PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Técnica de Detecção de Falhas de Escalonamento de Tarefas em Sistemas Embarcados Baseados em Sistemas Operacionais de Tempo Real

Dhiego Sant'Anna da Silva

Orientador: Prof. Dr. Fabian Luis Vargas

Porto Alegre
2011

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE ENGENHARIA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Técnica de Detecção de Falhas de Escalonamento de Tarefas em Sistemas Embarcados Baseados em Sistemas Operacionais de Tempo Real

Dhiego Sant'Anna da Silva

Orientador: Prof. Dr. Fabian Luis Vargas

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
2011

Técnica de Detecção de Falhas de Escalonamento de Tarefas em Sistemas Embarcados Baseados em Sistemas Operacionais de Tempo Real

Candidato: Dhiego Sant'Anna da Silva

Esta dissertação foi julgada para a 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

Comissão Julgadora:

Prof. Dra.
Letícia Maria Veiras Bolzani

Prof. Dra.
Taisy Silva Weber

Prof. Dr.
Fabian Luis Vargas

Tudo sob controle®

Agradecimentos

Aos meus pais, Jane e Enio, e minha avó, Lourdes, pelo apoio incondicional nesta e em todas as outras escolhas da minha vida.

Aos professores Fabian Vargas e Letícia Bolzani pela confiança, apoio e orientação.

A todos os colegas do laboratório Signals & Systems for Computing Group (SiSC) e Laboratório de Ensino e Pesquisa (LEP) pelas inúmeras contribuições e por tornarem estes dois anos de trabalho e dedicação tão prazerosos.

A CAPES pelo apoio financeiro que permitiu meu ingresso ao Programa de Pós-Graduação em Engenharia Elétrica (PPGEE).

A todos, o meu, MUITO OBRIGADO !!!

Resumo

A alta complexidade dos sistemas de tempo real aumentou significativamente a necessidade da utilização de Sistemas Operacionais de Tempo Real (RTOS - *Real Time Operating System*) com o objetivo de simplificar o projeto dos mesmos. Neste contexto, sistemas embarcados baseados em RTOS exploram uma série de funcionalidades e facilidades inerentes ao mesmo, tais como o gerenciamento de tarefas, a concorrência, o acesso à memória e as interrupções. Assim, o RTOS funciona com uma interface entre o software e o hardware.

Porém, sistemas de tempo real são frequentemente afetados por falhas transientes oriundas de diferentes fontes, tal como a interferência eletromagnética (EMI - *Eletromagnetic Interference*), que pode gerar falhas capazes de degradar seu comportamento, afetando tanto a aplicação em execução quanto o sistema operacional embarcado.

Neste contexto, a principal ideia por trás deste trabalho é a implementação de uma *Infrastructure Intellectual-Property* (I-IP) denominado RTOS-Guardian (RTOS-G), baseada em hardware, capaz de monitorar o fluxo de execução do RTOS com o intuito de detectar falhas que eventualmente alterem a ordem de execução das tarefas que compõem a aplicação. Ao final, experimentos práticos baseados em uma técnica de injeção de falhas por *hardware* demonstram que, quando comparado com os mecanismos implementados pelo RTOS que visam proteger e monitorar a execução das principais operações de controle funcional e de fluxo do RTOS, o RTOS-G garante uma detecção de falhas mais elevada e uma latência de detecção de falhas bastante inferior.

Abstract

The high complexity of real-time systems significantly increased the need of Real Time Operating Systems (RTOS) in order to simplify the design of them. In this context, RTOS based systems explore a number of features and facilities inherit in the RTOS, such as task management, competition, the memory access and interrupts. Thus, the RTOS performs like an interface between software and hardware.

However, real-time systems are often affected by transient faults from different sources, such as electromagnetic interference (EMI), which may affect system functional behavior by degrading not only the aplications running on the system, but also the RTOS as well.

In this context, the main idea behind this work is to implement an I-IP (*Infrastructure Intellectual-Property*) called RTOS-G, hardware-based, able to monitor the RTOS execution flow to detect faults affecting the sequence by which the processor executes the application tasks and the RTOS kernel as well. Finally, practical experiments are presented and discussed. When compared to RTOS native functions, such experiments demonstrate that the RTOS-G ensures a higher fault detection and a significantly lower fault latency.

Lista de Figuras

Lista de Tabelas

Nomenclatura

CI: Circuitos integrados

CPU: Central Processor Unit

DDR SDRAM: Double-Data-Rate Synchronous Dynamic Random Access Memory

EMI: Electro-Magnetic Interference

FC: Flow Checker

I-IP: Infrastructure Intellectual-Property

ISR: Interrupt Service Routines

ITRS: Internacional Technology Roadmap for Semiconductor

LCEG: List Checker and Error Generator

LUT: Look Up Table

PC: Personal Computer

RTOS-G: RTOS-Guardian

RTOS: Real Time Operating System

SEU: Single-Event Upset

SoC: System-on-Chip

SRAM: Static Random Access Memory

TC: Task Checker

UART: Universal asynchronous receiver/transmitter

VHDL: VHSIC Hardware Description Language

VHSIC: Very High Speed Integrated Circuits

VLWI: Very Long Word Instruction

Sumário

PARTE I - FUNDAMENTOS

1 Introdução

Atualmente é possível observar que o número de sistemas embarcados nas mais diversas aplicações e segmentos de mercado tais como automotivo, médico hospitalar e aviação, cresce constantemente na nossa sociedade. Além disto, é possível observar que a grande maioria deles suporta aplicações críticas de tempo real, isto é, estes sistemas devem respeitar rígidos limites no que diz respeito ao tempo de execução. Em mais detalhes, este tipo de aplicação requer sistemas capazes de gerarem resultados corretos e dentro do limite de tempo esperado [21]. Especificamente nestes casos, uma falha no sistema pode produzir efeitos catastróficos, representando desde prejuízos para a sociedade e para a economia, até risco para a vida de pessoas.

O aumento da hostilidade no ambiente causado substancialmente pela adoção de tecnologias *wireless*, tais como telefones celulares e equipamentos compatíveis com tecnologia Bluetooth, representa um grande desafio para a confiabilidade de sistemas embarcados de tempo real [41]. Mais especificamente, condições como interferência eletromagnética e distúrbios na fonte de alimentação podem causar falhas transientes ou perturbações que impedem que o sistema responda às exigências de tempo real [29] [6]. Atualmente, falhas transientes representam um dos principais problemas em aplicações críticas baseadas em sistemas embarcados. O *Internacional Technology Roadmap for Semiconductor* (ITRS) prevê para a próxima geração de circuitos integrados um aumento significativo do número de sistemas embarcados com falhas devido a este tipo específico de falha [2]. Sistemas embarcados baseados em RTOS estão sujeitos a *Single Event Upset* (SEU), causando falhas transientes que afetam as aplicações que estão rodando no sistema embarcado bem como o RTOS que está executando essa aplicação [29] [6]. Basicamente, este tipo de falha pode afetar o escalonamento das tarefas alterando o correto funcionamento do sistema.

Neste cenário, pesquisadores e engenheiros implementam e propõem novas técnicas capazes de aumentar a confiabilidade dos sistemas operacionais embarcados e do sistema como um todo. Estas técnicas se dividem em técnicas baseadas em *software* e técnicas baseadas em *hardware*. Entre as técnicas

baseadas em *software* é possível salientar técnicas que detectam erros no fluxo de execução da aplicação através do monitoramento e comparação de assinaturas calculadas em tempo de execução do programa com valores pré-calculados durante a compilação [31] [14] [19] [46] [17] [42] [18]. As técnicas baseadas em *hardware* exploram o uso de módulos denominados *watch-dog processors* com o objetivo de monitorar a execução do programa e os acessos aos módulos de memória [15] [44] [12] [28] [11] [8] [16]. Entretanto, estas propostas apresentam soluções somente para o nível de aplicação, elas não consideram falhas que afetam o RTOS e eventualmente se propagam para as tarefas e aplicação.

Considerando falhas que podem se propagar para as tarefas que compõem uma dada aplicação, cerca de 21% delas ocasionam falhas na aplicação [22]. Geralmente, estas falhas ocasionam a perda de deadlines bem como valores incorretos na saída. Por outro lado, os trabalhos apresentados em [29] e [20] demonstram que 34% das falhas injetadas nos registradores do processador levam a erros de escalonamento. Concluindo, 44% destas disfunções levam a falha total do sistema, 34% causam problemas de tempo real e os 22% restantes geram valores incorretos na saída do sistema.

1.1 Motivação

Os trabalhos apresentados em [38] e [39], propõem uma solução baseada em *hardware* capaz de detectar falhas nas tarefas da aplicação, baseada na implementação de um I-IP capaz de monitorar o fluxo de execução das tarefas e indicar erros de tempo e sequência. Entretanto, a técnica apresentada possui algumas limitações, visto que ela funciona apenas para um tipo específico de algoritmo de escalonamento denominado *Round-Robin*, e não permite o uso de interrupções. Assim, a principal motivação para esse trabalho é propor um novo I-IP capaz de monitorar sistemas embarcados baseados em RTOS que adotam escalonamento preemptivo.

1.2 Objetivos

Este trabalho de mestrado tem como principal objetivo propor uma técnica inovadora baseada em *hardware* capaz de aumentar a robustez de sistemas embarcados baseados em RTOS. A técnica proposta baseia-se na implementação de um núcleo I-IP (*Infrastructure Intellectual-Property*), capaz de detectar falhas em sistemas embarcados baseados em RTOS, que adotam o escalonamento preemptivo e com tratamento de interrupções. A detecção de falhas é feita através da monitoração em tempo real das tarefas da aplicação. A principal diferença entre o novo I-IP e o anteriormente proposto em [38] e [39] ,

é que o I-IP proposto neste trabalho representa uma solução mais flexível visto que é capaz de monitorar um número variável de tarefas baseando-se para isto no algoritmo de escalonamento preemptivo.

1.3 Apresentação dos Capítulos

O trabalho é dividido em três grandes partes. A primeira parte apresenta a teoria que fundamenta esse trabalho. A segunda parte apresenta a metodologia utilizada para implementação da proposta, diagramas de blocos das arquiteturas e a plataforma de teste desenvolvida para os experimentos de injeção de falhas. E, finalmente a terceira parte apresenta os resultados obtidos após a realização dos experimentos realizados com o intuito de analisar e validar a técnica proposta, bem como também a conclusão deste trabalho de mestrado.

2 *Sistemas Operacionais de Tempo Real*

2.1 Introdução

Um RTOS (*Real Time Operating System*) é um programa que agenda a execução de tarefas no tempo correto previsto para tal, gerencia os recursos do sistema, e proporciona uma base consistente e coerente para o desenvolvimento das aplicações. Basicamente, o RTOS pode ser classificado em *Hard-RTOS* e *Soft-RTOS*. A principal diferença entre eles é que o *Soft-RTOS* pode tolerar latências nas respostas sem diminuir a qualidade do seu serviço enquanto que o *Hard-RTOS* exige que todas as *deadlines* sejam respeitadas rigidamente para evitar que o sistema falhe [27].

De modo geral o RTOS é composto pelos seguintes componentes [27]:

-Escalonador: este componente determina qual e quando uma determinada tarefa será executada. A maioria dos kernels suportam os algoritmos de escalonamento Round-Robin e o preemptivo.

-Objetos: são elementos utilizados para desenvolvimento das aplicações dos sistemas embarcado de tempo real. Os principais objetos são tarefas, semáforos e fila de mensagens.

-Serviços: são as operações que o kernel do sistema operacional executa nos objetos tais como temporização, interrupções e gerenciamento de recursos.

2.2 Escalonador

2.2.1 Algoritmo de Escalonamento Round-Robin

Conforme anteriormente mencionado, existem fundamentalmente dois tipos de algoritmos de escalonamento. Um dos algoritmos mais antigos, mais simples e mais amplamente utilizados, é o escalonamento round-robin. A cada tarefa é atribuído um intervalo de tempo (*time slice*) durante o qual ela pode ser executada. Se a tarefa estiver em execução no fim de seu intervalo de tempo, é feita a troca da tarefa

em execução. Se o processo tiver sido bloqueado, ou terminado, antes do intervalo ter expirado, a troca de tarefa será feita neste momento [37]. A Figura 2.1 apresenta o algoritmo *round-robin*. Observando a figura abaixo é possível verificar que a aplicação em questão é composta por três tarefas denominadas Task1, Task2 e Task3 com o mesmo nível de prioridade e mesmo tempo disponível para execução [27].

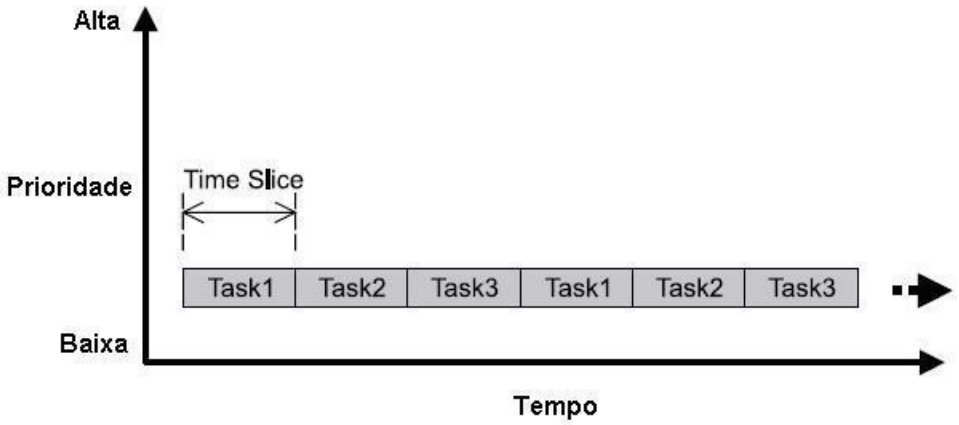


Figura 2.1: Escalonamento *Round-Robin* [27]

2.2.2 Algoritmo de Escalonamento Preemptivo

Entretanto quando consideramos o algoritmo preemptivo, observa-se que a tarefa a ser executada é sempre aquela com a maior prioridade de execução. Em outras palavras, a execução não segue uma ordem pré-definida e sim baseia-se na prioridade das tarefas que estão prontas para serem executadas. A Figura 2.2 apresenta o algoritmo preemptivo. Observando a figura a seguir, é possível verificar que o tempo dedicado a execução das tarefas que compõem a aplicação não é sempre o mesmo, podendo variar dependendo das prioridades das tarefas prontas para serem executadas. Em mais detalhes, isto significa que quando uma determinada tarefa de maior prioridade que a tarefa em execução, estiver pronta para ser executada, o algoritmo interrompe a execução da tarefa atual e passa a executar a tarefa de maior prioridade. Após a sua execução o algoritmo volta a executar a próxima tarefa pronta de maior prioridade, podendo voltar para a tarefa anterior ou até mesmo pular para uma terceira tarefa de maior prioridade ainda. Assim, a regra para este algoritmo é executar sempre a tarefa de maior prioridade que esteja pronta [27].

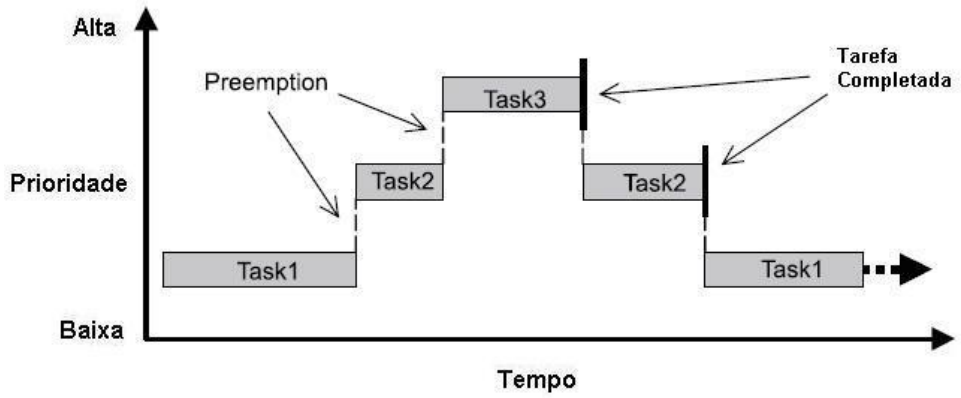


Figura 2.2: Escalonamento Preemptivo [27]

2.3 Objetos

2.3.1 Tarefas

Tarefa é uma *thread* independente que concorre com outras tarefas pelo tempo de execução em um determinado sistema. Dependendo do sistema operacional em questão, os estágios utilizados durante a execução das tarefas, podem receber diferentes nomes e serem classificados de diferentes formas. A Figura 2.3 mostra um exemplo composto pelos estados de controle típicos de RTOSs para a execução das tarefas. Assim, os principais estágios utilizados pelos sistemas operacionais que adotam o escalonamento preemptivo são: *Blocked, Ready* e *Running* [27].

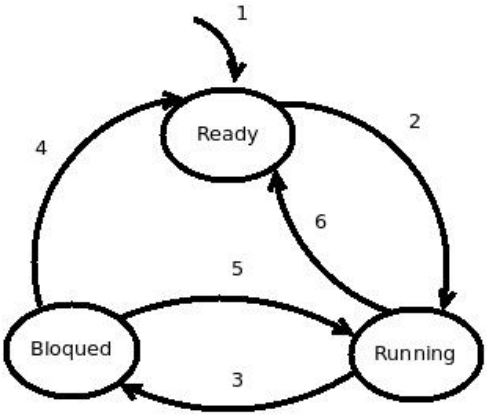


Figura 2.3: Típica máquina de estados para execução das tarefas [27]

Blocked: uma determinada tarefa está *blocked* quando a mesma requisitou um recurso que não está disponível ou está aguardando a ocorrência de algum evento [27].

Ready: uma determinada tarefa está *ready* quando a mesma está pronta para ser executada, mas existe outra tarefa com maior prioridade sendo executada [27].

Running: este estado representa uma tarefa que está sendo executada. É importante salientar que a tarefa que está sendo executada obrigatoriamente é aquela que possui a maior prioridade entre as tarefas prontas para serem executadas [27].

