UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
CÂMPUS PATO BRANCO
CURSO DE ENGENHARIA DE COMPUTAÇÃO

ROGERIO CORREA MEDEIROS

INTEGRAÇAO DE UMA REDE DE SENSORES SEM FIO COM UM RTOS TICKLESS

TRABALHO DE CONCLUSÃO DE CURSO

PATO BRANCO
2014

ROGÉRIO CORRÊA MEDEIROS

INTEGRAÇÃO DE UMA REDE DE SENSORES SEM FIO COM UM RTOS TICKLESS

Trabalho de Conclusão de Curso de graduação, apresentado à disciplina de TCC2, do Curso de Engenharia de Computação, da Universidade Tecnológica Federal do Paraná, Câmpus Pato Branco, como requisito parcial para obtenção do título de bacharel.

Orientador: Prof. Dr. Gustavo W. Denardin

PATO BRANCO
2014

MINISTÉRIO DA EDUCAÇÃO
Universidade Tecnológica Federal do Paraná
Câmpus Pato Branco
Departamento Acadêmico de Informática
Curso de Engenharia de Computação

TERMO DE APROVAÇÃO

Às 10h30 do dia 19 de agosto de 2014, na sala V007, UTFPR, Câmpus Pato Branco, reuniu-se a banca examinadora composta pelos professores Gustavo Weber Denardin (orientador), Fernando José Avancini Schenatto e Kathya Silvia Collazos Linares para avaliar o trabalho de conclusão de curso com o título **Integração de uma rede de sensores sem fio com um RTOS tickless**, do aluno **Rogério Corrêa Medeiros**, matrícula 1065190, do curso de Engenharia de Computação. Após a apresentação o candidato foi arguido pela banca examinadora. Em seguida foi realizada a deliberação pela banca examinadora que considerou o trabalho aprovado.

Gustavo Weber Denardin
Orientador (UTFPR)

Fernando José Avancini Schenatto
(UTFPR)

Kathya Silvia Collazos Linares
(UTFPR)

Beatriz Terezinha Borsoi
Coordenador de TCC

Marco Antonio de Castro Barbosa
Coordenador do Curso de
Engenharia de Computação

AGRADECIMENTOS

Aos meus pais Gilberto e Neila, que são a minha base e sempre acreditaram em mim, sem o apoio deles seria muito mais difícil vencer esse desafio.

Aos meus avós e demais familiares pela confiança e apoio.

Ao professor Gustavo W. Denardin pela orientação e confiança depositada.

À Everaldo Mazieira e Joelma Busato, pela acolhida, amizade e pelos ótimos momentos.

À Klaem e Catia Silva, pela acolhida no início da graduação.

Aos colegas Caciano Mattiello, Eduardo dos Santos, Igor Hoelscher, Nayara Ferreira, pela amizade e companheirismo.

À Joelma Pereira pelo suporte durante a etapa final da graduação.

Aos professores do DAINF e DAELE, gostaria de externar a minha satisfação pela oportunidade de crescer e aprender com vocês.

Julgue seu sucesso pelas coisas que você teve que renunciar para conseguir.

Dalai Lama.

RESUMO

Este trabalho tem como objetivo avaliar a redução do consumo de energia dos elementos que compõem redes de sensores baseadas em sistemas operacionais de tempo real, através da implementação de suporte ao modo de operação *tickless* em um sistema operacional de tempo real e utilização de um protocolo de acesso ao meio com controle de razão cíclica. O uso de um sistema operacional de tempo real é fundamental para o gerenciamento da concorrência e restrições temporais inerentes ao *software* de uma Rede de Sensores sem Fio (RSSF), devido a possuir os recursos necessários para uma melhor implementação da rede, como gerenciamento de tempo, semáforos, sincronização de tarefas, entre outros. No entanto, a utilização de um sistema operacional de tempo real apresenta um custo computacional, bem como um inerente aumento de consumo de energia devido ao seu modo de operação. Assim, dentre as técnicas existentes na literatura para a redução do consumo de energia de um sistema de tempo real, optou-se pela implementação de suporte ao modo de operação *tickless* do temporizador do sistema operacional de tempo real, bem como pela utilização de modos de baixo consumo de energia de processadores da arquitetura ARM. A outra fonte principal de consumo de energia em RSSF, o rádio, também teve seu consumo reduzido no dispositivo roteador pela utilização de um protocolo de acesso ao meio com controle de razão cíclica. Para análise dos resultados foi utilizada a estrutura mínima para representar uma RSSF, um nó coordenador e um nó roteador foram configurados de forma a manter uma comunicação utilizando a topologia em estrela e a redução no consumo médio de energia do sistema foi de 54,74%.

Palavras-chave: Sistema Operacional de Tempo Real. Redes de Sensores. Tickless.

ABSTRACT

This work aims to reduce the energy consumption of the elements of a sensor networks based on real-time operating systems (RTOS), by implementation a tickless timer for the RTOS and using a media access protocol with duty cycle control. The use of a RTOS is fundamental to manage the existing concurrence and time constraints in a WSN software. Such operating systems usually have a better implementation of network resources, such as time management, semaphores, task synchronization, among others. However, the use of an RTOS has a computational cost, as well as the inherent increase in power consumption due to its operating modes. Thus, among the existing techniques in the literature for the reduction of energy consumption of a real-time system, it was chosen to implement a tickless timer for the real-time operating system time management, as well as the use of a low power consumption mode available in the ARM processors architecture. Another major source of energy consumption in WSN, the radio, also had its consumption reduced in the router device by using a media access protocol with duty cycle control. For data analysis, a minimum structure was used to represent a WSN, i.e., a coordinator and a router were configured to maintain communication using a star topology and the obtained reduction in average power consumption of the system was 54.74%.

Keywords: Real-Time Operating System. Sensor Networks. Tickless.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

ACK: Acknowledgement.

BRTOS: Brazilian Real-Time Operating System.

CPU: Central Processing Unit.

FFD: Full Function Device.

HAL: Hardware Abstract Layer.

MAC: Media Access Control.

MMU: Memory Management Unit.

MPU: Memory Protection Unit.

RFD: Reduced Function Device.

RISC: Reduced Instruction Set Computer.

RSSF: Rede de Sensores Sem Fio.

RSASF: Rede de Sensores e Atuadores Sem Fio.

RTOS: Real-Time Operating System.

SO: Sistema Operacional.

WPAN – Wireless Personal Area Network.

SUMÁRIO

1 Introdução

Devido às recentes inovações na área da comunicação sem fio, sensores multifuncionais e da eletrônica digital, as Redes de Sensores sem Fio (RSSF) constituem uma tecnologia cada vez mais utilizada e que tem proporcionado avanços significativos no monitoramento, processamento e coordenação em diferentes contextos.

A combinação de sensores e atuadores em uma mesma rede constitui as chamadas Redes de Sensores e Atuadores sem Fio (RSASF) que, segundo Araújo, Villas e Boukerche (2007), "são capazes de tomar decisões e executar ações no ambiente em que estão inseridas, alavancando ainda mais o potencial das RSSF no monitoramento preciso de diferentes ambientes físicos que exigem resposta imediata".

Por outro lado, um dos requisitos mais importante para o projeto de uma RSSF, e que é objeto de vários estudos atualmente é o consumo de energia dos elementos da rede.

O consumo de energia das RSSF deve ser considerado, visto que, em certos cenários, cada nó de comunicação possui energia limitada e deve utilizá-la de forma eficaz para manter a conectividade da rede por um maior período de tempo (WILL, SCHELEISER e SCHILLER, 2009).

Uma forma de reduzir o consumo de energia em uma RSSF é manter seus rádios transmissores desligados durante o maior período possível, porém esses rádios devem acordar com frequência suficiente para não permitir que informação seja perdida. Isso está diretamente relacionado à frequência com que ocorrem os eventos na área de atuação de uma rede, visto que, quando não há informação a ser transmitida ou recebida, o rádio transmissor deve permanecer desligado.

Devido à complexidade de adaptação dos protocolos de comunicação existentes ocasionada pelas restrições de um sistema embarcado destinado a uma RSSF (pouca memória, pouca capacidade de processamento, etc.), associadas à concorrência entre as tarefas do sistema, recomenda-se o uso de um sistema operacional (SO) para a implementação do *software* dos dispositivos que compõem estas redes. Além disso, existem tarefas em RSSF que são dirigidas por temporização (o controle de ativação do rádio, por exemplo). Como existe mais de uma tarefa controlada por tempo, normalmente se utiliza um temporizador de *software*, que geralmente é controlado pelo *tick* do sistema (FAROOQ e KUNZ, 2011).

Originalmente foram desenvolvidos uma série de SOs para RSSF, entre os mais famosos estão o TinyOS, o Contiki e o Mantis. No entanto estes SOs não são sistemas de tempo real. Com o crescimento do conceito de redes de sensores, certas aplicações

começaram a exigir respostas com restrições de tempo real. Um sistema operacional de tempo real garante o cumprimento desse requisito, porém esse tipo de sistema não é orientado a baixo consumo de energia, principalmente pelo falo de que para controlar o tempo tais sistemas utilizam o *tick*, que gera um consumo excessivo de energia.

1.1 Objetivo Geral

Avaliar a redução do consumo de energia dos elementos que compõem redes de sensores baseadas em sistemas operacionais de tempo real.

1.2 Objetivos Específicos

1.2.1 Avaliar técnicas de modo de baixo consumo de energia que mais se adequem ao contexto de RSSF.

1.2.2 Implementar suporte ao *tickless* em um sistema operacional de tempo real para processadores da arquitetura ARM, a fim de reduzir o consumo de energia por parte dos processadores;

1.2.3 Utilizar um protocolo de acesso ao meio para controlar o *duty cicle* dos nós transmissores de RSSFs, com o objetivo de reduzir o consumo de energia por parte dos rádios transmissores.

1.3 Justificativa

Adicionar o suporte a operação em modo *tickless* ajuda a reduzir o consumo de energia, visto que é possível que o processador permaneça em modo de baixo consumo de energia enquanto aguarda a ocorrência de um evento ou interrupção, evitando assim, que o sistema acorde mesmo quando não há tarefa a ser executada.

Outra possibilidade de economia de energia está associada aos modos de baixo consumo de energia existentes tanto nos módulos de rádio quanto nos microcontroladores. Assim, torna-se possível otimizar o consumo de energia do sistema fazendo com que permaneçam nesse modo de baixo consumo e acordem somente para a transmissão ou recepção de pacotes.

1.4 Organização Textual

O restante deste trabalho está organizado da seguinte forma:

Capítulo 2 – Revisão Bibliográfica: apresenta os princípios e conceitos envolvidos na implementação de uma RSSF baseada em sistemas operacionais de tempo real.

