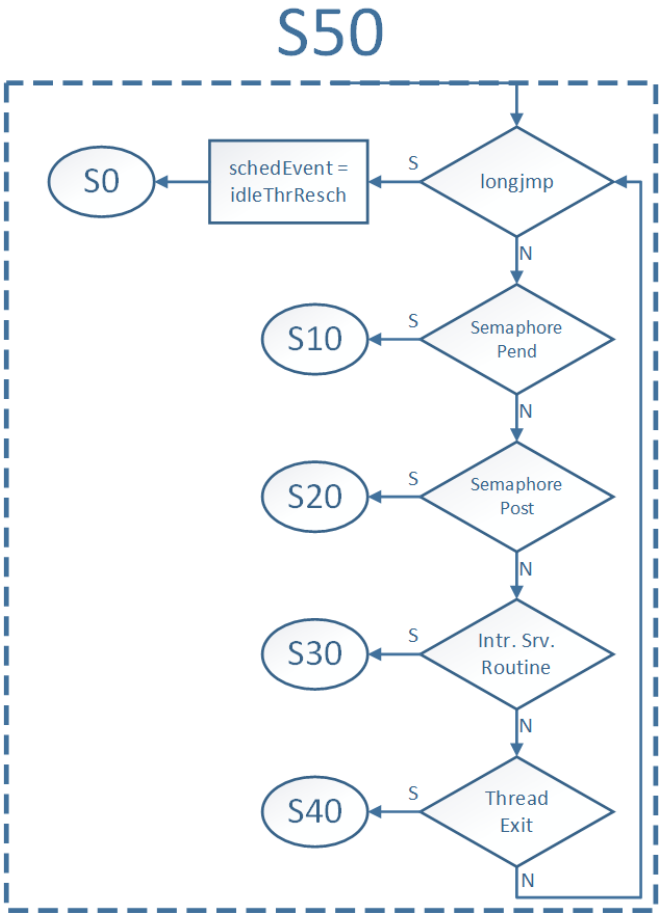


Figura 10.5-f: Fluxograma do Monitor de Eventos de Escalonamento

Segue um exemplo do fluxo de funções para o acesso a porta serial (UART).

1. Por exemplo, é solicitado acesso a uma porta serial protegida por semáforo ocorre o seguinte fluxo, partindo-se do estado S0, será avaliado o que ocorre no caso desta por estar sendo utilizada ou ociosa:

   É chamada a função *"OS_SemaphorePend"* no requerimento de acesso. Então a FSM avança para o S10;

   a. Serial Ocupada: A contagem do respectivo semáforo é zero, ou menor mais que uma tarefa aguardando, logo a função *"OS_ThreadReschedule"* é chamada e em seguida a função *"longjmp"* para reescalonar outra tarefa. Assim o evento *unvRsrcResch* é gerado e a máquina de estados retorna ao estado S0.

b. Serial ociosa: Neste caso a FSM continua no estado S10 até que outra função primária seja chamada, pois não a um *timeout* especificado para a chamada das funções secundárias (*OS_ThreadReschedule* e *longjmp*).

O próximo exemplo demonstra fluxo de funções no momento em que uma tarefa encerra o uso da interface serial (UART).

2. Liberação da porta serial pela tarefa em execução. Supondo que a FSM esteja partindo no estado S0.

   Ao final de seu uso a função *"OS_SemaphorePost"* é chamada. Assim a FSM passa ao estado S20.

   a. Caso o semáforo esteja com o contador menor que zero, ou seja, se outras tarefas estiverem requisitando este semáforo às mesmas estão bloqueadas. Dentre as tarefas que estão aguardando este recurso a de maior prioridade tem seu estado alterado para PRONTO. Então função *"OS_ThreadReschedule"* é chamada. A chamada desta função gera o evento *freeRsrc* (Recurso liberado) e então a FSM avança ao estado S21.

      i. Se houver outra tarefa de maior prioridade que a tarefa em execução estiver aguardando a porta serial o sistema operacional chama a função *longjmp* e o evento *semPostResch* é gerado e a FSM retorna ao estado S0, caso contrário;

      ii. Se não houver outras tarefas de maior prioridade aguardando a serial, a execução da tarefa atual continua normalmente e a FSM permanece no estado S21 até que seja chamada outra função primária de escalonamento.

10.1.3 Unidade de Controle

A Unidade de Controle (UC) é responsável pelo levantamento e processamento dos dados oriundos dos Monitores de Eventos de Escalonamento, tarefas em execução além dos sinais de Tick e Interrupção Externa. Com base nos eventos ocorridos e os dados informados pelos monitores é possível detectar algum ERRO e sinalizá-lo.

Esta unidade registra todos os sinais de entrada, pois enquanto os eventos ocorridos em um núcleo são processados existe a possibilidade da ocorrência de um evento em outro núcleo, o registro evita a perda destes dados inclusive para o caso de eventos simultâneos em vários núcleos. Os núcleos do processador são analisados por fatia de tempo ciclicamente, ou seja, $\{0, 1, \cdots, (n-1), 0, \cdots\}$, sendo n o número de núcleos do processador.

O tempo de processamento de uma CPU depende dos eventos ocorridos, variando entre dois e três ciclos de *clock*. O caso crítico ocorre ao ser escalonada uma tarefa.

O controle possui uma máquina de estados finitos com três estados.

- No primeiro estado são lidos os registradores de eventos internos do controle.
- No segundo estado é realizado processamento com base nos eventos ocorridos, por exemplo, um bloqueio de recurso. Todos as verificações (10.3 Verificações do RTOS) com exceção da Verificação ‘2’ são realizadas neste estado.
- O terceiro estado só ocorre se houver um evento de tarefa e não houver detecção de falha na Verificação ‘1’, ou seja, outra tarefa é escalonada pelo RTOS e a Verificação ‘1’ (10.3 Verificações do RTOS) não indica a ocorrência de algum erro. Devido a dependência da Verificação ‘2’ (10.3 Verificações do RTOS) do resultado da Verificação ‘1’ e do Gerenciamento de Recursos (vetor “*readyTasks*” mais especificamente) tornou-se necessária a criação de um terceiro estado para a realização desta verificação.

Na Figura 10.6 observa-se o fluxograma da unidade de controle. Neste fluxograma observa-se que a FSM avança para o estado ‘S2’ devido a um Evento de Tarefa.

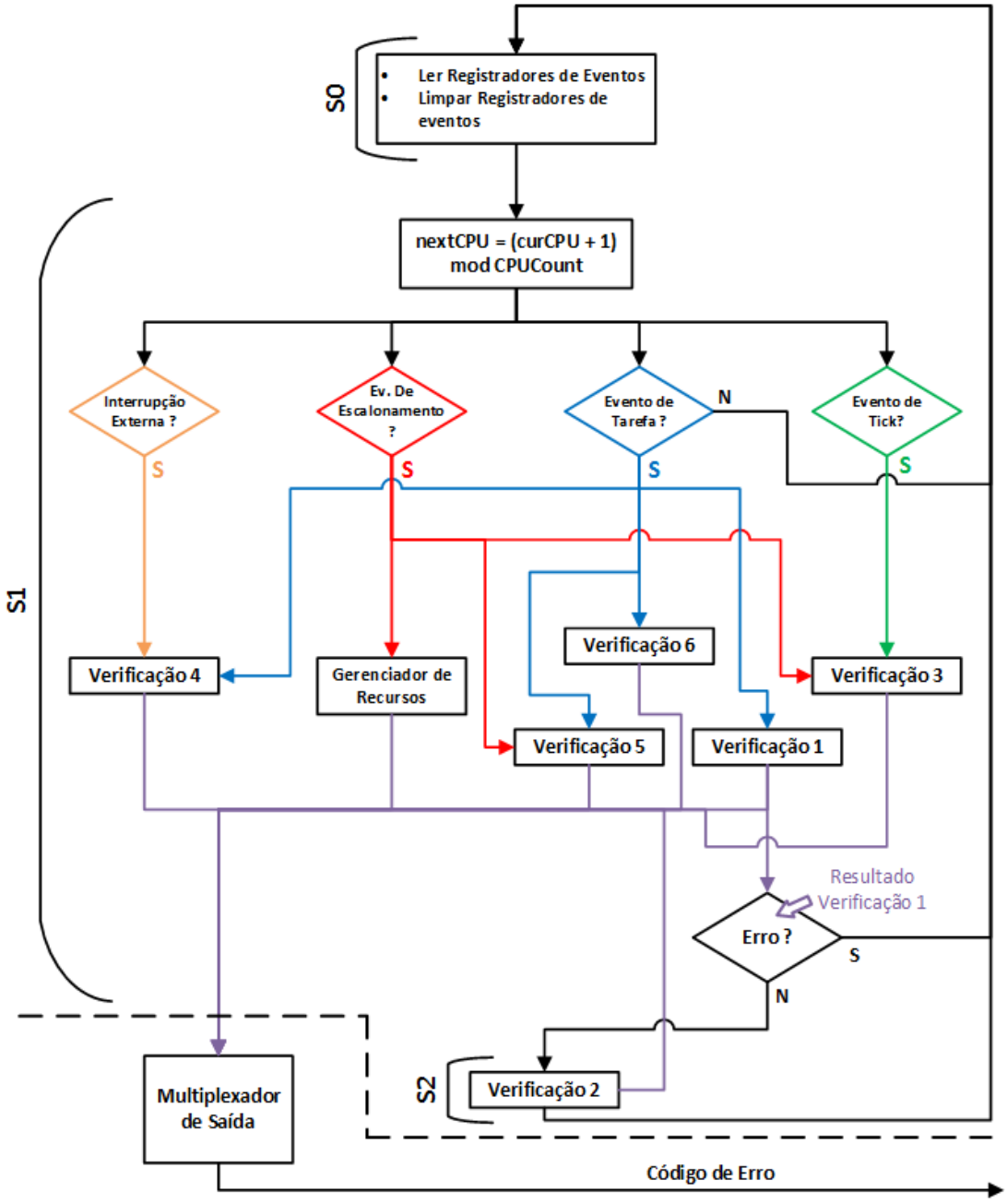


Figura 10.6: Fluxograma da Unidade de Controle do RTOS-Watchdog

O RTOS–WD retorna um vetor de 5 bits. Caso não seja detectado nenhum erro esse vetor terá o valor ‘11111’. Em outros casos os três bits mais significativos indicam qual verificação detectou erro, através de um multiplexador, e os dois bits menos significativos demonstram o código de erro. Os códigos dos erros estão especificados detalhadamente na seção 10.3 Verificações do RTOS.

Os eventos de escalonamento indicados nas figuras abaixo são os descritos na Tabela 10.2. Os outros sinais, operações e constantes seguem na Tabela 10.4.

Tabela 10.4: Legenda do fluxograma das Verificações do RTOS

Nome	Código	Tipo
Tarefas restritas ao núcleo sob análise	curCoreLocked(.)	Constante
O índice neste vetor é a tarefa. Se houver restrição no núcleo atual o bit é '1'.		
Tarefa de Interrupção Externa	intTask	Constante
Se a tarefa for igual a este valor então é a tarefa de interrupção externa.		
Tarefas com restrição de núcleo de execução	taskCoreLocked(.)	Constante
O índice neste vetor é a tarefa. Se esta tarefa possuir restrição para qualquer núcleo o bit é '1'. Este vetor deve ser o resultado da operação 'OU' entre as restrições de todos os núcleos.		
Número de tarefas	taskCount	Constante
Decremento de uma unidade	Dec(.)	Operação
Incremento de uma unidade	Inc(.)	Operação
Evento de escalonamento atual	curSched	Sinal
Tarefa atual em execução	curTask	Sinal
Código de Erro	ErrCode	Sinal
Estado da FSM da U.C.	fsmState	Sinal
Interrupção Externa	intEvent	Sinal
Evento de Interrupção Ocorrido	intHappened	Sinal
Contador de Timeout da Interrupção Externa	intTOCount	Sinal
Nenhum núcleo atendeu a Interrupção Externa	noCPUTriggeredInt	Sinal
Nenhum núcleo atendeu a Interrupção de Tick	noCPUTriggeredTck	Sinal
Prioridade de tarefa	Priority(.)	Sinal
Tarefas com o estado de pronto	readyTasks(.)	Sinal
Número de recursos bloqueados	rsrcBloqued	Sinal
Tarefas em execução	runningTasks	Sinal
Evento de escalonamento ocorrido	schedHappened	Sinal
Evento de Escalonamento	schEv	Sinal

Nome	Código	Tipo
Número de tarefas bloqueadas	taskBloqued	Sinal
Evento de troca de tarefa	taskEvent	Sinal
Contador de Timeout do Tick	tckTOCount	Sinal
Interrupção de Tick	tickEvent	Sinal
Evento de Tick ocorrido	tickHappened	Sinal

10.2 Monitoramento de recursos

A Unidade de Controle deve monitorar os recursos do sistema operacional assim como o estado das tarefas. Há dois vetores que armazenam o estado das tarefas. Um destes vetores é o *runningTasks* dentre seus bits os com valores ‘1’ correspondem as tarefas que estão em execução. O segundo vetor *readyTasks* seus bits com valor ‘1’ correspondem a tarefas prontas e com valor ‘0’ a tarefas bloqueadas. A largura destes vetores corresponde ao número de tarefas informado pelo projetista durante a síntese. Além dos vetores descritos anteriormente há dois contadores: *rsrcBloqued* e *tasksBloqued* o limite destes contadores correspondem ao número de tarefas informado pelo projetista durante a síntese.

Os contadores *tasksBloqued* e *rsrcBloqued* são incrementados simultaneamente na ocorrência do evento *unvRsrcResch*. Porém a operação oposta é realizada em duas etapas o contador *rsrcBloqued* é decrementado ao ser detectada a liberação de um recurso e o contador *tasksBloqued* no momento em que uma tarefa bloqueada for escalonada (10.3.1 Verificação 1: Disponibilidade de recursos), assim é possível se verificar se há recursos disponíveis para esta operação, uma vez que o RTOS–WD, não consegue detectar qual recurso esta sendo bloqueado ou liberado.

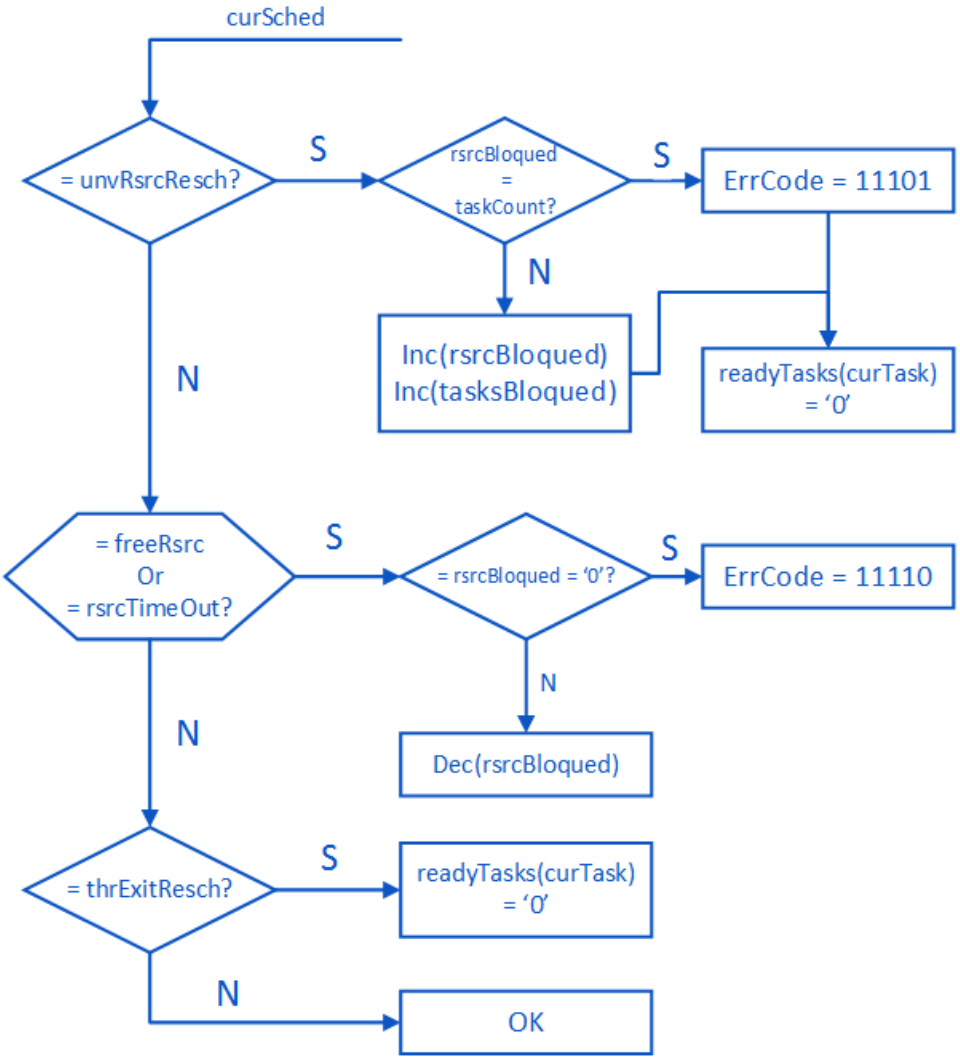


Figura 10.7: Gerenciador de Recursos

10.3 Verificações do RTOS

A Unidade de Controle realiza o procedimento de monitoramento da atividade de escalonamento de tarefas do RTOS. A seguir, descrevem-se em detalhes as seis verificações realizadas pela Unidade de Controle do RTOS-WD e a Verificação '7' é na realidade uma auto verificação do RTOS–WD. Nos fluxogramas abaixo os sinais são os descritos na seção anterior e os eventos são os descritos na Tabela 10.2.

A persistência do código de erro na saída do RTOS–Watchdog pode ser configurada no momento de sua síntese, assim evita-se que um código de erro seja perdido.

10.3.1 Verificação 1: Disponibilidade de recursos

Esta verificação compara o número de tarefas bloqueadas (*tasksBloqued*) com o número de recursos bloqueados (*rsrcBloqued*). Se a tarefa escalonada estiver com

estado de BLOQUEADA e o número de recursos bloqueados for maior ou igual ao de tarefas bloqueadas, é gerado um erro *'00101'*, indicando recursos livres insuficientes.