Assim, a figura 2.3 ilustra o seguinte comportamento:

1. A tarefa é iniciada e entra no estado *ready*;
2. A tarefa possui a maior prioridade entre as prontas para execução (tarefas *ready*), então é movida para o estado *running*;
3. A tarefa é bloqueada devido à requisição de algum recurso indisponível, então é movida para o estado *blocked*;
4. A tarefa que estava bloqueada é desbloqueada, mas como não possui a maior prioridade vai para o estado *ready*;
5. A tarefa que estava bloqueada é desbloqueada e como possui a maior prioridade começa a ser executada, estado *running*;
6. A tarefa volta para a fila de execução devido à presença de uma tarefa com maior prioridade, estado *ready*.

A Figura 2.4 mostra um exemplo de como o escalonador usa a lista de tarefas *ready* para mover as tarefas do estado *ready* para o *running*. O exemplo assume um sistema com um único processador e um algoritmo de escalonamento preemptivo baseado em prioridades, em que o 255 é a menor prioridade e o 0 é a maior. Neste exemplo, as tarefas 1, 2, 3, 4 e 5 estão prontas para serem executadas, e o RTOS as organiza em uma fila de acordo com a prioridade. A tarefa 1 possui a maior prioridade (70); as tarefas 2, 3 e 4 são as próximas na lista de prioridades (80); e a tarefa 5 é a da menor prioridade (90) [27].

Os seguintes passos explicam como o RTOS move as tarefas na lista [27]:

1. Tarefas 1, 2, 3, 4 e 5 estão prontas para serem executadas, e estão aguardando na lista *ready* ordenadas de acordo com suas prioridades.
2. Como a tarefa 1 possui a maior prioridade, ela é a primeira a ser executada. Se nada mais estiver sendo executado, o RTOS remove a tarefa 1 da lista *ready* e a tarefa recebe o estado de *running*.

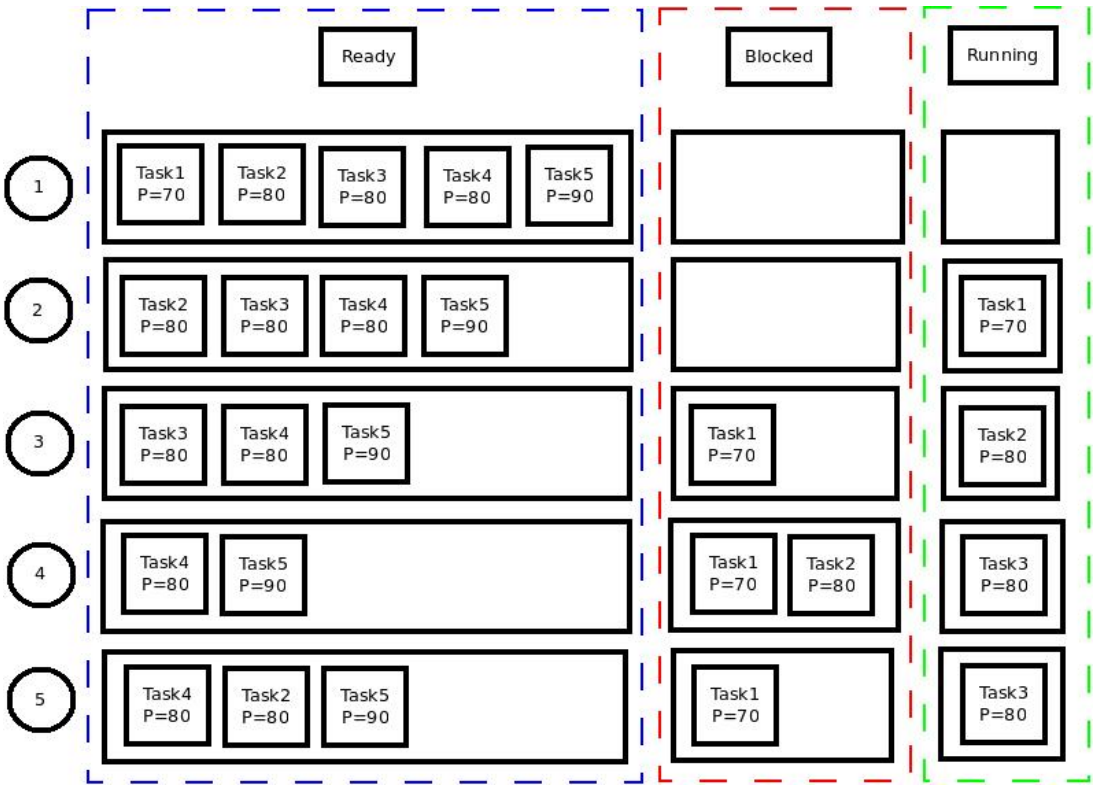


Figura 2.4: Exemplo de escalonamento de tarefas [27]

3. Durante a execução a tarefa 1 é bloqueada por requisitar recurso indisponível. Como resultado o RTOS move a tarefa para a lista *blocked*; a tarefa 2 que é a próxima da lista passa para o estado *running* e é removida da lista *ready*.

4. No próximo a tarefa 2 é bloqueada por requisitar recurso indisponível. O RTOS move a tarefa 2 para a lista *blocked*; a tarefa 3 que é a próxima da lista recebe o estado *running* e é removida da lista *ready*.

5. Durante a execução da tarefa 3 o recurso que a tarefa 2 requisitou é liberado. O RTOS retorna a tarefa 2 a lista *ready* e a organiza de acordo com a prioridade, antes de tarefa 5, e logo após tarefa 4. A tarefa 3 continua com o estado *running*

2.3.2 Semáforos

Um semáforo é um objeto do RTOS que uma ou mais tarefas podem adquirir ou liberar, com propósito de sincronizar e controlar o acesso a recursos. Recursos podem ser desde dispositivos de hardware (por exemplo, uma impressora) até uma informação (por exemplo, um registro em um banco de dados) [37] [36]. O semáforo é como uma chave que permite que uma determinada tarefa continue uma ope-

ração ou utilize um recurso. Se a tarefa conseguir adquirir o semáforo, ela poderá continuar a operação ou utilizar o recurso. Cada semáforo pode ser adquirido um número finito de vezes. Adquirir um semáforo é como adquirir uma cópia da chave de um apartamento, quando as chaves acabam, nenhuma mais pode ser adquirida. Do mesmo modo, quando um semáforo chega no seu limite, ele não pode mais ser adquirido até que alguém devolva uma chave ou libere o semáforo [27].

2.3.3 Fila de Mensagens

Uma fila de mensagens (*message queue*) é um objeto do RTOS, como um *buffer*, pelo qual tarefas enviam e recebem mensagens com fins de comunicação e sincronização. A fila de mensagens mantem a mensagem do emissor até que o receptor esteja pronto para recebe-la. Isso faz com que as tarefas não precisem enviar e receber a mensagem simultaneamente [27].

2.4 Exceções e Interrupções

Exceções e interrupções, também conhecidas como exceção síncrona e exceção assíncrona respectivamente, são parte do mecanismo da maior parte dos processadores embarcados. Através delas é possível romper o caminho normal de execução do processador. Este rompimento pode ser ativado intencionalmente pelo software da aplicação, por um erro ou por um evento externo não planejado. O que diferencia interrupções de exceções, ou mais precisamente o que diferencia exceção síncrona de exceção assíncrona, é a fonte do evento. A fonte da exceção síncrona é gerada internamente pelo processador devido a execução de alguma instrução. Por outro lado, a fonte da exceção assíncrona é um hardware externo [27].

2.5 Tick

Tick é uma variável utilizado pelo RTOS para medir o tempo, e é incrementada periodicamente através de uma interrupção. Sempre que ocorre a interrupção *tick* o RTOS verifica se é necessário reescalonar as tarefas. O reescalonamento pode ocorrer quando uma tarefa chega ao seu tempo limite de execução ou de espera de recursos. Se a tarefa desbloqueada durante a interrupção tiver prioridade menor que a tarefa que estava sendo executada, a tarefa desbloqueada irá para a lista de tarefas *Ready* e não ocorrerá reescalonamento [25].

2.6 Conclusão do Capítulo

Neste capítulo foi apresentada a teoria básica sobre sistemas operacionais de tempo real. Em mais detalhes esse capítulo abordou tópicos como algoritmos de escalonamento, interrupções e elementos comuns do RTOS como semáforos e fila de mensagens.

3 *Tolerância a Falhas*

3.1 Introdução

Tolerância a falhas é a habilidade de um circuito e/ou sistema continuar a execução correta das suas tarefas (sem degradação de desempenho), mesmo diante da ocorrência de falhas em seu hardware e/ou em software [26], evitando assim prejuízos físicos e materiais. Entre os conceitos relacionados aos sistemas tolerantes a falhas, podemos mencionar os descritos a seguir [26] [24] [7] [32]:

- Dependabilidade (*dependability*): é a qualidade de serviço provido por um sistema particular. Confiabilidade, disponibilidade, segurança, desempenhabilidade, mantenabilidade e testabilidade são exemplos de medidas usadas para quantificar a confiança de um sistema e são medidas quantitativas correspondentes a distintas percepções do mesmo atributo de um sistema.

- Confiabilidade (*reliability*): é uma função de tempo definida como a probabilidade que o sistema desempenha-se corretamente ao longo de um intervalo de tempo, onde o sistema tinha um desempenho correto no inicio do intervalo.

- Disponibilidade (*availability*): é uma função de tempo definida como a probabilidade que tem um sistema de operar corretamente, e estar disponível para desempenhar suas funções em um determinado intervalo de tempo.

- Segurança (*safety*): é a probabilidade de um sistema apresentar suas funções corretamente ou descontinuar suas funções, de maneira que não corrompa as operações de outros sistemas ou comprometa a segurança dos usuários do sistema.

- Desempenhabilidade (*performability*): em muitos casos, é possível projetar sistemas que possam continuar executando corretamente após a ocorrência de falhas de hardware ou software, mas o nível de desempenho de alguma maneira diminui.

- Mantenabilidade (*maintainability*): é uma medida da facilidade com que um sistema pode ser recuperado uma vez que apresentou defeito.

- Testabilidade (*testability*): é a habilidade de testar certos atributos de um sistema. Certos testes podem ser automatizados sob condição de integrar como parte do sistema e melhorar a testabilidade. A testabilidade está claramente relacionada à mantenabilidade, pela importância em minimizar o tempo exigido para identificação e localização de problemas específicos.

3.2 Falha, Erro e Defeito

Um resumo das definições de falha, erro e defeito são apresentados em [26] [24] [5] [23] [9] [7] [32] e mostrados abaixo.

- Falha: são causadas por fenômenos naturais de origem interna ou externa e ações humanas acidentais ou intencionais. Pode apresentar ocorrência tanto no âmbito de hardware quanto de software, sendo esta a causa do erro. Componentes envelhecidos e interferências externas são exemplos de fatores que podem levar o sistema à ocorrência de falhas.

- Erro: define-se que um sistema está em estado errôneo, ou em erro, se o processamento posterior a partir desse estado pode levar a um defeito. A causa de um erro é uma falha.

- Defeito: ocorre quando existe um desvio das especificações do projeto, esse não pode ser tolerado e deve ser evitado.

- Latência: período de tempo medido desde a ocorrência da falha até a manifestação da mesma.

A Figura 3.1 apresenta uma simplificação, sugerida por Dhiraj K. Pradhan [24], que explica a relação entre os conceitos definidos. Nela falhas estão associadas ao universo físico, erros ao universo da informação e defeitos ao universo do usuário.

Um exemplo para este modelo de três universos [24] seria um chip de memória, que apresenta uma falha do tipo *stuck-at-zero* em um de seus bits (falha no universo físico). Esta falha pode provocar uma interpretação errada da informação armazenada em uma estrutura de dados (erro no universo da informação). Como resultado deste erro, por exemplo, o sistema pode negar autorização de embarque para todos os passageiros de um voo (defeito no universo do usuário). É interessante observar que uma falha não necessariamente leva a um erro (pois a porção da memória sob falha pode nunca ser usada)

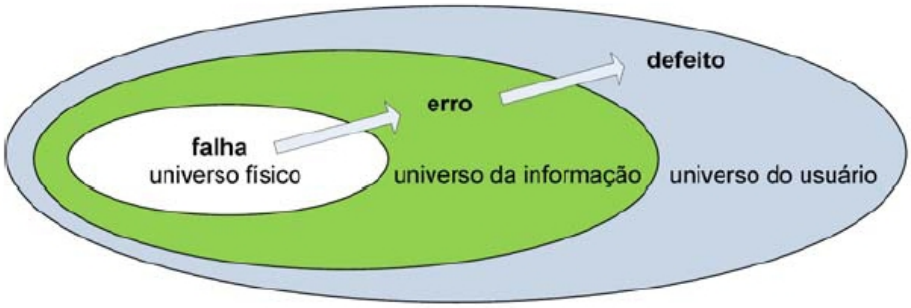


Figura 3.1: Relação entre falha, erro e defeito [24]

e um erro não necessariamente conduz a um defeito (no exemplo, a informação de voo lotado poderia eventualmente ser obtida a partir de outros dados redundantes da estrutura) [32].

Falhas são inevitáveis. Componentes físicos envelhecem e sofrem com interferências externas, sejam ambientais ou humanas. O software, e também os projetos de software e hardware, são vítimas de sua alta complexidade e da fragilidade humana em trabalhar com grande volume de detalhes ou com deficiências de especificação. Defeitos podem ser evitados usando-se técnicas de tolerância a falhas [32].

3.3 Tipos de Falhas

As falhas podem ser divididas em três categorias [43] [40] [7] [32] apresentadas a seguir.

1. Falhas de projeto: Resultam de erro humano, seja no projeto ou na especificação de um componente do sistema, resultando em parte na incapacidade de responder corretamente a certas entradas. A abordagem típica utilizada para detecção destes erros está baseada na verificação por simulação.

2. Falhas de fabricação: Resultam em uma gama de problemas no processamento, que se manifestam durante a fabricação. Os testes neste sistema são realizados adicionando-se hardware específico para teste.

3. Falhas operacionais: São caracterizadas pela sensibilidade do componente em condições ambientais.

Outro tipo de classificação de falhas segue a continuação [10] [7] [32]:

1. Falhas de Software. São causadas por especificações, projeto ou codificação incorreta de um programa. Embora o software “não falhe fisicamente” após ser instalado em um computador, falhas

latentes ou erros no código podem aparecer durante a operação. Isto pode ocorrer especialmente sob altas cargas de trabalho ou cargas não usuais para determinadas condições. Sendo assim, injeção de falhas por software são usadas principalmente para testes de programas ou mecanismos de tolerância a falhos implementados por software [32].

2. Falhas de Hardware. Ocorre durante a operação do sistema. São classificadas por sua duração:

   - Falhas permanentes. São causadas por defeitos de dispositivos irreversíveis em um componente devido a dano, fadiga ou manufatura imprópria. Uma vez ocorrida a falha permanente, o componente defeituoso pode ser restaurado somente pela reposição ou, se possível, pelo reparo.
   - Falhas transientes. São ocasionadas por distúrbios de ambiente tais como flutuações de voltagem, interferência eletromagnética ou radiação. Esses eventos tipicamente têm uma duração curta, sem causar danos ao sistema (embora o estado do sistema possa continuar errôneo). Falhas transientes podem ser até 100 vezes mais frequentes que falhas permanentes, dependendo do ambiente de operação do sistema [10]. As principais fontes de faltas transientes são radiação e interferência eletromagnética que geram principalmente evento único de perturbação (*Single-Event Upset* ou SEU), voltagem ou pulsos aleatórios de corrente [13].
   - Falhas intermitentes. Tendem a oscilar entre períodos de atividade errônea e dormência, podem permanecer durante a operação do sistema. Elas são geralmente atribuídas a erros de projeto que resultam em hardware instável.

3.4 Conclusão do Capítulo

Neste capítulo foi apresentado o tema de tolerância a falhas, incluindo a diferença entre falha, erro e defeito, e os tipos de falhas. Tolerância a falhas é um requisito importante em todos os projetos de sistemas computacionais. Os conceitos básicos da área devem estar bem difundidos entre os profissionais responsáveis pelo desenvolvimentos de equipamentos eletrônicos e softwares.

PARTE II - METODOLOGIA

4 *Descrição do Sistema Embarcado Utilizado*

4.1 Introdução

Neste capítulo será apresentado o sistema embarcado utilizado durante a realização deste trabalho. É importante salientar que foi fundamental que o sistema possuísse código "livre"e "aberto", desta forma foi possível fazer as modificações estruturais necessárias no processador para adaptação do RTOS-G proposto.

Cabe mencionar que a proposta descrita neste trabalho é genérica o bastante para ser aplicada a qualquer sistema embarcado, desde que haja informação suficiente sobre a arquitetura e organização do processador e que seu sistema operacional seja de domínio público, isto é, "aberto". Esta condição é obrigatória, pois o projetista do sistema embarcado deve ter acesso irrestrito a detalhes da implementação do RTOS, de forma a poder implementar o RTOS-G proposto.

4.2 Microprocessador Plasma

Assim, o microprocessador escolhido foi o Plasma CPU, que é um microprocessador RISC de 32bits implementado em VHDL (*VHSIC Hardware Description Language*), desenvolvido por Steve Rhoads e distribuído gratuitamente através do site Opencores. Ele possui controlador de interrupção, UART (*Universal asynchronous receiver/transmitter*), controlador SRAM (*Static Random Access Memory*) ou DDR SDRAM (*Double-Data-Rate Synchronous Dynamic Random Access Memory*) e controlador *Ethernet*. O processador executa todas as instruções MIPS I(TM) com exceção de operações load e store desalinhadas [35]. A figura 4.1 apresenta o diagrama de blocos da arquitetura básica do Plasma.