Capítulo 3 – Materiais e Métodos: descreve como o trabalho foi desenvolvido, os recursos necessários para implementação da ideia proposta e a metodologia adotada.

Capítulo 4 – Resultados: apresenta os resultados obtidos através dos testes e medições realizados antes e após a implementação das técnicas apresentadas neste trabalho.

Capítulo 5 – Conclusões: apresenta as dificuldades e facilidades encontradas durante o desenvolvimento do trabalho e sugestões de trabalhos futuros.

2 REVISÃO BIBLIOGRÁFICA

Esta seção apresenta os princípios e conceitos encontrados na literatura envolvidos no desenvolvimento de uma RSSF baseada em sistemas operacionais de tempo real.

2.1 Sistemas Operacionais

Um sistema de computação pode basicamente ser dividido em quatro componentes: o *hardware*, o sistema operacional, os programas aplicativos e o usuário. A Figura 1 apresenta a visão abstrata dos componentes de um sistema computacional.

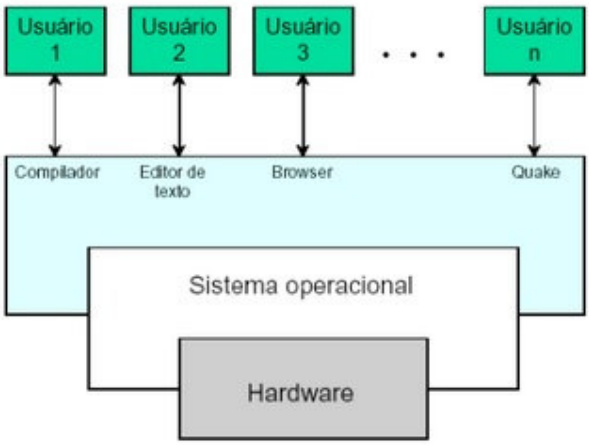

Figura 1 - Visão abstrata dos componentes de um sistema computacional
Fonte: Adaptado de Silberschatz (2010).

Um Sistema Operacional (SO) trata-se de um conjunto de programas elaborados para controlar e gerenciar o *hardware* e os recursos de *software* de um dispositivo. Ele também fornece uma base para os programas aplicativos e atua como intermediário entre o usuário e o *hardware* (TANEMBAUM, 2010). De forma resumida é possível listar como funções básicas de um SO: gerenciamento de tempo e recursos de CPU, gerenciamento de tarefas, gerenciamento de memória, gerenciamento de periféricos. Além disso também provê funcionalidades como, sistema de arquivos, protocolos de rede, etc.

Segundo Tanembaum (2010), um SO pode ser visto de duas maneiras: top-down e *bottom-up*. A chamada visão *top-down* é vista pela perspectiva do usuário ou projetista de aplicativos, provê abstração do *hardware*, fazendo o papel de intermediário entre o aplicativo e o *hardware* do sistema. De acordo com a visão *bottom-up* o SO opera como gerenciador de

recursos, ou seja, controla quais tarefas podem ser executadas, quando podem ser executadas e que recursos (*display*, comunicação, etc) podem ser utilizados.

Um aspecto interessante dos sistemas operacionais é a versatilidade, o quanto eles assumem diferentes abordagens ao cumprir essas tarefas de acordo com a aplicação para o qual ele seja utilizado. Dessa forma, alguns são projetados para serem convenientes, outros para serem eficientes, e outros para atender a alguma combinação de ambos aspectos (SILBERSCHATZ, 2010).

Dentre os diversos sistemas operacionais existentes, de *mainframe*, computadores pessoais (PCs), dispositivos móveis, tem os que se caracterizam por terem o tempo como parâmetro fundamental, os Sistemas Operacionais de Tempo Real.

2.2 Sistemas Operacionais de Tempo Real

Um Sistema Operacional de Tempo Real (RTOS – *Real Time Operating System)* é um sistema de computação que requer não somente que os resultados da computação estejam corretos, mas também que eles sejam produzidos dentro de um determinado prazo especificado. Resultados produzidos após esse prazo, mesmo se estiverem corretos, podem não ter valor real (SILBERSCHATZ, 2010).

A computação de tempo real é dividida em dois tipos: crítica e não-crítica.

> "Um sistema de tempo real crítico é aquele no qual as tarefas precisam necessariamente ocorrer em determinados instantes (ou em um determinado intervalo de tempo), o não cumprimento das restrições de tempo pode causar sérias consequências. Esse tipo de sistema é encontrado no controle de processos industriais, aviação e exército. Enquanto que um sistema de tempo real não crítico é aquele no qual o não cumprimento ocasional de um *deadline*, embora não desejável, é aceitável e não causa nenhum dano permanente" (TANEMBAUM, 2010).

RTOSes são frequentemente utilizados em aplicações embarcadas. Permitem ao desenvolvedor um conjunto de ferramentas que além de facilitar o desenvolvimento, permitem a mudança de plataforma caso seja necessário. Costumam consumir menos recursos do processador e podem ser utilizados em equipamentos com quantidade de memória menor se comparados aos sistemas operacionais de propósito geral (PEREIRA, 2007). No caso de um sistema embarcado destinado a uma rede de sensores não existem usuários, mas sim tarefas concorrentes, a partir da necessidade de gerenciar tarefas concorrentes com restrições temporais é recomendado o uso de um RTOS.

Entre os RTOSes mais populares disponíveis no mercado é possível citar o FreeRTOS, o QNX, o µC/OS, o BRTOS, entre outros. Neste trabalho são apresentadas as características principais do FreeRTOS, RTOS líder de mercado, e o BRTOS, que foi utilizado no sistema proposto.

2.2.1 RTOS Tickless

Em geral, um RTOS possui um tempo de avaliação periódica do sistema. Este tempo é a resolução mínima de execução a ser usada por uma tarefa e é comumente chamado de *tick* do sistema. Na Figura 2 é possível observar onde o sistema acorda a cada 1ms para verificar se existem atividades a serem executadas.

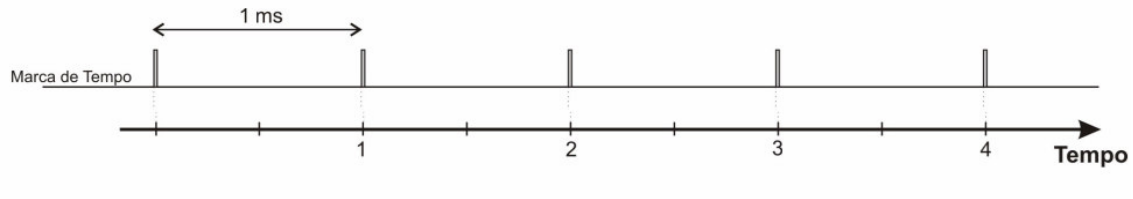

Figura 2 - Representação de *tick* com período igual a 1ms.
Fonte: Autoria própria.

A definição da frequência do *tick* é fundamental para o desempenho do sistema. Se a frequência for muito alta a energia e o tempo consumido ao entrar e sair de um estado de baixo consumo de energia para cada *tick* superam qualquer economia de energia (FreeRTOS, 2013). De forma oposta, se a frequência for muito baixa, pode haver perda da resolução de *soft timers*. Uma boa métrica é entre 1 e 10ms, porém muitas vezes as aplicações podem ficar horas sem precisar acordar e, portanto, o custo do *tick* torna-se muito alto nestes casos. Uma das principais abordagens para reduzir este custo é tornar o *tick* do sistema dinâmico.

O *tickless* é uma técnica que tem como objetivo principal reduzir o consumo de energia de um processador devido ao processamento do *tick* do sistema. Ao utilizar esta técnica, a ocorrência de interrupções do *tick* são reduzidas, evitando processamento desnecessário. Ainda, permite que o microcontrolador permaneça em um estado de baixo consumo de energia até que uma interrupção ocorra. No sistema *tickless* um ajuste no valor do contador de *tick* é realizado para corrigir no temporizador do sistema o tempo atual. A economia de energia que pode ser obtida por este método simples é limitada pela necessidade de sair periodicamente para processar interrupções e, em seguida, voltar a entrar no estado de baixo consumo.

A Figura 3 apresenta um exemplo de sistema com *tick* periódico. As marcações em vermelho indicam quando ocorre uma interrupção e/ou quando um processo deve ser

acordado. É possível observar que o sistema sai do modo de baixo consumo e volta a executar mesmo quando não há interrupção a ser tratada ou tarefa a ser executada, ocasionando um consumo de energia desnecessário. A Figura 4 apresenta um exemplo de sistema com suporte a *tickless*, onde é possível observar que o sistema acorda apenas quando há interrupção a ser tratada, o que reduz consideravelmente o consumo de energia.