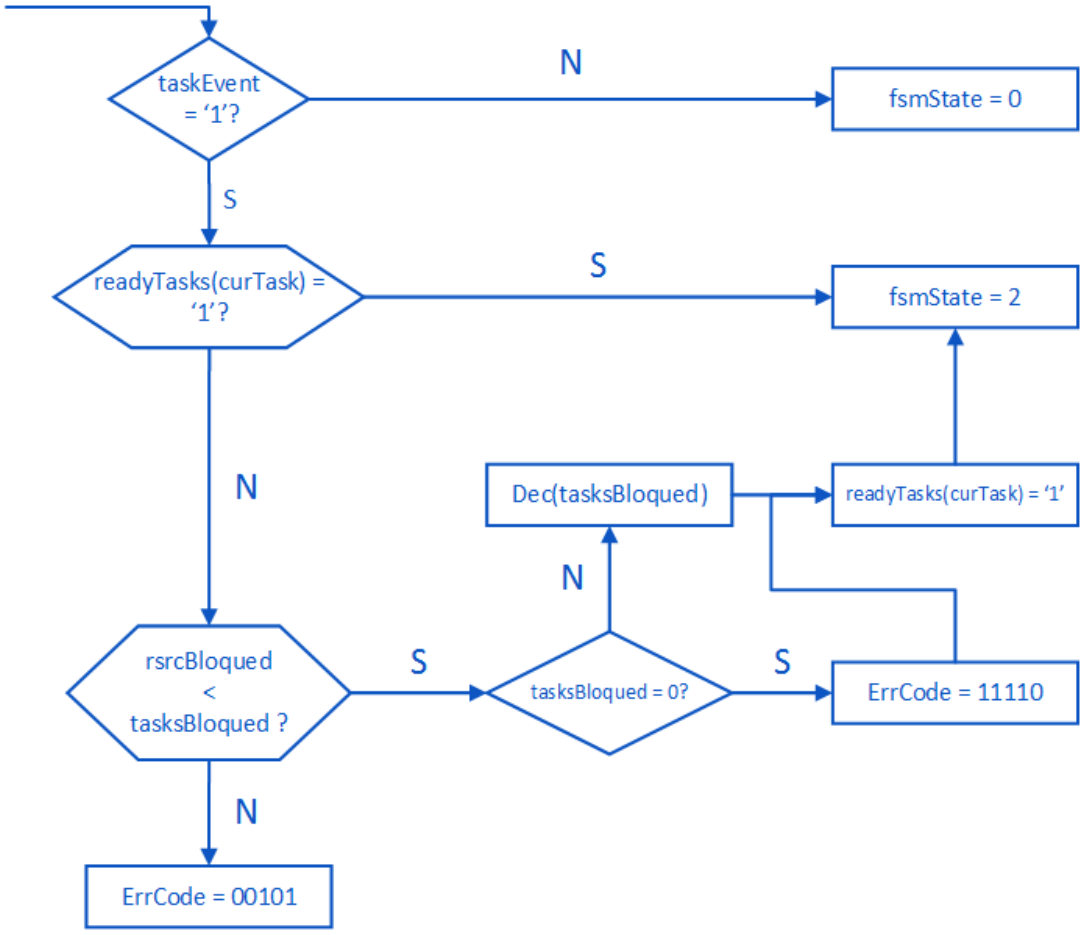


Figura 10.8: Fluxograma da Verificação 1 – Disponibilidade de recursos

10.3.2 Verificação 2: Inversão de prioridade

Verifica se a tarefa atual é a de maior prioridade disponível. Este comparador verifica a prioridade das tarefas prontas com a tarefa atual. Assim é possível se verificar se a tarefa atual é a esperada. Caso seja detectado um erro a unidade de controle reportará o erro *'01001'*. Devido a natureza de tempo real dos sistemas alvos deste trabalho, toda a lista é comparada simultaneamente como uma Content-Addressable Memory (CAM) o resultado desta verificação é obtido em um ciclo de *clock*. Entretanto esta verificação pode ser implementada de forma iterativa, o que iria requerer vários ciclos de *clock* para ser concluída conforme o número de tarefas.

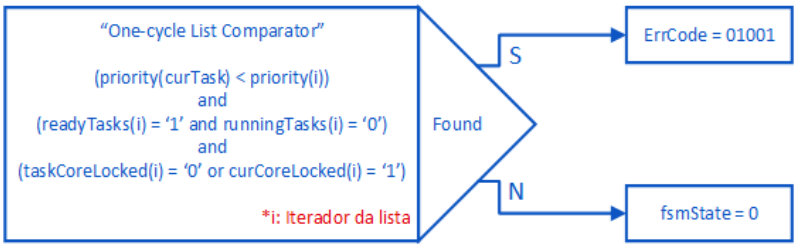


Figura 10.9: Fluxograma da Verificação 2 – Inversão de prioridade

10.3.3 Verificação 3: Tick

Esta verificação consiste em conferir se a interrupção gerada pelo *tick* é atendida pelo RTOS. Esta verificação retorna os seguintes códigos de erros:

- Se o árbitro de interrupção não selecionar nenhuma das CPUs para atender o evento é gerado o erro *'01100'*;
- Caso o RTOS não atenda um evento de *tick* após um *timeout*, configurado pelo projetista durante a síntese do RTOS–Watchdog, é gerado o erro *'01101'*. A diferença deste caso para o anterior é que no primeiro caso é uma falha de hardware, o Árbitro não selecionou um núcleo para atender esta interrupção, neste caso é falha de software, não houve a chamada para a função de atendimento;
- Ou se ocorrer um desvio para a função de atendimento do *tick* sem a ocorrência deste é gerado o erro *'01110'*.

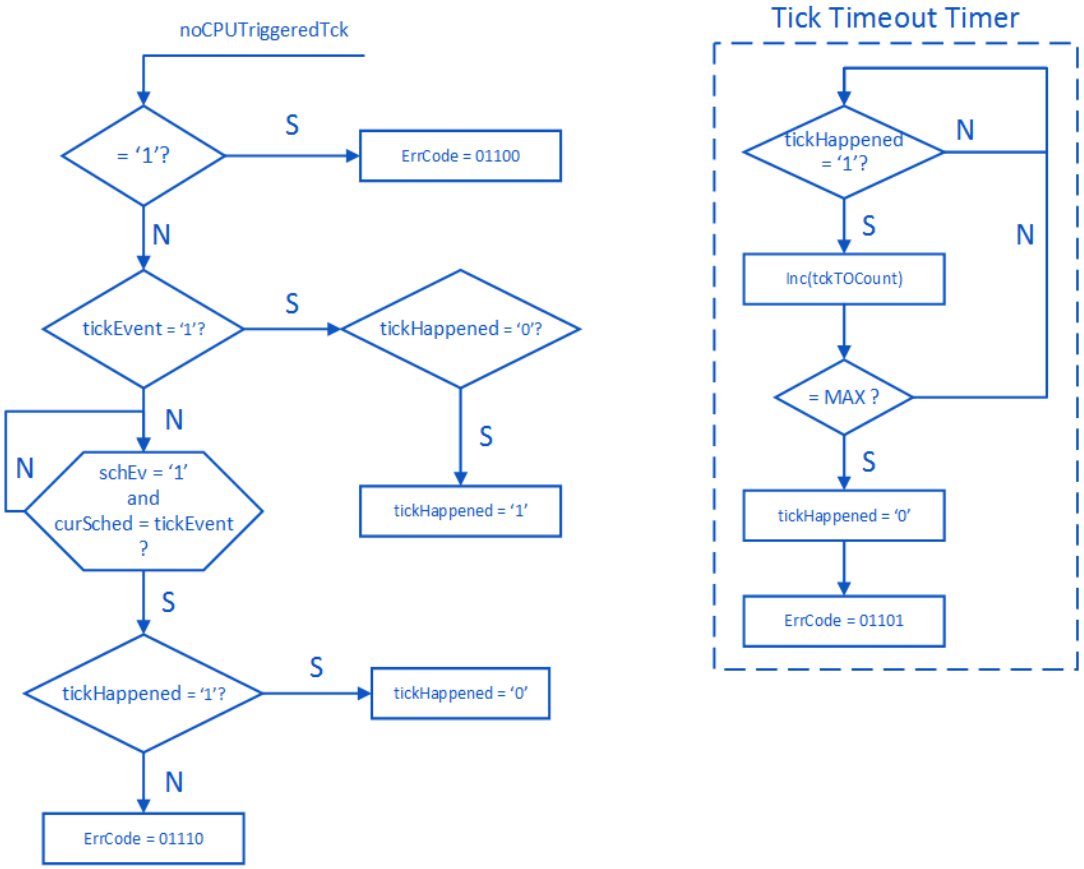


Figura 10.10: Fluxograma da Verificação 3 – Tick

10.3.4 Verificação 4: Interrupção Externa

Esta verificação é análoga à Verificação 3. Porém o alvo desta é o atendimento da interrupção externa. Esta verificação gera os seguintes códigos de erro:

- Se o árbitro de interrupção não selecionar nenhuma das CPUs para atender o evento é gerado o erro *'10000'*;
- É gerado o erro *'10001'* se o RTOS não atender a interrupção externa após um *timeout*. A diferença deste caso para o anterior é que no primeiro caso é uma falha de hardware neste caso é falha de software
- E o erro '10010' na ocorrência de um desvio para a função de atendimento da interrupção sem a ocorrência do devido evento.

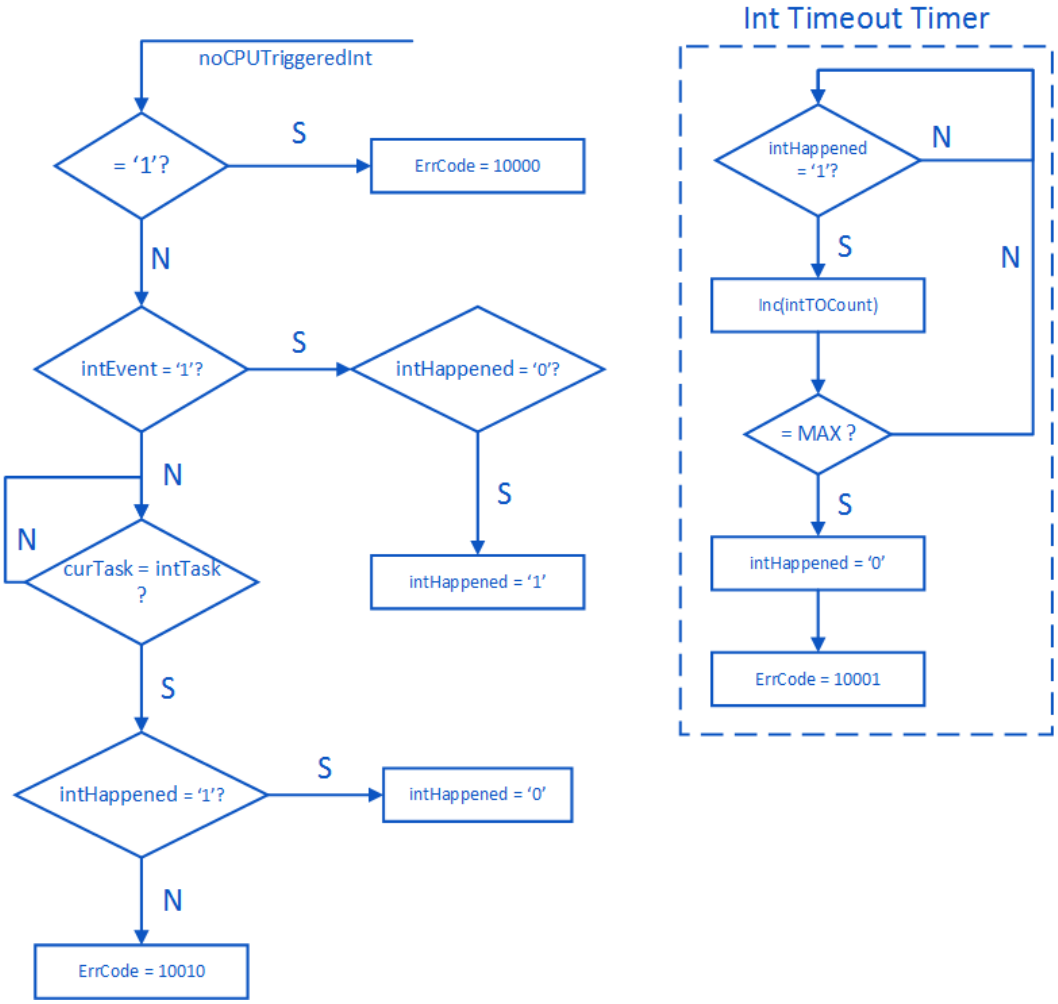


Figura 10.11: Fluxograma da Verificação 4 – Interrupção Externa

10.3.5 Verificação 5: Reescalonamento

Esta verificação consiste em conferir se os eventos de escalonamentos ocorridos foram executados, ou seja, houve a comutação da tarefa. Esta verificação pode retornar dois códigos de erro:

- *'10101'*: Um evento de reescalonamento deve ser atendido obrigatoriamente. Se ocorrer um segundo evento de reescalonamento sem que o anterior seja efetivado este erro é gerado;
- *'10110'*: Reescalonamento de tarefas inesperado realizado sem ocorrência de evento de reescalonamento.

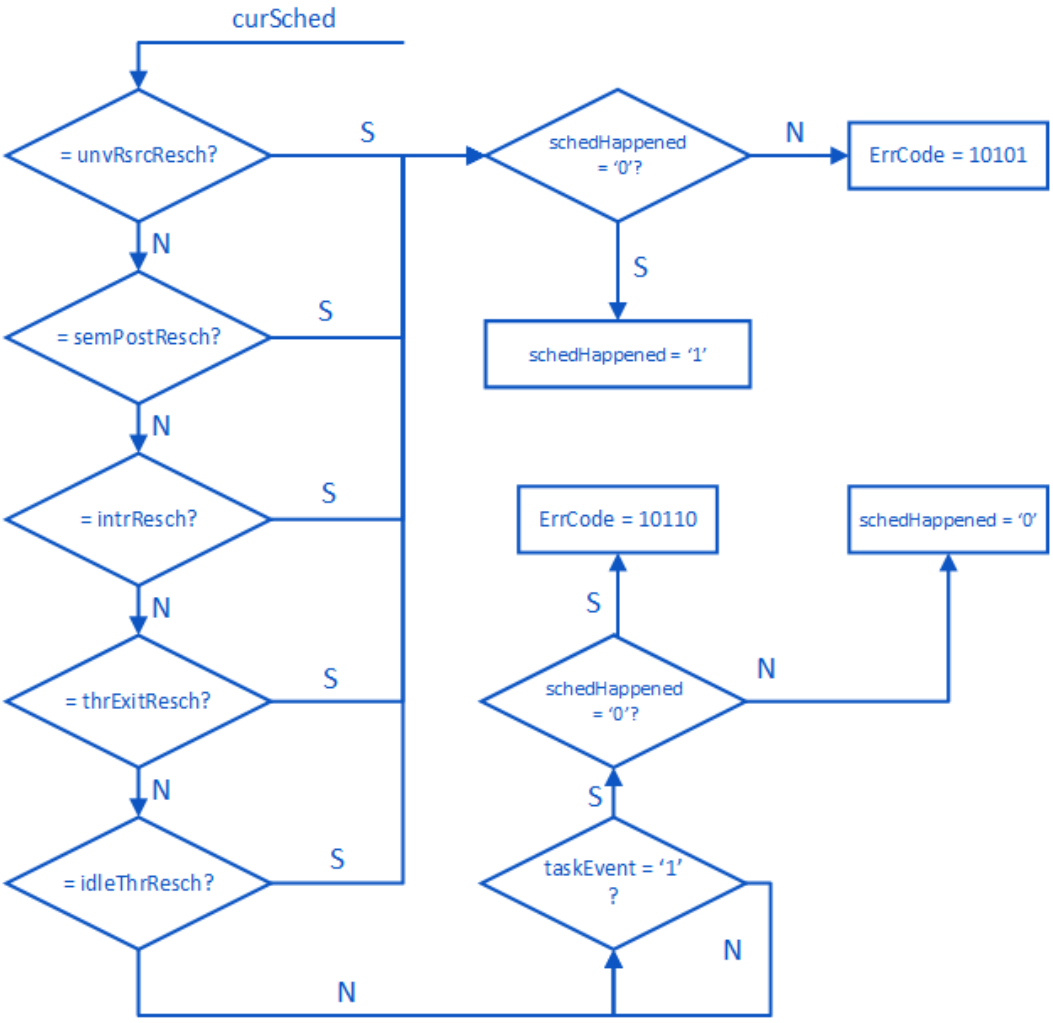


Figura 10.12: Fluxograma da Verificação 5 – Reescalonamento

10.3.6 Verificação 6: Núcleo de execução

Podem-se incluir restrições para as tarefas quanto ao núcleo que deve executá-las. Esta verificação tem a finalidade conferir se a tarefa em execução está respeitando a restrição de núcleo imposta, caso esta exista. Caso esta restrição seja desobedecida esta verificação gera o código de erro *'11001'*. Esta condição é verificada através dos vetores *taskCoreLocked* e *curCoreLocked* de restrição configurados no momento da síntese do RTOS–Watchdog:

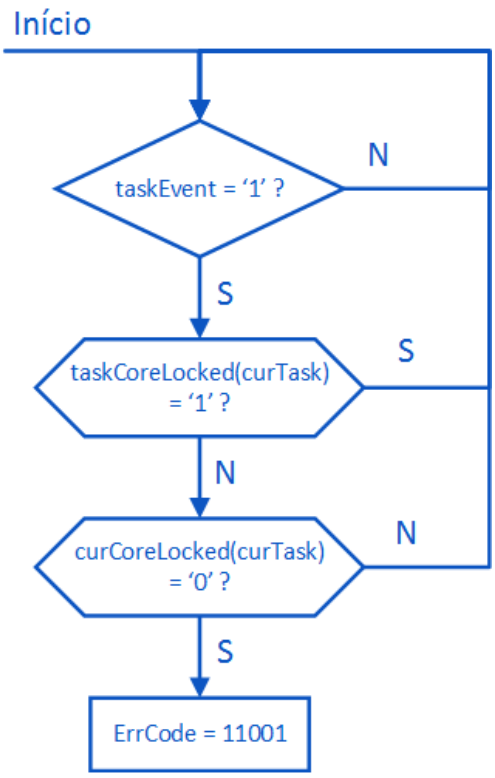


Figura 10.13: Fluxograma da Verificação 6 – Núcleo de execução

10.3.7 Verificação 7: Auto teste do RTOS–WD

Esta verificação tem a finalidade de detectar *overflow* ou *underflow* nos contadores de tarefas e recursos bloqueados. Estes eventos podem acontecer se houver um registro errôneo de algum evento, devido a uma falsa detecção por parte do RTOS–Watchdog. Esta verificação está inclusa nos atuadores destes contadores, Gerenciador de Recursos e Verificação 1, são gerados os seguintes códigos de erro:

- É gerado o código de erro '11101' (ver Figura 10.7) em caso de overflow, se o RTOS-WD detectar o bloqueio de recurso ou tarefas e os contadores já estarem com seus valores máximos (igual ao número de tarefas);
- Ou é gerado o código de erro '11110' (ver Figura 10.7 e Figura 10.8), no caso de *underflow*, onde é detectada uma liberação de recurso ou tarefas e os contadores já estiverem com valor zero.