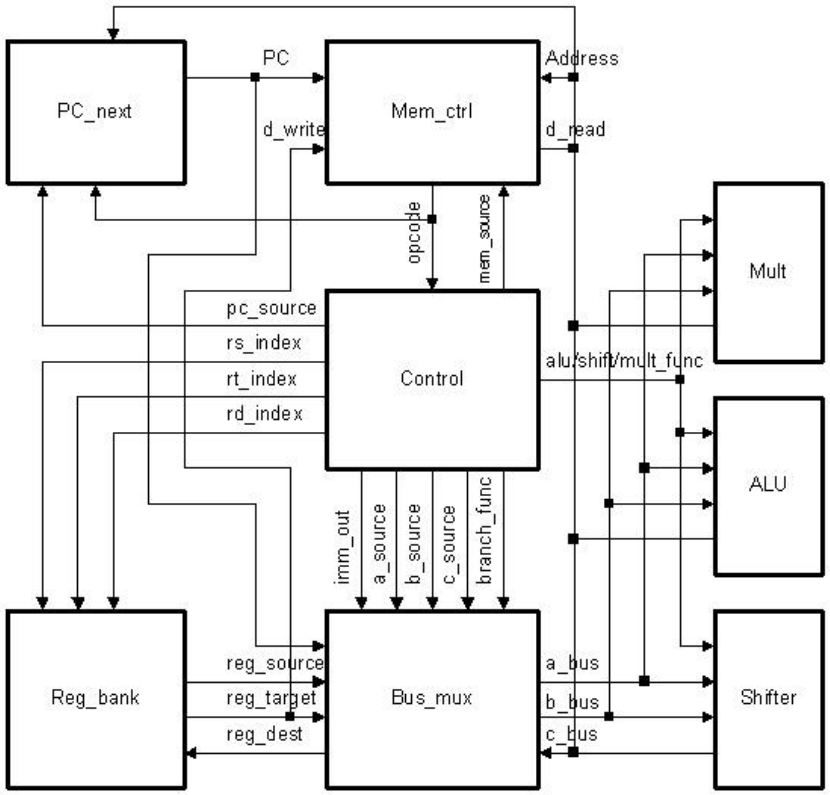


Figura 4.1: Diagrama de blocos da arquitetura básica do processador Plasma [35]

O microprocessador é implementado com pipeline de 2 ou 3 estágios, e um estágio adicional opcional para leitura e escrita de memória. A descrição abaixo descreve os passos que uma instrução de soma seguirá [35]:

- Estágio 0:
  1. A entidade "PC_next"passa o *program counter* para a entidade "Men_ctrl"que busca o *opcode* da memória.
- Estágio 1:
  1. A memória retorna o *opcode*.
- Estágio 2:
  1. "Men_ctrl"passa o *opcode* para a entidade "Control".
  2. "Control"converte o *opcode* de 32bits para um *opcode* VLWI (*Very Long Word Instruction*) de 60bits, e envia sinais de controle para outras unidades.
  3. Baseado nos sinais de controle *rsindex* e *rtindex*, "Reg_bank"envia *resource* e *regtarget* para o "Bus_mux".

4. Baseado nos sinais de controle *asource* e *bsource*, "Bus_mux"multiplexa *regsource* no *abus* e *regtarget* no *bbus*.

- Estágio 3: (parte do estágio 2 se estiver sendo usado pipeline de dois estágios)

  1. Baseado no sinal de controle *alufunc*, "ALU"soma os valores do *abus* e *bbus* e coloca o resultado no *cbus*.
  2. Baseado no sinal de controle *csource*, "Bus_mux"multiplexa *cbus* no *regdest*.
  3. Baseado no sinal de controle *rdindex*, "Reg_bank"salva *regdest* no registrador correto.

- Estágio 4: (parte do estágio 3 se estiver sendo usado pipeline de dois estágios)

  1. Se necessário lê ou escreve na memória.

4.3 Sistema Operacional de Tempo Real - Plasma RTOS

O RTOS do microprocessador Plasma é escrito em linguagem C, suporta semáforos, *mutex*, fila de mensagens, *timers*, *heaps* e possui escalonador baseado em prioridades. Cada tarefa possui 3 estados: *"Running*, *"blocked"* e *"ready"*, onde as tarefas são organizadas de acordo com suas prioridades.

4.4 Detecção de Falhas do RTOS nativo do Plasma

A detecção de falhas no RTOS é realizada através de funções denominadas *assert( )*. No caso em que o argumento da função *assert* apresentar um valor falso, o RTOS envia uma mensagem através da comunicação serial, indicando o número de linha do código fonte onde se encontra aquela função *assert*. Desta maneira, o RTOS só informa se algum valor errado foi produzido pelo RTOS [32].

A tabela 4.1 mostra os argumentos que são validados pelas funções *assert* do RTOS. Entre os argumentos, destacam-se aqueles utilizados para validar a coerência de dados entre processos (*mutex*, *mQueue*, *timer*, *block*, *ThreadHead*, *thread*, *semaphore*). Outros argumentos como "$thread \rightarrow magic(0) == Thread\_Magic$"são usados na verificação de *overflows* de memória [32].

Tabela 4.1: Argumentos da função *assert* do RTOS nativo para detecção de falhas [32].

Argumento
$((uint32)memory\&3) == 0$
$heap \rightarrow magic == Heap\_Magic$
$thread \rightarrow magic[0] == Heap\_Magic$
$threadCurrent \rightarrow magic[0] == Thread_{M}agic$
$threadNext \rightarrow state == Thread_{R}eady$
$InterruptInside[OS_{C}puIndex()] == 0$
$mutex \rightarrow thread == OS\_ThreadSelf()$
$mutex \rightarrow count > 0$
$SpinLockArray[cpuIndex] < 10$
$ThreadHead$
$thread$
$semaphore$
$mutex$
$mQueue$
$timer$
$Block$

4.5 Conclusão do Capítulo

Neste capítulo foi apresentado o sistema escolhido para implementação e validação do RTOS-G proposto, o processador e o RTOS Plasma. Foi exposto o hardware do processador e seus estágios do *pipeline*, e o mecanismo de detecção falhas nativo do Plasma RTOS.

5 *Proposta*

5.1 Introdução

Neste capítulo é apresentada a implementação do módulo proposto capaz de detectar falhas em sistemas embarcados baseados em RTOS que adotam escalonamento preemptivo. A proposta genérica para desenvolvimento do RTOS-G é descrita abaixo:

1. Escolha da CPU e do RTOS. É fundamental que o RTOS e a CPU possuam código "livre"e "aberto";
2. Estudo das funções responsáveis pelo escalonamento das tarefas;
3. Implementação do fluxograma que descreve o fluxo das funções que executam o processo de escalonamento das tarefas;
4. Simplificação do fluxograma acima descrito, ou seja, deve ser possível identificar todos os caminhos que geram reescalonamento com o menor número de funções possíveis;
5. Migração do fluxograma simplificado para hardware (implementação do RTOS-G);
6. Integração do RTOS-G ao sistema, ou seja, estanciamento do núcleo RTOS-G no barramento do processador/memória;
7. Validação do sistema através de simulação e avaliação da eficiência do mesmo em termos de detecção de falhas através de procedimentos de injeção de falhas.

A figura 5.1 apresenta a visão geral da arquitetura de um *System-on-Chip* (SoC) contendo o RTOS-G proposto. O RTOS-G é conectado ao barramento de endereços da RAM do sistema embarcado, e também recebe as seguintes informações: *Start*, *Tick* e um sinal de interrupção externa. Em mais detalhes o RTOS-G é composto de cinco módulos funcionais, a seguir é explicado com mais detalhes cada módulo.

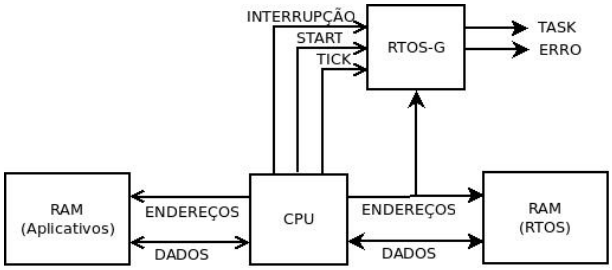


Figura 5.1: Visão geral da arquitetura de um *System-on-Chip* contendo o RTOS-G proposto

5.2 Arquitetura do RTOS-G

O núcleo RTOS-G é composto por módulos funcionais denominados TC (*Task Checker*), FC (*Flow Checker*) e LCEG (*List Checker and Error Generator*), e duas memórias conforme a figura 5.2.

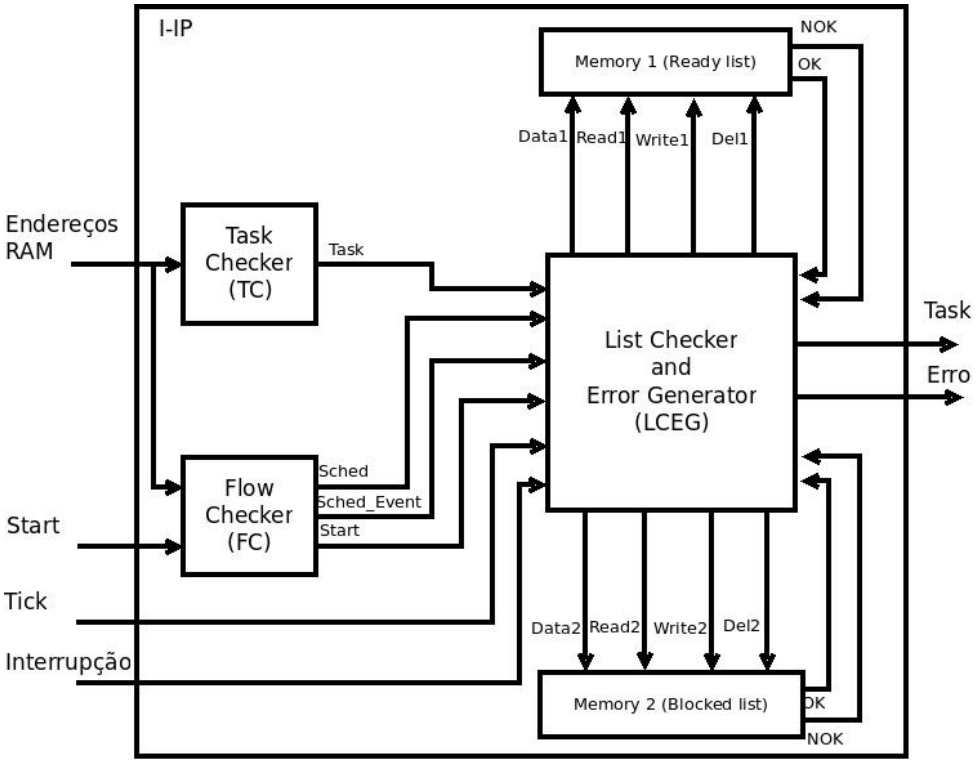


Figura 5.2: Arquitetura interna do RTOS-G proposto

O módulo TC é responsável pela identificação da tarefa em execução através do monitoramento dos endereços acessados pelo processador, isto é possível pois cada uma das tarefas está associada a endereços previamente definidos e conhecidos na memória, isto é, para cada tarefa, há um espaço de endereçamento previamente alocado em memória. O módulo TC envia um sinal chamado *Task* para o módulo LCEG indicando a tarefa que está sendo executada.

O módulo FC possui o fluxograma simplificado do fluxo de funções do RTOS que são responsáveis

pelo escalonamento das tarefas, desta forma o módulo FC indica os eventos de escalonamento, liberação ou bloqueio de recursos e a troca de estados das tarefas. O monitoramento das funções do RTOS é feito da mesma maneira implementada pelo módulo TC, pois cada função está associada a endereços previamente definidos e conhecidos na memória. Em mais detalhes o módulo inicia o monitoramento no momento que recebe o sinal *Start* que é enviado pelo Plasma CPU, através de um pino de I/O, no momento que a função OS_Start é executada. A função OS_Start é executada após a inicialização e criação das tarefas do RTOS, ela inicia o processo de escalonamento. Os sinais de saída do módulo FC são:

- Sched: Indica os eventos de escalonamento;
- Sched_event: Envia um pulso a cada evento do sinal *Sched*;
- Start: Repassa o sinal *Start* enviado pelo Plasma CPU.

O Módulo LCEG recebe os eventos de escalonamento e a tarefa em execução dos blocos anteriores. Com estas informações ele organiza as tarefas de acordo com seus estados e prioridades, separando as tarefas em duas listas: “tarefas prontas” (*ready*) e “tarefas bloqueadas” (*blocked*), que são armazenadas nos módulos de memória 1 e 2 respectivamente. O módulo LCEG também recebe do processador os sinais de interrupção e *tick*, que são utilizados para verificação de erros. Os sinais de saída do módulo LCEG são:

- Task: Indica a tarefa em execução;
- Erro: Indica os erros detectados pelo RTOS-G.

O RTOS-G possui duas memórias onde são armazenadas as tarefas e seus determinados estados e prioridades. Os de controle da memória são descritos abaixo:

- Data: Dados que serão lidos ou escritos na memória;
- Read: Habilita operação de leitura;
- Write: Habilita operação de escrita;
- Del: Permite a memória apagar um determinado dado;
- Ok: Sinal que indica que a operação foi executada com sucesso;

- NOk: Sinal que indica que a operação falhou. Pode ser por um erro, dado não encontrado ou memória cheia.

A seguir cada um dos módulos será explicado com mais detalhes. Todas as informações necessárias para o monitoramento do RTOS, como endereços das funções e tarefas, são passadas ao RTOS-G no momento da sintetização do módulo.

5.2.1 Módulo Task Checker (TC)

Identifica a tarefa em execução a partir do endereço acessado na memória. A cada ciclo de relógio o endereço acessado na memória pela CPU é comparado com a faixa de endereço referente a cada tarefa. Se o endereço acessado for igual ao endereço de alguma tarefa, o módulo TC indicará qual a tarefa está sendo executada e passará esta informação para o módulo LCEG. A tarefa indicada pelo módulo é sempre a última a ser acessada na memória, caso o endereço acessado não seja de nenhuma tarefa da lista do módulo, ele continuará indicando a última tarefa executada. A figura 5.3 apresenta o diagrama interno do Módulo TC.

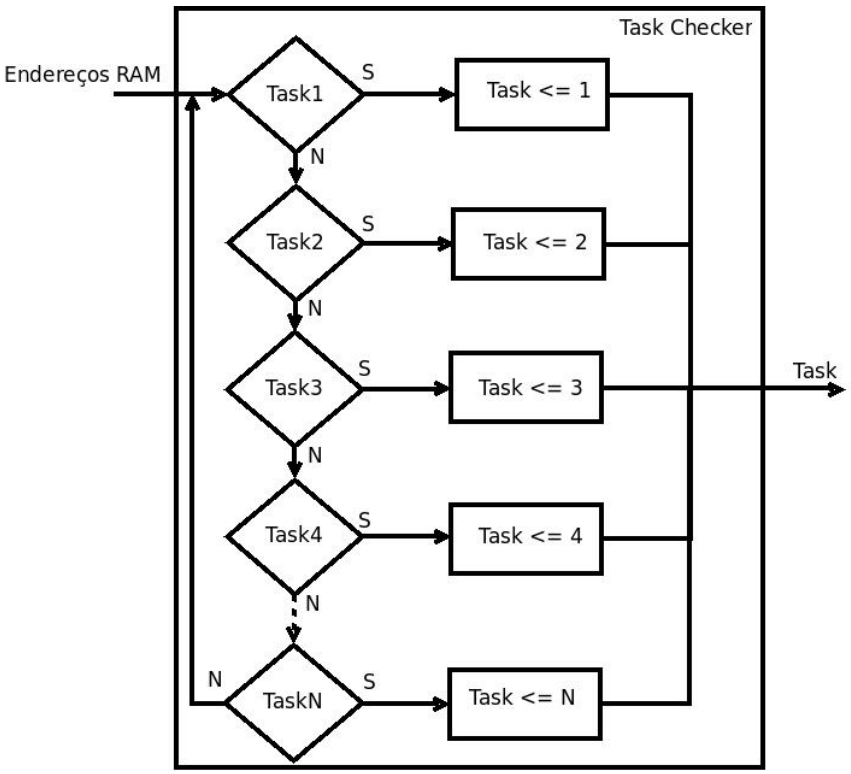


Figura 5.3: Arquitetura interna do Módulo *Task Checker*

5.2.2 Módulo Flow Checker (FC)

Monitora e analisa as funções do RTOS referentes ao escalonamento das tarefas. O módulo FC identifica os eventos de escalonamento, a liberação ou bloqueio de recursos e a troca de estado das tarefas. Estas informações são enviadas ao módulo LCEG. A seguir é apresentada a lista de funções do Plasma RTOS que fazem parte do processo de reescalonamento das tarefas:

- OS_SemaphorePend: Adquire semáforo;
- OS_SemaphorePost: Libera semáforo;
- OS_ThreadPriorityInsert: Adiciona uma tarefa a uma lista e ordena as tarefas da lista de acordo com a prioridade;
- OS_ThreadPriorityRemove: Remove uma tarefa de uma lista e reorganiza a lista de acordo com a prioridade;
- OS_ThreadTimeoutInsert: Adiciona uma tarefa a lista *timeout*, que são tarefas aguardando por um evento de tempo pré-definido;
- OS_ThreadTimeoutRemove: Remove a tarefa da lista *timeout*;
- OS_InterruptServiceRoutine: Esta função é executada toda vez que ocorre uma interrupção. Ela chama a função que é programada para ser executada durante a interrupção;
- OS_ThreadTickToggle: Interrupção *tick*. Esta tarefa é executada cada vez que ocorre um *tick*. Ela irá alterar as *flags* de controle da interrupção e chamará a função OS_ThreadTick;
- OS_ThreadTick: Incrementa a variável de controle de tempo do RTOS e verifica se alguma tarefa que está aguardando por um tempo definido alcançou seu *timeout*;
- OS_ThreadReschedule: Reescalona as tarefas. Esta função é executada quando um recurso é liberado ou bloqueado, e quando ocorre o sinal de *tick*. Ela verifica se a tarefa em execução é a de maior prioridade da lista *ready*. Durante um evento de *tick* pode acontecer um reescalonamento se houver uma tarefa com prioridade igual à tarefa em execução;
- Longjmp: Altera o registrador do processador incluindo o *stack-pointer* para a próxima tarefa que será executada. Está função é executada no final da função OS_ThreadReschedule se acontecer reescalonamento;

O Módulo FC possui dois processos. O primeiro é a identificação da função em execução de acordo com o endereço acessado na memória, da mesma forma que o módulo TC. A informação da função sendo executada, é passada ao segundo processo do módulo, onde é analisado o fluxo das funções. A figura 5.4 apresenta o fluxograma do processo de identificação de eventos do módulo FC. O processo tem como saída dois sinais. O *Sched* indica qual evento ocorreu e o *sched_event* gera um pulso a cada evento. Os eventos indicados são os listados abaixo:

- (001): Reescalonamento devido a recurso indisponível;
- (010): Reescalonamento devido a recurso liberado;
- (011): Recurso liberado;
- (100): Reescalonamento por *Tick*;
- (101): Evento *Timeout*;
- (110): Evento *Tick*.

