UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS

SAMIR SILVA KHAOULE

Análise da performance de um Sistema Operacional de Tempo-Real utilizando a ferramenta de *benchmark Thread-Metric*

São Carlos
2011

SAMIR SILVA KHAOULE

Análise da performance de um Sistema Operacional de Tempo-Real utilizando a ferramenta de *benchmark Thread- Metric*

Trabalho de Conclusão de Curso apresentado
à Escola de Engenharia de São Carlos, da
Universidade de São Paulo

Curso de Engenharia Elétrica com ênfase em
Eletrônica

ORIENTADOR: Professor Doutor Evandro Luís Linhares Rodrigues

São Carlos
2011

Ficha catalográfica preparada pela Seção de Tratamento
da Informação do Serviço de Biblioteca – EESC/USP

Khaoule, Samir Silva.
K45a Análise da performance de um sistema operacional em tempo real utilizando a ferramenta de *benchmark Thread-Metric*es. / Samir Silva Khauole Raimondi ; orientador Evandro Luís Linhares Rodrigues –- São Carlos, 2011.

Monografia (Graduação em Engenharia Elétrica com ênfase em Eletrônica) -- Escola de Engenharia de São Carlos da Universidade de São Paulo, 2011.

1. Tempo real. 2. RTOS. 3. SOTR. 4. *Thread-Metric*. 5. *Free*RTOS. 6. BRTOS. I. Titulo.

FOLHA DE APROVAÇÃO

Nome: Samir Silva Khaoule

Título: "Análise da Performance de um Sistema Operacional de Tempo-Real Utilizando a Ferramenta de Benchmark Thread-Metric"

Trabalho de Conclusão de Curso defendido e aprovado em 25 / 11 / 2011,

com NOTA 8,0 (oito, e 80), **pela comissão julgadora:**

Prof. Assistente Carlos Goldenberg - EESC/USP

Profa. Assistente Luiza Maria Romeiro Coda - EESC/USP

Prof. Associado Homero Schiabel
Coordenador da CoC-Engenharia Elétrica
EESC/USP

Dedico esse trabalho e a graduação à minha família, que está presente em tudo que faço.

Agradecimentos

Agradeço às pessoas que passaram pela minha vida, a tudo que me trouxe até aqui. Aos amigos que reconheci na faculdade, ao Neves, Biondo, Nishizawa e Michel que foram parceiros em todos projetos que realizei, ao Zé, ao Willian, Paulo e Diogo que testaram minha paciência. Sou grato ao professor Evandro, ao professor Gesualdo e ao professor Gustavo pela orientação.

Um imenso carinho aos meus pais que depois de tantos esforços descansam mais aliviados com a formatura de seu filho caçula. Muito obrigado por me ensinarem que a todos os planos que fazemos devemos sinceridade e identificação.

“Vivendo, se aprende; mas o que se aprende, mais, é só a fazer outras maiores perguntas.” (João Guimarães Rosa)

Resumo

O propósito maior desse trabalho é entender os conceitos que envolvem um RTOS (Sistema Operacional de Tempo-Real) a partir da apresentação de um método para a avaliação do seu desempenho, a ferramenta de benchmark *Thread-Metric*. O trabalho testa tarefas básicas de um RTOS, tais como: troca de contexto cooperativo, troca de contexto preemptivo, processamento de interrupção sem preempção, processamento de interrupção com preempção, passagem de mensagem, processamento de semáforo e alocação e desalocação de memória. Os testes foram realizados em dois Sistemas Operacionais de Tempo-Real, o *FreeRTOS* e o BRTOS. Este trabalho oferece um direcionamento a quem busca conhecer mais acerca das características básicas de um RTOS.

Palavras-Chave: tempo-real, RTOS, SOTR, *Thread-Metric*, *FreeRTOS,* BRTOS.

Abstract

The main purpose of this work is to understand the concepts involving an RTOS (Real-Time Operating System) from the presentation of a method for evaluating its performance, the suite benchmark Thread-Metric. This work tests a set of commons services: Cooperative Context Switching, Preemptive Context Switching, Interrupt Processing without Preemption, Interrupt Processing with Preemption, Message Passing, Semaphore Processing and Memory Allocation/Deallocation. This tests had been done in two Real-Time Operating Systems, the FreeRTOS and BRTOS. This work provides a direction to those seeking to know more about the basics features of an RTOS.

Key Words: real-time, RTOS, Thread-Metric, FreeRTOS, BRTOS.

Lista de Figuras

Lista de Tabelas

Lista de Abreviaturas

RTOS - *Real Time Operation Systems*

SOTR - Sistema Operacional de Tempo-Real

SOPG – Sistema Operacional de Propósito Geral

SCB – *Semaphore Control Block*

ISR - *Interrupt Service Routine*

API - *Application Programming Interface*

Sumário

Capítulo 1 – Introdução

1.1 Apresentação

Na medida em que o uso de sistemas computacionais prolifera na sociedade atual, aplicações com requisitos de tempo real tornam-se cada vez mais comuns. Essas aplicações variam muito em relação à complexidade e às necessidades de garantia no atendimento de restrições temporais. Entre os sistemas mais simples, estão os controladores inteligentes embutidos em utilidades domésticas, tais como lavadoras de roupa e videocassetes. Na outra extremidade do espectro de complexidade estão os sistemas militares de defesa, os sistemas de controle de plantas industriais (químicas e nucleares) e o controle de tráfego aéreo e ferroviário. Algumas aplicações de tempo real apresentam restrições de tempo mais rigorosas do que outras e são conhecidos como *hard real-time systems*; entre esses, encontram-se os sistemas responsáveis pelo monitoramento de pacientes em hospitais, sistemas de supervisão e controle em plantas industriais e os sistemas embarcados em robôs e veículos (de automóveis até aviões e sondas espaciais). Entre aplicações que não apresentam restrições tão críticas (conhecidos com *soft real-time systems*), normalmente, são citados os vídeogames, as teleconferências através da Internet e as aplicações de multimídia em geral. Todas essas aplicações que apresentam a característica adicional de estarem sujeitas a restrições temporais, são agrupados no que é usualmente identificado como Sistemas de Tempo Real (FARINES, 2000).

Essa distinção entre *soft* e *hard real time system* é compreensível. Tendo-se como exemplo um sistema de radar aeroespacial que recebe informações de posicionamento das aeronaves para que possíveis colisões sejam detectadas e evitadas, se o sistema não tratar estas entradas dentro de suas restrições de tempo, poderá causar uma tragédia, o que caracteriza um sistema *hard real-time*. Tendo-se como exemplo a leitura de dados para a reprodução de um arquivo mp3, se o sistema não tratar estes dados dentro de suas restrições de tempo, o operador poderá ter a sensação de que o sistema “travou”, o que poderá causar até a desistência da utilização do produto. Mas nada que comprometa o

funcionamento de um serviço essencial, o que configura um sistema *soft real-time.* Na figura 1 abaixo podemos encontrar exemplos de *soft* e *hard real-time systems.*

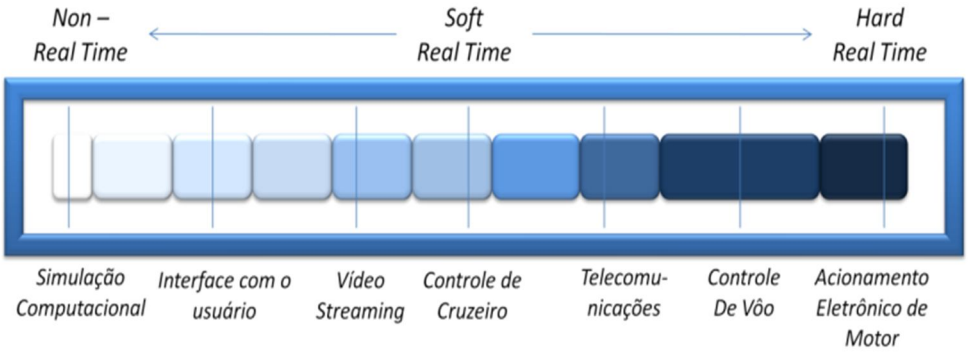

Figura 1- Faixa de exemplos de sistemas tempo-real. Fonte (BARR, 2007, apud GIRIO, 2010)

Um sistema operacional tempo-real (RTOS) é um programa que agenda a realização de tarefas de maneira temporal, gerenciando os recursos do sistema e fornecendo uma base sólida para o desenvolvimento do código de uma aplicação (LI, 2003).

A velocidade do processador é determinante na execução das instruções requeridas para desempenhar qualquer serviço de um RTOS, mas isso não pode ser considerado sinônimo de sucesso devido ao alto custo e alto consumo de potência. Um bom processador leva a uma boa performance, mas uma performance com um custo por vezes inaceitável na maioria dos sistemas embarcados. É provável que um processador de 2 GHz execute o programa em um tempo satisfatório, mas requererá um alto custo, um consumo de energia elevado, ou exigirá um espaço físico que tornará o sistema embarcado pouco portátil.

A pergunta que surge é a seguinte: até onde os designers do sistema podem reduzir a performance do processador sem comprometer os requisitos do sistema de tempo-real? Uma grande parte da resposta reside na quantidade de ciclos do processador que o RTOS requer. Assim, pode-se descobrir o quanto um RTOS influencia na performance do sistema.

Uma ferramenta de *benchmark* que testa um RTOS em suas diversas funções permitiria aos desenvolvedores fazerem uma avaliação a respeito do comportamento do Sistema Operacional de Tempo-Real.

A ferramenta de testes deve ser desvinculada da interferência do fornecedor do RTOS testado, pois uma *benchmark* produzida pelo fornecedor seria inadequada para fazer comparações, uma vez que cada fornecedor, provavelmente, se baseará em códigos diferentes, e o que estará sendo medido serão diferentes taxas de eventos, mesmo que contenham o mesmo principio (LAMIE, 2007, tradução nossa).

O trabalho tem o intento de mostrar que é possível utilizar as mesmas condições para avaliar diferentes RTOSes, com uma ferramenta de benchmark que não é vinculada a nenhum fornecedor, a *Thread-Metric,* e, assim, ter uma noção comparativa entre os Sistemas Operacionais de Tempo-Real testados.

1.2 Objetivos

Este trabalho tem o propósito de quantificar as funcionalidades de um RTOS de forma que possa ser reproduzida por quem precise dessa avaliação.

Através de testes serão comparados diferentes RTOSes cumprindo as mesmas funções em um mesmo processador.

Também serão comparados os desempenhos de um mesmo RTOS em diferentes microcontroladores.

1.3 Organização da Monografia

A monografia se estrutura de acordo com a segmentação em capítulos proposta a seguir:

Capítulo 1: Introdução - Faz um apresentação geral do tema da monografia de modo a fornecer uma ideia abrangente do trabalho.

Capítulo 2: Revisão de Literatura – Tem como objetivo apresentar o tema e conceitos que serão utilizados no decorrer do trabalho.

Capítulo 3: Materiais e Métodos – Este capítulo apresenta como o trabalho foi desenvolvido, cobrindo a metodologia e os materiais utilizados para que os objetivos fossem alcançados.

Capítulo 4: Resultados e Conclusões – Este capítulo apresenta as análises e os resultados obtidos com a implementação da metodologia. Aponta também possíveis melhorias a serem feitas futuramente e a conclusão do projeto.

Capítulo 2 – Revisão de Literatura

2.1 Sistemas Operacionais

Um Sistema Operacional (SO) realiza basicamente duas funções não relacionadas: estender a máquina e gerenciar recurso. Visto como máquina estendida, o SO auxilia o usuário ao fornecer uma linguagem mais simples do que a linguagem que encontraria se tivesse que entrar em contato diretamente com o hardware, apresentando uma interface mais conveniente, escondendo conceitos desagradáveis para um usuário típico. Da perspectiva de gerenciador de recursos, o SO serve para fornecer uma alocação ordenada e controlada de processadores, memórias e dispositivos de entrada e saída entre vários programas que competem por ele, evitando uma confusão entre a realização dos comandos dos diferentes programas (TANENBAUM, 2003).

Todo sistema operacional oferece meios para que um programa seja carregado na memória principal e executado. Talvez o serviço mais importante oferecido seja o que permite a utilização de arquivos e diretórios. O acesso aos periféricos também é feito através do sistema operacional. À medida que diversos usuários compartilham o computador, cabe também ao sistema operacional garantir que cada usuário possa trabalhar sem sofrer interferência danosa dos demais.

Os programas solicitam serviços ao sistema operacional através das chamadas de sistema. São similares às chamadas de sub-rotinas, só que, enquanto as chamadas de sub-rotinas são transferências para procedimentos normais do programa, as chamadas de sistema transferem a execução para o sistema operacional. O retorno da chamada de sistema, assim como o retorno de uma sub-rotina, faz com que a execução do programa seja retomada a partir da instrução que segue a chamada. A parte do sistema operacional responsável por implementar as chamadas de sistema é normalmente chamada de núcleo ou *kernel*. Os principais componentes do *kernel* de qualquer sistema operacional são a gerência de processador, a gerência de memória, o sistema de arquivos e a gerência de entrada e saída. (OLIVEIRA, 2001).

Dentre os diversos tipos de Sistemas Operacionais existentes, de multiprocessadores, de computadores pessoais, de servidores, ou embarcados, tem os que se caracterizam por terem o tempo como parâmetro fundamental, os Sistemas Operacionais de Tempo Real (RTOS).

2.1.1 Sistemas de Tempo Real

Um RTOS agenda tarefas de maneira temporal, pois o importante nos sistemas em tempo real não é a velocidade e sim a previsibilidade. Ramamritham e Stankovic (1996, apud GÍRIO, 2010) definem sistemas de tempo real como aqueles que não dependem apenas de um correto resultado lógico da computação, mas também do instante em que estes são produzidos. O aspecto temporal não é uma questão de desempenho e sim de funcionalidade do sistema.

Nos sistemas tempo real críticos (*hard real-time*) o não atendimento de um requisito temporal pode resultar em consequências catastróficas tanto no sentido econômico quanto em vidas humanas. Para sistemas deste tipo é necessária uma análise de escalonabilidade em tempo de projeto. Esta análise procura determinar se o sistema vai ou não atender os requisitos temporais mesmo em um cenário de pior caso, quando as demandas por recursos computacionais são maiores. Quando os requisitos temporais não são críticos (*soft real-time*) eles descrevem o comportamento desejado. O não atendimento de tais requisitos reduz a utilidade da aplicação, mas não resulta em consequências catastróficas.

Sistemas Operacionais de Propósito Geral (SOPG) encontram dificuldades em atender as demandas específicas das aplicações de tempo real. Fundamentalmente, SOPG são construídos com o objetivo de apresentar um bom comportamento médio, ao mesmo tempo em que distribuem os recursos do sistema de forma equitativa entre os processos e os usuários. Existe pouca preocupação com previsibilidade temporal. Mecanismos como caches de disco, memória virtual, fatias de tempo do processador etc, melhoram o desempenho médio do sistema, mas tornam mais difícil fazer afirmações sobre o comportamento de um processo em particular frente às restrições temporais. Aplicações com restrições de tempo real estão menos interessadas em uma

distribuição uniforme dos recursos e mais interessadas em atender requisitos tais como períodos de ativação e *deadlines*.(OLIVEIRA, 2001)

2.1.2 *Threads* e Processos

O conceito de processo (também conhecido como tarefa) é utilizado para diferenciar um programa de sua execução. Um processo é comumente tido como um programa em execução. Ou seja, enquanto um programa é uma sequência de instruções escritas que não altera seu estado e pode nem ao menos estar sendo utilizada naquele instante, o processo tem um viés mais dinâmico, ele altera o seu próprio estado à medida que executa um programa. O processo que faz as chamadas de sistema ao executar um programa.

Um processo pode assumir os estados descritos apresentados na figura 1.