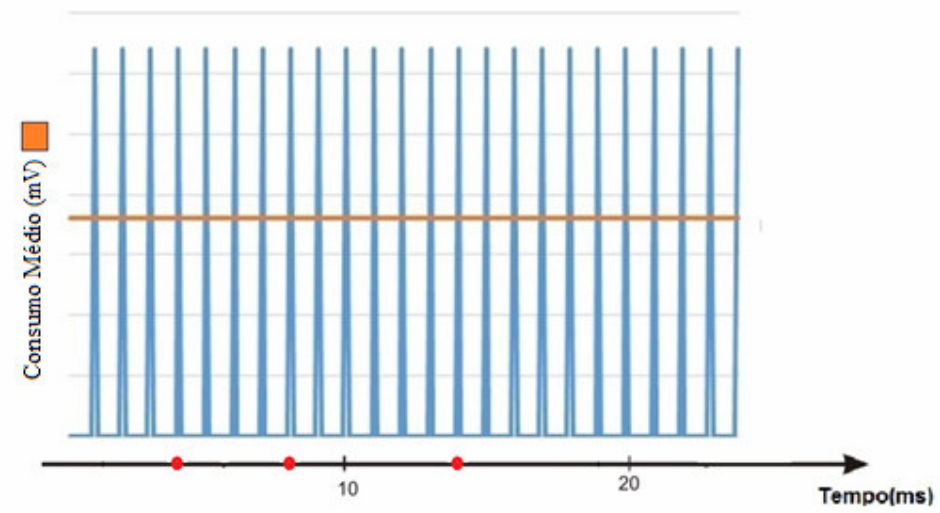

Figura 3 - Consumo médio de sistema com *tick* periódico
Fonte: Autoria própria

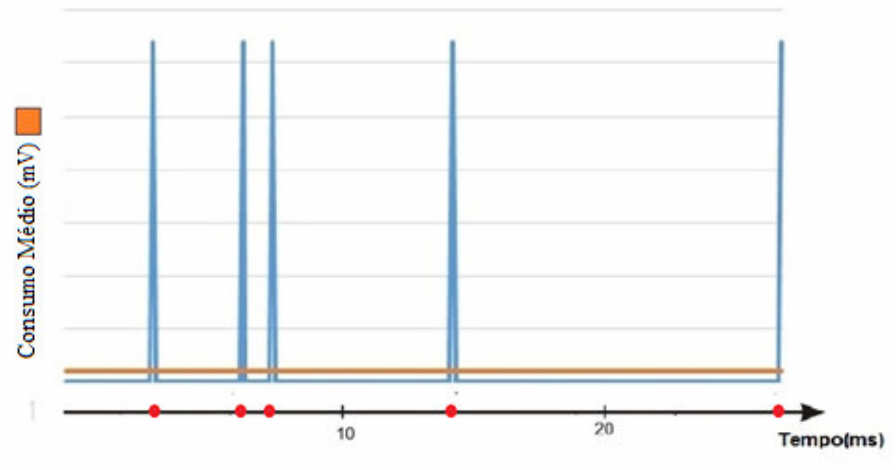

Figura 4 - Consumo médio de sistema *tickless.*
Fonte: Autoria própria

2.2.2 FreeRTOS

O FreeRTOS é um sistema operacional de tempo real pequeno, simples e de fácil uso. Um aspecto importante para o sucesso do FreeRTOS no mercado é a sua portabilidade, atualmente o sistema é oficialmente portável para 34 arquiteturas diferentes, entre elas a PIC, ARM e PC, arquiteturas bastante utilizadas pelas indústrias da área de microeletrônica (GALVÃO, 2009).

O FreeRTOS fornece para os seus usuários serviços de gerenciamento de tarefa, comunicação e sincronização entre tarefas, controle de memória, gerenciamento do tempo e controle dos dispositivos de entrada e saída, tornando assim o desenvolvimento das aplicações de tempo real mais simples e prático, já que dessa forma o foco do projetista é voltado para as funcionalidades do sistema e a comunicação com o *hardware* é tratada pelo FreeRTOS.

O algoritmo de escalonamento do FreeRTOS, assim como a maioria dos sistemas operacionais de tempo real, é baseada em prioridades. Isso indica que tarefas com maiores prioridades têm a preferência sobre as tarefas de menores prioridades do mesmo estado. Uma consequência desse algortimo é que a prioridade da tarefa que está em execução deve ser maior ou igual às demais tarefas de estado pronto.

Um exemplo desse algoritmo de escalonamento pode ser vista na Figura 5, na qual as tarefas de menor prioridade são interrompidas pelo escalonador assim que uma tarefa de maior prioridade encontra-se no estado pronto.

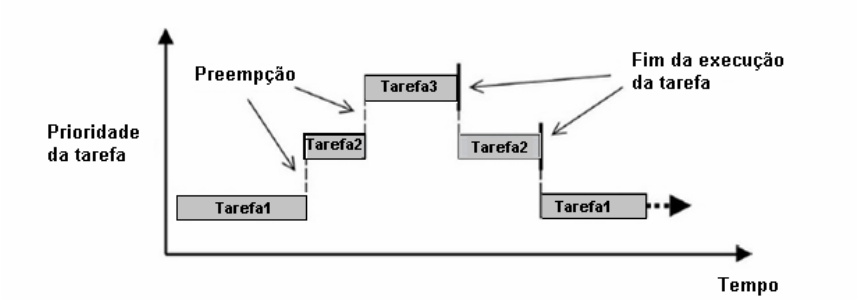

Figura 5 - Funcionamento do escalonador do FreeRTOS.
Fonte: Adaptado de Galvão (2009).

Vale ressaltar que as tarefas de mesma prioridade dividem o tempo de execução entre si, utilizando o algoritmo de escalonamento Round-Robin.

Além disso, o FreeRTOS possui suporte a *tickless* para microcontroladores de algumas arquiteturas, entre elas a ARM. No entanto, não utiliza os modos de baixo consumo de energia mais avançados desses processadores. De acordo com a documentação oficial do sistema, o *tickless* é implementado da seguinte forma: o *tick* do sistema é interrompido em

períodos em que a *Idle Task* é a única tarefa disponível para ser executada e as demais tarefas estão em estado bloqueado ou suspenso, em seguida é feito um ajuste no valor de contador do *tick* quando a interrupção do *tick* reinicia.

2.2.3 BRTOS

O BRTOS é um sistema operacional de tempo real desenvolvido com o objetivo de ser um sistema operacional simples, com o mínimo de consumo de memória de dados e programa, devido ao reduzido poder computacional e memória disponíveis nos microcontroladores de baixo custo de 8, 16 e 32 bits disponíveis no mercado. Além da reduzida carga computacional o sistema também teve como requisitos um baixo consumo de energia e a integração de pilhas de protocolos para redes de sensores sem fio (DENARDIN e BARRIQUELLO, 2010).

O BRTOS foi implementado com gerenciador de tarefas preemptivo baseado em prioridades. Além do gerenciamento de tarefas, disponibiliza gerenciamento de tempo e memória, sincronização de tarefas, semáforos, *mutex* (semáforo de acesso mútuo exclusivo), filas e caixas de mensagens.

A licença utilizada no BRTOS é a MIT. Esta licença de *software* foi criada pelo Massachusettes Institute of Technology. A licença MIT é permissiva, ou seja, permite a reutilização de código licenciado dentro de *softwares* proprietários, desde que a licença seja distribuída com este *software*. A licença também é compatível com a licença GPL, o que significa que a GPL permite a combinação e redistribuição de códigos que utilizam a licença MIT.

2.2.4 Comparativo de desempenho: BRTOS x FreeRTOS

Segundo Laime (2007), "o desempenho de um RTOS está diretamente relacionado à plataforma em que foi desenvolvido, ao processador utilizado, à velocidade do *clock*, ao compilador e ao *design* empregado".

Como o objetivo deste projeto é reduzir ao máximo o consumo de energia por parte dos processadores que compõem os dispositivos de uma rede de sensores, o RTOS utilizado tem papel fundamental para que essa meta seja alcançada.

Para a avaliação de desempenho de um RTOS é necessário utilizar uma ferramenta de benchmark específica para este propósito. Para o comparativo entre o BRTOS e o FreeRTOS foi utilizada a Suite Thread-Metric, um benchmark *open-source*. Essa Suite

permite medir a sobrecarga de diversos RTOSes em um determinado processador através de um conjunto de testes e comparar o desempenho de diferentes processadores executando o mesmo RTOS. Cada teste possui um tempo de duração igual a 30 segundos, ao fim de cada tarefa o contador de programa é incrementado, ao final do teste quanto maior for o número do contador, mais vezes a tarefa foi realizada durante o intervalo de tempo determinado e melhor o desempenho do RTOS nesse quesito (KHAOULE, 2011).

A Thread-Metric consiste dos seguintes teste de benchmark:

- 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

De acordo com Khaoule (2011), utilizando um microcontrolador *Freescale ColdFire* V1 e as versões 1.6.5 e 6.1.0 do BRTOS e do FreeRTOS, respectivamente, nos testes da Thread-Metric foram obtidos os resultados, com base no valor do contador de programa, apresentados nas tabelas 1 e 2.

Tabela 1 - Resultados do 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	**4597600**

Fonte: Khaoule (2011, p. 50).

Tabela 2 - Resultados do teste do BRTOS**.**

Processamento de mensagens	**926802**

Processamento de interrupção com preempção	**1827156**
Processamento de interrupção sem preempção	**3210365**
Troca de contexto preemptivo	**2410233**
Processamento de semáforo	**5312494**
Processamento básico	**4602100**

Fonte: Khaoule (2011, p. 50-51).

A partir dos valores apresentados nas tabelas 1 e 2 é possível construir o gráfico apresentado pela Figura 6.

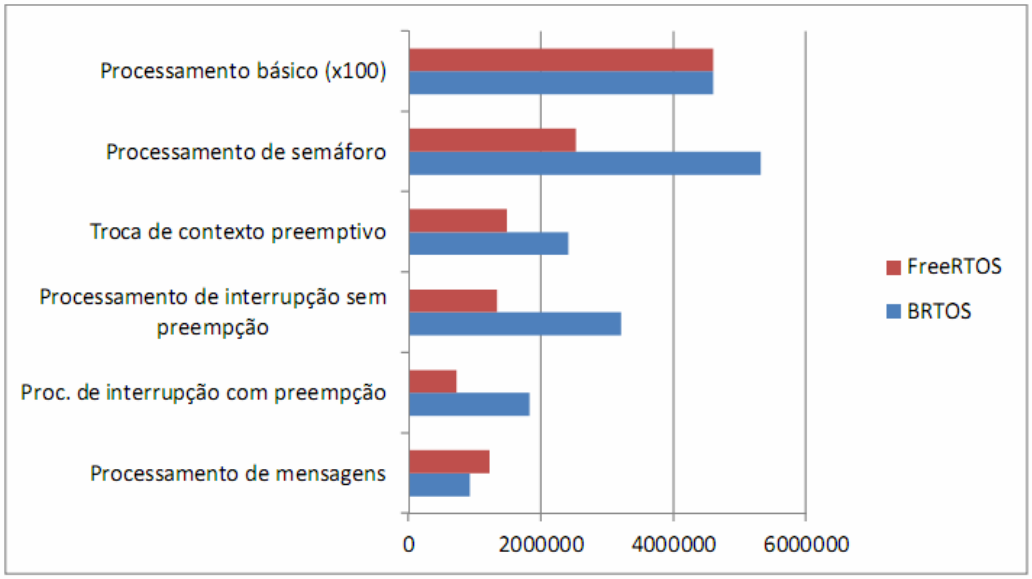

Figura 6 - Comparação entre os desempenhos do BRTOS e FreeRTOS.
Fonte: Khaoule (2011).

Com base no gráfico apresentado na Figura 6 é possível verificar uma superioridade de desempenho do BRTOS na maioria dos testes, obtendo desempenho inferior ao FreeRTOS apenas no processamento de mensagens, e desempenho equivalente em relação ao processamento básico.

Atualmente o BRTOS encontra-se na sua versão 1.7x, que sofreu correções, modificações e melhorias. A partir da versão 1.75 foi implementado um melhor serviço de filas, que possibilita a criação de filas para diferentes tamanhos de dados, melhorando, dessa forma, o desempenho no teste de processamento de mensagens. Essa alteração permitiu ao BRTOS obter desempenho equivalente ao FreeRTOS em relação ao processamento de mensagens. (DENARDIN e BARRIQUELLO, 2010).