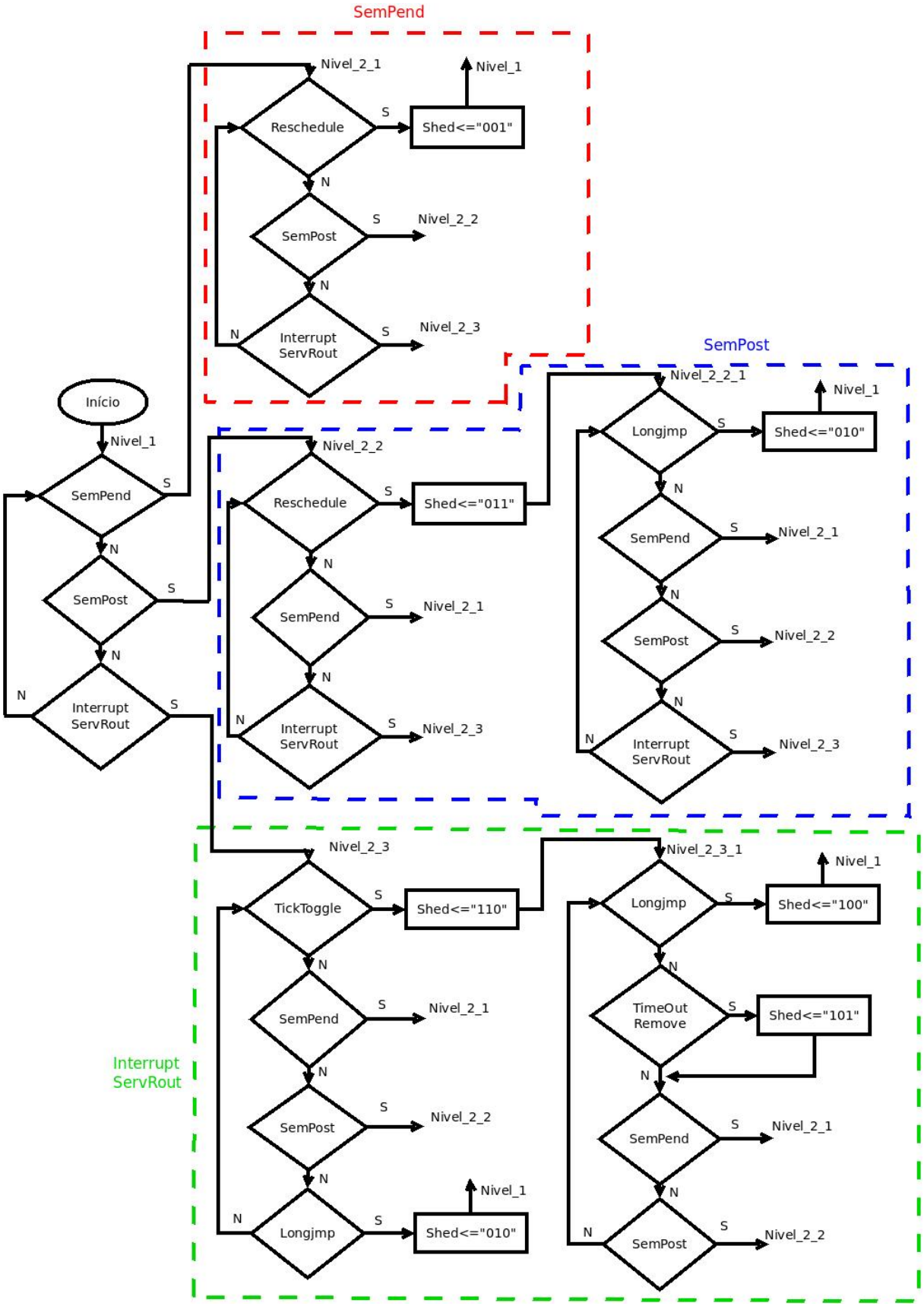


Figura 5.4: Diagrama de blocos do módulo *Flow Checker*

As tabelas a seguir apresentam o fluxo de funções do RTOS em diferentes eventos e como o módulo FC trata cada evento. Os fluxos de funções descritos abaixo mostram a visão do RTOS-G através do monitoramento do barramento de endereços da memória. Quando várias funções são chamadas apartir da mesma função, o processador retorna ao endereço da primeira função para chamar a função seguinte, isso causa a repetição da primeira função antes de executar a próxima função do fluxo.

Tabela 5.1: Fluxo de funções quando uma tarefa tenta adquirir um recurso que não está disponível.

#	Plasma RTOS	Módulo Flow Checker (figura 5.4)	Explicação
1	OS_SemaphorePend	Vai para o nível 2_1.	Verifica se o semáforo está disponível.
2	OS_ThreadPriorityInsert		Tarefa que tentou adquirir o semáforo é adicionada a lista *blocked*.
3	OS_SemaphorePend		*
4	OS_ThreadTimeoutInsert		Tarefa é adicionada a lista *timeout*.
5	OS_SemaphorePend		*
6	OS_ThreadReschedule	Indica evento ("001") e volta para o nível 1.	Defini a próxima tarefa a ser executada.
7	OS_ThreadPriorityRemove		Remove a tarefa escolhida da lista *ready*.
8	OS_ThreadReschedule		*
9	longjmp		Reescalona as tarefas.

* Retorna à função anterior para chamar a seguinte.

Se o semáforo estivesse disponível, após a chamada da função OS_SemaphorePend, o RTOS continuaria a execução da tarefa. Nenhuma das funções listadas na tabela 5.1 seriam executadas. O módulo FC continuaria no nível 2_1 e ficaria aguardando pelos próximos eventos. Caso a tarefa tentasse adquirir outro semáforo o módulo FC já estaria no nível 2_1 e ficaria apenas aguardando as funções seguintes. Se acontecer outro evento como uma interrupção ou liberação de semáforo, o módulo irá para o nível referente a cada caso.

Tabela 5.2: Fluxo de funções quando um recurso é liberado e o mesmo gera reescalonamento.

#	Plasma RTOS	Módulo Flow Checker (figura 5.4)	Explicação
1	OS_SemaphorePost	Vai para o nível 2_2.	Tarefa solicita a liberação do semáforo.
2	OS_ThreadTimeoutRemove		Se houver uma tarefa aguardando pelo semáforo ela será removida da lista de *timeout*.
3	OS_SemaphorePost		*
4	OS_ThreadPriorityRemove		Se houver uma tarefa aguardando pelo semáforo ela será removida da lista de *blocked*.
5	OS_SemaphorePost		*
6	OS_ThreadPriorityInsert		Se houver uma tarefa aguardando pelo semáforo ela será adicionada à lista de *ready*.
7	OS_SemaphorePost		*
8	OS_ThreadReschedule	Indica evento ("011") e vai para o nível 2_2_1	Verifica se a tarefa em execução continua sendo a de maior prioridade da lista *ready*.
9	OS_ThreadPriorityInsert		Tarefa em execução é adicionada à lista *ready*.
10	OS_ThreadReschedule		*
11	OS_ThreadPriorityRemove		Remove a tarefa de maior prioridade da lista *ready*.
12	OS_ThreadReschedule		*
13	longjmp	Indica evento ("010") e volta ao nível 1	Reescalona as tarefas.

* Retorna à função anterior para chamar a seguinte.

Conforme o fluxo de funções da tabela 5.2, o módulo FC indica dois eventos. O primeiro é a liberação de recurso, e o segundo o reescalonamento em razão do recurso liberado.

Tabela 5.3: Fluxo de funções quando um recurso é liberado e não gera reescalonamento.

#	Plasma RTOS	Módulo Flow Checker (figura 5.4)	Explicação
1	OS_SemaphorePost	Vai para o nível 2_2.	Tarefa solicita a liberação do semáforo.
2	OS_ThreadTimeoutRemove		Se houver uma tarefa aguardando pelo semáforo ela será removida da lista de *timeout*.
3	OS_SemaphorePost		*
4	OS_ThreadPriorityRemove		Se houver uma tarefa aguardando pelo semáforo ela será removida da lista *blocked*.
5	OS_SemaphorePost		*
6	OS_ThreadPriorityInsert		Se houver uma tarefa aguardando pelo semáforo ela será adicionada à lista *ready*.
7	OS_SemaphorePost		*
8	OS_ThreadReschedule	Indica evento ("011") e vai para o nível 2_2_1	Verifica se a tarefa em execução continua sendo a de maior prioridade da lista *ready*.
9	OS_SemaphorePost	Vai para o nível 2_2	Encerra a função e retorna a executar a tarefa atual.

* Retorna à função anterior para chamar a seguinte.

A diferença do fluxo apresentado na tabela 5.3 para tabela 5.2, está na execução da função OS_ThreadReschedule, durante a verificação da lista *ready*, não há nenhuma tarefa com prioridade maior que a tarefa em execução. O módulo FC apenas indicará recurso liberado.

Tabela 5.4: Interrupção *Tick* com liberação de recursos devido a *timeout* e reescalonamento.

#	Plasma RTOS	Módulo Flow Checker (figura 5.4)	Explicação
1	OS_InterruptServiceRoutine	Vai para o nível 2_3.	Interrupção.
2	OS_ThreadTickToggle	Indica evento ("110") e vai para o nível 2_3_1	Interrupção *tick*.
3	OS_ThreadTick		Incrementa o contador de tempo e verifica ocorrência de *timeout*
4	OS_ThreadTimeoutRemove	Indica evento ("101") e continua no nível 2_3_1	Tarefa é liberada por *timeout*.
5	OS_ThreadTick		*
6	OS_ThreadPriorityRemove		Remove a tarefa liberada da lista *blocked*.
7	OS_ThreadTick		*
8	OS_ThreadPriorityInsert		Adiciona a tarefa liberada à lista *ready*.
9	OS_ThreadTick		*
10	OS_ThreadTimeoutRemove	Indica evento ("101") e continua no nível 2_3_1	Outra tarefa é liberado por *timeout*.
11	OS_ThreadTick		*
12	OS_ThreadPriorityRemove		Remove a tarefa liberada da lista *blocked*.
13	OS_ThreadTick		*
14	OS_ThreadPriorityInsert		Adiciona a tarefa liberada à lista *ready*.
15	OS_ThreadTick		*
16	OS_ThreadReschedule		Como RTOS está tratando uma interrupção, apenas será indicado a necessidade de verificação de reescalonamento.
17	OS_InterruptServiceRoutine		Encerra a interrupção e chama a função de reescalonamento.
18	OS_ThreadReschedule		Verifica se a tarefa em execução continua sendo a de maior prioridade da lista *ready*.
19	OS_ThreadPriorityInsert		Tarefa em execução é adicionada à lista *ready*.
20	OS_ThreadReschedule		*
21	OS_ThreadPriorityRemove		Remove a tarefa de maior prioridade da lista ready.
22	OS_ThreadReschedule		*
23	longjmp	Indica evento ("100") e volta ao nível 1	Reescalona as tarefas.

* Retorna à função anterior para chamar a seguinte.

Conforme a tabela 5.4, mais de uma tarefa pode ser desbloqueada por *timeout* no mesmo *tick*. O módulo FC identifica estes eventos pelo número de vezes que a função OS_ThreadTimeoutRemove é executada. Sempre que o RTOS estiver tratando uma interrupção, ele não irá reescalonar as tarefas, ape-

nas irá indicar a necessidade de verificação de reescalonamento através da variável *ThreadNeedReschedule*. No fim da função OS_InterruptServiceRoutine é verificado se a variável *ThreadNeedReschedule* é verdadeira, se for, o RTOS irá chamar a função de reescalonamento.

Tabela 5.5: Interrupção externa com liberação de recursos.

#	Plasma RTOS	Módulo Flow Checker (figura 5.4)	Explicação
1	OS_InterruptServiceRoutine	Vai para o nível 2_3.	Interrupção.
2	OS_SemaphorePost	Vai para o nível 2_2	Recurso será liberado.
3	OS_ThreadTimeoutRemove		Se houver uma tarefa aguardando pelo semáforo ela será removida da lista de *timeout*.
4	OS_SemaphorePost		*
5	OS_ThreadPriorityRemove		Se houver uma tarefa aguardando pelo semáforo ela será removida da lista *blocked*.
6	OS_SemaphorePost		*
7	OS_ThreadPriorityInsert		Se houver uma tarefa aguardando pelo semáforo ela será adicionada à lista de *ready*.
8	OS_SemaphorePost		*
9	OS_ThreadReschedule	Indica evento ("011") e vai para o nível 2_2_1	Como RTOS está tratando uma interrupção, apenas será indicado a necessidade de verificação de reescalonamento.
10	OS_SemaphorePost	Vai para o nível 2_2	Encerra a função semáforo.
11	OS_InterruptServiceRoutine	Vai para o nível 2_3	Encerra a interrupção e chama a função de reescalonamento.
12	OS_ThreadReschedule		Verifica se a tarefa em execução continua sendo a de maior prioridade da lista *ready*.
13	OS_ThreadPriorityInsert		Tarefa em execução é adicionada à lista *ready*.
14	OS_ThreadReschedule		*
15	OS_ThreadPriorityRemove		Remove a tarefa de maior prioridade da lista *ready*.
16	OS_ThreadReschedule		*
17	longjmp	Indica evento ("010") e volta ao nível 1	Reescalona as tarefas.

* Retorna à função anterior para chamar a seguinte.

No fluxo da tabela 5.5, a interrupção envia uma mensagem a outra tarefa, fazendo com que um semáforo seja liberado. A liberação do semáforo gera um reescalonamento que é realizado após o fim da interrupção. Quando uma interrupção é executada e não gera nenhum evento de reescalonamento, o RTOS retorna a execução do ponto onde ocorreu a interrupção, e o módulo FC não indicará eventos.

5.2.3 Módulo List Checker and Error Generator (LCEG)

O módulo LCEG recebe os eventos de escalonamento e a tarefa em execução dos módulos TC e FC. Com estas informações ele organiza as tarefas de acordo com seus estados e prioridades em duas listas: *Ready* e *blocked*. Cada lista é armazenada em uma memória, a memória 1 possui a lista ready e a memória 2 a lista *blocked*. A tarefa em execução (*running*) é a indicada pelo módulo TC. Quando uma tarefa entra em execução ela não é removida da lista *ready*.

Note que este monitor não é capaz de identificar qual o semáforo que está sendo adquirido ou liberado, logo não é possível saber qual semáforo as tarefas bloqueadas estão aguardando. Para resolver este problema, é feita a contagem de tarefas bloqueadas, recursos liberados e tarefas liberadas por *timeout*. Cada vez que uma tarefa for bloqueada serão incrementadas as variáveis *tasksblocked* e *resourceblocked*. Quando uma tarefa é liberada por *timeout* ou um recurso é liberado, a variável *resourceblocked* é decrementada. Quando ocorrer um reescalonamento e o RTOS-G detectar que a tarefa que entrou em execução está na lista *blocked*, o RTOS-G irá comparar as variáveis *tasksblocked* e *resourceblocked*. Se a variável *resourceblocked* for menor que a *tasksblocked*, significa que alguma tarefa foi liberada seja por *timeout* ou por recursos liberados, logo uma tarefa que está na lista *blocked*, pode vir a ser executada.

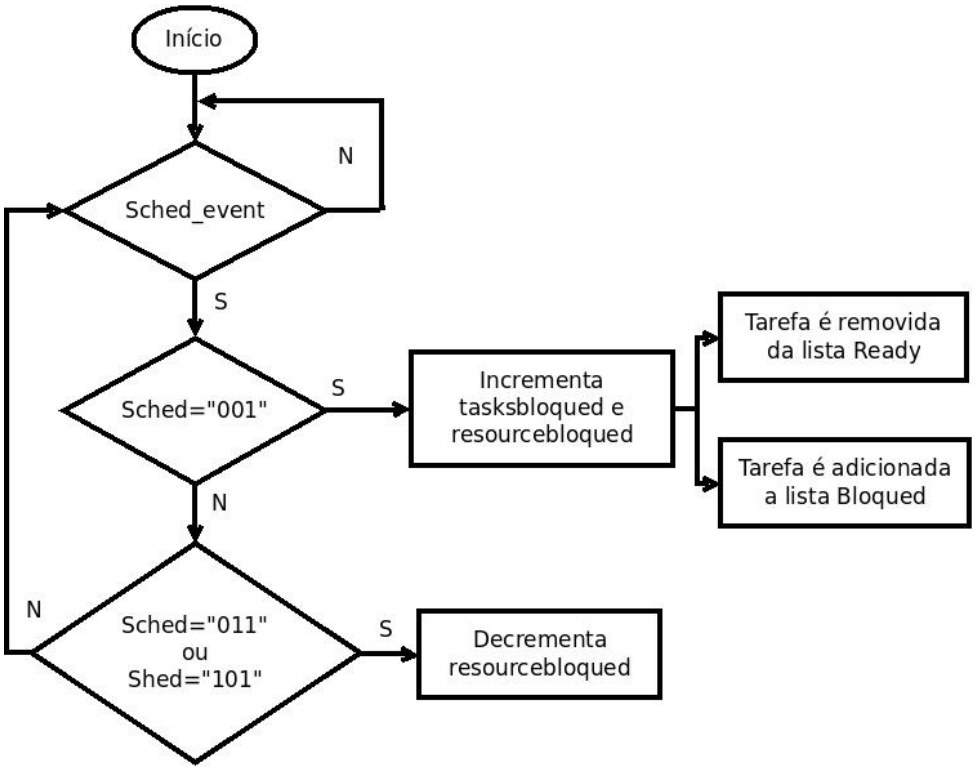


Figura 5.5: Primeiro estágio módulo LCEG: Fluxograma de controle das variáveis *tasksblocked* e *resourceblocked*.

A figura 5.5 apresenta o fluxograma de controle das variáveis *tasksblocked* e *resourceblocked*. Cada vez que um evento for indicado pelo módulo FC, será verificado se ocorreu algum destes eventos: Recurso bloqueado, recurso liberado e *timeout*. O sinal de recurso bloqueado (Sched="001") indica que a tarefa em execução tentou adquirir um recurso indisponível e será bloqueada. Neste caso o módulo irá incrementar o contador *tasksblocked* e *resourceblocked*, e a tarefa em execução será movida da lista *ready* (memória 1) para a lista *blocked* (memória 2). Quando um recurso é liberado ou ocorre um *timeout*, apenas o contador *resourceblocked* será decrementado, nada será alterado nas listas já que o RTOS-G não consegue identificar qual recurso foi liberado, logo não pode identificar qual tarefa foi desbloqueada. O *resourceblocked* sendo menor que o *taskblocked*, indica que alguma tarefa da lista *blocked* deveria estar na lista *ready*.