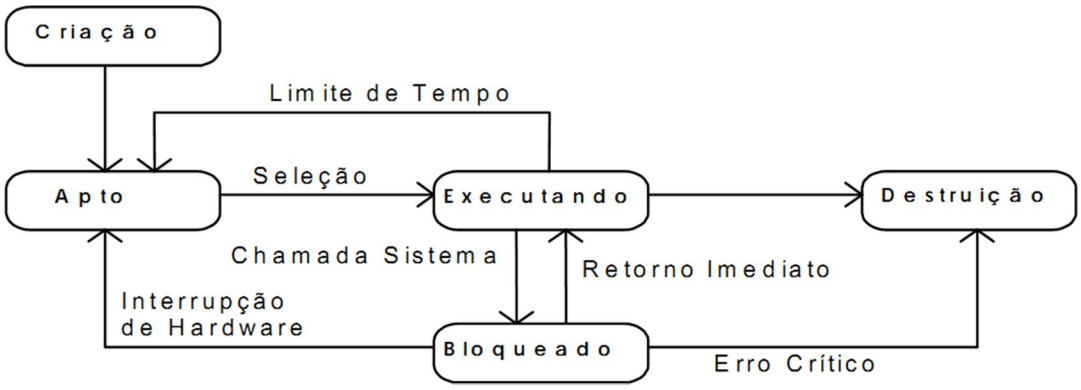

Figura 2- Estados de um processo. Fonte (Oliveira, 2001)

Um processo no estado executando pode fazer as chamadas de sistema, enquanto a chamada não é atendida, ele fica no estado bloqueado (*blocked*) e só volta a disputar o processador após a conclusão da chamada.

A mudança de estado de qualquer processo é iniciada por um evento. Esse evento aciona o sistema operacional, que então altera o estado de um ou mais processos. A transição do estado executando para bloqueado é feita através de uma chamada de sistema. Uma chamada de sistema é necessariamente feita pelo processo no estado executando. Ele fica no estado bloqueado até o atendimento. Com isso, o processador fica livre. O sistema operacional então seleciona um processo da fila de aptos (*ready*) para receber o processador. O processo selecionado passa do estado de apto para o estado executando. O módulo do

sistema operacional que faz essa seleção é chamado escalonador (*scheduler*) (OLIVEIRA, 2001).

Alguns outros caminhos também são possíveis no grafo de estados. Algumas chamadas de sistema são muito rápidas. Por exemplo, leitura da hora atual. Não existe acesso a periférico, mas apenas consulta às variáveis do próprio sistema operacional. Nesse caso, o processo não precisa voltar para a fila de apto, ele simplesmente retorna para a execução após a conclusão da chamada. Muitos sistemas procuram evitar que um único processo monopolize a ocupação do processador. Se um processo está há muito tempo no processador, ele volta para o fim da fila de aptos. Um novo processo da fila de aptos ganha o processador. Esse mecanismo cria um caminho entre o estado executando e o estado apto. (OLIVEIRA, 2001).

Um processo é uma abstração que reúne uma série de atributos como espaço de endereçamento, descritores de arquivos abertos, permissões de acesso, quotas, etc. Um processo possui ainda áreas de código, dados e pilha de execução. Também é associado ao processo um fluxo de execução. Por sua vez, uma *thread* nada mais é que um fluxo de execução. Na maior parte das vezes, cada processo é formado por um conjunto de recursos mais uma única *thread*. A ideia de *multithreading* é associar vários fluxos de execução (várias *threads*) a um único processo. Em determinadas aplicações, é conveniente disparar várias *threads* dentro do mesmo processo (programação concorrente). É importante notar que as *threads* existem no interior de um processo, compartilhando entre elas os recursos do processo, como o espaço de endereçamento (código e dados). Devido a essa característica, a gerência de *threads* (criação, destruição, troca de contexto) é "mais leve" quando comparada com processos. Threads são muitas vezes chamadas de processos leves, em que seus únicos atributos próprios estão ligados ao contexto de execução, isto é, aos registradores do processador, os demais atributos estão vinculados ao processo que a contém (FARINES, 2000).

Aplicações de tempo real são usualmente organizadas na forma de várias *threads,* ou tarefas concorrentes. Logo, um requisito básico para os sistemas

operacionais de tempo real é oferecer suporte para processos e *threads*. Embora programas concorrentes possam ser construídos a partir de processos, o emprego de *threads* aumenta a eficiência do mesmo. Devem ser providas chamadas de sistema para criar e destruir processos e *threads*, suspender e retomar processos e *threads*, além de chamadas para manipular o seu escalonamento.

As *threads*, assim como os processos, também podem assumir diferentes estados, como pode ser visto na figura 2:

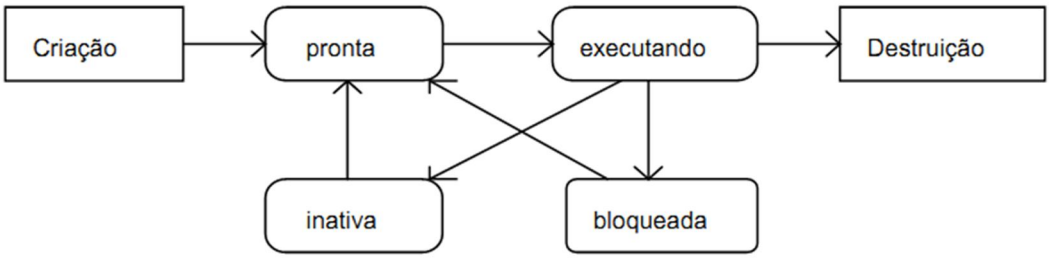

Figura 3- Estados de uma *thread*. Fonte (Farines, 2000)

Sua organização é similar à de um processo, mas com um tipo especial de bloqueio, a situação na qual a *thread* solicita sua suspensão por um intervalo de tempo, ou até uma hora futura pré-determinada, esse é o estado inativo da *thread*. Ela voltará a ficar pronta quando chegar o momento certo. Este estado é típico de uma *thread* com execução periódica, quando a ativação atual já foi concluída e ela aguarda o instante da próxima ativação. Threads periódicas também podem ser implementadas através da criação de uma nova *thread* no início de cada período e de sua destruição tão logo ela conclua seu trabalho relativo àquele período. Entretanto, o custo (*overhead*) associado com a criação e a destruição de *threads* é maior do que o custo associado com a suspensão e reativação, resultando em um sistema menos eficiente (FARINES, 2000).

2.1.3 Escalonamento

No escalonamento do processador é decidido qual processo será executado a seguir. Na escolha de um algoritmo de escalonamento, utiliza-se como critério básico o objetivo de aumentar a produtividade do sistema e, ao mesmo tempo, diminuir o tempo de resposta percebido pelos usuários. Esses dois objetivos podem tornar-se conflitantes em determinadas situações. Variância

elevada significa que a maioria dos processos recebe um serviço (tempo de processador) satisfatório, enquanto alguns são bastante prejudicados. Provavelmente, será melhor sacrificar o tempo médio de resposta para homogeneizar a qualidade do serviço que os processos recebem.

Quando os processos de um sistema possuem diferentes prioridades, essa prioridade pode ser utilizada para decidir qual processo é executado a seguir. Considere o cenário no qual números menores indicam processos mais importantes. Um processo com prioridade 5 (vamos chamá-lo P5) está executando. Nesse momento, termina o acesso a disco de um processo com prioridade 2 (P2), e ele está pronto para iniciar um ciclo de processador. Nessa situação, o sistema pode comportar-se de duas formas distintas. O processo que chega na fila do processador respeita o ciclo de processador em andamento, ou seja, o P2 será inserido normalmente na fila. O processo em execução somente libera o processador no final do ciclo de processador. Tem-se nesse caso a "prioridade não-preemptiva". O processo P2, por ser mais importante que o processo em execução, recebe o processador imediatamente. O processo P5 é inserido na fila conforme a sua prioridade. Essa solução é conhecida como "prioridade preemptiva" (OLIVEIRA 2001).

Existem diversos outros tipos de escalonamento, dentre os quais, se destaca o *Round-Robin*, é um escalonamento baseado em fatias de tempo (*time slice*), ou seja, as tarefas compartilham o mesmo tempo de execução da CPU. Um contador geralmente realiza essa contagem de tempo a cada *tick* do relógio, quando uma fatia de tempo acaba, o contador é zerado e a tarefa que estava sendo realizada é colocada no fim da fila. Se uma nova tarefa entra no processo, ela é colocada no fim da fila.

Geralmente, os escalonamentos do tipo *Round-Robin* são feitos com preempção. Nesse caso, se uma tarefa de maior prioridade entra em estado *ready*, a tarefa que está sendo executada para, a sua contagem é guardada, a tarefa de maior prioridade é executada e após isso a tarefa antiga é retomada a partir do ponto onde ela havia parado (GIRIO, 2010)

2.1.4 Semáforos

Semáforo é um objeto do *kernel* que uma ou mais tarefas podem adquirir e liberar com o propósito de sincronização ou exclusão mútua. Um semáforo é como uma chave que permite uma tarefa realizar alguma operação ou acessar um recurso. O semáforo tem associado a ele um bloco de controle (SCB – *Semaphore Control Block*), um único ID, um valor (binário ou contador) e uma lista de tarefas em espera (LI, 2003).

Os principais tipos de semáforos, descritos por Li, são:

- Binário:

Semáforo que pode assumir os valores 0 (não disponível) e 1 (disponível). Na inicialização ambos os valores podem ser assumidos.

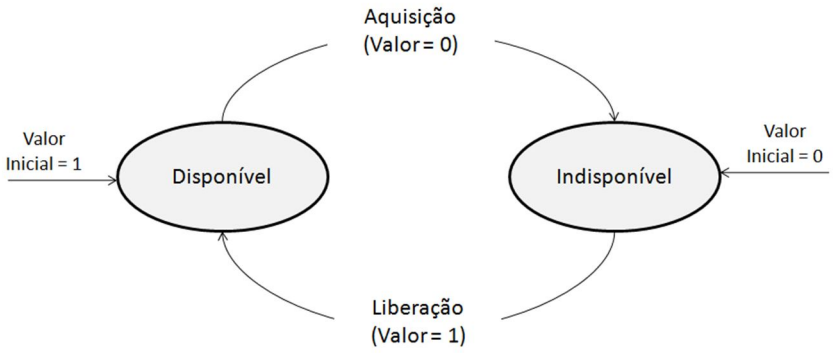

Figura 4- Máquina de Estados do Semáforo Binário. Fonte (Li,2003)

- Contadores:

Semáforo que utiliza um contador para permitir que aquisição e a liberação sejam realizadas múltiplas vezes. O contador é inicializado no momento da criação. Quando o valor do contador atinge 0 o semáforo torna-se indisponível, caso contrário pode ser adquirido. Uma vez em 0 o contador precisa esperar um semáforo ser liberado para se tornar disponível.

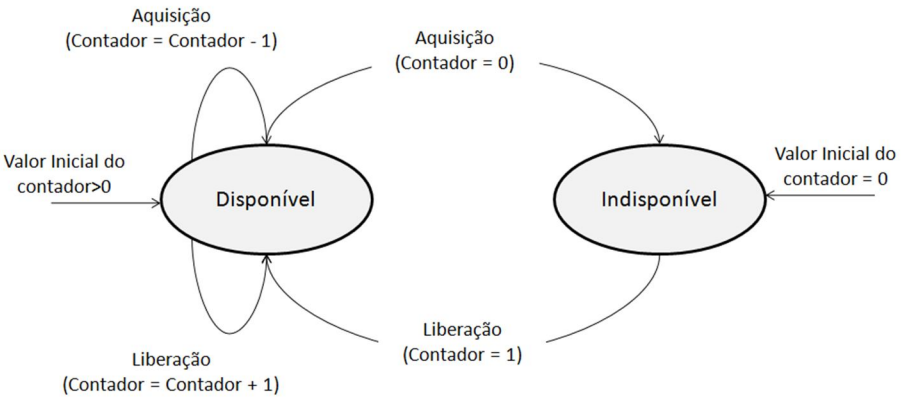

Figura 5- Máquina de Estados do Semáforo Contador. Fonte (Li, 2003)

- Exclusão mútua (*mutex*):

Consiste em um semáforo binário com protocolos de exclusão mútua tais como propriedade, acesso recursivo, exclusão segura de tarefas dentre outros para evitar problemas de exclusão mútua.

Um semáforo de exclusão mútua apresenta estados 0 (bloqueado) e 1 (desbloqueado). Na criação o semáforo é iniciado em estado desbloqueado. A partir do momento que uma tarefa o adquire (ou bloqueia) se torna bloqueado, até ser liberado pela tarefa ou desbloqueado.

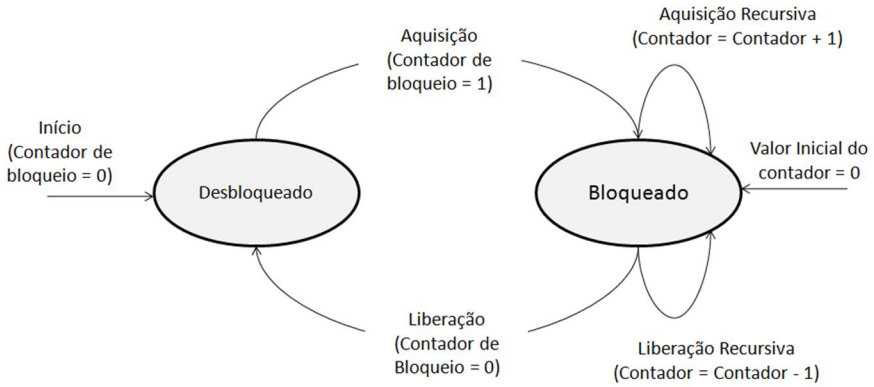

Figura 6- Máquina de Estado do Semáforo de Exclusão Mútua (Mutex). Fonte(Li,2003)

Semáforos são úteis para sincronização, execução de tarefas múltiplas ou coordenação de acesso a recursos compartilhados.

2.1.5 Processo de interrupção

Sistemas de tempo-real são reativos por natureza, e costumam responder a eventos externos via interrupção. O *hardware* reage ao evento transferindo

controle para uma rotina de tratamento de interrupção (ISR, do inglês *interrupt service routine*) feita pelo RTOS ou pelo usuário. Um RTOS normalmente salvaria o contexto da *thread* interrompida e executaria a *thread* de maior prioridade (tanto pode ser a *thread* interrompida, ou uma nova *thread* posta na fila dos prontos pela ISR) após a conclusão da ISR.

Um processo de interrupção geralmente suspende os *threads* ativos, salva os dados relativos ao *thread* suspenso, transfere o controle para uma ISR, faz um processamento na ISR para determinar a ação a ser tomada, recebe e salva qualquer dado importante que chegue e seja associado com a interrupção. Configura qualquer valor (de saída) específico de um dispositivo. Além disso, limpa as interrupções do hardware para permitir que novas interrupções sejam reconhecidas, transfere o controle para a *thread* selecionada, incluindo os dados, que foram salvos do contexto de sua última interrupção.

A implementação desses processos em um RTOS fazem uma diferença significativa na performance de um sistema de tempo-real. (LAMIE, 2007, tradução nossa).

2.1.6 Fila de mensagens

As filas de mensagens se assemelham a buffers que servem como meio de tarefas e interrupções (ISR's) enviarem e receberem mensagens para comunicação e sincronização de dados. As filas armazenam mensagens até o destinatário estar pronto para lê-las (LI, 2003).

Uma fila de mensagens tem associada a ela um bloco de controle, um nome, um único ID, *buffers* de memória, tamanho da fila, tamanho máximo das mensagens e uma lista de tarefas em espera (LI, 2003).

As mensagens podem conter diferentes informações como dados de sensores, imagens a serem projetadas em um display ou um evento de teclado (LI,2003).

