UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
ESCOLA DE ENGENHARIA
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
PROGRAMA DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA

RODRIGO SCHMIDT ALLGAYER

FEMTONODE: ARQUITETURA DE NÓ-SENSOR RECONFIGURÁVEL E CUSTOMIZÁVEL PARA REDE DE SENSORES SEM FIO

Porto Alegre
2009

RODRIGO SCHMIDT ALLGAYER

FEMTONODE: ARQUITETURA DE NÓ-SENSOR RECONFIGURÁVEL E CUSTOMIZÁVEL PARA REDE DE SENSORES SEM FIO

Dissertação de mestrado apresentada ao Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal do Rio Grande do Sul como parte dos requisitos para a obtenção do título de Mestre em Engenharia Elétrica.
Área de concentração: Controle e Automação

ORIENTADOR: Prof. Dr. Carlos Eduardo Pereira

Porto Alegre
2009

RODRIGO SCHMIDT ALLGAYER

FEMTONODE: ARQUITETURA DE NÓ-SENSOR RECONFIGURÁVEL E CUSTOMIZÁVEL PARA REDE DE SENSORES SEM FIO

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica e aprovada em sua forma final pelo Orientador e pela Banca Examinadora.

Orientador: _____
Prof. Dr. Carlos Eduardo Pereira, Doutor pela Universidade de Stuttgart – Stuttgart, Alemanha

Banca Examinadora:

Prof. Dr. Leandro Buss Becker, UFSC
Doutor pela Universidade Federal do Rio Grande do Sul – Porto Alegre, Brasil

Prof. Dr. João César Netto, UFRGS
Doutor pela Universite Catholique de Louvain – Lovaina, Bélgica

Prof. Dr. Marcelo Götz, UFRGS
Doutor pela Universität Paderborn – Paderborn, Alemanha

Prof. Dr. Valner Brusamarello, UFRGS
Doutor pela Universidade Federal de Santa Catarina – Florianópolis, Brasil

Coordenador do PPGEE: _____
Prof. Dr. Arturo Suman Bretas

Porto Alegre, fevereiro de 2009.

AGRADECIMENTOS

Ao órgão de fomento brasileiro CNPq por financiar os meus estudos durante a realização do Mestrado.

Ao professor e orientador Carlos Eduardo Pereira pelo apoio e dedicação para com a realização deste trabalho.

Aos professores Apolinar González e Walter Mata por terem me acolhido durante a realização de estudos na cidade de Colima no México.

À empresa Texas Instruments, especialmente a Hamilton, por terem auxiliado no desenvolvimento deste trabalho com a doação de equipamentos de comunicação sem fio ao laboratório de automação.

Aos meus pais, Ricardo e Arlete, pelo carinho e incentivo prestado em todos os momentos da minha vida, fazendo de nossa família um laço de união e uma estrutura base para enfrentar os desafios da sociedade.

Ao meu irmão Renan pelo carinho e companheirismo em todos os momentos.

À minha namorada Paola por ter suportado e apoiado nos momentos difíceis.

A todos os colegas e funcionários do PPGEE-UFRGS pelo auxílio e companheirismo durante o período do mestrado.

RESUMO

Com o crescimento e o desenvolvimento de aplicações para Redes de Sensores sem Fio (RSSF), os nós-sensores passaram a realizar o tratamento de eventos mais complexos, exigindo um maior desempenho de processamento e flexibilidade do hardware. Estas novas características visam atender os requisitos de diversas aplicações, assim como, apresentarem plataformas customizáveis que possuem somente os recursos necessários para atender estes requisitos.

As RSSF, muitas vezes, necessitam de uma arquitetura flexível e que estejam aptas a adaptar-se a alterações de projeto ou do próprio ambiente em que se encontram inserido. A utilização de arquiteturas reconfiguráveis é uma solução que introduz esta flexibilidade e uma grande capacidade de processamento ao nó-sensor. Comparando com as arquiteturas ASICs (Application Specific Integrated Circuit), as arquiteturas reconfiguráveis apresentam um custo reduzido no desenvolvimento de aplicações, visto que a plataforma não fica fixa a somente uma aplicação.

A reconfigurabilidade permite um ganho no custo e tempo de projeto, além de possibilitar o desenvolvimento de plataformas genéricas para atender um número maior de aplicações. A proposta destas plataformas é, não apenas oferecer uma plataforma eficiente e flexível, mas também potencializar a aplicação em sistemas mais complexos que necessitem de uma maior capacidade de processamento.

Este trabalho apresenta o desenvolvimento de um nó-sensor reconfigurável e customizável para RSSF denominado FemtoNode. O FemtoNode possui em sua plataforma reconfigurável um processador especificado a partir de uma linguagem orientada a objetos Java e um módulo de comunicação sem fio para suportar comunicação entre os nós da rede.

A arquitetura proposta foi validada com o estudo de caso de uma rede de sensores heterogênea composta por nós-sensores com plataformas distintas, sendo a análise realizada na presente dissertação.

Palavras-chave: Rede de Sensores sem Fio, Arquitetura Reconfigurável, Sistemas Embarcados, Programação Orientada a Objetos.

ABSTRACT

With the growth and the development of new applications for Wireless Sensor Networks (WSN), sensors nodes are able to handle more complex events that require higher processing performance and hardware flexibility. These new features are intended to meet the requirements of various applications, as well to provide customized platforms that have only the resources needed to meet these requirements.

WSNs often need a flexible architecture able to adapt to design and environment changes. The use of reconfigurable architectures is an alternative to bring more flexibility and more processing capability for the sensor-node. Compared with ASICs (Application Specific Integrated Circuit) architectures, which have a high cost in production setup, reconfigurable architectures enable a reduction in these costs because its architecture is not fixed to a single application.

Reconfigurability allows a gain in the project costs and time development, and it enables the development of generic platforms to deal with a greater number of applications. Therefore, the proposal target architecture that aims to provide a flexible and efficient platform that require greater processing capacity which support the development of applications.

In this work a reconfigurable and customizable sensor-node called FemtoNode is proposed. The FemtoNode has a reconfigurable platform and a wireless module to support applications for WSNs, using an object-oriented language Java as specification language of its architecture.

The proposed concepts were validated with a case study of an heterogeneous wireless sensor composed of sensors nodes with different platforms, whose results are described in this work.

Keywords: Wireless Sensor Networks, Reconfigurable Architecture, Embedded Systems, Object-oriented Programming.

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

LISTA DE ABREVIATURAS

API	Application Programming Interface
CPU	Central Processing Unit
DSSS	Direct Sequence Spread Spectrum
FHSS	Frequency Hopping Spread Spectrum
FPGA	Field Programmable Gate Array
IEEE	Institute of Electrical and Electronics Engineers
ISM	Industrial, Scientific, Medical
ISO	International Organization for Standardization
JTAG	Joint Test Action Group -IEEE 1149.1
MAN	Metropolitan Area Network
MANET	Mobile Ad-Hoc Networks
MEMS	Micro-Electro-Mechanical Systems
OFDM	Orthogonal Frequency-Division Multiplexing
OSI	Open Systems Interconnection
PAN	Personal Area Network
PC	Personal Computer
PDA	Personal Digital Assistant
PIN	Personal Identification Number
PPM	Pulse Position Modulation
RF	Rádio-Freqüência
RFID	Radio-Frequency Identification
RFU	Reconfigurable Function Unit
RSSF	Rede de Sensores Sem Fio
RTSJ	Real Time Specification for Java
SPI	Serial Peripheral Interface
TDD	Time-Division Duplex

TDMA Time Division Multiple Access

VHDL VHSIC Hardware Description Language

VHSIC Very High Speed Integrated Circuit

WLAN Wireless Local Area Network

WPAN Wireless Personal Area Network

WSN Wireless Sensor Network

SUMÁRIO

1 INTRODUÇÃO

1.1 Contextualização do trabalho

As Redes de Sensores sem Fio (RSSF) têm evoluído com o crescimento tecnológico de equipamentos eletro-eletrônicos e com os sistemas denominados MEMS (Micro-Electro-Mechanical Systems) (CHANDRAKASAN et al., 1999). Aplicações estão surgindo em diversas áreas como: militar, médica, industrial, agropecuária e doméstica, possibilitando, também, atuações em ambientes hostis e inóspitos aos seres-humanos.

Uma RSSF pode ser caracterizada como um grande sistema distribuído conectado através da comunicação sem fio. A comunicação entre os nós-sensores é estabelecida diretamente entre nós origem e destino (single-hop) ou indiretamente através de nós intermediários por uma comunicação multi-saltos (multi-hop) (ROMER; MATTERN, 2004).

Estes sensores são colocados dentro ou próximo do fenômeno a ser observado, sendo que as posições não precisam ser pré-determinadas ou pré-calculadas. Assim, sendo as posições dos sensores aleatórias, implica em uma maior complexidade dos protocolos de comunicação que devem ter a capacidade de auto-organização e descobrimento de novas rotas de comunicação. Outra característica importante é a limitação de energia existente nos nós-sensores, necessitando de uma cooperação no transporte dos dados de uma forma eficiente e com baixo consumo de energia.

Pode-se classificar estas redes como sendo uma evolução das redes de comunicação sem fio Ad-Hoc, ou MANETs (Mobile Ad-Hoc Networks), onde não é necessária uma infra-estrutura pré-existente para prover comunicação entre os nós, pois as redes são auto-organizáveis. No entanto, as RSSF possuem alguns requisitos específicos devido ao elevado número de nós da rede, elevada taxa de falhas e limitado consumo de energia, que exigem uma arquitetura diferenciada da utilizada em redes tradicionais (AKYILDIZ et al., 2002).

1.2 Motivação

A arquitetura do nó-sensor é fundamental para desenvolver uma aplicação eficiente para uma RSSF. Deseja-se ter um módulo de processamento dedicado com o menor consumo de energia possível e com um transceptor de comunicação sem fio para realizar a disseminação da informação pela rede.

Atualmente, os nós-sensores passaram a realizar o tratamento de eventos mais complexos, exigindo um maior desempenho de processamento e flexibilidade do hardware, estando esta arquitetura apta a adaptar-se a alterações na aplicação ou do próprio ambiente em que se encontra inserida. A utilização de arquiteturas reconfiguráveis é uma solução que introduz esta flexibilidade e uma capacidade maior de processamento ao nó-sensor.

Os esforços dos fabricantes de dispositivos de lógica programável, componente principal em arquiteturas reconfiguráveis, para desenvolver dispositivos com um consumo de energia cada vez mais reduzido e com uma capacidade maior em agregar mais unidades lógicas está permitindo a utilização em aplicações de RSSF (RABAEY, 2000a). Este avanço tecnológico busca agregar e/ou substituir as arquiteturas ASICs, tradicionais em dispositivos nós-sensores, através do desenvolvimento de arquiteturas híbridas ou arquiteturas que utilizam *softcores*. Realizando uma comparação, pode-se verificar que as arquiteturas reconfiguráveis possibilitam uma redução de custo, visto que a arquitetura de hardware não fica restrita a somente uma aplicação, ao contrário das arquiteturas ASICs que, além disso, apresentam um elevado custo de setup de produção (GRAY et al., 2002).

A flexibilidade destas arquiteturas facilita a otimização da lógica sintetizada, permitindo a síntese de microcontroladores dedicados e otimizados para uma determinada aplicação. Assim, apenas os recursos necessários para a aplicação são sintetizados, resultando em um processador menor, ocupando menos área na FPGA, com melhor desempenho e com menor potência dissipada. Estas características são muito importantes para aplicações em RSSF onde a energia consumida deve ser a menor possível para não afetar a vida útil[1] do nó-sensor.

Com o aumento da complexidade das aplicações para RSSF, também há a necessidade de utilização de linguagens que abstraiam a plataforma utilizada. O aumento da abstração na descrição da aplicação e, conseqüentemente, na síntese do microcontrolador, facilita a implementação do projeto e a reutilização de código. A utilização de linguagem Assembly e C, largamente utilizadas em sistemas embarcados, tornam o código complexo e difícil de ser reaproveitado. Assim, a linguagem Java tem sido muito utilizada no desenvolvimento de sistemas embarcados por utilizar o conceito de orientação a objetos, conferindo reuso do código gerado para a aplicação. Outras características também motivaram o uso da linguagem Java (CARNAHAN, 1999), como:

- O alto nível de abstração presente em Java conduz a uma melhora na produtividade dos programadores, pois é claramente mais fácil de ser dominada do que C++.
- Java é relativamente segura, mantendo componentes de software (incluindo a própria Java Virtual Machine) protegidos entre si.
- Java suporta o carregamento dinâmico de novas classes, sendo altamente dinâmica, oferecendo suporte à criação de objetos e de threads em tempo de execução. As tecnologias ligadas à Java dão suporte a criação de aplicações distribuídas.
- Java foi projetada para dar suporte à integração de componentes e reutilização. As tecnologias ligadas à Java foram desenvolvidas com considerações criteriosas, utilizando conceitos e técnicas que foram averiguados pela comunidade de programadores.
- A linguagem de programação Java e a plataforma Java dão suporte à portabilidade de aplicações. Java oferece uma semântica de execução bem-definida, o que torna possível essa portabilidade.

Com a inclusão de uma especificação de programação para aplicações tempo-real (RTSJ - Real-time Specification for Java) (GOSLING; BOLLELLA, 2000), a linguagem

[1] Tempo de funcionamento dos nós da rede determinado pela duração da fonte de energia.

Java ganhou campo em aplicações mais críticas onde se deseja previsibilidade temporal. Os maiores desafios de Java frente à programação em tempo-real foram o escalonamento, o gerenciamento de memória e a sincronização de tarefas.

1.3 Objetivos

Este trabalho tem como objetivo o desenvolvimento de uma arquitetura reconfigurável e customizável para nós-sensores em RSSF denominada FemtoNode. O FemtoNode utiliza uma plataforma de hardware reconfigurável onde é sintetizado um microcontrolador *softcore* RT-FemtoJava (ITO; CARRO; JACOBI, 2001; WEHRMEISTER, 2005). Dado que o RT-FemtoJava é customizável, o seu código é adaptado às necessidades da aplicação, sendo que somente o hardware necessário é sintetizado, reduzindo assim a área de hardware necessária e o consumo de energia do nó-sensor. Reduzindo-se os recursos não utilizados durante a síntese da arquitetura do nó-sensor, viabiliza-se a implementação em arquiteturas reconfiguráveis com menos unidades lógicas disponíveis e contempla-se uma maior portabilidade da aplicação entre outras arquiteturas.

A descrição da aplicação, para posterior síntese do microcontrolador em uma arquitetura reconfigurável, é realizada utilizando uma linguagem de programação orientada a objetos Java. Para esta descrição, é utilizado um ambiente de desenvolvimento com uma ferramenta de síntese que gera os arquivos de descrição de hardware a partir da descrição da aplicação e do microcontrolador em Java, denominada SASHIMI (ITO; CARRO; JACOBI, 2001; SASHIMI: MANUAL DO USUÁRIO, 2006), a qual será detalhada posteriormente.

O FemtoNode possui um módulo de comunicação sem fio incorporado em sua arquitetura. Este módulo possui uma interface com o microcontrolador implementada em linguagem de descrição de hardware para comunicação com um transceptor de rádio-freqüência externo. O módulo comunica-se com o microcontrolador FemtoJava, realizando a troca de dados e possibilitando a configuração de parâmetros do transceptor. A tecnologia de comunicação sem fio escolhida para utilização no FemtoNode foi o padrão IEEE802.15.4, pois possui características direcionadas a aplicações em RSSF devido a baixa taxa de transferência de dados e consumo de energia.

Visando facilitar a utilização do módulo de comunicação sem fio implementado à arquitetura do microcontrolador por parte dos desenvolvedores da aplicação, uma API de comunicação foi disponibilizada. A API-Wireless torna transparente o meio de comunicação entre os nós-sensores e a aplicação, facilitando a configuração e a comunicação com o módulo de comunicação sem fio.

1.4 Organização do trabalho

Os capítulos que seguem descrevem e ilustram os conceitos propostos por este trabalho da seguinte maneira. No capítulo 2 são apresentados os conceitos de RSSF buscando explorar suas características e requisitos, apresentando benefícios da utilização de arquiteturas reconfiguráveis. Em seguida, no Capítulo 3, é realizado um estudo do estado da arte de arquiteturas de nós-sensores para RSSF que possuem arquiteturas reconfiguráveis em sua plataforma. No Capítulo 4 é apresentada a arquitetura de hardware proposta por este trabalho, o FemtoNode e o Capítulo 5 aborda a validação da plataforma desenvolvida. Por fim, no Capítulo 6 é apresentada a conclusão deste trabalho e são sugeridas possíveis direções para trabalhos futuros.

2 REDE DE SENSORES SEM FIO (RSSF)

As Redes de Sensores ganharam um grande destaque ao incorporarem comunicação sem fio entre os diferentes nós da rede, transformando-se em Redes de Sensores sem Fio. Esta comunicação possibilitou a utilização destas redes em novas aplicações que exigem mobilidade ou facilidade na distribuição dos nós-sensores sobre uma determinada área de monitoramento. Através da comunicação entre os nós da rede, as informações coletadas são disseminadas dentro da rede até uma estação base ou um ponto de acesso pré-determinado, como representado na Figura 1.

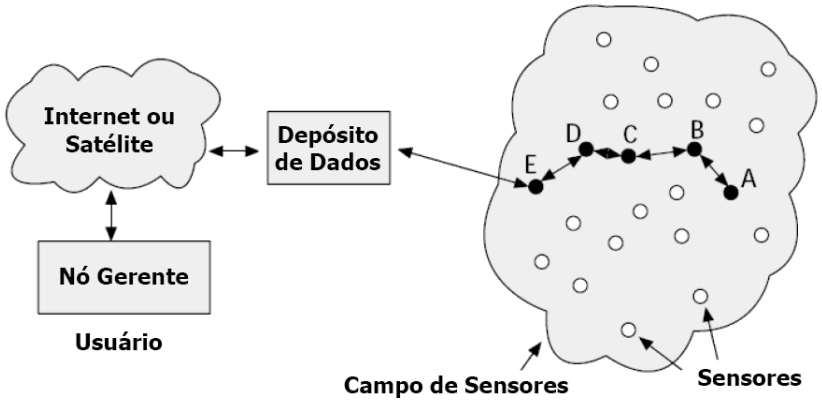


Figura 1: Rede de sensores sem fio com nós-sensores auto-organizáveis (AKYILDIZ et al., 2002).

Segundo a taxonomia descrita em (TILAK; ABU-GHAZALEH; HEINZELMAN, 2002), as RSSF fundamentam-se sobre o trinômio: sensor, observador e fenômeno. O sensor é o dispositivo que implementa a monitoração física de um fenômeno ambiental e transmite esta informação através da rede. Um sensor produz uma resposta mensurável às mudanças nas condições físicas, tais como temperatura, pressão, umidade, entre outros fenômenos. Na maioria dos modelos de dispositivos sensores, a habilidade de detecção diminui com o aumento da distância do sensor ao fenômeno e melhora com o aumento do tempo de exposição.

O observador é o usuário final interessado em obter as informações disseminadas pela rede em relação a um dado fenômeno. Ele pode indicar interesses (ou consultas) para a rede e receber respostas destas consultas. Além disso, podem existir múltiplos observadores em uma rede de sensores.

O fenômeno é a entidade de interesse do observador, que está sendo monitorada e cuja informação será analisada/filtrada pela rede de sensores. Além disso, múltiplos fenômenos podem ser observados paralelamente em uma rede.

Com base nas características do fenômeno, nas necessidades do observador e nas especificações do sensor, é possível determinar as características da rede de sensores que atingirão os requisitos necessários. Apesar destas características e requisitos serem específicos para as aplicações da rede e das limitações tecnológicas dos nós-sensores, existem diversas similaridades entre as RSSFs existentes. Pode-se destacar alguns requisitos como: tolerância a falhas, consumo de energia, escalabilidade e custo de produção (AKYILDIZ et al., 2002).

O emprego de uma RSSF em um ambiente potencialmente hostil, com uma aplicação freqüentemente crítica e em modo contínuo, eleva muito as chances de falhas. Os nós-sensores podem falhar ou tornarem-se inoperantes por falta de energia, dano físico, ou ainda, interferência. As RSSFs devem ser tolerantes a falha, empregando mecanismos para sobrepor os problemas provenientes da ausência de alguns nós como, por exemplo, o descobrimento de novas rotas de comunicação (KOUSHANFAR; POTKONJAK; SANGIOVANNI-VINCENTELL, 2002). Os níveis de tolerância a falha determinam os diferentes algoritmos de controle da rede, dependendo do ambiente e da aplicação.

Como as fontes de energia são limitadas, o gerenciamento torna-se um fator muito importante em RSSF. Assim, os nós-sensores devem cooperar entre si de modo a transportar os dados de uma forma eficiente quanto ao consumo de energia. A comunicação, principal consumidor de energia em um nó-sensor, deve ser utilizada o mínimo possível. Algumas técnicas são utilizadas para redução do consumo de energia nos nós como: formação de clusters de sensores, redução da carga computacional no sensor, protocolos eficientes, circuitos de baixo consumo, modos de baixo consumo de energia e produção local de energia.

A ordem e a grandeza do número de nós na RSSF podem variar de dezenas a centenas ou milhares dependendo da aplicação, porém a rede deve estar preparada para a remoção e a inserção de novos nós-sensores. Deve-se observar que uma elevada densidade de sensores espalhados em uma determinada área, ao mesmo tempo em que melhora as rotas de transmissão de dados, também aumenta o número de colisões nas transmissões.

Como uma RSSF é feita a partir de um grande número de nós-sensores, o custo na produção destes módulos é fundamental para viabilizar a produção e a aplicação de uma rede. Assim, o custo unitário dos nós-sensores deve-se manter muito reduzido aplicando-se soluções dedicadas ou reconfiguráveis.

Os problemas de segurança em RSSFs são similares àqueles encontrados em redes ad-hoc típicas. A segurança em sentido amplo envolve os aspectos relativos à disponibilidade, à confidencialidade, à integridade e ao não repúdio de uma mensagem enviada. No entanto, os mecanismos de proteção desenvolvidos para redes ad-hoc não são diretamente aplicáveis em RSSFs devido à complexidade e ao poder de processamento exigido da arquitetura. Assim, necessita-se do desenvolvimento de soluções específicas para RSSFs (DJENOURI; KHELLADI; BADACHE, 2005).

Apesar dos grandes desafios encontrados para se construir uma RSSF, existem muitas vantagens na sua utilização e por isso que cada vez mais este tipo de rede está sendo aplicado em diferentes áreas como: militar, médicas, ambientais, domésticas e comerciais (ALEMDAR; IBNKAHLA, 2007).

2.1 Aplicações das Redes de Sensores sem Fio

Os nós-sensores podem integrar diversos tipos de sensores como: temperatura, pressão, luminosidade, presença, posição, umidade, entre outros, realizando a coleta de dados

continuamente ou apenas detectando a realização de eventos.

Atualmente, existe um ramo de pesquisa em rede de sensores, onde os nós da rede têm a capacidade de realizar o sensoriamento ou atuar no ambiente, sendo estes denominados de nós-atuadores. Estas redes são denominadas de Redes de Sensores e Atuadores sem Fio (XIA et al., 2007).

Esta variedade de recursos que os nós podem assumir, faz com que as RSSFs estejam cada vez mais presentes em diferentes áreas (ARAMPATZIS; LYGEROS; MANESIS, 2005). Algumas áreas de atuação para as RSSFs são:

Aplicações Militares As aplicações militares foram as responsáveis pelo crescimento da pesquisa na área de RSSF. A rápida instalação, a auto-organização e a tolerância a falha é um grande atrativo para a área militar (III; TUMMALA; MCEACHEN, 2008).

As RSSF podem auxiliar no preparo das estratégias e no monitoramento das tropas amigas e inimigas (HE et al., 2004). Sistemas de mira podem integrar sensores para visualização noturna e identificação de inimigos.

Cada soldado pode ter um sensor acoplado ao seu corpo para identificar sinais vitais. Assim, a informação pode ser constantemente analisada por uma base médica que, eventualmente, poderá prestar uma assistência mais rapidamente ao soldado.

Aplicações Médicas Outra área de grande potencial para uso das RSSFs é o setor médico. No futuro, o avanço da tecnologia MEMS resultará em nós-sensores microscópicos. Com isso, poderão ser inseridos em animais e seres-humanos, permitindo a monitoração de vários aspectos vitais, como nível de oxigenação das células, insulina ou colesterol.

Os cabos ligados a pacientes serão substituídos por sensores sem fios e a monitoração dos sinais vitais poderá ser armazenada e analisada posteriormente (JAFARI et al., 2005).

Muitas aplicações para o auxílio à pessoas portadoras de deficiências físicas podem ser realizadas com o uso de RSSFs (SCHWIEBERT; GUPTA; WEINMANN, 2001).

Aplicações Ambientais Aplicações de monitoramento ambiental são excelentes campos de atuação para as RSSFs. Por exemplo, o uso de RSSF no monitoramento da migração de pássaros e movimentação de pequenos animais é apresentado em (MAINWARING et al., 2002).

O uso de sensores para observação do ambiente florestal e análise do habitat de animais, como também, prevenção de desastres ecológicos e queimadas, pode ser de grande utilidade para a humanidade (SZEWCZYK et al., 2004).

As RSSFs também podem ser utilizadas para melhorar a precisão em estudos meteorológicos (PATHAN; HONG; LEE, 2006).

Aplicações Domésticas As RSSFs também são muito utilizadas para aplicações em automação residencial, integrando ambientes inteligentes. Sensores podem ser embutidos em eletrodomésticos, criando uma rede de cooperação entre eles. Ambientes inteligentes podem ser criados com a utilização de sensores para detecção de eventos ou comandos como apresentado em (WHEELER, 2007).

Contudo, a arquitetura do nó-sensor é fundamental para determinar em que aplicação a RSSF pode ser utilizada.

2.2 Arquiteturas de hardware

Nós-sensores são dispositivos autônomos equipados com capacidades de sensoriamento, processamento e comunicação. Quando estes nós são dispostos em uma rede de um modo ad-hoc, formam as redes de sensores. Os nós coletam dados via sensores, processam localmente ou coordenadamente entre vizinhos e enviam a informação para um usuário ou, em geral, para uma estação base (base station ou data sink).

A arquitetura do nó-sensor é fundamental para desenvolver uma aplicação eficiente para uma RSSF. Deseja-se ter um módulo de processamento dedicado com o menor consumo de energia necessário e com um transceptor para realizar a disseminação da informação na rede (KHEMAPECH; DUNCAN; MILLER, 2005). Existem nós-sensores que, dadas as suas dimensões, taxa de transmissão e alcance, por exemplo, são ideais para uma aplicação e totalmente inadequados para outras. Assim, verifica-se que entre a aplicação e a rede física, deve-se ter diversos outros elementos que determinam as funções do nó-sensor, como representado na Figura 2.

Figura 2: Diferentes elementos localizados entre a aplicação e a rede física.

Geralmente, o nó-sensor é composto pelos seguintes componentes de hardware: unidade de sensoriamento, unidade de processamento, unidade de comunicação e unidade de energia. Na Figura 3 está representada a arquitetura de hardware de um nó-sensor de uma RSSF.

- *Unidade de sensoriamento:* A funcionalidade principal da unidade de sensoriamento é mensurar as grandezas físicas do fenômeno a ser observado. Os sinais analógicos gerados pelo sensor são digitalizados por um conversor analógico-digital (ADC) e destinados a unidade de processamento que realizará a análise destes valores.

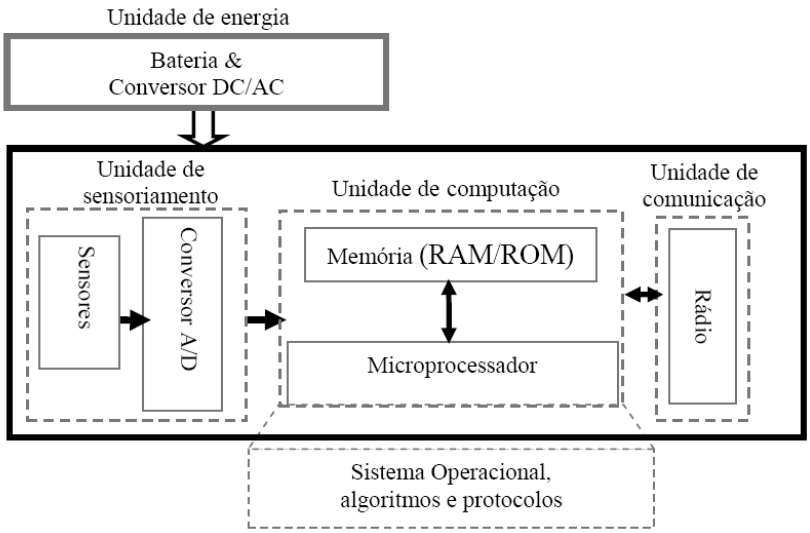


Figura 3: Principais componentes de um nó-sensor sem fio.