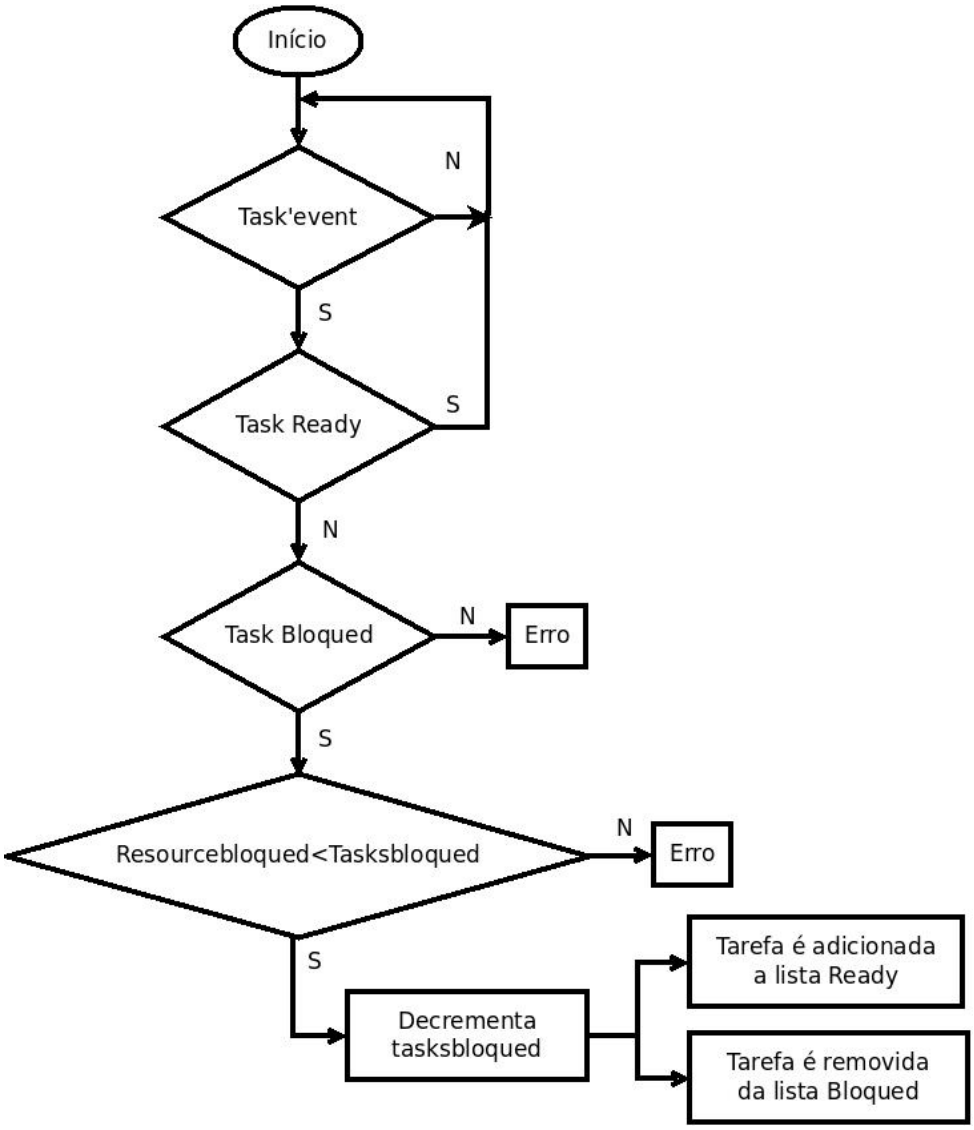


Figura 5.6: Segundo estágio módulo LCEG: Fluxograma de controle do estado da tarefa em execução.

A figura 5.6 apresenta o fluxograma de verificação do estado da tarefa em execução. A cada rees-

calonamento de tarefas, é verificado se a nova tarefa em execução está na lista *ready*, se estiver o teste é encerrado. Se a tarefa não estiver na lista *ready*, será verificado a lista *blocked*. Caso a tarefa não esteja em nenhuma lista será indicado que ocorreu um erro. Mas se a tarefa estiver na lista de *blocked*, será verificado se a variável *resourceblocked* é menor que a *tasksblocked*, se for verdadeiro então algum recurso já foi liberado, logo a tarefa pode entrar em execução. Neste caso a variável *tasksblocked* é decrementada e a tarefa é movida da lista *blocked* para *ready*. Se a variável *resourceblocked* não for menor que a *tasksblocked* será indicado um erro.

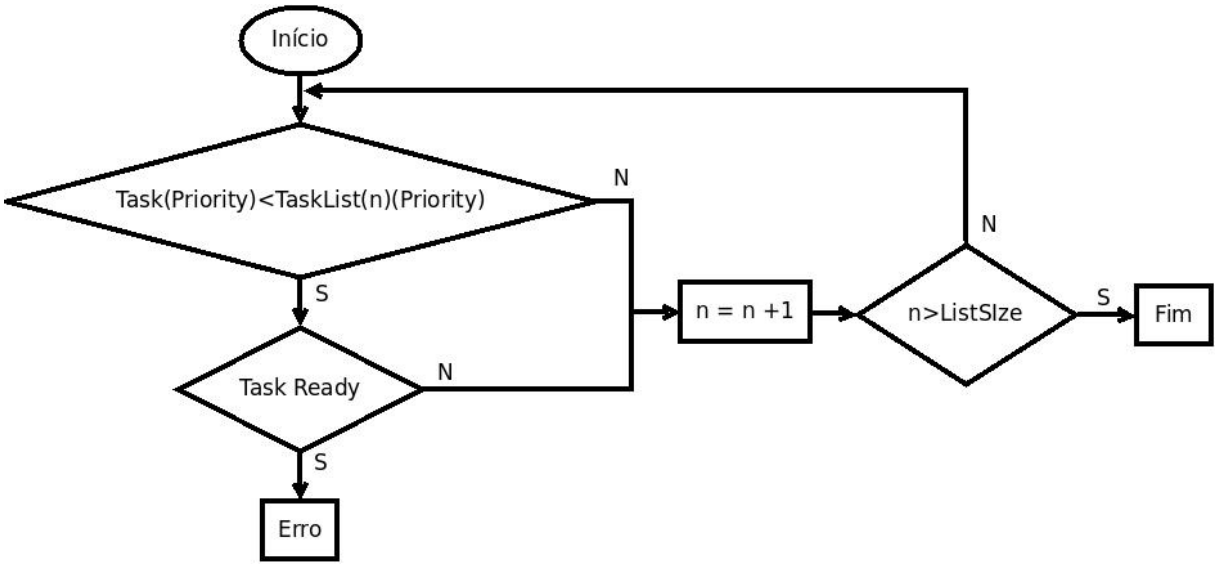


Figura 5.7: Terceiro estágio módulo LCEG: Fluxograma de controle da prioridade da tarefa em execução.

A figura 5.7 apresenta o fluxograma de verificação da prioridade da tarefa em execução. Este estágio verifica se a tarefa em execução é a de maior prioridade da lista *ready*. Esta verificação é feita após a execução do segundo estágio (figura 5.6). O módulo LCEG possui uma lista geral com a prioridade de todas as tarefas do RTOS (*ready* e *blocked*). O terceiro estágio irá comparar a prioridade da tarefa em execução, com a prioridade das tarefas presentes na lista geral do módulo LCEG. Toda vez que for encontrado uma tarefa com prioridade maior que a tarefa em execução, será verificado se a tarefa de maior prioridade está na lista *ready*, se estiver será indicado um erro. O teste termina quando ocorre um erro ou o terceiro estágio passa por toda a lista geral. Este estágio executa uma verificação toda vez que ocorre reescalonamento.

Os próximos estágios são muito similares, estes estágios setam *flags* para controle dos eventos de *tick*, interrupção e reescalonamento. O quarto estágio (figuras 5.8) faz o controle dos eventos de *tick*. Este estágio recebe o *tick* do processador Plasma, e o evento de *tick* indicado pelo módulo FC (Sched="110"). Quando ocorre o sinal de *tick* do Plasma a variável Tick_control será setada em 1, quando o modulo FC

indica um evento de *tick* a variável *Tick_control* é setada em 0. O erro é indicado quando o módulo FC indica o evento de *tick* e o *Tick_control* está em 0, isto indica que o RTOS gerou um reescalonamento por *tick* sem a ocorrência do sinal *tick* do Plasma. Ou quando ocorre um *tick* no Plasma e a variável já está em 1, isto indica que o RTOS não executou uma interrupção por *tick*.

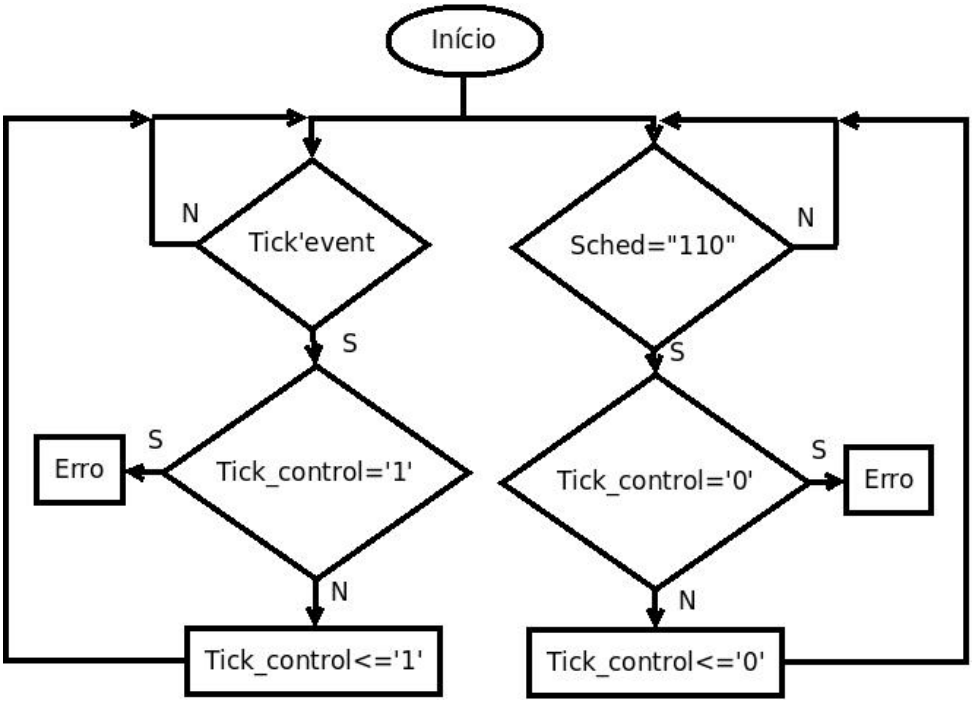


Figura 5.8: Quarto estágio módulo LCEG: Fluxograma de controle do reescalonamento por *tick*.

O quinto estágio (figura 5.9) monitora a interrupção externa. Cada vez que o módulo recebe o sinal da interrupção externa a variável *Int_control* recebe 1, quando a tarefa interrupção é executada pelo RTOS, a variável recebe 0. Erros são indicados quando a interrupção é executada pelo RTOS sem que o sinal da interrupção tenha sido recebido e quando o módulo recebe o sinal externo da interrupção mas a variável *Int_control* já está em 1, indicando que uma interrupção anterior não foi executada.

O sexto estágio (figura 5.10) verifica se o RTOS está reescalonando as tarefas de acordo com os eventos de reescalonamento indicados pelo módulo FC. Cada vez que é indicado reescalonamento por recurso indisponível (Sched="001"), recurso liberado (Sched="010") ou *tick* (Sched="100"), a variável *Shed_control* recebe 1. Quando ocorre uma troca de tarefas a variável *Shed_control* recebe 0. Os erros são indicados quando ocorre uma troca de tarefas sem que a variável *Sched_control* esteja em 1, isto indica que o RTOS alterou a tarefa em execução em que tenha ocorrido algum evento de reescalonamento. Ou quando um reescalonamento é indicado mas a variável *Sched_control* já está em 1, isto indica que algum evento de reescalonamento não foi executado.

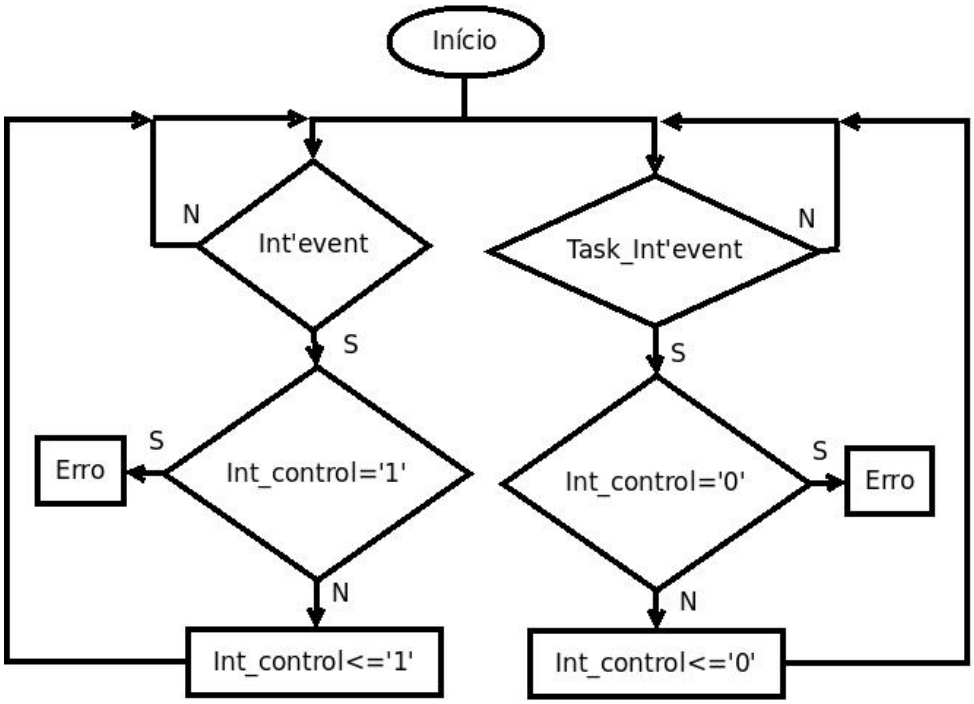


Figura 5.9: Quinto estágio módulo LCEG: Fluxograma de controle da interrupção externa.

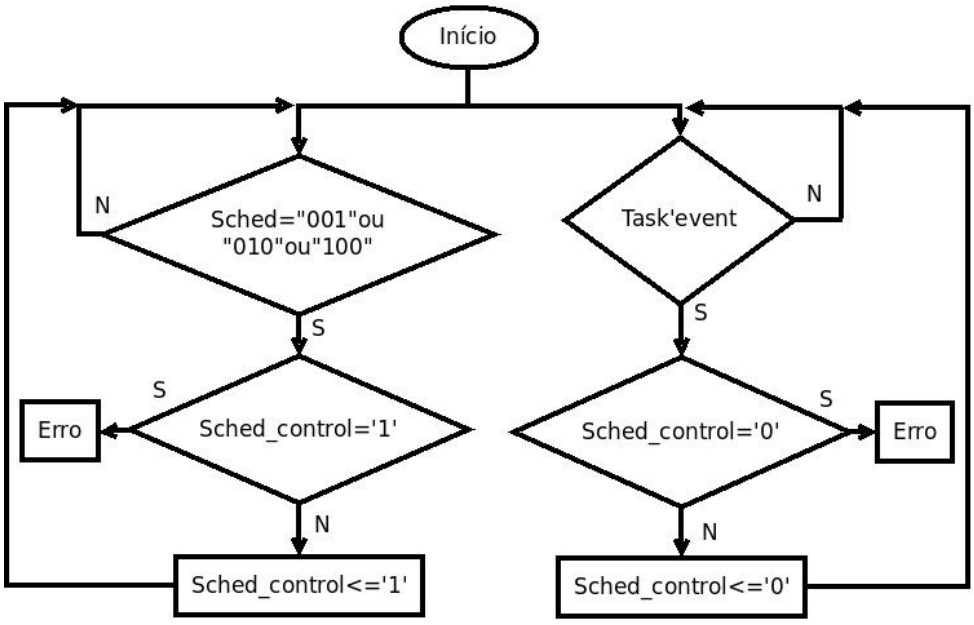


Figura 5.10: Sexto estágio módulo LCEG: Fluxograma de controle dos eventos de reescalonamento.

5.3 Conclusão do Capítulo

Neste capítulo foi apresentada a proposta deste trabalho, o RTOS-G para detecção de falhas em sistemas embarcados baseados em sistemas operacionais de tempo real. Foram expostos diagramas dos módulos internos do RTOS-G, onde foi exposto como são feitas: a identificação da tarefa em execução; a identificação dos eventos do RTOS que levam ao reescalonamento; a forma de controle das listas de

tarefas (*ready e blocked*); e os erros que são detectados pelo RTOS-G. O erros detectados pelo RTOS-G estão resumidos abaixo:

1. Tarefa em estado bloqueada sendo executada;
2. Tarefa em execução não consta nas listas do RTOS-G (erro do monitor);
3. Tarefa em execução não é a de maior prioridade da lista de tarefas prontas;
4. Não ocorreu o evento de *tick* (indicado pelo módulo FC) após o sinal de *tick* (processador Plasma);
5. Ocorreu evento de *tick* (indicado pelo módulo FC) sem ter ocorrido o sinal de *tick* (processador Plasma);
6. A interrupção não foi executada após ter ocorrido o sinal de interrupção;
7. A interrupção foi executada sem ocorrer o sinal de interrupção;
8. Ocorreu um evento de reescalonamento mas as tarefas não foram reescalonadas;
9. Tarefas forma reescalonadas sem ter ocorrido um evento de reescalonamento.

Concluindo, pode ser visto que o RTOS-G proposto é genérico o bastante para ser aplicado a qualquer sistema embarcado, desde que se tenha informação suficiente sobre a arquitetura do processador (sinais internos como o *tick*), do sistema operacional (funções de reescalonamento) e que se baseie em endereços fixos de memória para as tarefas e funções.