10.4 Limitações da proposta

O RTOS-WD não é capaz de identificar qual recurso esta sendo liberado ou se a tarefa aguardava especificamente este recurso. O que é verificado é se a tarefa escalo-

nada em função de um semáforo é a de maior prioridade em relação às tarefas prontas. Como consequência não é possível atestar que o recurso liberado é o aguardado de fato pela tarefa escalonada.

Apesar do objetivo de não serem necessárias alterações, duas pequenas alterações se tornaram necessárias. A primeira alteração foi a criação de sinalização de início do RTOS Apêndice H e a segunda um sinal por onde o RTOS sinaliza a tarefa em execução para o RTOS–Watchdog Apêndice I.

O RTOS é capaz de detectar falhas através de *assertions* (ver Apêndice J para exemplo de uso). Se a condição testada for falsa é detectado um erro e o RTOS disponibiliza na saída padrão qual *assertion* o detectou (arquivo e linha correspondente no código-fonte do *kernel*). No *kernel* há trinta e uma *assertions* espalhadas. O RTOS é capaz de detectar estouro de pilha através de *magic-number* (número conhecido escrito no final da pilha ou final de blocos para alocação dinâmica que não deve ser alterado). São verificados ponteiros passados como parâmetros em funções.

Dentre os componentes do RTOS–Watchdog o Monitor de Eventos de Escalonamento (MEE) é o módulo com maior dependência do sistema operacional empregado no sistema, pois depende diretamente do fluxo de funções do escalonador. E eventos presentes conforme seu algoritmo de escalonamento.

A latência de detecção em relação ao momento em que ela ocorreu depende de quantos núcleos há no processador, uma vez que os eventos ocorridos em cada núcleo são processados por fatias de tempo. O tempo de análise para cada núcleo depende dos eventos ocorridos, ocorrência de um evento de tarefa faz com que a análise de um núcleo chegue a até três ciclos de *clock*. A latência de detecção em relação ao RTOS pode variar bastante, pois depende dos pontos onde são geradas as *assertions*, mais especificamente do ponto dentro da função. Em um caso extremo onde há troca de tarefas em todos os núcleos, em um processador de dois núcleos a detecção é realizada em oito ciclos de *clock*. Este caso não foi detectado em nenhuma das simulações, para fins de ilustração deste caso foi criada a Figura 10.14. Os sinais cpuID e fsmState são os mesmos descritos em 11 Validação da proposta. O sinal ocorrência é no instante onde ocorre esta falha e o sinal detecção o instante em que é detectada pelo RTOS–Watchdog.

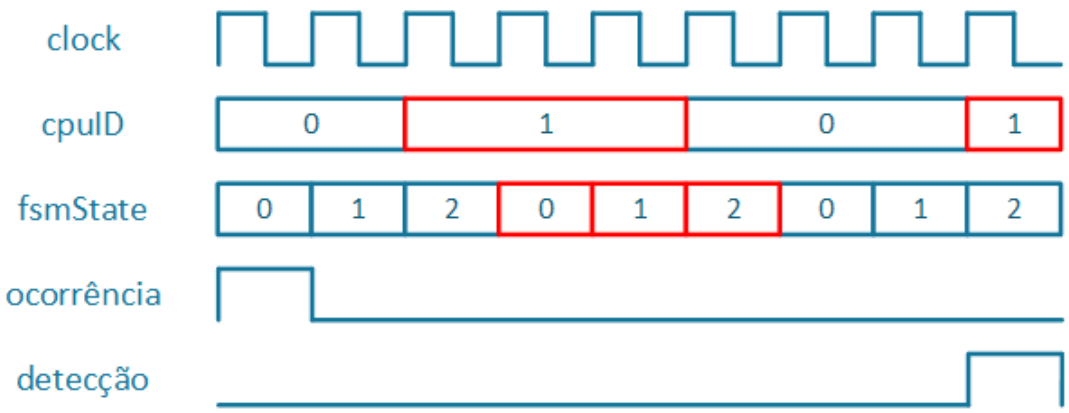


Figura 10.14: Cálculo da latência de detecção RTOS–Watchdog

10.5 Desenvolvimento do Estudo de Caso

O Plasma Multicore será a plataforma utilizada para o desenvolvimento e validação do RTOS–Watchdog. Para os experimentos práticos realizados para a avaliação será utilizada uma plataforma de prototipação própria desenvolvida pelo grupo SiSC.

10.5.1 Plataforma de prototipação

A plataforma foi desenvolvida pelo grupo SiSC [1], é uma placa de circuito impresso de oito camadas com duas Xilinx Virtex 4, cujos *part number* são XC4VFX12-10SF363. Uma FPGA Xilinx Spartan 3 com *part number* XC3S500E-4FTG256. Um processador ARM 7 NXP LPC-2378, capaz de controlar o gerador de EMI conduzida (ver 8.1.2) na fonte de alimentação das FPGAs Virtex 4. Bancos de Memórias tipo SRAM. Interface UART RS-232 e ótica.

Esta plataforma já foi utilizada para o desenvolvimento de diversos experimentos que resultaram em publicações [29] [30] [31].

11 Validação da proposta

A validação da proposta foi realizada através de simulação. Foi utilizado o simulador ModelSim™ desenvolvido pela Mentor Graphics®. A validação foi realizada com o Plasma utilizando até quatro núcleos, porém há a possibilidade da utilização de mais núcleos. A interrupção externa foi simulada com um sinal periódico com a metade da frequência do *tick*. Nas simulações a seguir é demonstrado o funcionamento do RTOS-WD, com o Plasma utilizando dois núcleos a mesma condição que será utilizada nos experimentos. Pois não há a disposição FPGAs que suportem o Plasma com mais núcleos.

Será demonstrado inicialmente o comportamento da FSM de controle, a seguir serão injetadas algumas falhas no *kernel* do RTOS, então será observado se estas serão detectadas pelo RTOS-WD como o esperado.

Na Figura 11.1 são observados três sinais:

- O sinal *tasks_o* é a tarefa que o sistema operacional esta executando em cada núcleo. (0) corresponde ao núcleo ‘0’ e (1) ao núcleo ‘1’.
- O sinal *cpuID* este indica a CPU que está sendo processada no ciclo corrente da máquina de estados, este é alterado no início do segundo estágio, porém a CPU seguinte somente será analisada no ciclo seguinte (estado 0).
- O sinal *fsmState* é o estado atual da máquina de estados da unidade de controle.

Observa-se que a duração dos ciclos da máquina de estados (tempo decorrido entre os estados ‘fsmstate = 0’ na Figura 11.1). Este tempo está compreendido entre 2 e 3 ciclos de *clock*.

Quando há ocorrência de Evento de Tarefa *(Task Event)* e se não for detectado erro pela Verificação 1, ocorre o caso crítico. Este caso está demonstrado entre cursores.

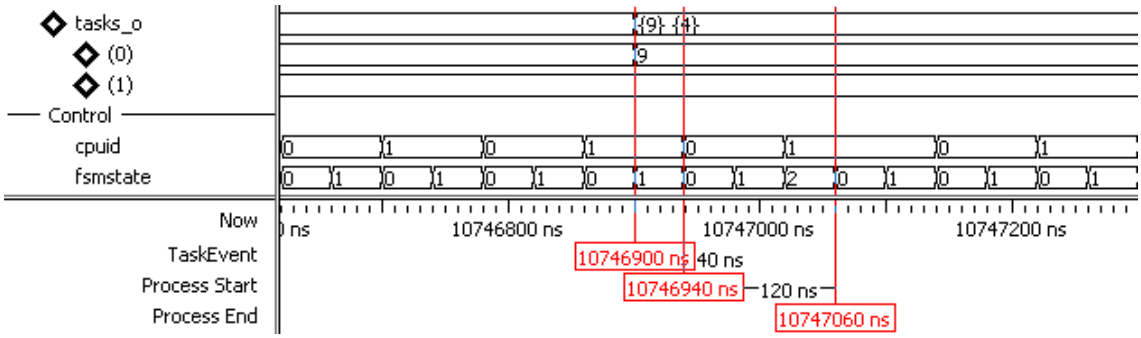


Figura 11.1: Simulação da FSM do controle (caso crítico).

A seguir, serão observados os efeitos da injeção de falhas no RTOS e o comportamento do RTOS-WD para sua validação. Estas ‘falhas’ são injetadas por software e utilizam métodos ‘legais’ para troca de prioridade ou núcleo de execução para as tarefas, ou faz uso de replicação de função de escalonamento, e são injetadas através de GPIO. Porém o RTOS-WD não é reconfigurado então há a detecção de erros.

Será destacado o comportamento do sistema sem falhas e após a injeção de falhas. O benchmark (12.2.1 Benchmark I) será o mesmo em todas as simulações, para a validação da técnica proposta.

Na Figura 11.2 é simulado o efeito de um erro de escalonamento trocando-se a prioridade da Tarefa ‘8’.

A Tarefa ‘7’ libera um recurso, o qual é requisitado pela Tarefa ‘8’. Como a Tarefa ‘8’ possui menor prioridade esta deve aguardar a Tarefa ‘7’ ser interrompida. Após a injeção da ‘falha’ a prioridade da Tarefa ‘8’ é incrementada ficando maior que a da Tarefa ‘7’, resultando na interrupção desta antes da mesma ser bloqueada. A Verificação ‘2’ que é responsável por conferir inversões de prioridade detecta este erro como o esperado.

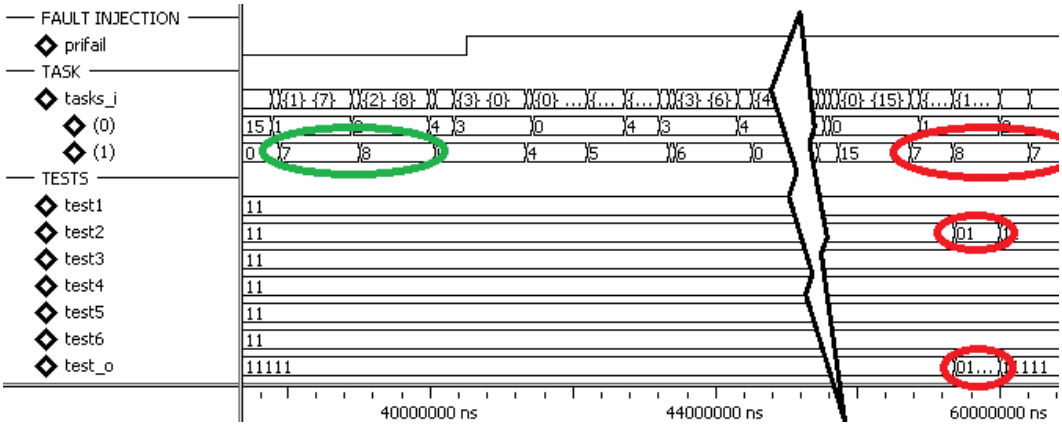


Figura 11.2: Simulação FI: Inversão de prioridade

Na Figura 11.3, é demonstrado como é detectado o não atendimento do *tick*. Este evento ocorre em ambas as bordas do sinal de *tick* (*tick_i*). A injeção da falha é controlada pelo sinal '*test3fi*'. No benchmark utilizado nesta simulação quando há um evento de *tick* geraria normalmente um escalonamento da tarefa '5', conforme destacado em verde. Percebe-se que o mesmo não ocorre no *tick* seguinte devido ao não atendimento do *tick*. Finalmente observa-se que o IP detectou esta falha por *timeout*. A falha foi injetada no *kernel*, modificando se o serviço de interrupção (OS_InterruptServiceRoutine) para não chamar a função de atendimento do *tick*.

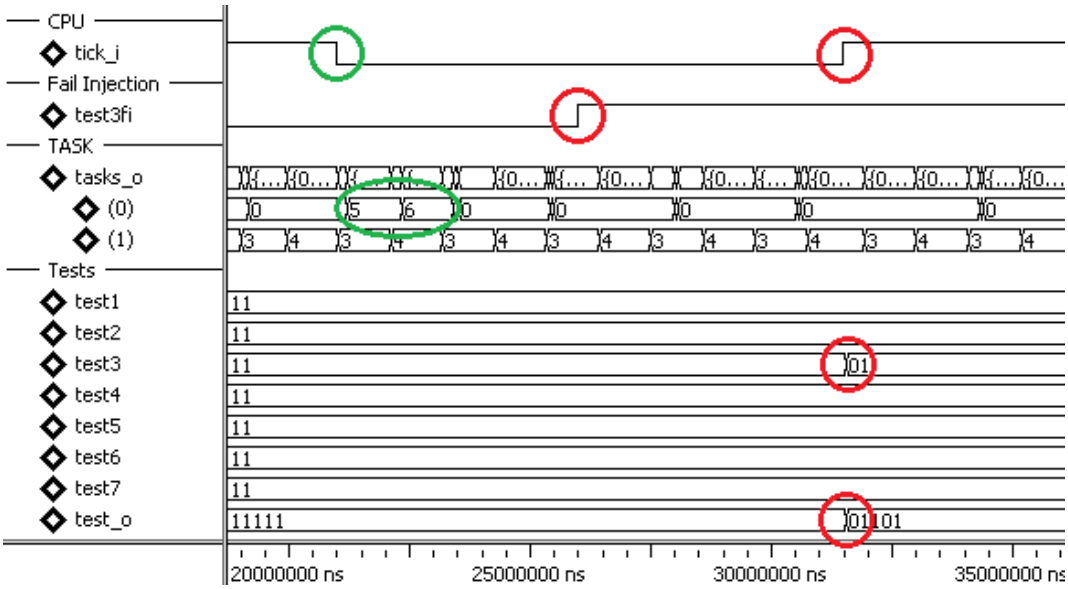


Figura 11.3: Simulação FI: Interrupção de Tick não atendida

A Verificação '4' (Figura 11.4) é similar à Verificação '3', porém ao invés de monitorar a execução da função do *tick* (OS_ThreadTick) no *kernel* do RTOS, nesta verificação é monitorado se a tarefa de interrupção (15) foi executada. A interrupção externa é acionada em ambas as bordas do sinal *interrupt_i*, que por sua vez é o pino 31 do GPIOA do Plasma. Em verde é demonstrado o evento de interrupção corretamente atendido. Em vermelho observa-se que o evento não foi atendido, e a detecção de erro pelo IP por *timeout*, na verificação '4' conforme esperado. A falha foi injetada no *kernel*, modificando se o serviço de interrupção (OS_InterruptServiceRoutine) para não chamar a tarefa de interrupção.

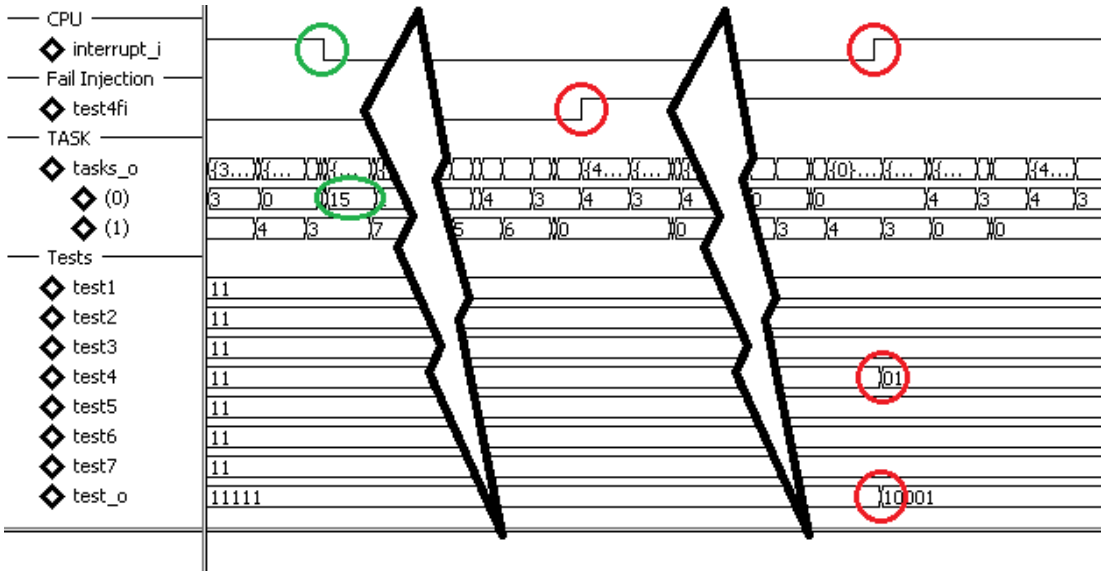


Figura 11.4: Simulação FI: Interrupção Externa não atendida

A Verificação ‘5’ confere se há eventos de reescalonamentos não atendidos ou reescalonamento de tarefa inesperado. A simulação de falha foi realizada com a inserção de outra função, *longjmp*, em um endereço não monitorado pelo RTOS-WD. Esta falha é acionada quando a tarefa ‘4’ é escalonada. Assim o evento de escalonamento não é detectado pelo RTOS-WD, porém o escalonamento de tarefas é realizado. Em destaque na Figura 11.5 estão os momentos em que a falha começou a ser injetada e uma ocorrência de erro devido a esta falha. Conforme esperado a Verificação ‘5’ detectou este erro.