2.1.7 Gerenciamento de Memória

O Gerenciamento de memória organiza a hierarquia de memórias. O gerenciador de memórias mantém o controle sobre quais partes da memória estão em uso e quais não estão, com essa informação ele aloca memória aos processos que precisam dela e libera a memória quando o processo é executado, além de gerenciar a troca de processos (*swapping*) entre memória e disco quando a memória principal não é suficiente para conter todos os processos (TANEMBAUM, 2003)

2.2 FreeRTOS

O *FreeRTOS* é um sistemas operacional de tempo-real pequeno, simples e de fácil uso. O seu código fonte, feito em C com partes em assembler, é aberto e possui aproximadamente 2.242 linhas de código, que são basicamente distribuídas em quatro arquivos (task.c, queue.c, croutine.c e list.c). Além disso, outra característica marcante desse sistema esta na sua portabilidade, sendo o mesmo oficialmente portável para 29 arquiteturas diferentes, entre elas a PIC, ARM, Zilog Z80 e PC, arquiteturas bastante utilizadas pelas indústrias de microeletrônica.

Por ser um sistema operacional, o *FreeRTOS* comporta-se como uma camada abstrata localizada entre a aplicação e o hardware essa camada tem como principal objetivo esconder da aplicação detalhes do *hardware* que será utilizado. Tornando o desenvolvimento das aplicações de tempo real mais simples e prático, pois o usuário necessita apenas preocupar-se com as funcionalidades do sistema deixando os demais detalhes de *hardware* para o *FreeRTOS*.(Galvão, 2009)

2.3 BRTOS

BRTOS é um sistema operacional de tempo real desenvolvido por Gustavo Weber Denardin, professor da Universidade Tecnológica Federal do Paraná – Campus Pato Branco (atualmente doutorando em Eng. Elétrica na Universidade Federal de Santa Maria) e por Carlos Henrique Barriquello, também doutorando em Eng. Elétrica na Universidade Federal de Santa Maria.

O BRTOS foi desenvolvido com o intuito de ser um sistema operacional simples, com o mínimo de consumo de memória de dados e programa. Isto deve-se ao reduzido poder computacional e memória disponíveis nos microcontroladores de baixo custo de 8, 16 e 32 bits disponíveis no mercado. Através da utilização de um RTOS torna-se possível desenvolver aplicações mais organizadas e estruturadas, gerenciando de forma eficiente o ambiente de tarefas concorrentes, inerente a sistemas embarcados.

A utilização de um Sistema Operacional de Tempo-Real provê ao desenvolvedor de sistemas embarcados um conjunto de ferramentas que além de facilitar o desenvolvimento, torna o código gerado mais portável, permitindo a mudança de plataforma caso seja necessário. Imagine que você possua um produto utilizando um microcontrolador que deixou de ser fabricado. Se o produto foi desenvolvido de forma não portável, o tempo de adaptação para outro microcontrolador disponível no mercado será maior.

Projetos de sistemas embarcados de menor complexidade ainda utilizam a linguagem de programação Assembly. Embora seja possível criar um código altamente otimizado, esta linguagem é extremamente dependente da CPU utilizada, além de não ser estruturada. Este tipo de linguagem dificulta muito a portabilidade do sistema desenvolvido para outra plataforma. Devido a este motivo, a linguagem de programação utilizada no projeto foi o C. Mesmo o código sendo desenvolvido em C, é possível que o código não seja 100% compatível com todos os ambientes de programação existentes no mercado. (BRTOS,2010)

2.4 Considerações Finais

Esse capítulo apresentou alguns conceitos pertinentes ao entendimento do restante da monografia. O processo de desenvolvimento para o alcance dos resultados pode ser visto no capítulo que se segue.

Capítulo 3 – Materiais e Método

A performance de um Sistema Operacional é sensível à plataforma em que foi desenvolvido, ao processador utilizado, à velocidade do *clock*, ao compilador e ao design empregado (LAMIE, 2007).

Para que os testes não priorizem um Sistema Operacional de Tempo-Real em detrimento de outro, optou-se por um *benchmark* free-source, facilmente adaptável para diferentes RTOSes, a *Thread-Metric*. A análise é feita através de um conjunto de sete testes que avaliam a velocidade de um RTOS na execução de tarefas normalmente realizadas nos diferentes RTOSes disponíveis.

3.1 Materiais

3.1.1 Thread-Metric

A *Thread-Metric* da Threadx é uma suíte de *benchmark open source* produzida para se ajustar a diferentes RTOSes.

Essa suíte consiste em 10 arquivos:

- tm_api.h:

API e constantes dos testes.

- tm_basic_processing_test.c:

Teste básico para determinar as capacidades do processador.

- tm_cooperative_scheduling_test_c:

Teste de escalonamento cooperativo.

- tm_preemptive_scheduling_test_c:

Teste de escalonamento com preempção.

- tm_interrupt_processing_test.c:

Teste de processamento de interrupção sem preempção.

- tm_interrupt_preemption_processing_test.c:

Teste de processamento de interrupção com preempção.

- tm_message_processing_test.c:

Teste de processamento de troca de mensagem.

- tm_synchronization_processing_test.c:

Teste do processamento do semáforo.

- tm_memory_allocation_test.c:

Teste de alocação de memória.

- tm_porting_layer.c:

Provém uma shell genérica para a portabilidade do RTOS.

Pensando na portabilidade, *Thread-Metric* é feito para usar uma Interface de Programação de Aplicativo (API) simples que pode ser modificada de RTOS pra RTOS no arquivo tm_api.h. O arquivo tm_porting_layer.c contém todas as *shells* utilizadas pelos testes de verdade, as *shells* traduzem os testes para os diferentes RTOSes, portanto, precisam ser adaptados para o RTOS específico.

Por exemplo, tem-se o serviço realizado pelo tm_basic_processing_test.c na figura 6 abaixo:

```
/* Define the basic processing test initialization.  */

void  tm_basic_processing_initialize(void)
{

    /* Create thread 0 at priority 10.  */
    tm_thread_create(0, 10, tm_basic_processing_thread_0_entry);

    /* Resume thread 0.  */
    tm_thread_resume(0);

    /* Create the reporting thread. It will preempt the other
       threads and print out the test results.  */
    tm_thread_create(5, 2, tm_basic_processing_thread_report);
    tm_thread_resume(5);
}
```

Figura 7- Serviço realizado pelo tm_basic_processing_test.c

Dentro dela pode-se perceber que são usadas duas funções: tm_tread_create e tm_thread_resume, essas duas funções têm seu funcionamento descrito no tm_porting_layer.c como encontra-se na figura 7:

```
/* This function takes a thread ID and priority and attempts to create the
   file in the underlying RTOS.  Valid priorities range from 1 through 31,
   where 1 is the highest priority and 31 is the lowest. If successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be returned.   */
int  tm_thread_create(int thread_id, int priority, void (*entry_function)(void))
{

}


/* This function resumes the specified thread.  If successful, the function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_thread_resume(int thread_id)
{

}
```

Figura 8- Funções explicadas no tm_porting_layer.c

Deve-se usar essas instruções e criar uma função, com o mesmo nome e os mesmos parâmetros, para cada RTOS que se deseja testar, como, por exemplo, para o BRTOS, visto na figura 8:

```
/* ------------------------------------------------------------------------------------

/* This function takes a thread ID and priority and attempts to create the
   file in the underlying RTOS.  Valid priorities range from 1 through 31,
   where 1 is the highest priority and 31 is the lowest. If successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be returned.   */
int  tm_thread_create(int thread_id, int priority, void (*entry_function)(void))
{
    (void)thread_id; // stops compiler warnings
    if(InstallTask(entry_function,NULL,TM_BRTOS_THREAD_STACK_SIZE,(INT8U)priority)){
        return TM_ERROR;
    }

    if(BlockPriority((INT8U)priority)){
        return TM_ERROR;
    }
    return TM_SUCCESS;
}

/* ------------------------------------------------------------------------------------

/* This function resumes the specified thread.  If successful, the function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_thread_resume(int thread_id)
{
    // we do not use id, but priority instead
    if(UnBlockPriority((INT8U)thread_id)){
        return TM_ERROR;
    }
    return TM_SUCCESS;
}

/* ------------------------------------------------------------------------------------
```

Figura 9- Exemplo de função criada para o porting layer

Os códigos dos testes, assim como os códigos do *porting layer* do BRTOS e do *FreeRTOS* se encontram nos anexos (1, 2 e 3).

3.2 Metodologia

A análise consiste na realização de sete testes que são pertinentes às características usuais de um RTOS:

- Troca de contexto cooperativo
- Troca de contexto preemptivo
- Processamento de interrupção
- Processamento de interrupção com preempção
- Passagem de mensagem
- Processamento de semáforo
- Alocação e desalocação de memória

Cada teste é programado para durar trinta segundos, ao fim de cada tarefa, o contador do programa é incrementado, ao final do teste, quanto maior for o número do contador, mais vezes a tarefa foi realizada e melhor o desempenho do Sistema Operacional nesse quesito.

3.2.1 Troca de contexto Cooperativo

Nesse teste é avaliada a velocidade com que o escalonador de um RTOS troca o contexto de execução de uma tarefa para outra de mesma prioridade. Para tanto cinco tarefas são criadas com a mesma prioridade. Cada *Thread* roda até sua conclusão, incrementa o contador e cede a vez à próxima *thread* em um escalonamento *round-robin.*

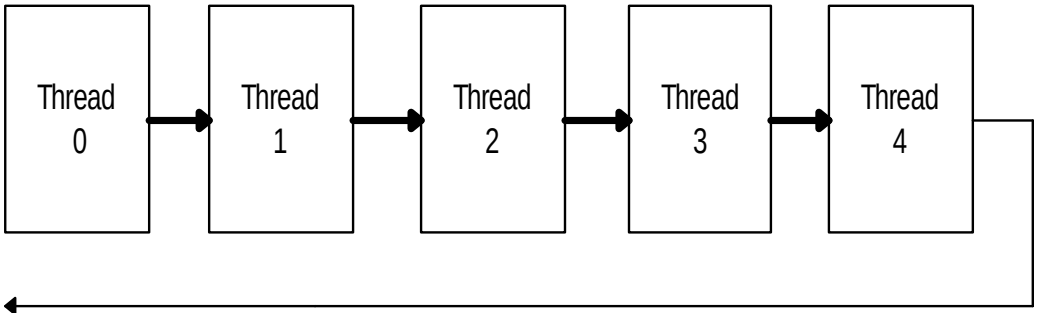

Figura 10- Teste de troca de contexto cooperativo. Retrabalhado de (LAMIE)

3.2.2 Troca de contexto Preemptivo

Esse teste é o equivalente ao teste descrito anteriormente, mas utilizando a preemptividade, ou seja, as *threads* não possuem prioridades iguais. As cinco *thread* agora são criadas com prioridades diferentes. Uma *thread* executa até ser preterida por uma *thread* de maior prioridade. Todas as *threads* iniciam suspensas

(menos a de menor prioridade). A *thread* de menor prioridade ativa a *thread* de prioridade imediatamente maior, e assim por diante, até que a *thread* com maior prioridade das cinco seja executada. Cada *thread* incrementa seu contador de execução e se suspende. O processo é reiniciado quando o processador retorna para a *thread* de menor prioridade que incrementa seu contador, vai pra *thread* com prioridade imediatamente maior, iniciando todo o processo novamente.

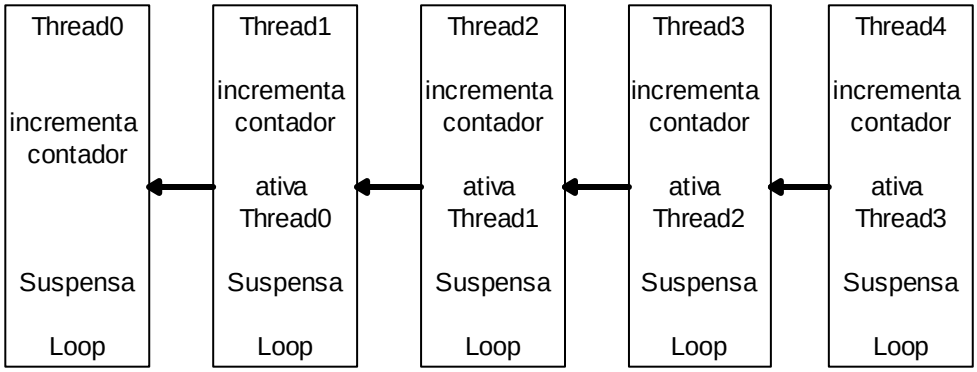

Figura 11- Teste de troca de contexto preemptivo. Retrabalhado de (LAMIE.2007)

3.2.3 Processamento de interrupção sem preempção

O processamento de interrupção consiste em saltar para uma interrupção, executar a rotina de tratamento desta interrupção, executar o escalonador para determinar qual *thread* deverá ser executada, restaurar o contexto desta *thread* (caso necessário) e saltar para a execução desta *thread*.

Neste teste não há preempção, a *thread* interrompida volta a ser executada. O tempo total medido é o tempo em que uma *thread* não está sendo executada. Muitos RTOSes anunciam, ou fazem propaganda de baixa latência de interrupção, mas deixam a desejar na latência das tarefas (*threads*). Ambas as latências são críticas, sendo o tempo total o que realmente importa. O teste realizado considera a latência de interrupção e a sobrecarga para ativação das *threads*.

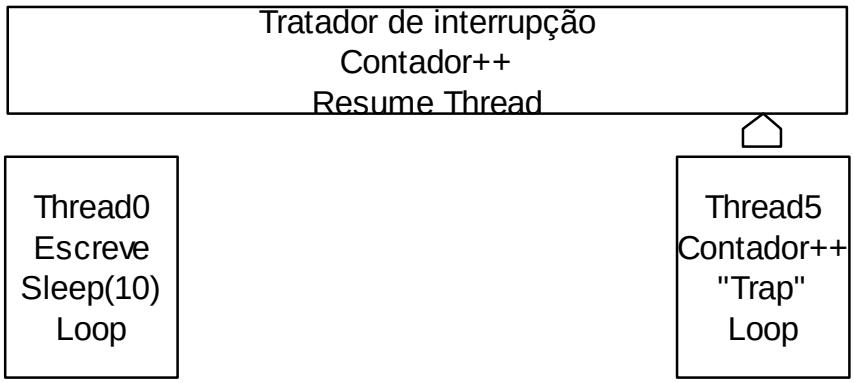

Figura 12- Teste de Processamento de interrupção sem preempção. Retrabalhado de (LAMIE, 2007)

3.2.4 Processamento de interrupção com preempção

Esse teste mede o desempenho de um RTOS em situações onde uma *thread* diferente é selecionada pelo escalonador para ser executada na saída de uma interrupção. Este caso ocorre quando o tratamento de uma interrupção adiciona à lista de *ready* uma *thread* de maior prioridade que a *thread* que estava sendo executada. O tempo para executar o salvamento de contexto da *thread* interrompida e restaurar o contexto para uma nova *thread* é incluído na pontuação deste teste. Para realizar este teste a Thread1 se suspende, permitindo que a Thread5 execute e force a interrupção por *software* "*trap*". O alimentador de interrupção acorda a Thread1. Ou seja, o escalonador troca o contexto da Thread5 para a Thread1 e a Thread1, de maior preemptividade, é executada após a interrupçao.