- *Unidade de processamento:* A unidade de processamento é o elemento principal na tarefa de colaborar com os demais nós na integração das informações. Esta unidade é responsável por analisar e armazenar as informações provenientes dos sensores e minimizar os dados a serem transmitidos na rede através de um processamento local. Atualmente, há diversos tipos de unidades de processamento: microcontroladores, microprocessadores e FPGAS. Cada unidade possui suas características como: capacidade de armazenamento de dados, velocidade de processamento, consumo de energia e módulos integrados de interfaceamento como, por exemplo, conversores ADCs.

- *Unidade de comunicação:* A unidade de comunicação é composta por um transceptor. O transceptor é a unidade responsável pela comunicação entre nós-sensores em uma RSSF. Para as RSSF, quanto menor a energia empregada para a transmissão, mais autonomia terá este nó-sensor. Em uma RSSF podem ser exploradas três possibilidades para a comunicação sem fio: comunicação ótica (lasers), infravermelho e radiofreqüência (RF). Existem diversas técnicas empregadas na comunicação para maximizar a relação sinal/ruído como: modulações, filtragem e demodulações. Em geral, os rádios podem operar em quatro modos: transmitindo, recebendo, ocioso e "dormente". Estes modos facilitam no gerenciamento de energia.

- *Unidade de energia:* A unidade de energia é responsável por armazenar a energia para o funcionamento do nó-sensor. Existem, atualmente, diversos tipos de baterias que podem ou não ser recarregáveis. Cada tipo de bateria possui suas características de tempo de duração, ciclo de recargas e tempo de recarga. A recarga das baterias pode ser realizada através de mecanismos que captam energia de algum fenômeno físico como luz solar ou vibrações mecânicas. O gerenciamento do consumo de energia é um dos maiores problemas para as RSSF pois, dependendo da aplicação, os nós-sensores deverão ter uma autonomia de alguns meses ou até alguns anos, uma vez que são instalados em regiões inóspitas e de difícil acesso.

2.3 Arquiteturas Reconfiguráveis em Rede de Sensores sem Fio

A quantidade de aplicações para Redes de Sensores sem Fio está aumentando rapidamente (PEI et al., 2008). As RSSFs estão sendo empregadas em aplicações cada vez mais complexas em diversas áreas. Porém, na grande maioria das aplicações, a arquitetura do nó-sensor é definida a partir dos requisitos da aplicação, ficando limitada a atender somente aquelas características. Com essa finalidade, arquiteturas de hardware fixo, como ASICs, são utilizadas. Isto significa que não é possível alterar as características do nó-sensor caso a aplicação necessite de novos recursos.

Entretanto, há situações em que a arquitetura do nó-sensor necessita ser alterada ou reconfigurada para atender alterações do ambiente ou novos objetivos de projeto. Estas mudanças podem variar de acordo com os requisitos da aplicação e da migração de tarefas entre nós-sensores. Pode-se considerar que uma das característica das RSSFs é a readaptação ou reconfiguração, atendendo às necessidades que surgem no ambiente a ser monitorado ou mesmo dos próprios nós quando estes não atendem às reais necessidades de monitoramento do ambiente. Neste caso, arquiteturas de hardware reconfiguráveis, como FPGAs e CPLDs, são necessárias (CALDAS et al., 2005).

Avanços na área de circuitos integrados baseados em arquiteturas de hardware reconfiguráveis, com um grande número de recursos computacional e reduzido consumo de energia, tem proporcionado a migração e direcionamento de implementações, antes utilizando componentes discretos e ASICs, para esta classe de componentes. Estes componentes combinam flexibilidade, desempenho e um melhor custo efetivo. Sua utilização proporciona a implementação de uma variedade de alternativas customizadas de circuitos lógicos, oferecendo a facilidade de se realizar atualizações (updates), melhoramentos (upgrades), correções à distância e apresentando excelente portabilidade (reuso de lógica). Estas características tornam interessantes a utilização destes componentes para implementar soluções otimizadas e eficientes para aplicações que requeiram computação intensiva.

Uma RSSF que utilize um protocolo de roteamento hierárquico, ou seja, através da formação de clusters, pode aproveitar a maior capacidade de processamento das arquiteturas reconfiguráveis (COMMURI; TADIGOTLA, 2007). Assim, os cluster-heads podem utilizar algoritmos de agregação ou compressão de dados muito mais complexos, resultando em uma minimização do tráfego de dados na rede e executando um processamento local, o que aumenta a rapidez na realização das tarefas. Como exemplo, pode-se citar situações de emergência (incêndio, inundação, entre outras) que se beneficiariam de um processamento local.

O nó-sensor pode ser equipado com interfaces de entrada/saída genéricas para possibilitar a conexão de sensores, atuadores e dispositivos de comunicação sem fio. O hardware reconfigurável encarrega-se de realizar as conexões a um módulo de processamento que irá atender as características de comunicação deste dispositivo. Por exemplo, em (AIBE; YASUNAGA, 2004) foi desenvolvido uma interface denominada de Meta-I/O, onde o hardware reconfigurável encarrega-se de adaptar-se a interface do dispositivo externo.

Muitos estudos de pesquisa revelaram que as FPGAs podem ser mais energeticamente eficientes em comparação a processadores de uso específico, devido ao alto throughput[1] alcançado nestas arquiteturas por causa do paralelismo. Nos trabalhos (ABNOUS et al., 1998) e (MENCER; MORF; FLYNN, 1998) foram realizados implementações do algoritmo de criptografia IDEA e de filtros FIR e IIR em arquiteturas reconfiguráveis e pro-

[1] Taxa de transferência de dados efetiva de um sistema.

cessadores RISC e DSP. As FPGAs alcançaram um melhor desempenho em todas as aplicações.

2.3.1 Reconfigurabilidade computacional

Um sistema computacional pode ser implementado utilizando três diferentes tipos de arquiteturas de hardware: microprocessadores de propósito geral (GPP), circuitos integrados de aplicação específica (ASICs) ou dispositivos programáveis em hardware (FPGAs) (COMPTON; HAUCK, 2002). Os sistemas implementados utilizando microprocessadores de propósito geral apresentam um menor custo e facilidade de implementação, porém, por não serem especificamente projetados para a tarefa ao qual estão sendo utilizados, muitas vezes perdem em desempenho, ou seja, capacidade de processamento mais rápido e precisão da tarefa ao qual o dispositivo se destina. Com a intenção de buscar um maior desempenho, utilizam-se circuitos integrados de aplicação específica (ASIC). Os ASICs, por serem otimizados para desenvolver tarefas específicas, apresentam um desempenho superior aos GPPs, entretanto, possuem um alto custo de produção, sendo seu uso justificado a partir da produção de um grande volume de unidades, de forma a baratear as unidades individuais deste componente.

Assim, pode-se colocar os microprocessadores de propósito geral e os ASICs em extremos opostos considerando flexibilidade e desempenho, conforme apresentado na Figura 4. Os GPPs apresentam grande flexibilidade de aplicações e baixo custo, mas perdem em eficiência. Os ASICs apresentam alto nível de eficiência, porém, alto custo de implementação e se restringe a uma aplicação específica.

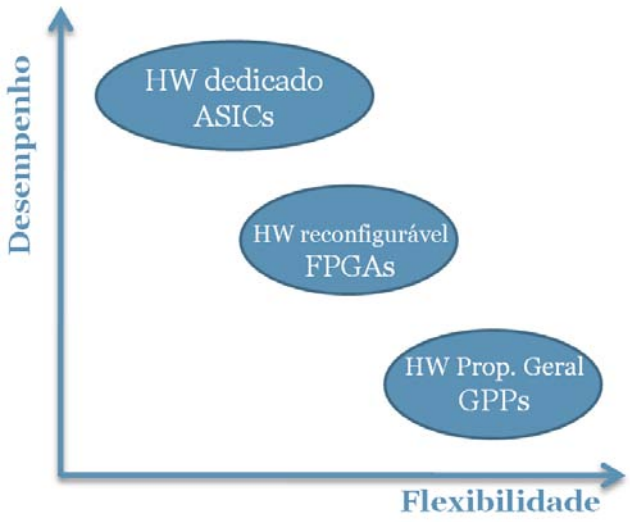


Figura 4: Comparativo entre arquiteturas de hardware.

Entre a flexibilidade da arquitetura de hardware dos GPPs e o desempenho dos ASICs pode-se destacar a presença dos dispositivos de lógica programáveis denominados de FPGAs (Field-Programmable Gate Array) e CPLDs (Complex Programmable Logic Device) em uma posição intermediária a estas características. Estes dispositivos têm a característica de agregar a flexibilidade dos processadores de propósito geral com a "customização" de hardware que é oferecida nos ASICs. Adicionalmente, apresentam o seu custo mais acessível que os ASICs, especialmente, a cada nova geração de projeto.

Os tempos de projeto de sistemas digitais têm-se tornado cada vez mais curtos, pela necessidade de lançamento de novos produtos no mercado. Ao mesmo tempo, o avanço

tecnológico vem permitindo circuitos com um número crescente de componentes. Prevendo o gargalo do tempo de projeto, a utilização de dispositivos reconfiguráveis e uma linguagem de descrição de hardware se fazem necessárias (CARRO, 2001).

Na medida em que os dispositivos de Lógica Programável tornam-se cada vez mais densos e rápidos, o processo de concepção de projetos eletrônicos se beneficia destes circuitos fazendo com que o tempo de projeto seja reduzido em muitos dias. Assim sendo, devido à grande complexidade dos projetos capazes de serem implementados em lógica programável, tais circuitos passam a ser utilizados em projetos industriais como substituto dos circuitos ASIC.

Muitas aplicações emergentes necessitam que suas funcionalidades permaneçam flexíveis mesmo depois do sistema ter sido manufaturado. Tal flexibilidade é fundamental, uma vez que requisitos dos usuários, características dos sistemas, padrões e protocolos podem mudar durante a vida do produto. Essa maleabilidade também pode prover novas abordagens de implementação voltadas para ganhos de desempenho, redução dos custos do sistema ou redução do consumo geral de energia.

A flexibilidade funcional é comumente obtida através de atualizações de software, mas desta forma a mudança é limitada somente à parte programável dos sistemas. Desenvolvimentos recentes na tecnologia de matrizes de elementos lógicos programáveis introduzem suporte para modificações rápidas em circuitos digitais via reconfiguração em tempo de projeto ou em tempo de execução, sendo esta com ou sem a interrupção da operação do circuito (GARCIA et al., 2006). Em dispositivos reconfiguráveis, como FPGAs, pode-se ter quatro tipos de reconfiguração:

- *Reconfiguração total:* É a forma de configuração onde o dispositivo reconfigurável é inteiramente alterado.

- *Reconfiguração parcial:* É a forma de configuração que permite que somente uma porção do sistema digital seja reconfigurada. Uma reconfiguração parcial pode ser não-disruptiva ou disruptiva. A reconfiguração parcial é não-disruptiva quando as porções do sistema que não estão sendo reconfigurados permanecem completamente funcionais durante o ciclo de reconfiguração e a reconfiguração parcial é disruptiva quando outras partes do sistema são afetadas, tipicamente necessitando de uma parada no sistema inteiro.

- *Reconfiguração dinâmica:* A reconfiguração dinâmica é a forma de reconfigurar o dispositivo em tempo de execução, sendo denominada como run-time reconfiguration (RTR), on-the-fly reconfiguration ou in-circuit reconfiguration. Reconfiguração dinâmica é outra forma de expressar a reconfiguração parcial não-disruptiva (GÖTZ; RETTBERG; PEREIRA, 2005). O termo implica em não haver necessidade de reiniciar o circuito ou remover elementos durante a reconfiguração.

- *Chaveamento de contexto:* É a capacidade de um dispositivo ou sistema de ser configurado em tempo de execução, durante a operação do sistema, em função de um conjunto de arquivos de configuração pré-carregados em uma memória de controle interna ao dispositivo. Pode estar associado a procedimentos de reconfiguração parcial ou total.

A reconfigurabilidade também contribui para a economia de recursos, onde uma dada tarefa pode ser realizada em várias fases, carregando-se configurações de hardware para realizar cada uma das fases seqüencialmente. Desta forma o tamanho do sistema pode

ser menor, o que implica na redução de preço, área e consumo de energia. Pode ainda ser usada como tecnologia para construção de sistemas tolerantes a falhas, através da auto-verificação e auto-reconfiguração, substituindo elementos defeituosos por elementos reserva disponíveis (MESQUITA, 2002).

Outro benefício é o aproveitamento de código através da utilização de linguagens de descrição de hardware. Com isso, projetos podem ser realizados muito mais rapidamente e atualizações podem ser realizadas em locais específicos, conservando-se e reaproveitando-se os demais componentes da arquitetura.

2.3.1.1 Hardware reconfigurável

As arquiteturas reconfiguráveis ou arquiteturas de sistemas computacionais reconfiguráveis são aquelas onde os blocos (módulos) lógicos construtivos básicos podem ser reconfigurados, na sua lógica ou funcionalidade interna, e os blocos de interconexão responsáveis pela interligação desses blocos lógicos construtivos e pela definição da estrutura da arquitetura também podem ser reconfigurados. Esses blocos lógicos construtivos normalmente implementam ou são as unidades funcionais de processamento, armazenamento, comunicação ou entrada e saída de dados (TODMAN et al., 2005).

Existem diversos tipos e classes de dispositivos programáveis (configuráveis) com capacidade de implementar funções lógicas como: EPROM (Erasable Programmable Read Only Memory), PLA (Programmable Logic Array), PAL (Programmable Array Logic), etc. Devido à necessidade de implementar funções mais complexas, surgiram os dispositivos conhecidos como CPLD (Complex Programmable Logic Devices) e outros tipos de dispositivos programáveis como o MPGA (Mask Programmable Gate Array) e o FPGA (Field Programmable Gate Array) (MARTINS et al., 2003).

Os FPGAs representam uma classe de dispositivos de lógica programável com a capacidade de reconfiguração e de poderem ser configurados diversas vezes após a sua fabricação, sendo implementados circuitos digitais, como processadores, interfaces, controladores e decodificadores. Diferentemente dos dispositivos programáveis de menor densidade tais como os CPLDs, os FPGAs apresentam uma grande densidade interna de blocos lógicos configuráveis (CLBs) que podem ser interconectados através de uma rede interna de roteamento de sinais. Atualmente, a capacidade destes dispositivos está na faixa de milhões de portas lógicas.

Os FPGAs possuem três conjuntos de elementos de configuração. O primeiro consiste de vários circuitos idênticos, compostos por alguns flip-flops e lógica combinacional extra, sendo conhecidos por CLBs (Configuration Logical Blocks). Os CLBs possuem os elementos funcionais para a construção de lógicas pelo usuário. Geralmente, são constituídos de 2 a 4 flip-flops e por lógicas combinacionais. O segundo consiste de circuitos de interfaceamento das saídas dos CLBs com o exterior do FPGA, chamados de IOBs (Input Output Blocks). Eles são constituídos por buffers bidirecionais com saída em alta-impedância. Através da programação adequada de um IOB, configura-se uma conexão do FPGA para funcionar como entrada, saída, bidirecional ou coletor-aberto. O terceiro grupo é composto pelas interconexões. Os recursos de interconexões possuem trilhas para conectar as entradas e saídas dos CLBs e IOBs para as redes apropriadas. Geralmente, a configuração é estabelecida por programação interna das células de memória estática, que determinam funções lógicas e conexões internas implementadas no FPGA entre os CLBs e os IOBs. O processo de escolha das interconexões é chamado de roteamento. Na Figura 5 está representado a arquitetura interna de uma FPGA.

O arquivo de configuração é obtido a partir da síntese de arquivos de descrição de

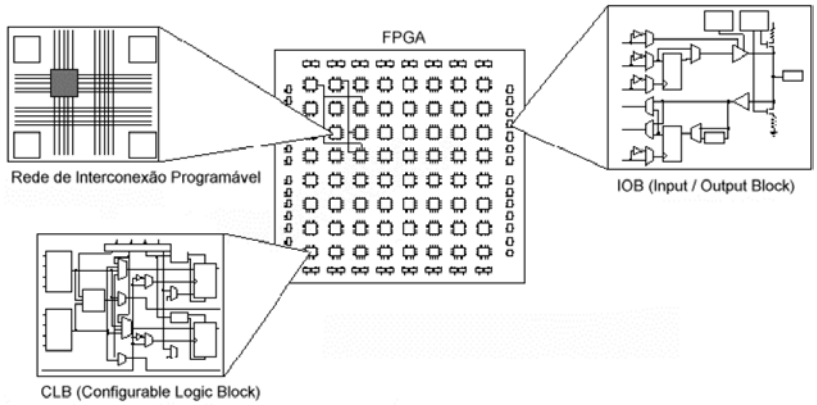


Figura 5: Arquitetura interna de uma FPGA (MARTINS et al., 2003).

hardware (HDL) ou de esquemáticos. Apesar de terem programação volátil, placas com FPGAs normalmente possuem algum tipo de memória não-volátil para que a programação seja feita de forma autônoma e rápida.

Tecnologias de FPGAs permitem uma reconfiguração dinâmica ou em tempo de execução (BURNS et al., 1997).

2.3.1.2 Linguagem de descrição de hardware

As linguagens de descrição de hardware, ou HDLs, foram desenvolvidas a partir da necessidade da criação de uma maneira simples e completa de se especificar um sistema digital. Seu principal mérito é permitir que todas as etapas de um projeto (especificação, simulação, descrição de implementação e síntese) sejam realizadas a partir de uma única notação.

Existem diversas linguagens desenvolvidas com capacidade para descrever a especificação e implementação de circuitos digitais, como por exemplo: VHDL (IEEE 1076), Verilog (IEEE 1364), ABEL (Advanced Boolean Equation Language), HardwareC, ISPS (Instruction Set Interchange Format), XNF (Xilinx Netlist Format) e AHDL (Altera Hardware Description Language). Dessas linguagens, duas tornaram-se padrão e são amplamente utilizadas atualmente: a VHDL, padrão na Europa, e a Verilog, utilizada por questões específicas do mercado dos Estados Unidos (SMITH, 1996). A popularização das HDLs deu-se principalmente pelo surgimento das ferramentas de síntese para dispositivos programáveis, que permitem a implementação de um hardware digital a partir de sua descrição em VHDL ou Verilog.

O VHDL (VHSIC Hardware Description Language) é uma linguagem de descrição de hardware voltada para a simulação de sistemas eletrônicos, sendo posteriormente, utilizada para síntese de circuitos digitais em dispositivos programáveis, como por exemplo FPGAs. Inicialmente, a linguagem VHDL foi utilizada pelo Departamento de Defesa dos Estados Unidos para descrição técnica e projeto de novos circuitos integrados, visando acelerar o projeto e auxiliar na manutenção dos equipamentos eletrônicos utilizados pelas forças armadas. Em 1987, tornou-se um padrão pela organização internacional IEEE e muito utilizada pelos projetistas.

A linguagem VHDL permite a descrição de hardware em três níveis de abstração: circuitos digitais (portas lógicas básicas), máquina de estados e programação estruturada. Esta linguagem possui as seguintes características (ASHENDEN, 1990) :

- Auxilia no intercâmbio de projetos entre grupos de pesquisa sem a necessidade de

alteração;

- Permite ao projetista considerar no seu projeto os atrasos comuns aos circuitos digitais;
- A linguagem independe da tecnologia atual, ou seja, pode ser desenvolvido um sistema hoje e implementá-lo depois;
- Os projetos são fáceis de serem modificados;
- O custo de produção de um circuito dedicado é elevado, enquanto que usando VHDL e Dispositivos Programáveis, isto passa a ser muito menor;
- Reduz consideravelmente o tempo de projeto e implementação.

2.3.2 Microcontroladores dedicados em Hardware Reconfigurável

Atualmente, plataformas híbridas, onde se tem a combinação de uma FPGA com uma CPU (ASIC) denominadas de CSoC (Configurable System on a Chip), apresentam um ganho em flexibilidade, consumo de energia e capacidade de processamento. Pode-se ter essa implementação de duas maneiras: *hardcore* e *softcore* (PLESSL et al., 2003). Um *softcore* é uma CPU sintetizada e mapeada aos recursos lógicos de uma FPGA, por exemplo, o MicroBlaze da Xilinx. Um *hardcore* é uma CPU dedicada e integrada a FPGA em sua fabricação, por exemplo, o PowerPC nas FPGAs da fabricante Xilinx.

A grande vantagem de se poder usar um processador *softcore* está no fato de que seu código pode ser adaptado às necessidades da aplicação, sendo que somente o hardware necessário é sintetizado no FPGA, reduzindo assim a quantidade de lógica necessária, resultando em um processador menor, ocupando menos área na FPGA, com melhor desempenho e com menor potência dissipada (YASUURA et al., 1998).

O aumento da abstração na descrição da aplicação e, conseqüentemente, na síntese do microcontrolador, facilita a implementação do projeto e a reutilização de código. A utilização de linguagem Assembly e C, largamente utilizadas em sistemas embarcados, tornam o código complexo e difícil de ser reaproveitado. Assim, a linguagem Java tem sido muito utilizada. Projetistas de sistemas embarcados têm adotado linguagem Java em função da independência do hardware hospedeiro oferecida por essa tecnologia, conferindo alto grau de portabilidade às aplicações. Portanto, portabilidade e reuso de código, através da orientação a objetos da linguagem Java, podem proteger os investimentos prévios em desenvolvimento de software (TAN et al., 2006).

O ambiente SASHIMI foi desenvolvido na perspectiva de sintetizar um microcontrolador dedicado (*softcore*) a partir de uma descrição em linguagem de alto nível.

2.3.2.1 RT-FemtoJava

O FemtoJava (ITO; CARRO; JACOBI, 2001) é um microcontrolador de pilha que executa instruções Java nativamente. Ele caracteriza-se por ser um *softcore*, ou seja, um microcontrolador descrito em linguagem de hardware VHDL, podendo ser sintetizado em arquiteturas reconfiguráveis. O RT-FemtoJava apresenta, em sua arquitetura, memórias RAM e ROM integradas, portas de entrada e saída mapeadas em memória e um mecanismo de tratamento de interrupções com dois níveis de prioridade. Na memória ROM, ou memória de programa, estão localizadas as instruções do programa Java desenvolvido

separado método a método, sendo o restante da memória destinado aos códigos de tratamento de interrupções. Na memória RAM, ou memória de dados, estão localizados os registradores do sistema de E/S, de interrupção e dos contadores de eventos.

A arquitetura da unidade de processamento implementa um subconjunto de instruções da Máquina Virtual Java, e seu funcionamento é consistente com a especificação da linguagem Java. O microcontrolador executa Java bytecodes nativamente, ou seja, não há nenhuma máquina virtual interpretando o código para uma determinada arquitetura, o que diminuiria o desempenho do sistema. A utilização de registradores para armazenar elementos da pilha possibilita ainda um ganho de desempenho, redução na área ocupada em FPGA e o processamento realizado simultaneamente aos acessos à memória. A arquitetura do microcontrolador está representada na Figura 6.

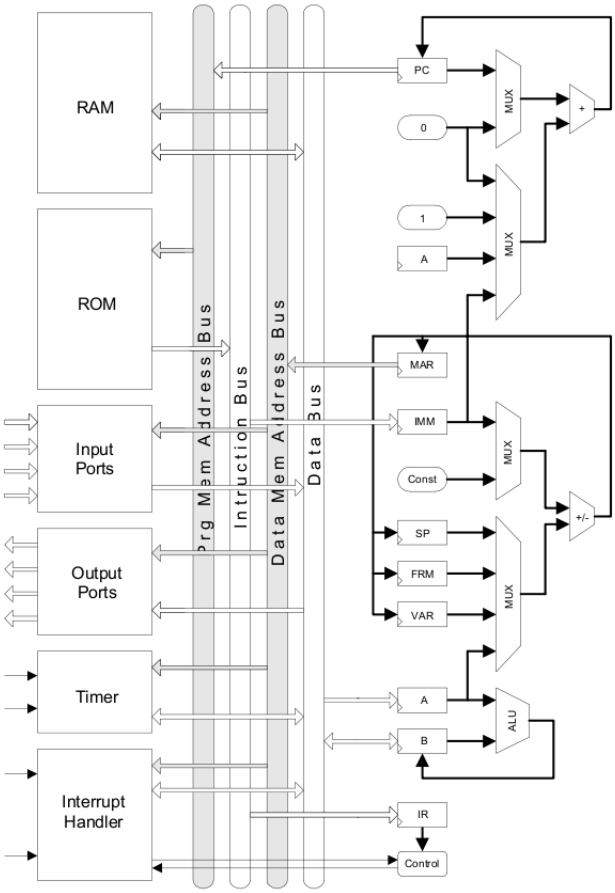


Figura 6: Arquitetura do microcontrolador FemtoJava (ITO; CARRO; JACOBI, 2001).

Atualmente existem duas versões do microcontrolador FemtoJava: Multiciclo e Pipeline (BECK; CARRO, 2004). Para o desenvolvimento deste trabalho foi escolhida a arquitetura multiciclo por apresentar previsibilidade na execução de aplicações, pois implementa alguns mecanismos para programação tempo-real.

A aplicação e a especificação completa do sistema contendo o microcontrolador são descritas em linguagem de programação Java, o que possibilita uma maior abstração do hardware no desenvolvimento da aplicação com a utilização de uma linguagem orientada a objetos voltada à portabilidade do código gerado. Após a descrição da aplicação, utiliza-se um compilador padrão Java para compilação e geração do arquivo *.class*, onde se encontram os bytecodes Java. A partir da aplicação compilada, inicia-se o processo de geração do microcontrolador FemtoJava utilizando-se a ferramenta SASHIMI.

SASHIMI “Systems As Software and Hardware In MIcrocontrollers” (ITO; CARRO; JACOBI, 2001; SASHIMI: MANUAL DO USUÁRIO, 2006) é uma ferramenta de pós-compilação que se destina à geração do microcontrolador FemtoJava a partir da especificação em Java, realizando a extração automática do subconjunto de instruções Java necessário para implementar o software da aplicação. O microcontrolador é gerado de forma particular para cada aplicação desenvolvida, possuindo apenas os requisitos necessários para atender a especificação, resultando em um melhor aproveitamento dos recursos e a obtenção de um microcontrolador otimizado. A Figura 7 ilustra o fluxo de projeto definido para o SASHIMI, partindo do código fonte da aplicação até a síntese do microcontrolador dedicado descrito em VHDL. É possível realizar a simulação e validação do código ainda antes de utilizar a ferramenta SASHIMI e, também, após a geração dos arquivos VHDL que descrevem a arquitetura do FemtoJava.

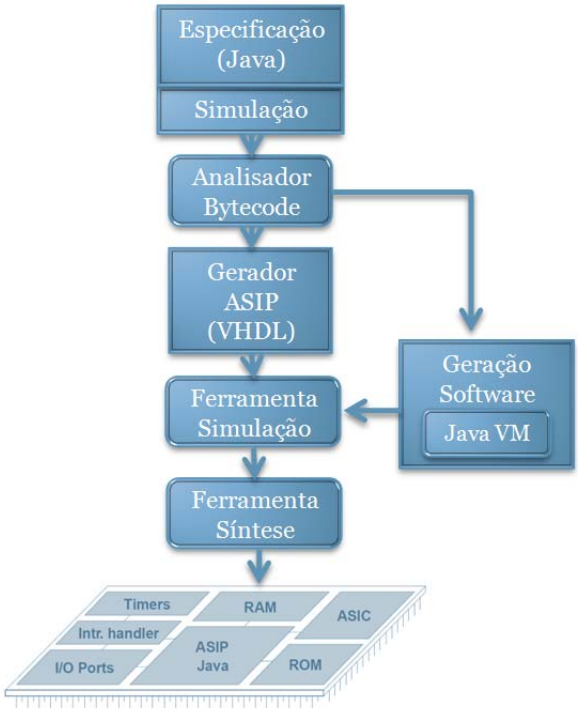


Figura 7: Fluxo de projeto da ferramenta SASHIMI (SASHIMI: MANUAL DO USUÁRIO, 2006).