De acordo com o desempenho do RTOS é possível que o sistema execute as tarefas em um menor intervalo de tempo. Devido a esse processamento mais eficiente é possível retornar ao modo de baixo consumo de energia mais rapidamente.

2.3 Redes de Sensores sem Fio

Devido à facilidade de instalação em ambientes com pouco acesso, as redes de sensores utilizando comunicação sem fio estão sendo cada vez mais aplicadas aos mais diferentes cenários. A facilidade de instalação deve-se a estas redes não necessitarem de estruturas de cabeamento e configurações individuais para iniciar uma rede. Segundo Denardin (2012), "estas redes possuem características únicas, como por exemplo: preocupações com consumo de energia, baixo ciclo de trabalho, fluxo de dados concentrados, processamento distribuído, entre outros."

Uma RSSF pode ser definida como uma rede composta por nós sensores que, cooperativamente, monitoram o ambiente em que estão inseridos. Aplicações típicas para redes de sensores incluem monitoramento de ambientes, gestão de infraestrutura, coleta de dados, aplicações militares, entre outras.

Os nós sensores são dispositivos que normalmente possuem um transceptor de rádio, sensores, fonte de energia, microcontrolador, módulos de localização e atuadores. A Figura 7 apresenta o diagrama de blocos de um nó sensor genérico:

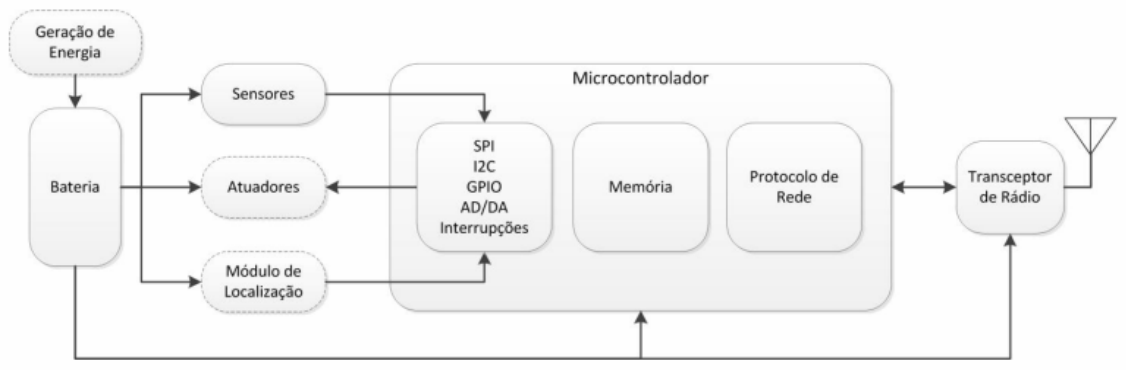

Figura 7 - Diagrama de blocos de um nó sensor.
Fonte: Soares (2012).

Na maioria das aplicações de RSSFs, a rede possui apenas um nó coordenador (*sink),* que normalmente é estático. Este nó é responsável por iniciar uma rede, bem como por coletar, processar e analisar as informações disponibilizadas pelos sensores.

Segundo Denardin (2012), as redes de sensores podem ser classificadas em duas categorias:

- Categoria 1 – quase sempre baseadas em redes em malha com conectividade multi-hop, utilizando roteamento dinâmico.
- Categoria 2 – redes ponto-a-ponto ou multiponto-a-ponto (topologia em estrela) com conectividade single-hop, utilizando roteamento estático.

Além disso, RSSF podem ter as seguintes configurações, conforme apresentado na Figura 8.

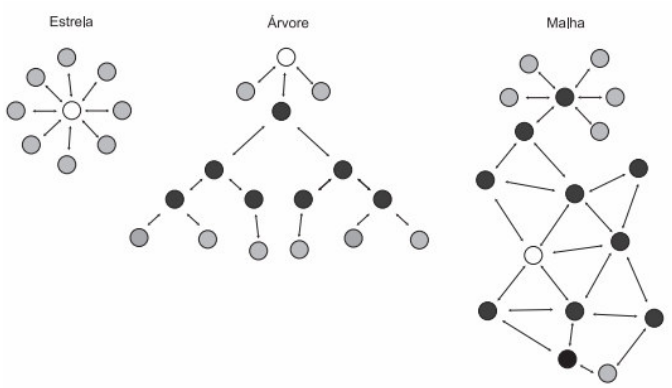

Figura 8 - Topologias de rede suportadas pelo padrão IEEE 802.15.4™.
Fonte: Denardin (2012, p. 30).

Normalmente a quantidade de nós e o alcance limitado dos rádios requer o uso de protocolos de acesso ao meio para evitar a colisão e perda de pacotes.

Atualmente o padrão de comunicação sem fio IEEE 802.15.4™ é o protocolo de camada física e de acesso ao meio para aplicação em redes de sensores e sem fio. O objetivo principal da sua criação é oferecer suporte à conectividade sem fio para uma grande variedade de aplicações industriais, residenciais e médicas.

2.3.1 IEEE 802.15.4™

O padrão IEEE 802.15.4™ especifica a camada física e a subcamada de controle de acesso ao meio para redes sem fio de área pessoal (WPAN – Wireless Personal Area Networks). As redes WPAN são projetadas para pequenas distâncias, baixo custo e baixas taxas de transferência. Uma rede WPAN é composta por dois tipos de dispositivos: dispositivos de função completa (FFD – *Full Function Devices*) e dispositivos de função reduzida (RFD – *Reduced Function Devices).* Um dispositivo FFD pode se comunicar com dispositivos FFD e RFD, enquanto dispositivos RFD somente se comunicam com dispositivos

FFD. (Denardin, Roteamento Geográfico para redes de sensores e atuadores sem fio em redes urbanas de comunicação, 2012).

Certas redes definem tipos de dispositivos lógicos com base nos tipos de dispositivos de uma rede WPAN. No padrão ZigBee, por exemplo, existem três tipos de dispositivos, classificados quanto a sua função na rede (ZigBee Alliance, 2006):

- Coordenador da rede: este dispositivo é do tipo FFD e é responsável por iniciar e definir os principais parâmetros da rede. Pode também ser utilizado como *gateway* para redes maiores, como a internet, e normalmente é o destino da maior parte dos dados adquiridos por outros nós pertencentes a rede;
- Roteador: é um dispositivo FFD que suporta o roteamento de pacotes de dados operando como um dispositivo de enlace intermediário na conexão de diferentes componentes da rede. Através destes dispositivos uma mensagem de dados é encaminhada por múltiplos saltos até o seu destino;
- Dispositivo final: são dispositivos RFD que contém as funcionalidades mínimas para se comunicar com um coordenador ou roteador.

O protocolo MAC deve tornar eficiente e confiável o acesso ao meio para os diversos dispositivos que compõem a rede, evitando que mais de um nó acesse o meio ao mesmo tempo. O mecanismo de acesso ao meio utilizado no padrão IEEE 802.15.4™ é conhecido por CSMA/CA (Carrier Sense Multiple Access with Collision Avoidance). Um nó que desejar acessar o meio utilizando esta técnica deve inicialmente verificar se o canal está ocupado. Caso o canal esteja desocupado, o nó poderá utilizá-lo para transmitir um pacote. Se estiver ocupado, o nó deve aguardar um tempo pré-determinado antes de voltar a analisar o canal. Este tempo pré-determinado, mas aleatório, evita que vários nós esperem um mesmo tempo e tentem acessar o meio concorrentemente (Denardin, Roteamento Geográfico para redes de sensores e atuadores sem fio em redes urbanas de comunicação, 2012).

2.4 Arquitetura ARM

Os microcontroladores e microprocessadores ARM (*Advanced* RISC *Machine*) surgiram no início da década de 80 a partir de um projeto de um fabricante britânico de computadores chamado Acorn Computer Group. Na época a Acorn, assim como seus concorrentes (Atari, Apple, etc.), fabricava microcomputadores que utilizavam microprocessadores de 8 bits. A partir da necessidade de um microprocessador mais rápido,

de programação simples e construção barata, e após pesquisas e análises das opções disponíveis no mercado, a empresa decidiu iniciar um projeto inovador de microprocessador (PEREIRA, 2007).

Os microprocessadores são CPUs de 32 bits que utilizam a filosofia RISC (Reduced Instruction Set Computer). Esses processadores são um marco na história da indústria de semicondutores devido à velocidade com que foram difundidos e a grande presença no mercado. Uma grande parcela desse sucesso deve-se às suas características, entre elas destacam-se design simples, alta velocidade, grande diversidade de modelos e fabricantes e quantidade de *software* disponível.

Outra vantagem relevante é a de que alguns ARMs possuem um *hardware* adicional para o controle e gerenciamento da memória. Existem basicamente duas arquiteturas de gerenciadores de memória disponíveis: as MMUs (*Memory Management Units*) e as MPUs (*Memory Protection Units*). Ambas têm como objetivo controlar o acesso à memória pela CPU, impedindo que programas executados num modo não privilegiado possam acessar posições de memória específicas (PEREIRA, 2007). Também é possível delimitar regiões de memória que não podem ser acessadas e especificar o tipo de acesso. Esse tipo de controle é fundamental aos sistemas operacionais, pois elimina os riscos de que uma aplicação sobrescreva áreas do seu código, além disso isola trechos do programa, impedindo que uma tarefa interfira em outra.

As principais características da arquitetura ARM são a simplicidade, baixo custo, pequeno consumo de energia e modularidade. As CPUs ARM foram projetadas para serem simples, baratas e integradas aos mais diversos periféricos, permitindo que sejam utilizadas nas mais diferentes aplicações. Se destacaram na aplicação para sistemas embarcados principalmente devido ao seu baixo consumo de energia, sendo assim ótimos candidatos a serem utilizados em redes de sensores sem fio.

Existem diversas variações das CPUs ARM, diferenciadas pela versão da arquitetura e família a qual pertencem. Atualmente a arquitetura ARM encontra-se na sua sétima versão e garante desempenhos mais elevados.

2.4.1 Família ARM CORTEX

A família ARM CORTEX utiliza a sétima geração da arquitetura ARM e permite atingir níveis de desempenho superiores aos das versões anteriores, podendo atingir a marca de 2000 MIPS @ 1GHz. Entre suas principais características destacam-se a arquitetura

superescalar (execução de duas instruções simultaneamente) com pipeline de até 13 estágios, predição dinâmica de desvios, memória cache primária(L1) de 16 ou 32Kb e secundária (L2) de 64 Kb a 2Mb (PEREIRA, 2007).