6 *Validação e Avaliação da Proposta*

6.1 Introdução

Neste capítulo são apresentados os experimentos utilizados para analisar e validar o comportamento da técnica proposta. Também serão expostas a placa em que foram realizados os experimentos, a plataforma para injeção de ruído eletromagnético e a norma utilizada na realização dos experimentos com ruído conduzido (SISC [4]).

6.2 Procedimento de Injeção de Falhas

A capacidade de detecção do RTOS-G em comparação com o Plasma RTOS, foi avaliada através de experimentos com interferência eletromagnética conduzida, de acordo com a norma IEC 61000-4-29. A figura 6.1 apresenta o diagrama de blocos do sistema utilizado para realização dos experimentos. Foram utilizados os três FPGAs da plataforma de testes. No FPGA0 foram implementados o processador Plasma e o RTOS-G. No FPGA1 foi implementado o Chipscope [3]. Os sinais de erro do RTOS-G e a tarefa em execução do RTOS são enviados do FPGA 0 para o FPGA1, onde estes sinais são monitorados pelo Chipscope. No FPGA_CLK foi implementado um gerador de *clock* para sincronizar as FPGAs. Os sinais de erro do Plasma RTOS são enviados através da UART para o PC (*Personal Computer*) onde é feito o controle do experimento.

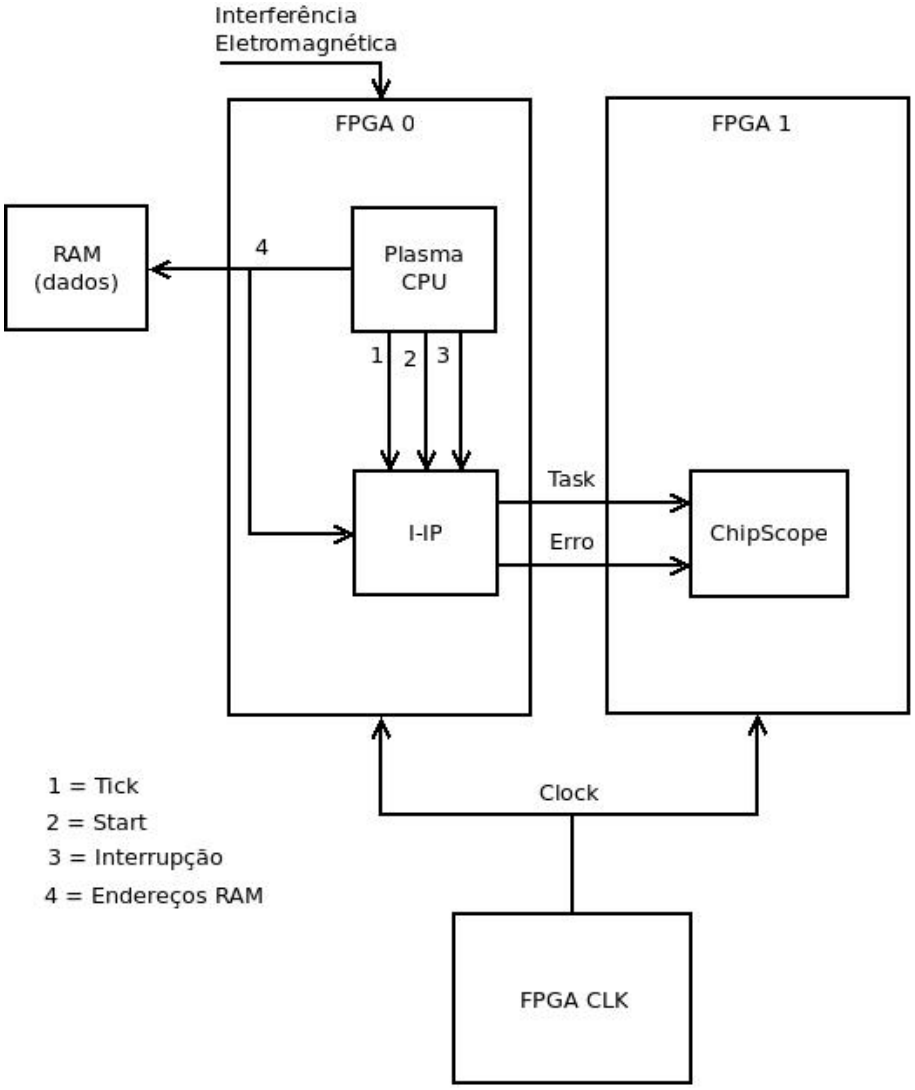


Figura 6.1: Diagrama da Plataforma de Testes.

A injeção de falhas é realizada mediante a geração de quedas de tensão na linha de alimentação do core do FPGA, conforme a norma IEC 61.000-4-29 [1], utilizando a plataforma para injeção de ruído apresentado em [34]. A interferência utilizada foi a do tipo queda conforme figura 6.2.

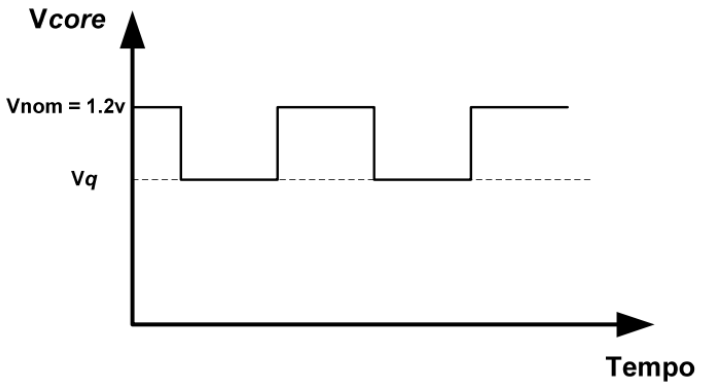


Figura 6.2: Variação da tensão.

Após ser implementado todo o sistema apresentado na figura 6.1, o Chipscope é configurado para capturar os sinais *erro* e *task* enviados pelo RTOS-G, assim que for indicado um erro. O RTOS indica erros através de interrupção da UART, é enviada uma mensagem indicando a linha da *assert* que foi detectado o erro. Através do PC é monitora ambos os eventos, indicação de erros do RTOS-G e do Plasma RTOS, assim que é indicado um erro a interferência é removida, os dados são salvos e a FPGA0 é novamente implementada com o Plasma e RTOS-G para realização do próximo experimento.

6.3 Placa para Implementação da Injeção de Falhas

A placa para realização dos experimentos foi desenvolvida pelo laboratório SISC [4] e apresentada em [30]. Ela foi baseada nas normas de teste de susceptibilidade de circuitos integrados a interferências eletromagnéticas (EMI - *Electro-Magnetic Interference*) irradiadas e conduzidas. O diagrama esquemático genérico é apresentado na figura 6.3 e dispõem dos seguintes componentes [30]:

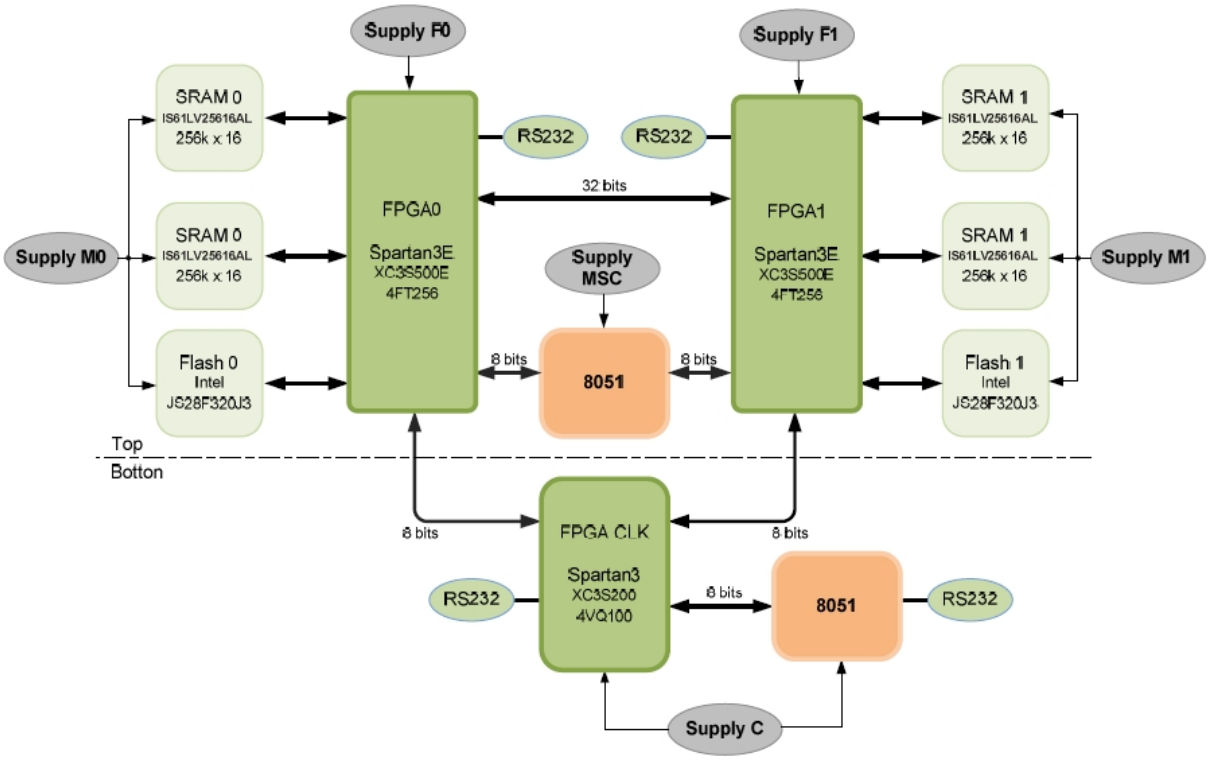


Figura 6.3: Diagrama genérico placa SISC [30].

- 2 FPGA's Xilinx XC3S500E (500k portas, 256 pinos, 360 Kbits de RAM interna, 20 multiplicadores e 4 DCM's);

- 1 FPGA Xilinx XC3S200 (200k portas, 144 pinos, 216 Kbits de RAM interna, 12 multiplicadores e 2 DCM's);
- 4 memórias SRAM (do inglês, Static Random Access Memory) IS61LV25616AL-10T, produzidas pela ISSI, que formam dois bancos de memória de 1Mbyte com configuração de 256x16 para cada FPGA;
- 2 memórias Flash Intel JS28F320J3 32Mbits e tempo de acesso de 110 ns;
- 2 microcontroladores (core 8051) produzidos pela Texas Instruments;
- 3 osciladores de frequência igual a 49.152 MHz (para cada FPGA);
- 2 cristais de frequência igual a 11.0592 MHz (para cada microcontrolador);
- Comunicação serial padrão RS-232 (para cada FPGA e microcontrolador);
- 3 reguladores de tensão LM317 para o controle independente dos níveis de tensão de alimentação;
- 1 sensor de temperatura serial 12 bits LM74, produzido pela National Semiconductor;
- 4 botões e 4 LED's;
- 2 conectores JTAG independentes para programação e debug dos FPGA's;
- Jumper's para seleção e controle independente dos níveis de tensão alimentação.

A figura 6.4 e figura 6.5 apresentam respectivamente a vista superior e inferior da placa de testes com os principais componentes destacados [30]. A placa foi desenvolvida para aceitar alimentação externa, desta maneira é possível realizar testes de injeção de ruído nas linhas de alimentação dos FPGA sob teste.

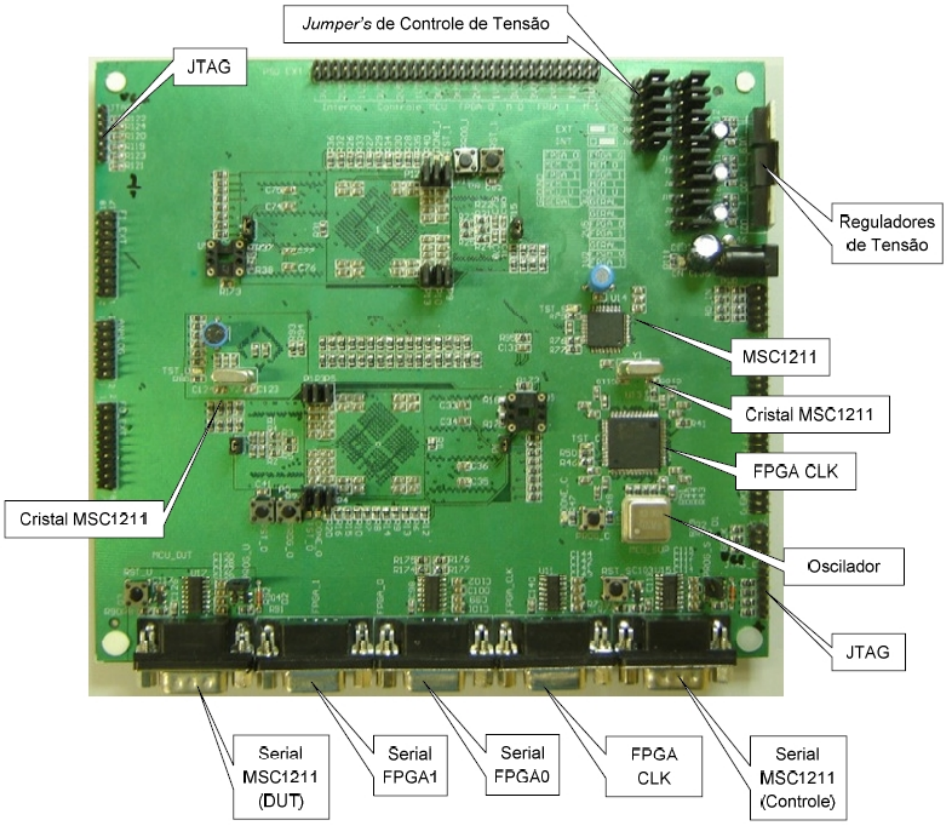


Figura 6.4: Vista superior placa SISC [30].

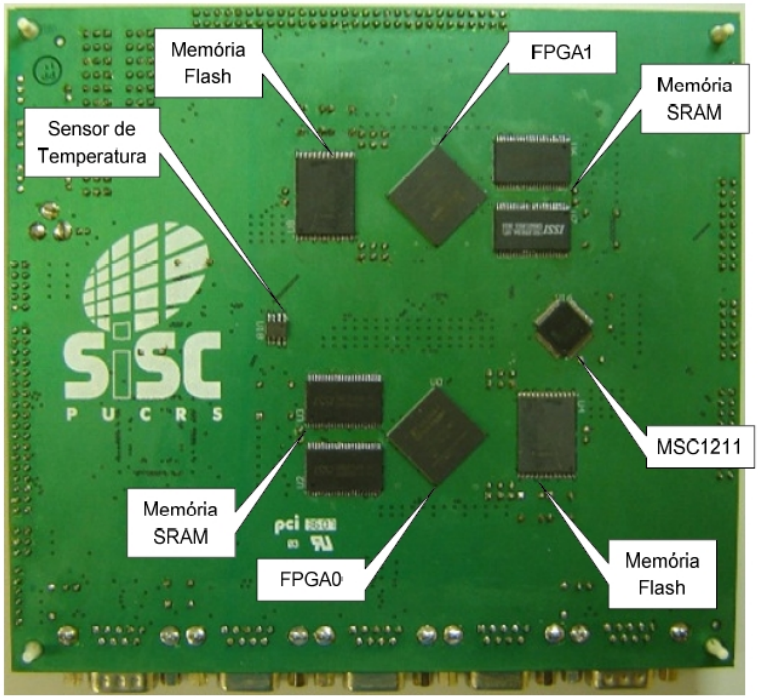


Figura 6.5: Vista infeior placa SISC [30].

6.4 Plataforma para Injeção de Ruídos

A plataforma para injeção de ruídos foi desenvolvida por [34] , figura 6.6. A plataforma foi concebida a partir da necessidade de avaliar e validar trabalhos desenvolvidos no grupo de pesquisas SISC, da PUCRS. Mais especificamente, era necessário o desenvolvimento de um ambiente flexível e de fácil utilização, que permitisse a injeção de ruído eletromagnético conduzido para a realização de experimentos envolvendo a validação e a avaliação de técnicas de tolerância a falhas bem como da robustez de circuitos integrados e sistemas [34].

Figura 6.6: Plataforma para Injeção de Ruídos [34].

As características técnicas principais da implementação do gerador são [34]:

- Tensão máxima de saída 9,99 V;
- Corrente máxima de saída 2,00 A;
- Temporização mínima 1 ms e máxima 2000 ms;
- Tempo de resposta de habilitação 20 ms;
- Tempo máximo de subida do sinal 185 ns/V;
- Regulagem de carga <5,00 %.

6.5 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 testes de imunidade de equipamentos e/ou dispositivos alimentados por fontes de corrente contínua externas de baixa tensão [32].

Tendo em vista a existência de diversos tipos de distúrbios relacionados às linhas de alimentação de equipamentos e dispositivos elétricos e eletrônicos, faz-se necessário, para o bom entendimento desta dissertação, a definição de: queda de tensão, pequena interrupção e variação de tensão [32].

- Queda de tensão: é caracterizada como uma súbita redução na tensão de alimentação, do equipamento e/ou dispositivo, seguida da sua recuperação em um curto período de tempo [1] [32].
- Pequena Interrupção: é caracterizada como o desaparecimento momentâneo da tensão por um período de tempo não maior que um minuto. Na prática, quedas de tensão superiores a 80 % de seu valor são consideradas interrupções [1] [32].
- Variação de tensão: é caracterizada como uma mudança gradual da tensão de alimentação para um valor maior ou menor do que a tensão nominal , podendo ser a sua duração curta ou longa [1] [32].

As tabelas 6.1, 6.2 e 6.3 a seguir apresentam os níveis de tensão percentuais relativos à tensão nominal e os tempos de duração sugeridos pela norma para testes em equipamentos e dispositivos elétricos e eletrônicos [32].

Tabela 6.1: Níveis de tensão e duração recomendados para quedas de tensão [1] [32].

Teste	Nível de Tensão (%)	Duração(s)
Queda de Tensão	40 a 70 ou x	0,01 0,03 0,1 0,3 1 x

X: valor definido de acordo com a especificação do produto

Tabela 6.2: Níveis de tensão e duração recomendadas para interrupções [1] [32].