Para a correta geração do microcontrolador FemtoJava a partir da ferramenta SASHIMI é necessário observar a sintaxe de programação definida pela ferramenta. No arquivo Java gerado, deve estar presente um método chamado `initSystem()` na classe a ser sintetizada, e por conveniência, define-se que a classe principal da aplicação contenha este método e o método `main()`. Outros métodos apropriados são necessários caso utilize-se interfaces de entrada/saída, temporizadores ou interrupções no microcontrolador. A sintaxe deve seguir o exemplo representado na Figura 8.

```
1  import saito.sashimi.*;
2
3  public class Exemplo implements IOInterface, TimerInterface, IntrInterface
4  {
5
6    /**********   Variaveis para Simulacao  **********/
7
8    /**********   Variaveis para Sistema   ***********/
9
10 /***********   Metodos para Sintese   ***********/
11
12       public static void initSystem(){
13       }
14
15       public static void idleTask(){
16       }
17
18       public void tf0Method(){
19   // timer 0
20       }
21
22       public void tf1Method(){
23   // timer 1
24       }
25
26       public void int0Method(){
27   // interrupcao 0
28       }
29
30       public void int1Method(){
31   // interrupcao 1
32       }
33
34       public void spiMethod(){
35   // porta serial
36       }
37
38       public synchronized int read(int channel){
39   // metodo para simulacao I/O
40        return 0;
41       }
42
43       public synchronized void write(int value, int channel){
44   // metodo para simulacao I/O
45       }
46
47       public static void main(String argv[])
48       {
49   // interfaces
50        Exemplo femtoJava = new Exemplo();
51        FemtoJavaIO.setIOClass(femtoJava);
52        FemtoJavaTimer.setTimerClass(femtoJava);
53        FemtoJavaInterruptSystem.setInterruptClass(femtoJava);
54
55   // inicio sintese sashimi
56        initSystem();
57       }
58 }
```

Figura 8: Sintaxe do código para realização da síntese pela ferramenta SASHIMI.

Em (WEHRMEISTER, 2005; WEHRMEISTER; PEREIRA; BECKER, 2006) foi desenvolvida uma API para incorporar ao ambiente SASHIMI o padrão RTSJ (Real Time Specification for Java), originando o RT-FemtoJava. A RTSJ (GOSLING; BOLLELLA, 2000) foi uma proposta de extensão da linguagem Java, desenvolvida pela Sun Microsystems para dar suporte a aplicações com requisitos temporais, tornando-a uma plataforma

aplicável a qualquer sistema embarcado, independente de suas características e quantidade de recursos.

Assim, com a API RTSJ desenvolvida para a ferramenta SASHIMI, as aplicações receberam suporte a tarefas concorrentes e foram permitidas especificações de restrições temporais. A estrutura de escalonamento desenvolvida consiste em uma tarefa adicional encarregada de alocar o processador para aquelas tarefas da aplicação que estão prontas para execução, com base na política de escalonamento implementada. Essa abordagem é a mesma utilizada em sistemas operacionais de tempo-real. O FemtoJava suporta até oito tarefas estáticas (não há criação de tarefas em tempo de execução), periódicas e independentes. Assim, possibilita-se o desenvolvimento de sistemas de tempo-real embarcados otimizados.

2.4 Comunicação sem fio

A comunicação sem fio representou um avanço tecnológico fundamental para o crescimento de aplicações com rede de sensores e para a criação das RSSF, possibilitando mobilidade e uma maior flexibilidade na disposição dos dispositivos nós-sensores na rede, tornando-os adaptados a diversas aplicações. O desenvolvimento e pesquisa de novas tecnologias visam uma maior taxa de transferência de dados entre dispositivos, redução do consumo de energia, alcance da comunicação e segurança na troca de dados na rede.

Em RSSF é mais conveniente a utilização de duas tecnologias de comunicação sem fio: radiofreqüência e ótica (laser e infravermelho).

2.4.1 Radiofreqüência

A comunicação através de radiofreqüência (RF) é a tecnologia mais empregada em RSSF devido à mobilidade necessária entre os nós-sensores. A RF permite uma comunicação de fácil programação e que não exige linha de visada entre os nós, permitindo a transposição de obstáculos. Módulos de RF são facilmente encontrados em circuitos integrados dedicados.

Estudos demonstraram que o consumo de energia para executar 1000 instruções em um processador é equivalente ao consumo para transmissão de um bit de informação a cerca de cem metros de distância (LEVIS; CULLER, 2002). Assim, para evitar o consumo excessivo de energia, os transceptores são ligados apenas quando é necessário enviar ou receber dados, ficando em modo de baixo consumo nos demais momentos.

As interferências e ruídos gerados por outros equipamentos ou pelos próprios nós-sensores da rede ou por reflexões do sinal emitido, devem ser tratadas com mecanismos elaborados para minimizar o impacto para a rede de comunicação. Dentre esses mecanismos pode-se destacar a modulação e as técnicas de codificação do sinal através da transmissão em espectro espalhado (Spread Spectrum).

O Comitê 802 do IEEE (Institute of Electrical and Electronics Engineers) desenvolveu e publicou uma série de normas para redes locais (LANs) e redes metropolitanas (MANs) que foram adotadas mundialmente, inclusive pela ISO (International Organization for Standardization). Dentre elas pode-se destacar os padrões 802.11 (WLANs) e 802.15 (WPANs) que dizem respeito a redes sem fio.

Alguns dos padrões aprovados pelo IEEE e que são utilizados para a comunicação entre dispositivos em RSSF são:

- Bluetooth (IEEE802.15.1)

- UWB (IEEE802.15.3)
- ZigBee (IEEE802.15.4)
- Wi-Fi (IEEE802.11)

Um estudo detalhado e comparativo dos protocolos de RF citados pode ser encontrado em (ALLGAYER, 2008a).

O protocolo IEEE802.15.4 é muito utilizado em aplicações destinadas a RSSF e será utilizado neste trabalho, por isso, será detalhado na seção que segue.

2.4.1.1 Protocolo IEEE802.15.4

A Zigbee Alliance (ZIGBEE ALLIANCE, 2008), uma associação de empresas do ramo da tecnologia, definiu em 2003 um padrão para as camadas física e enlace denominado IEEE802.15.4 e um padrão para as camadas de rede e aplicação denominado Zigbee. Estes padrões de comunicação visavam integrar dispositivos de baixo custo, baixa potência e confiáveis para redes de comunicação sem fio (IEEE802.15.4, 2003; BARONTI et al., 2007). A arquitetura de comunicação dos protocolos IEEE802.15.4 e Zigbee estão representados na Figura 9.

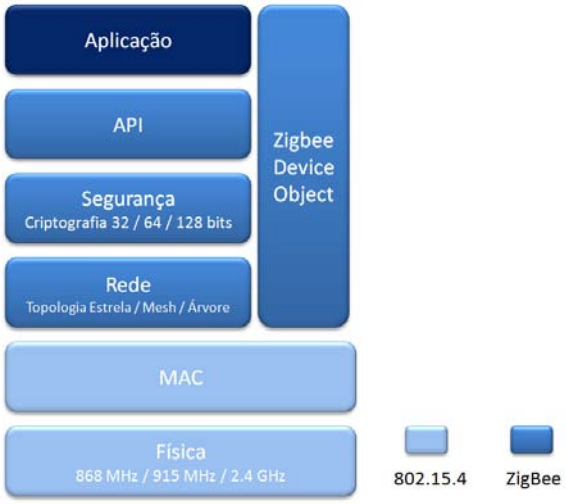


Figura 9: Arquitetura dos protocolos IEEE802.15.4 e Zigbee (ZIGBEE ALLIANCE, 2008).

A camada física definida no padrão IEEE802.15.4 pode utilizar três bandas de freqüência: 2450 MHz (16 canais), 915 MHz (10 canais) e 868 MHZ (1 canal), sendo as duas últimas utilizadas somente na Europa e na América, respectivamente. Todas as bandas utilizam o método de transmissão denominado DSSS (Direct Sequence Spread Spectrum) onde se realiza o espalhamento da energia do sinal em uma banda de transmissão mais larga em canais de 22 MHz. Esta técnica reduz o consumo de energia devido à baixa potência empregada na transmissão dos dados. A taxa de transmissão destes dispositivos pode ser visualizada na Tabela 1, onde se observa uma maior taxa de transmissão para os dispositivos que operam na banda de freqüências de 2450 MHz, pois é utilizada uma modulação O-QPSK onde cada símbolo transmitido representa 4 bits de dados.

O padrão IEEE802.15.4 define dois tipos de nós em uma rede: FFDs (Full Function Devices) e os RFDs (Reduced Function Devices). Os nós FFDs são dispositivos mais

Tabela 1: Especificação da camada física do IEEE802.15.4.

	2450 MHz	**915 MHz**	**868 MHz**
Taxa	250 Kbps	40 Kbps	20 Kbps
No. canais	16	10	1
Modulação	O-QPSK	BPSK	BPSK
Bit/símbolo	4	1	1

complexos e necessitam de um hardware com maiores recursos para a implantação da pilha de protocolos, conseqüentemente, consomem mais energia. Eles podem assumir a função de coordenador da rede ou de nó end-device (dispositivo final), localizando-se no final de um ramo da rede. Os nós RFDs podem somente atuar como nós end-devices, geralmente equipados com sensores e atuadores, podendo somente se comunicar com um nó FFD.

As redes podem assumir duas topologias: estrela ou peer-to-peer, como representado na Figura 10. Outras topologias devem ser construídas nas camadas superiores do protocolo, que não são definidas pelo IEEE802.15.4. Nas redes com topologia estrela é adotada uma configuração mestre-escravo, onde os dispositivos escravos são os end-devices (RFDs) e se comunicam apenas com o dispositivo mestre (FFD) denominado coordenador da rede PAN (Personal Area Network). Nas redes com topologia peer-to-peer existem mais de um dispositivo FFD que podem realizar comunicação entre si, disseminando a informação na rede e encaminhando para seus end-devices que são dispositivos RFDs. Nesta topologia de rede existe apenas um coordenador PAN que é designado inicialmente.

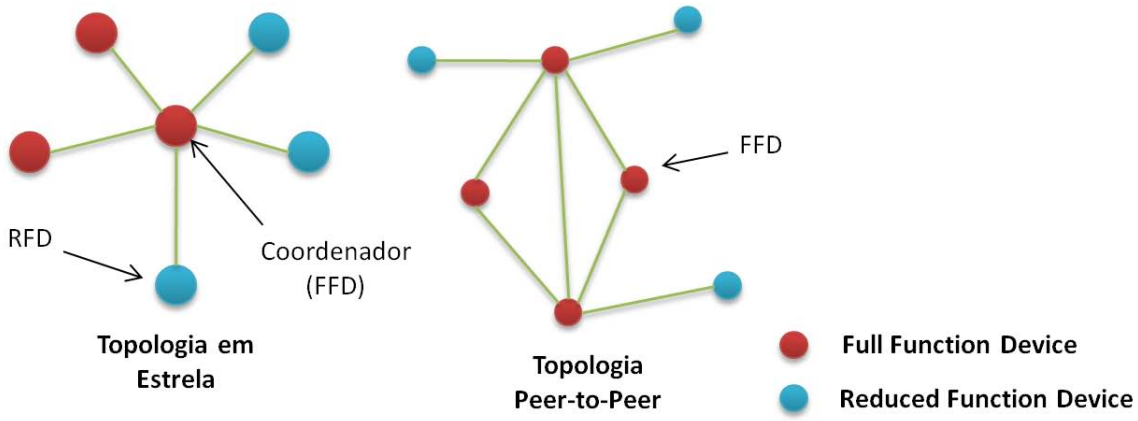


Figura 10: Topologia das redes IEEE802.15.4 (IEEE802.15.4, 2003).

A camada MAC provê mecanismos de acesso ao meio e é responsável pela comunicação ponto a ponto entre os nós. Esta comunicação é gerenciada pelo coordenador PAN e pode ocorrer utilizando ou não sinalizadores (beacons). Em redes com sinalizadores, o tempo do canal é dividido em superframes, que são inicializados por transmissões de sinalizadores pelo coordenador. O superframe é dividido em uma parte ativa e uma inativa. Toda a comunicação é realizada na parte ativa, e durante a parte inativa, os nós podem entrar no modo de economia de energia ou realizar outras atividades. A parte ativa do superframe é dividida em uma parte denominada CAP (Contention Access Period), onde os dispositivos disputam o canal utilizando o protocolo CSMA-CA com slots e outra parte

denominada CFP (Contention Free Period), onde os dispositivos não disputam o canal e possuem slots de tempo reservado pelo coordenador PAN da rede. Toda a parte ativa é dividida em slots de mesmo tamanho, sendo o sinalizador transmitido no slot zero. Este mecanismo garante a realização de comunicações com baixa latência.

Em redes sem a utilização de sinalizadores, os dispositivos disputam o canal de comunicação com o coordenador PAN através do mecanismo CSMA-CA sem slots de tempo reservados. No algoritmo CSMA-CA, o nó que pretende transmitir verifica se o canal está livre. Caso afirmativo, o nó inicia a transmissão, porém, caso o canal não esteja livre, ele aguarda um tempo aleatório para tentar novamente. O nó irá procurar pelo canal livre por até cinco vezes, e depois disso irá abortar e reportar falha para as camadas superiores. Esse mecanismo permite a coexistência de redes IEEE802.15.4 com outros protocolos para redes de comunicação sem fio (HOWITT; GUTIERREZ, 2003).

A segurança dos dados na transmissão de dados no padrão IEEE802.15.4 são realizados na camada MAC. Nesta camada é utilizado o padrão AES (Advanced Encryption Standard ou Padrão de Criptografia Avançado) como algoritmo de criptografia, descrevendo uma variedade de rotinas de segurança (CALLAWAY et al., 2002). Estas rotinas têm como objetivo promover a confidencialidade, a integridade e a autenticidade dos frames da camada MAC, utilizando chaves com comprimento de 128, 192 ou 256 bits em blocos de 128 bits.

No protocolo IEEE802.15.4 as camadas localizadas acima das camadas física e enlace podem ser especificadas pela aplicação ou utilizar as camadas de rede e aplicação denominadas ZigBee.

2.4.2 Ótica

A comunicação ótica consiste em um transmissor e um receptor que efetuam a transmissão dos dados através de uma codificação com sinais óticos. Atualmente, existem duas tecnologias que são muito utilizadas: o laser e o infravermelho.

O Infravermelho (IR) é uma radiação eletromagnética cujo comprimento de onda é maior do que o da luz visível, e por conseqüência não é visível ao olho humano. Os comprimentos de onda utilizados são da ordem de 780 nm a 1 mm (KAHN; BARRY, 1997).

As comunicações realizadas com IR são padronizadas pela IrDA (Infrared Data Association) (INFRARED DATA ASSOCIATION, 2008), uma instituição não-lucrativa criada em 1994. Esta associação é responsável por padronizar e certificar os dispositivos que utilizam a tecnologia de comunicação através de infravermelho. O padrão define a comunicação através de pulsos de infravermelho onde o dispositivo receptor deve estar dentro de um cone de +-15^{o} a partir do centro e a uma distância máxima de 1 metro. A comunicação é feita em half-duplex visto que enquanto um dispositivo está enviando dados ele não pode receber e vice-versa pelo fato de o transmissor de um dispositivo ofuscar o seu próprio receptor. A taxa de transmissão varia de 2,4 Kbps a 16 Mbps.

Existem duas topologias de redes infravermelhas: redes em linha de visada (transmissor e receptor alinhados) e redes dispersas (transmissão a partir da reflexão dos sinais em obstáculos). Ao contrário de outras tecnologias de comunicação sem fio, a comunicação infravermelho pode sofrer efeitos provocados pela luz solar devido à emissão de raios infravermelhos pelo sol.

O Laser (Light Amplification by Stimulated Emission of Radiation) é uma radiação eletromagnética com características muito especiais: ela é monocromática (possui freqüência muito bem definida) e coerente (possui relações de fase bem definidas). Ope-

ram na faixa de freqüência de 100 THz e utilizam comunicação ponto-a-ponto com linha de visada.

A comunicação a laser é muito semelhante com o infravermelho e, devido à potência mais elevada, podem alcançar distâncias maiores, sendo da ordem de 400 metros. Porém, estão sujeitos a interferências climáticas que podem afetar a transmissão.

2.5 Acesso ao meio (MAC)

O controle do acesso ao meio (MAC - Medium Access Control) é fundamental em RSSFs e tem como objetivo minimizar o gasto de energia, reduzindo o tempo de comunicação e evitando a ocorrência de colisões nas transmissões (DEMIRKOL; ERSOY; ALAGOZ, 2006). Em um nó-sensor o componente de hardware que mais consome energia é o responsável pela comunicação dos dados entre os dispositivos, assim, é necessário que este permaneça a maior parte do tempo ocioso, evitando retransmissões e reduzindo o tamanho dos pacotes de dados. Mecanismos eficientes de controle de acesso ao meio físico, compartilhado por diversos nós-sensores em uma RSSF, são importantes para aumentar a vida útil da rede e para o gerenciamento de energia dos dispositivos.

As principais fontes de desperdício de energia de um protocolo MAC para RSSF são:

- *Escuta sem tráfego:* o nó-sensor não sabe quando irá receber uma mensagem e necessita deixar o rádio ligado em espera.
- *Colisões:* dois nós-sensores transmitem ao mesmo tempo gerando o corrompimento destes dados devido à interferência.
- *Escuta desnecessária:* nó-sensor recebe pacotes destinados a outros nós.
- *Flutuações de tráfego:* quando há uma concentração elevada de nós-sensores em um determinado local e detectam um determinado evento ao mesmo tempo, assim, competindo o canal para transmitir o mesmo evento para a rede.

A camada de acesso ao meio deve priorizar o consumo de energia porém, existem outros parâmetros como latência e vazão que devem ser negociados em detrimento da economia de energia.

Ao contrário das tradicionais redes sem fio, as RSSFs são desenvolvidas para um propósito específico, com características de tráfego altamente dependentes da aplicação. Com isto, não pode ser utilizado protocolos MAC das redes sem fio tradicionais (AKYILDIZ et al., 2002). Como as aplicações têm características diferentes como a forma de coleta de dados, fenômeno observado ou modo de comunicação entre os nós, não existe um protocolo de acesso ao meio que seja adequado a todas as aplicações.

No projeto de protocolos MAC, deve-se observar algumas questões importantes, como: tipos de alocações e quantidade de canais, grau de organização dos nós e tipo de notificação recebida pelos nós (HALKES; DAM; LANGENDOEN, 2005). Podem-se acrescentar outras características relevantes como a dinâmica do ambiente e requisitos de QoS. Estes protocolos são divididos entre os que propõem a alocação estática do canal, ou seja, divisão do tempo em intervalos (slots) e os protocolos que propõem a alocação dinâmica do canal, onde os nós disputam o meio para transmissão dos dados. Os protocolos que utilizam a alocação estática do canal não possuem colisões nas transmissões e podem empregar mecanismos para minimizar a latências dos pacotes enviados, porém precisam implementar técnicas de sincronização eficientes.

Alguns protocolos MAC desenvolvidos para RSSF são:

- S-MAC (Sensor-MAC)
- ARC (Adaptive Rate Control)
- T-MAC (Time-out-MAC)
- B-MAC (Berkeley MAC)
- DE-MAC (Distributed Energy aware MAC)
- TRAMA (Traffic Adaptive Multiple Access)

Um estudo detalhado e comparativo dos protocolos MAC para RSSF pode ser encontrado em (ALLGAYER, 2008a).

Atualmente, a proposta é o desenvolvimento de protocolos de acesso ao meio integrados com os protocolos de roteamento e dispositivos de comunicação, sendo denominada de Cross-layer (MELODIA; VURAN; POMPILI, 2005). Informações de distância e disposição dos nós na rede são utilizadas para determinar os parâmetros do protocolo MAC buscando melhorar, além do consumo de energia, a vazão e a latência da rede.

2.6 Roteamento da informação

O roteamento da informação em RSSFs é realizado pelos próprios nós-sensores que compõem a rede, através de mecanismos de comunicação multi-hop (saltos múltiplos). Esta característica as diferencia de redes convencionais, onde o roteador é responsável por esta tarefa. Em RSSFs a aleatoriedade da topologia da rede e a grande quantidade de nós-sensores inviabilizam tabelas de roteamento estáticas, ou até mesmo, a pequena quantidade de memória presente no hardware destes nós não comporta o armazenamento destas tabelas. O overhead que o roteamento baseado em endereços físicos traz à comunicação, muitas vezes, aumenta o consumo de energia dos dispositivos. Assim, novos protocolos de roteamento tiveram de ser desenvolvidos ou adaptados para atender aos requisitos necessários para RSSFs (AKKAYA; YOUNIS, 2005).

Um requisito importante de redes de sensores é a tolerância a falhas, ou seja, caso um dos nós da rede apresente algum problema, outros nós devem ser capazes de suprir a perda daquele nó para a rede, sendo possível garantir a comunicação e disseminação da informação mesmo na ausência de um ou mais nós.

A segurança da RSSF deve ser levada em conta pelo protocolo de roteamento de dados, pois a maioria dos nós intrusos utiliza-se das fragilidades destes protocolos para realizar invasões à rede (SU et al., 2006).

As RSSFs caracterizam-se por serem redes baseadas em dados, onde os nós-sensores transmitem informação em relação a determinados eventos ou a partir de requisições realizadas pelo observador. Técnicas de agregação de dados realizados pelos nós são muito eficientes em RSSFs e têm como objetivo reduzir o overhead e o número de pacotes na comunicação multi-hop (LUO; LIU; DAS, 2007).

O roteamento pode ser dividido em três categorias (AL-KARAKI; KAMAL, 2004):

- *roteamento plano:* todos os nós estão presentes na mesma hierarquia de rede. A atividade de roteamento é tratada de forma idêntica por todos os nós.

- *roteamento hierárquico:* são estabelecidas duas classes distintas de nós: nós fontes e nós líderes de grupo (cluster heads). Os nós fontes simplesmente coletam e enviam os dados para o nó líder de seu grupo que pode executar uma fusão/agregação destes dados antes de enviá-lo para o ponto de acesso (observador).

- *roteamento geográfico:* utiliza informações geográficas para realizar o roteamento dos dados. Estas informações costumam incluir a localização dos nós vizinhos. Estes protocolos utilizam alguma técnica de localização como um sistema de posicionamento de coordenadas (GPS - Global Position System), ou mesmo um sistema local válido somente para os nós da rede ou válidos somente para subconjuntos de nós vizinhos.

Pode-se citar alguns protocolos de rotamento de dados para RSSF, como:

- Flooding e Gossiping
- SPIN
- LEACH
- GEAR

Em (ALLGAYER, 2008a) pode-se encontrar um estudo detalhado e comparativo de protocolos de roteamento de dados para RSSF.

Contudo, em RSSFs a troca de informações entre as diferentes camadas tem sido cada vez mais utilizada, como já comentado anteriormente para a camada de acesso ao meio. Assim, a interação dos protocolos de roteamento com os requisitos da aplicação e com os mecanismos de acesso ao meio visa otimizar o roteamento e melhorar o desempenho da rede.

3 ANÁLISE DO ESTADO DA ARTE

Neste capítulo será apresentada uma análise do estado da arte no que diz respeito a arquiteturas reconfiguráveis para aplicações em RSSF. A maioria dos trabalhos destacam a eficiência destas arquiteturas na construção de nós-sensores flexíveis, econômicos e com uma grande capacidade de processamento.

A seguir serão apresentados alguns trabalhos relevantes nesta área, destacando pontos interessantes de pesquisa e que foram utilizados para o desenvolvimento deste trabalho.

3.1 Trabalhos relacionados

3.1.1 RANS-300

O RANS-300 (Reconfigurable Architecture for Network Sensor - 300k gates) (JÚNIOR, 2004; CALDAS et al., 2005) é um nó-sensor que incorpora recursos reconfiguráveis de hardware a fim de aprimorar e expandir o conjunto de funcionalidades e de monitoramento desempenhado pelos nós-sensores convencionais. Estes recursos possibilitam o processamento de eventos complexos e que exijam maior eficiência computacional e precisão. O nó-sensor tem a capacidade de desligar estes recursos quando estes não são necessários, minimizando assim o consumo de energia.

A arquitetura do hardware do nó-sensor foi dividida em dois blocos: bloco de alto desempenho e bloco de baixa potência. Os blocos estão representados na Figura 11. No *bloco de alto desempenho* foi incorporado uma FPGA (Spartan II-E 300k gates) conectada a uma interface de cartão de memória flash (Compact Flash device) e a uma interface externa onde podem ser adicionados novos módulos sensores. Este bloco introduz capacidade de reprogramação e reconfiguração do hardware do nó-sensor. No *bloco de baixa potência* está localizado um microcontrolador de baixo consumo Texas MSP430F149, uma interface de comunicação sem fio Chipcon CC1000 (315/433/868/915 MHz) e entrada para sensores. Este bloco é responsável por executar as funções principais do nó-sensor e gerencia a energia através do acionamento do *bloco de alto desempenho* apenas quando necessário, reduzindo o consumo de energia.

Com esta configuração, a arquitetura RANS-300 propõe tornar os nós-sensores mais flexíveis a mudanças no ambiente de monitoramento, possibilitando adaptação e tolerância a falhas permanentes através da reconfiguração. O nó-sensor pode ser reprogramado a partir de arquivos pré-armazenados na memória não-volátil ou carregar os arquivos através da interface de comunicação sem fio, conforme demonstrado na Figura 12.

Neste trabalho apresenta-se três cenários de aplicações para o RANS-300. O primeiro cenário descreve a utilização do nó-sensor com sua capacidade de reconfiguração, onde o próprio nó-sensor decide a configuração a ser utilizada através de arquivos pré-gravados

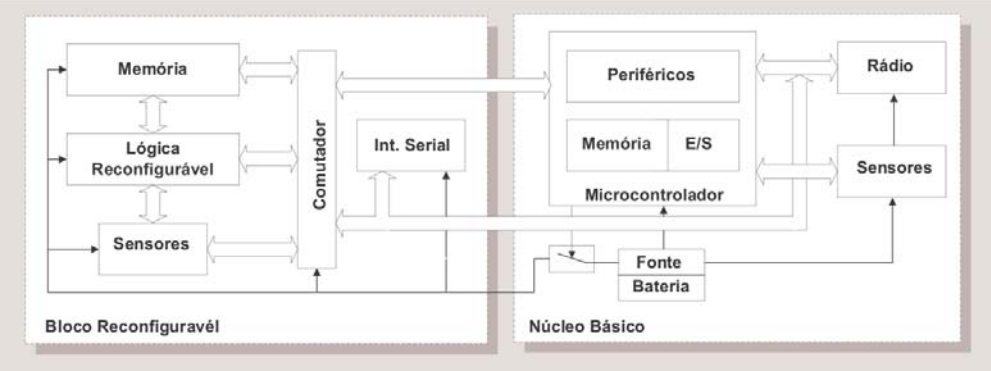


Figura 11: Diagrama de blocos do RANS-300 (CALDAS et al., 2005).

ou recebidos pela interface de comunicação sem fio. O dispositivo também tem a capacidade de verificar updates em um banco de dados da rede. O segundo cenário utiliza o nó-sensor como um armazenador de dados da rede de sensores. Neste cenário o nó-sensor utiliza a sua grande capacidade de memória, RAM e FPGA, para receber e armazenar em sua memória local os dados coletados por outros nós-sensores, transferindo-os posteriormente para o cartão de memória através da interface externa. Assim, o nó-sensor assumiria a função da estação base ou *sink node* armazenando dados temporariamente ou permanentemente. O terceiro cenário utiliza a interface externa para leitura e escrita de cartões de memória flash para interfacear outros tipos de dispositivos compatíveis com esse padrão como: transceptores Wi-Fi e Bluetooth, modems, receptores GPS, entre outros.