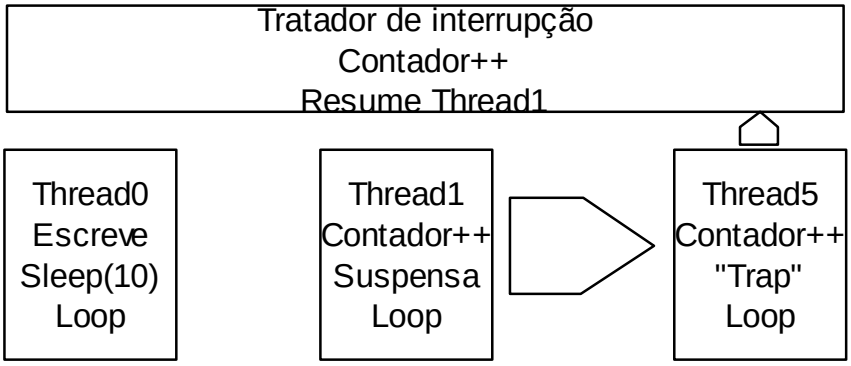

Figura 13- Teste de processamento de interrupção com preempção. Retrabalhado de (LAMIE, 2007)

3.2.5 Passagem de mensagem

Esse teste mede o desempenho com que um RTOS leva para passar uma mensagem de 16-bytes por valor, ou seja, copiada da origem para uma fila e recebida por uma *thread*. Mensagens maiores do que 16 bytes geralmente são passadas por referência (o ponteiro é enviado ao invés dos dados). Para realizar este teste uma *thread* envia uma mensagem de 16 bytes para uma fila e retira desta fila os mesmos 16 bytes. A *thread* incrementa um contador depois de enviar e receber a sequência completa de dados.

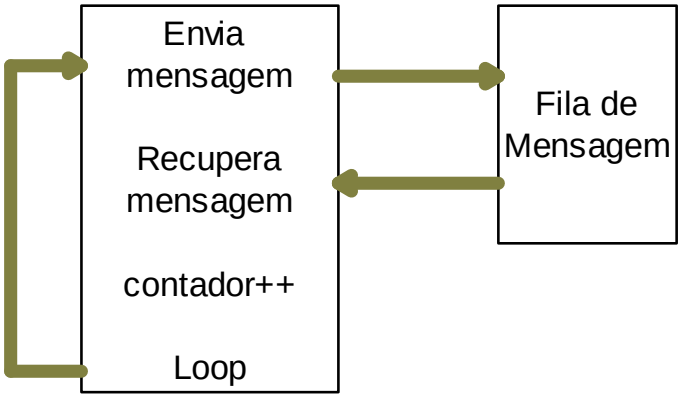

Figura 14- Teste de passagem de mensagem. Retrabalhado de (LAMIE, 2007)

3.2.6 Processamento de semáforo

Esse teste mede o desempenho de um RTOS na “obtenção” e “liberação” de um semáforo. Para realizar este teste uma tarefa obtém um semáforo e imediatamente o libera. Depois deste procedimento a tarefa incrementa o contador de execução.

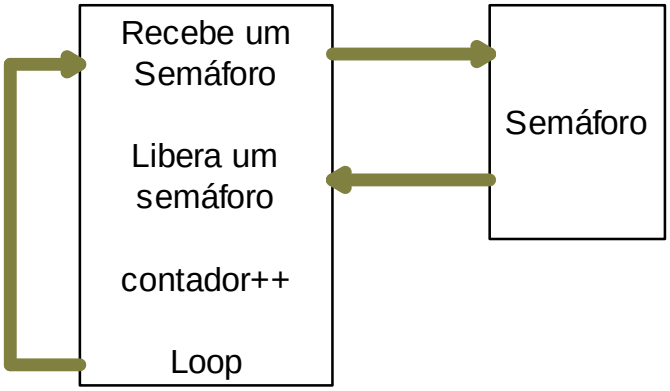

Figura 15- Teste de processamento de semáforo. Retrabalhado de (LAMIE,2007)

3.2.7 Alocação e desalocação de memória

Mede o tempo que leva para um RTOS alocar um bloco de memória de tamanho fixo (128 bytes) para uma tarefa. Se um RTOS não possuir gerenciamento de blocos de memória a função malloc do Ansi C deverá ser utilizada (provavelmente obtendo baixo desempenho). Para realizar este teste uma tarefa aloca um bloco de memória de 128 bytes e logo em seguida desaloca este mesmo bloco. Depois de liberar o bloco de memória a tarefa incrementa o contador de execução.

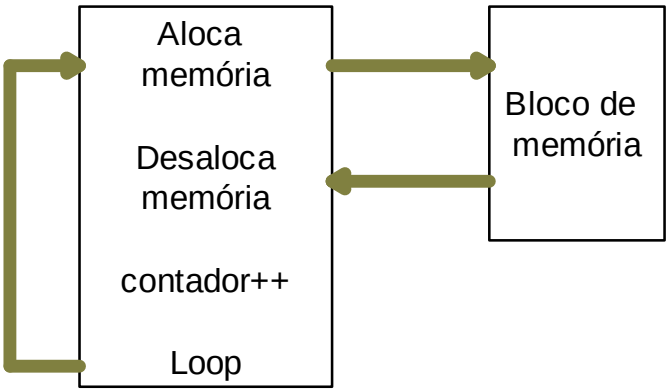

Figura 16- Teste de alocação de memória. Retrabalhado de (LAMIE, 2007)

Estas informações foram coletadas nos sites do BRTOS e da EETimes.

Capítulo 4 – Resultados e Conclusões

Foram escolhidos dois Sistemas Operacionais de Tempo- Real para serem testados utilizando a ferramenta *Thread-Metric,* o *FreeRTOS* e o BRTOS.

O FreeRTOS foi escolhido devido à sua popularidade, sendo fácil encontrar material publicado ao seu respeito, o FreeRTOS também se destaca por sua portabilidade, umas vez que já se encontra adaptado a 29 arquiteturas.

O BRTOS não é tão popular, mas é muito bem documentado e desenvolvido por doutorandos brasileiros que estão continuamente implementando novas aplicações e assistindo aos usuários do RTOS.

Este capítulo irá mostrar todos os resultados obtidos dos métodos citados no capítulo 3 e as conclusões tiradas a respeito.

4.1 Resultados

Os testes, têm duração de 30 segundos, eles foram realizados utilizando o programa *CodeWarrior* da *Freescale* mostrado na figura 16:

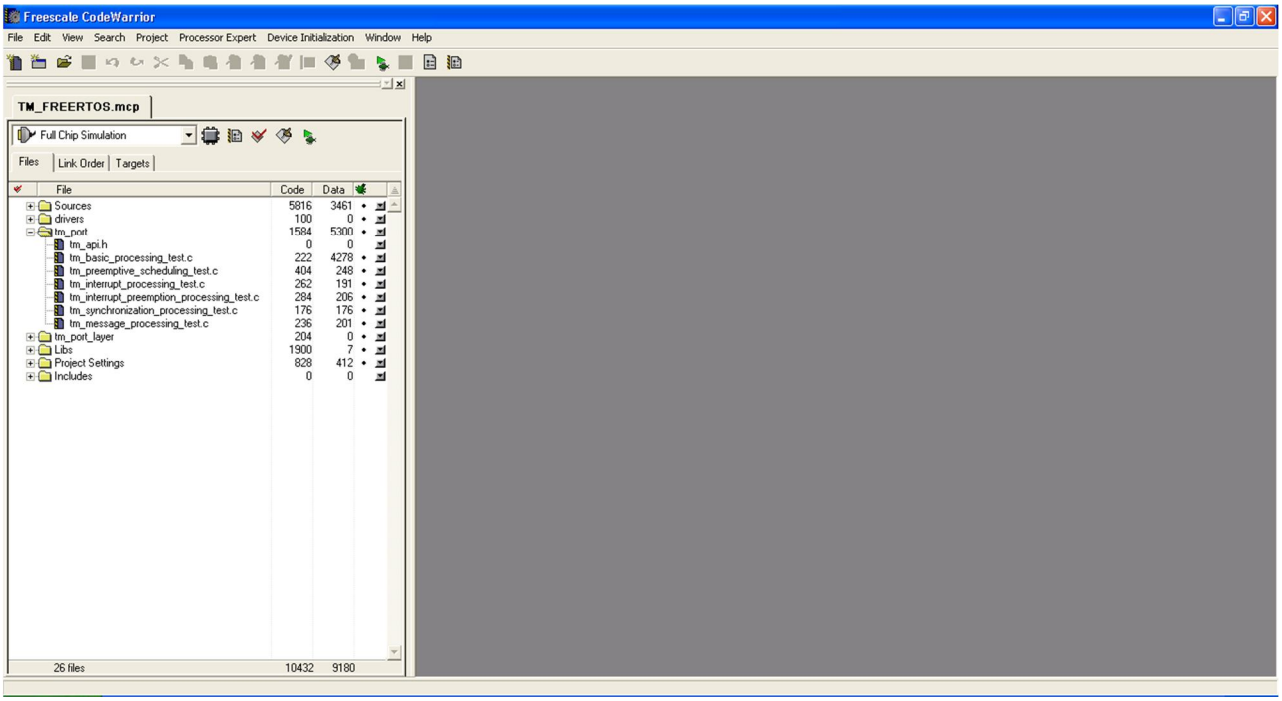

Figura 17- *CodeWarrior* pertencente à *Freescale*

Cada teste quando rodado, tem seu resultado mostrado em um terminal como ilustra a figura 17:

Figura 18- Terminal para a apresentação de dados

Note que há uma diferença de 30 segundos entre cada resultado apresentado e que eles não costumam variar entre si.

Os testes de troca de contexto cooperativo e alocação e desalocação de memória não foram feitos, pois os RTOSes não apresentam essa funcionalidade.

Primeiro foram realizados os testes no *FreeRTOS* versão 6.1.0 utilizando-se um microcontrolador *Freescale ColdFire V1* e obteve-se os seguintes resultados (tabela 1).

Tabela 1- Teste do *FreeRTOS*

Processamento de mensagens	1223565
Processamento de interrupção com preempção	722364
Processamento de interrupção sem preempção	1338999
Troca de contexto preemptivo	1484377
Processamento de semáforo	2529635
Processamento básico (x100)	4597600

Utilizando-se agora o BRTOS versão 1.6.5 e o mesmo microcontrolador com a mesma taxa de *clock*, obteve-se o resultado disposto na tabela 2:

Tabela 2- Teste do BRTOS

Processamento de mensagens	926802
Processamento de interrupção com preempção	1827156

Processamento de interrupção sem preempção	3210365
	2410233
Processamento de semáforo	5312494
Processamento básico (x100)	4602100

Uma comparação gráfica entre os desempenhos pode ser vista abaixo (figura 18):

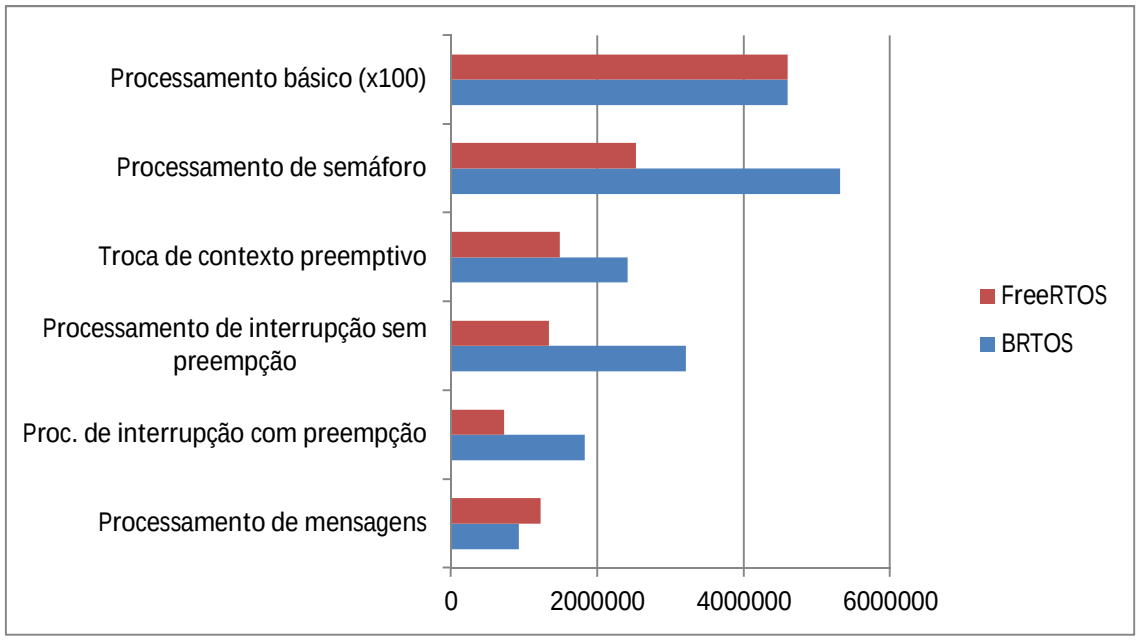

Figura 19- Comparação entre os desempenhos do BRTOS e *FreeRTOS*

O que está sendo medido é o valor apresentado pelo contador do programa, que indica quantas vezes a tarefa foi executada, portanto, o teste indica uma superioridade do BRTOS na maioria dos testes, exceto no processamento de mensagens, no qual o *FreeRTOS* leva vantagem.

Além de comparar a atuação de dois RTOSes em um mesmo microcontrolador, também é possível comparar a performance de um mesmo RTOS em microcontroladores diferentes, para tanto foi realizado o teste utilizando o BRTOS e dois microcontroladores, o *ColdFire V1* do teste anterior e o *HCS08* também da *freescale*. Por falta de RAM não foi feito o teste de processamento básico no *HCS08*, o resultado encontra-se na tabela 3 e na figura 19.

Tabela 3- Testes nos microcontroladores *ColdFire V1* e *HCS08*

	ColdFire V1	*HCS08*
Processamento de mensagens	926802	143335

Proc. de interrupção com preempção	1827156	1486721
Processamento de interrupção sem preempção	3210365	1709353
Troca de contexto preemptivo	2410233	810126
Processamento de semáforo	5312494	1702058

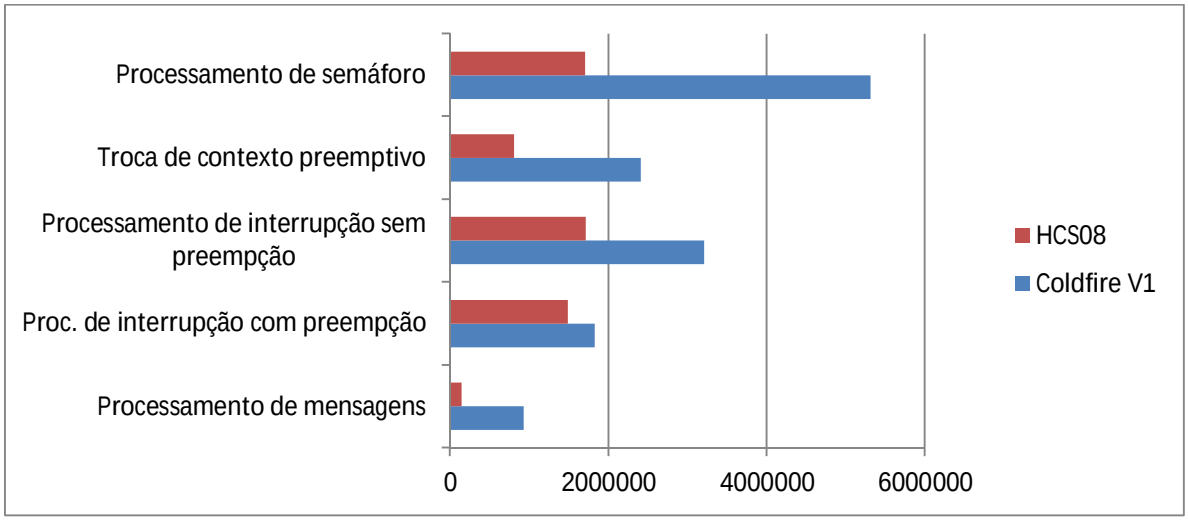

Figura 20- Comparação entre os desempenhos do *ColdFire V1* e do *HCS08*

Em todos os testes o *ColdFire* se revelou superior, mas com algumas diferenças relativas maiores em uns quesitos, como no quesito processamento de mensagens em que o *ColdFire* foi quase 6,5 vezes superior, no entanto, eles quase se equipararam em outros testes, como no Processo de interrupção com preempção.