A família CORTEX divide-se em três grupos:

- CORTEX-A: focada em aplicações de alto desempenho baseadas em sistemas operacionais que utilizem sistemas de memória virtual.
- CORTEX-R: focada em aplicações de alta performance e tempo real.
- CORTEX-M: versão simplificada da família CORTEX e focada em aplicações de baixo custo e baixo consumo de energia.

De acordo com ARM (2013), o CORTEX-M0+ é o menor processador ARM disponível. Foi projetado para obter alta eficiência energética, consequência da redução do número de acessos a memória flash e a ROM, obtida através da redução de 3 estágios de pipeline (no CORTEX-M0) para 2 (no CORTEX-M0+). Além disso o consumo de energia dos elementos do sistema também foi otimizado. A rápida execução de código proporcionada pelo conjunto de instruções *Thumb* aliada a uma alteração no acesso as I/Os que otimiza esse processo permite reduzir o *clock* do processador ou aumentar o tempo do modo de suspensão.

3 MATERIAIS E MÉTODOS

3.1 Materiais

Neste tópico são apresentados os recursos utilizados no desenvolvimento do projeto e algumas características sobre as suas relevâncias para o trabalho.

3.1.1 Recursos de Software

Para o desenvolvimento do suporte ao modo de operação *tickless* e programação do microcontrolador foi escolhida a IDE baseada em eclipse CoIDE, um ambiente *open-source* para o desenvolvimento de *software* para microcontroladores ARM CORTEX, onde a linguagem de programação utilizada é C. A CoIDE suporta diversos compiladores e para o desenvolvimento deste trabalho foi utilizado o GNU GCC. Ainda inclui todas as ferramentas necessárias para o desenvolvimento, sendo que entre as mais utilizadas estão o CoDebugger que permite realizar a depuração de programas executados tanto na memória *flash* quanto na memória RAM.

Para o gerenciamento das tarefas concorrentes com restrições temporais do *software* da RSSF o RTOS utilizado foi o BRTOS, que obteve melhores resultados nos testes apresentados na seção anterior.

3.1.2 Recursos de Hardware

Para o desenvolvimento do projeto foi utilizada a ferramenta de desenvolvimento FRDM-KL25Z da fabricante Freescale™, que possui um CORTEX-M0+. Essa plataforma possui suporte ao OpenSDA, um adaptador padrão para depuração de sistemas microcontrolados, e várias opções para comunicação serial, programação da memória *flash* e *debug*.

Figura 9 - Ferramenta de desenvolvimento FRDM-KL25Z.
Fonte: Mouser Eletronics (2014).

Para a implementação da rede foram utilizados os rádios MRF24J40MA, da Microchip Technology®, que segundo o *datasheet* do fabricante, consomem 2 μA em modo *sleep*, 23mA durante a transmissão de dados e 19mA durante o modo de espera de recepção.

Figura 10 - Módulo de rádio MRF24J40MA.
Fonte: Farnell (2014).

Um computador pessoal também foi utilizado no desenvolvimento do trabalho.

3.2 Metodologia

Para o desenvolvimento e implementação do trabalho optou-se por, primeiramente, implementar o suporte ao modo de operação *tickless* no BRTOS e, em seguida, integrar a pilha de protocolos de uma RSSF a este sistema. As etapas realizadas durante o desenvolvimento do trabalho estão apresentadas no fluxograma a seguir.

Figura 11 – Fluxograma de desenvolvimento.
Fonte: Autoria própria.

3.2.1 Implementação do suporte ao modo *tickless* no BRTOS

Como comentado anteriormente, a escolha do processador e do sistema operacional para se implementar o suporte ao modo *tickless* é de fundamental importância. Assim, tendo-se definido o RTOS é possível iniciar o desenvolvimento do modo de operação supracitado utilizando as especificidades do ARM CORTEX-M0+, o qual já possui port oficial do BRTOS.

Assim, segundo Martinez e Manzanarez (2012) os modos de execução disponíveis para o CORTEX-M0 da Freescale™ são:

- Run
- Wait
- Stop
- VLPR (Very Low Power Run)
- VLPW (Very Low Power Wait)
- VLPS (Very Low Power Stop)
- LLS (Low Leakage Stop)
- VLLS3 (Very Low Leakage Stop 3)
- VLLS2 (Very Low Leakage Stop 2)
- VLLS1 (Very Low Leakage Stop 1)
- VLLS0 (Very Low Leakage Stop 0)
- BAT (Backup battery only)

O objetivo de tornar o *tick* do sistema dinâmico é permitir ao microcontrolador permanecer em um modo de baixo consumo de energia até a ocorrência de uma interrupção ou evento. Dessa forma é necessário conhecer a quantidade de energia consumida pelo processador em cada um dos modos disponíveis para que seja escolhido o mais adequado ao sistema.

No *datasheet* do fabricante é possível encontrar as informações a respeito do consumo de energia do microcontrolador em cada modo de execução, a tabela 3 apresenta o consumo de corrente do microcontrolador no modo *Run* nas seguintes condições: *clock* do processador igual a 48MHz, *clock* de barramento e *flash* igual a 24MHz, *clocks* periféricos desligados e executando um *loop* infinito a partir da memória *flash*, quando alimentado a 3.0V.

Tabela 3 - Consumo de corrente do microcontrolador no modo Run com *clocks* periféricos desligados.

Modo	Consumo Médio	Consumo Máximo	Unidade
Run	5.1	6.3	mA

Fonte: Freescale (2014).

A tabela 4 apresenta o consumo de corrente do microcontrolador no modo Run nas mesmas condições citadas anteriormente porém, com os *clocks* periféricos ligados.

Tabela 4 - Consumo de corrente do microcontrolador no modo Run com *clocks* periféricos ligados.

Modo	Temperatura de operação	Consumo Médio	Consumo Máximo	Unidade
Run	25°C	6.4	7.8	mA
	125°C	6.8	8.3	mA

Fonte: Freescale (2014).

De acordo o *datasheet* do fabricante o modo com menor consumo de energia é o VLLS0, no qual o consumo médio de corrente do microcontrolador é de 760 nA e o máximo é de 3577nA, à 50 °C, porém nesse modo a memória RAM é completamente desligada e a cada transição para o modo *Run* ocorre um *reset*, tornando inviável a utilização desse modo visto que o sistema não pode ser reiniciado a cada ocorrência de *tick* e as informações do sistema não podem ser perdidas. Depois do modo VLLS0 os modos que consomem menos energia são VLLS1, VLLS2 e VLLS3, que também são inviáveis devido a cada transição para o modo *Run* ocorrer um *reset*.

O modo com menor consumo de corrente e que é viável para o sistema é o modo LLS. Nesse modo toda a memória RAM permanece ligada e a transição para o modo *Run* pode ocorrer através de uma interrupção sem ocorrer um *reset*. A Tabela 5 foi extraída do *datasheet* do fabricante e apresenta o consumo de corrente do microcontrolador quando no modo LLS e alimentação igual a 3.0V.

Tabela 5 - Consumo de energia do microcontrolador no modo LLS.

Modo	Temperatura de operação	Consumo Médio	Consumo máximo	Unidade
LLS	25°C	1.9	3.7	µA
	50°C	3.6	39	µA
	70°C	6.5	43	µA
	85°C	13	49	µA
	100°C	30	69	µA

Fonte: Freescale (2014).

Apesar de evidente a economia proporcionada pelo modo LLS foram necessários alguns testes já que os valores informados no *datasheet* pelo fabricante são válidos para condições bem específicas. No modo *Run*, por exemplo, o valor medido foi obtido enquanto o processador apenas executava um *loop* infinito. Para verificar o real consumo de energia do microcontrolador algumas medições foram realizadas. O FRDM-KL25Z permite verificar o consumo do microcontrolador através de um *header*, onde foi soldado um resistor de 10 ohms. Dessa forma foi possível medir a diferença de potencial no resistor e calcular a corrente consumida através da Primeira lei de ohm: V = R.i.

Por padrão, o BRTOS utiliza o modo *wait* para economia de energia no ARM. Durante os primeiros testes o microcontrolador estava executando uma versão do BRTOS com configuração padrão para economia de energia e com o seguinte cenário: período de *tick* igual a 1ms e a única tarefa executada era a leitura do acelerômetro presente no kit, que foi programada para acontecer a cada 100ms, conforme é possível verificar nas Figuras 11 e 12. Além disso foi possível calcular a corrente fluindo no microcontrolador, aproximadamente 4.54mA.

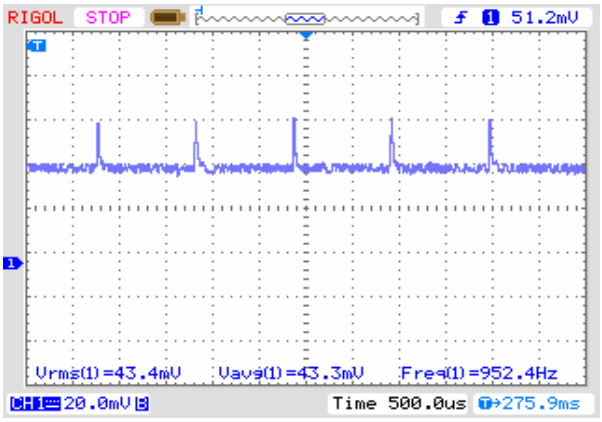

Figura 11 - Período de *tick* igual a 1ms durante o modo *Run*.
Fonte: Autoria própria.

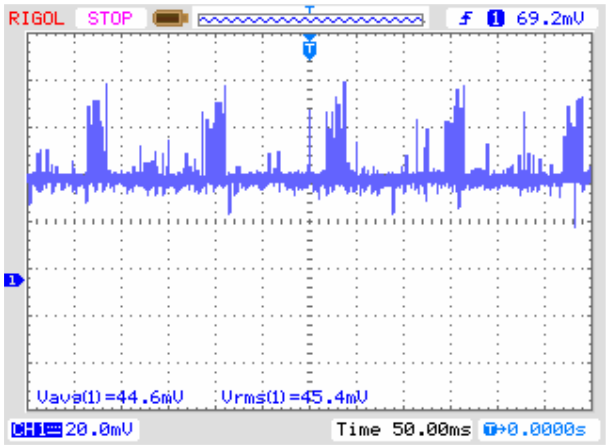

Figura 12 - Tarefa para leitura do acelerômetro com período igual a 100ms.
Fonte: Autoria própria.