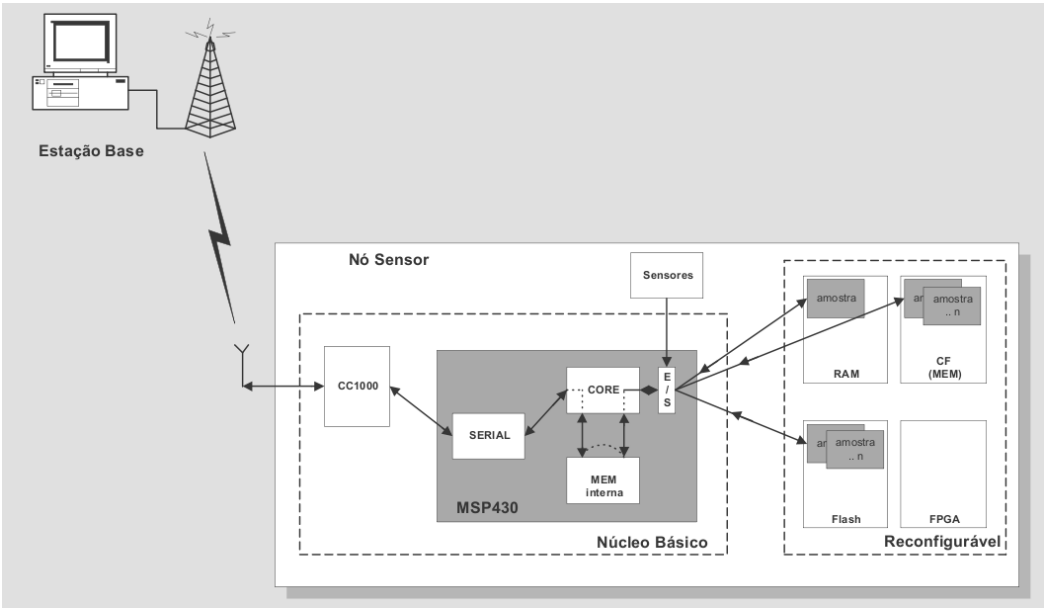


Figura 12: Reconfiguração do hardware remotamente (JÚNIOR, 2004).

Aperfeiçoamentos são propostos pelo autor para utilização de técnicas de redução do consumo de energia como modos de baixo consumo dos microcontroladores e técnicas de mudança dinâmica de tensão e freqüência como DVS (dynamic voltage scaling).

3.1.2 REWISE

O framework REWISE (Reconfigurable Wireless Intelligent Sensor Network) (WILDER et al., 2008) introduz um dispositivo de lógica programável em um nó-sensor para reconfiguração, em tempo-real, através da infra-estrutura de comunicação das RSSF. A utilização de FPGA visa permitir um balanço entre aumento de processamento, requisitos de energia e flexibilidade.

A reconfiguração dinâmica do nó-sensor é realizada com base em situações como: mudanças nos objetivos da missão da rede, necessidades de atualização ou reparo de erros, buscando adaptar-se às mudanças no ambiente. Essa reconfiguração é realizada a partir do interesse da estação base ou dos nós-sensores vizinhos. A arquitetura de reconfiguração do framework proposto pode ser denominada de "Wireless JTAG", onde as plataformas podem ser reconfiguradas simultaneamente e a distância. Isto permite a manutenção e reconfiguração de dispositivos que se encontram em áreas de difícil acesso.

Um exemplo de aplicação do framework REWISE é um sistema de vigilância militar, onde nós-sensores são dispostos próximos às linhas inimigas e monitoram a movimentação no local através da coleta de vídeo, áudio ou fontes de radiação infravermelha. Um controle central realizará decisões estratégicas baseados nestas informações coletadas. É proposto um sistema com três camadas hierárquicas diferentes: sensor, gateway e estação base, como apresentado na Figura 13. Cada camada possui um repositório de hardware/-software, evitando utilizar a comunicação sem fio para a reconfiguração, visando a economia de energia. Assim, a reconfiguração das camadas ocorre através de um comando proveniente das camadas superiores.

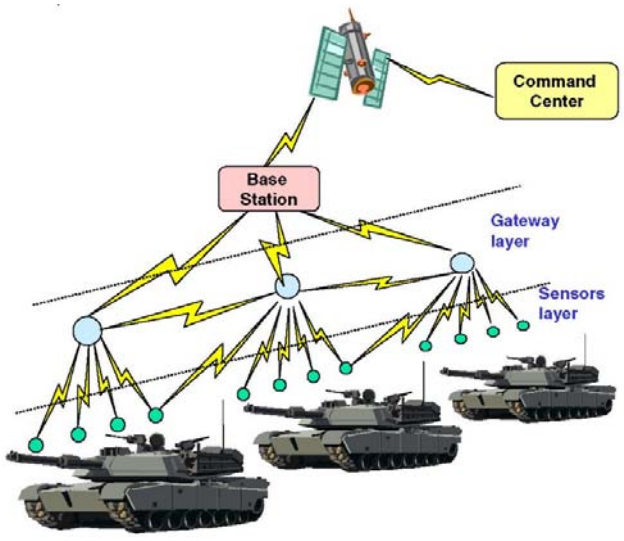


Figura 13: Cenário de aplicação do REWISE em RSSF para vigilância militar (WILDER et al., 2008).

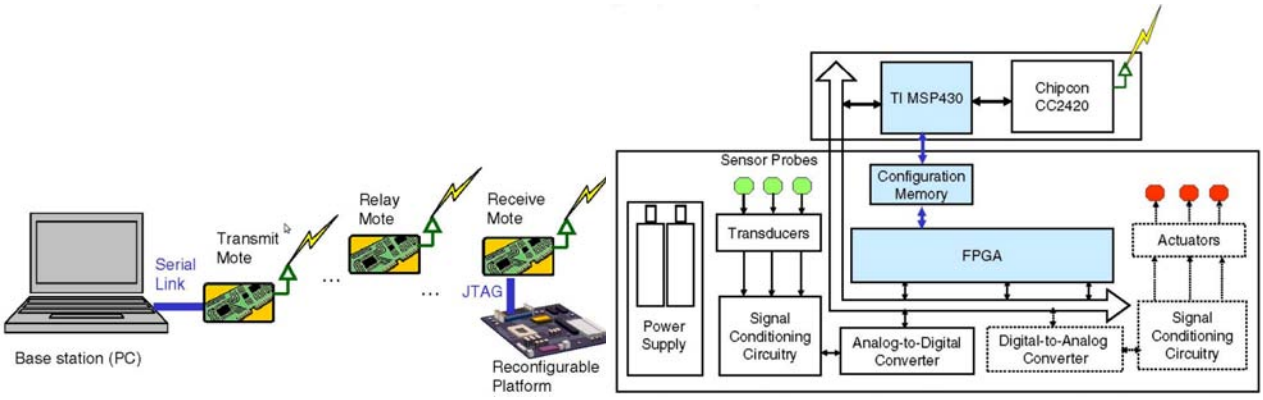
A *camada dos sensores* é responsável por coletar os dados do ambiente através dos seus nós-sensores. Estes dados podem ser pré-processados e enviados para as camadas superiores, os gateways. Os nós-sensores podem ser reconfigurados caso haja necessidade de alterar suas funções.

A *camada dos gateways* está localizada acima da camada dos sensores e possui mais capacidade de processamento, memória e energia, podendo conter algum dispositivo de lógica reconfigurável para possibilitar a reconfiguração de funções. Esta camada possui um número menor de nós denominados gateways. Cada gateway é responsável por

receber e processar os dados provenientes de alguns sensores, além de realizar a reconfiguração dos nós-sensores caso necessário.

A *camada da estação base* possui o controle centralizado da rede, tomando as decisões com base nos dados recebidos das camadas inferiores.

Os testes da arquitetura de camadas descrita anteriormente foram realizados utilizando um PC como estação base, dispositivos nós-sensores (motes) e um dispositivo reconfigurável (FPGA), como pode ser visualizado na Figura 14(a). O PC utiliza um software para gerar os arquivos para a configuração da plataforma reconfigurável no formato XSVF. Os arquivos são enviados para os nós-sensores através de uma interface de comunicação sem fio utilizando uma comunicação multi-saltos (multi-hop). Todos os nós-sensores (motes) possuem um microcontrolador Texas MSP430 e uma interface de comunicação sem fio Chipcon CC2420. Porém, um deles possui uma conexão através da JTAG de uma FPGA como demonstrado na Figura 14(b).



(a) (b)

Figura 14: Plataforma reconfigurável do framework REWISE (WILDER et al., 2008).

Simulações determinaram que a taxa de transmissão da interface de comunicação sem fio, utilizando uma porta RS-232 e o protocolo IEEE802.15.4, é de 83 Kbps. Sendo que a FPGA da família Virtex necessita de um arquivo de configuração XSVF contendo 675 Kbytes, a reconfiguração demora 65 segundos para ser transmitida e 142 segundos para realizar a reconfiguração completa do nó. Técnicas de reconfiguração parcial devem ser utilizadas para reduzir o tamanho dos arquivos XSVF e reduzir o tempo de transmissão.

3.1.3 PicoNode

O PicoNode (RABAEY et al., 2000) é um nó-sensor desenvolvido pela Universidade da Califórnia com o objetivo de ser pequeno, leve, de baixo custo e com um consumo de energia abaixo de 500μW. O nó-sensor possui um módulo de comunicação sem fio de baixo alcance e de reduzido consumo de energia denominado PicoRadio. Na Figura 15 está representado o nó-sensor PicoNode.

O módulo de comunicação PicoRadio propõe alterações no protocolo de comunicação para permitir uma redução no consumo de energia, eliminando o overhead na comunicação. O artigo analisa as camadas MAC e de rede desenvolvendo fórmulas para estimar o consumo de energia na comunicação entre os nós-sensores.

A plataforma desenvolvida para o PicoNode visa o baixo consumo de energia, implementando as alterações no protocolo de comunicação descritas anteriormente. A plataforma busca ser flexível permitindo reconfiguração dinâmica e adaptabilidade para a rede.

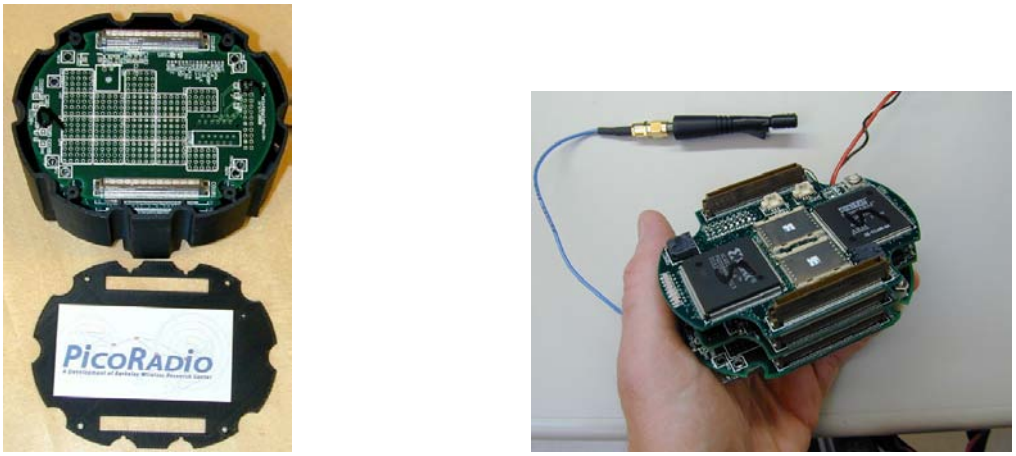


Figura 15: Ilustração do PicoNode (RABAEY et al., 2000).

A implementação de uma arquitetura reconfigurável demonstra uma redução no consumo de energia no desenvolvimento de aplicações de processamento de sinais, como demonstrado em referências anteriores do autor. A arquitetura é composta por quatro módulos: processador embarcado, arquitetura de lógica programável, camada física configurável e parametrizável e um módulo de interconexão sintetizável. A arquitetura do PicoNode está representada na Figura 16.

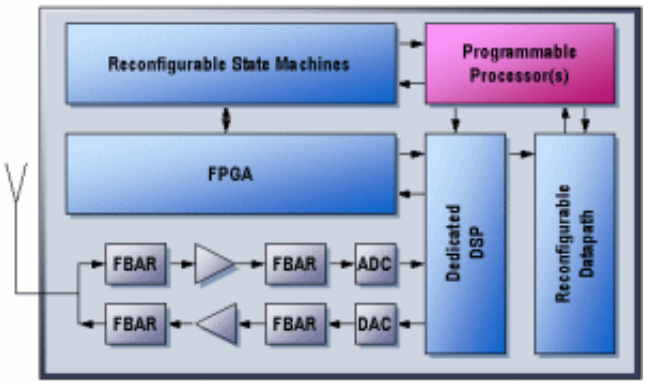


Figura 16: Arquitetura de hardware do PicoNode (RABAEY et al., 2000).

A fim de quantificar a diferença entre plataformas, uma implementação da camada de enlace em três diferentes plataformas foi desenvolvida com os resultados apresentados na Tabela 2. As simulações foram realizadas com a mesma freqüência de operação (25 MHz) e a mesma tensão de alimentação (3 Volts). Pode-se verificar que a plataforma FPGA apresentou um desempenho intermediário entre as plataformas ASIC e ARM, sendo que a plataforma ARM necessitou de 400 vezes mais potência para realizar o teste do que a plataforma ASIC.

Tabela 2: Consumo de energia em plataformas diferentes.

	ASIC	**FPGA**	**ARM**
Potência	0.26mW	2.1mW	114mW
Energia	10.2pJ/op	81.4J/op	n*457pJ/op

O trabalho conclui que a comunicação é o ponto principal a ser otimizado para realizar uma economia de energia para o desenvolvimento de um nó-sensor low-power. A utilização de arquiteturas reconfiguráveis auxilia na redução do consumo de energia com o aumento da capacidade de processamento e do throughput do nó-sensor, porém é preciso explorar a otimização do sistema cuidando para não sacrificar a flexibilidade da plataforma.

3.1.4 WURM

Sistemas embarcados em peças do vestuário possuem o desafio de combinar um alto desempenho para realizar tarefas como processamento de vídeo, com um baixo consumo de energia e com flexibilidade para atender ao dinamismo do ambiente em que atuam. Com este objetivo, uma arquitetura de hardware para um nó-sensor foi desenvolvida, denominada de WURM (Wearable Unit with Reconfigurable Modules) (PLESSL et al., 2003).

Sistemas computacionais embarcados em roupas e acessórios constituem uma área de aplicação para RSSF (YANG, 2006), onde diversos nós-sensores são distribuídos na roupa para coletar e enviar, geralmente através de uma interface de comunicação sem fio, para um módulo central, diferentes eventos como representado na Figura 17.

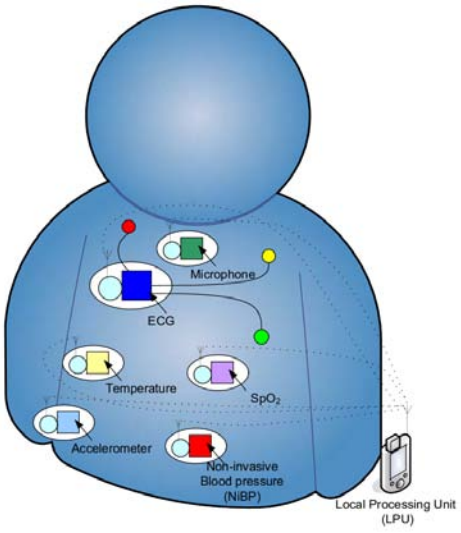


Figura 17: Sistema computacional embarcado em peças do vestuário (LO et al., 2005).

O conceito da arquitetura WURM é composto por uma arquitetura de hardware e uma arquitetura de software que compõem um nó de um sistema de computação embarcado em roupas e acessórios.

Arquitetura de hardware A arquitetura de hardware possui uma CPU, uma unidade lógica programável, memória, algumas interfaces de entrada/saída e uma interface de comunicação sem fio, como representado na Figura 18. A CPU é responsável por controlar e executar tarefas que exijam pequena e média capacidade de processamento. Por outro lado, o hardware reconfigurável é utilizado para executar tarefas que exijam uma maior capacidade de processamento e, também, realizar o interfaceamento de sensores e atuadores.

Arquitetura de software A arquitetura de software do WURM, como representado na Figura 19, é composta por um sistema operacional denominado WURM-OS que

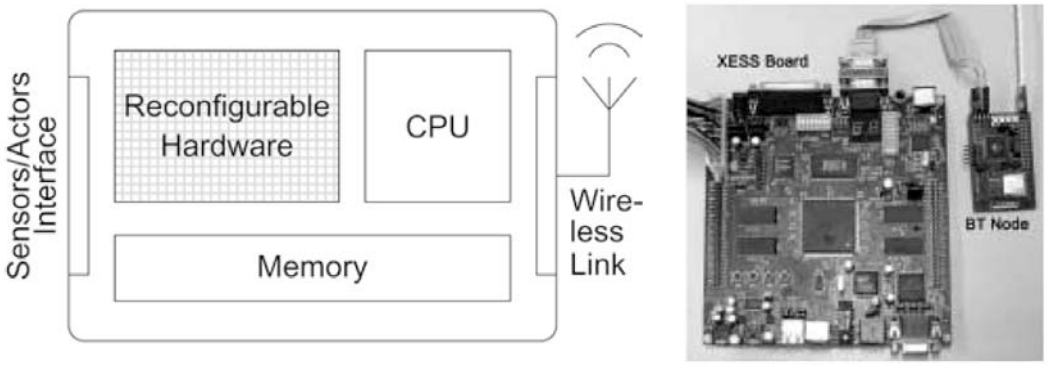


Figura 18: Arquitetura de hardware do WURM (PLESSL et al., 2003).

gerencia os recursos de hardware disponíveis como CPU, hardware reconfigurável, memória e interfaces de entrada/saída. O WURM-OS é baseado em um kernel de tempo-real para microprocessadores que fornece serviços para execução de tarefas em hardware e em software. Tarefas em software são executadas na CPU concorrentemente através de um escalonamento preemptivo. Tarefas em hardware são executadas paralelamente no hardware reconfigurável.

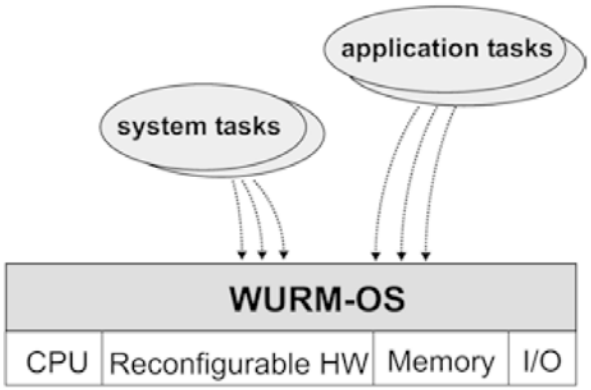


Figura 19: Arquitetura de software do WURM (PLESSL et al., 2003).

O protótipo do nó WURM foi desenvolvido em uma plataforma XESS XSV800 que contém uma FPGA Xilinx Virtex XCV800-4. Nesta plataforma foi implementada uma CPU *softcore* e uma interface de comunicação sem fio com tecnologia Bluetooth. A configuração das tarefas e do *softcore* na FPGA é realizada através da porta paralela de um PC host, que futuramente será substituído por uma CPU dedicada.

Como estudo de caso foi implementado um sistema de codificação e decodificação de áudio em diversos formatos (PCM e ADPCM). Pacotes UDP com o áudio codificado são enviados para o nó WURM através da interface Bluetooth. Tarefas em software, executadas na CPU, reconhecem o formato de codificação do áudio e iniciam a configuração de um co-processador na arquitetura reconfigurável para realizar a decodificação do áudio por uma tarefa em hardware. A CPU *softcore* utilizada foi uma 32-bits LEON CPU (GAISLER, 2004) com o sistema operacional de tempo-real denominado RTEMS (RTEMS C USER'S GUIDE, 2003).

3.1.5 mPlatform

A mPlatform (LYMBEROPOULOS; PRIYANTHA; ZHAO, 2007) é uma plataforma modular reconfigurável para rede de sensores que possibilita o processamento de tarefas tempo-real em múltiplos processadores heterogêneos. A comunicação entre os módulos do nó-sensor é realizada através de um barramento de interconexão dos diferentes módulos, permitindo o compartilhamento dos dados entre estes módulos. A arquitetura de comunicação é implementada em um dispositivo de lógica programável CPLD, desacoplando o processador das tarefas de comunicação.

Um dos principais desafios na concepção de um nó-sensor através de arquiteturas de hardware modulares é o modo de realizar o compartilhamento de dados, entre os diferentes módulos, sem comprometer as restrições de tempos das tarefas. A mPlatform utiliza uma arquitetura reconfigurável para realizar a comunicação dos dados entre os módulos, permitindo flexibilidade e eficiência para atender os seguintes requisitos:

- *Eficiência:* o processador de comunicação deve ajustar os parâmetros para permitir que o canal de comunicação opere à máxima velocidade possível.
- *Independência:* o canal de comunicação deve prover uma comunicação transparente entre os processadores, evitando uma configuração dos processadores para este propósito.
- *Escalabilidade:* os atrasos na comunicação devem permanecer previsíveis mesmo com a adição de novos módulos. O barramento de comunicação deve estar acessível aos módulos em qualquer momento, evitando esperas na comunicação.
- *Reconfigurabilidade:* o canal de comunicação deve permitir a reconfiguração e o ajuste por parte do usuário sem que haja a substituição de hardware.

Estes requisitos são satisfeitos através da utilização de uma CPLD como processador responsável pela comunicação, permitindo que o processamento dos dados seja realizado independentemente da comunicação dos dados. Para que não haja colisão nas transmissões de dados entre módulos, utiliza-se um protocolo baseado no TDMA.

A mPlatform permite a adição e remoção dos módulos de um modo plug-and-play, onde os requisitos da aplicação determinam os módulos a serem adicionados, conforme representado na Figura 20. Por exemplo, uma aplicação que requer o monitoramento de um ambiente com baixa taxa de amostragem necessita apenas de um microcontrolador de 8bits, por exemplo um MSP430, com uma interface de comunicação sem fio. Porém, uma aplicação mais complexa, como o monitoramente de sinais fisiológicos do corpo humano, pode necessitar de uma taxa de amostragem muito maior em caso de emergência, assim, um processador com uma capacidade maior de processamento é necessária, por exemplo um ARM7. Cada módulo possui uma arquitetura de hardware (módulos de processamento, comunicação sem fio, sensores e baterias) e a composição dos mesmos permite atingir os requisitos de diversas aplicações. Como as aplicações possuem restrição no consumo de energia, os módulos possuem a capacidade de desligar alguns componentes ou diminuir a voltagem ou a freqüência de operação.

A estrutura de software do mPlatform possui múltiplos processadores e utiliza a alocação e o escalonamento das tarefas nos diferentes processadores dos módulos. Os módulos têm a capacidade de migrar tarefas dependendo da energia disponível dos módulos ou da disponibilidade de recursos.

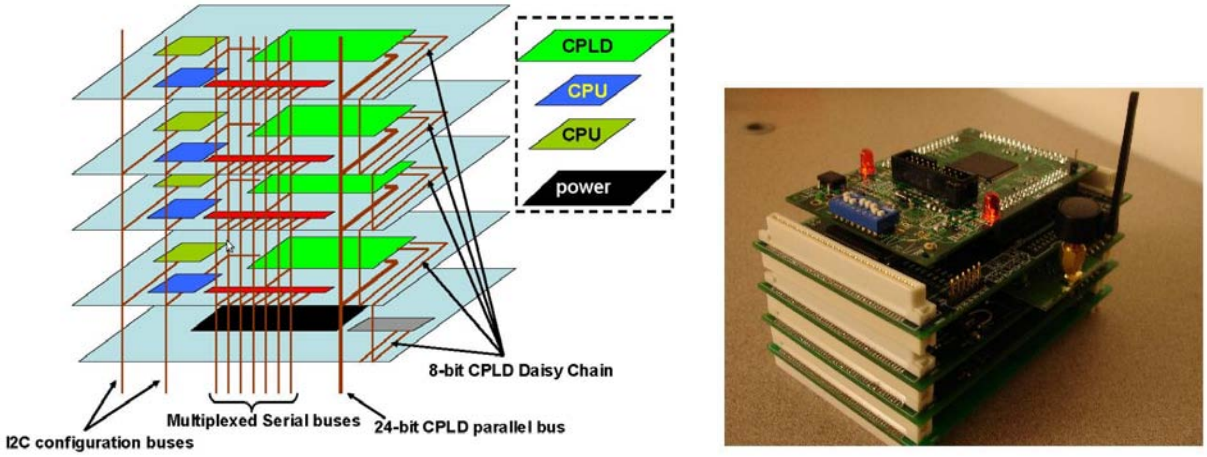


Figura 20: Arquitetura da plataforma mPlatform (LYMBEROPOULOS; PRIYANTHA; ZHAO, 2007).

O estudo de caso proposto no trabalho foi um sistema de localização através do som utilizando quatro microfones dispersos em um ambiente. O algoritmo utilizado é denominado SSL (Sound Source Location). O estudo foi realizado para analisar os diferentes protocolos de comunicação entre os módulos do nó-sensor, pois cada módulo contém um processador trabalhando em freqüências diferentes.

3.1.6 Plataforma Modular Reconfigurável

No artigo (PORTILLA; RIESGO; CASTRO, 2007) e (PORTILLA et al., 2006) é apresentada uma plataforma para RSSF que permite a utilização em diferentes aplicações, evitando a necessidade de re-confecção do sistema para atender os novos requisitos da aplicação. O nó-sensor apresenta modularidade no hardware, incluindo uma FPGA responsável por permitir a reconfigurabilidade da arquitetura, sendo possível a reconfiguração remota do sistema.

A camada de processamento é o principal componente do nó-sensor e onde está localizada a capacidade de modularização e reconfiguração da arquitetura de hardware. Esta camada possui um microcontrolador ADuC831 da Analog Devices e uma FPGA Spartan II da Xilinx, como representado na Figura 21. A combinação de um microcontrolador com um dispositivo de lógica programável introduz versatilidade e reconfigurabilidade, sendo que a FPGA acelera o processamento dos sinais proveniente dos sensores para o microcontrolador. Na FPGA estão implementados diversos protocolos para interfaceamento de sensores e atuadores como: SPI, PWM, I2C, entre outros. Para isso, uma biblioteca de interfaces foi desenvolvida para facilitar a inserção de novos sensores e atuadores ao nó-sensor.

A camada responsável pela comunicação possui um módulo de comunicação com tecnologia Bluetooth e outro módulo com tecnologia ZigBee. Estes módulos podem ser trocados dependendo dos requisitos da aplicação. O módulo Bluetooth possui um baixo consumo e uma taxa de transferência de dados de 723,2 Kbps. Por outro lado, o módulo ZigBee possui um baixo consumo de energia e uma baixa taxa de transferência de dados, sendo mais recomendado para aplicações que necessitam transmitir apenas dados de sensores simples.

A camada de sensoriamento possui os sensores e atuadores necessários para atender os requisitos da aplicação. Novos sensores e atuadores podem ser facilmente acoplados

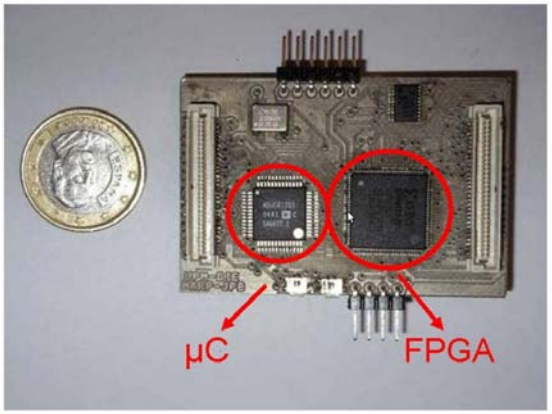


Figura 21: Camada de processamento (PORTILLA; RIESGO; CASTRO, 2007).

ao sistema. Os sensores são conectados diretamente à FPGA onde é realizado o processamento do sinal antes dos dados serem enviados ao microcontrolador, que irá disseminar a informação pela rede.

A camada de energia é responsável pela alimentação da arquitetura do nó-sensor proveniente de baterias. Nesta camada estão localizados os reguladores de tensão para suprir as diferentes tensões necessárias para alimentação dos componentes do nó-sensor.

Contudo, este trabalho apresentou uma arquitetura de hardware para RSSF que possibilita reconfiguração física, utilizando a modularidade da plataforma, e reconfiguração lógica do hardware, através da FPGA. A Figura 22 demonstra a modularidade da arquitetura e a interconexão dos módulos. Permitindo a adaptação a diversos requisitos das aplicações com um baixo custo de implementação.