4.2 Conclusões Gerais

A intenção do trabalho não é fazer campanha a favor de um, ou outro RTOS, uma vez que diferentes RTOSes servirão para o mesmo propósito, e nem todos os projetos exigem o máximo de sua performance. Dito isso, através da busca por um método de análise de desempenho, pode-se observar as características recorrentes dentre distintos Sistemas Operacionais, tais como: processos e *threads*, escalonamento, semáforos, processo de interrupção, fila de mensagens e gerenciamento de memórias.

Utilizando a *suite Thread-Metric*, constatou-se que alguns testes podem ter melhores resultados em um RTOS, como a superioridade apresentada pelo *FreeRTOS* no processamento de mensagens, e o mesmo RTOS apresentar

desempenho inferior em outros quesitos, pois viu-se que nos demais testes o BRTOS foi superior, o que deixa patente a necessidade de uma compreensão do projeto antes de escolher o RTOS. Foi mostrado que também é possível utilizar o teste para comparar os microcontroladores, caso esse seja o enfoque que queira se dar. O trabalho é pertinente para quem deseja conhecer os conceitos que envolvem um RTOS.

Referências Bibliográficas

DE OLIVEIRA, Rômulo Silva; CARISSIMI, Alexandre da Silva; TOSCANI, Simão Sirineo. Sistemas Operacionais. **Revista de Informática Teórica e Aplicada - RITA -**. [S.l : s.n], v. 8, n. 3, Dezembro, 2001.

DENARDIN, Gustavo Weber. O que é o BRTOS?. 06, Outubro, 2010. Disponível em < http://brtosblog.wordpress.com/2010/10/06/o-que-e-o-brtos/ > Acesso em: 23, Outubro, 2011

FARINES, Jean-Marie; FRAGA, Joni da Silva; DE OLIVEIRA, Rômulo Silva. **Sistemas de Tempo Real.** Florianópolis, Escola de Computação, 2000.

GALVÃO, Stephenson de S.L. **Modelagem do Sistema Operacional de Tempo Real *FreeRTOS*.** Natal: UFRN, 2009. Qualificação de Mestrado - Programa de Pós-gradução em Sistemas e Computação, Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte, Natal, 2009.

GIRIO, Gustavo Murilo. **Utilização do Sistema Operacional tempo-real MQX embarcado para aplicações de telemetria**. São Carlos: USP,2010. Trabalho de Conclusão de Curso. Escola de Engenharia de São Carlos, Universidade de São Paulo, São Carlos, 2010.

LAMIE, William; CARBONE, John. Measure your RTOS's real-time performance. 21, Agosto, 2007. Disponível em: < www.eetimes.com/design/embedded/4007081/Measure-your-RTOS-s-real-time-performance > Acesso em: 20, Setembro, 2011

LI, Qing; **Real Time concepts for Embedded Systems**. [S.l]. CMP Books, 2003

TANENBAUM, Andrew S. **Sistemas Operacionais modernos.** 2.ed. São Paulo: Prentice Hall,2003.

Anexo 1 – Código dos testes do *Thread-Metric*

1.1 tm_api.c

```
/**************************************************************************/
/*                                                                        */
/*            Copyright (c) 1996-2007 by Express Logic Inc.               */
/*                                                                        */
/*  This Original Work may be modified, distributed, or otherwise used in */
/*  any manner with no obligations other than the following:              */
/*                                                                        */
/*    1. This legend must be retained in its entirety in any source code  */
/*       copies of this Work.                                             */
/*                                                                        */
/*    2. This software may not be used in the development of an operating */
/*     system product.                                                    */
/*                                                                        */
/*  This Original Work is hereby provided on an "AS IS" BASIS and WITHOUT */
/*  WARRANTY, either express or implied, including, without limitation,   */
/*  the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A  */
/*  PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF this         */
/*  ORIGINAL WORK IS WITH the user.                                       */
/*                                                                        */
/*  Express Logic, Inc. reserves the right to modify this software        */
/*  without notice.                                                       */
/*                                                                        */
/*  Express Logic, Inc.                     info@expresslogic.com         */
/*  11423 West Bernardo Court               http://www.expresslogic.com   */
/*  San Diego, CA  92127                                                  */
/*                                                                        */
```

```
/*
*/
/**************************************************************************
**/

/**************************************************************************
**/
/**************************************************************************
**/
/**
*/
/** Thread-Metric Component
*/
/**
*/
/** Application Interface (API)
*/
/**
*/
/**************************************************************************
**/
/**************************************************************************
**/


/**************************************************************************
**/
/*
*/
/*  APPLICATION INTERFACE DEFINITION                        RELEASE
*/
/*
*/
/*    tm_api.h                                            PORTABLE C
*/
/*                                                          4.1
*/
/*  AUTHOR
*/
/*
*/
/*    William E. Lamie, Express Logic, Inc.
*/
/*
*/
/*  DESCRIPTION
*/
/*
*/
/*    This file defines the basic Application Interface (API)
*/
/*    implementation source code for the Thread-Metrics performance
*/
/*    test suite. All service prototypes and data structure definitions
*/
/*    are defined in this file.
*/
```

```
/*
*/
/*  RELEASE HISTORY
*/
/*
*/
/*    DATE              NAME                        DESCRIPTION
*/
/*
*/
/*  03-01-2004     William E. Lamie         Initial Version 4.0
*/
/*  03-05-2007       William E. Lamie         CMP Release New Banner 4.1
*/
/*
*/
/************************************************************************
**/

#ifndef  TM_API_H
#define  TM_API_H

/* Determine if a C++ compiler is being used.  If so, ensure that
standard
   C is used to process the API information.  */

#ifdef   __cplusplus

/* Yes, C++ compiler is present.  Use standard C.  */
extern   "C" {

#endif

/* Define API constants.  */

#define TM_SUCCESS  0
#define TM_ERROR    1


/* Define the time interval in seconds. This can be changed with a -D
compiler option.  */

#ifndef TM_TEST_DURATION
#define TM_TEST_DURATION    30
#endif


/* Define RTOS Neutral APIs. RTOS vendors should fill in the guts of the
following
   API. Once this is done the Thread-Metric tests can be successfully
run.  */

void   tm_initialize(void (*test_initialization_function)(void));
int    tm_thread_create(int thread_id, int priority, void
(*entry_function)(void));
int    tm_thread_resume(int thread_id);
int    tm_thread_suspend(int thread_id);
```

```
void    tm_thread_relinquish(void);
void    tm_thread_sleep(int seconds);
int     tm_queue_create(int queue_id);
int     tm_queue_send(int queue_id, unsigned long *message_ptr);
int     tm_queue_receive(int queue_id, unsigned long *message_ptr);
int     tm_semaphore_create(int semaphore_id);
int     tm_semaphore_get(int semaphore_id);
int     tm_semaphore_put(int semaphore_id);
int     tm_memory_pool_create(int pool_id);
int     tm_memory_pool_allocate(int pool_id, unsigned char **memory_ptr);
int     tm_memory_pool_deallocate(int pool_id, unsigned char *memory_ptr);


/* Determine if a C++ compiler is being used.  If so, complete the
standard
   C conditional started above.  */
#ifdef   __cplusplus
        }
#endif

#endif
```

1.2 tm_basic_processing_test.c

```
/**************************************************************************
**/
/*
*/
/*            Copyright (c) 1996-2007 by Express Logic Inc.
*/
/*
*/
/*  This Original Work may be modified, distributed, or otherwise used in
*/
/*  any manner with no obligations other than the following:
*/
/*
*/
/*    1. This legend must be retained in its entirety in any source code
*/
/*       copies of this Work.
*/
/*
*/
/*    2. This software may not be used in the development of an operating
*/
/*     system product.
*/
/*
*/
/*  This Original Work is hereby provided on an "AS IS" BASIS and WITHOUT
*/
/*  WARRANTY, either express or implied, including, without limitation,
*/
```

```
/*  the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A
*/
/*  PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF this
*/
/*  ORIGINAL WORK IS WITH the user.
*/
/*
*/
/*  Express Logic, Inc. reserves the right to modify this software
*/
/*  without notice.
*/
/*
*/
/*  Express Logic, Inc.                     info@expresslogic.com
*/
/*  11423 West Bernardo Court               http://www.expresslogic.com
*/
/*  San Diego, CA  92127
*/
/*
*/
/**************************************************************************
**/

/**************************************************************************
**/
/**************************************************************************
**/
/**
*/
/** Thread-Metric Component
*/
/**
*/
/**   Basic Processing Test
*/
/**
*/
/**************************************************************************
**/
/**************************************************************************
**/

#include "tm_api.h"


/* Define the counters used in the demo application...  */

unsigned long   tm_basic_processing_counter;


/* Test array.  We will just do a series of calculations on the
   test array to eat up processing bandwidth. The idea is that
   all RTOSes should produce the same metric here if everything
   else is equal, e.g. processor speed, memory speed, etc.  */
```

```
unsigned long   tm_basic_processing_array[1024];


/* Define the test thread prototypes.  */

void            tm_basic_processing_thread_0_entry(void);


/* Define the reporting thread prototype.  */

void            tm_basic_processing_thread_report(void);


/* Define the initialization prototype.  */

void            tm_basic_processing_initialize(void);


/* Define main entry point.  */

void main()
{

    /* Initialize the test.  */
    tm_initialize(tm_basic_processing_initialize);
}


/* Define the basic processing test initialization.  */

void  tm_basic_processing_initialize(void)
{

    /* Create thread 0 at priority 10.  */
    tm_thread_create(0, 10, tm_basic_processing_thread_0_entry);

    /* Resume thread 0.  */
    tm_thread_resume(0);

    /* Create the reporting thread. It will preempt the other
       threads and print out the test results.  */
    tm_thread_create(5, 2, tm_basic_processing_thread_report);
    tm_thread_resume(5);
}


/* Define the basic processing thread.  */
void  tm_basic_processing_thread_0_entry(void)
{

int     i;

    /* Initialize the test array.   */
    for (i = 0; i < 1024; i++)
    {

        /* Clear the basic processing array.  */
```

```
        tm_basic_processing_array[i] =  0;
    }

    while(1)
    {

        /* Loop through the basic processing array, add the previous
           contents with the contents of the tm_basic_processing_counter
           and xor the result with the previous value...   just to eat
           up some time.  */
        for (i = 0; i < 1024; i++)
        {

            /* Update each array entry.  */
            tm_basic_processing_array[i] =  (tm_basic_processing_array[i]
+ tm_basic_processing_counter) ^ tm_basic_processing_array[i];
        }

        /* Increment the basic processing counter.  */
        tm_basic_processing_counter++;
    }
}


/* Define the basic processing reporting thread.  */
void  tm_basic_processing_thread_report(void)
{

unsigned long   last_counter;
unsigned long   relative_time;


    /* Initialize the last counter.  */
    last_counter =  0;

    /* Initialize the relative time.  */
    relative_time =  0;

    while(1)
    {

        /* Sleep to allow the test to run.  */
        tm_thread_sleep(TM_TEST_DURATION);

        /* Increment the relative time.  */
        relative_time =  relative_time + TM_TEST_DURATION;

        /* Print results to the stdio window.  */
        printf("**** Thread-Metric Basic Single Thread Processing Test
**** Relative Time: %lu\n", relative_time);

        /* See if there are any errors.  */
        if (tm_basic_processing_counter == last_counter)
        {

            printf("ERROR: Invalid counter value(s). Basic processing
thread died!\n");
```

```
        }

        /* Show the time period total.  */
        printf("Time Period Total:  %lu\n\n", tm_basic_processing_counter
- last_counter);

        /* Save the last counter.  */
        last_counter =  tm_basic_processing_counter;
    }
}
```

1.3 tm_cooperative_scheduling_test.c

```
/**************************************************************************
**/
/*
*/
/*            Copyright (c) 1996-2007 by Express Logic Inc.
*/
/*
*/
/*  This Original Work may be modified, distributed, or otherwise used in
*/
/*  any manner with no obligations other than the following:
*/
/*
*/
/*    1. This legend must be retained in its entirety in any source code
*/
/*       copies of this Work.
*/
/*
*/
/*    2. This software may not be used in the development of an operating
*/
/*     system product.
*/
/*
*/
/*  This Original Work is hereby provided on an "AS IS" BASIS and WITHOUT
*/
/*  WARRANTY, either express or implied, including, without limitation,
*/
/*  the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A
*/
/*  PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF this
*/
/*  ORIGINAL WORK IS WITH the user.
*/
/*
*/
/*  Express Logic, Inc. reserves the right to modify this software
*/
/*  without notice.
```

```
*/
/*
*/
/*  Express Logic, Inc.                     info@expresslogic.com
*/
/*  11423 West Bernardo Court               http://www.expresslogic.com
*/
/*  San Diego, CA  92127
*/
/*
*/
/**************************************************************************
**/

/**************************************************************************
**/
/**************************************************************************
**/
/**
*/
/** Thread-Metric Component
*/
/**
*/
/**   Cooperative Scheduling Test
*/
/**
*/
/**************************************************************************
**/
/**************************************************************************
**/

#include "tm_api.h"


/* Define the counters used in the demo application...  */

unsigned long   tm_cooperative_thread_0_counter;
unsigned long   tm_cooperative_thread_1_counter;
unsigned long   tm_cooperative_thread_2_counter;
unsigned long   tm_cooperative_thread_3_counter;
unsigned long   tm_cooperative_thread_4_counter;


/* Define the test thread prototypes.  */

void            tm_cooperative_thread_0_entry(void);
void            tm_cooperative_thread_1_entry(void);
void            tm_cooperative_thread_2_entry(void);
void            tm_cooperative_thread_3_entry(void);
void            tm_cooperative_thread_4_entry(void);


/* Define the reporting thread prototype.  */

void            tm_cooperative_thread_report(void);
```

```c
/* Define the initialization prototype.  */

void            tm_cooperative_scheduling_initialize(void);


/* Define main entry point.  */

void main()
{

    /* Initialize the test.  */
    tm_initialize(tm_cooperative_scheduling_initialize);
}


/* Define the cooperative scheduling test initialization.  */

void  tm_cooperative_scheduling_initialize(void)
{

    /* Create all 5 threads at priority 3.  */
    tm_thread_create(0, 3, tm_cooperative_thread_0_entry);
    tm_thread_create(1, 3, tm_cooperative_thread_1_entry);
    tm_thread_create(2, 3, tm_cooperative_thread_2_entry);
    tm_thread_create(3, 3, tm_cooperative_thread_3_entry);
    tm_thread_create(4, 3, tm_cooperative_thread_4_entry);

    /* Resume all 5 threads.  */
    tm_thread_resume(0);
    tm_thread_resume(1);
    tm_thread_resume(2);
    tm_thread_resume(3);
    tm_thread_resume(4);

    /* Create the reporting thread. It will preempt the other
       threads and print out the test results.  */
    tm_thread_create(5, 2, tm_cooperative_thread_report);
    tm_thread_resume(5);
}


/* Define the first cooperative thread.  */
void  tm_cooperative_thread_0_entry(void)
{

    while(1)
    {

        /* Relinquish to all other threads at same priority.  */
        tm_thread_relinquish();

        /* Increment this thread's counter.  */
        tm_cooperative_thread_0_counter++;
    }
}
```

```c
/* Define the second cooperative thread.  */
void  tm_cooperative_thread_1_entry(void)
{

    while(1)
    {

        /* Relinquish to all other threads at same priority.  */
        tm_thread_relinquish();

        /* Increment this thread's counter.  */
        tm_cooperative_thread_1_counter++;
    }
}

/* Define the third cooperative thread.  */
void  tm_cooperative_thread_2_entry(void)
{

    while(1)
    {

        /* Relinquish to all other threads at same priority.  */
        tm_thread_relinquish();

        /* Increment this thread's counter.  */
        tm_cooperative_thread_2_counter++;
    }
}


/* Define the fourth cooperative thread.  */
void  tm_cooperative_thread_3_entry(void)
{

    while(1)
    {

        /* Relinquish to all other threads at same priority.  */
        tm_thread_relinquish();

        /* Increment this thread's counter.  */
        tm_cooperative_thread_3_counter++;
    }
}


/* Define the fifth cooperative thread.  */
void  tm_cooperative_thread_4_entry(void)
{

    while(1)
    {

        /* Relinquish to all other threads at same priority.  */
        tm_thread_relinquish();
```

```
        /* Increment this thread's counter.  */
        tm_cooperative_thread_4_counter++;
    }
}


/* Define the cooperative test reporting thread.  */
void  tm_cooperative_thread_report(void)
{

unsigned long   total;
unsigned long   relative_time;
unsigned long   last_total;
unsigned long   average;


    /* Initialize the last total.  */
    last_total =  0;

    /* Initialize the relative time.  */
    relative_time =  0;

    while(1)
    {

        /* Sleep to allow the test to run.  */
        tm_thread_sleep(TM_TEST_DURATION);

        /* Increment the relative time.  */
        relative_time =  relative_time + TM_TEST_DURATION;

        /* Print results to the stdio window.  */
        printf("***** Thread-Metric Cooperative Scheduling Test ****
Relative Time: %lu\n", relative_time);

        /* Calculate the total of all the counters.  */
        total =  tm_cooperative_thread_0_counter +
tm_cooperative_thread_1_counter + tm_cooperative_thread_2_counter
                    + tm_cooperative_thread_3_counter +
tm_cooperative_thread_4_counter;

        /* Calculate the average of all the counters.  */
        average =  total/5;

        /* See if there are any errors.  */
        if ((tm_cooperative_thread_0_counter < (average - 1)) ||
            (tm_cooperative_thread_0_counter > (average + 1)) ||
            (tm_cooperative_thread_1_counter < (average - 1)) ||
            (tm_cooperative_thread_1_counter > (average + 1)) ||
            (tm_cooperative_thread_2_counter < (average - 1)) ||
            (tm_cooperative_thread_2_counter > (average + 1)) ||
            (tm_cooperative_thread_3_counter < (average - 1)) ||
            (tm_cooperative_thread_3_counter > (average + 1)) ||
            (tm_cooperative_thread_4_counter < (average - 1)) ||
            (tm_cooperative_thread_4_counter > (average + 1)))
        {
```

```
            printf("ERROR: Invalid counter value(s). Cooperative counters
should not be more that 1 different than the average!\n");
        }

        /* Show the time period total.  */
        printf("Time Period Total:  %lu\n\n", total - last_total);

        /* Save the last total.  */
        last_total =  total;
    }
}
```