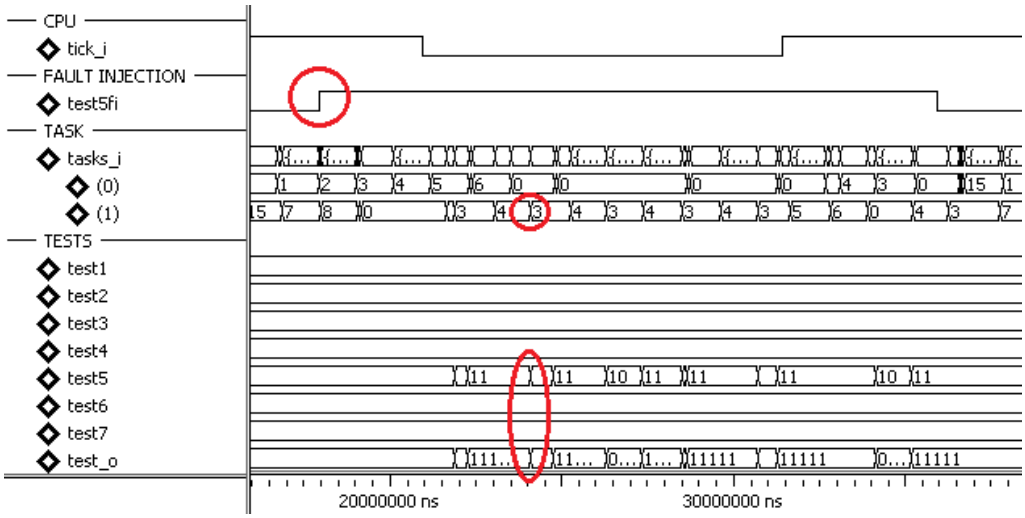


Figura 11.5: Simulação FI: Escalonamento de tarefas sem evento

Na Figura 11.6 é injetada uma falha que fará com que a tarefa ‘2’, que deve ser executada exclusivamente no núcleo ‘0’ passe a ser executada pelo núcleo ‘1’. A falha foi injetada trocando a restrição da tarefa ‘2’ para o outro núcleo.

Conforme esperado a Verificação ‘6’ que é responsável pela verificação de núcleo a detectou conforme destacado em vermelho.

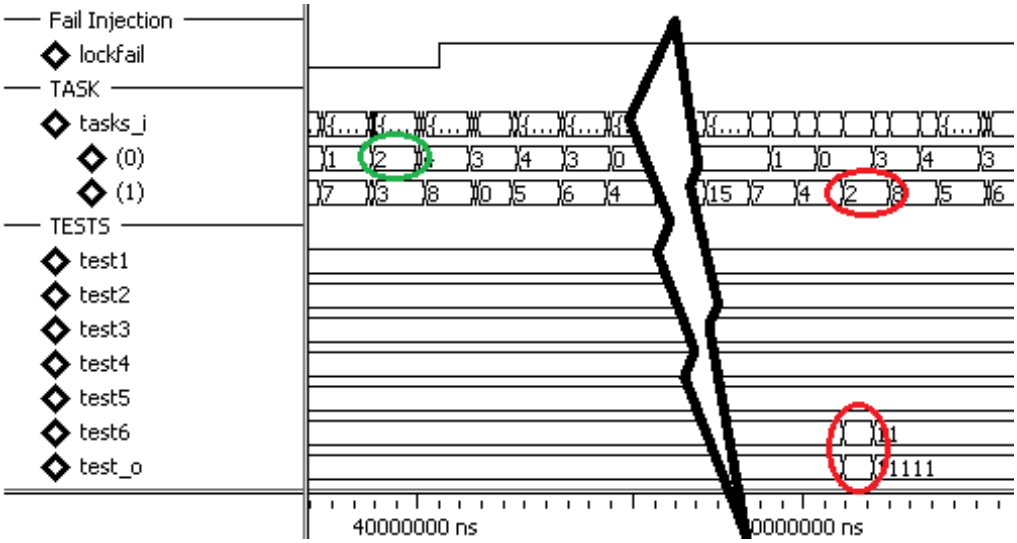


Figura 11.6: Simulação FI: Bloqueio de núcleo desrespeitado

PARTE IV – RESULTADOS E CONCLUSÕES

12 Avaliação Experimental

Para a avaliação experimental, o sistema será emulado em FPGA, e será submetido à Injeção de Ruído Eletromagnético tipo conduzido, conforme IEC 61000-4-29 [11] o método escolhido para a injeção do ruído é o Queda de Tensão.

Serão realizados 1000 experimentos com os benchmarks desenvolvidos. Os dados serão capturados através do ChipScope™ Pro componente da Xilinx® ISE Design Suite.

O ChipScope™ Pro insere analisador lógico, diretamente no projeto. Permitindo a visualização de quaisquer sinais internos ou nós do sistema projetado Os sinais são capturados na velocidade de operação do sistema através da interface de programação [32].

Neste caso o sinal capturado pelo ChipScope™ Pro será código de erro gerado pelo RTOS–WD.

Outra ferramenta irá capturar a saída padrão do RTOS, conectada a uma interface serial do Host (Figura 12.2).

Uma característica deste tipo de experimento é que não há controle sobre as falhas injetadas nem em respeito à quantidade nem localização, ou seja, não é determinístico. Porém as falhas são mais próximas às falhas reais as quais o sistema está sujeito em campo. O único parâmetro disponível são as falhas observadas. Para futuro entendimento sobre a nomenclatura utilizada no restante deste trabalho, na Figura 12.1 é demonstrada a relação entre as falhas injetadas, não detectadas e detectadas e observadas.

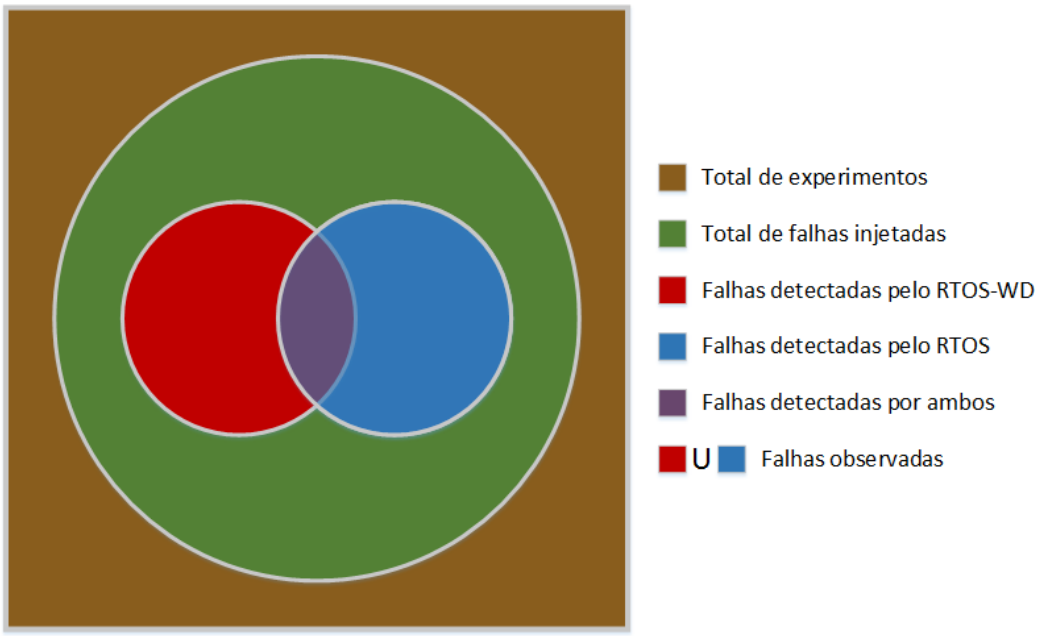


Figura 12.1: Diagrama Venn – Relação entre falhas

12.1.1 Procedimento de Injeção de Falhas

A FPGA utilizada nos experimentos já possui uma dose de radiação TID de 51,9 krads em acordo com o método 1019.8 para TID Test Procedure do standard MIL-STD-883H. [33]

Será aplicado ao circuito um distúrbio tipo Queda de tensão (Voltage Dip), conforme explicado na seção 9 Norma IEC 61.000-4-29. A injeção de ruído será efetuada na linha de alimentação do *core* da FPGA 1 (vide Figura 12.2) na qual o sistema está em execução.

Durante o experimento a saída padrão do RTOS, pela conexão UART entre o Plasma e o Host, será capturada para análise posterior. O estado do RTOS–WD será capturado através da ferramenta ChipScope™ Pro.

Para ler dados confiáveis o ChipScope™ Pro será sintetizado na FPGA 2 (vide Figura 12.2) isolada do ruído. A transmissão do estado do RTOS–WD será realizada pela conexão JTAG entre a FPGA 2 e o Host. Para a sincronia entre o ChipScope™ Pro e o SoC o *clock* será gerado na FPGA 2 e compartilhado com a FPGA 1.

A FPGA 1 será configurada com o *bitstream* do Plasma com o RTOS-WD, será então inicializado o RTOS, após um intervalo de tempo aleatório será injetado o ruído conduzido, por período também aleatório.

Após os experimentos os arquivos gerados serão processados através de um analisador desenvolvido para este proposito.

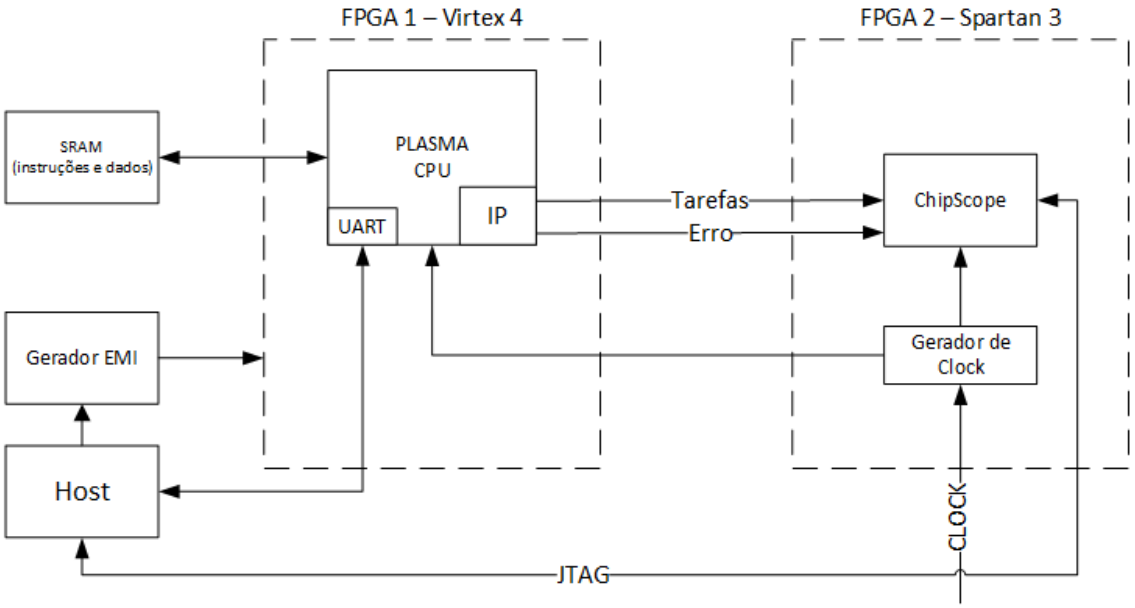


Figura 12.2: Setup de injeção de falhas

12.2 Benchmarks

Foram desenvolvidos três benchmarks para a avaliação do RTOS-Watchdog. Tendo como alvo sistema com dois núcleos. Estes benchmarks têm como objetivo exercitar o escalonador de tarefas do sistema operacional, fazendo uso de semáforos, filas de mensagens e *mutexes*.

Para os diagramas dos benchmarks deve-se considerar “Int” a interrupção externa, “Tx” são as tarefas. Tarefas restritas à um determinado núcleo, encontram-se dentro de uma caixa com a identificação no núcleo explicita em seu topo. Fora destas áreas estão as tarefas sem restrições cabendo ao RTOS selecionar um núcleo ocioso para seu escalonamento.

12.2.1 Benchmark I

O Benchmark I possui oito tarefas, uma interrupção externa e quatro ‘recursos’ que são variáveis globais, utiliza semáforos para controle de fluxo. São utilizados semáforos para o controle de fluxo. As tarefas ‘1’ e ‘7’ dependem da interrupção externa para serem liberadas. Na Tabela 12.1 é demonstrada a organização das tarefas do benchmark I.

Tabela 12.1: Organização das tarefas Benchmark I

Tarefa	**Núcleo**	**Prioridade**
1	0	4
2		3
3	SEM RESTRIÇÕES	2
4		1
5		2
6		1
7	1	4
8		3

Eventos de escalonamento presentes no Benchmark I

- unvRsrcResch
- freeRsrc
- semPostResch
- tickEvent
- intrResch
- rsrcTimeOut
- idleThrResch

Na Figura 12.3 está demonstrado o diagrama do Benchmark I. No Apêndice E encontra-se o código-fonte escrito para as tarefas deste benchmark.

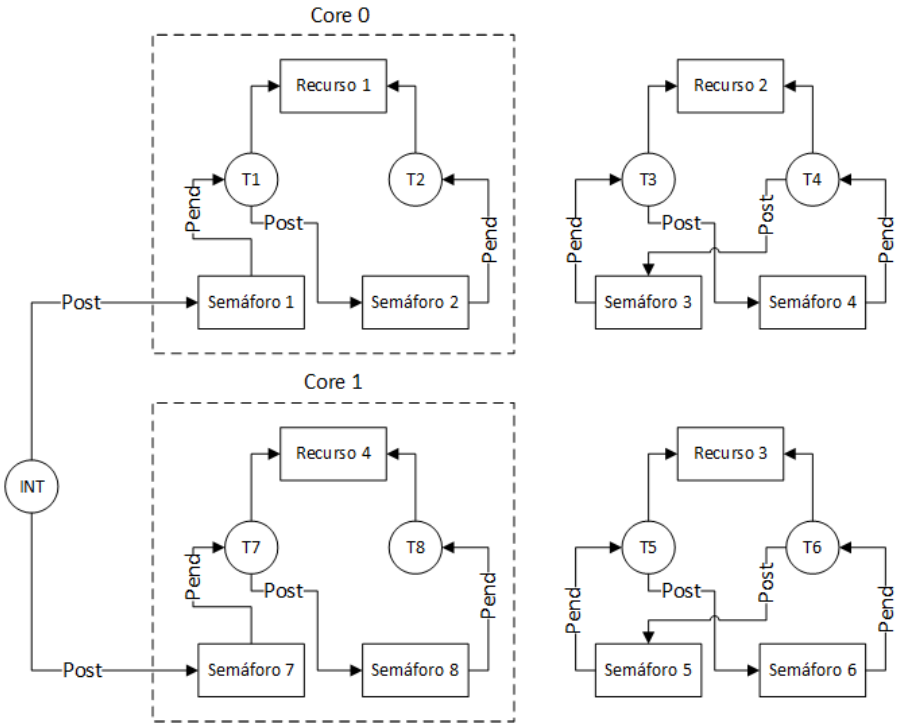


Figura 12.3: Esquema Benchmark I

12.2.2 Benchmark II

O benchmark II possui sete tarefas e uma interrupção externa, faz uso de semáforos, filas de mensagens e *mutexes*. A tarefa ‘1’ realiza um processamento numa variável local e transmite a tarefa ‘2’ que realiza outro processamento em cima deste valor. As tarefas ‘3’ e ‘4’ concorrem por dois recursos protegidos por *mutex*. Finalmente as tarefas ‘5’ e ‘6’ se alternam devido ao semáforo. A organização das tarefas do benchmark II seguem na Tabela 12.2.

Tabela 12.2: Organização das tarefas Benchmark II

Tarefa	**Núcleo**	**Prioridade**
1	0	5
2		4
3	SEM RESTRIÇÕES	3
4		
5	1	2
6		1

Eventos de escalonamento presentes no benchmark II

- unvRsrcResch:
- freeRsrc
- semPostResch
- tickEvent
- intrResch
- rsrcTimeOut
- idleThrResch

Segue na Figura 12.4 o diagrama do benchmark II. No Apêndice F encontra-se o código-fonte escrito para as tarefas deste benchmark.