Em seguida foi necessário configurar o microcontrolador para entrar no modo LLS durante os períodos de *sleep*. Primeiramente definiu-se a fonte de *clock* utilizada pelo timer no modo de baixo consumo pois o *clock* do processador é desligado nesse modo. O Power Management Controller (PMC) gera um *clock* de 1 kHz, que está habilitado em todos modos de operação, incluindo todos os modos de baixo consumo, exceto VLLS0. Esta fonte de 1 kHz é comumente referida como LPO.

Antes de entrar no modo LLS, deve ser configurada a LLWU (Low-Leakage Wake-up Unit) para tratar os eventos que irão acordar o microcontrolador. A interrupção de LLWU é a única que ocorre no modo LLS e permite acordar o processador por alguns módulos, como o *low-power timer* (LPTMR) ou por transição de pino.

O *timer* é reconfigurado antes de entrar em modo de baixo consumo para somente acordar quando a tarefa mais próxima de ser executada esteja pronta para executar. Para

calcular este tempo em relação ao tempo atual é necessário ter acesso ao valor atual da contagem no sistema, para isso o BRTOS possui a função OSGetTickCount. Para garantir que a função retorne o valor correto do contador de *tick* é necessário que as interrupções sejam desabilitadas antes da leitura da variável que armazena esse valor. Dessa forma o sistema garante que não ocorrerá uma interrupção que poderá alterar o valor do contador de *tick* antes da leitura pela função. Em seguida as interrupções são habilitadas novamente. O escopo da função OSGetTickCount é apresentado na Figura 13.

```
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
/////      Get the current tick count                              /////
////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////
INT16U OSGetTickCount(void)
{
  OS_SR_SAVE_VAR
  INT16U cnt;

  OSEnterCritical();
  cnt = OSTickCounter;
  OSExitCritical();
  return cnt;
}
```

Figura 13 - Função OSGetTickCount.
Fonte: Autoria própria.

Para tratar das transições entre os modos de execução do microcontrolador, tratamento das interrupções e ajuste do contador foram criadas as funções BRTOSStopModeSet e BRTOS_WakeUP.

A função BRTOSStopModeSet foi criada para selecionar o modo de execução do microcontrolador durante os períodos de *sleep* e calcular o tempo de acordar da próxima tarefa. Uma parte do escopo da função BRTOSStopModeSet está apresentada na Figura 14.

```
void BRTOSStopModeSet(unsigned long ulStopMode)
{
    volatile int dummyread;
    //
    // Check the arguments.
    //
    xASSERT((ulStopMode == SYSCTL_STOP_MODE_VLPS) ||
            (ulStopMode == SYSCTL_STOP_MODE_LLS)  ||
            (ulStopMode == SYSCTL_STOP_MODE_VLLS0)||
            (ulStopMode == SYSCTL_STOP_MODE_VLLS1)||
            (ulStopMode == SYSCTL_STOP_MODE_VLLS2)||
            (ulStopMode == SYSCTL_STOP_MODE_VLLS3)||
            (ulStopMode == SYSCTL_STOP_MODE_NORMAL));
```

Figura 14 - Função BRTOSStopModeSet.
Fonte: Autoria própria.

Em seguida o arquivo responsável pela abstração da camada de *hardware* (HAL – Hardware Abstract Layer) foi alterado para chamar a função BRTOSStopModeSet durante os períodos de *wait* quando ativada uma *flag* de *tickless*, conforme pode ser visto na Figura 15.

```
/// Defines the low power command of the choosen microcontroller
#if TICKLESS == 1
void BRTOSStopModeSet(unsigned long ulStopMode);
#define OS_Wait BRTOSStopModeSet(0x00000001)
#else
#define OS_Wait __asm(" WFI ");
#endif
```

Figura 15 - Alteração na camada de abstração de *hardware*.
Fonte: Autoria própria.

Em seguida foi medido o consumo médio de energia do microcontrolador ao se substituir o *wait* usado no teste anterior pelo LLS. A corrente atingiu a média de 567µA, conforme pode ser visto na Figura 16.

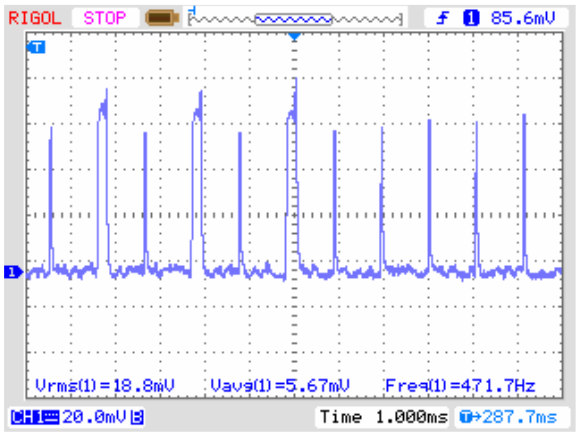

Figura 16 - Período de tick igual a 1ms durante modo LLS.
Fonte: Autoria própria.

A maior dificuldade encontrada ao se trabalhar com o modo LLS foi a ausência de meios de depuração. Quando no modo LLS, a ferramenta de depuração da CoIDE não funcionava de forma adequada impossibilitando o processo de depuração do código. Para isso foi necessário realizar o *debug* utilizando outro modo do microcontrolador para detectar eventuais falhas e corrigi-las para em seguida executar o sistema no modo LLS para medir o real consumo de energia. Como não foi possível utilizar a ferramenta de depuração tradicional no modo de baixo consumo de energia, utilizou-se uma estratégia diferente de depuração para corrigir eventuais problemas ainda existentes: comandos para o LED RGB da FRDM-KL25Z para acompanhar o comportamento do sistema, além de um osciloscópio utilizado para monitorar o pino de interrupção.

A transição para o modo *Run* deve acontecer quando ocorrer uma interrupção por *timer*, indicando o tempo em que uma tarefa da lista de tarefas prontas deve ocorrer, ou uma interrupção por pino, utilizada para indicar quando um pacote de dados foi recebido pelo rádio ou transmitido com sucesso. A função BRTOS_WakeUP determina a fonte da interrupção, e no tratamento da interrupção é utilizada a função OSIncCounter para atualizar o valor do contador do sistema. A função OSIncCounter padrão do BRTOS incrementa em um o valor do contador de *tick* até um valor máximo de 64000 armazenado em uma variável de 16 bits. Para implementação do *tickless* ela teve que ser alterada para aceitar valores de incremento diferentes de um.O escopo da função OSIncCounter modificada é apresentado na Figura 17.

```
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
/////      Update the tick count                           /////
////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////
#if (TICKLESS == 1)
void OSIncCounter(INT16U inc)
{
    if(inc !=1)
    {
        INT32U timeout = (INT32U)((INT32U)OSTickCounter + (INT32U)inc);

        if (timeout >= TICK_COUNT_OVERFLOW)
        {
          OSTickCounter = (INT16U)(timeout - TICK_COUNT_OVERFLOW);
        }
        else
        {
          OSTickCounter = (INT16U)timeout;
        }
    } else
    {
        OSTickCounter++;
        if (OSTickCounter == TickCountOverFlow) OSTickCounter = 0;
    }
}
```

Figura 17 - Função OSIncCounter modificada.
Fonte: Autoria própria.

Caso ocorra interrupção por *timer* o valor do contador do *tick* é atualizado pela função OSIncTickless, que desabilita as interrupções e aplica a correção no valor do contador de *tick* incrementado pelo tempo até a execução da próxima tarefa pronta mais um para em seguida reconfigurar o *timer*. A Figura 18 apresenta o escopo da função OSIncTickless.

```
void OSIncTickLess(void)
{
    OS_SR_SAVE_VAR

    OSEnterCritical();
    if(normal_run==0)
    {
        normal_run=1;
        GPIOPinReset(0x400FF040, 0x00040000);
        OSIncCounter(time_next_task+1);

        LPTMDisable();
        Reconfigure_LPTM(0);
        LPTMEnable();
    } else
    {
        OSIncCounter(1);
    }
    OSExitCritical();

}
```

Figura 18 - Função IncTickless.
Fonte: Autoria própria.

Novamente o arquivo responsável pela HAL foi alterado. A função responsável por tratar a interrupção de *timer* foi modificada para chamar a função OSIncTickless de acordo com uma diretiva de pré-compilação, conforme trecho do escopo da função apresentado na Figura 19.

```
void TickTimer(void)
{
  // ************************
  // Entrada de interrupcao
  // ************************

  // Interrupt handling
  TICKTIMER_INT_HANDLER;

#if (TICKLESS == 1)
  OSIncTickLess();
#else
  OSIncCounter(1);
#endif
```

Figura 19 - Função TickTimer modificada.
Fonte: Autoria própria.

Se ocorrer uma interrupção por pino o valor do contador é incrementado pelo tempo até a execução da próxima tarefa pronta mais o valor do contador do LPO.

Posteriormente foi testado o sistema após a implementação do suporte ao *tickless*. Nesse cenário, além da leitura de dados do acelerômetro, a cada 250ms, também era realizada a verificação do Watchdog timer com um período de 100ms. Utilizando a 1ª Lei de Ohm é possível calcular a corrente fluindo no sistema, que alcançou a média de 7,76µA, como pode ser visto na Figura 20.

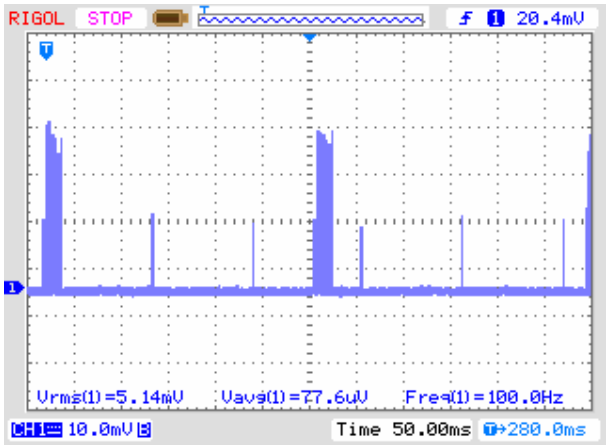

Figura 20 - Sistema com suporte à *tickless*.
Fonte: Autoria própria.

3.2.2 Integração dos protocolos da RSSF ao BRTOS *tickless*

Após implementar suporte ao *timer tickless* no RTOS, o próximo passo para buscar a redução do consumo de energia em uma RSSF foi integrar a pilha de protocolos de uma determinada RSSF a este sistema, bem como projetar métodos de redução de consumo de energia por parte do rádio RF. Devido às dificuldades encontradas para realizar a depuração quando utilizando o modo LLS optou-se por, primeiramente, integrar os protocolos da RSSF ao BRTOS com *tick* periódico.

RSSF são compostas por dispositivos lógicos que podem ser classificados como coordenador, roteador ou dispositivo final. Entre as topologias de rede utilizadas em RSSF a mais simples é a topologia em estrela.