1.4 tm_preemptive_scheduling_test.c

```
/**************************************************************************
**/
/*
*/
/*            Copyright (c) 1996-2007 by Express Logic Inc.
*/
/*
*/
/*  This Original Work may be modified, distributed, or otherwise used in
*/
/*  any manner with no obligations other than the following:
*/
/*
*/
/*    1. This legend must be retained in its entirety in any source code
*/
/*       copies of this Work.
*/
/*
*/
/*    2. This software may not be used in the development of an operating
*/
/*     system product.
*/
/*
*/
/*  This Original Work is hereby provided on an "AS IS" BASIS and WITHOUT
*/
/*  WARRANTY, either express or implied, including, without limitation,
*/
/*  the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A
*/
/*  PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF this
*/
/*  ORIGINAL WORK IS WITH the user.
*/
/*
*/
/*  Express Logic, Inc. reserves the right to modify this software
*/
```

```
/*  without notice.                                                       */
/*                                                                        */
/*  Express Logic, Inc.                     info@expresslogic.com         */
/*  11423 West Bernardo Court               http://www.expresslogic.com   */
/*  San Diego, CA  92127                                                  */
/*                                                                        */
/**************************************************************************/

/**************************************************************************/
/**************************************************************************/
/**                                                                       */
/** Thread-Metric Component                                               */
/**                                                                       */
/**   Preemptive Scheduling Test                                          */
/**                                                                       */
/**************************************************************************/
/**************************************************************************/

#include "tm_api.h"


/* Define the counters used in the demo application...  */

unsigned long   tm_preemptive_thread_0_counter;
unsigned long   tm_preemptive_thread_1_counter;
unsigned long   tm_preemptive_thread_2_counter;
unsigned long   tm_preemptive_thread_3_counter;
unsigned long   tm_preemptive_thread_4_counter;


/* Define the test thread prototypes.  */

void            tm_preemptive_thread_0_entry(void);
void            tm_preemptive_thread_1_entry(void);
void            tm_preemptive_thread_2_entry(void);
void            tm_preemptive_thread_3_entry(void);
void            tm_preemptive_thread_4_entry(void);


/* Define the reporting thread prototype.  */
```

```
void            tm_preemptive_thread_report(void);


/* Define the initialization prototype.  */

void            tm_preemptive_scheduling_initialize(void);


/* Define main entry point.  */

void main()
{

    /* Initialize the test.  */
    tm_initialize(tm_preemptive_scheduling_initialize);
}


/* Define the preemptive scheduling test initialization.  */

void  tm_preemptive_scheduling_initialize(void)
{

    /* Create thread 0 at priority 10.  */
    tm_thread_create(0, 10, tm_preemptive_thread_0_entry);

    /* Create thread 1 at priority 9.  */
    tm_thread_create(1, 9, tm_preemptive_thread_1_entry);

    /* Create thread 2 at priority 8.  */
    tm_thread_create(2, 8, tm_preemptive_thread_2_entry);

    /* Create thread 3 at priority 7.  */
    tm_thread_create(3, 7, tm_preemptive_thread_3_entry);

    /* Create thread 4 at priority 6.  */
    tm_thread_create(4, 6, tm_preemptive_thread_4_entry);

    /* Resume just thread 0.  */
    tm_thread_resume(0);

    /* Create the reporting thread. It will preempt the other
       threads and print out the test results.  */
    tm_thread_create(5, 2, tm_preemptive_thread_report);
    tm_thread_resume(5);
}


/* Define the first preemptive thread.  */
void  tm_preemptive_thread_0_entry(void)
{

    while(1)
    {

        /* Resume thread 1.  */
        tm_thread_resume(1);
```

```
        /* We won't get back here until threads 1, 2, 3, and 4 all
execute and
           self-suspend.  */

        /* Increment this thread's counter.  */
        tm_preemptive_thread_0_counter++;
    }
}

/* Define the second preemptive thread.  */
void  tm_preemptive_thread_1_entry(void)
{

    while(1)
    {

        /* Resume thread 2.  */
        tm_thread_resume(2);

        /* We won't get back here until threads 2, 3, and 4 all execute
and
           self-suspend.  */

        /* Increment this thread's counter.  */
        tm_preemptive_thread_1_counter++;

        /* Suspend self!  */
        tm_thread_suspend(1);
    }
}

/* Define the third preemptive thread.  */
void  tm_preemptive_thread_2_entry(void)
{

    while(1)
    {

        /* Resume thread 3.  */
        tm_thread_resume(3);

        /* We won't get back here until threads 3 and 4 execute and
           self-suspend.  */

        /* Increment this thread's counter.  */
        tm_preemptive_thread_2_counter++;

        /* Suspend self!  */
        tm_thread_suspend(2);
    }
}


/* Define the fourth preemptive thread.  */
void  tm_preemptive_thread_3_entry(void)
{
```

```
    while(1)
    {

        /* Resume thread 4.  */
        tm_thread_resume(4);

        /* We won't get back here until thread 4 executes and
           self-suspends.  */

        /* Increment this thread's counter.  */
        tm_preemptive_thread_3_counter++;

        /* Suspend self!  */
        tm_thread_suspend(3);
    }
}


/* Define the fifth preemptive thread.  */
void  tm_preemptive_thread_4_entry(void)
{

    while(1)
    {

        /* Increment this thread's counter.  */
        tm_preemptive_thread_4_counter++;

        /* Self suspend thread 4.  */
        tm_thread_suspend(4);
    }
}


/* Define the preemptive test reporting thread.  */
void  tm_preemptive_thread_report(void)
{

unsigned long   total;
unsigned long   relative_time;
unsigned long   last_total;
unsigned long   average;


    /* Initialize the last total.  */
    last_total =  0;

    /* Initialize the relative time.  */
    relative_time =  0;

    while(1)
    {

        /* Sleep to allow the test to run.  */
        tm_thread_sleep(TM_TEST_DURATION);
```

```
        /* Increment the relative time.  */
        relative_time =  relative_time + TM_TEST_DURATION;

        /* Print results to the stdio window.  */
        printf("**** Thread-Metric Preemptive Scheduling Test ****
Relative Time: %lu\n", relative_time);

        /* Calculate the total of all the counters.  */
        total =  tm_preemptive_thread_0_counter +
tm_preemptive_thread_1_counter + tm_preemptive_thread_2_counter
                    + tm_preemptive_thread_3_counter +
tm_preemptive_thread_4_counter;

        /* Calculate the average of all the counters.  */
        average =  total/5;

        /* See if there are any errors.  */
        if ((tm_preemptive_thread_0_counter < (average - 1)) ||
            (tm_preemptive_thread_0_counter > (average + 1)) ||
            (tm_preemptive_thread_1_counter < (average - 1)) ||
            (tm_preemptive_thread_1_counter > (average + 1)) ||
            (tm_preemptive_thread_2_counter < (average - 1)) ||
            (tm_preemptive_thread_2_counter > (average + 1)) ||
            (tm_preemptive_thread_3_counter < (average - 1)) ||
            (tm_preemptive_thread_3_counter > (average + 1)) ||
            (tm_preemptive_thread_4_counter < (average - 1)) ||
            (tm_preemptive_thread_4_counter > (average + 1)))
        {

            printf("ERROR: Invalid counter value(s). Preemptive counters
should not be more that 1 different than the average!\n");
        }

        /* Show the time period total.  */
        printf("Time Period Total:  %lu\n\n", total - last_total);

        /* Save the last total.  */
        last_total =  total;
    }
}
```

1.5 tm_interrupt_processing_test.c

```
/**************************************************************************
**/
/*
*/
/*            Copyright (c) 1996-2007 by Express Logic Inc.
*/
/*
*/
/*  This Original Work may be modified, distributed, or otherwise used in
*/
/*  any manner with no obligations other than the following:
```

```
*/
/*
*/
/*    1. This legend must be retained in its entirety in any source code
*/
/*       copies of this Work.
*/
/*
*/
/*    2. This software may not be used in the development of an operating
*/
/*     system product.
*/
/*
*/
/*  This Original Work is hereby provided on an "AS IS" BASIS and WITHOUT
*/
/*  WARRANTY, either express or implied, including, without limitation,
*/
/*  the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A
*/
/*  PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF this
*/
/*  ORIGINAL WORK IS WITH the user.
*/
/*
*/
/*  Express Logic, Inc. reserves the right to modify this software
*/
/*  without notice.
*/
/*
*/
/*  Express Logic, Inc.                     info@expresslogic.com
*/
/*  11423 West Bernardo Court               http://www.expresslogic.com
*/
/*  San Diego, CA  92127
*/
/*
*/
/**************************************************************************
**/

/**************************************************************************
**/
/**************************************************************************
**/
/**
*/
/** Thread-Metric Component
*/
/**
*/
/**   Interrupt Processing Test
*/
/**
```

```
*/
/****************************************************************************
**/
/****************************************************************************
**/

#include "tm_api.h"


/* Define the counters used in the demo application...  */

unsigned long   tm_interrupt_thread_0_counter;
unsigned long   tm_interrupt_handler_counter;


/* Define the test thread prototypes.  */

void            tm_interrupt_thread_0_entry(void);
void            tm_interrupt_handler_entry(void);


/* Define the reporting thread prototype.  */

void            tm_interrupt_thread_report(void);


/* Define the interrupt handler.  This must be called from the RTOS.  */

void            tm_interrupt_handler(void);


/* Define the initialization prototype.  */

void            tm_interrupt_processing_initialize(void);


/* Define main entry point.  */

void main()
{

    /* Initialize the test.  */
    tm_initialize(tm_interrupt_processing_initialize);
}


/* Define the interrupt processing test initialization.  */

void  tm_interrupt_processing_initialize(void)
{

    /* Create thread that generates the interrupt at priority 10.  */
    tm_thread_create(0, 10, tm_interrupt_thread_0_entry);

    /* Create a semaphore that will be posted from the interrupt
       handler.  */
    tm_semaphore_create(0);
```

```
    /* Resume just thread 0.  */
    tm_thread_resume(0);

    /* Create the reporting thread. It will preempt the other
       threads and print out the test results.  */
    tm_thread_create(5, 2, tm_interrupt_thread_report);
    tm_thread_resume(5);
}


/* Define the thread that generates the interrupt.  */
void  tm_interrupt_thread_0_entry(void)
{

int status;


    /* Pickup the semaphore since it is initialized to 1 by default. */
    status = tm_semaphore_get(0);

    /* Check for good status.  */
    if (status != TM_SUCCESS)
        return;

    while(1)
    {

        /* Force an interrupt. The underlying RTOS must see that the
           the interrupt handler is called from the appropriate software
           interrupt or trap. */

        asm("trap");  /* This is PowerPC specific.  */

        /* We won't get back here until the interrupt processing is
complete,
           including the setting of the semaphore from the interrupt
           handler.  */

        /* Pickup the semaphore set by the interrupt handler. */
        status = tm_semaphore_get(0);

        /* Check for good status.  */
        if (status != TM_SUCCESS)
            return;

        /* Increment this thread's counter.  */
        tm_interrupt_thread_0_counter++;
    }
}


/* Define the interrupt handler.  This must be called from the RTOS trap
handler.
   To be fair, it must behave just like a processor interrupt, i.e. it
must save
   the full context of the interrupted thread during the preemption
```

```
processing. */
void  tm_interrupt_handler(void)
{

    /* Increment the interrupt count.  */
    tm_interrupt_handler_counter++;

    /* Put the semaphore from the interrupt handler.  */
    tm_semaphore_put(0);
}


/* Define the interrupt test reporting thread.  */
void  tm_interrupt_thread_report(void)
{

unsigned long   total;
unsigned long   last_total;
unsigned long   relative_time;
unsigned long   average;


    /* Initialize the last total.  */
    last_total =  0;

    /* Initialize the relative time.  */
    relative_time =  0;

    while(1)
    {

        /* Sleep to allow the test to run.  */
        tm_thread_sleep(TM_TEST_DURATION);

        /* Increment the relative time.  */
        relative_time =  relative_time + TM_TEST_DURATION;

        /* Print results to the stdio window.  */
        printf("**** Thread-Metric Interrupt Processing Test ****
Relative Time: %lu\n", relative_time);

        /* Calculate the total of all the counters.  */
        total =  tm_interrupt_thread_0_counter +
tm_interrupt_handler_counter;

        /* Calculate the average of all the counters.  */
        average =  total/2;

        /* See if there are any errors.  */
        if ((tm_interrupt_thread_0_counter < (average - 1)) ||
            (tm_interrupt_thread_0_counter > (average + 1)) ||
            (tm_interrupt_handler_counter < (average - 1)) ||
            (tm_interrupt_handler_counter > (average + 1)))
        {

            printf("ERROR: Invalid counter value(s). Interrupt processing
test has failed!\n");
```

```
        }

        /* Show the total interrupts for the time period.  */
        printf("Time Period Total:  %lu\n\n",
tm_interrupt_handler_counter - last_total);

        /* Save the last total number of interrupts.  */
        last_total =  tm_interrupt_handler_counter;
    }
}
```