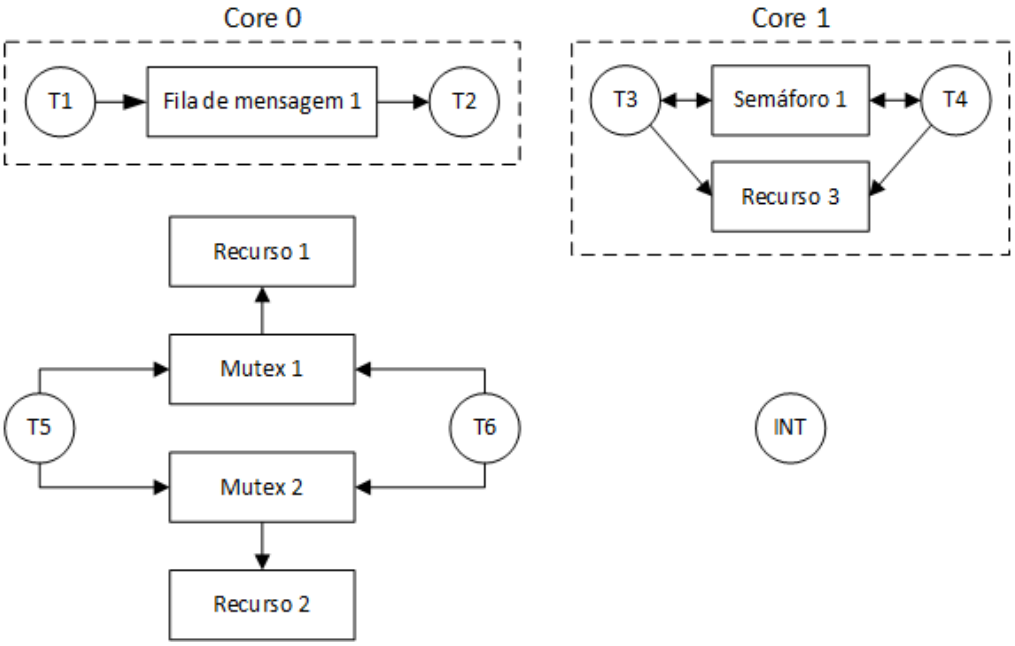


Figura 12.4: Esquema Benchmark II

12.2.3 Benchmark III

O Benchmark III possui sete tarefas e uma interrupção externa, faz uso de semáforos e filas de mensagens. A tarefa ‘1’ realiza um processamento numa variável local e transmite a tarefa ‘2’ que realiza outro processamento em cima deste valor, e o retorna para a tarefa ‘1’. A tarefa ‘1’ ao receber o novo valor da tarefa ‘2’ realiza um segundo processamento e envia o novo valor a tarefa ‘3’. As tarefas ‘4’ e ‘5’ compartilham um recurso com o auxilio de dois semáforos para garantir a exclusão mútua assim como a sequencia no acesso. Finalmente as tarefas ‘6’ e ‘7’ acessam um segundo recurso, porém o semáforo tem a finalidade de exclusão mútua somente, a tarefas ‘6’ é encerrada após 3000 *ticks* (aproximadamente 30s) e ‘7’ é encerrada após 4500 *ticks* (aproximadamente 45s). Neste benchmark todas as tarefas possuem restrição de núcleo para execução, conforme a Tabela 12.3.

Tabela 12.3: Organização das tarefas Benchmark III

Tarefa	**Núcleo**	**Prioridade**
1	0	3
2	1	
3	1	

Tarefa	Núcleo	Prioridade
4	0	1
5	1	
6	0	2
7	1	

Eventos de escalonamento presentes no benchmark III

- unvRsrcResch:
- freeRsrc
- semPostResch
- tickEvent
- intrResch
- rsrcTimeOut
- idleThrResch
- thrExitResch

A Figura 12.5 demonstra o diagrama do Benchmark III. No Apêndice G encontra-se o código-fonte escrito para as tarefas deste benchmark.

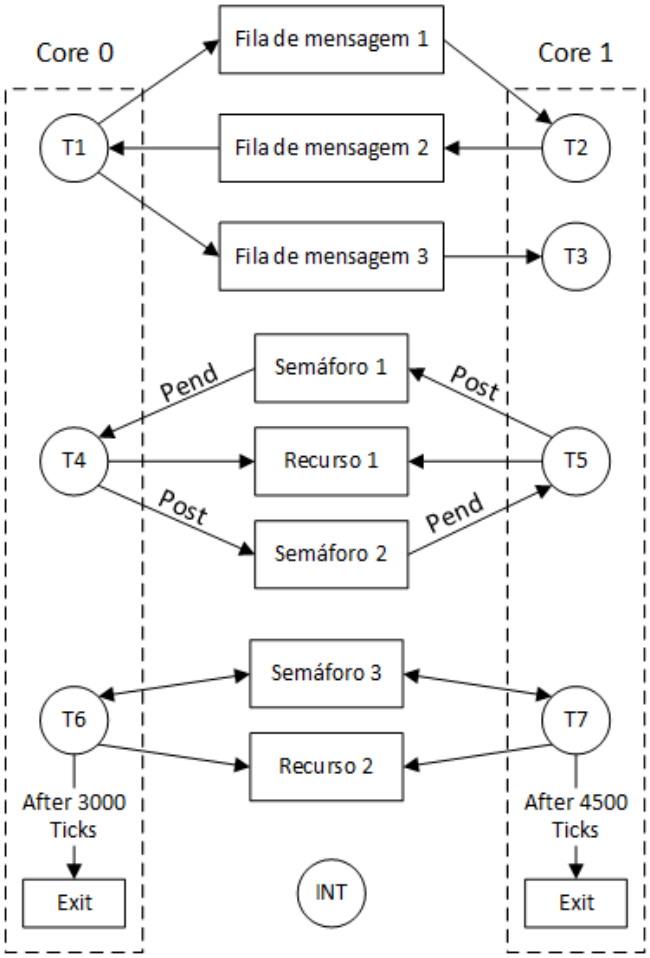


Figura 12.5: Esquema Benchmark III

12.3 Resultados Experimentais

Os resultados foram obtidos a partir da analise da saída padrão do RTOS capturada durante a injeção do ruído e a captura realizada pelo ChipScope™ Pro com o estado do RTOS–WD conforme visto em 12.1.1 Procedimento de Injeção de Falhas.

Nos gráficos RTOS Assertions, estas estão com a legenda no formato $\langle arquivo \rangle : \langle linha \rangle$, por exemplo: rtos.c:429, o objetivo é somente mostrar a distribuição das detecções pelo RTOS. Pode haver, entretanto mais de uma *assertion* e/ou mais que um código de falha sinalizado pelo RTOS–WD, porém somente os primeiros são contabilizados.

Nos gráficos RTOS–WD Detections as legendas são os códigos de falhas (vide 10.3 Verificações do RTOS) detectadas pelo RTOS–WD.

TODOS os valores porcentuais estão calculados sobre o número de experimentos (consultar Figura 12.1 para entendimento) independente se houve detecção de falha ou não.

12.3.1 Benchmark I

Neste benchmark o RTOS realizou *assertions* em três pontos distintos. Distribuídos conforme a Figura 12.6, em 47,25 % dos experimentos.

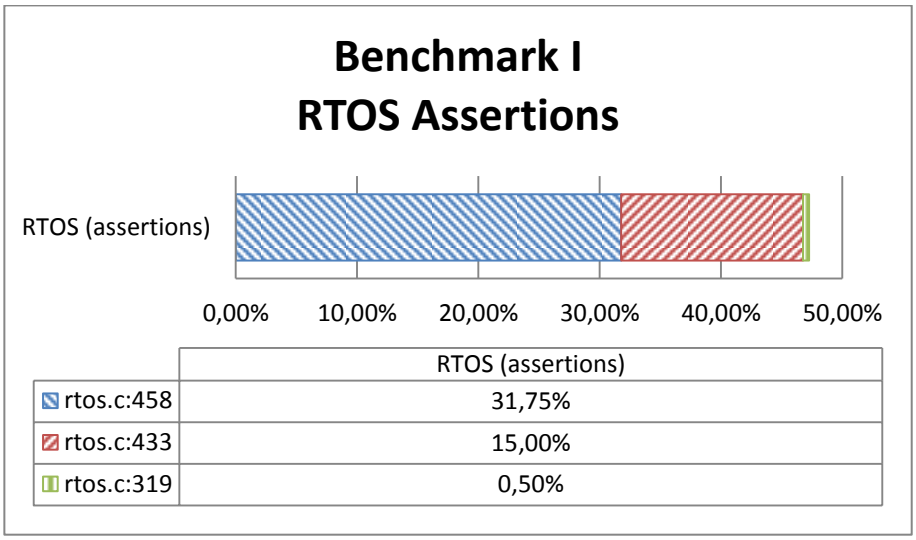
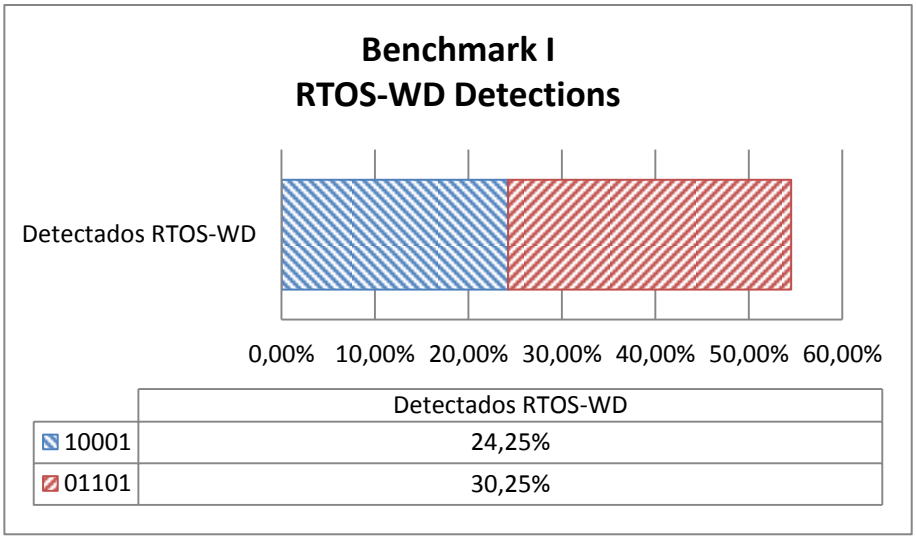


Figura 12.6: Benchmark I – RTOS Assertions

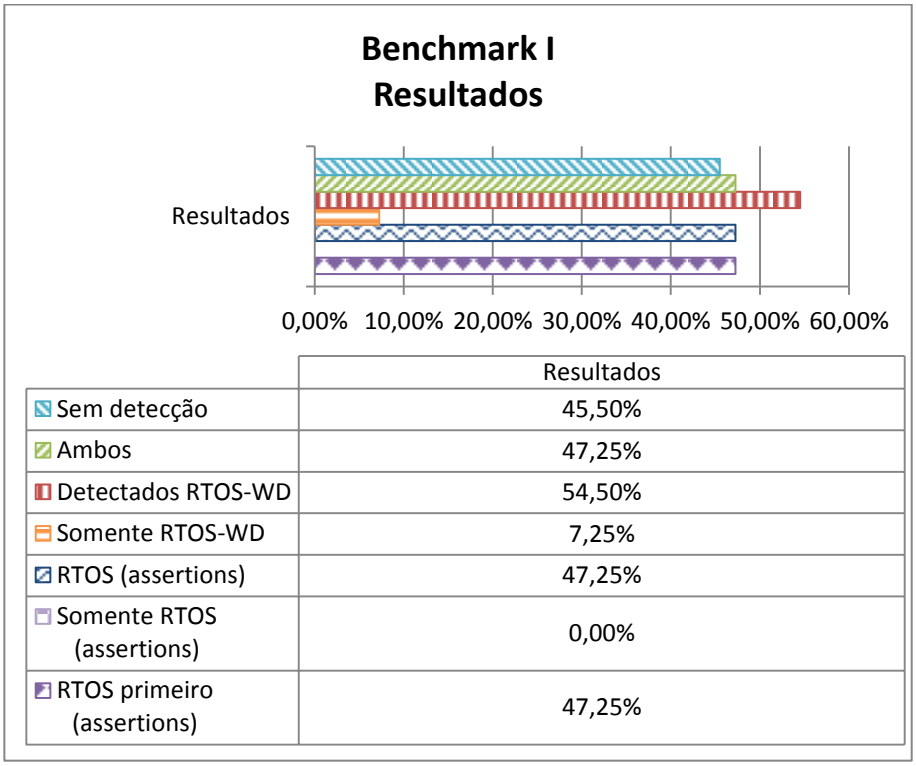
Na Figura 12.7 está demonstrada a distribuição de falhas detectadas pelo RTOS–WD. Foram detectadas falhas em 54,50 % dos experimentos pelo RTOS–WD. As falhas detectadas são referentes a *timeouts* para o atendimento de interrupções externas (10001) e de *ticks* (*01101*), pode ser que estas falhas sejam provenientes somente de um atraso no atendimento de algumas destas interrupções ou devido ao sistema ter entrado em estado de *deadlock*.



	Detectados RTOS-WD
10001	24,25%
01101	30,25%

Figura 12.7: Benchmark I – Detecções RTOS–WD

Nos resultados finais para o Benchmark I (Figura 12.8) observa-se que em 45,50 % dos experimentos não houve detecção de falhas. Das falhas observadas 100,00 % foram detectadas pelo RTOS–WD enquanto apenas 47,25 % foram detectadas pelo RTOS.

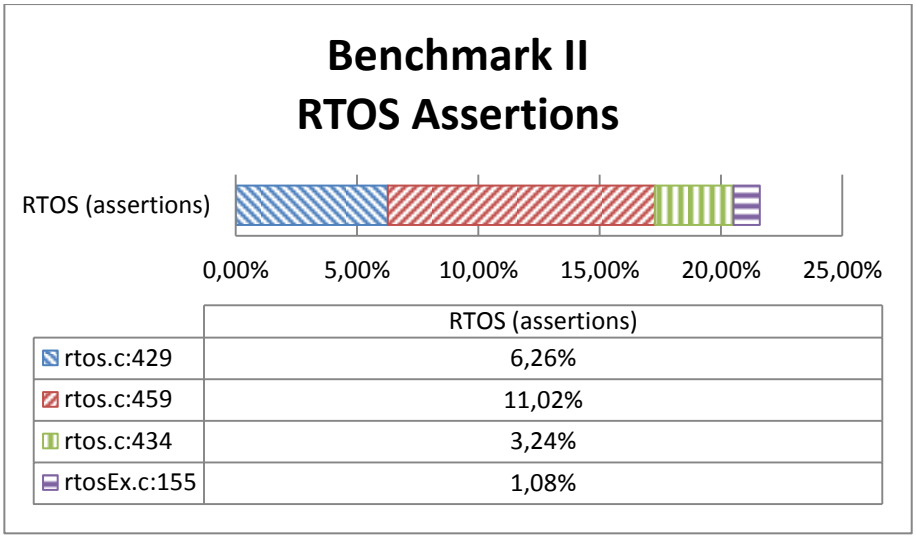


	Resultados
Sem detecção	45,50%
Ambos	47,25%
Detectados RTOS-WD	54,50%
Somente RTOS-WD	7,25%
RTOS (assertions)	47,25%
Somente RTOS (assertions)	0,00%
RTOS primeiro (assertions)	47,25%

Figura 12.8: Benchmark I – Resultados

12.3.2 Benchmark II

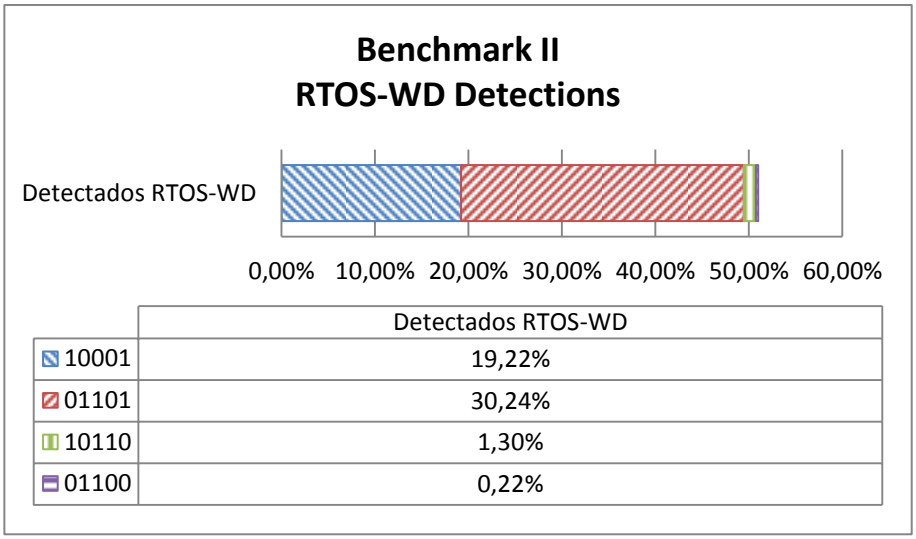
Neste benchmark o RTOS realizou *assertions* em quatro pontos distintos. Distribuídos conforme a Figura 12.9, em 21,60 % dos experimentos.



	RTOS (assertions)
rtos.c:429	6,26%
rtos.c:459	11,02%
rtos.c:434	3,24%
rtosEx.c:155	1,08%

Figura 12.9: Benchmark II – RTOS Assertions

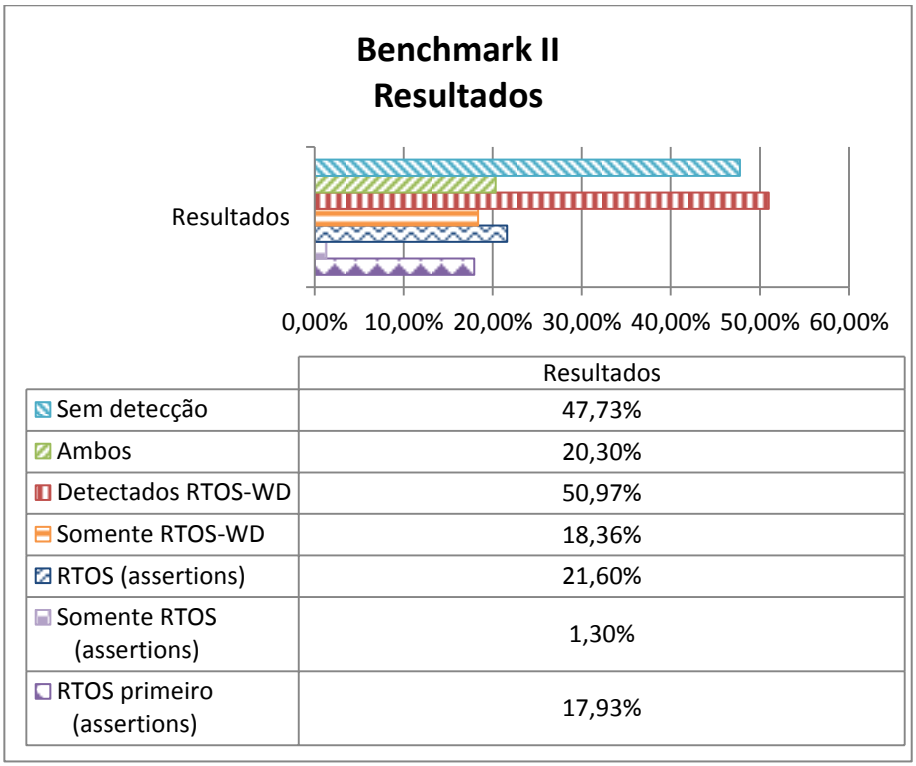
Na Figura 12.10 está demonstrada a distribuição de falhas detectadas pelo RTOS–WD. Foram detectadas falhas em 50,97 % dos experimentos pelo RTOS–WD. A maioria das falhas detectadas é referente a *timeouts* para o atendimento de interrupções externas (10001) e de *ticks* (*01101*), pode ser que estas falhas sejam provenientes somente de um atraso no atendimento de algumas destas interrupções ou devido ao sistema ter entrado em estado de *deadlock*. O código de falhas 01100 significa que o arbitro de interrupção não gerou o sinal para um dos núcleos para este atender a interrupção de *tick*.