Teste	Nível de Tensão (%)	Duração(s)
Pequena Interrupção	0	0,001 0,003 0,01 0,03 0,1 0,3 1 x

X: valor definido de acordo com a especificação do produto

Tabela 6.3: Níveis de tensão e duração recomendados para variação de tensão [1] [32].

Teste	Nível de Tensão (%)	Duração(s)
Variação de Tensão	85 a 120 ou 80 a 120 ou x	0,01 0,03 0,1 0,3 1 x

X: valor definido de acordo com a especificação do produto

6.6 Programas de Teste Desenvolvidos

Para avaliar a cobertura de falhas do RTOS-G foram desenvolvidos com base em outros trabalhos três programas de teste. Estes programas foram desenvolvidos com diferentes complexidades afim de analisar e validar o comportamento do RTOS-G. A figura 6.7 apresenta o primeiro programa de testes, neste foi implementado oito tarefas iguais e uma tarefa que é chamada por interrupção, todas as tarefas atualizam o valor de uma variável que é protegida por um semáforo e a interrupção atualiza o valor de uma segunda variável, esta não está protegida por semáforo.

Prioridades das tarefas do programa de testes 1 (1 é a menor prioridade e 4 a maior):

- Task1 = 1;
- Task2 = 2;
- Task3 = 3;
- Task4 = 4;
- Task5 = 1;
- Task6 = 2;
- Task7 = 3;
- Task8 = 4.

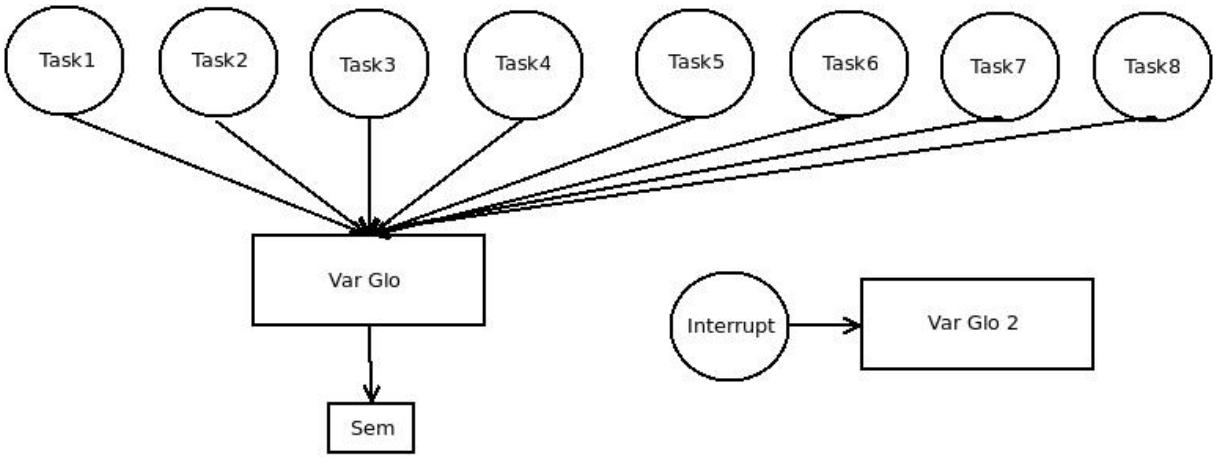


Figura 6.7: Programa de teste 1.

O programa de testes 2, figura 6.8, possui seis tarefas e uma interrupção. As tarefas de número 1 a 4 atualizam o valor de uma variável global protegida por um semáforo, a tarefa 5 envia uma mensagem a tarefa 6 e a interrupção atualiza o valor de uma segunda variável global.

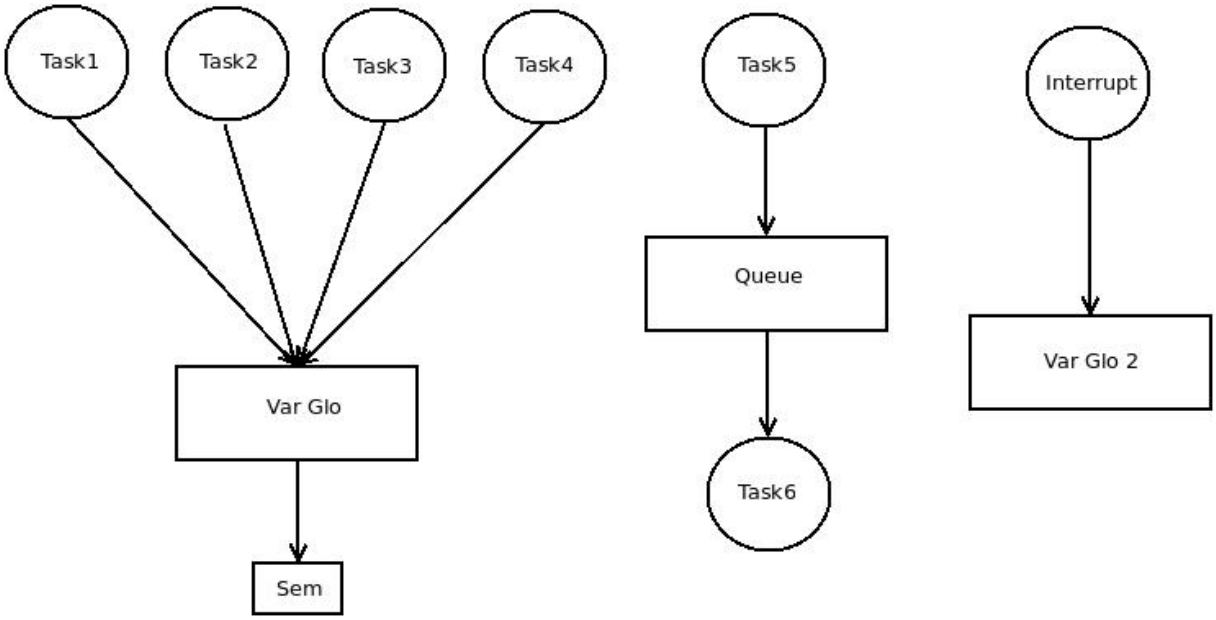


Figura 6.8: Programa de teste 2.

Prioridades das tarefas do programa de testes 2 (1 é a menor prioridade e 6 a maior):

- Task1 = 1;
- Task2 = 2;
- Task3 = 3;
- Task4 = 4;
- Task5 = 5;
- Task6 = 6.

O programa de testes 3, figura 6.9, possui sete tarefas e uma interrupção. A tarefa 1 e 2 atualizam o valor de uma variável global protegida por um semáforo, a tarefa 5 e 6 também atualiza o valor de uma segunda variável global, mas está protegida por um *mutex*. A tarefa 3 envia uma mensagem para a tarefa 4 e a interrupção envia uma mensagem para a tarefa 7.

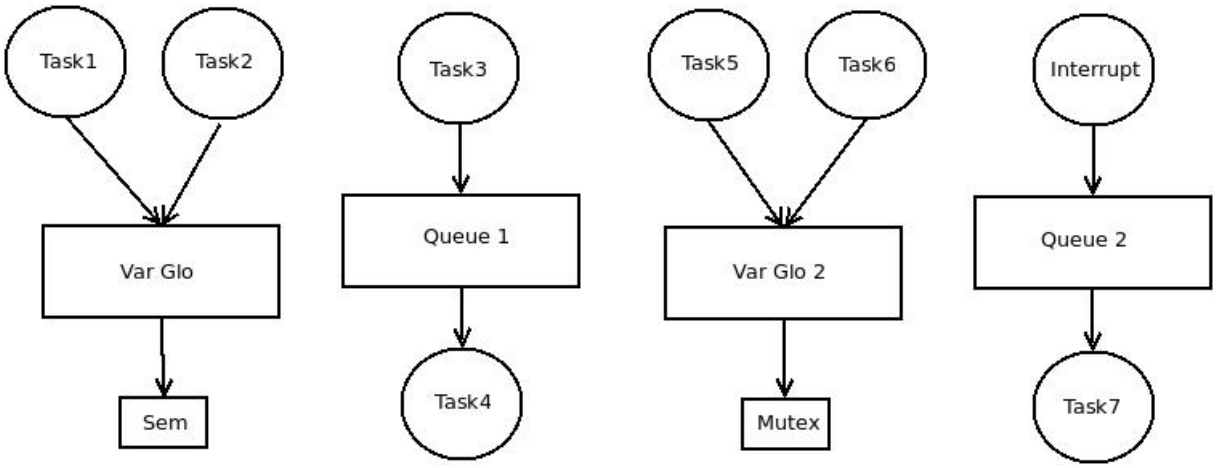


Figura 6.9: Programa de teste 3.

Prioridades das tarefas do programa de testes 3 (1 é a menor prioridade e 7 a maior):

- Task1 = 1;
- Task2 = 2;
- Task3 = 3;
- Task4 = 4;
- Task5 = 5;
- Task6 = 6;
- Task7 = 7.

6.7 Conclusão do Capítulo

Neste capítulo foi apresentada a placa utilizada na implementação do RTOS-G, a plataforma para injeção de ruído, a norma utilizada durante os experimentos e os programas de teste desenvolvidos para validar o comportamento do RTOS-G.

PARTE III - RESULTADOS, CONCLUSÕES E TRABALHOS FUTUROS

7 *Resultados*

No presente capítulo são apresentados os resultados obtidos a partir dos experimentos descritos no capítulo 6.2, utilizando os programas de teste definidos na seção 6.6. Os resultados estão divididos em: *Overhead* de espaço e latência, e resultados da injeção de falhas.

7.1 Overhead de espaço e latência

A tabela 7.1 apresenta o *overhead* relacionado ao espaço utilizado na FPGA quando adicionado o RTOS-G ao sistema. A utilização do RTOS-G com as memórias, gera um *overhead* de 23,39%, considerando o monitoramento de 14 tarefas. Deste *overhead* de 781 LUTs (*Look Up Table*) de 4 entradas, 316 LUTs são referentes as memórias. Ao considerar o RTOS-G sem as memórias, o *overhead* é de 13,93%.

Tabela 7.1: *Overhead* de espaço.

	Plasma CPU (#)	Plasma CPU + RTOS-G + Memórias (#)	*Overhead*
Número total de LUTs de 4 entradas	3338	4119	23,39%
Número total de Block RAMs	4	4	0%

A tabela 7.2 apresenta a latência de detecção de falhas do RTOS-G. Os erros são classificados abaixo:

Erro1: Tarefa em estado bloqueada sendo executada.

Erro2: Tarefa em execução não consta nas listas do RTOS-G (erro do monitor).

Erro3: Tarefa em execução não é a de maior prioridade da lista de tarefas prontas.

Erro4: Não ocorreu o reescalonamento por *tick* após o sinal de *tick*.

Tabela 7.2: Latência de detecção de erros.

Tipo de Erro	Latência (ciclos de *clocks*)
Erro1	12
Erro2	11
Erro3	6*
Erro4	x
Erro5	6
Erro6	x
Erro7	6
Erro8	x
Erro9	6

* Este valor depende do número de tarefas a serem verificadas.

Erro5: Ocorreu reescalonamento por *tick* sem ter ocorrido o sinal de *tick*.

Erro6: A interrupção não foi executada após ter ocorrido o sinal de interrupção.

Erro7: A interrupção foi executada sem ocorrer o sinal de interrupção.

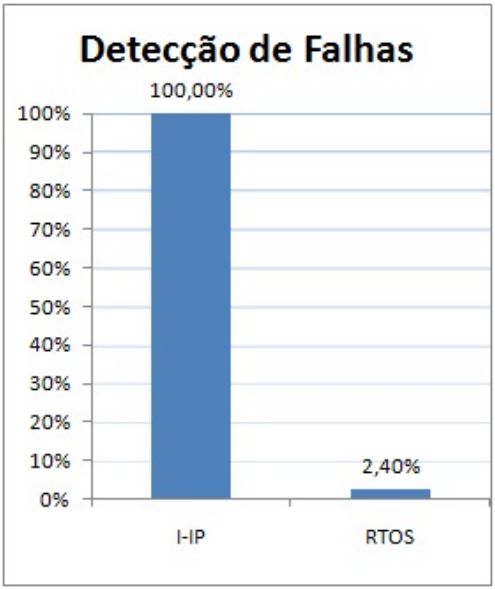
Erro8: Ocorreu um evento de reescalonamento mas as tarefas não foram reescalonadas.

Erro9: Tarefas foram reescalonadas sem ter ocorrido um evento de reescalonamento.

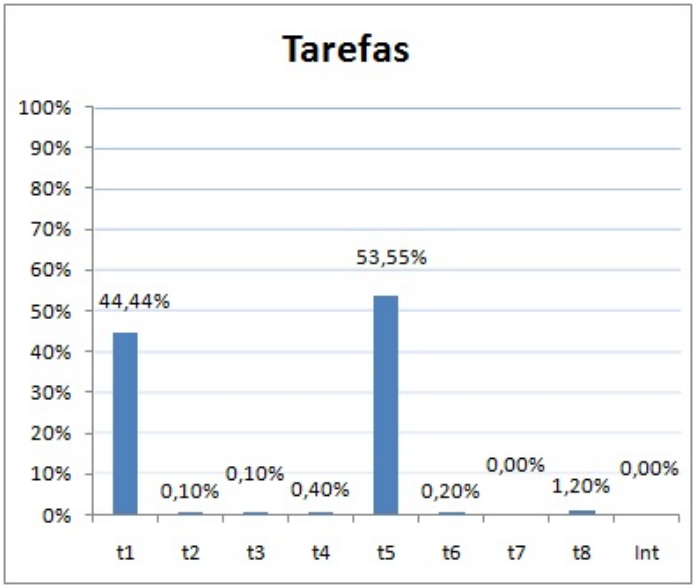
A latência de detecção de falhas do *Erro3* depende do número de tarefas que terão que ser comparadas. O processo de verificação de dados na memória consome 3 ciclos de relógio para cada verificação. Por exemplo se o RTOS-G tiver que verificar o estado de 4 tarefas, o processo de verificação nas memórias levará 12 ciclos de memória.

7.2 Resultado Obtidos

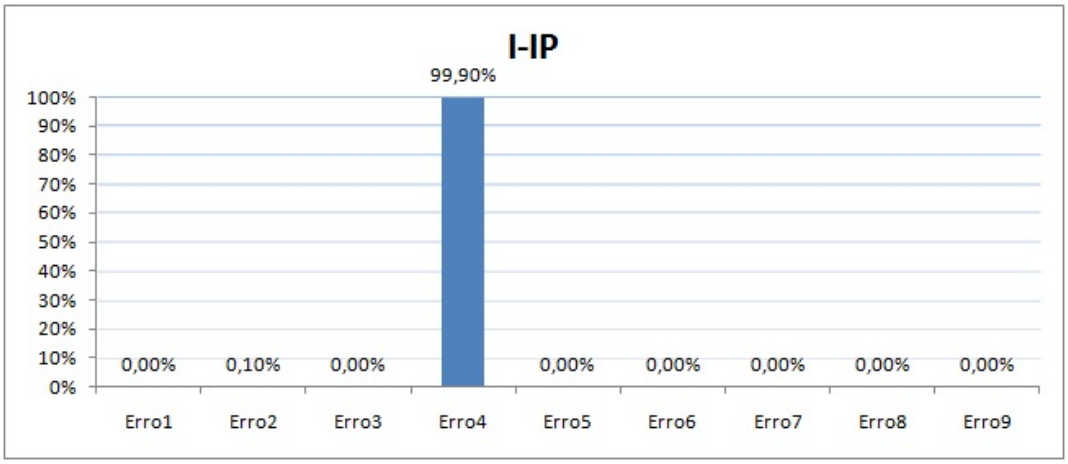
Foram realizados 1000 experimentos com cada programa de teste, totalizando 3000 experimentos. Note que neste método de injeção de falhas não é possível controlar o número de falhas injetadas. A capacidade de detecção encontrada se baseia nas falhas que foram detectadas pelos RTOS-G e pelo Plasma-RTOS. Houve experimentos em que não foram detectadas falhas, estes experimentos foram descartados, mas isto não significa que não ocorram falhas. Os resultados dos experimentos são apresentados nas figuras abaixo. A figura 7.1 apresenta o resultado do primeiro programa de teste.



(a) Porcentagem de falhas detectadas pelo Plasma RTOS e o RTOS-G.



(b) Tarefa em que a falha foi detectada pelo RTOS-G.

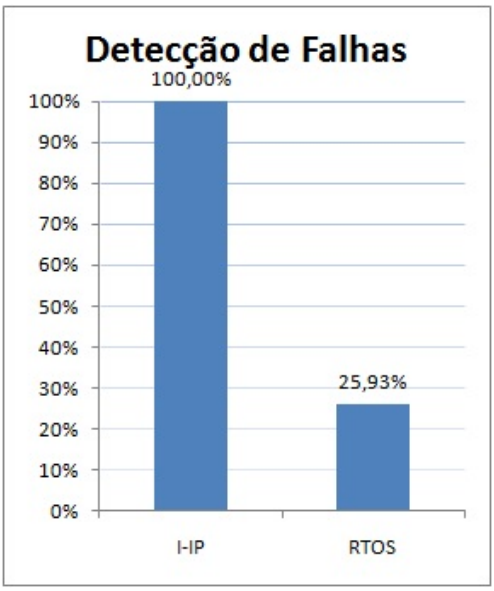


(c) Falhas indicadas pelo RTOS-G.

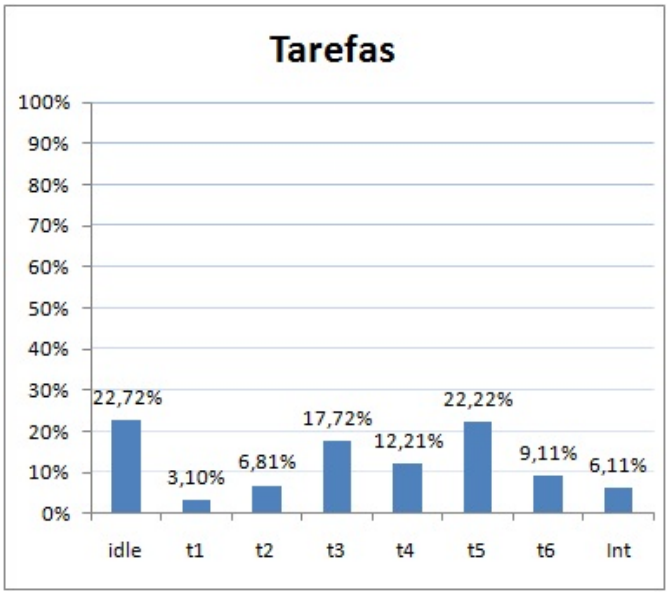
Figura 7.1: Resultado do programa de teste 1.