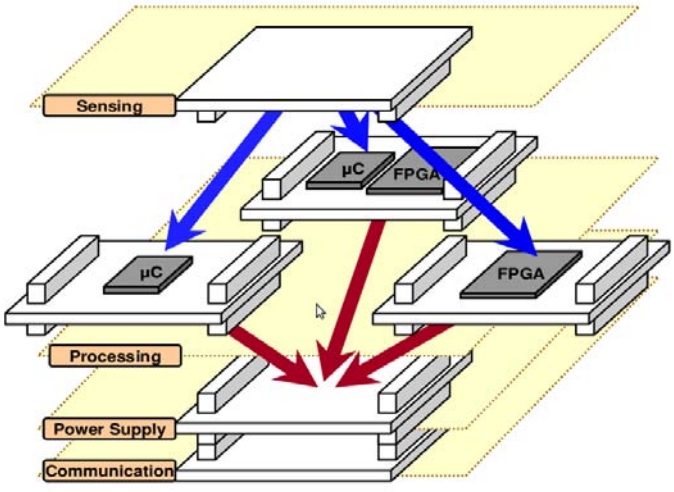


Figura 22: Nó-sensor com sua arquitetura modular (PORTILLA; RIESGO; CASTRO, 2007).

3.1.7 MedNode, Medical Motherboard e RFab

Massey, em (MASSEY et al., 2007), descreve a necessidade de utilizar reconfiguração de hardware e de software para obter flexibilidade e adaptabilidade em sistemas médicos embarcados. Estes dispositivos devem estar aptos a responder em situações de

emergência, como no diagnóstico de um ataque cardíaco do paciente monitorado. No trabalho é proposto um mecanismo de adaptação automatizada, onde o dispositivo se autoreconfigura, buscando atender as modificações dos requisitos do ambiente. Dois níveis de reconfiguração são explorados nestes dispositivos médicos:

- *Reconfiguração de hardware:* adapta o dispositivo para inclusão de novos sensores ou a reconfiguração de recursos do dispositivo.

- *Reconfiguração de software:* utiliza algoritmos que se adaptam ao ambiente para fazer um melhor uso dos recursos disponíveis e permitir atualização através de uma interface de comunicação sem fio.

A proposta é de uma plataforma plug-and-play onde há a adição de novos periféricos, neste caso sensores, sem a instalação manual de hardware e software adicionais. O dispositivo deve definir métodos para automaticamente adaptar o sistema embarcado à adição ou remoção de sensores em tempo de execução. Assim, dispositivos médicos que possuem estas características são citados: Medical Motherboard, CustoMed e RFab.

A *Medical Motherboard* (PARK; MACKENZIE; JAYARAMAN, 2002) é uma plataforma que permite a adição de sensores pequenos e com processamento de baixo consumo de energia denominados motes. A Medical Motherboard é equipada com uma FPGA para poder prover a reconfiguração de hardware necessária. Esta plataforma é flexível à adição e remoção de sensores motes, assim como a perda de um sensor ou da comunicação dos mesmos. A reconfigurabilidade da Medical Motherboard demonstra o benefício destas arquiteturas em sistemas médicos heterogêneos.

A *CustoMed* (Customizable Medical Devices) é uma plataforma composta por diversos MedNodes que são colocados no corpo humano, como demonstrado na Figura 23. Os MedNodes são dispositivos que possuem capacidade de processamento, sensores externos, uma bateria e um módulo de comunicação sem fio. Estes dispositivos suportam diversos tipos de sensores para monitoramento de sinais fisiológicos do corpo humano. A reconfiguração da plataforma é obtida com a adaptação à inserção e remoção destes nós-sensores no ambiente de monitoramento.

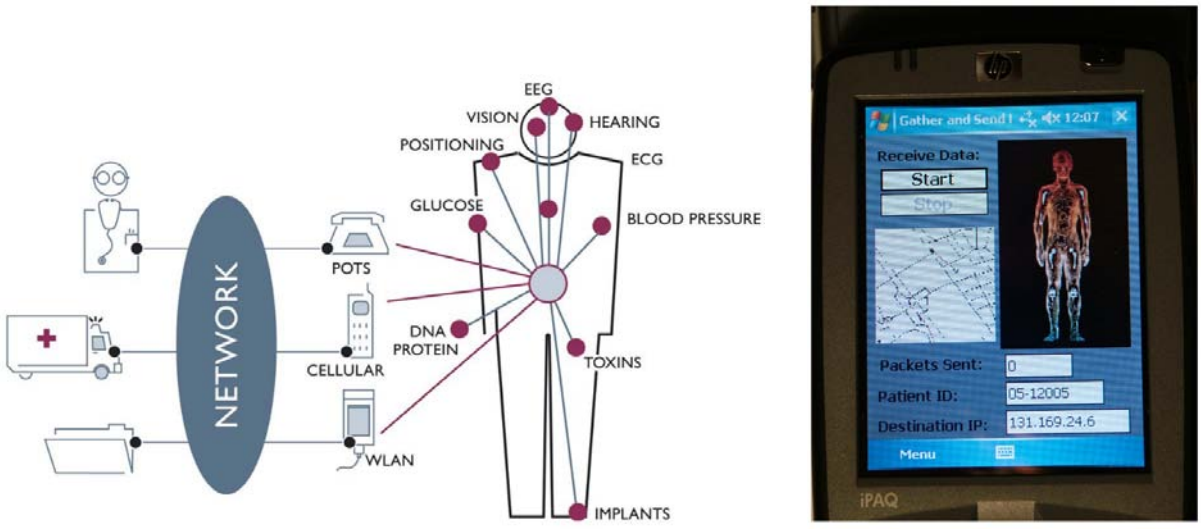


Figura 23: *MedNodes* monitorando sinais do corpo humano e enviando a um PDA (MASSEY et al., 2007).

A *RFab* (Reconfigurable Fabric) é uma plataforma "vestível", onde sensores são dispostos em uma roupa ou vestimenta para coletar sinais vitais do corpo humano. Os sensores são interconectados com fios e rasgos ou rupturas na roupa podem interromper a comunicação. Assim, uma reconfiguração entre os nós-sensores busca restabelecer a comunicação.

Este trabalho demonstra os benefícios da reconfiguração tanto de hardware como de software para superar a limitação de recursos, buscando balancear o consumo de energia, consumo de memória e a interoperabilidade em ambientes dinâmicos.

3.1.8 Plataforma Reconfigurável para Sensores Inteligentes - Oxímetro de pulso

Em (JOVANOV et al., 2004) é proposta uma plataforma reconfigurável para sensores inteligentes em aplicações médicas. A plataforma desenvolvida pretende integrar um sistema de monitoramento embarcado em vestuários para monitoração contínua de sinais fisiológicos do corpo humano. Com isso, os sensores integrariam uma rede que irá transmitir os sinais monitorados para análise e, assim, detectar e prevenir condições anormais de funcionamento.

A plataforma reconfigurável proposta é uma combinação de um microcontrolador de baixo consumo e um dispositivo de lógica programável, permitindo flexibilidade para suportar diversos tipos de sensores de monitoramento fisiológico, assim como, permitir a substituição dos sensores dinamicamente. O microcontrolador executa os algoritmos para o nó-sensor, reconfigura o dispositivo de lógica programável e controla o sistema. O hardware reconfigurável é utilizado para permitir uma aceleração no processamento de sinais críticos e na execução de protocolos de comunicação, garantindo as restrições de tempo da aplicação e reduzindo o consumo de energia. Esta plataforma possibilita a migração de código, para realização de atualizações de software e de parâmetros, e a reconfiguração de hardware para adequar o dispositivo às condições do ambiente ou do paciente, sendo a reconfiguração em tempo de projeto ou execução.

A reconfiguração do dispositivo pode ser realizada através de pedidos externos ou pelo próprio dispositivo, ou seja, auto-reconfiguração. A auto-reconfiguração é realizada a partir da necessidade de melhorar alguns requisitos como: relação sinal-ruído, consumo de energia, precisão nas medidas e nível de segurança.

Como estudo de caso, foi desenvolvido um oxímetro de pulso reconfigurável apresentado na Figura 24. O oxímetro de pulso é um equipamento utilizado sobre a pele humana para verificar o fluxo da corrente sangüínea, saturação de oxigênio no sangue, batimentos cardíacos e a amplitude da pulsação. Este equipamento utiliza diodos emissores de infravermelho e fotodiodos que recebem os sinais e realiza o condicionamento em um dispositivo de lógica programável, neste caso, um CPLD CoolRunner-II de baixo consumo da Xilinx. O dispositivo de lógica programável também é responsável por gerar os sinais de sincronização para um microcontrolador (MSP430F149 da Texas Instruments) responsável por realizar a conversão analógica-digital, filtragem e comunicação com uma estação de monitoramento. A princípio, a comunicação é realizada através de uma interface RS-232, sendo prevista uma comunicação sem fio para implementações futuras.

O objetivo é incorporar um aumento de sensibilidade e desempenho, utilizando a plataforma reconfigurável proposta, em dispositivos oxímetro de pulso já existentes, resultando em um equipamento portátil, com dimensões reduzidas e baixo consumo.

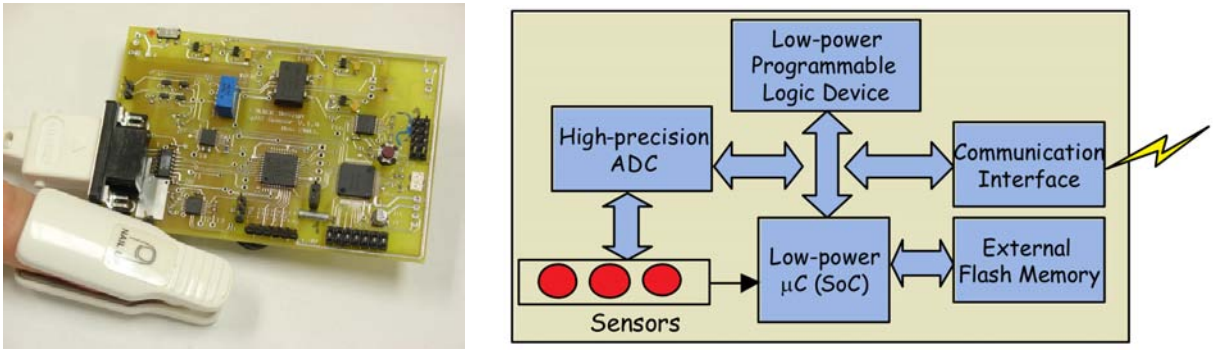


Figura 24: Arquitetura do oxímetro de pulso reconfigurável (JOVANOV et al., 2004).

3.1.9 Arquitetura de hardware com reconfiguração dinâmica - LEON2+RFU

No trabalho desenvolvido em (HINKELMANN; ZIPF; GLESNER, 2007) é proposta uma plataforma de nó-sensor genérica para RSSF, utilizando uma arquitetura de hardware que permite reconfiguração dinâmica, obtendo-se uma arquitetura flexível, energeticamente eficiente e com alto desempenho. O núcleo da arquitetura é composto pela combinação de um processador RISC e uma unidade que pode ser dinamicamente reconfigurável denominada RFU (Reconfigurable Function Unit), otimizada para o processamento de dados em aplicações para RSSF.

A utilização de nós-sensores com arquiteturas genéricas e reconfiguráveis possui diversas vantagens para o desenvolvimento de aplicações em RSSF, pois possibilitam a redução de custos e tempo de projeto. Além disso, a plataforma pode adaptar-se às condições do ambiente em tempo de execução.

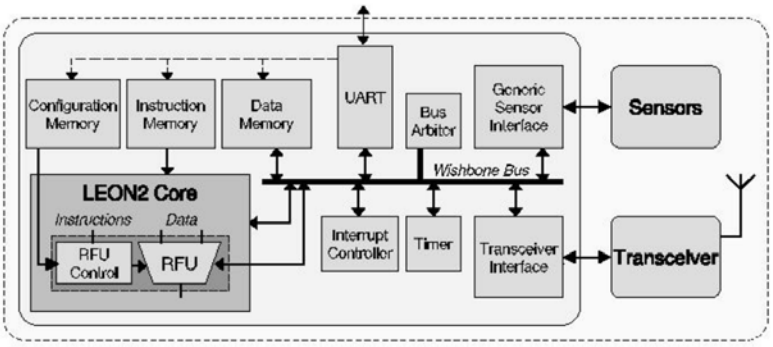
A arquitetura do nó-sensor está representada na Figura 25(a) e a estrutura modular da unidade de reconfiguração dinâmica (RFU) está demonstrada na Figura 25(b).

O nó-sensor possui um processador RISC denominado LEON2 de 32bits, sendo sua arquitetura modificada para uma melhor utilização dos recursos do nó-sensor. Juntamente com o processador, está integrada a RFU desenvolvida especialmente para aplicações em RSSFs. Esta unidade possui flexibilidade para desenvolver operações típicas em RSSFs como: correção de erros, codificação, entre outras. Enquanto o processador executa tarefas de controle do sistema, a RFU pode executar tarefas de processamento de dados, aumentando a eficiência do sistema com o paralelismo no processamento. A diferença desta plataforma de hardware reconfigurável para outras plataformas é a utilização freqüente de reconfiguração, em tempo de execução, da RFU.

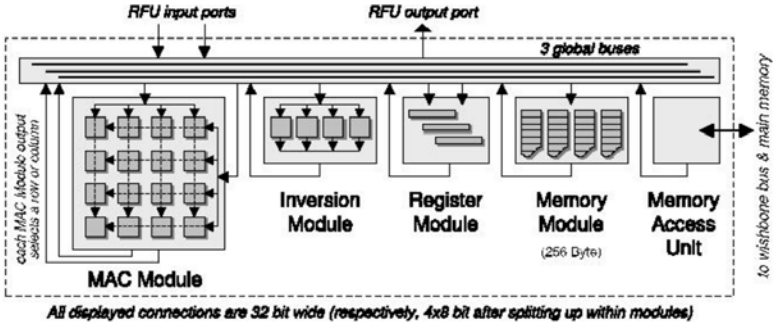
Um mecanismo de reconfiguração multicamadas foi desenvolvido e permite reconfigurações dinâmicas rápidas entre diferentes tarefas.

Como estudo comparativo entre plataformas foram realizados testes com tarefas típicas de RSSF como: detecção e correção de erros (CRC-8 e BCH) e algoritmos de codificação (AES). Cada tarefa foi implementada utilizando uma plataforma com RFU e outra sem RFU. Os resultados estão demonstrados nas Figuras 26(a) e 26(b). Em ambos os testes a plataforma com a RFU apresentou um melhor desempenho e um menor consumo de energia. Durante os testes pode-se verificar que a reconfiguração das funções para execução das tarefas consome um percentual de energia, porém não é um fator crucial em relação ao consumo total do nó-sensor.

Contudo, a plataforma desenvolvida apresentou um ganho em desempenho e consumo de energia com um pequeno acréscimo em área no chip utilizado, porém sem apresentar



(a)



(b)

Figura 25: (a)Arquitetura do nó-sensor genérico. (b)Estrutura modular da RFU (HINKELMANN; ZIPF; GLESNER, 2007).

restrições na flexibilidade da arquitetura.

Task	Version	Execution [cycles]	Reconf. [cycles]	Speed-up factors
CRC-8	software	928	-	1
CRC-8	RFU	7	18	132 (37)
AES key gen.	software	538	-	1
AES key gen.	RFU	57	33	9,4 (6,0)
AES encrypt.	software	1475	-	1
AES encrypt.	RFU	114	69	12,9 (8,1)
BCH decod.	software	2893	-	1
BCH decod.	RFU	347	55	8,3 (7,2)

(a)

Task	Version	Execution [nJ]	Reconf. [nJ]	Gain factors
CRC-8	software	387,6	-	1
CRC-8	RFU	4,8	12,2	81 (23)
AES key gen.	software	234,6	-	1
AES key gen.	RFU	39,2	22,5	6,0 (3,8)
AES encrypt.	software	640,6	-	1
AES encrypt.	RFU	104,9	43,1	6,1 (4,3)
BCH decod.	software	1208,7	-	1
BCH decod.	RFU	245,5	34,4	4,9 (4,3)

(b)

Figura 26: (a)Latência na execução de tarefas e reconfiguração. (b)Consumo de energia na execução de tarefas e reconfiguração (HINKELMANN; ZIPF; GLESNER, 2007).

3.1.10 Cluster Head Reconfigurável - RCH

As RSSFs apresentam uma diversidade de nós-sensores distribuídos em um ambiente com o objetivo de coletarem dados e transmiti-los pela rede. Porém, muitos destes dados são redundantes e faz-se necessário um dispositivo para eliminá-los e agregá-los. Em (COMMURI; TADIGOTLA, 2007) é proposto a utilização de uma rede hierárquica, sendo o *cluster head* um dispositivo reconfigurável com o objetivo de realizar a fusão dos dados da rede.

O *cluster head* reconfigurável, denominado RCH (Reconfigurable Cluster Head), foi desenvolvido sobre uma plataforma FPGA e pode ser reconfigurado em tempo de exe-

cução para implementar diferentes estratégias de agregação de dados e protocolos de comunicação. A habilidade de reconfiguração dinâmica é essencial em redes em que não é determinado o número de dispositivos nós-sensores no desenvolvimento inicial da aplicação. Assim, facilmente pode-se reconfigurar a estratégia de agregação de dados ao número de dispositivos da rede, o que não é possível em nós-sensores compostos somente por microcontroladores de arquitetura fixa.

A arquitetura do dispositivo é composta por uma FPGA Virtex-II da Xilinx equipada com um módulo de comunicação sem fio Wi.232 DTS da Radiotronix Inc. Na Figura 27 está representada a arquitetura do dispositivo *cluster head* proposto.

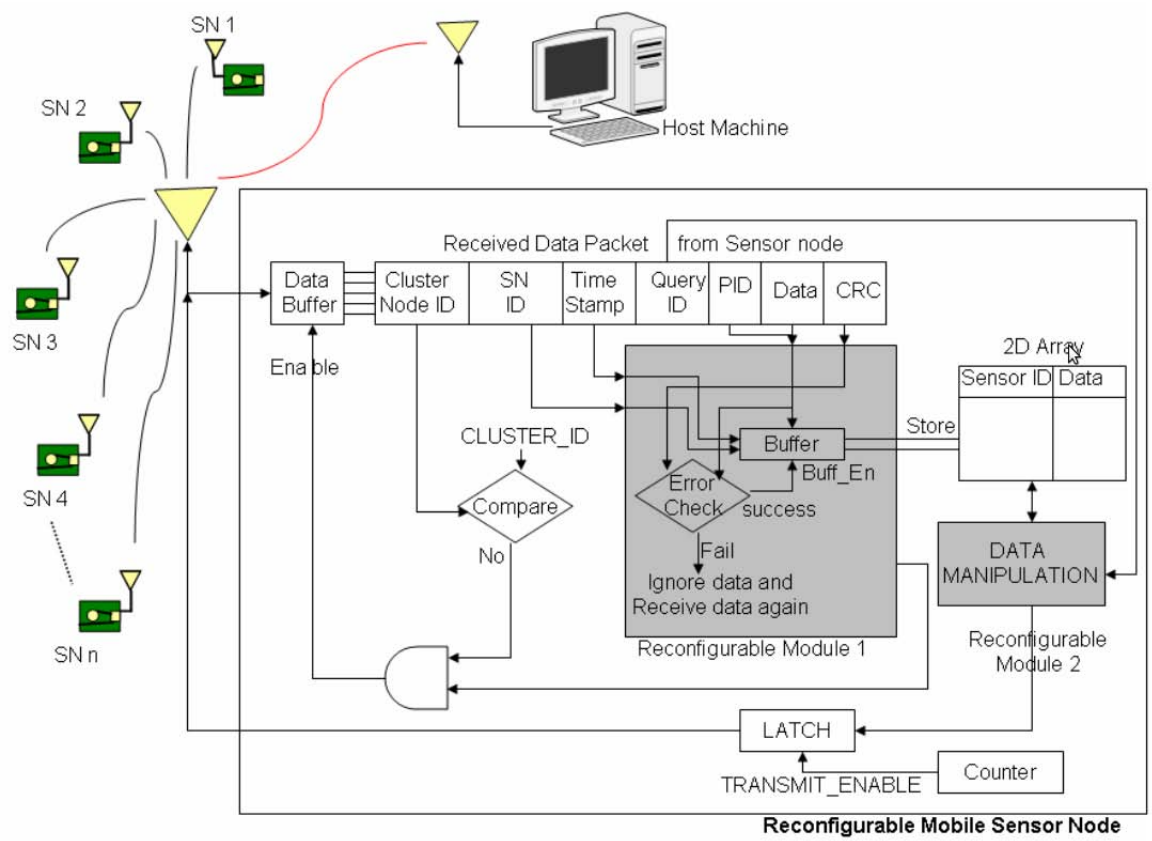


Figura 27: Arquitetura do dispositivo Cluster Head na rede (COMMURI; TADIGOTLA, 2007).

A implementação do *cluster head* reconfigurável demonstrou que a utilização de diferentes algoritmos de agregação de dados, sendo reconfigurados em tempo de execução, torna a rede mais eficiente. A implementação resultou em uma redução do consumo de energia da rede, ou seja, aumentando sua vida útil em operação, além de uma redução na ocupação do hardware e no tempo de processamento, como demonstrado nos gráficos da Figura 28. Porém, pode-se verificar que o consumo de energia aumenta rapidamente com o aumento no número de operações de agregação. Contudo, a implementação apresentada pode reduzir o tráfego de dados na rede, aumentando o desempenho da mesma.

3.1.11 Multi-Radio Platform

No trabalho desenvolvido em (KOHVAKKA et al., 2006) é proposta uma plataforma para RSSF que se diferencia de outras atualmente existentes por permitir a implementação de até quatro rádios de comunicação no mesmo nó-sensor. O nó-sensor é capaz

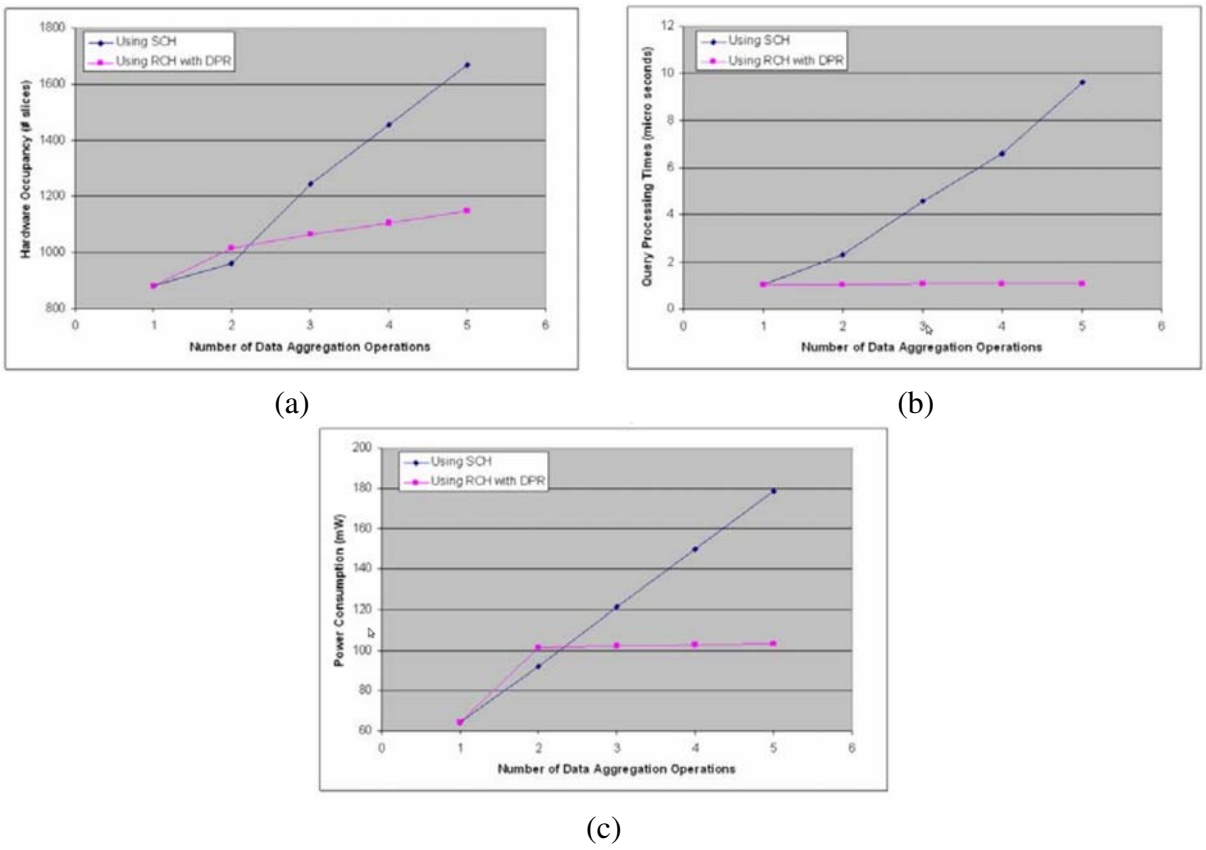


Figura 28: (a)Efeito do número de agregações na ocupação do hardware. (b)Efeito do número de operações de agregação no tempo de processamento. (c)Efeito do número de agregações no consumo de energia (COMMURI; TADIGOTLA, 2007).

de se comunicar com até quatro outros nós-sensores ao mesmo tempo, aumentando a taxa de dados transmitidos e reduzindo a latência na comunicação. Este nó-sensor, com múltiplos rádios, pode ser aplicado como roteador, gateway ou como um nó-sensor com alto-desempenho.

A utilização de até quatro rádios no mesmo nó-sensor permite o envio e o recebimento de dados simultaneamente com diversos nós-sensores vizinhos. A possibilidade de transmissão de dados por mais de um caminho e por mais de uma freqüência, ao mesmo tempo, aumenta a tolerância a falhas na comunicação. O módulo de comunicação sem fio utilizado foi o nRF2401A da Nordic Semiconductor que opera na freqüência de 2,4 GHz e possui uma taxa de transmissão de até 1 Mbps.

O processamento dos dados no nó-sensor é realizado por até quatro processadores *softcore* sintetizados em uma FPGA Cyclone da Altera. O *softcore* utilizado é o Nios II que realiza a comunicação entre processos através de uma memória compartilhada. Na Figura 29 está representado o nó-sensor e a sua arquitetura de hardware. Este processador é de 32 bits e com uma memória cache de 512 bytes, possuindo um consumo de energia baixo que aumenta linearmente com o número de processadores sintetizados na FPGA. O acesso aos rádios de comunicação sem fio é realizado por um controlador centralizado que realiza a interface com os processadores.

A reconfigurabilidade dos nós-sensores possibilita uma heterogeneidade entre os nós da rede, onde cada nó-sensor pode ser otimizado para realizar uma dada tarefa.

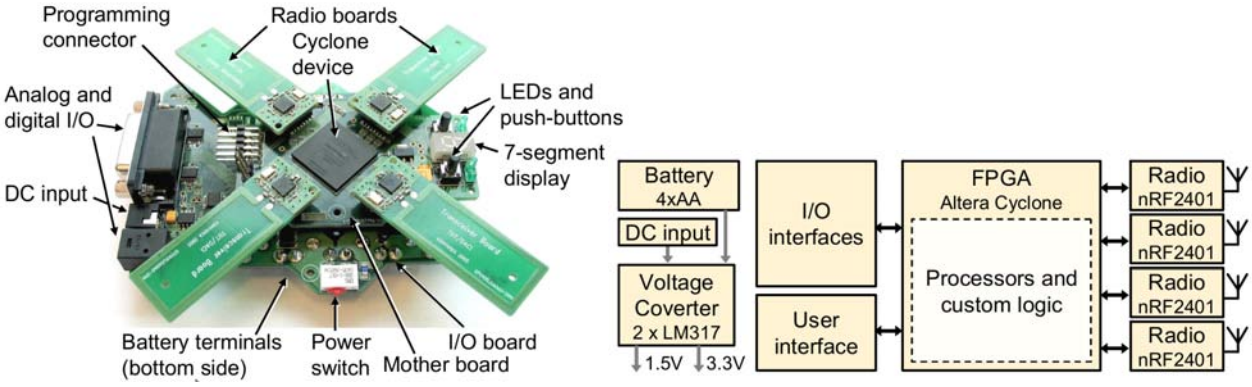


Figura 29: Arquitetura da plataforma Multi-Rádio (KOHVAKKA et al., 2006).

No artigo (KOHVAKKA et al., 2006) são apresentadas duas aplicações para a plataforma desenvolvida para demonstrar a minimização da latência na comunicação e o aumento da taxa de transferência de dados. Na Figura 30 pode-se visualizar a comunicação de um nó-sensor com outros dois nós utilizando dois módulos de comunicação, verificando-se os tempos e os atrasos envolvidos no processo.