	Detectados RTOS-WD
10001	19,22%
01101	30,24%
10110	1,30%
01100	0,22%

Figura 12.10: Benchmark II – Detecções RTOS–WD

Nos resultados finais para o Benchmark II (Figura 12.11) observa-se que em 47,73 % dos experimentos não houve detecção de falhas. Das falhas observadas 97,52 % foram detectadas pelo RTOS–WD enquanto apenas 41,32 % foram detectadas pelo RTOS.



	Resultados
Sem detecção	47,73%
Ambos	20,30%
Detectados RTOS-WD	50,97%
Somente RTOS-WD	18,36%
RTOS (assertions)	21,60%
Somente RTOS (assertions)	1,30%
RTOS primeiro (assertions)	17,93%

Figura 12.11: Benchmark II – Resultados

12.3.3 Benchmark III

Neste Benchmark o RTOS realizou *assertions* em dois pontos distintos. Distribuídos conforme a Figura 12.12, em 53,66 % dos experimentos.

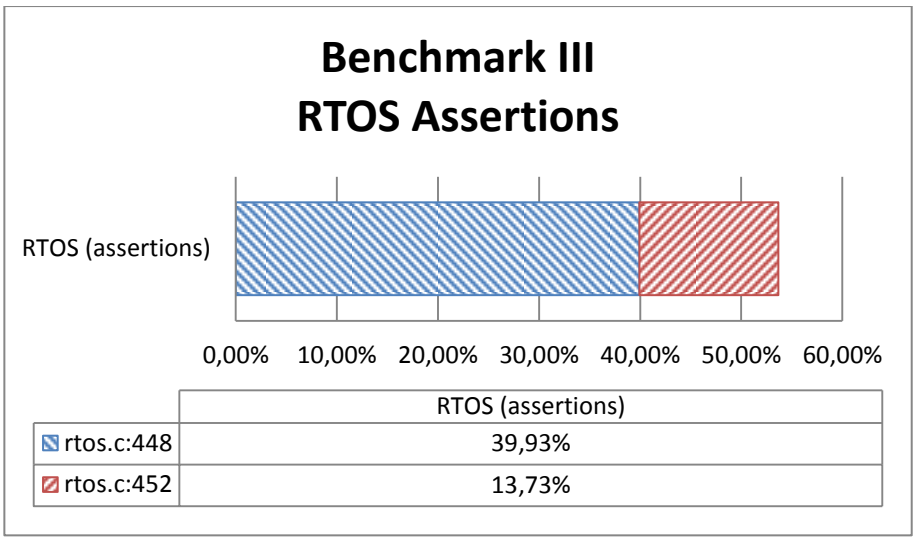
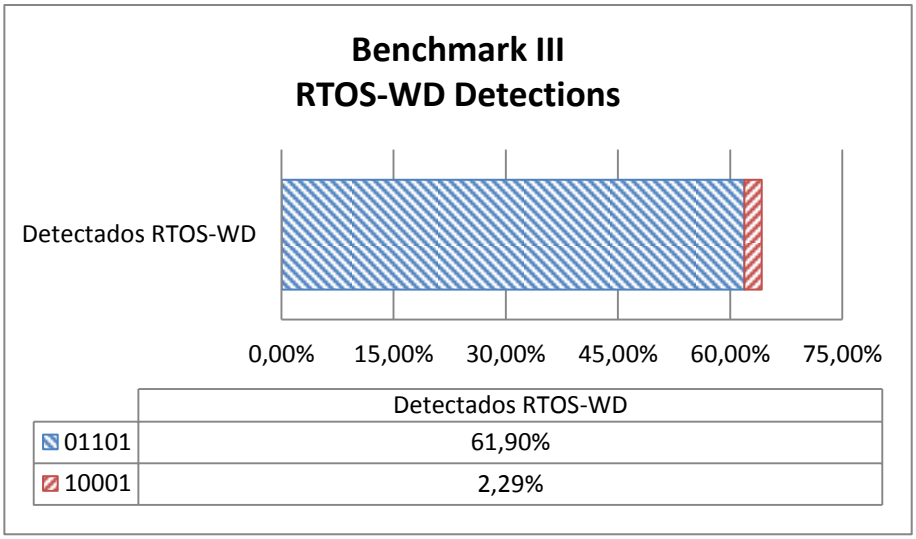


Figura 12.12: Benchmark III – RTOS Assertions

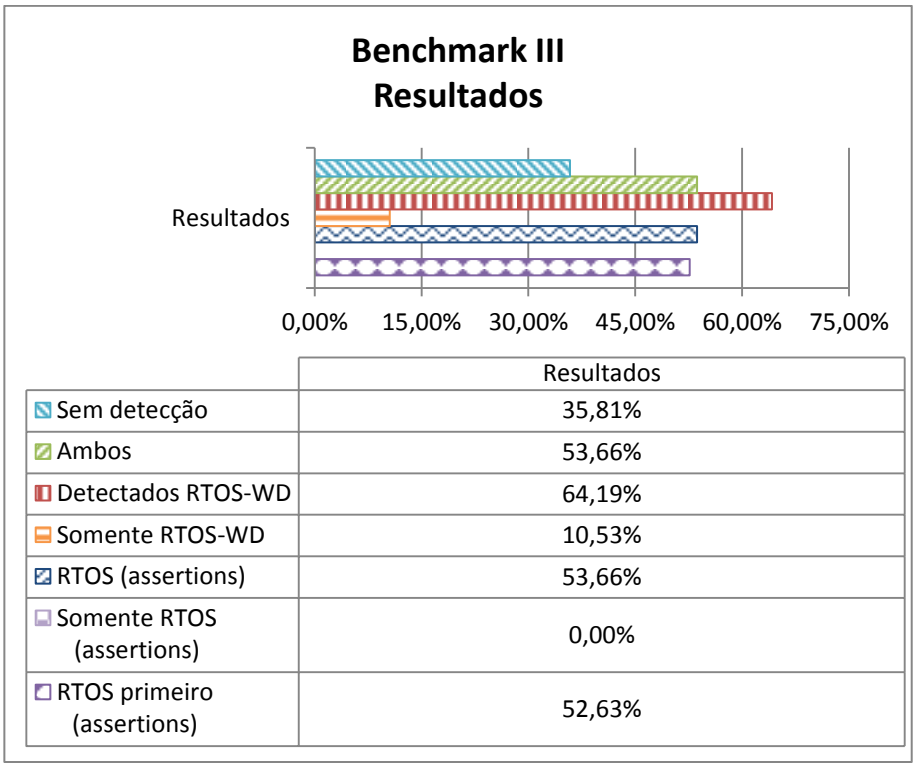
Na Figura 12.13 está demonstrada a distribuição de falhas detectadas pelo RTOS–WD. Foram detectadas falhas em 64,19 % dos experimentos pelo RTOS–WD. A maioria das falhas detectadas é referente a *timeouts* para o atendimento de interrupções externas (10001) e de *ticks* (*01101*), pode ser que estas falhas sejam provenientes somente de um atraso no atendimento de algumas destas interrupções ou devido ao sistema ter entrado em estado de *deadlock*.



	Detectados RTOS-WD
01101	61,90%
10001	2,29%

Figura 12.13: Benchmark III – Detecções RTOS–WD

Nos resultados finais para o Benchmark III (Figura 12.14) observa-se que em 35,81 % dos experimentos não houve detecção de falhas. Das falhas observadas 100,00 % foram detectadas pelo RTOS–WD enquanto 83,60 % foram detectadas pelo RTOS.



	Resultados
Sem detecção	35,81%
Ambos	53,66%
Detectados RTOS-WD	64,19%
Somente RTOS-WD	10,53%
RTOS (assertions)	53,66%
Somente RTOS (assertions)	0,00%
RTOS primeiro (assertions)	52,63%

Figura 12.14: Benchmark III – Resultados

13 Análise de overheads

Vários fatores influenciam no tamanho final do RTOS–WD como:

- O número de núcleos do processador: Pois são necessários MEE, vetores com os *locks* das tarefas para cada núcleo além de ajustes em alguns registradores no controle (realizados automaticamente no momento da síntese).
- O número de tarefas: Uma vez deve haver uma posição de memória para cada tarefa com a sua prioridade, um bit em dois registradores para o monitoramento de seu estado. A largura dos vetores para a representação da tarefa e etc. A largura da representação da tarefa é calculada pelo *parser* desenvolvido para adquirir os endereços das funções de escalonamento monitoradas pelo MEE (10.1.2 Monitor de Eventos de Escalonamento). A largura deste sinal deve representar todas as tarefas univocamente conforme os critérios descritos em 10.1.1 Monitoramento de Tarefa.
- Níveis de prioridades: A largura da memória de prioridades pode ser facilmente ajustável conforme a necessidade.
- Os contadores para os *timeouts* de interrupções e para a persistência do código de erro também influenciam.

A seguir serão verificados os *overheads* para os Benchmarks I, II e III utilizados até então. O *overhead* será obtido através do relatório de Nível de Utilização do Módulo (Module Level Utilization, Apêndice B, Apêndice C e Apêndice D) fornecido pela ferramenta de síntese do ISE. Lendo estes relatórios nos apêndices acima citados constata-se que em valores médios o RTOS–Watchdog ocupou 463 *slices*, a Unidade de Controle ocupou em valores médios 187 *slices* e os MEEs 138 *slices* cada um.

Lembrando que a *Idle Thread* (5.2.1 Tarefas) deve ser representada com prioridade zero. Para todos os casos são utilizados somente processadores com dois núcleos.

São analisados os overheads em *Slices*, *Slice Regs*, e LUTs. Os outros itens descritos no relatório e omitidos nas análises à seguir não registraram *overhead* em virtude da inclusão do RTOS–Watchdog no microprocessador.

13.1.1 Benchmark I

Nesse benchmark há oito tarefas e as prioridades atribuídas às tarefas podem ser representadas em cinco níveis, logo a memória de prioridades possui nove posições (oito tarefas + *idle thread*) e três bits de largura. Os *timouts* para a interrupção externa e *tick* estão ajustados para 4095 ciclos resultando em 12 *bits* cada. A persistência do código de erro é de 1000 ciclos, ou seja, 10 *bits*. A Tabela 13.1 foi concebida com base nos dados do relatório no Apêndice B.

Tabela 13.1: Benchmark I - Relatório de Overhead

Módulo	Slices	Slice Reg	LUTs
RTOS–WD	466	193	810
Total	5487	1381	8527
Overhead	8,49 %	13,98 %	9,50 %

13.1.2 Benchmark II

Neste benchmark há seis tarefas e as prioridades atribuídas às tarefas podem ser representadas em seis níveis, logo a memória de prioridades possui sete posições (seis tarefas + *idle thread*) e três bits de largura. Os *timeouts* para a interrupção externa e *tick* estão ajustados para 6144 ciclos resultando em 13 *bits* cada. A persistência do código de erro é de 1000 ciclos, ou seja, 10 *bits*. A Tabela 13.2 foi concebida com base nos dados do relatório no Apêndice C.

Tabela 13.2: Benchmark II - Relatório de Overhead

Módulo	Slices	Slice Reg	LUTs
RTOS–WD	450	186	774
Total	5452	1357	8583
Overhead	8,25 %	13,71 %	9,02 %

13.1.3 Benchmark III

Neste benchmark há sete tarefas e as prioridades atribuídas às tarefas podem ser representadas em quatro níveis, logo a memória de prioridades possui oito posições

(sete tarefas + *idle thread*) e dois bits de largura. Os *timeouts* para a interrupção externa e *tick* estão ajustados para 6144 ciclos resultando em 13 *bits* cada. A persistência do código de erro é de 1000 ciclos, ou seja, 10 *bits*. A Tabela 13.3 foi concebida com base nos dados do relatório no Apêndice D.

Tabela 13.3: Benchmark III - Relatório de Overhead

Módulo	Slices	Slice Reg	LUTs
RTOS–WD	472	191	820
Total	5484	1379	8530
Overhead	8,61 %	13,85 %	9,61 %

13.1.4 Resultados da análise de overhead

Dos dados descritos em: 13.1.1 Benchmark I, 13.1.2 Benchmark II e 13.1.3 Benchmark III. Observa-se que o overhead varia para cada caso conforme previsto, uma vez que a configuração do RTOS–WD é realizada antes da síntese lógica. Na Figura 13.1 encontram-se os overheads para cada benchmark assim como a média destes. O valor médio para o número de Slices foi de 8,45 %, Slice Regs 13,84 % e LUTs 9,38 %. Conforme dito nas seções acima os outros aspectos disponíveis no relatório e não dispostos no gráfico não foram gerados *overheads*.

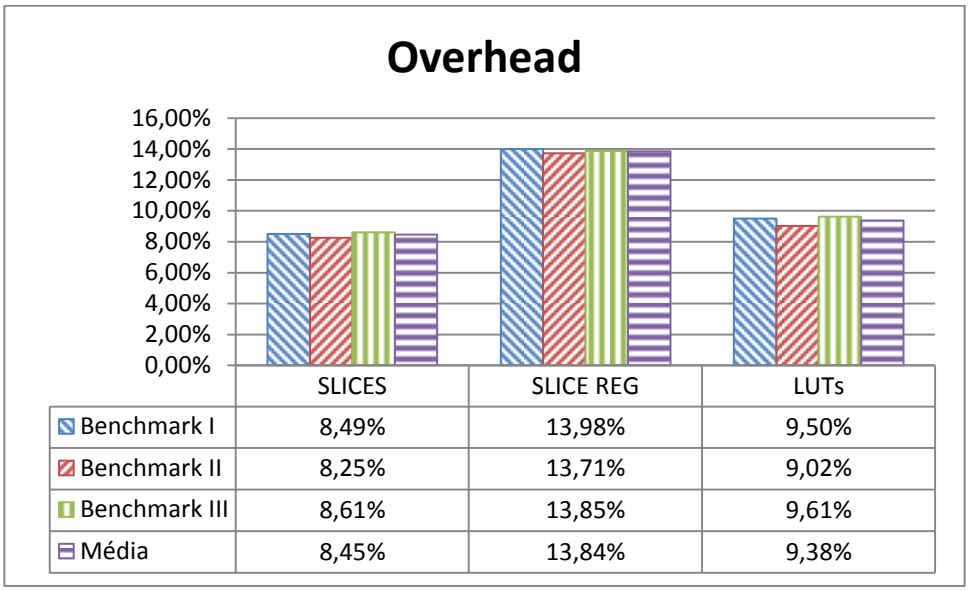


Figura 13.1: Resultados da análise de overhead

14 Conclusão

Pelos resultados obtidos verifica-se a alta capacidade de detecção de falhas do RTOS–Watchdog. Recalculando as taxas de detecção tanto do RTOS–Watchdog e do RTOS pelas falhas observadas (ver Figura 12.1) ao invés do número de experimentos, obtém-se que em valores médios o RTOS–Watchdog sinalizou 99,17 % das falhas detectadas e o RTOS sinalizou 70,54 % das falhas detectadas. No melhor dos casos o RTOS–Watchdog conseguiu detectar 100% das falhas contra 41,32 % pelo RTOS.

A confiabilidade de um sistema crítico é um fator crucial. Com overhead (13.1.4 Resultados da análise de overhead) de 8,45 % de *slices* é um valor baixo se levarmos em conta que é uma técnica completamente baseada em hardware além de não exercer influencia no desempenho do processador. Para uma estimativa mais precisa do overhead em área (mm^2, por exemplo) seria necessário uma síntese física do sistema.

Para sistemas com mais núcleos o *overhead* tende a diminuir, pois um núcleo adicional consome mais área que o hardware adicional para o seu monitoramento pelo RTOS–Watchdog.

15 Trabalhos Futuros

Como sugestão para trabalhos futuros pode-se trabalhar para a identificação dos recursos (através dos semáforos que os protegem) que estão sendo aguardados ou liberados pelas tarefas, aumentando-se assim a capacidade de detecção do RTOS–Watchdog.

Ou ainda adaptá-lo para aplicações “dinâmicas”, ou seja, com suporte para a criação de tarefas ou alterações de suas propriedades tais como, prioridade e restrição do núcleo de execução em tempo de execução.

16 Bibliografia

[1] [Online]. Available: http://www.ee.pucrs.br/sisc/.

[2] N. Ignat, B. Nicolescu, Y. Savaria e G. Nicolescu, "Soft-error classification and impact analysis on Real-Time Operating Systems," IEEE Design, Automation and Test *in Europe, Março 2006.*

[3] E. Toulopis, J. A. Flint, V. A. Chouliaras e D. D. Ward, "Study of the effects of SEU-Induced Faults on a Pipeline-Microprocessor," *IEEE Transactions on Computers, pp. 1585 - 1596, Dezembro* 2007.

[4] B. Nicolescu, N. Ignat, Y. Savaria e G. Nicolescu, "Sensitivity of Real-Time Operating Systems to Transient *Faults: A* case study for MicroC kernel," *RADECS Radiation and Its Effects on Components and Systems,* pp. F1-1 - F1-6, 19-23 Setembro 2005.

[5] "Wikipédia," [Online]. Available: http://pt.wikipedia.org/wiki/Multin%C3%BAcleo. [Acesso em 08 Março 2013].

[6] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi e D. A. Connors, "PLR: A Software Approach to Transient Fault Tolerance for Multicore Architectures*," IEEE Transactions on Dependable and Secure Computing,* pp. 135-148, Abril-Junho 2009.

[7] O. Nahmsuk, P. P. Shirvani e E. J. McCluskey, "Error Detection by Duplicated Instructions in Super-Scalar Processors," IEEE Transactions on Reliability, pp. 63-75, Março 2002.