1.6 tm_interrupt_preemption_processing_test.c

```
/**************************************************************************
**/
/*
*/
/*            Copyright (c) 1996-2007 by Express Logic Inc.
*/
/*
*/
/*  This Original Work may be modified, distributed, or otherwise used in
*/
/*  any manner with no obligations other than the following:
*/
/*
*/
/*    1. This legend must be retained in its entirety in any source code
*/
/*       copies of this Work.
*/
/*
*/
/*    2. This software may not be used in the development of an operating
*/
/*     system product.
*/
/*
*/
/*  This Original Work is hereby provided on an "AS IS" BASIS and WITHOUT
*/
/*  WARRANTY, either express or implied, including, without limitation,
*/
/*  the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A
*/
/*  PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF this
*/
/*  ORIGINAL WORK IS WITH the user.
*/
/*
*/
/*  Express Logic, Inc. reserves the right to modify this software
*/
/*  without notice.
```

```
*/
/*
*/
/*  Express Logic, Inc.                     info@expresslogic.com
*/
/*  11423 West Bernardo Court               http://www.expresslogic.com
*/
/*  San Diego, CA  92127
*/
/*
*/
/**************************************************************************
**/

/**************************************************************************
**/
/**************************************************************************
**/
/**
*/
/** Thread-Metric Component
*/
/**
*/
/**   Interrupt Preemption Processing Test
*/
/**
*/
/**************************************************************************
**/
/**************************************************************************
**/

#include "tm_api.h"


/* Define the counters used in the demo application...  */

unsigned long   tm_interrupt_preemption_thread_0_counter;
unsigned long   tm_interrupt_preemption_thread_1_counter;
unsigned long   tm_interrupt_preemption_handler_counter;


/* Define the test thread prototypes.  */

void            tm_interrupt_preemption_thread_0_entry(void);
void            tm_interrupt_preemption_thread_1_entry(void);
void            tm_interrupt_preemption_handler_entry(void);


/* Define the reporting thread prototype.  */

void            tm_interrupt_preemption_thread_report(void);


/* Define the interrupt handler.  This must be called from the RTOS.  */
```

```
void            tm_interrupt_preemption_handler(void);


/* Define the initialization prototype.  */

void            tm_interrupt_preemption_processing_initialize(void);


/* Define main entry point.  */

void main()
{

    /* Initialize the test.  */
    tm_initialize(tm_interrupt_preemption_processing_initialize);
}


/* Define the interrupt processing test initialization.  */

void  tm_interrupt_preemption_processing_initialize(void)
{

    /* Create interrupt thread at priority 3.  */
    tm_thread_create(0, 3, tm_interrupt_preemption_thread_0_entry);

    /* Create thread that generates the interrupt at priority 10.  */
    tm_thread_create(1, 10, tm_interrupt_preemption_thread_1_entry);

    /* Resume just thread 1.  */
    tm_thread_resume(1);

    /* Create the reporting thread. It will preempt the other
       threads and print out the test results.  */
    tm_thread_create(5, 2, tm_interrupt_preemption_thread_report);
    tm_thread_resume(5);
}


/* Define the interrupt thread.  This thread is resumed from the
   interrupt handler.  It runs and suspends.  */
void  tm_interrupt_preemption_thread_0_entry(void)
{

    while(1)
    {

        /* Increment this thread's counter.  */
        tm_interrupt_preemption_thread_0_counter++;

        /* Suspend. This will allow the thread generating the
           interrupt to run again.  */
        tm_thread_suspend(0);
    }
}

/* Define the thread that generates the interrupt.  */
```

```
void  tm_interrupt_preemption_thread_1_entry(void)
{

    while(1)
    {

        /* Force an interrupt. The underlying RTOS must see that the
           the interrupt handler is called from the appropriate software
           interrupt or trap. */

        asm("trap");  /* This is PowerPC specific.  */

        /* We won't get back here until the interrupt processing is
complete,
           including the execution of the higher priority thread made
ready
           by the interrupt.  */

        /* Increment this thread's counter.  */
        tm_interrupt_preemption_thread_1_counter++;
    }
}


/* Define the interrupt handler.  This must be called from the RTOS trap
handler.
   To be fair, it must behave just like a processor interrupt, i.e. it
must save
   the full context of the interrupted thread during the preemption
processing. */
void  tm_interrupt_preemption_handler(void)
{

    /* Increment the interrupt count.  */
    tm_interrupt_preemption_handler_counter++;

    /* Resume the higher priority thread from the ISR.  */
    tm_thread_resume(0);
}


/* Define the interrupt test reporting thread.  */
void  tm_interrupt_preemption_thread_report(void)
{

unsigned long   total;
unsigned long   relative_time;
unsigned long   last_total;
unsigned long   average;


    /* Initialize the last total.  */
    last_total =  0;

    /* Initialize the relative time.  */
    relative_time =  0;
```

```
    while(1)
    {

        /* Sleep to allow the test to run.  */
        tm_thread_sleep(TM_TEST_DURATION);

        /* Increment the relative time.  */
        relative_time =  relative_time + TM_TEST_DURATION;

        /* Print results to the stdio window.  */
        printf("**** Thread-Metric Interrupt Preemption Processing Test
**** Relative Time: %lu\n", relative_time);

        /* Calculate the total of all the counters.  */
        total =  tm_interrupt_preemption_thread_0_counter +
tm_interrupt_preemption_thread_1_counter +
tm_interrupt_preemption_handler_counter;

        /* Calculate the average of all the counters.  */
        average =  total/3;

        /* See if there are any errors.  */
        if ((tm_interrupt_preemption_thread_0_counter < (average - 1)) ||
            (tm_interrupt_preemption_thread_0_counter > (average + 1)) ||
            (tm_interrupt_preemption_thread_1_counter < (average - 1)) ||
            (tm_interrupt_preemption_thread_1_counter > (average + 1)) ||
            (tm_interrupt_preemption_handler_counter < (average - 1)) ||
            (tm_interrupt_preemption_handler_counter > (average + 1)))
        {

            printf("ERROR: Invalid counter value(s). Interrupt processing
test has failed!\n");
        }

        /* Show the total interrupts for the time period.  */
        printf("Time Period Total:  %lu\n\n\n",
tm_interrupt_preemption_handler_counter - last_total);

        /* Save the last total number of interrupts.  */
        last_total =  tm_interrupt_preemption_handler_counter;
    }
}
```

1.7 tm_memory_allocation_test.c

```
/**************************************************************************
**/
/*
*/
/*            Copyright (c) 1996-2007 by Express Logic Inc.
*/
/*
*/
/*  This Original Work may be modified, distributed, or otherwise used in
```

```
*/
/*  any manner with no obligations other than the following:
*/
/*
*/
/*    1. This legend must be retained in its entirety in any source code
*/
/*       copies of this Work.
*/
/*
*/
/*    2. This software may not be used in the development of an operating
*/
/*     system product.
*/
/*
*/
/*  This Original Work is hereby provided on an "AS IS" BASIS and WITHOUT
*/
/*  WARRANTY, either express or implied, including, without limitation,
*/
/*  the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A
*/
/*  PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF this
*/
/*  ORIGINAL WORK IS WITH the user.
*/
/*
*/
/*  Express Logic, Inc. reserves the right to modify this software
*/
/*  without notice.
*/
/*
*/
/*  Express Logic, Inc.                     info@expresslogic.com
*/
/*  11423 West Bernardo Court               http://www.expresslogic.com
*/
/*  San Diego, CA  92127
*/
/*
*/
/**************************************************************************
**/

/**************************************************************************
**/
/**************************************************************************
**/
/**
*/
/** Thread-Metric Component
*/
/**
*/
/**   Memory Allocation Test
```

```
*/
/**
*/
/*****************************************************************************
**/
/*****************************************************************************
**/

#include "tm_api.h"


/* Define the counters used in the demo application...  */

unsigned long   tm_memory_allocation_counter;


/* Define the test thread prototypes.  */

void            tm_memory_allocation_thread_0_entry(void);


/* Define the reporting thread prototype.  */

void            tm_memory_allocation_thread_report(void);


/* Define the initialization prototype.  */

void            tm_memory_allocation_initialize(void);


/* Define main entry point.  */

void main()
{

    /* Initialize the test.  */
    tm_initialize(tm_memory_allocation_initialize);
}


/* Define the memory allocation processing test initialization.  */

void  tm_memory_allocation_initialize(void)
{

    /* Create thread 0 at priority 10.  */
    tm_thread_create(0, 10, tm_memory_allocation_thread_0_entry);

    /* Resume thread 0.  */
    tm_thread_resume(0);

    /* Create a memory pool.  */
    tm_memory_pool_create(0);

    /* Create the reporting thread. It will preempt the other
       threads and print out the test results.  */
```

```
    tm_thread_create(5, 2, tm_memory_allocation_thread_report);
    tm_thread_resume(5);
}


/* Define the memory allocation processing thread.  */
void  tm_memory_allocation_thread_0_entry(void)
{

int             status;
unsigned char   *memory_ptr;


    while(1)
    {

        /* Allocate memory from pool.  */
        tm_memory_pool_allocate(0, &memory_ptr);

        /* Release the memory back to the pool.  */
        status =  tm_memory_pool_deallocate(0, memory_ptr);

        /* Check for invalid memory allocation/deallocation.  */
        if (status != TM_SUCCESS)
            break;

        /* Increment the number of memory allocations sent and received.
*/
        tm_memory_allocation_counter++;
    }
}


/* Define the memory allocation test reporting thread.  */
void  tm_memory_allocation_thread_report(void)
{

unsigned long   last_counter;
unsigned long   relative_time;


    /* Initialize the last counter.  */
    last_counter =  0;

    /* Initialize the relative time.  */
    relative_time =  0;

    while(1)
    {

        /* Sleep to allow the test to run.  */
        tm_thread_sleep(TM_TEST_DURATION);

        /* Increment the relative time.  */
        relative_time =  relative_time + TM_TEST_DURATION;

        /* Print results to the stdio window.  */
```

```
        printf("**** Thread-Metric Memory Allocation Test **** Relative
Time: %lu\n", relative_time);

        /* See if there are any errors.  */
        if (tm_memory_allocation_counter == last_counter)
        {

            printf("ERROR: Invalid counter value(s). Error
allocating/deallocating memory!\n");
        }

        /* Show the time period total.  */
        printf("Time Period Total:  %lu\n\n\n",
tm_memory_allocation_counter - last_counter);

        /* Save the last counter.  */
        last_counter =  tm_memory_allocation_counter;
    }
}
```

1.8 tm_message_processing_test.c

```
/**************************************************************************
**/
/*
*/
/*            Copyright (c) 1996-2007 by Express Logic Inc.
*/
/*
*/
/*  This Original Work may be modified, distributed, or otherwise used in
*/
/*  any manner with no obligations other than the following:
*/
/*
*/
/*    1. This legend must be retained in its entirety in any source code
*/
/*       copies of this Work.
*/
/*
*/
/*    2. This software may not be used in the development of an operating
*/
/*     system product.
*/
/*
*/
/*  This Original Work is hereby provided on an "AS IS" BASIS and WITHOUT
*/
/*  WARRANTY, either express or implied, including, without limitation,
*/
/*  the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A
*/
/*  PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF this
```

```
*/
/*  ORIGINAL WORK IS WITH the user.
*/
/*
*/
/*  Express Logic, Inc. reserves the right to modify this software
*/
/*  without notice.
*/
/*
*/
/*  Express Logic, Inc.                     info@expresslogic.com
*/
/*  11423 West Bernardo Court               http://www.expresslogic.com
*/
/*  San Diego, CA  92127
*/
/*
*/
/**************************************************************************
**/

/**************************************************************************
**/
/**************************************************************************
**/
/**
*/
/** Thread-Metric Component
*/
/**
*/
/**   Message Processing Test
*/
/**
*/
/**************************************************************************
**/
/**************************************************************************
**/

#include "tm_api.h"


/* Define the counters used in the demo application...  */

unsigned long   tm_message_processing_counter;
unsigned long   tm_message_sent[4];
unsigned long   tm_message_received[4];


/* Define the test thread prototypes.  */

void            tm_message_processing_thread_0_entry(void);


/* Define the reporting thread prototype.  */
```

```
void            tm_message_processing_thread_report(void);


/* Define the initialization prototype.  */

void            tm_message_processing_initialize(void);


/* Define main entry point.  */

void main()
{

    /* Initialize the test.  */
    tm_initialize(tm_message_processing_initialize);
}


/* Define the message processing test initialization.  */

void  tm_message_processing_initialize(void)
{

    /* Create thread 0 at priority 10.  */
    tm_thread_create(0, 10, tm_message_processing_thread_0_entry);

    /* Resume thread 0.  */
    tm_thread_resume(0);

    /* Create a queue for the message passing.  */
    tm_queue_create(0);

    /* Create the reporting thread. It will preempt the other
       threads and print out the test results.  */
    tm_thread_create(5, 2, tm_message_processing_thread_report);
    tm_thread_resume(5);
}


/* Define the message processing thread.  */
void  tm_message_processing_thread_0_entry(void)
{

    /* Initialize the source message.   */
    tm_message_sent[0] =  0x11112222;
    tm_message_sent[1] =  0x33334444;
    tm_message_sent[2] =  0x55556666;
    tm_message_sent[3] =  0x77778888;

    while(1)
    {

        /* Send a message to the queue.  */
        tm_queue_send(0, tm_message_sent);

        /* Receive a message from the queue.  */
```

```
        tm_queue_receive(0, tm_message_received);

        /* Check for invalid message.  */
        if (tm_message_received[3] != tm_message_sent[3])
            break;

        /* Increment the last word of the 16-byte message.  */
        tm_message_sent[3]++;

        /* Increment the number of messages sent and received.  */
        tm_message_processing_counter++;
    }
}


/* Define the message test reporting thread.  */
void  tm_message_processing_thread_report(void)
{

unsigned long   last_counter;
unsigned long   relative_time;


    /* Initialize the last counter.  */
    last_counter =  0;

    /* Initialize the relative time.  */
    relative_time =  0;

    while(1)
    {

        /* Sleep to allow the test to run.  */
        tm_thread_sleep(TM_TEST_DURATION);

        /* Increment the relative time.  */
        relative_time =  relative_time + TM_TEST_DURATION;

        /* Print results to the stdio window.  */
        printf("**** Thread-Metric Message Processing Test **** Relative
Time: %lu\n", relative_time);

        /* See if there are any errors.  */
        if (tm_message_processing_counter == last_counter)
        {

            printf("ERROR: Invalid counter value(s). Error
sending/receiving messages!\n");
        }

        /* Show the time period total.  */
        printf("Time Period Total:  %lu\n\n",
tm_message_processing_counter - last_counter);

        /* Save the last counter.  */
        last_counter =  tm_message_processing_counter;
    }
```

```
}
```