Uma RSSF em topologia estrela é composta por inúmeros roteadores associados à um único coordenador. Sendo o foco do trabalho a redução do consumo de energia dos elementos que compõem uma RSSF e não a estrutura da rede em si, foi utilizada a estrutura mínima para representar uma RSSF. Portanto, um nó coordenador e um nó roteador foram

configurados de forma a manter uma comunicação utilizando a topologia em estrela. Além disso, na abordagem proposta, apenas o roteador foi configurado para entrar em modo *sleep* enquanto que o coordenador permanece ativo durante todo o tempo.

Foi necessária uma alteração no driver do roteador para que o rádio entrasse em modo *sleep.* Os rádios foram configurados para entrar em modo de economia de energia após o fim da leitura de um pacote de dados da rede, ao fim de envio de um pacote e configurados para acordar na interrupção do rádio.

Para avaliar o comportamento da rede foi instalada uma aplicação no roteador que tinha como objetivo solicitar a troca de estado de um LED para o nó coordenador. Já o nó coordenador foi preparado para receber os pacotes enviados por essa aplicação e alterar o estado do LED RGD do FRDM-KL25Z.

Para medir o consumo de energia do rádio RF, um resistor de 10 ohms foi soldado ao seu pino da alimentação. Dessa forma foi possível medir a diferença de potencial no resistor e calcular a corrente fluindo no circuito ao aplicar a Primeira lei de Ohm.

Devido ao referencial de tensão dos resistores utilizados para medir o consumo do rádio RF e do microcontrolador não serem o mesmo e, devido a instituição não possuir osciloscópios com ponteiras diferenciais não foi possível medir o consumo total do dispositivo (microcontrolador e rádio simultaneamente).

4 RESULTADOS

Neste capítulo são apresentados os resultados obtidos durante os testes para avaliar o consumo de energia dos elementos da RSSF antes e após a implementação das técnicas de baixo consumo de energia.

Primeiramente foi avaliado o consumo de energia do dispositivo coordenador, sem suporte a *tickless*. Na Figura 21 é apresentado o consumo médio do microcontrolador, onde é possível verificar o tempo de transmissão de um pacote seguido pelo tempo de recepção. Já na Figura 22 é apresentado o consumo médio do rádio RF, os picos de consumo são consequência da transição do modo de recepção para o modo de transmissão.

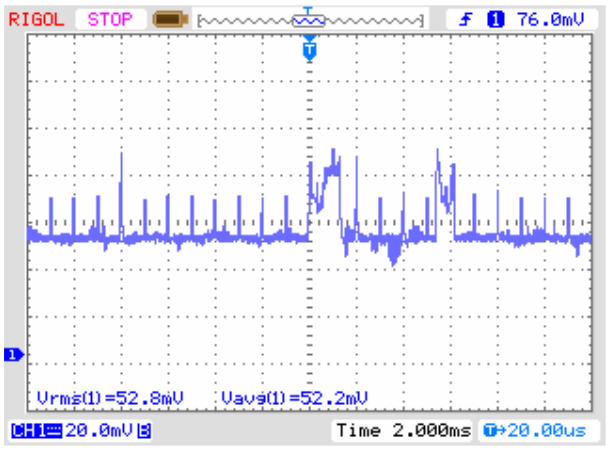

Figura 21 - Consumo do microcontrolador do coordenador com *tick*.
Fonte: Autoria própria.

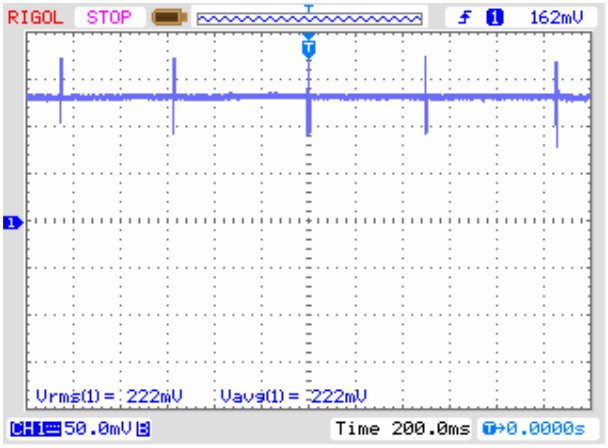

Figura 22 - Consumo do rádio do coordenador.
Fonte: Autoria própria.

Utilizando a primeira lei de ohm foi calculada a corrente fluindo no microcontrolador, aproximadamente 5,22mA, e no rádio, aproximadamente 22,2mA, totalizando 27,42mA no dispositivo coordenador.

Em seguida foi utilizada a versão do BRTOS com suporte ao *tickless* e novamente foi medida a corrente no microcontrolador, que atingiu a média de 207µA, conforme apresentado na Figura 23.

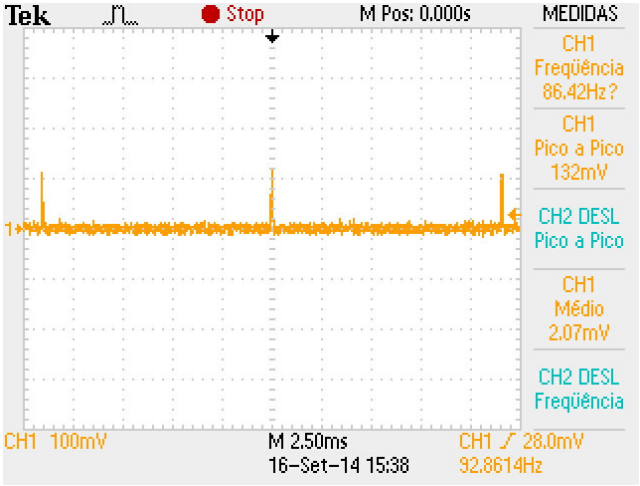

Figura 23 - Consumo do microcontrolador do coordenador com *tickless*.
Fonte: Autoria própria.

De forma análoga ao coordenador, avaliou-se o nó roteador. Primeiramente utilizou-se uma versão do BRTOS com *tick* periódico e foi medido o consumo de energia do microcontrolador e do rádio sem a implementação dos períodos de *sleep*. Os resultados obtidos estão apresentados na Figuras 24 e 25, sendo a corrente no dispositivo roteador igual a 26,84mA.

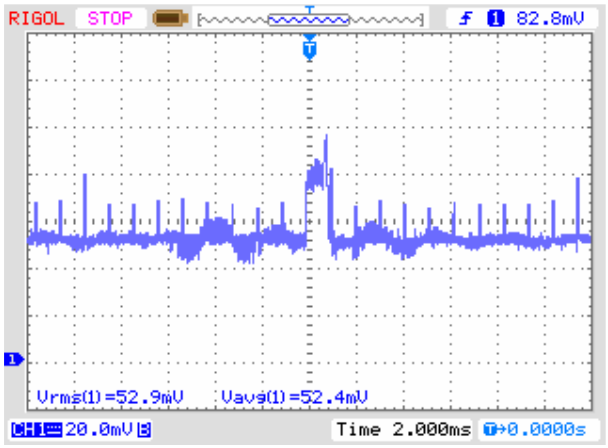

Figura 24 - Consumo do microcontrolador do roteador com *tick* periódico.
Fonte: Autoria própria.

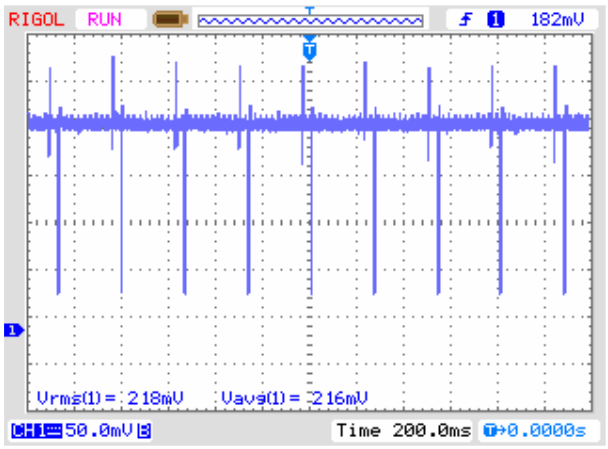

Figura 25 - Consumo do rádio do roteador sem modo de baixo consumo.
Fonte: Autoria própria.

Com o roteador preparado para entrar em modo de baixo consumo de energia, integrou-se a este driver modificado a versão do BRTOS com suporte ao modo *tickless*. A Figura 26 apresenta a corrente no microcontrolador do roteador com suporte ao *tickless*, aproximadamente 138µA.

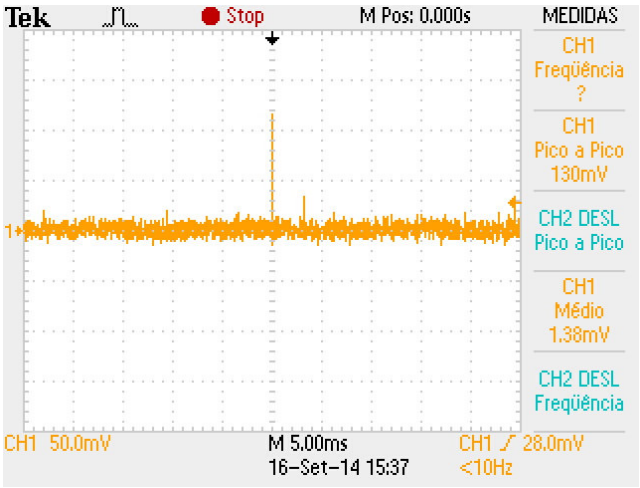

Figura 26 – Consumo do microcontrolador do roteador com suporte a tickless.
Fonte: Autoria própria.

A seguir mediu-se o consumo de energia do rádio do roteador após a implementação dos períodos de *sleep*. A Figura 27 apresenta o consumo médio do rádio acordando periodicamente a cada 1 segundo para transmissão de uma mensagem de *ping* necessária para a manutenção da rede.

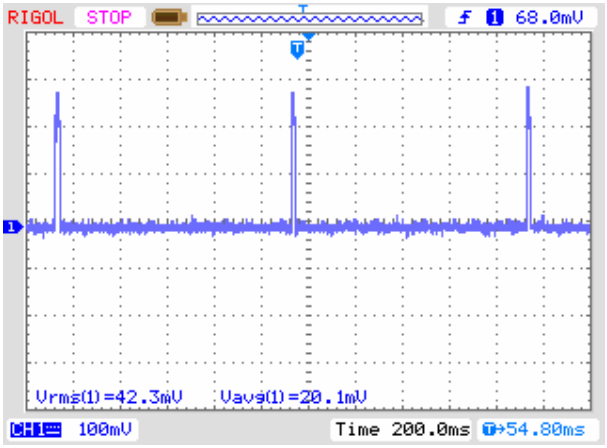

Figura 27 – Consumo do rádio do roteador com modo de baixo consumo.
Fonte: Autoria própria.