[8] S. K. S. Hari, M.-L. Li, P. R. B. Choi e S. V. Adve, "mSWAT: Low-Cost Hardware Fault Detection and Diagnosis for Multicore Systems," *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on,* pp. 122-132, 2009.

[9] J. Tarrillo, L. Bolzani e F. Vargas, "A Hardware-Scheduler for Fault Detection in RTOS-based Embedded Systems," IEEE 12th Euromicro Conference and Digital System Design, pp. 341-347, 27-29 Agosto *2009.*

[10] D. S. d. Silva, Técnica de Detecção de Falhas de Escalonamento de Tarefas em Sistemas Embarcados Baseados em *Sistemas Operacionais de Tempo Real, Porto*

Alegre: PUCRS, 2011.

[11] IEC - International Electrotechnical Commission, Electromagnetic compatibility (EMC) - Part 4-29: Testing and Measurement Techniques - Voltage Dips, Short Interruptions and Voltage Variations on d.c. Input Power Port Immunity Tests (61.0004-29), Geneva, 2000.

[12] F. G. Zacchigna, Diseño, Implementación y Evaluación de un processador multi-núcleo, Buenos Aires: Universidad de Buenos Aires, 2012.

[13] A. Avižienis, J.-C. Laprie, B. Randell e C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, vol. I, n. 1, pp. 11-33, 2004.

[14] D. K. Pradhan, Fault-tolerant computer system design, Prentice Hall, 1996.

[15] J. Tarrillo, Escalonador em Hardware para Detecção de Falhas em Sistemas Embarcados de *Tempo Real, Porto* Alegre: PUCRS, 2009.

[16] Q. LI e C. YAO, Real-Time Concepts for Embedded Systems, San Francisco: CMP Books, 2003.

[17] A. S. Tanenbaum, Sistemas Operacionais Modernos., Prentice Hall, 2003.

[18] "Kernel Definition," The Linux Information Project, [Online]. Available: http://www.linfo.org/kernel.html. [Acesso em 1 Março 2014].

[19] "Semáforo (computação)," Wikipédia, [Online]. Available: http://pt.wikipedia.org/wiki/Sem%C3%A1foro_(computa%C3%A7%C3%A3o). [Acesso em 11 2013].

[20] A. S. Tanenbaum e A. S. Woodhull, Operating Systems Design and Implementation, Upper Saddle River: Prentice Hall, Inc., 1997.

[21] "Semaphore (programming)," Wikipedia, [Online]. Available: http://en.wikipedia.org/wiki/Semaphore_(programming). [Acesso em 11 2013].

[22] "Computer Multitasking," Wikipedia, the free encyclopedia, [Online]. Available: http://en.wikipedia.org/wiki/Computer_multitasking. [Acesso em 17 02 2014].

[23] S. Rhoads, "Plasma," 2001. [Online]. Available: opencores.org/project,plasma.

[24] "OpenCores," OpenCores, [Online]. Available: http://opencores.org. [Acesso em 1 Janeiro 2012].

[25] "Plasma - most MIPS I(TM) opcodes," OpenCores, [Online]. Available:

http://opencores.org/project,plasma. [Acesso em 1 Jan 2012].

[26] "Conducted electromagnetic interference - Wikipedia, the free encyclopedia," Wikipedia, the free encyclopedia, [Online]. Available*:* http://en.wikipedia.org/wiki/Conducted_electromagnetic_*interference. [Acesso* em 31 01 2014].

[27] "Electromagnetic interference - Wikipedia, the free encyclopedia," Wikipedia, the free encyclopedia, [Online]. Available: *http*://en.wikipedia.org/wiki/Electromagnetic_interference. [*Accessed 26 01* 2014].

[28] "EMI - Conducted Interference," Crane Aerospace & Electronic Power Solutions, [Online]. Available: http://*www*.interpoint.com/product_documents/DC_DC_Converters_EMI_*Conducted_Interference.pdf*. [Acesso em 2014 01 31].

[29] D. SILVA, C. C. OLIVEIRA, L. BOLZANI e F. VARGAS, "An Intellectual Property Core *to Detect Task Scheduling-Related Faults in RTOS-Based Embedded Systems*," SASE 2012, 2012.

[30] C. OLIVEIRA, J. D. Benfica, L. Poehls, F. Vargas, J. Lipovetzky, A. *Lutenberg, E. Gatti, F. Hernandez e A.* Boyer, "Reliability Analysis of *an On-Chip Watchdog* for Embedded Systems Exposed to Radiation and EMI," *9th International Workshop on Electromagnetic Compatibility of Integrated Circuits*, pp. 1-6, 2013.

[31] D. Silva, C. Oliveira, L. Poehls e F. Vargas, "An Intellectual Property Core to *Detect* Task Schedulling-Related Faults in RTOS-Based Embedded Systems," *17th IEEE International* On-Line Testing *Symposium (IOLTS'11)*, pp. 19-24, 2011.

[32] Xilinx, "ChipScope Pro and the Serial I/O Toolki," Xilinx, [Online]. Available: http://www.xilinx.com/tools/cspro.htm. [Acesso em 25 02 2014*]*.

[33] J. Benfica, L. M. Poehls, F. Vargas, J. Lipovetzky, A. Lutenberg, S. *García*, E. Gatti e F. Hernandez, "Evaluating *the Effects* of Combined Total Ionizing Dose Radiation and Electromagnetic Interference," IEEE TRANSACTIONS ON NUCLEAR SCIENCE, vol. 59, Agosto 2012.

[34] W. H. Robinson, M. L. Alles, T. A. Bapty, B. L. Bhuva, *J. D.* Black, A. B. Bonds,

L. W. Massengill, S. K. Neema, R. D. Schrimpf e J. M. Scott, "Soft Error Considerations for Multicore Microprocessor Design," *Integrated Circuit Design and Technology*, Maio 2007.

[35] J. Ohlsson, M. Rimén e U. Gunneflo, "A Study of the Effects of Transient Fault *Injection into a* 32-bit RISC with Built-in Watchdog," *Fault-Tolerant Computing*, pp. 316 - 325, Jul 1992.

[36] J.-C. Laprie, "Dependable Computing and Fault Tolerance: Concepts and Terminology," IEEE Fault-Tolerant Computing, 27-30 Jun *1995.*

[37] G. KORNAROS e D. PNEVMATIKATOS, "A Survey and Taxonomy of On-Chip Monitoring of Multicore Systems-on-Chip," *ACM Trans. Des. Autom.* Electron. Syst., Março 2013.

[38] N. Oh, P. P. Shirvani e E. J. McCluskey, "Control-Flow Checking by Software Signatures*," IEEE Transactions On Reliability*, Mar 2002.

[39] D. A. Petterson e J. L. Hennessy, Computer Organization and Design, Amsterdam: Elsevier, 2012.

[40] G. Miremadi e J. Torin, "Evaluating Processor-Behavior and Three Error-Detection Mechanisms Using Physical Fault-Injection," IEEE *Transactions* on Reliability*, Vol. 44, Nº 3, pp*. 441-454, Set 1995.

[41] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs e G. Leber, "Comparison of Physical and Software-implemented Fault Injection *Techniques," IEEE Transactions on Computers*, Vol. 52, Nº 9, pp. 1115-1133, 2003.

[42] D. Mosse, R. Melhem e S. G, "A non-preemptive real-time scheduler with recovery from transient *faults* and its implementation," IEEE Transaction on Software Engineering, Vol. 29, *Nº.* 8, pp. 752-767, 2003.

[43] V. Izosimov, P. Pop, P. Eles e Z. Peng, "Design optimization of time- and *cost*-constrained fault-tolerant distributed embedded systems," IEEE Design Automation and Test in *Europe*, pp. 864-869, 2005.

[44] P. Shirvani e R. M. E. J. Saxena, "Software-implemented EDAC protection against SEUs," IEEE Transactions on Reliability, Vol. 49, Nº 3, pp. 273-284, Set 2000.

[45] J. Tarrillo, L. Bolzani, F. Vargas, E. Gatti, F. Hernandez e L. Fraigi, "Fault-Detection Capability Analysis of a Hardware-Scheduler IP-Core in

Electromagnetic Interference Environment," *7th IEEE East-West Design & Test Symposium,* 2009.

[46] A. Silberschatz, P. B. Galvin e G. Gagne, Operating System Concepts, John Wiley & Sons, 1997.

[47] E. Sicard, F. Vargas, F. Hernandez, F. Fiori e J. P. Teixeira, "Design and Test on *Chip for EMC," IEEE Design &* Test of Computers, Vol. 23, pp. 502-503, 2006.

[48] H. P. E. W. M. F. Vranken e R. C. v. Wuijtswinkel, "Design for Testability in *Hardware-Software Systems," IEEE Design & Test* of Computers, pp. 79-87, 1996.

[49] V. D. Agrawal, C. R. Kime e K. K. Saluja, "A Tutorial on Built-In Self Test," *IEEE Design & Test of Computers,* pp. 73-82, Mar 1993.

[50] Y. Zorian, "Guest Editor's Introduction: Advances in Infrastructure IP," IEEE CS, Maio-Junho 2003.

APÊNDICES

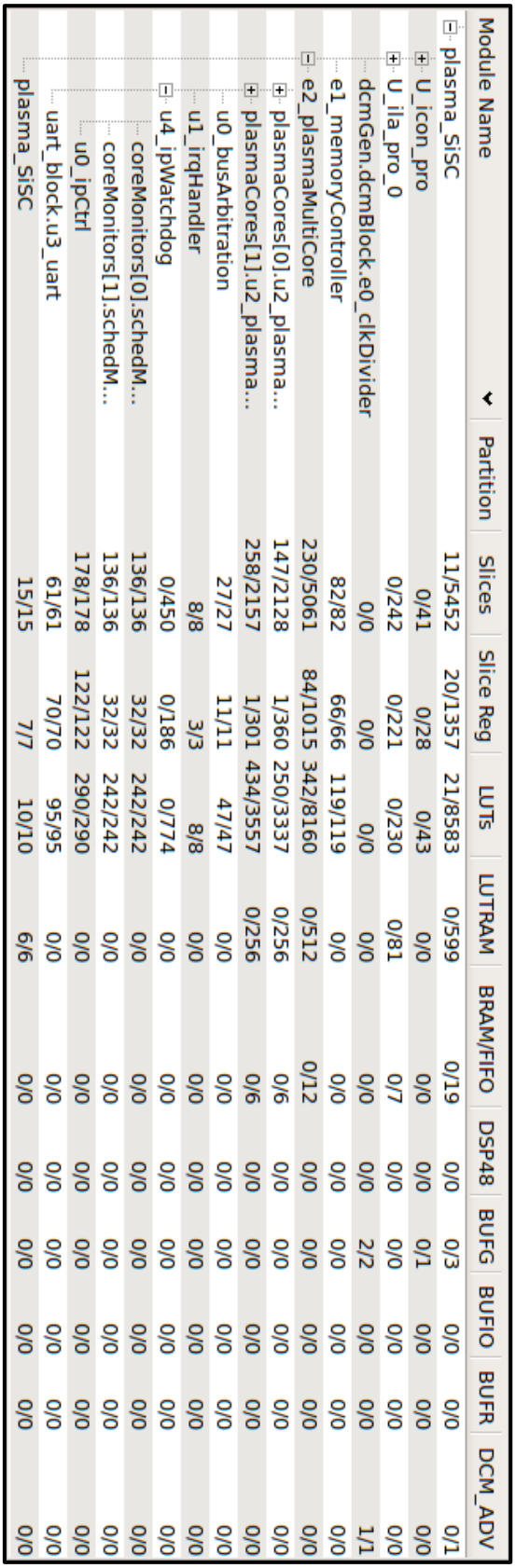
Apêndice A: RTOS-WD Constants Package

```
--------------------------------------------------------------------------------
-- Author: Christofer C. de Oliveira
--
-- Create Date: 15:16 12/04/2013
-- Design Name: ipConstants
-- Module Name: RTOS Watchdog Constants Pack
-- Project Name: RTOS Watchdog for Plasma Multicore
-- Target Devices: Multicore Plasma
-- Revision:
--  Revision 1.00
-- Additional Comments:
--  Benchmark Name: Benchmark0
--  Automatically generated by ipConstPackGen
--------------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;

package ipConstants is

  constant taskCount : integer := 9;

  constant taskLen : integer := 4;

  constant longjmp_i : std_logic_vector(31 downto 0) := x"100001c8";
  constant longjmp_f : std_logic_vector(31 downto 0) := x"10000208";

  constant OS_ThreadTimeoutRemove_i : std_logic_vector(31 downto 0) := x"100002dc";
  constant OS_ThreadTimeoutRemove_f : std_logic_vector(31 downto 0) := x"10000340";

  constant OS_ThreadReschedule_i : std_logic_vector(31 downto 0) := x"10000724";
  constant OS_ThreadReschedule_f : std_logic_vector(31 downto 0) := x"100009a4";

  constant OS_InterruptServiceRoutine_i : std_logic_vector(31 downto 0) := x"100009a4";
  constant OS_InterruptServiceRoutine_f : std_logic_vector(31 downto 0) := x"10000b80";

  constant OS_SemaphorePost_i : std_logic_vector(31 downto 0) := x"10000b80";
  constant OS_SemaphorePost_f : std_logic_vector(31 downto 0) := x"10000c40";

  constant OS_SemaphorePend_i : std_logic_vector(31 downto 0) := x"10000d48";
  constant OS_SemaphorePend_f : std_logic_vector(31 downto 0) := x"10000ecc";

  constant OS_ThreadTick_i : std_logic_vector(31 downto 0) := x"100017cc";
  constant OS_ThreadTick_f : std_logic_vector(31 downto 0) := x"10001878";

  constant OS_ThreadExit_i : std_logic_vector(31 downto 0) := x"10001dbc";
  constant OS_ThreadExit_f : std_logic_vector(31 downto 0) := x"10001e5c";

  constant TaskInt_i : std_logic_vector(31 downto 0) := x"10002980";
  constant TaskInt_f : std_logic_vector(31 downto 0) := x"10002a08";

  constant OS_IdleThread_i : std_logic_vector(31 downto 0) := x"10002a08";
  constant OS_IdleThread_f : std_logic_vector(31 downto 0) := x"10002a60";

end;
```

Apêndice B: Module Level Utilization Report – Benchmark I

Module Name	Partition	Slices	Slice Reg	LUTs	LUTRAM	BRAM/FIFO	DSP48	BUFG	BUFIO	BUFR	DCM_ADV
plasma_SiSC		11/5487	20/1381	21/8527	0/601	0/20	0/0	0/3	0/0	0/0	0/1
U_icon_pro		0/41	0/28	0/43	0/0	0/0	0/0	0/1	0/0	0/0	0/0
U_ila_pro_0		0/244	0/225	0/232	0/83	0/8	0/0	0/0	0/0	0/0	0/0
dcmGen.dcmBlock.e0_clkDivider		0/0	0/0	0/0	0/0	0/0	0/0	2/2	0/0	0/0	1/1
e1_memoryController		81/81	66/66	119/119	0/0	0/0	0/0	0/0	0/0	0/0	0/0
e2_plasmaMultiCore		228/5095	85/1035	342/8102	0/512	0/12	0/0	0/0	0/0	0/0	0/0
plasmaCores[0].u2_plasma...		160/2105	1/369	277/3346	0/256	0/6	0/0	0/0	0/0	0/0	0/0
plasmaCores[1].u2_plasma...		206/2200	1/303	315/3454	0/256	0/6	0/0	0/0	0/0	0/0	0/0
u0_busArbitration		26/26	12/12	47/47	0/0	0/0	0/0	0/0	0/0	0/0	0/0
u1_irqHandler		8/8	3/3	8/8	0/0	0/0	0/0	0/0	0/0	0/0	0/0
u4_ipWatchdog		0/466	0/193	0/810	0/0	0/0	0/0	0/0	0/0	0/0	0/0
coreMonitors[0].schedM...		134/134	32/32	239/239	0/0	0/0	0/0	0/0	0/0	0/0	0/0
coreMonitors[1].schedM...		134/134	32/32	239/239	0/0	0/0	0/0	0/0	0/0	0/0	0/0
u0_ipCtrl		198/198	129/129	332/332	0/0	0/0	0/0	0/0	0/0	0/0	0/0
uart_block.u3_uart		62/62	70/70	95/95	0/0	0/0	0/0	0/0	0/0	0/0	0/0
plasma_SiSC		15/15	7/7	10/10	6/6	0/0	0/0	0/0	0/0	0/0	0/0

Apêndice C: Module Level Utilization Report – Benchmark II



Module Name	Partition	Slices	Slice Reg	LUTs	LUTRAM	BRAM/FIFO	DSP48	BUFG	BUFIO	BUFR	DCM_ADV
plasma_SiSC		11/5452	20/1357	21/8583	0/599	0/19	0/0	0/3	0/0	0/0	0/1
U_icon_pro		0/41	0/28	0/43	0/0	0/0	0/0	0/1	0/0	0/0	0/0
U_ila_pro_0		0/242	0/221	0/230	0/81	0/7	0/0	0/0	0/0	0/0	0/0
dcmGen.dcmBlock.e0_clkDivider		0/0	0/0	0/0	0/0	0/0	0/0	2/2	0/0	0/0	1/1
e1_memoryController		82/82	66/66	119/119	0/0	0/0	0/0	0/0	0/0	0/0	0/0
e2_plasmaMultiCore		230/5061	84/1015	342/8160	0/512	0/12	0/0	0/0	0/0	0/0	0/0
plasmaCores[0].u2_plasma...		147/2128	1/360	250/3337	0/256	0/6	0/0	0/0	0/0	0/0	0/0
plasmaCores[1].u2_plasma...		258/2157	1/301	434/3557	0/256	0/6	0/0	0/0	0/0	0/0	0/0
u0_busArbitration		27/27	11/11	47/47	0/0	0/0	0/0	0/0	0/0	0/0	0/0
u1_irqHandler		8/8	3/3	8/8	0/0	0/0	0/0	0/0	0/0	0/0	0/0
u4_ipWatchdog		0/450	0/186	0/774	0/0	0/0	0/0	0/0	0/0	0/0	0/0
coreMonitors[0].schedM...		136/136	32/32	242/242	0/0	0/0	0/0	0/0	0/0	0/0	0/0
coreMonitors[1].schedM...		136/136	32/32	242/242	0/0	0/0	0/0	0/0	0/0	0/0	0/0
u0_ipCtrl		178/178	122/122	290/290	0/0	0/0	0/0	0/0	0/0	0/0	0/0
uart_block.u3_uart		61/61	70/70	95/95	0/0	0/0	0/0	0/0	0/0	0/0	0/0
plasma_SiSC		15/15	7/7	10/10	6/6	0/0	0/0	0/0	0/0	0/0	0/0