1.9 tm_porting_layer.c

```
/**************************************************************************
**/
/*
*/
/*            Copyright (c) 1996-2007 by Express Logic Inc.
*/
/*
*/
/*  This Original Work may be modified, distributed, or otherwise used in
*/
/*  any manner with no obligations other than the following:
*/
/*
*/
/*    1. This legend must be retained in its entirety in any source code
*/
/*       copies of this Work.
*/
/*
*/
/*    2. This software may not be used in the development of an operating
*/
/*     system product.
*/
/*
*/
/*  This Original Work is hereby provided on an "AS IS" BASIS and WITHOUT
*/
/*  WARRANTY, either express or implied, including, without limitation,
*/
/*  the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A
*/
/*  PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF this
*/
/*  ORIGINAL WORK IS WITH the user.
*/
/*
*/
/*  Express Logic, Inc. reserves the right to modify this software
*/
/*  without notice.
*/
/*
*/
/*  Express Logic, Inc.                     info@expresslogic.com
*/
/*  11423 West Bernardo Court               http://www.expresslogic.com
*/
/*  San Diego, CA  92127
*/
/*
```

```
*/
/****************************************************************************
**/

/****************************************************************************
**/
/****************************************************************************
**/
/**
*/
/** Thread-Metric Component
*/
/**
*/
/**   Porting Layer (Must be completed with RTOS specifics)
*/
/**
*/
/****************************************************************************
**/
/****************************************************************************
**/


/* Include necessary files.  */

#include    "tm_api.h"


/* This function called from main performs basic RTOS initialization,
   calls the test initialization function, and then starts the RTOS
function.  */
void  tm_initialize(void (*test_initialization_function)(void))
{

}


/* This function takes a thread ID and priority and attempts to create
the
   file in the underlying RTOS.  Valid priorities range from 1 through
31,
   where 1 is the highest priority and 31 is the lowest. If successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be
returned.   */
int  tm_thread_create(int thread_id, int priority, void
(*entry_function)(void))
{

}


/* This function resumes the specified thread.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_thread_resume(int thread_id)
{
```

```
}


/* This function suspends the specified thread.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_thread_suspend(int thread_id)
{

}


/* This function relinquishes to other ready threads at the same
   priority.  */
void tm_thread_relinquish(void)
{

}


/* This function suspends the specified thread for the specified number
   of seconds.  If successful, the function should return TM_SUCCESS.
   Otherwise, TM_ERROR should be returned.  */
void tm_thread_sleep(int seconds)
{

}


/* This function creates the specified queue.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_queue_create(int queue_id)
{

}


/* This function sends a 16-byte message to the specified queue.  If
successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be
returned.  */
int  tm_queue_send(int queue_id, unsigned long *message_ptr)
{

}


/* This function receives a 16-byte message from the specified queue.  If
successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be
returned.  */
int  tm_queue_receive(int queue_id, unsigned long *message_ptr)
{

}
```

```
/* This function creates the specified semaphore.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_semaphore_create(int semaphore_id)
{

}


/* This function gets the specified semaphore.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_semaphore_get(int semaphore_id)
{

}


/* This function puts the specified semaphore.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_semaphore_put(int semaphore_id)
{

}


/* This function creates the specified memory pool that can support one
or more
   allocations of 128 bytes.  If successful, the function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_memory_pool_create(int pool_id)
{

}


/* This function allocates a 128 byte block from the specified memory
pool.
   If successful, the function should return TM_SUCCESS. Otherwise,
TM_ERROR
   should be returned.  */
int  tm_memory_pool_allocate(int pool_id, unsigned char **memory_ptr)
{

}


/* This function releases a previously allocated 128 byte block from the
specified
   memory pool. If successful, the function should return TM_SUCCESS.
Otherwise, TM_ERROR
   should be returned.  */
int  tm_memory_pool_deallocate(int pool_id, unsigned char *memory_ptr)
{
```

```
}
```

Anexo 2 – Código do arquivo tm_porting_layer.c adaptado para o BRTOS

```
/**************************************************************************
**/
/*
*/
/*            Copyright (c) 1996-2007 by Express Logic Inc.
*/
/*
*/
/*  This Original Work may be modified, distributed, or otherwise used in
*/
/*  any manner with no obligations other than the following:
*/
/*
*/
/*    1. This legend must be retained in its entirety in any source code
*/
/*       copies of this Work.
*/
/*
*/
/*    2. This software may not be used in the development of an operating
*/
/*   system product.                                                    */
/*
*/
/*  This Original Work is hereby provided on an "AS IS" BASIS and WITHOUT
*/
/*  WARRANTY, either express or implied, including, without limitation,
*/
/*  the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A
*/
/*  PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF this
*/
/*  ORIGINAL WORK IS WITH the user.
*/
/*
*/
/*  Express Logic, Inc. reserves the right to modify this software
*/
/*  without notice.
*/
/*
*/
/*  Express Logic, Inc.                     info@expresslogic.com
*/
/*  11423 West Bernardo Court               http://www.expresslogic.com
*/
/*  San Diego, CA  92127
*/
/*
```

```
*/
/*********************************************************************
**/

/*********************************************************************
**/
/*********************************************************************
**/
/**
*/
/** Thread-Metric Component
*/
/**
*/
/**   Porting Layer (Must be completed with RTOS specifics)
*/
/**
*/
/*********************************************************************
**/
/*********************************************************************
**/


/* Include necessary files.  */

#include    "tm_api.h"

/* Include BRTOS API header */
#include "BRTOS.h"
#include "hardware.h"


#define TM_BRTOS_THREAD_STACK_SIZE   300

BRTOS_Sem* Sema;
BRTOS_Sem* Sema_1;

// Declara uma estrutura de fila
OS_QUEUE_32   QBuffer;
BRTOS_Queue   *Q;

INT8U Decrementa( int i)
{
  INT16U CPU_SR = 0;

  OSEnterCritical();
  if (i > 0)
    i--;
  OSExitCritical();
  return OK;
}

INT8U Incrementa( int i)
{
  INT16U CPU_SR = 0;
```

```
  OSEnterCritical();
  if (i < 255)
    i++;
  OSExitCritical();
  return OK;
}

/* ------------------------------------------------------------------------
---------------------------------------- */

/* This function called from main performs basic RTOS initialization,
   calls the test initialization function, and then starts the RTOS
function.  */
void  tm_initialize(void (*test_initialization_function)(void))
{

  ////////////////////////////////////////////////////////
  /////      Initialize Events                       /////
  ////////////////////////////////////////////////////////

  BRTOS_Init();

  // for printf use
  init_SCI2();

  test_initialization_function();

  // Start Task Scheduler
  if(BRTOSStart() != OK)
  {
    while(1){};
  };
}

/* ------------------------------------------------------------------------
---------------------------------------- */

/* This function takes a thread ID and priority and attempts to create
the
   file in the underlying RTOS.  Valid priorities range from 1 through
31,
   where 1 is the highest priority and 31 is the lowest. If successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be
returned.   */
int  tm_thread_create(int thread_id, int priority, void
(*entry_function)(void))
{
    // BRTOS has inverted priority order
    //priority = 31 - priority;

    (void)thread_id; // stops compiler warnings

if(InstallTask(entry_function,NULL,TM_BRTOS_THREAD_STACK_SIZE,(INT8U)prio
rity)){
        return TM_ERROR;
    }
```

```
    if(BlockPriority((INT8U)priority)){
        return TM_ERROR;
    }
    return TM_SUCCESS;
}

/* ------------------------------------------------------------------------
---------------------------------------- */

/* This function resumes the specified thread.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_thread_resume(int thread_id)
{
    if(UnBlockTask((INT8U)thread_id)){
        return TM_ERROR;
    }
    return TM_SUCCESS;
}

/* ------------------------------------------------------------------------
---------------------------------------- */

/* This function suspends the specified thread.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_thread_suspend(int thread_id)
{
    if(BlockTask((INT8U)thread_id)){
        return TM_ERROR;
    }
    return TM_SUCCESS;
}


/* This function relinquishes to other ready threads at the same
   priority.  */
void tm_thread_relinquish(void)
{

}


/* This function suspends the specified thread for the specified number
   of seconds.  If successful, the function should return TM_SUCCESS.
   Otherwise, TM_ERROR should be returned.  */
void tm_thread_sleep(int seconds)
{
    (void)DelayTask((INT16U)(seconds*100));
}


/* This function creates the specified queue.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_queue_create(int queue_id)
{
```

```
    if (OSQueue32Create(&QBuffer,32, (BRTOS_Queue**)queue_id)) {
        return TM_ERROR;
    }
    return TM_SUCCESS;
}


/* This function sends a 16-byte message to the specified queue.  If
successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be
returned.  */
int  tm_queue_send(int queue_buffer, unsigned long *message_ptr)
{

    while(*message_ptr){
      if(OSWQueue32((OS_QUEUE_32 *)queue_buffer,(INT32U) *message_ptr)){
          return TM_ERROR;
      }
      message_ptr++;
    }
    return TM_SUCCESS;
}


/* This function receives a 16-byte message from the specified queue.  If
successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be
returned.  */
int  tm_queue_receive(int queue_buffer, unsigned long *message_ptr)
{
    while(((OS_QUEUE_32 *)queue_buffer)->OSQEntries)  {

      if(OSRQueue32((OS_QUEUE_32 *)queue_buffer,(INT32U*) message_ptr)){
          return TM_ERROR;
      }
      message_ptr++;
    }
    return TM_SUCCESS;
}


/* This function creates the specified semaphore.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_semaphore_create(int semaphore_id)
{
     if(OSSemCreate (1, (BRTOS_Sem**)semaphore_id)){
      return TM_ERROR ;
     }
     return TM_SUCCESS;
}


/* This function gets the specified semaphore.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
```

```
int  tm_semaphore_get(int semaphore_id)
{
     if(OSSemPend ((BRTOS_Sem*)semaphore_id, 0)){
     //if(Decrementa (semaphore_id)){
         return TM_ERROR ;
     }
     return TM_SUCCESS;
}


/* This function puts the specified semaphore.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_semaphore_put(int semaphore_id)
{
      if(OSSemPost ((BRTOS_Sem*)semaphore_id)){
      //if(Incrementa (semaphore_id)){
         return TM_ERROR ;
     }
     return TM_SUCCESS;
}


/* This function creates the specified memory pool that can support one
or more
   allocations of 128 bytes.  If successful, the function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_memory_pool_create(int pool_id)
{
  (void)pool_id;
}


/* This function allocates a 128 byte block from the specified memory
pool.
   If successful, the function should return TM_SUCCESS. Otherwise,
TM_ERROR
   should be returned.  */
int  tm_memory_pool_allocate(int pool_id, unsigned char **memory_ptr)
{
    (void)pool_id;
    (void)memory_ptr;
}


/* This function releases a previously allocated 128 byte block from the
specified
   memory pool. If successful, the function should return TM_SUCCESS.
Otherwise, TM_ERROR
   should be returned.  */
int  tm_memory_pool_deallocate(int pool_id, unsigned char *memory_ptr)
{
    (void)pool_id;
    (void)memory_ptr;
    return TM_SUCCESS;
}
```

Anexo 3 – Código do arquivo tm_porting_layer.c adaptado para o *FreeRTOS*

```
/**************************************************************************
**/
/*
*/
/*            Copyright (c) 1996-2007 by Express Logic Inc.
*/
/*
*/
/*  This Original Work may be modified, distributed, or otherwise used in
*/
/*  any manner with no obligations other than the following:
*/
/*
*/
/*    1. This legend must be retained in its entirety in any source code
*/
/*       copies of this Work.
*/
/*
*/
/*    2. This software may not be used in the development of an operating
*/
/*   system product.                                                    */
/*
*/
/*  This Original Work is hereby provided on an "AS IS" BASIS and WITHOUT
*/
/*  WARRANTY, either express or implied, including, without limitation,
*/
/*  the warranties of NON-INFRINGEMENT, MERCHANTABILITY or FITNESS FOR A
*/
/*  PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY OF this
*/
/*  ORIGINAL WORK IS WITH the user.
*/
/*
*/
/*  Express Logic, Inc. reserves the right to modify this software
*/
/*  without notice.
*/
/*
*/
/*  Express Logic, Inc.                     info@expresslogic.com
*/
/*  11423 West Bernardo Court               http://www.expresslogic.com
*/
/*  San Diego, CA  92127
*/
/*
*/
/**************************************************************************
**/

/**************************************************************************
```

```
**/
/****************************************************************************
**/
/**
*/
/** Thread-Metric Component
*/
/**
*/
/**   Porting Layer (Must be completed with RTOS specifics)
*/
/**
*/
/****************************************************************************
**/
/****************************************************************************
**/


/* Include necessary files.  */

#include "tm_api.h"

/* Include FreeRTOS API header */
/* Scheduler includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "semphr.h"


#define TM_FREERTOS_THREAD_STACK_SIZE   ( ( unsigned short ) 100 )


/* ------------------------------------------------------------------------
------------------------------------------ */

/* This function called from main performs basic RTOS initialization,
   calls the test initialization function, and then starts the RTOS
function.  */
void  tm_initialize(void (*test_initialization_function)(void))
{

  init_SCI();

  test_initialization_function();

  /* Start the scheduler. */
      vTaskStartScheduler();
}

/* ------------------------------------------------------------------------
------------------------------------------ */

/* This function takes a thread ID and priority and attempts to create
the
   file in the underlying RTOS.  Valid priorities range from 1 through
```

```
31,
   where 1 is the highest priority and 31 is the lowest. If successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be
returned.   */
int  tm_thread_create(int thread_id, int priority, void
(*entry_function)(void))
{
     xTaskCreate((void*)entry_function, NULL,
TM_FREERTOS_THREAD_STACK_SIZE, NULL, priority,( xTaskHandle*) thread_id);

     //vTaskSuspend((xTaskHandle)(*( xTaskHandle*)thread_id));
     return TM_SUCCESS;
}

/* ------------------------------------------------------------------------
---------------------------------------- */

/* This function resumes the specified thread.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_thread_resume(int thread_id)
{
    vTaskResume((xTaskHandle)thread_id);
    return TM_SUCCESS;
}

/* ------------------------------------------------------------------------
---------------------------------------- */

/* This function suspends the specified thread.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_thread_suspend(int thread_id)
{
    vTaskSuspend(( xTaskHandle)thread_id);
    return TM_SUCCESS;
}


/* This function relinquishes to other ready threads at the same
   priority.  */
void tm_thread_relinquish(void)
{

}


/* This function suspends the specified thread for the specified number
   of seconds.  If successful, the function should return TM_SUCCESS.
   Otherwise, TM_ERROR should be returned.  */
void tm_thread_sleep(int seconds)
{
    //DelayTaskHMSM((INT8U) 0, (INT8U) 0, (INT8U) seconds, (INT16U) 0);
    (void)vTaskDelay((portTickType)(seconds*100));
}
```

```
/* This function creates the specified queue.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_queue_create(int queue_id)
{


    return TM_SUCCESS;
}


/* This function sends a 16-byte message to the specified queue.  If
successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be
returned.  */
int  tm_queue_send(int queue_buffer, unsigned long *message_ptr)
{
    if (xQueueSend( (xQueueHandle)queue_buffer, message_ptr,
portMAX_DELAY )!= pdPASS)
    {
      return TM_ERROR;
    }

    return TM_SUCCESS;
}


/* This function receives a 16-byte message from the specified queue.  If
successful,
   the function should return TM_SUCCESS. Otherwise, TM_ERROR should be
returned.  */
int  tm_queue_receive(int queue_buffer, unsigned long *message_ptr)
{
    if (xQueueReceive( (xQueueHandle)queue_buffer, message_ptr,
portMAX_DELAY ) != pdPASS)
    {
      return TM_ERROR;
    }

    return TM_SUCCESS;
}


/* This function creates the specified semaphore.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_semaphore_create(int semaphore_id)
{


     return TM_SUCCESS;
}

/* This function gets the specified semaphore.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
```

```
int  tm_semaphore_get(int semaphore_id)
{
     xSemaphoreTake((xSemaphoreHandle) semaphore_id, 0);
     return TM_SUCCESS;
}


/* This function puts the specified semaphore.  If successful, the
function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_semaphore_put(int semaphore_id)
{
     xSemaphoreGive((xSemaphoreHandle) semaphore_id);
     return TM_SUCCESS;
}


/* This function creates the specified memory pool that can support one
or more
   allocations of 128 bytes.  If successful, the function should
   return TM_SUCCESS. Otherwise, TM_ERROR should be returned.  */
int  tm_memory_pool_create(int pool_id)
{
  (void)pool_id;
}


/* This function allocates a 128 byte block from the specified memory
pool.
   If successful, the function should return TM_SUCCESS. Otherwise,
TM_ERROR
   should be returned.  */
int  tm_memory_pool_allocate(int pool_id, unsigned char **memory_ptr)
{
    (void)pool_id;
    (void)memory_ptr;
}


/* This function releases a previously allocated 128 byte block from the
specified
   memory pool. If successful, the function should return TM_SUCCESS.
Otherwise, TM_ERROR
   should be returned.  */
int  tm_memory_pool_deallocate(int pool_id, unsigned char *memory_ptr)
{
    (void)pool_id;
    (void)memory_ptr;
    return TM_SUCCESS;
}
```