Resultado do programa de teste 1.

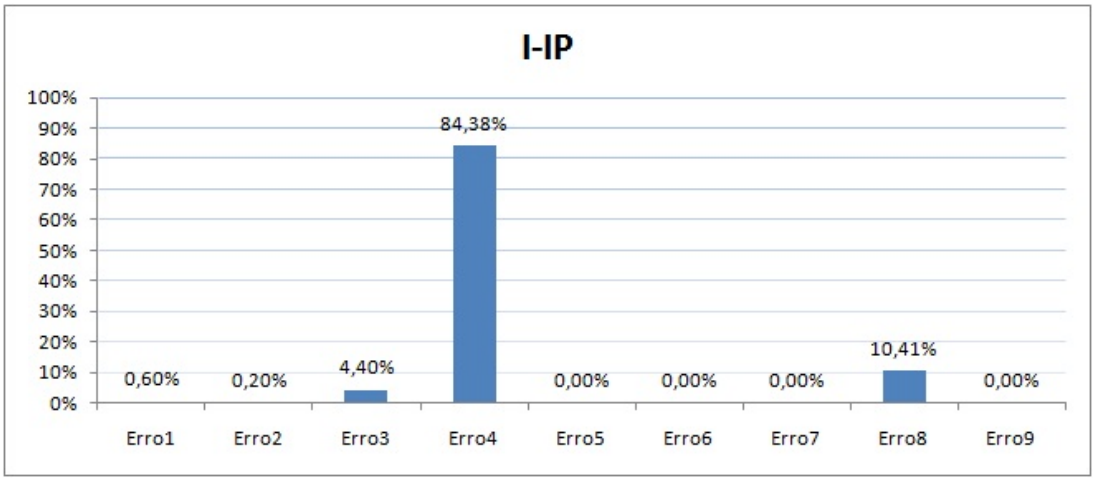
A figura 7.1a apresenta a porcentagem de falhas detectadas pelo Plasma RTOS e o RTOS-G. A figura 7.1b apresenta a tarefa que estava sendo executada quando o erro foi detectado pelo RTOS-G. Por exemplo em 44,44% dos experimentos, a tarefa 1 estava sendo executada durante a detecção de erro pelo RTOS-G. A figura 7.1c apresenta os erros detectados pelo RTOS-G para o programa de teste 1.



(a) Porcentagem de falhas detectadas pelo Plasma RTOS e o RTOS-G.



(b) Tarefa em que a falha foi detectada pelo RTOS-G.

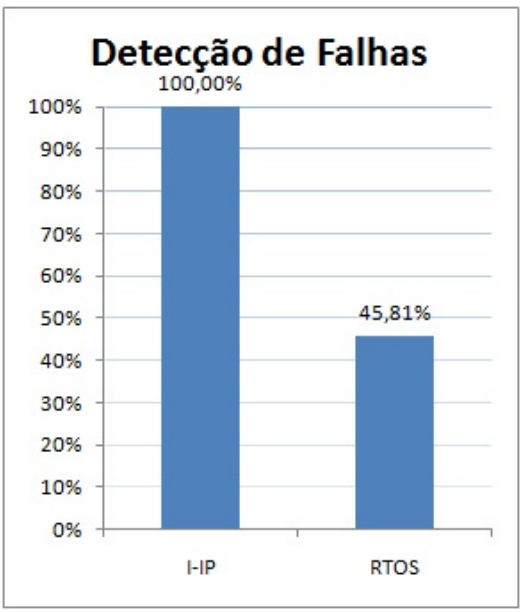


(c) Falhas indicadas pelo RTOS-G.

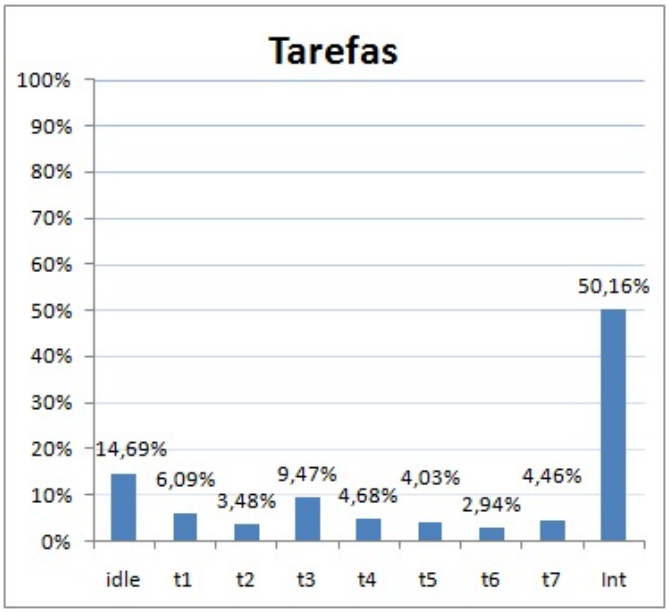
Figura 7.2: Resultado do programa de teste 2.

Resultado do programa de teste 2.

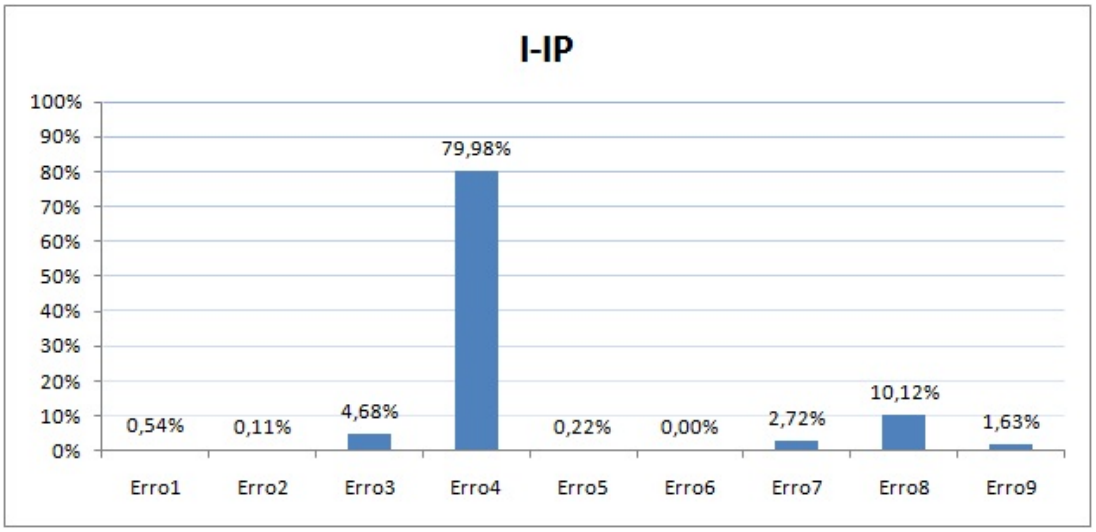
A figura 7.2a apresenta a porcentagem de falhas detectadas pelo Plasma RTOS e o RTOS-G. A figura 7.2b apresenta a tarefa que estava sendo executada quando o erro foi detectado pelo RTOS-G. A figura 7.2c apresenta os erros detectados pelo RTOS-G para o programa de teste 1.



(a) Porcentagem de falhas detectadas pelo Plasma RTOS e o RTOS-G.



(b) Tarefa em que a falha foi detectada pelo RTOS-G.



(c) Falhas indicadas pelo RTOS-G.

Figura 7.3: Resultado do programa de teste 3.

Resultado do programa de teste 3.

A figura 7.3a apresenta a porcentagem de falhas detectadas pelo Plasma RTOS e o RTOS-G. A figura 7.3b apresenta a tarefa que estava sendo executada quando o erro foi detectado pelo RTOS-G. A figura 7.3c apresenta os erros detectados pelo RTOS-G para o programa de teste 1.

7.3 Conclusão do Capítulo

Analisando os resultados dos experimentos, nota-se que o uso de recursos mais complexos causam maior ocorrência de erros. Em outras palavras, o sistema tem maior probabilidade de sofrer um erro, quando a tarefa em execução utiliza recursos mais complexos. Também podemos notar que quanto mais complexos são os programas de teste, maior a detecção de falhas do Plasma RTOS. Isto se deve ao fato de que quanto mais recursos são utilizados, mais *assert* o sistema testa. Como no programa de teste 1, o RTOS apenas atualiza uma variável global, a indicação de erros do RTOS é menor que no programa de teste 3, em que o sistema utiliza *mutex* e fila de mensagens. Durante a execução do programa de teste 3, mais *assert* são verificadas pelo RTOS, levando a uma maior taxa de detecção de erros.

8 Conclusão

Tendo em vista o aumento de sistemas embarcados nas mais diversas aplicações e, por outro lado, a capacidade de detecção de falhas das técnicas atuais, que não consideram falhas que afetam o escalonamento das tarefas do RTOS, este trabalho propõe uma técnica baseada em hardware para detecção de falhas em sistemas embarcados RTOS que afetam o fluxo de execução das tarefas. O módulo RTOS-G proposto foi desenvolvido e analisado através de experimentos de injeção de falhas. A maior contribuição deste trabalho consiste no aumento da robustez do sistema embarcado, visto que o RTOS-G monitora erros que não são monitorados pelas funções nativas do RTOS, através da análise do fluxo das tarefas.

Foi apresentado um estudo de caso para o microprocessador Plasma e seu sistema operacional Plasma RTOS. Conforme os resultados obtidos nos experimentos, o RTOS-G proposto apresentou menor latência e maior capacidade de detecção de falhas contra o RTOS. Nos experimentos, o RTOS-G detectou falhas em 100% dos experimentos contra 45,81% de detecção do RTOS no melhor caso. Concluindo, estamos convencidos que a proposta apresentada representa uma solução interessante para sistemas embarcados de tempo real baseados em RTOS.

9 Trabalhos Futuros

Como sugestão para trabalhos futuros, podemos citar a principal limitação do módulo RTOS-G, que é a impossibilidade de identificar os recursos que estão sendo liberados ou bloqueados. A identificação de tais recursos do RTOS irão aumentar a capacidade de detecção do RTOS-G, visto que será possível identificar qual recurso cada tarefa está esperando. Então, quando um recurso for liberado, poderá ser identificado qual tarefa será desbloqueada.

Também podemos sugerir experimentos mais complexos e com mais controle na injeção de falhas. Visto que não há controle das falhas gerados no sistema, não podemos garantir que realmente ocorreu uma quando o RTOS ou RTOS-G indicam a presença da mesma. Outra sugestão de trabalhos futuros é a implementação do módulo RTOS-G em diferentes sistemas operacionais e a realização de respectivos experimentos de injeção de falhas para verificar a confiabilidade do módulo nestes novos contextos.

Referências Bibliográficas

[1] *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 (61.000-4-29). Geneva, Switzerland : s.n., 2000, IEC - International Electrotechnical Commission, p. 37. Norma Técnica.*

[2] http://public.itrs.net.

[3] http://www.xilinx.com/tools/cspro.htm.

[4] Sisc – signals & systems for computing group. http://www.ee.pucrs.br/ sisc/.

[5] T. Anderson and P.A. Lee. *Fault Tolerance - Principles and Practice*. Englewood Cliffs :Prentice-Hall, 1981.

[6] J. Arlat, Y. Crouzet, JU. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computer*, 52(9), September 2003.

[7] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE TRANSACTIONS ON DEPENDABLE AND SECURE COMPUTING*, 1(1), 2004.

[8] E. A. Vargas F. and Gough M. P. Bezerra. Improving reconfigurable systems reliability by combining periodical test and redundancy techniques. *Journal of Electronic Testing: Theory and Applications - JETTA*, 17:163–174, May 2001.

[9] Letícia Maria Veiras Bolzani. Explorando uma solução híbrida: Hardware + software. Master's thesis, Faculdade de Engenharia, Programa de Pós-Graduação em Engenharia Elétrica., Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS, Dissertação de Mestrado, 2005.

[10] J A Clark and D K Pradhan. Fault injection a method for validating computer-system. *IEEE Computer*, 28:47–56, 1995.

[11] N. R. e McCluskey E. J. Saxena. Control-flow checking using watchdog assists and extended-precision. *IEEE Trans. on Computers*, 39:554–559, April 1990.

[12] J. B. and Shen J. P. Eifert. Processor monitoring using asynchronous signatured instruction streams. *Proceedings of the FTCS'25*, III:106–111, 1996.

[13] Benso et al A. A watchdog processor to detect data and control flow errors. *9th IEEE On-Line Testing Sym*, page 144, 2003.

[14] Z. et al Alkhalifa. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Trans. on Parallel and Distributed Systems*, 10:627–641, June 1999.

[15] P. et al. Bernardi. A new hybrid fault detection technique for systems-on-a-chip. *IEEE Transactions on Computers*, 55:185–198, February 2006.

[16] L. et al. Bolzani. Hybrid soft error detection by means of infrastructure ip cores. *10th IEEE International On-Line Testing Symposium (IOLTS'04)*, 2004.

[17] O. et al Goloubeva. Soft-error detection using control-flow assertions. *18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2003.

[18] G. A. et al. Kanawati. Evaluation of integrated system-level checks for on-line error detection. *IEEE Int. Computer Performance and Dependability Symposium*, pages 292–301, 1996.

[19] G. et al Miremadi. Two software techniques for on-line error detection. *22nd Int. Symp. on Fault-Tolerant Computing*, pages 328–335, 1992.

[20] B. et al. Nicolescu. Sensitivity of real-time operating systems to transient faults: A case study for microc kernel. *Radiation and Its Effects on Components and Systems. RADECS*, 2005.

[21] N. Ignat, B. Nicolescu, Y. Savari, and G. Nicolescu. Soft-error classification and impact analysis on real-time operating systems. *IEEE Design, Automation and Test in Europe*, 2006.

[22] N. Ignat, B. Nicolescu, Y. Savari, and G. Nicolescu. Soft-error classification and impact analysis on real-time operating systems. *IEEE Design, Automation and Test in Europe*, 2006.

[23] R. K. Iyer and Z. Kalbarczyk. *Hardware and Software Error Detection*. http://www.crhc.uiuc.edu/ kalbar/MotorolaCourse/HW&SW_ErrorDetection.pdf, 2002.

[24] Pradhan D. K. Fault-tolerant computer system design. *Prentice-Hall*, 1995.

[25] J. J. Labrosse. *MicroC/OS-II The Real-Time Kernel*. CMP Books, 2002.

[26] J. C. Laprie. Dependable computing and fault-tolerance: Concepts and terminology. *New York : IEEE Proceedings*, 1985.

[27] Qing Li. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003.

[28] H. and Silva J. G. Madeira. On-line signature learning and checking: Experimental evaluation. *IEEE CompEuro 91: Advanced Computer Technology, Reliable Systems and Applications*, pages 642–646, May 1991.

[29] G. Miremadi and J. Torin. Evaluating processor-behavior and three error-detection mechanisms using physical fault-injection. *IEEE Transactions on Reliability*, 44(3), September 1995.

[30] M. Moraes. Validação de uma técnica para o aumento da robustez de soc's a flutuações de tensão no barramento de alimentação. Master's thesis, Programa de Pós-Graduação em Engenharia Elétrica, PPGEE, PUCRS, 2008.

[31] N. Shirvani P. P. and McCluskey E. J. Oh. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51:111–122, March 2002.

[32] Jimmy Fernando Tarrillo Olano. Escalonador em hardware para detecção de falhas em sistemas embarcados de tempo real. Master's thesis, Pontifícia Universidade Católica do Rio Grande do Sul PUCRS, 2009.

[33] K. Pagiamtzis and A. Sheikholeslami. Content-addressable memory (cam) circuits and architectures: A tutorial and survey. *IEEE Journal of Solid-State Circuits*, 41(3), March 2006.

[34] Darcio Pinto Prestes. Plataforma para injeção de ruído eletromagnético conduzido em circuitos integrados. Master's thesis, Programa de Pós-Graduação em Engenharia Elétrica, PPGEE, PUCRS, 2010.

[35] Steve Rhoads. Plasma most mips i (tm) opcodes. http://opencores.org/project,plasma.

[36] Andrew S. Tanenbaum. *Sistemas Operacionais Modernos*. Pearson Education, 2003.

[37] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating Systems Design and Implementation*. Pearson Education, 2007.

[38] J. Tarrilho, L. Bolzani, and F. Vargas. A hardware-scheduler for fault detection in rtos-based embedded systems. *12th Euromicro Conference on Digital System Design*, 2009.

[39] J. Tarrillo, L. Bolzani, F. Vargas, E. Gatti, F. Hernandez, and L. Fraigi. Fault-detection capability analysis of a hardware-scheduler ip-core in electromagnetic interference environment. *[1]IEEE 7th East-West Design & Test Symposium*, 2009.

[40] S Thiagrajan. Survey of fault-tolerant techniques in modern micro-processors. *homepages.cae.wisc.edu/ ece753/INFO.html*, July 2006.

[41] E. Touloupis, J. A. Flint, V. A. Chouliaras, and D. D. Ward. Study of the effects of seu induced faults on a pipeline protected microprocessor. *IEEE TC*, 2007.

[42] N. J. and Hwu W. W. Warter. A software based approach to achieving optimal performance for signature control flow checking. *Newcastle Upon Tyne, UK*, pages 442–449 FTCS'20, 1990.

[43] C Weaver and T Austin. A fault tolerant approach to microprocessor design. *Proc. Int. Conf. on Dependable Sys*, 2001.

[44] K. and Shen J. P. Wilken. Continuous signature monitoring: Low-cost concurrent detection of processor control erros. *IEEE Trans. of Computer-Aided Design*, 9:629–641, June 1990.

[45] Xilinx. Content-addressable memory v6.1. *http://www.xilinx.com/support/documentation/ipmeminterfacestorele* 2008.

[46] S. S. Chen F. C. and Yau K. H. Yau. An approach to real-time control flow checking. *2nd International Conference on Computer Software and Applications*, pages 163–168, 1978.

ANEXO 1 - Artigos publicados