Apêndice D: Module Level Utilization Report – Benchmark III

Module Name	Partition	Slices	Slice Reg	LUTs	LUTRAM	BRAM/FIFO	DSP48	BUFG	BUFIO	BUFR	DCM_ADV
plasma_SiSC		11/5484	20/1379	21/8530	0/601	0/20	0/0	0/3	0/0	0/0	0/1
U_icon_pro		0/41	0/28	0/43	0/0	0/0	0/0	0/1	0/0	0/0	0/0
U_ila_pro_0		0/244	0/225	0/232	0/83	0/8	0/0	0/0	0/0	0/0	0/0
dcmGen.dcmBlock.e0_clkDivider		0/0	0/0	0/0	0/0	0/0	0/0	2/2	0/0	0/0	1/1
e1_memoryController		82/82	66/66	119/119	0/0	0/0	0/0	0/0	0/0	0/0	0/0
e2_plasmaMultiCore		227/5091	85/1033	342/8105	0/512	0/12	0/0	0/0	0/0	0/0	0/0
plasmaCores[0].u2_plasma...		162/2100	1/368	277/3339	0/256	0/6	0/0	0/0	0/0	0/0	0/0
plasmaCores[1].u2_plasma...		208/2195	1/303	315/3454	0/256	0/6	0/0	0/0	0/0	0/0	0/0
u0_busArbitration		27/27	13/13	47/47	0/0	0/0	0/0	0/0	0/0	0/0	0/0
u1_irqHandler		8/8	3/3	8/8	0/0	0/0	0/0	0/0	0/0	0/0	0/0
u4_ipWatchdog		0/472	0/191	0/820	0/0	0/0	0/0	0/0	0/0	0/0	0/0
coreMonitors[0].schedM...		143/143	32/32	257/257	0/0	0/0	0/0	0/0	0/0	0/0	0/0
coreMonitors[1].schedM...		143/143	32/32	257/257	0/0	0/0	0/0	0/0	0/0	0/0	0/0
u0_ipCtrl		186/186	127/127	306/306	0/0	0/0	0/0	0/0	0/0	0/0	0/0
uart_block.u3_uart		62/62	70/70	95/95	0/0	0/0	0/0	0/0	0/0	0/0	0/0
plasma_SiSC		15/15	7/7	10/10	6/6	0/0	0/0	0/0	0/0	0/0	0/0

Apêndice E: Benchmark I – Código-fonte

```
uint32
  resource1,
  resource2,
  resource3,
  resource4;

OS_Semaphore_t
  *ST1,  // Contagem inicial 0
  *ST2,  // Contagem inicial 0
  *ST3,  // Contagem inicial 1
  *ST4,  // Contagem inicial 0
  *ST5,  // Contagem inicial 1
  *ST6,  // Contagem inicial 0
  *ST7,  // Contagem inicial 0
  *ST8;  // Contagem inicial 0
/*******************************************************/
// Restrição de Núcleo: 0
void Task1(void *arg) {
  int i;

  for (; ;) {
    OS_SemaphorePend(ST1, OS_WAIT_FOREVER);

    for (i=0; i < 15000; ++i) {
      resource1++;
    }

    OS_SemaphorePost(ST2);
  }
}
/*******************************************************/
// Restrição de Núcleo: 0
void Task2(void *arg) {
  int i;

  for (; ;) {
    OS_SemaphorePend(ST2, OS_WAIT_FOREVER);

    for (i=0; i < 15000; ++i) {
      resource1++;
    }

  }
}
/*******************************************************/
// Restrição de Núcleo: Livre
void Task3(void *arg) {
  int i;

  for (; ;) {
    OS_SemaphorePend(ST3, OS_WAIT_FOREVER);

    for (i=0; i < 10000; ++i) {
      resource2++;
    }

    OS_SemaphorePost(ST4);
    OS_ThreadSleep(1);
```

```
  }
}
/*****************************************************/
// Restrição de Núcleo: Livre
void Task4(void *arg) {
  int i;

  for (; ;) {
    OS_SemaphorePend(ST4, OS_WAIT_FOREVER);

    for (i=0; i < 15000; ++i) {
      resource2++;
    }

    OS_SemaphorePost(ST3);
  }
}
/*****************************************************/
// Restrição de Núcleo: Livre
void Task5(void *arg) {
  int i;

  for (; ;) {
    OS_SemaphorePend(ST5, OS_WAIT_FOREVER);

    for (i=0; i < 10000; ++i) {
      resource3++;
    }

    OS_SemaphorePost(ST6);
  }
}
/*****************************************************/
// Restrição de Núcleo: Livre
void Task6(void *arg) {
  int i;

  for (; ;) {
    OS_SemaphorePend(ST6, OS_WAIT_FOREVER);

    for (i=0; i < 15000; ++i) {
      resource3++;
    }

    OS_SemaphorePost(ST5);
  }
}
/*****************************************************/
// Restrição de Núcleo: 1
void Task7(void *arg) {
  int i;

  for (; ;) {
    OS_SemaphorePend(ST7, OS_WAIT_FOREVER);

    for (i=0; i < 5000; ++i) {
      resource4++;
    }

    OS_SemaphorePost(ST8);
```

```
    for (i=0; i < 5000; ++i) {
      resource4++;
    }
  }
}
/*******************************************************/
// Restrição de Núcleo: 1
void Task8(void *arg) {
  int i;

  for (; ;) {
    OS_SemaphorePend(ST8, OS_WAIT_FOREVER);

    for (i=0; i < 15000; ++i) {
      resource4++;
    }
  }
}
/*******************************************************/
void TaskInt(void *arg) {
  uint32 mask, state;
  int IntCount;

  // RTOS-WD SYNC
  MemoryWrite(IP_TASK_ID, 0x7FFFFFFF);

  // Toggle looking for IRQ_GPIO31 or IRQ_GPIO31_NOT
  state = OS_CriticalBegin();

  mask = MemoryRead(IRQ_MASK);
  mask ^= IRQ_GPIO31 | IRQ_GPIO31_NOT;
  MemoryWrite(IRQ_MASK, mask);

  OS_CriticalEnd(state); // OS_SpinUnlock(state);

  for (IntCount = 0; IntCount < 7500; IntCount++);

  OS_SemaphorePost(ST1);

  for (; IntCount < 15000; IntCount++);

  OS_SemaphorePost(ST7);
}
```

Apêndice F: Benchmark II – Código-fonte

```
OS_Semaphore_t
  *S1;  // Contagem inicial 1

OS_Mutex_t
  *M1,
  *M2;

OS_MQueue_t
  *Q1;  // Tamanho 1 x 4bytes

int
  resource1,
  resource2;
/*******************************************************/
// Restrição de Núcleo: 0
void Task1(void *arg) {
  int i, j;

  for(; ;) {
    MemoryWrite(GPIO0_OUT, 0x1);

    for (i = 0; i < 10000; i++);

    while (OS_MQueueSend(Q1, &i) == -1)
      for (j = 0; j < 200; j++);

    OS_ThreadSleep(1);
  }
}
/*******************************************************/
// Restrição de Núcleo: 0
void Task2(void *arg) {
  int i;

  for(; ;) {

    OS_MQueueGet(Q1, &i, OS_WAIT_FOREVER);

    for (; i < 20000; i++);

  }
}
/*******************************************************/
// Restrição de Núcleo:
//   Tarefa 3: Livre
//   Tarefa 4: Livre
void Task34(void *arg) {
  int i;

  for(; ;) {
    OS_MutexPend(M1);

    for (i = 0; i < 15000; i++)
      resource1++;

    OS_MutexPost(M1);

    for (i = 0; i < 2000; i++);
```

```
    OS_MutexPend(M2);

    for (i = 0; i < 15000; i++)
      resource2++;

    OS_MutexPost(M2);

  }
}
/*******************************************************/
// Restrição de Núcleo:
//   Tarefa 5: 1
//   Tarefa 6: 1
void Task56(void *arg) {
  int i;

  for(; ;) {
    OS_SemaphorePend(S1, OS_WAIT_FOREVER);

    for (i = 0; i < 10000; i++);

    OS_SemaphorePost(S1);

    OS_ThreadSleep(1);
  }
}
/*******************************************************/
void TaskInt(void *arg) {
  uint32 mask, state;
  int IntCount;

  // RTOS-WD SYNC
  MemoryWrite(IP_TASK_ID, 0x7FFFFFFF);

  // Toggle looking for IRQ_GPIO31 or IRQ_GPIO31_NOT
  state = OS_CriticalBegin();

  mask = MemoryRead(IRQ_MASK);
  mask ^= IRQ_GPIO31 | IRQ_GPIO31_NOT;
  MemoryWrite(IRQ_MASK, mask);

  OS_CriticalEnd(state);

  for (IntCount = 0; IntCount < 10000; IntCount++);
}
```

Apêndice G: Benchmark III – Código-fonte

```
OS_Semaphore_t
  *S1,  // Contagem inicial 1
  *S2,  // Contagem inicial 0
  *S3;  // Contagem inicial 1

OS_MQueue_t
  *Q1,  // Tamanho 1 x 4bytes
  *Q2,  // Tamanho 1 x 4bytes
  *Q3;  // Tamanho 1 x 4bytes

uint32
  resource1,
  resource2;
/*******************************************************/
// Restrição de Núcleo: 0
void Task1(void *arg) {
  int i, j;

  for(; ;) {
    for (i = 0; i < 20000; i++);

    while (OS_MQueueSend(Q1, &i) == -1)
      for (j = 0; j < 200; j++);

    OS_ThreadSleep(1);

    OS_MQueueGet(Q2, &i, OS_WAIT_FOREVER);

    for (; i < 60000; i++);

    while (OS_MQueueSend(Q3, &i) == -1)
      for (j = 0; j < 200; j++);

    OS_ThreadSleep(1);

  }
}
/*******************************************************/
// Restrição de Núcleo: 1
void Task2(void *arg) {
  int i, j;

  for(; ;) {
    OS_MQueueGet(Q1, &i, OS_WAIT_FOREVER);

    for (; i < 40000; i++);

    while (OS_MQueueSend(Q2, &i) == -1)
      for (j = 0; j < 200; j++);

  }
}
/*******************************************************/
// Restrição de Núcleo: 1
void Task3(void *arg) {
  int i;

  for(; ;) {
```

```
    OS_MQueueGet(Q3, &i, OS_WAIT_FOREVER);

    for (; i < 80000; i++);

  }
}
/*****************************************************/
// Restrição de Núcleo: 0
void Task4(void *arg) {
  int i;

  for(; ;) {
    OS_SemaphorePend(S1, OS_WAIT_FOREVER);

    for (i = 0; i < 15000; i++)
      resource1++;

    OS_SemaphorePost(S2);
  }
}
/*****************************************************/
// Restrição de Núcleo: 1
void Task5(void *arg) {
  int i;

  for(; ;) {
    OS_SemaphorePend(S2, OS_WAIT_FOREVER);

    for (i = 0; i < 15000; i++)
      resource1++;

    OS_SemaphorePost(S1);
  }
}
/*****************************************************/
// Restrição de Núcleo:
//   Tarefa 6: 0
//   Tarefa 7: 1
// Argumento: Numero de ticks para encerramento
//   Tarefa 6: 3000
//   Tarefa 7: 4500
void Task67(void *arg) {
  int i;
  int tickCount;

  for (tickCount = *((int*)arg); tickCount > 0; --tickCount) {

    OS_SemaphorePend(S3, OS_WAIT_FOREVER);

    for (i = 0; i < 15000; ++i) {
      resource2++;
    }

    OS_SemaphorePost(S3);
    OS_ThreadSleep(1);
  }

  OS_ThreadExit();
}
/*****************************************************/
void TaskInt(void *arg) {
```

```
  uint32 mask, state;
  int IntCount;

  // RTOS-WD SYNC
  MemoryWrite(IP_TASK_ID, 0x7FFFFFFF);

  // Toggle looking for IRQ_GPIO31 or IRQ_GPIO31_NOT
  state = OS_CriticalBegin();

  mask = MemoryRead(IRQ_MASK);
  mask ^= IRQ_GPIO31 | IRQ_GPIO31_NOT;
  MemoryWrite(IRQ_MASK, mask);

  OS_CriticalEnd(state);

  for (IntCount = 0; IntCount < 10000; IntCount++);
}
```

Apêndice H: Alteração *OS_Start()*

```
void OS_Start(void) {
  // RTOS-WD Starting Control
  // Escreve 1 em 0x200000B0 (o valor é irrelevante, pois é conhecido
  // qual núcleo acessou o barramento)
  MemoryWrite(IP_START_REG, 1);

  ThreadSwapEnabled = 1;
  (void)OS_CriticalBegin(); // OS_SpinLock();
  OS_ThreadReschedule(1);
  OS_CriticalEnd(0); // OS_SpinUnlock(0); // TODO  la agregue yo
}
```

Apêndice I: Alteração *longjmp(jmp_buf env, int taskID)*

```
  .global  longjmp
  .ent     longjmp
longjmp:
  .set noreorder
  lw  $16, 0($4)    #s0
  lw  $17, 4($4)    #s1
  lw  $18, 8($4)    #s2
  lw  $19, 12($4)   #s3
  lw  $20, 16($4)   #s4
  lw  $21, 20($4)   #s5
  lw  $22, 24($4)   #s6
  lw  $23, 28($4)   #s7
  lw  $30, 32($4)   #s8
  lw  $28, 36($4)   #gp
  lw  $29, 40($4)   #sp
  lw  $31, 44($4)   #lr

  # Informa o taskID ($5) para o RTOS-WD
  # O endereço reservado é 0x200000B4
  lui $8, 0x2000
  sw  $5, 0xB4($8) #0x200000B4

  jr  $31
  ori $2, $0, 1

  .set reorder
  .end longjmp
```

Apêndice J: Uso das *assertions* pelo PlasmaRTOS

```
// Macro assert verifica se a condição 'A' é verdadeira ou falsa
// Se a condição A for falsa a função OS_Assert é chamada e
// imprime o arquivo e a linha onde a macro foi inclusa a macro
// assert(.) pode se criar quaisquer condições desde que em caso
// de erro ela seja igual a '0'
#define assert(A) if((A)==0){ OS_Assert(__FILE__, __LINE__); }

// Se roundRobin for 1 tarefas de igual prioridade são escalonadas
// Se roundRobin for 0 somente tarefas de maior prioridade são
// escaladas
void OS_ThreadReschedule(int roundRobin) {
  OS_Thread_t
    *threadNext, // Ponteiro para a próxima tarefa
    *threadCurrent; // Ponteiro para a tarefa atual

  int
    rc, // Codigo de retorno de setjmp se for 0 e
        //1 se for de longjmp
    cpuIndex = OS_CpuIndex(); // Núcleo no qual a função
                              // esta sendo executada

  // Se esta função for chamada durante interrupção
  // Define flag para o RTOS escalonar tarefa
  // após o atendimento de interrupção
  if (ThreadSwapEnabled == 0 || InterruptInside[cpuIndex]) {
    ThreadNeedReschedule[cpuIndex] |= 2 + roundRobin;
    return;
  }

  ThreadNeedReschedule[cpuIndex] = 0;

  // Carrega primeira tarefa da lista PRONTAS para threadNext
  threadNext = ThreadHead;

  // Verifica se a primeira tarefa na lista de tarefas prontas
  // tem seu estado definido como PRONTO, caso não esteja vazia
  assert(threadNext == NULL || threadNext->state == THREAD_READY);

  // Percorre a lista até uma tarefa que possa ser executada
  // neste núcleo, respeitando a restrição de núcleo (core lock)
  while(threadNext && threadNext->cpuLock != -1 &&
    threadNext->cpuLock != cpuIndex)
  {
    threadNext = threadNext->next;
  }
  if (threadNext == NULL)
    return;

  // Verifica se o seu estado está definido como PRONTO
  assert(threadNext->state == THREAD_READY);

  threadCurrent = ThreadCurrent[cpuIndex];

  // Verifica se a próxima tarefa deve ser escalonada
  // ou se a tarefa atual é a de maior prioridade
  if (threadCurrent == NULL ||
    threadCurrent->state == THREAD_PEND ||
    threadCurrent->priority < threadNext->priority ||
```

```
    (roundRobin && threadCurrent->priority == threadNext->priority))
  {
    // Swap threads
    ThreadCurrent[cpuIndex] = threadNext;
    (threadCurrent) {
      // Verifica estouro de pilha por magic number
      assert(threadCurrent->magic == THREAD_MAGIC);

      // Insere a tarefa atual na lista de tarefas prontas
      if (threadCurrent->state == THREAD_RUNNING)
        OS_ThreadPriorityInsert(&ThreadHead, threadCurrent);

      // Salva contexto da tarefa atual
      rc = setjmp(threadCurrent->env); // setjmp retorna 0
      if (rc) {
        return;  // Retornou de longjmp(), longjmp retorna 1
      }
    }

    // Verifica se ThreadCurrent foi corretamente definida
    assert(threadNext == ThreadCurrent[OS_CpuIndex()]);
    // Verifica se o seu estado esta definido como PRONTO
    assert(threadNext->state == THREAD_READY);

    // Remove a tarefa selecionada da lista de tarefas prontas
    OS_ThreadPriorityRemove(&ThreadHead, threadNext);

    // Altera seu estado para RUNNING
    threadNext->state = THREAD_RUNNING;

    // Indica em qual núcleo esta tarefa está sendo executada
    threadNext->cpuIndex = OS_CpuIndex();

    //Carrega o novo contexto de tarefa
    longjmp(threadNext->env, threadNext->id /*1*/);
  }
}
```