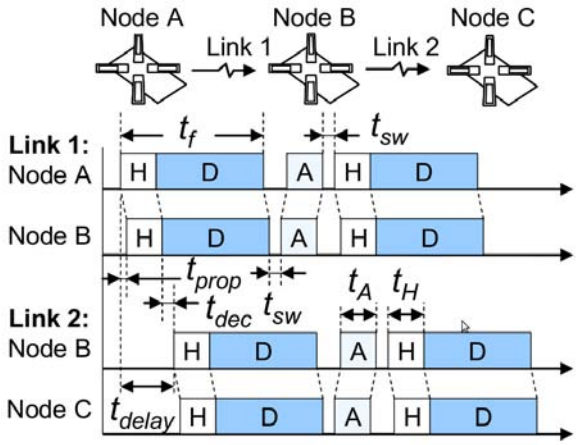


Figura 30: Temporização da comunicação entre nós (KOHVAKKA et al., 2006).

3.1.12 Cyclops

As RSSFs são muito utilizadas em diversas aplicações para sensoriamento de ambientes, porém poucas utilizam processamento de imagens para realizar esta tarefa. A visão é um dos sentidos mais importantes nos seres-humanos. Com isso, em (RAHIMI et al., 2005) os autores exploram a utilização de RSSFs para captar e transmitir imagens do ambiente onde a rede se encontra. Os nós-sensores desenvolvidos utilizam uma câmera CMOS e foram denominados de Cyclops.

Os Cyclops utilizam a plataforma de hardware dos dispositivos Mica2 da Crossbow e implementam alguns componentes para acelerar o processamento de imagens como: câmera CMOS, microcontrolador, CPLD e memória SRAM, como representado na Figura 31. A câmera CMOS utilizada é a ADCM-1700 da Agilent que necessita de um relógio com baixa velocidade e possui baixo consumo de energia. O microcontrolador utilizado é um ATmega128L e é responsável por controlar o sistema sensor do Cyclops, além de ajustar os parâmetros da imagem. O CPLD utilizado é um CoolRunner da Xilinx

responsável por gerar os sinais de sincronismo para a câmera e para as memórias. As memórias SRAM são utilizadas para armazenar as imagens, pois a memória interna do microcontrolador é muito pequena.

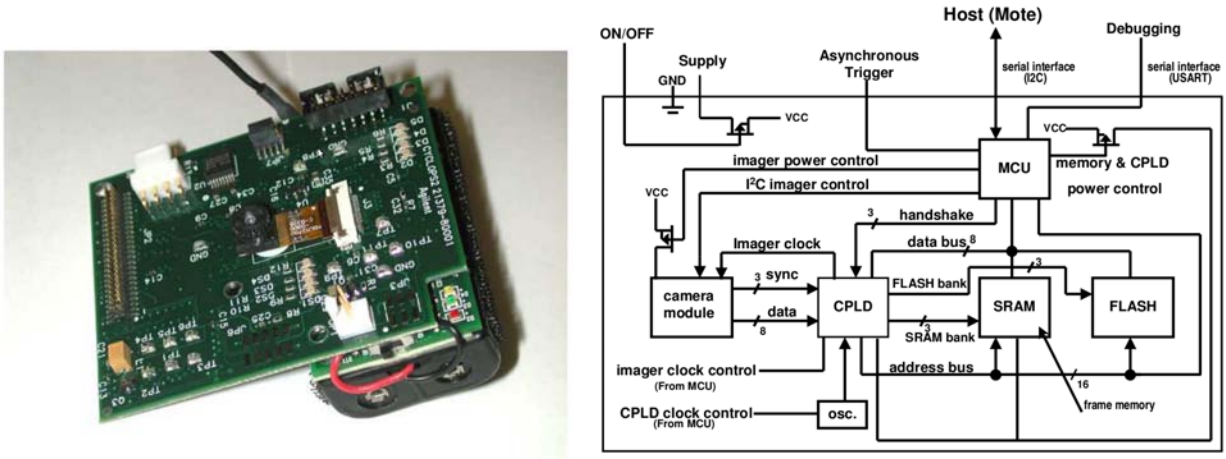


Figura 31: Arquitetura de hardware da 1a geração do Cyclops (RAHIMI et al., 2005).

Como estudos de caso foram desenvolvidos dois algoritmos para detecção de objetos e de gestos executados por pessoas. Através da plataforma Cyclops, outras aplicações podem ser desenvolvidas utilizando a captura de imagens do ambiente. Ela pode ser combinada com outros sensores para aumentar a confiabilidade das medidas ou em aplicações de segurança e supervisão de ambientes.

3.2 Discussão

Analisando os trabalhos apresentados que utilizam arquiteturas reconfiguráveis na arquitetura do nó-sensor, pode-se verificar que a reconfigurabilidade introduz diversas características interessantes e benéficas para o desempenho da rede. Dentre estas características, pode-se destacar as seguintes:

- *Flexibilidade:* a confecção de plataformas genéricas e flexíveis possibilitam a utilização em um número maior de aplicações, economizando tempo e custo de projeto.
- *Adaptabilidade:* a modificação dos requisitos da aplicação e das características do ambiente em que as RSSFs estão incluídas necessitam de mecanismos que se adaptem a estas mudanças, não reduzindo o desempenho na realização das tarefas.
- *Reconfigurabilidade:* a possibilidade de alterar o hardware do nó-sensor possibilita um ganho em custo e tempo de projeto e permite adaptação aos requisitos da rede em tempo de execução.
- *Processamento:* arquiteturas reconfiguráveis apresentam um aumento de processamento aos dispositivos por realizar diversas operações em hardware. Podem ser utilizadas como co-processadores realizando paralelismo no processamento de tarefas.

Além disto, pode-se verificar que existem propostas que utilizam arquiteturas híbridas, com um microcontrolador e uma arquitetura reconfigurável no mesmo nó-sensor, e outras

que utilizam um *softcore* sintetizado em uma arquitetura reconfigurável. As duas propostas possuem características interessantes para o nó-sensor e buscam torná-lo flexível e adaptável. Sua utilização proporciona a facilidade de se realizar atualizações (*updates*), melhoramentos (*upgrades*) e correções a distância. Estas características tornam ideal a utilização destes componentes para implementar soluções otimizadas e eficientes para aplicações que requeiram computação intensiva.

Destaca-se o resultado obtido nos diferentes trabalhos referentes à economia de energia proporcionada pela utilização das arquiteturas reconfiguráveis, onde pode-se verificar que o aumento na capacidade de processamento beneficia a vida útil do nó-sensor e, por conseqüência, a vida útil da rede.

Porém, dentre os trabalhos analisados, não foi verificado uma abordagem referente à otimização e utilização de arquiteturas dedicadas aos requisitos da aplicação. Os microcontroladores apresentados possuem diversos recursos que, muitas vezes, não são utilizados e acabam por apresentar um consumo de energia adicional ao nó-sensor. Assim, a utilização de uma ferramenta que realize a síntese de um microcontrolador dedicado é importante para aplicações em RSSFs.

Outra característica que não foi apresentada é a utilização de uma linguagem de programação orientada a objetos que auxilie no reuso de código e na portabilidade das aplicações desenvolvidas. A síntese de um microcontrolador que implemente uma máquina virtual que respeite a especificação Java e seja capaz de interpretar opcodes Java, introduzindo estas características ao nó-sensor.

Buscando suprir as lacunas apresentadas nesta análise, foi proposto para este trabalho a criação de um nó-sensor reconfigurável e customizável baseado na especificação Java denominado FemtoNode, o qual será descrito nos capítulos que seguem.

4 FEMTONODE

A arquitetura do nó-sensor é fundamental para desenvolver uma aplicação eficiente para uma RSSF. Deseja-se ter um módulo de processamento dedicado com o menor consumo de energia possível e com um transceptor de comunicação sem fio para realizar a disseminação da informação pela rede.

O FemtoNode constitui um nó-sensor para RSSF contendo um microcontrolador RT-FemtoJava, sintetizado a partir da ferramenta SASHIMI, e um módulo de comunicação sem fio adicionado à arquitetura do microcontrolador. Sendo o RT-FemtoJava customizável, o seu código é adaptado às necessidades da aplicação, sendo que somente o hardware necessário é sintetizado, reduzindo assim a área de hardware necessária e a potência dissipada. Esta característica beneficia o nó-sensor, pois consumo de energia é um requisito a ser minimizado em aplicações para RSSF, dado que a energia é um recurso limitado. Reduzindo-se os recursos não utilizados durante a síntese da arquitetura do nó-sensor, viabiliza-se a implementação em arquiteturas reconfiguráveis com menos unidades lógicas disponíveis e contempla-se uma maior portabilidade da aplicação entre estas arquiteturas.

A comunicação entre os nós-sensores da rede é realizada através de um módulo de comunicação sem fio incorporado à arquitetura do FemtoNode. O módulo é constituido de um transceptor de rádio-freqüência que utiliza o padrão IEEE802.15.4 e um Módulo Wireless, descrito em VHDL, que faz a comunicação entre o microcontrolador Femto-Java e o transceptor. O Módulo Wireless utiliza os barramentos de dados e endereços do microcontrolador FemtoJava para comunicação com o processador, realizando a troca de dados e possibilitando a configuração de parâmetros do transceptor.

4.1 Arquitetura

A arquitetura do FemtoNode pode ser dividida em duas partes: unidade de processamento e unidade de comunicação, juntamente com a API de comunicação. A união destes três elementos é fundamental para a execução das tarefas do nó-sensor.

A seguir, cada um dos elementos será descrito.

4.1.1 Unidade de processamento

A unidade de processamento é responsável por realizar as tarefas de execução e processamento descritas pelo código de programa da aplicação. Ela é composta pelo microcontrolador RT-FemtoJava multiciclo descrito na seção 2.3.2.1.

O microcontrolador FemtoJava possui quatro barramentos em sua arquitetura para comunicação com os periféricos, sendo barramentos distintos para o acesso e endereça-

mento das memórias de programa e dados. Estes barramentos são parametrizáveis, sendo que para a implementação do FemtoNode foram utilizados os barramento de endereços e dados de 16 bits e 32 bits, respectivamente. O FemtoNode utiliza esses barramentos para realizar a comunicação com o *Módulo Wireless* incorporado à arquitetura do FemtoJava, realizando a interface entre o microcontrolador e o transceptor de rádio-freqüência. Na Figura 32 estão representados as ligações entre o barramento do FemtoJava e o *Módulo Wireless*.

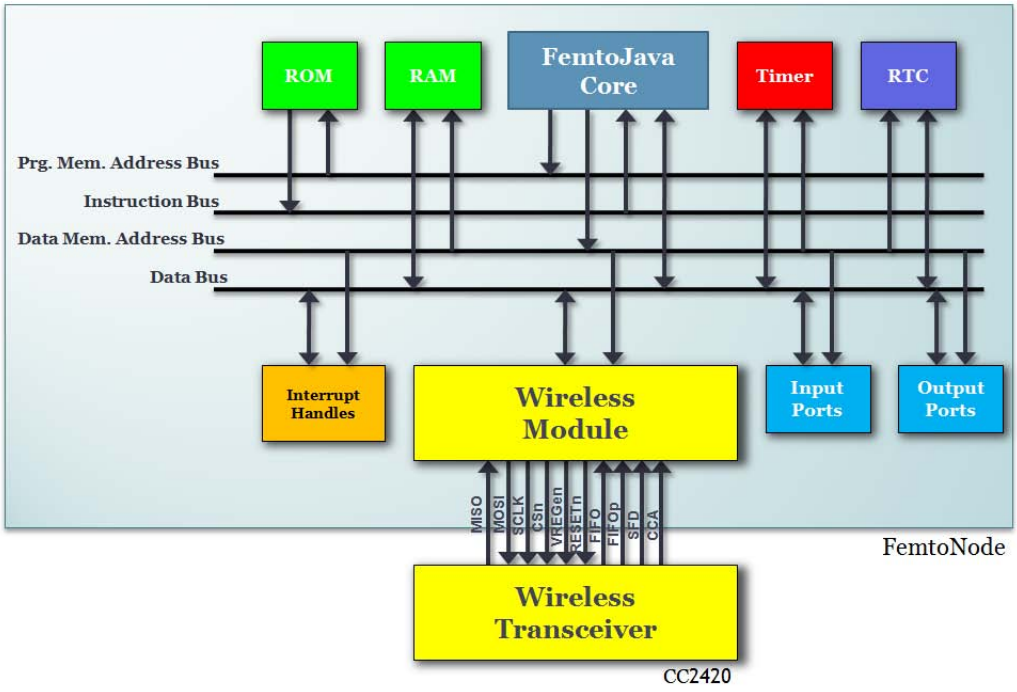


Figura 32: Arquitetura de hardware do nó-sensor FemtoNode.

O gerenciamento das escritas e leituras do sistema de entrada e saída do microcontrolador Java são de responsabilidade do arbritador do barramento. Ele considera o endereço contido no barramento para determinar a porta sobre a qual o microcontrolador deve operar. A escrita dos dados em um periférico ou em uma porta devem ser primeiro armazenados na pilha juntamente com o endereço correspondente. A mesma estrutura é utilizada para a leitura, onde o dado obtido da porta é armazenado na pilha de operandos.

A memória do FemtoJava está dividida em duas partes, a memória de dados e a memória de programa. Na memória de dados os endereços iniciais, da posição 00H até a posição 10H, contém alguns registradores mapeados em memória, que tem a finalidade de possibilitar a programação das interrupções e dos temporizadores assim como permitir as operações de I/O. Com a introdução da API-RTSJ, as posições de memória do endereço 10H até 12H foram ocupados pelo registrador do relógio de tempo-real. A partir do endereço 10H a memória é utilizada para guardar as constantes e os campos dos objetos, para a pilha do FemtoJava, e como conseqüência para a alocação dos frames dos métodos.

Como o *Módulo Wireless* realiza a comunicação com o microcontrolador através dos barramentos de endereços e dados, foi necessário adicionar quatro novos endereços para referenciar escritas e leituras realizadas pelo módulo. Assim, estes quatro novos endereços receberam suas posições na pilha da memória de dados ocupando os endereços da posição 13H até 16H, conforme representado na Figura 33.

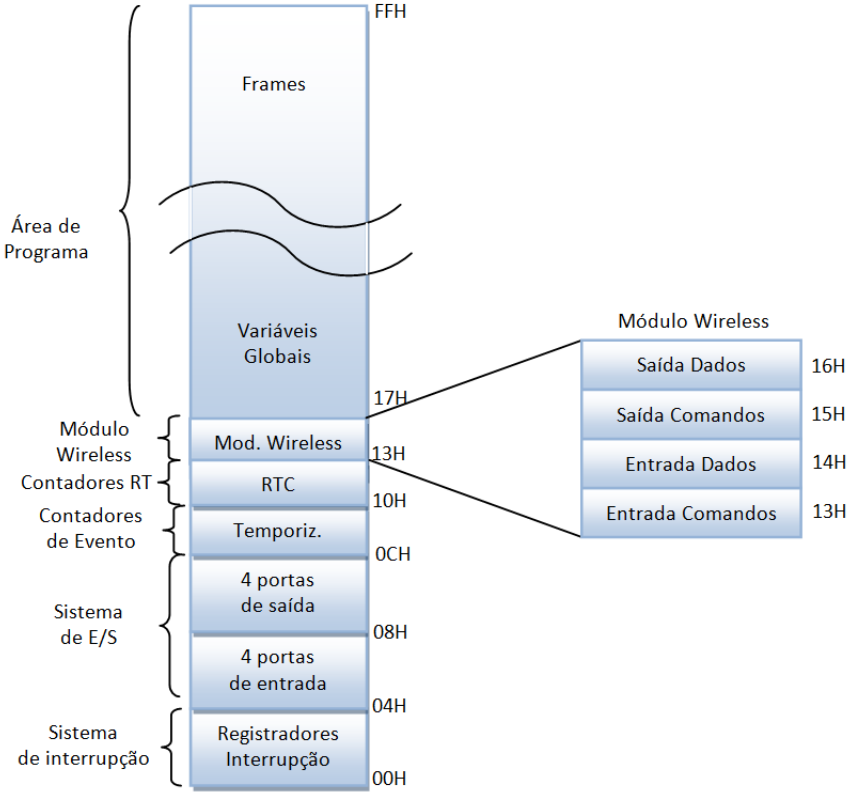


Figura 33: Organização da memória de dados do FemtoJava com o *Módulo Wireless*.

Para facilitar a comunicação foi determinado que dois endereços serão utilizados para informar ao FemtoJava ou ao *Módulo Wireless* a operação que será realizada pelo periférico através de comandos e, também, foi determinado que dois endereços serão para o envio e recebimento de dados. A Tabela 3 apresenta a descrição dos comandos para comunicação entre o FemtoJava e o *Módulo Wireless*.

A especificação da Máquina Virtual Java não define mecanismos para manipular interrupções, pois visava a portabilidade entre diferentes plataformas de hardware. Entretanto o sistema de tratamento de interrupção é um componente importante a ser incluído em um microcontrolador, para facilitar a comunicação entre o processador e os seus periféricos, liberando o processador de ficar esperando a requisição do periférico. Assim, o microcontrolador FemtoJava implementou cinco interrupções distintas com dois níveis de prioridade para facilitar a requisição de periféricos ao processador. O *Módulo Wireless* utiliza a interrupção INT1 para realizar a requisição de comunicação com o processador.

4.1.2 Unidade de comunicação

Na arquitetura do FemtoNode foi incorporado um módulo de comunicação sem fio para realizar a transferência de dados entre os nós-sensores da rede, possibilitando a disseminação da informação coletada. Este módulo é constituido por um transceptor de rádio-freqüência e um componente denominado *Módulo Wireless*. O transceptor de rádio-freqüência escolhido foi o CC2420 da empresa Texas Instruments que utiliza o padrão IEEE802.15.4 e será descrito na seção 4.1.2.1. O *Módulo Wireless* foi implementado em linguagem de descrição de hardware e inserido na arquitetura do FemtoJava. Ele é responsável por realizar a interface entre o microcontrolador e o transceptor, sendo descrito na seção 4.1.2.2.

Tabela 3: Comandos para comunicação entre Processador e Módulo Wireless.

Comando	Descrição
0x01	Endereço Remetente 16bits
0x02	Endereço Destino 16bits
0x03	Dados
0x04	Endereço PAN
0x05	Tamanho dos dados
0x06	Enviar
0x07	Configuração
0x08	Status
0x09	Erro de envio
0x0a	Tipo de datagrama
0x0b	Endereço Remetente 64bits
0x0c	Endereço Destino 64bits
0x0d	Valor do RSSI

4.1.2.1 Rádio transceptor - CC2420

O CC2420 (INSTRUMENTS, 2007) é um módulo de comunicação sem fio desenvolvido pela empresa Texas Instruments. Este módulo tem como objetivo apresentar um baixo consumo de energia (Tabela 4) e uma baixa taxa de transferência (250Kbps), sendo ideal para a utilização em aplicações de RSSFs. O CC2420 opera na faixa de freqüência de 2,4GHz e utiliza o protocolo IEEE802.15.4 que define as camadas física e enlace do padrão ZigBee, como descrito na seção 2.4.1.1. A fabricação de protótipos utilizando o CC2420 é simplificada, pois se necessita de poucos componentes passivos externos para realizar a montagem. O módulo pode ser visualizado na Figura 34.

Figura 34: Módulo de comunicação sem fio CC2420.

O módulo possui quatro estados de trabalho: transmitindo, recebendo, baixo consumo e desligado. Outras características importantes são:

- Transceptor de 2,4 GHz, operando entre 2400 MHz - 2483.5 MHz
- Propagação DSSS (Direct Sequence Spread Spectrum)
- Taxa de transferência de dados de 250 Kbps

- Modulação O-QPSK
- Baixo consumo de energia (vide Tabela 4)
- Sensibilidade elevada (-95 dBm)
- Potência de transmissão programável
- 128 bytes de transmissão de dados FIFO
- 128 bytes de recepção de dados FIFO
- Configuração através de protocolo SPI

Tabela 4: Consumo de corrente do módulo CC2420.

Modo	Consumo de Corrente
Ocioso	426 μA
Recepção	18.8 mA
Transmissão	17.4 mA (pot. máx.)

O módulo de comunicação apresenta o protocolo SPI (Serial Peripheral Interface) para acesso aos registradores internos, possibilitando a configuração e operação do módulo facilmente por dispositivos digitais como microcontroladores e FPGAs. Deve-se respeitar a freqüência máxima de 10 MHz na transmissão de dados.

O CC2420 possui oito sinais para realizar a comunicação entre o módulo e um FPGA ou microcontrolador como representado na Figura 35.

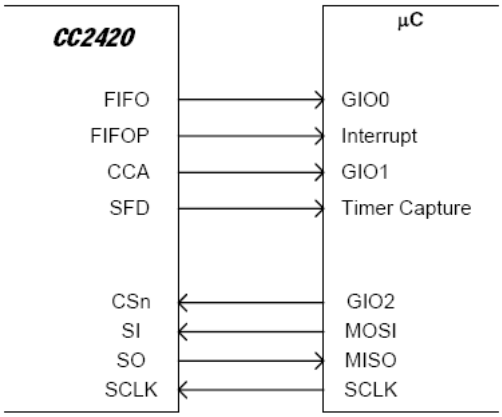


Figura 35: Diagrama da conexão entre FPGA e o módulo CC2420.

Os sinais utilizados são:

- Sinais do módulo CC2420:

  FIFO: sinal responsável pelo estado do buffer de recepção. Ativo quando possui dados no buffer.

  FIFOP: sinal programável responsável pelo estado do buffer de recepção. Ativo quando há um número pré-determinado de dados no buffer.

CCA: sinal responsável pelo monitoramento do estado do meio de comunicação. Ativo quando o canal está liberado para comunicação.

SFD: sinal responsável pela indicação de recepção ou envio do preâmbulo do datagrama.

- Sinais do protocolo SPI:

CSn: Chip Select. Pino utilizado para seleção de dispositivos para realização da comunicação. Dispositivo selecionado com CSn em nível lógico zero.

SI: Slave Input. Pino utilizado para recepção de dados.

SO: Slave Output. Pino utilizado para transmissão de dados.

SCLK: Slave Clock. Pino responsável pelo relógio da transmissão de dados.

Conjuntamente com a interface SPI, deve-se verificar os outros sinais FIFO, FIFOP, SFD e CCA para realizar a comunicação com o módulo, verificando o seu status. Além destes sinais, o VREG_EN e RF_RESET devem ser tratados na inicialização do módulo.

O módulo CC2420 possui dois registradores para realizar o envio e recebimento de dados através da comunicação por RF: TXFIFO e RXFIFO, respectivamente. Ambos registradores suportam 128 bytes de dados cada um. Os sinais FIFO e FIFOP, em nível lógico alto, indicam se o registrador RXFIFO encontra-se vazio, com dados ou cheio.

A comunicação entre os dispositivos baseia-se no padrão IEEE802.15.4, assim deve-se respeitar o formato dos datagramas estabelecidos para realizar o envio e recebimento de dados. Como se pode observar na Figura 36, o datagrama é dividido em duas partes: cabeçalho e campo de dados, onde se encontram os campos destinados à camada de enlace.

O cabeçalho do datagrama é constituído por um campo de sincronização e um campo onde se identifica a quantidade de bytes existentes no campo de dados.

A parte destinada à camada de enlace é subdividida nos seguintes campos: campo de controle, número de seqüência, campo de endereço, campo de dados e código de detecção de erros. O campo de controle é responsável por informar e configurar o datagrama que será enviado, sendo necessário informar o tipo do datagrama. O campo do número de seqüência é utilizado para informar a ordem dos datagramas que estão sendo enviados para que os dados sejam ordenados no destinatário. O campo de endereço possui o endereço do remetente e do destinatário, ou broadcast, podendo ser de 64 ou 16 bits, além de incluir o código da rede (PAN code), caso necessário. O campo de dados contém as informações a serem enviadas e deve respeitar um tamanho máximo para que os campos destinados a camada de enlace não sejam maiores do que 127 bytes, limitados pelo byte de identificação do tamanho do datagrama e não pelo tamanho do registrador de envio de dados. Por fim, o campo de detecção de erros é responsável por conter um número gerado por um algoritmo específico com o qual é possível realizar a identificação da integridade dos dados. Este pode ser gerado automaticamente pelo módulo utilizando o padrão CRC-16 ou calculado e inserido pelo usuário.

4.1.2.2 Módulo Wireless

Com o objetico de realizar a interface entre o processador e o transceptor de rádio-freqüência desenvolveu-se um módulo denominado de *Módulo Wireless*. O *Módulo Wireless* realiza a comunicação com o processador através dos barramentos de endereço e dados e, também, utilizando o sistema de interrupção para requisição ao processador.

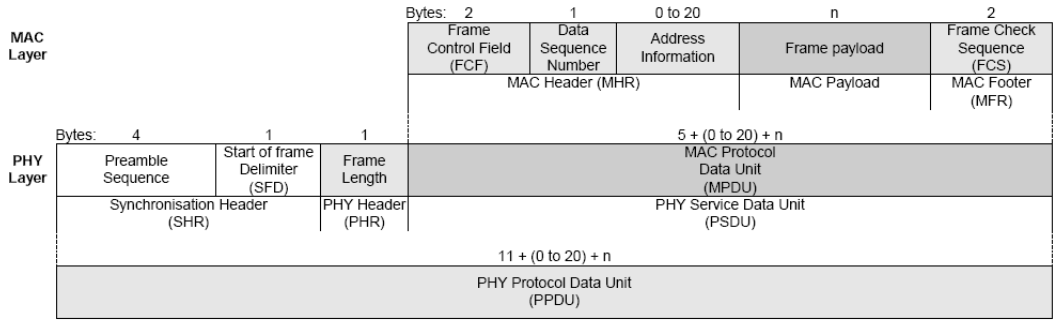


Figura 36: Datagrama do protocolo IEEE802.15.4 (IEEE802.15.4, 2003).

Para realizar a comunicação com o transceptor de rádio-freqüência CC2420, o módulo utiliza o protocolo SPI (MOTOROLA, 1996), sendo a comunicação realizada em modo full-duplex.

Como o FemtoNode foi projetado para utilização em arquiteturas reconfiguráveis, o *Módulo Wireless* foi desenvolvido em linguagem de descrição de hardware VHDL. Na Figura 37 tem-se uma visão geral de como o sistema está implementado.

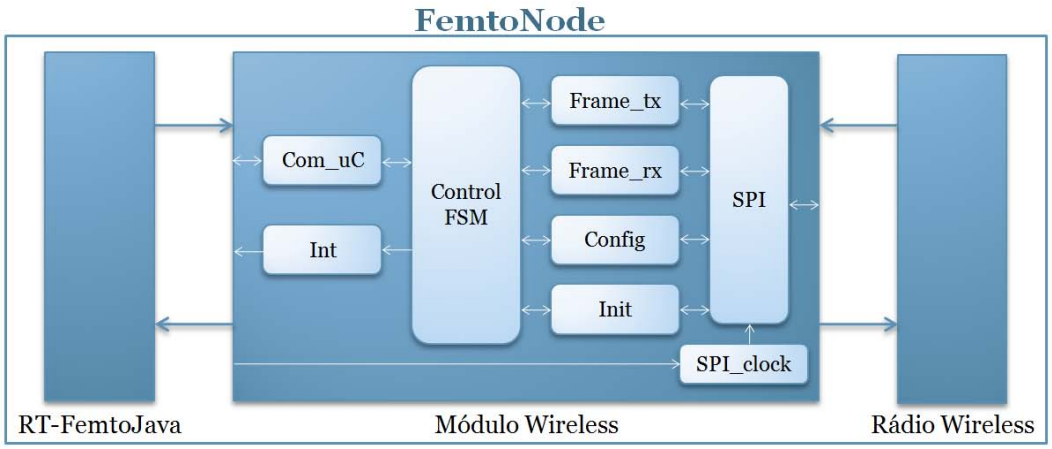


Figura 37: Arquitetura de hardware do nó-sensor FemtoNode.

Com o objetivo de facilitar o entendimento dos diferentes mecanismos que formam o *Módulo Wireless*, a seguir será realizado uma descrição do funcionamento de cada uma das partes integrantes e que estão apresentadas na Tabela 5.

Tabela 5: Arquivos VHDL que compõem o *Módulo Wireless*.