A corrente no dispositivo roteador após a implementação do suporte ao modo *tickless* e com o rádio entrando em modo *sleep* foi de 2.15mA.

A tabela 6 apresenta os dados referentes à corrente em cada dispositivo antes da aplicação das técnicas de baixo consumo, enquanto que a tabela 7 apresenta os dados referentes aos dispositivos após a aplicação das técnicas para reduzir o consumo de energia.

Tabela 6 - Resultados do consumo médio de cada dispositivo antes da aplicação das técnicas de baixo consumo.

	Consumo médio do microcontrolador	**Consumo médio do rádio**	**Consumo Total**	**Unidade**
Coordenador com *tick* periódico	5,22	22,20	27,42	mA
Roteador com *tick* periódico e rádio sem modo de baixo consumo	5,24	21,60	26,84	mA
Total			54,26	mA

Fonte: Autoria própria.

Tabela 7 - Resultados do consumo médio de cada dispositivo após a aplicação das técnicas de baixo consumo.

	Consumo médio do microcontrolador	**Consumo médio do rádio**	**Consumo Total**	**Unidade**
Coordenador *tickless*	0,21	22,20	22.41	mA

Roteador *tickless* e rádio com modo de baixo consumo	0,14	2,01	2,15	mA
Total			24,56	mA

Fonte: Autoria própria

É possível observar que o elemento que mais consome energia no sistema é o rádio. Na abordagem proposta o rádio do coordenador permanece ligado durante todo o tempo, implicando em uma redução no consumo médio do coordenador em 18,27%. Já no roteador, além do consumo no microcontrolador ter sido otimizado, o rádio também foi configurado para entrar em modo de baixo consumo, implicando em uma economia de 91,99% no consumo médio de energia do roteador. A economia no consumo médio do sistema foi de 54,74%.

5 CONCLUSÕES

Entre as técnicas para redução do consumo de energia de RSSFs existentes na literatura, a implementação de suporte ao *tickless* em um RTOS e a utilização de um protocolo de acesso ao meio para controle do *duty cycle* dos nós transmissores se mostraram mais adequadas devido ao baixo ciclo de trabalho característico de RSSFs. Essas técnicas otimizam o consumo de energia durante os períodos em que o sistema está ocioso.

O uso de um RTOS foi fundamental para se trabalhar com RSSFs pois ele possui os recursos necessários (gerenciamento de tempo, sincronização de tarefas, semáforos, entre outros) para gerenciar as tarefas concorrentes com restrições temporais existentes em RSSFs. Ao tornar o *tick* do sistema dinâmico o sistema evita o processamento das interrupções de *tick* mesmo quando não há eventos a serem tratados pela RSSF. Além de tornar o *tick* do sistema dinâmico, utilizar um microcontrolador ARM projetado para aplicações de baixo consumo de energia tornou mais expressiva a economia de energia obtida, superior a 95% no processador. Utilizar os recursos disponíveis pelo processador foi fundamental para obter resultados satisfatórios.

Um dos grandes obstáculos para o desenvolvimento de produtos na área de sistemas embarcados é o equilibro entre desempenho e consumo de energia. Um RTOS com suporte à *tickless* é recomendado para projetos nos quais baixo ciclo de trabalho seja uma característica e consumo de energia um requisito.

O nível de redução no consumo de energia de RSSFs está diretamente relacionado à aplicação para a qual a rede foi projetada (em razão das técnicas adotadas não reduzirem o consumo dos sensores e/ou atuadores utilizados) e à atividade na rede, visto que os rádios são os elementos que possuem o consumo de energia mais elevado quando ligados. A economia obtida através da utilização de um protocolo de acesso ao meio foi consequência do baixo ciclo de trabalho característico de RSSFs e na abordagem proposta reduziu o consumo médio do rádio do dispositivo roteador em aproximadamente 91%.

Para trabalhos futuros podem ser avaliadas redes com um número maior de elementos e/ou utilizando topologias que exigem um algoritmo de roteamento mais complexo, como árvore ou malha.

6 REFERÊNCIAS

ARAÚJO, R. B.; VILLAS, L. A.; BOUKERCHE, A. **Uma soluçào de QoS com Processamento Centrado para Redes de Atuadores e Sensores sem Fio**. Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos. [S.l.]: [s.n.]. 2007. p. 309-322.

ARM. Cortex-M0 Processor - ARM. **ARM® - The Archictecture for the Digital World**, 2013. Disponivel em: <http://www.arm.com/products/processors/cortex-m/cortex-m0.php>. Acesso em: 17 Agosto 2013.

BALL, S. R. **Embedded Microprocessor Systems:** Real World Design. 3ª. ed. [S.l.]: Newnes, 2009.

DENARDIN, G. W. **Roteamento Geográfico para redes de sensores e atuadores sem fio em redes urbanas de comunicação**. Santa Maria: [s.n.], 2012.

DENARDIN, G. W.; BARRIQUELLO, C. H. Brazilian RTOS blog | O blog oficial do BRTOS, um sistema operacional 100% brasileiro, 2010. Disponivel em: <http://brtosblog.wordpress.com>. Acesso em: 1 Junho 2013.

FAROOQ, M. O.; KUNZ, T. **Operating System for Wireless Sensor Networks: A Survey**. Carleton University. Ottawa. 2011.

FREERTOS. FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems supporting 34 microcontroller architectures. Disponivel em: <http://www.freertos.org>. Acesso em: 29 Maio 2013.

GALVÃO, S. S. L. **Modelagem do Sistema Operacional de Tempo Real FreeRTOS**. UFRN. Natal. 2009.

KHAOULE, S. S. **Análise da Performance de um Sistema Operacional de Tempo Real utilizando a ferramenta de benchmark Thread-Metric**. São Carlos: [s.n.], 2011.

LAIME, W. Electronics Design & Engineering - EE Times India, 2007. Disponivel em: <http://www.eetindia.co.in/ARTICLES/2007MAY/PDF/EEIOL_2007MAY03_EMS_INTD_TA.pdf>. Acesso em: 15 Agosto 2013.

MARTINEZ, D.; MANZANAREZ, C. **Using Low Power Modes on Kinetis Family**. Guadalajara: [s.n.], 2012.

PEREIRA, F. **Tecnologia ARM:** microcontroladores de 32 bits. 1ª. ed. São Paulo: Érica, 2007.

SILBERSCHATZ, A. **Fundamentos de Sistemas Operacionais**. 8ª. ed. Rio de Janeiro: LTC, 2010.

SOARES, S. A. F. **Rede de Sensores sem Fio para Localização e Monitoramento de Pequenos Ruminantes**. Bahia: [s.n.]. 2012.

TANEMBAUM, A. S. **Sistemas Operacionais Modernos**. 3ª. ed. São Paulo: Pearson, 2010.

WILL, H.; SCHELEISER, K.; SCHILLER, J. **A Real-Time Kernel for Wireless Sensor Networks Employed in Rescue Scenarios**. The 4th IEEE International Workshop on Pratical Issues In Building Sensor Network Applications. Zurich: [s.n.]. 2009. p. 20-23.

YE, H.; HEIDEMANN, J.; ESTRIN, D. **An Energy-Efficient MAC Protocol for Wireless Sensor Networks**. INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE (Volume:3). California: [s.n.]. 2002. p. 1567-1576.

ZIGBEE ALLIANCE. ZigBee Specifications. 2006. Disponível em: <http://www.zigbee.org>. Acesso em: 2 de Junho de 2014.

ANEXO A – TABELA DO CONSUMO DE ENERGIA DO ARM CORTEX-M0+ DA FREESCALE™.

Symbol	Description	Min.	Typ.	Max.	Unit	Notes
I_{DD_VLPR}	Very low power run mode current - 4 MHz core / 0.8 MHz bus and flash, all peripheral clocks enabled, code of while(1) loop executing from flash • at 3.0 V	—	300	745	µA	5, 4
I_{DD_VLPW}	Very low power wait mode current - core disabled / 4 MHz system / 0.8 MHz bus / flash disabled (flash doze enabled), all peripheral clocks disabled • at 3.0 V	—	135	496	µA	5
I_{DD_STOP}	Stop mode current at 3.0 V					
	at 25 °C	—	345	490		
	at 50 °C	—	357	827	µA	
	at 70 °C	—	392	869		
	at 85 °C	—	438	927		
	at 105 °C	—	551	1065		
I_{DD_VLPS}	Very-low-power stop mode current at 3.0 V					
	at 25 °C	—	4.4	16		
	at 50 °C	—	10	35	µA	
	at 70 °C	—	20	50		
	at 85 °C	—	37	112		
	at 105 °C	—	81	201		
I_{DD_LLS}	Low leakage stop mode current at 3.0 V				µA	
	at 25 °C	—	1.9	3.7		
	at 50 °C	—	3.6	39		
	at 70 °C	—	6.5	43		
	at 85 °C	—	13	49		
	at 105 °C	—	30	69		
I_{DD_VLLS3}	Very low-leakage stop mode 3 current at 3.0 V				µA	
	at 25 °C	—	1.4	3.2		
	at 50 °C	—	2.5	19		
	at 70 °C	—	5.1	21		
	at 85 °C	—	9.2	26		
	at 105 °C	—	21	38		
I_{DD_VLLS1}	Very low-leakage stop mode 1 current at 3.0V					
	at 25°C	—	0.7	1.4		
	at 50°C	—	1.3	13	µA	
	at 70°C	—	2.3	14		
	at 85°C	—	5.1	17		
	at 105°C	—	13	25		

Symbol	Description	Min.	Typ.	Max.	Unit	Notes
I_{DD_VLLS0}	Very low-leakage stop mode 0 current (SMC_STOPCTRL[PORPO] = 0) at 3.0 V	—	381	943	nA	
	at 25 °C	—	956	11760		
	at 50 °C	—	2370	13260		
	at 70 °C	—	4800	15700		
	at 85 °C	—	12410	23480		
	at 105 °C					
I_{DD_VLLS0}	Very low-leakage stop mode 0 current (SMC_STOPCTRL[PORPO] = 1) at 3.0 V	—	176	860		6
	at 25 °C	—	760	3577	nA	
	at 50 °C	—	2120	11660		
	at 70 °C	—	4500	18450		
	at 85 °C	—	12130	22441		
	at 105 °C					