Arquivos	**Descrição**
`modulo_wireless`	Topo da arquitetura
`frame_rx`	Envia mensagens
`frame_tx`	Recebe mensagens
`mod_config`	Configuração do rádio
`mod_int`	Inicialização do rádio
`spi_com`	Protocolo SPI

modulo_wireless

O modulo_wireless é responsável pelo gerenciamento da máquina de estados, garantindo o correto funcionamento do *Módulo Wireless*, através do seqüenciamento das operações. Inicialmente, o modulo_wireless solicita ao mecanismo *mod_init* para realizar a inicialização do transceptor, preparando-o para a comunicação com os parâmetros necessários.

Ele também realiza a verificação das requisições de escrita e leitura do processador, através da leitura do barramento de endereços. Os dados enviados pelo processador são encaminhados ao frame_tx que é responsável por enviá-los ao transceptor de rádio-freqüência. Os dados recebidos pelo transceptor, através do frame_rx, são encaminhados ao processador. O envio é realizado através da escrita no barramento de dados e endereços após ser gerada uma interrupção para informar ao processador sobre uma requisição de transferência de dados.

mod_init

O mod_init é responsável pela inicialização do transceptor com a configuração correta. A configuração do módulo CC2420 exige a inicialização e configuração de alguns registradores internos ao componente, com o objetivo de definir o modo de trabalho. Há três etapas fundamentais que devem ser seguidas para o correto funcionamento do módulo.

A etapa de inicialização do módulo é necessária para configuração inicial do modo de trabalho do componente, conforme representado no diagrama de estados da Figura 38. Inicialmente aciona-se o regulador de tensão interno e reinicializa-se o módulo para certificação de que o mesmo encontra-se no estado inicial. Posteriormente, realiza-se o acionamento do oscilador interno responsável por gerar o relógio para o módulo. Finalmente, configuram-se os registradores com as opções definidas pelo usuário e coloca-se o módulo em estado de recepção.

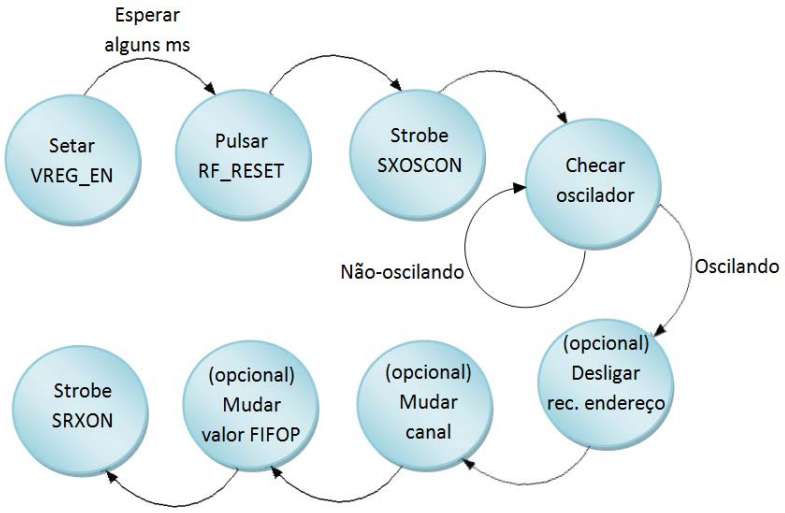


Figura 38: Diagrama de estados da inicialização do módulo.

mod_config

O `mod_config` é necessário para possibilitar a configuração do transceptor através do envio de dados realizado pelo processador. As configurações disponíveis são: canal de comunicação, potência de saída do transceptor, mecanismos de reconhecimento de endereços, entre outras configurações que o transceptor disponibiliza ao usuário.

Este módulo também é responsável pela verificação do estado do transceptor.

`frame_tx`

O `frame_tx` é responsável pelo envio do datagrama IEEE802.15.4 através do registrador TXFIFO do transceptor. Este módulo é responsável pela ordenação dos campos do datagrama da forma correta.

A transmissão de dados através do módulo de comunicação sem fio é realizada conforme o diagrama de estados representado na Figura 39. Inicialmente, realiza-se a transferência dos dados que compõem o datagrama para o registrador TXFIFO utilizando o `spi_com`. Verificando-se o sinal CCA, que indica se o meio está livre para comunicação, inicia-se a transmissão dos dados por rádio-freqüência ao ser enviado um comando strobe STXON ao transceptor. Ao finalizar a transmissão do datagrama, o módulo retorna ao estado de espera por novos dados.

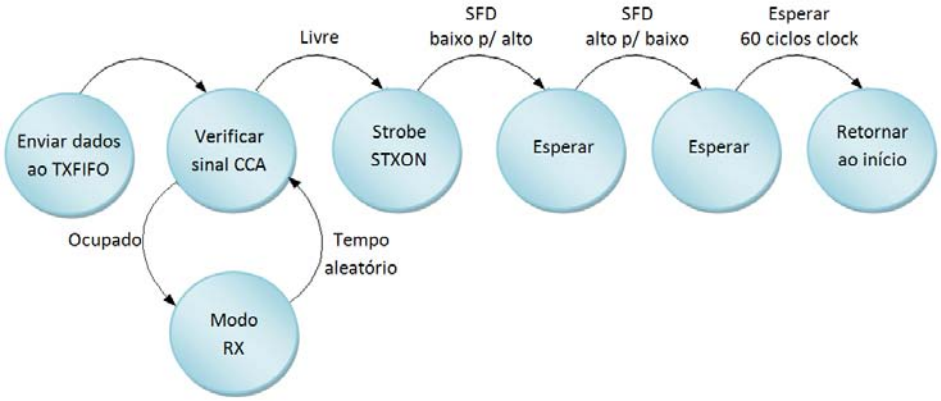


Figura 39: Diagrama de estados da transmissão de dados.

`frame_rx`

O `frame_rx` é responsável pelo recebimento do datagrama IEEE802.15.4 através do registrador RXFIFO do transceptor. Este módulo realiza a ordenação dos campos do datagrama da forma correta para posterior envio ao `modulo_wireless` para serem encaminhados ao processador.

A recepção de dados é indicada ao módulo pelo transceptor através dos sinais FIFO e FIFOP. Quando o sinal FIFO está em nível lógico alto, indica-se que existem dados a receber no registrador RXFIFO. O mesmo ocorre com o sinal FIFOP, porém este pode ser configurado para que somente apresente-se em sinal lógico alto ao receber um valor pré-configurado de dados. Posteriormente, o usuário realiza um acesso de leitura dos dados que estão localizados no registrador RXFIFO e verifica se o código de checagem de erro confere com os dados recebidos, caso esta operação não tenha sido configurada para ser realizada automaticamente pelo módulo. Ao final, o módulo retorna ao estado de espera por novos dados. O diagrama de estados que representa a recepção de dados está representado na Figura 40.

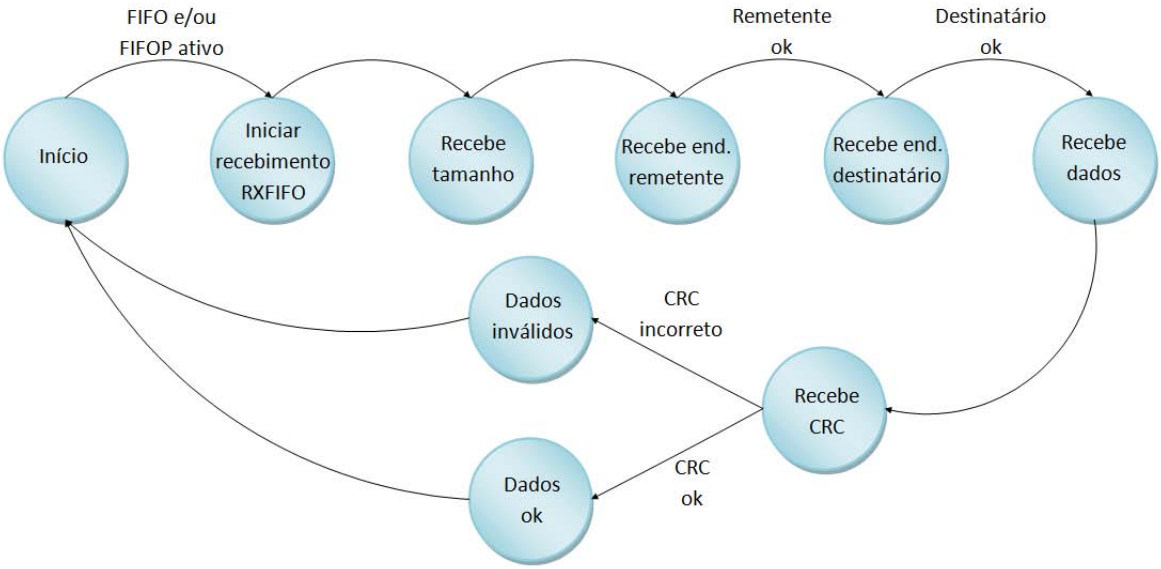


Figura 40: Diagrama de estados da recepção de dados.

`spi_com`

O `spi_com` realiza a comunicação com o transceptor CC2420 através do protocolo SPI. O SPI é um protocolo de comunicação síncrono e opera no modo full-duplex, sendo composto por quatro sinais: SCLK (sinal de clock, gerado pelo mestre (master)), MOSI (sinal para envio de dados do mestre para o escravo (slave)), MISO (sinal para envio de dados do escravo para o mestre) e CSn (sinal gerado pelo mestre que aciona a comunicação com um dos escravos). O protocolo SPI não permite o endereçamento, a comunicação só pode ser feita entre dois pontos, sendo um deles o mestre e outro o escravo. Utiliza-se o sinal CSn para selecionar o dispositivo com quem o mestre irá realizar a comunicação. Neste trabalho, o mestre da comunicação será o FemtoNode e o dispositivo escravo será o transceptor CC2420.

A Figura 41 apresenta o diagrama de tempos para realização da comunicação com o módulo. Através da interface SPI é possível realizar a configuração, leitura e escrita dos registradores disponíveis no transceptor. A comunicação é realizada com o envio e o recebimento, simultâneo, de 24 bits, sendo enviados os bits mais significativos primeiramente. Os 8 bits mais significativos enviados são utilizados para representação do tipo de acesso (leitura ou escrita) e o endereço do registrador a ser acessado. Os 16 bits menos significativos são utilizados para representação dos dados. Simultaneamente, recebe-se o status do módulo CC2420 nos 8 bits mais significativos e os dados nos demais bits, caso seja uma operação de leitura.

O código dos arquivos VHDL que constituem o Módulo Wireless estão contidos no Apêndice B.

4.2 API de comunicação

Com o objetivo de oferecer uma maior facilidade para a utilização dos recursos da arquitetura do FemtoNode pelos desenvolvedores de aplicações, uma API foi desenvolvida denominada de API-Wireless. As APIs podem ser definidas como um conjunto de classes com funções e procedimentos públicos que permitem abstrair e facilitar a pro-

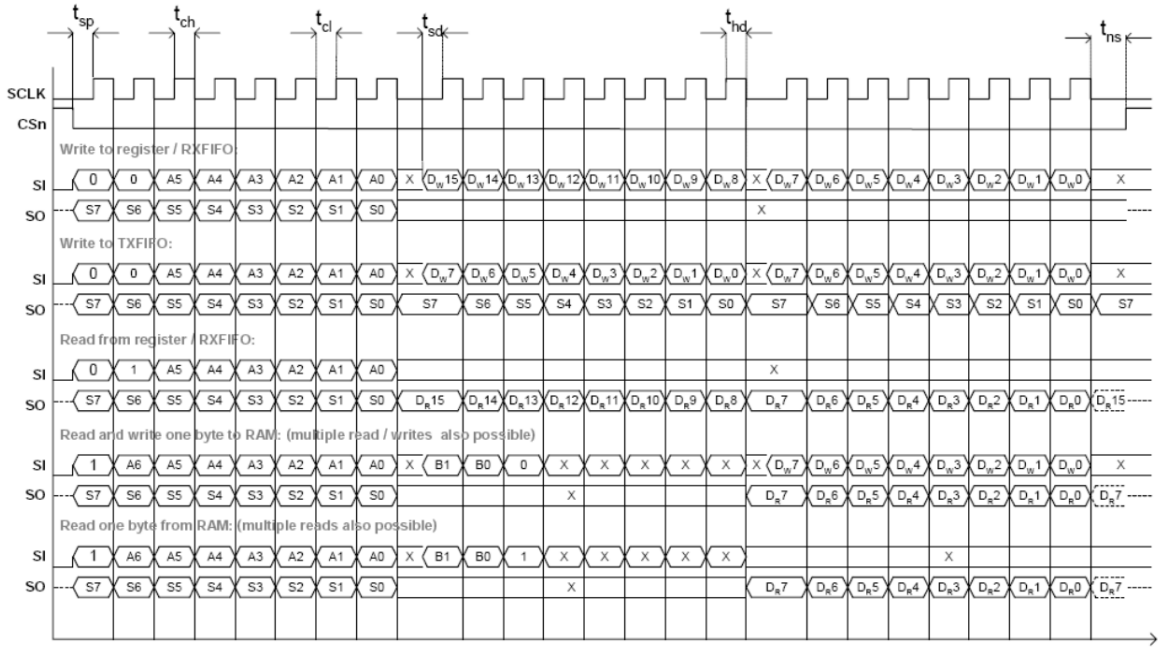


Figura 41: Protocolo de acesso aos registradores do módulo CC2420 (INSTRUMENTS, 2007).

gramação ao desenvolvedor (MICROSYSTEMS, 2009a). Assim, a API-Wireless torna transparente o meio de comunicação entre os nós-sensores para a aplicação, oferecendo de forma simplificada, através de métodos, os acessos ao módulo de comunicação sem fio.

A API-Wireless foi desenvolvida com base na API-COM (SILVA JÚNIOR, 2008; SILVA JÚNIOR et al., 2006) desenvolvida para o ambiente SASHIMI destinada ao FemtoJava. A API-COM segue o modelo OSI-ISO (TANENBAUM, 2003), sendo que cada camada presta serviço para a camada superior até chegar à aplicação. A API-COM subdividiu o protocolo de comunicação em três camadas específicas: transporte, rede e enlace. Porém, a API-Wireless utilizará somente as camadas de rede e enlace, integrando uma nova camada denominada Physical_CC2420 para acesso aos métodos de configuração do transceptor de rádio-freqüência.

Ao contrário de redes tradicionais, o uso de protocolos de transporte em RSSFs nem sempre é necessário. A maioria das aplicações admite a perda de dados, assim um mecanismo elaborado para garantia do envio de dados não é justificado (AKYILDIZ et al., 2002). Geralmente os nós enviam as informações por múltiplos caminhos para evitar retransmissões. Portanto, muitas aplicações utilizam somente as camadas física, enlace e rede para o desenvolvimento de protocolos para RSSFs.

Na Figura 42 está representado o diagrama de classes da API de comunicação onde é possível verificar as camadas e suas relações com as demais, além dos métodos da API-RTSJ que são utilizados. A seguir é realizada uma descrição das classes que compõem a API-Wireless e suas funções no modelo de comunicação do nó-sensor FemtoNode.

`Network`

Na classe `Network` estão implementados os métodos para o envio e recebimento de dados, além dos métodos destinados a utilização dos mecanismos de eventos de recepção de dados.

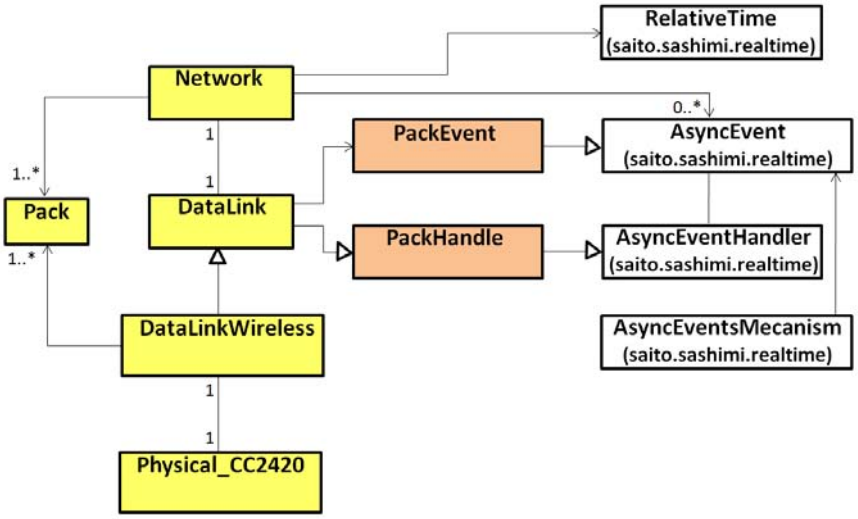


Figura 42: Diagrama de classes da API-Wireless.

A camada de rede é responsável por prover o serviço de roteamento das informações, identificando o destinatário e encontrando um caminho entre a origem e o destino desta mensagem.

`Datalink`

Na API-COM, a classe `Datalink` é abstrata para possibilitar que a API de comunicação possa ser reusada em diferentes tecnologias e padrões de rede. Estendendo-se esta classe é possível implementar classes específicas para um determinado tipo de rede. Para a comunicação wireless, foi desenvolvida a classe denominada *DataLinkWireless* com o objetivo de adaptar a API-COM ao *Módulo Wireless*. Nesta classe também está presente o método para atendimento da interrupção para recebimento de dados.

A classe *Pack* é utilizada para transportar um pacote de dados em um formato não dependente do tipo de rede utilizada. Esta classe sofreu algumas alterações para suportar datagramas do padrão IEEE802.15.4. Na classe `Datalink` são instanciados dois objetos do tipo *Pack* para armazenar as informações dos dados recebidos e as informações dos dados que devem ser enviados.

`Physical_CC2420`

Com o objetivo de fornecer métodos para a realização da configuração do transceptor de comunicação sem fio CC2420 foi implementada a classe `Physical_CC2420`. Nesta classe estão localizados os métodos para realizar a configuração dos parâmetros de potência de saída do transceptor, canal de comunicação e ativar e desativar o funcionamento do transceptor visando economia de energia.

Além disso, nesta classe localizam-se os métodos para realizar o envio e recebimento de comandos e dados ao barramento de comunicação do microcontrolador FemtoJava.

Os dispositivos de comunicação, em sistemas embarcados, consomem muita energia se comparado ao custo energético para a realização de ciclos pelo processador. Assim, a comunicação deve estar ativa somente quando necessária. Outro fator que deve ser evitado é a espera do processador causada pela comunicação enquanto aguarda mensagens

ou quando há problemas na conexão. Estes dois fatores são prejudiciais à rede de sensores, pois provocam prejuízos no desempenho dos nós da rede e também na vida útil dos mesmos. Buscando evitar o desperdício de energia, alguns mecanismos foram implementados da API-RTSJ para o tratamento de eventos assíncronos.

Utilizando mecanismos de eventos, pode-se minimizar o tempo de espera do processador no aguardo pela chegada de mensagens. Deve ser escolhido um mecanismo de tratamento de eventos dado por `AsyncEventsMechanism`. Assim, toda vez que um evento ocorrer, o método `setMsgRdyEvent()` será chamado. Dentro deste método pode-se programar uma chamada para o método `receiveMsg()` ou outra abordagem caso ocorra um evento.

A arquitetura do FemtoNode com a API-Wireless pode ser visualizado na Figura 43, onde se encontra localizado entre a aplicação e o Módulo Wireless. A aplicação pode utilizar a API para acessar os recursos necessários para a comunicação sem fio ou utilizar diretamente os recursos do microcontrolador FemtoJava.

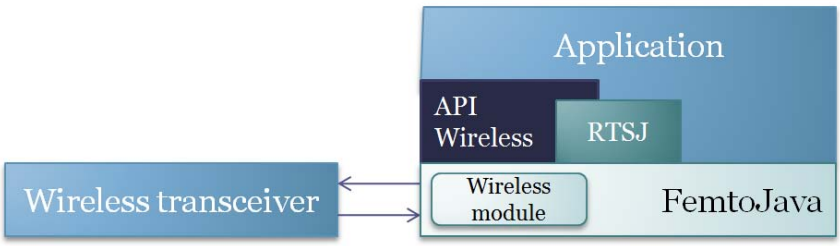


Figura 43: Arquitetura FemtoNode com a API-Wireless.

A descrição das classes e seus respectivos métodos estão contidos no Apêndice A deste trabalho.

4.3 Protótipo

O protótipo do nó-sensor FemtoNode foi mapeado para duas versões de FPGAs da família Xilinx com o auxílio da ferramenta de síntese ISE da Xilinx. As duas versões de FPGAs utilizadas neste trabalho são: Spartan 3-AN (XILINX, 2008b) e Virtex II-Pro (XILINX, 2008a). Para auxiliar no desenvolvimento, foi utilizado o kit de desenvolvimento destas FPGAs que já apresentam alguns dispositivos para auxiliar no desenvolvimento.

A família Virtex-II apresenta o modelo XC2VP30 que possui um processador PowerPC embarcado. Este modelo possui 30.000 elementos lógicos, memória RAM expansível até 2 Gbytes, e um clock de 100MHz na placa de desenvolvimento.

A família Spartan-3 possui um custo menor e apresenta um número reduzido de elementos lógicos comparado com a família Virtex. A diferenciação está na presença de memória não volátil para armazenamento de código e de um sistema de hibernação para gerenciamento do consumo de energia do sistema. A Spartan 3-AN modelo XC3S700AN possui aproximadamente 700.000 gates e 6 Kbits de memória flash e um clock de 50MHz na placa de desenvolvimento.

Estas duas versões de FPGAs apresentam características distintas que serão interessantes para verificação de implementações diferentes do nó-sensor proposto.

Como o módulo do transceptor de rádio-freqüência CC2420 apresenta um conector não presente nas placas de desenvolvimento da Xilinx, foi necessário desenvolver uma

placa para adaptar o módulo. A Figura 44 apresenta os dois protótipos do nó-sensor FemtoNode desenvolvidos nas placas Virtex-II e Spartan 3, respectivamente.

(a) (b)

Figura 44: Protótipo do FemtoNode: (a)Virtex II-Pro (b)Spartan 3-AN.

Com o objetivo de comparar a utilização dos recursos da FPGA com a síntese do FemtoNode, foram mapeados o Módulo Wireless e o FemtoNode contendo o Módulo Wireless e o RT-FemtoJava. Na Tabela 6 está representado o número de recursos utilizados pelo *Módulo Wireless* na implementação na FPGA Virtex II e na Spartan 3.

Tabela 6: Utilização de recursos do Módulo Wireless nas FPGAs.

	Virtex			**Spartan**		
Recursos	**Utilizado**	**Disponível**	**Porcentagem**	**Utilizado**	**Disponível**	**Porcentagem**
Slices	300	13.696	2%	304	5.888	5%
Flip-Flops	283	27.392	1%	285	11.776	2%
LUTs	374	27.392	1%	378	11.776	3%
IOBs	66	556	11%	66	372	17%
BUFGMUXs	3	16	18%	3	24	12%

Na Tabela 7 está representado o número de recursos utilizados pelo FemtoNode constituído pelo RT-FemtoJava e o Módulo Wireless. Por apresentar uma capacidade menor, a placa Spartan 3-AN não suporta a síntese do FemtoNode com a API-RTSJ, pois necessita uma quantidade maior de blocos de RAM do que a placa disponibiliza.

Tabela 7: Utilização de recursos do FemtoNode na FPGA Virtex II-Pro.

Recursos	**Utilizado**	**Disponível**	**Porcentagem**
Slices	1.590	13.696	11%
Flip-Flops	715	27.392	2%
LUTs	2.595	27.392	9%
IOBs	24	556	4%
BUFGMUXs	4	16	25%
RAMB16s	33	136	24%
MULT18x18s	3	136	2%

5 IMPLEMENTAÇÃO/VALIDAÇÃO DA ARQUITETURA

O uso de redes de sensores e atuadores sem fio em aplicações industriais apresentam diversos benefícios em termos de flexibilidade na instalação e manutenção dos dispositivos, suporte a monitoramento remoto das variáveis envolvidas, redução do custo e de problemas relacionados a cabos de comunicação de dados, reduzindo o perigo de quebra dos mesmos.Também reduz o problema com a variedade de conectores existentes nos mais diversos equipamentos industriais, permitindo uma portabilidade entre equipamentos (BONIVENTO; CARLONI; SANGIOVANNI-VINCENTELLI, 2006; WILLIG, 2008). Redes distribuídas de sensores sem fio em ambientes industriais beneficiam a aplicação por introduzirem adaptabilidade e flexibilidade aos processos, melhorando a qualidade e a confiabilidade dos mesmos.

A possibilidade de comunicação sem a necessidade de instalação de cabos, facilita a conexão entre os diversos processos existentes em uma indústria, como está representado na Figura 45.

As redes de sensores e atuadores sem fio podem ser utilizadas para o monitoramento e o controle de diversas variáveis existentes em processos de controle industriais como por exemplo: temperatura, pressão, umidade, vibração, entre outros (JEONG; NOF, 2008). A possibilidade da implantação de um monitoramento remoto das variáveis apresentadas, auxilia no controle e na tomada de decisões em eventuais emergências. Outra vantagem é a possibilidade da colocação de sensores e atuadores em locais de difícil acesso, onde sensores com cabos não seriam possíveis de serem instalados.

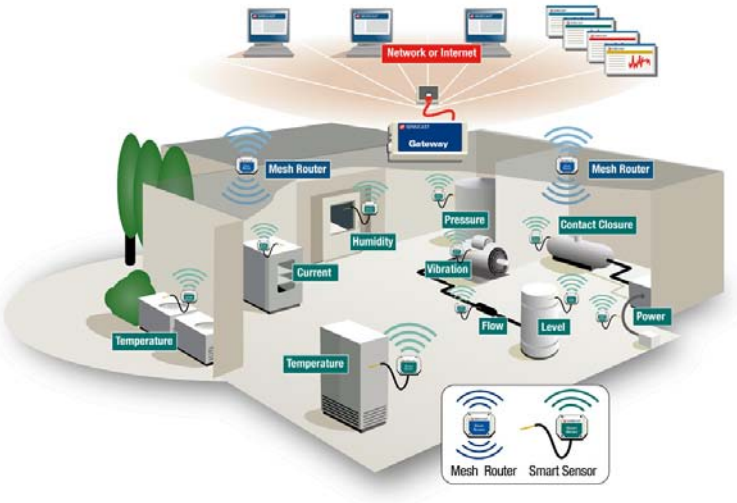


Figura 45: Aplicações industriais de redes de sensores heterogêneas (SENSICAST, 2009).

A coexistência entre os diversos protocolos de comunicação sem fio existentes e a implementação de controles de acesso ao meio faz com que as redes de sensores tornem-se confiáveis e seguras em relação a interferências provenientes de outras redes (SIKORA; GROZA, 2005).

Com base nos benefícios apresentados da utilização de RSSFs em aplicações industriais, foi proposto um estudo de caso utilizando uma aplicação de um processo industrial. Este estudo de caso servirá para a validação da arquitetura proposta neste trabalho utilizando-se da implementação de uma Rede de Sensores e Atuadores Heterogênea. Esta rede é composta por nós-sensores e nós-atuadores com plataformas que apresentam características distintas, tanto de hardware como de software, porém que apresentam suporte a aplicações Java.

A rede proposta para a validação é composta por dois tipos de nós-sensores: FemtoNode e SunSPOT, como representado na Figura 46.

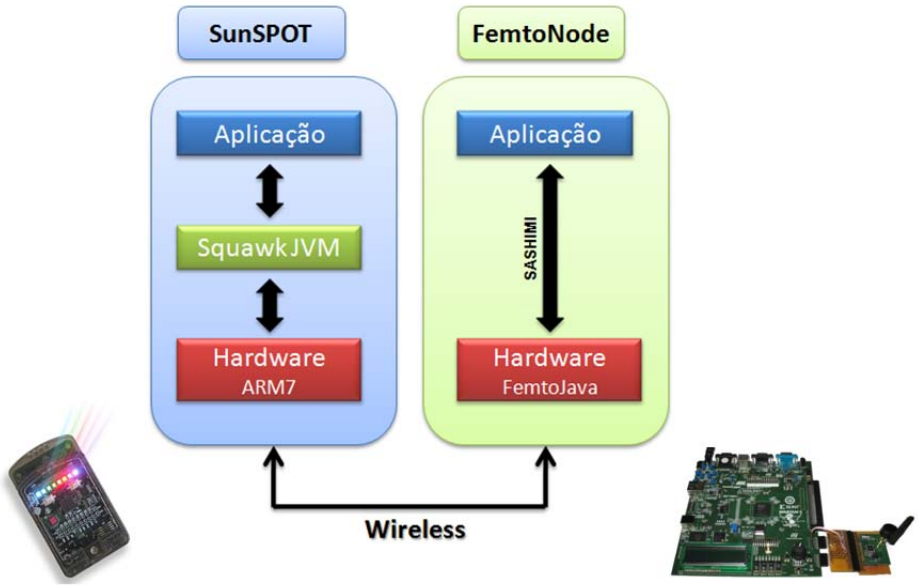


Figura 46: Arquitetura de validação com plataformas diferentes.

O dispositivo SunSPOT (Sun Small Programmable Object Technology) (MICROSYSTEMS, 2009b) é um equipamento desenvolvido pela Sun Microsystems Laboratories para o desenvolvimento de aplicações Java para Rede de Sensores sem Fio. Estes dispositivos não possuem um sistema operacional, sendo que utilizam uma máquina virtual denominada "Squawk" para realizar o suporte a programação Java neste sistema embarcado. A máquina virtual Squawk foi quase toda desenvolvida com linguagem Java para dispositivos com poucos recursos de memória e computacionais, como sistemas embarcados, além de possibilitar a execução de mais de uma tarefa (thread) concorrentemente.

A arquitetura dos SunSPOTs é modular e composta pelos seguintes módulos: bateria, placa mãe e placa de sensores. Esta modularidade pode ser visualizada na Figura 47. A placa mãe é composta por um processador ARM920T, da família ARM9, de 32 bits e 180 MHz de freqüência. O SunSPOT possui 4 Mbytes de memória Flash e 512 Kbytes de memória RAM. Nesta placa também está localizado o rádio transceptor CC2420 com sua antena acoplada à placa para realizar a transferência de dados com outros dispositivos da rede.

A placa de sensores é composta pelos seguintes sensores: acelerômetro de 3 eixos,

Figura 47: Modularidade do nó-sensor SunSPOT (MICROSYSTEMS, 2009b).

sensor de luminosidade e sensor de temperatura, além de apresentar 2 botões e 8 leds tricolores para realização da interface com o usuário.

A empresa Sun Microsystems desenvolveu uma API que abrange além das classes padrões J2ME, todos os dispositivos acoplados ao SunSPOT, como: comunicação sem fio, sensores, entre outros. Assim, o acesso a estes dispositivos é facilitado e o desenvolvedor pode preocupar-se apenas no desenvolvimento da aplicação. Também foi desenvolvido um script para facilitar o processo de compilação do programa descrito em linguagem Java para o binário a ser programado na memória do dispositivo. O fluxograma da compilação do código de programa da aplicação está representado na Figura 48.

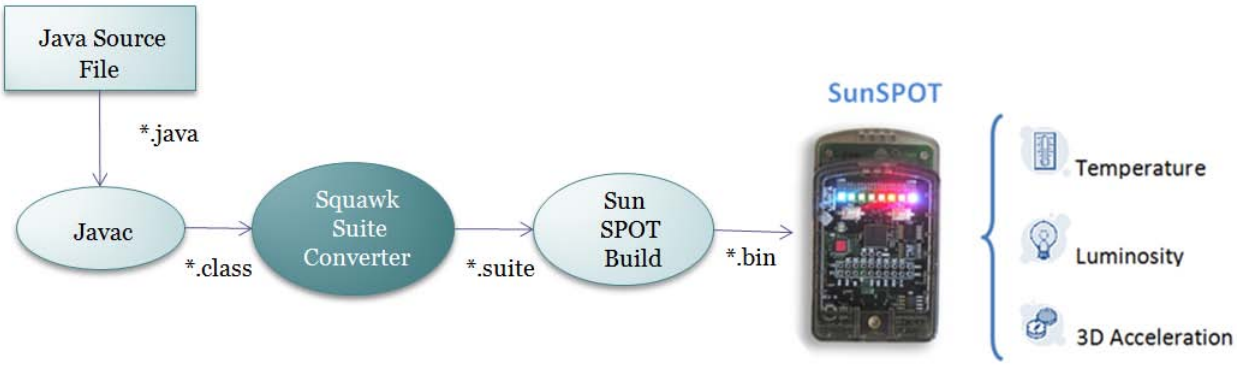


Figura 48: Fluxograma de compilação da aplicação para a plataforma SunSPOT.

5.1 Testes de comunicação FemtoNode + SunSPOT

Os testes foram realizados utilizando um controle distribuído em uma planta industrial, realizando o controle de temperatura. Este processo é composto por três elementos: sensor, controlador e atuador. O sensor é responsável por coletar a temperatura, podendo ser composto por uma rede de sensores sem fio, onde cada sensor coleta a temperatura de um dado ponto do processo e envia-o ao controlador. O controlador recebe os dados provenientes dos sensores e, através de um algoritmo de controle, envia dados para o atuador. O atuador é responsável por realizar a ação no sistema através de uma fonte de calor, neste caso, uma resistência térmica.

O controle proposto é constituído de uma Rede de Sensores Heterogênea por apresentar nós de diferentes características, como representado na Figura 49. Os SunSPOTs são responsáveis por realizar o sensoriamento da temperatura e a atuação no sistema, sendo que o FemtoNode realiza o algoritmo de controle baseado nas informações do sensor e envia-os para o atuador.

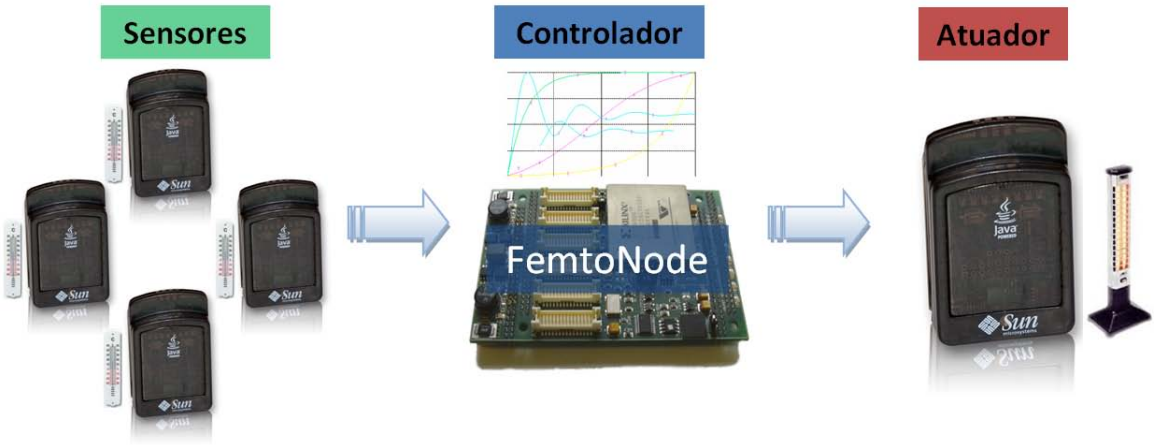


Figura 49: Rede composta por nós SunSPOTs e FemtoNode.

Com o objetivo de demonstrar a aplicação desenvolvida, a seguir será descrito e analisado o código da aplicação para as diferentes plataformas.

A implementação da classe Sensor do sistema de controle foi desenvolvida para o SunSPOT e está representada na Figura 50. A sintaxe de programação segue o padrão J2ME utilizada para sistemas embarcados com suporte a Java. Na classe *Sensor* são realizados as alocações dos objetos para interface com o sensor de temperatura e com os LEDs, além da instância de um objeto para interface com o módulo de comunicação sem fio. A aplicação inicia a partir do método `startApp()`, onde é realizada uma chamada para o método de configuração do módulo de comunicação sem fio e a inicialização de uma thread que realizará a coleta da temperatura do ambientente a cada 100 ms e, posteriormente, enviando ao Controlador. A API de comunicação utilizada para programação dos SunSPOTs foi desenvolvida pela própria Sun Microsystems e possibilita o envio, recebimento e configuração do módulo de comunicação.

```
1  import com.sun.spot.peripheral.Spot;
2  import com.sun.spot.peripheral.radio.I802_15_4_PHY;
3  import com.sun.spot.peripheral.radio.IRadioPolicyManager;
4  import com.sun.spot.peripheral.radio.RadioPacket;
5  import com.sun.spot.util.Utils;
6  import com.sun.spot.sensorboard.peripheral.*;
7  import javax.microedition.midlet.MIDlet;
8
9  public class Sensor extends MIDlet {
10
11     private ITriColorLED [] leds = EDemoBoard.getInstance().getLEDs();
12     private ITemperatureInput tempSensor = EDemoBoard.getInstance().getADCTemperature();
13     private I802_15_4_PHY radio = Spot.getInstance().getI802_15_4_PHY();
14
15     int tempC;
16     long myAdrr = 0x0000242A;
17     long controlAddr = 0x00004273;
18
19     private void setupRadio() {
20         radio.plmeSet(I802_15_4_PHY.PHY_CURRENT_CHANNEL, 26);
21         int result = radio.plmeSetTrxState(I802_15_4_PHY.RX_ON);
22         if (I802_15_4_PHY.SUCCESS != result) {
```

```
23              throw new SpotFatalException("Transicao de trx_off to rx_on falhou!");
24          }
25      }
26
27      private void envia(int temp){
28          RadioPacket rp = RadioPacket.getDataPacket();
29          rp.setSourceAddress(myAdrr);
30          rp.setDestinationAddress(controlAddr);
31          rp.setMACPayloadLength(temp.sizeof());
32          rp.setMACPayloadAt(0, temp);
33          radio.pdDataRequest(rp);
34      }
35
36      private void run()throws IOException {
37    while(true){
38      tempC = tempSensor.getCelsius();
39                  envia(tempC);
40                  Utils.sleep(100);
41          }
42      }
43
44      protected void startApp() throws MIDletStateChangeException {
45          setupRadio();
46           try {
47              run();
48          } catch (IOException ex) {
49              ex.printStackTrace();
50          }
51      }
52
53      protected void pauseApp() {
54          // Nao utilizado pela Squawk VM
55      }
56
57      protected void destroyApp(boolean unconditional) throws MIDletStateChangeException {
58      }
59  }
```

Figura 50: Código Java do nó Sensor.

Como o FemtoNode possui em sua arquitetura um microcontrolador com suporte a API-RTSJ, o desenvolvimento do controlador segue a especificação Java para sistemas de tempo real utilizando restrições temporais para o cálculo do algoritmo de controle do processo. No controlador são implementadas três classes: *Controlador*, *Controle* e *Receptor*.

A classe *Controlador* é responsável por realizar a alocação dos objetos de controle e recebimento de dados, como apresentado na Figura 51. O método `initSystem()` representa o código que será sintetizado pela ferramenta SASHIMI e nele são feitas as inicializações do sistema antes de entrar no laço que representa o tempo ocioso do sistema tempo-real embarcado. O sistema de controle possui duas threads periódicas que são alocadas na classe *Controlador*, sendo escolhida uma política de escalonamento denominada de "Earliest Deadline First" (EDF). A temporização das threads são realizadas dentro do código de programa das mesmas que será analisado a seguir. A utilização da API-Wireless para envio e recebimento de dados está representada pela alocação dos objetos *phy*, *dlink* e *net*, que representam as camadas física, enlace e rede, respectivamente.

```
1
2  import saito.sashimi.*;
3  import saito.sashimi.realtime.*;
4  import saito.sashimi.Api_Wireless.*;
5
6  public class Controlador {
7
```

```
8     public static Physical_CC2420 phy = new Physical_CC2420();
9     public static DataLinkWireless dlink = new DataLinkWireless(phy);
10    public static Network net = new Network(dlink);
11
12    public static Controle controle = new Controle(net);
13    public static Receptor receptor = new Receptor(net, controle);
14
15    public static EDFScheduler scheduler = new EDFScheduler();
16
17
18    public static void idleTask(){
19      while(!scheduler.isAllTasksFinished())
20        FemtoJava.sleep();
21    }
22
23        public static void initSystem()
24        {
25          Scheduler.setDefaultScheduler(scheduler);
26          controle.addToFeasibility();
27          receptor.addToFeasibility();
28          controle.start();
29          receptor.start();
30          scheduler.setupTimer();
31          idleTask();
32        }
33
34  //------------------------------------------
35  //          MAIN
36  //------------------------------------------
37
38        public static void main(String argv[])
39        {
40           Controlador femtoNodeControl = new Controlador();
41           femtoNodeControl.initSystem();
42        }
43
44  }
```

Figura 51: Código Java do nó Controlador - controlador.java.

A classe *Receptor* é responsável pelo recebimento dos dados de temperatura enviados pelo sensor e seu código fonte está representado na Figura 52. Nesta classe é implementada uma thread que extende a classe *RealtimeThread* para verificar periodicamente o recebimento de dados pelo módulo de comunicação sem fio. Ao receber os dados, a classe armazena na variável *temperatura* através do método `setTemp()`, e retorna a aguardar novos dados. Esta thread periódica é ativada a cada 10 ms e possui um deadline de 10 ms. O código da thread é executado dentro do método `mainTask()`, onde encontra-se um laço infinito a espera do recebimento de dados, verificando-se a variável *newMsg* da classe *Network*. Ao receber mensagem, executa-se o método `receiveMsg()` para o preenchimento de um objeto *Pack*. No final do laço, o método `waitForNextPeriod()` é executado, fazendo com que a thread aguarde um próximo período de ativação.

Não foi definido o código de tratamento de exceção provocado pela perda do deadline, por esta razão o método `exceptionTask()` não está implementado.

```
1  import saito.sashimi.*;
2  import saito.sashimi.realtime.*;
3  import saito.sashimi.Api_Wireless.*;
4
5  public class Receptor extends RealtimeThread {
6
7     private static AbsoluteTime m_taskActiveTime = new AbsoluteTime(0, 0, 0);
8     private static RelativeTime m_taskPrevProcTime = new RelativeTime(0, 0, 0);
9     private static AbsoluteTime m_ReceptorResumeTime = new AbsoluteTime(0, 0, 0);
10
```

```
11    private static RelativeTime _5_ms = new RelativeTime(0, 5, 0);
12    private static RelativeTime _10_ms = new RelativeTime(0, 10, 0);
13
14    private static PeriodicParameters relParams = new PeriodicParameters(null, //tempo
          inicial
15                                        null, //tempo final
16                                        _10_ms, //periodo
17                                        _5_ms, //custo
18                                        _10_ms); //deadline
19
20
21    public static Pack inPack = new Pack();
22    public static Network net;
23    public static Controle controle;
24    private boolean finalizar = false;
25
26
27    Receptor(Network Net, Controle Cont){
28      super(null, relParams);
29      m_ResumeTime = m_ReceptorResumeTime;
30      m_ActiveTime = m_taskActiveTime;
31      m_PreviousProcessTime = m_taskPrevProcTime;
32      net = Net;
33      controle = Cont;
34    }
35
36    public void mainTask(){
37      while(!finalizar){
38        if(net.newMsg){
39          net.receiveMsg(inPack);
40          controle.setTemp(inPack.Data);
41        }
42        waitForNextPeriod();
43      }
44      this.finish();
45    }
46
47    public void exceptionTask() {}
48    protected void initializeStack() {}
49  }
```

Figura 52: Código Java do nó Controlador - receptor.java.

A classe *Controle* implementa uma thread periódica para realizar o algoritmo de controle do processo. O período da thread é de 100ms definido pelo escalonador do sistema. Esta classe recebe da classe *Receptor* o valor de temperatura e realiza o cálculo do valor de potência a ser aplicado no sistema, a partir de um algoritmo de controle. O código do algoritmo de controle foi retirado para facilitar o entendimento dos métodos presentes na classe. Ao finalizar o cálculo da variável de controle, o controlador envia o valor ao nó-atuador do sistema preenchendo um objeto *Pack* e enviando através do método `sendMsg()` da classe *Network*. O código fonte da aplicação está representado na Figura 53.

```
1  import saito.sashimi.*;
2  import saito.sashimi.realtime.*;
3  import saito.sashimi.Api_Wireless.*;
4
5  public class Controle extends RealtimeThread {
6
7    private static final int MEU_END = 0x4273;     //Controlador End. MAC 16 bits
8  //  private static final int SENSOR_END = 0x242A; //Sensor End. MAC 16 bits
9    private static final int ATUADOR_END = 0x4273;  //Atuador End. MAC 16 bits
10
11   private static AbsoluteTime m_taskActiveTime = new AbsoluteTime(0, 0, 0);
12   private static RelativeTime m_taskPrevProcTime = new RelativeTime(0, 0, 0);
13   private static AbsoluteTime m_ControllerResumeTime = new AbsoluteTime(0, 0, 0);
```

```
14
15    private static RelativeTime _5_ms = new RelativeTime(0, 5, 0);
16    private static RelativeTime _100_ms = new RelativeTime(0, 100, 0);
17
18    private static PeriodicParameters relParams = new PeriodicParameters(null, //tempo
          inicial
19                                        null, //tempo final
20                                        _100_ms, //periodo
21                                        _5_ms, //custo
22                                        _100_ms); //deadline
23    private static RelativeTime sendTimeOut = new RelativeTime(0, 5, 0);
24
25
26    public static Pack outPack = new Pack();
27    public static Network net;
28
29    private int temperatura;
30    private int potencia;
31    public boolean finalizar = false;
32
33
34    Controle(Network Net){
35      super(null, relParams);
36      m_ResumeTime = m_ControllerResumeTime;
37      m_ActiveTime = m_taskActiveTime;
38      m_PreviousProcessTime = m_taskPrevProcTime;
39      net = Net;
40    }
41
42    public void setTemp(int temp){
43      temperatura = temp;
44    }
45
46    public int controlePID(int varSensor){
47      int varAtuador = 0;
48      //Codigo Controle PID -> varSensor - Controle - varSaida
49      return varAtuador;
50    }
51
52    public void mainTask(){
53      while(!finalizar){
54        potencia = controlePID(temperatura);
55        outPack.addrSrc_16 = MEU_END;
56        outPack.addrDest_16 = ATUADOR_END;
57        outPack.Data = potencia;
58        net.sendPack(outPack, sendTimeOut);
59        waitForNextPeriod();
60      }
61      this.finish();
62    }
63
64    public void exceptionTask() {}
65    protected void initializeStack() {}
66
67  }
```

Figura 53: Código Java do nó Controlador - controle.java.

Assim como descrito anteriormente para a implementação da classe Sensor, foi realizada a implementação da classe *Atuador* seguindo a sintaxe para o dispositivo SunSPOT. Na classe *Atuador* são realizadas as alocações dos objetos utilizados para realizar a interface com o módulo de comunicação sem fio e os LEDs. A variável de atuação é visualizada com uma chamada ao método `System.out.println()`, que neste caso enviará os valores ao PC através da porta de comunicação USB. O Atuador aguarda o recebimento de dados proveniente do Controlador e atua no processo conforme o valor da variável de potência recebida. Na Figura 54 está representado o código da implementação

da classe Atuador.

```
1  import com.sun.spot.peripheral.Spot;
2  import com.sun.spot.peripheral.radio.I802_15_4_PHY;
3  import com.sun.spot.peripheral.radio.IRadioPolicyManager;
4  import com.sun.spot.peripheral.radio.RadioPacket;
5  import com.sun.spot.util.Utils;
6  import com.sun.spot.sensorboard.peripheral.*;
7  import javax.microedition.midlet.MIDlet;
8
9  public class Atuador extends MIDlet {
10
11     private ITriColorLED [] leds = EDemoBoard.getInstance().getLEDs();
12
13     I802_15_4_PHY radio = Spot.getInstance().getI802_15_4_PHY();
14     RadioPacket rp = RadioPacket.getDataPacket();
15
16     private void setupRadio() {
17         radio.plmeSet(I802_15_4_PHY.PHY_CURRENT_CHANNEL, 26);
18         int result = radio.plmeSetTrxState(I802_15_4_PHY.RX_ON);
19         if (I802_15_4_PHY.SUCCESS != result) {
20             throw new SpotFatalException("Transicao de trx_off to rx_on falhou!");
21         }
22     }
23
24     private int recebe() {
25         radio.pdDataIndication(rp);
26         rp.decodeFrameControl();
27         return rp.getMACPayloadBigEndShortAt(0);
28     }
29
30     private void atuar(int potencia){
31         System.out.println("Valor de atuação: " + potencia);
32     }
33
34     private void run()throws IOException {
35              while(true){
36                 int potencia = recebe();
37                 atuar(potencia);
38             }
39     }
40
41     protected void startApp() throws MIDletStateChangeException {
42         setupRadio();
43          try {
44             run();
45         } catch (IOException ex) {
46             ex.printStackTrace();
47         }
48
49     }
50
51     protected void pauseApp() {
52         // Nao utilizado pela Squawk VM
53     }
54
55
56     protected void destroyApp(boolean unconditional) throws MIDletStateChangeException {
57         for (int i = 0; i < 8; i++) {
58             leds[i].setOff();
59         }
60     }
61 }
```

Figura 54: Código Java do nó Atuador.

O código da aplicação foi gerado utilizando os compiladores referentes aos diferentes dispositivos. No caso da plataforma FemtoNode foi utilizada a ferramenta SASHIMI para compilar o programa e gerar o código referente a memória ROM e RAM do FemtoJava,

para posterior síntese em uma arquitetura reconfigurável. Para a plataforma SunSPOT, foi utilizado um compilador para geração dos arquivos binários para o microcontrolador ARM7.

6 CONCLUSÃO

Com o aumento da complexidade de aplicações para Rede de Sensores sem Fio, plataformas que possibilitam a reconfiguração de sua lógica combinacional são importantes e possuem uma grande contribuição para o desenvolvimento de novas arquiteturas para nós-sensores.

A arquitetura desenvolvida neste trabalho para nós-sensores, denominada FemtoNode, introduz flexibilidade em sua arquitetura com a utilização de uma arquitetura reconfigurável, buscando otimizar a utilização de área por parte do nó-sensor.

O FemtoNode utiliza um microcontrolador *softcore* em sua arquitetura, a grande vantagem está no fato de que seu código pode ser adaptado às necessidades da aplicação e somente o hardware necessário é sintetizado no FPGA, reduzindo assim a quantidade de lógica necessária e o consumo de energia.

A utilização da linguagem Java possibilita uma maior facilidade na especificação da aplicação e no reuso de sua lógica programada no desenvolvimento de outras aplicações, explorando os conceitos de orientação a objetos. Como a linguagem Java especifica a utilização de uma máquina virtual entre o hardware e a aplicação, proporciona-se a portabilidade da aplicação para outras plataformas que implementem uma máquina virtual. Assim, Redes de Sensores Heterogêneas que possuem diversas plataformas de hardware em uma mesma rede podem utilizar a portabilidade desta linguagem para resolver a incompatibilidade de recursos.

A comunicação sem fio é muito importante em Rede de Sensores por possibilitar a troca de informações entre os nós de uma forma eficiente e flexível, podendo os nós-sensores serem móveis ou estáticos. Porém, este tipo de comunicação consome uma elevada quantidade de energia e a escolha correta do protocolo de comunicação a ser utilizado é muito importante para o desempenho da rede. Assim, a escolha do protocolo IEEE802.15.4 para o FemtoNode foi adequada por apresentar um baixo consumo de energia e uma reduzida taxa de transferência de dados, ideal para aplicações de RSSFs onde não há um volume grande de dados a serem transmitidos.

A transparência dos mecanismos de comunicação para o desenvolvedor da aplicação é um fator muito importante e que foi contemplado neste trabalho com o desenvolvimento da API-Wireless. A API-Wireless apresenta recursos para o programador realizar o envio e recebimento de dados, assim como, a configuração do módulo de comunicação sem fio.

Para finalizar as conclusões deste trabalho são realizadas algumas considerações sobre trabalhos futuros utilizando o assunto abordado neste trabalho:

- Os requisitos das RSSFs, muitas vezes, apresentam uma complexidade no desenvolvimento da aplicação e a utilização de uma metodologia baseada em modelos pode auxiliar no projeto. Estas metodologias têm como objetivo abstrair o conhecimento

da plataforma de hardware e auxiliar na caracterização do sistema. A SensorML (BOTTS; ROBIN, 2007) propõe uma metodologia para descrição de sistemas sensores podendo ser utilizada para o desenvolvimento de aplicações em rede de sensores. Assim, o desenvolvimento de uma ferramenta de geração de código para plataformas de nós-sensores a partir de modelos SensorML auxiliaria o desenvolvimento de aplicações para rede de sensores (ALLGAYER; PEREIRA, 2008b).

- O ambiente onde as RSSFs encontram-se inseridas podem apresentar características e requisitos que se alteram dinamicamente. Assim, nós-sensores que possuem recursos para dinamicamente alterar suas características de projeto, com base na adaptação, podem apresentar uma melhor contribuição à rede, buscando atender estes novos requisitos do ambiente.

- Protocolos baseados em comunicação através de slots de tempo podem apresentar uma maior previsibilidade em comunicações sem fio, onde não é possível obter garantia de restrição temporal.

REFERÊNCIAS

ABNOUS, A. *et al*. Evaluation of a low-power reconfigurable DSP architecture. In: RECONFIGURABLE ARCHITECTURES WORKSHOP (RAW-98), 5., 1998, Orlando, Florida, USA. **Proceedings...** Berlin: Springer, 1998. p.55–60.

AIBE, N.; YASUNAGA, M. Reconfigurable I/O interface for mobile equipments. In: IEEE INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE TECHNOLOGY, 2004, Brisbane, Australia. **Proceedings...** [S.l.: s.n.], 2004. p.359–362.

AKKAYA, K.; YOUNIS, M. A survey on routing protocols for wireless sensor networks. **Ad Hoc Networks**, [S.l.], v.3, p.325–349, 2005.

AKYILDIZ, I. F. *et al*. A survey on sensor networks. **IEEE Communications Magazine**, [S.l.], v.40, n.8, p.102–114, Aug. 2002.

AL-KARAKI, J. N.; KAMAL, A. E. Routing techniques in wireless sensor networks: a survey. **IEEE Wireless Communications**, [S.l.], v.11, n.6, p.6–28, Dec. 2004.

ALEMDAR, A.; IBNKAHLA, M. Wireless sensor networks: applications and challenges. In: INTERNATIONAL SYMPOSIUM ON SIGNAL PROCESSING AND ITS APPLICATIONS - ISSPA-2007, 9., 2007, Sharjah, United Arab Emirates. **Proceedings...** New York: IEEE, 2007. p.1–6.

ALLGAYER, R. S. **Arquitetura Reconfigurável em Rede de Sensores Sem Fio**. 2008a. Trabalho Individual — Programa de Pós-Graduação em Engenharia Elétrica, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil.

ALLGAYER, R. S.; PEREIRA, C. E. Síntese de rede de sensores sem fio, reconfiguráveis, a partir de sensorML. In: WORKSHOP ON SENSOR NETWORKS AND APPLICATIONS - WSENA'08, 2008b, Gramado, Brasil. **Proceedings...** [S.l.: s.n.], 2008b.

ARAMPATZIS, T.; LYGEROS, J.; MANESIS, S. A survey of applications of wireless sensors and wireless sensor networks. In: INTERNATIONAL SYMPOSIUM ON CONTROL AND AUTOMATION INTELLIGENT CONTROL, 2005, Limassol, Cyprus. **Proceedings...** New York: IEEE, 2005. p.719–724.

ASHENDEN, P. J. **The VHDL Cookbook**. 1rd.ed. Adelaide, Australia: University of Adelaide, Australia, 1990. 111p.

BARONTI, P. *et al*. Wireless sensor networks: a survey on the state of the art and the 802.15.4 and zigbee standards. **Computer Communication**, Newton, MA, USA, v.30, n.7, p.1655–1695, 2007.

BECK, A. C. S.; CARRO, L. A VLIW low power Java processor for embedded applications. In: SYMPOSIUM ON INTEGRATED CIRCUITS AND SYSTEM DESIGN - SBCCI'04, 17., 2004, Pernambuco, Brazil. **Proceedings...** New York: ACM, 2004. p.157–162.

BONIVENTO, A.; CARLONI, L. P.; SANGIOVANNI-VINCENTELLI, A. Platform-based design of wireless sensor networks for industrial applications. In: CONFERENCE ON DESIGN, AUTOMATION AND TEST IN EUROPE - DATE'06, 2006, Munich, Germany. **Proceedings...** Leuven: European Design and Automation Association, 2006. p.1103–1107.

BOTTS, M.; ROBIN, A. **Sensor Model Language (SensorML) Implementation Specification**. [S.l.]: Open Geospatial Consortium Inc., 2007.

BURNS, J. *et al*. A dynamic reconfiguration run-time system. In: IEEE SYMPOSIUM ON FPGA-BASED CUSTOM COMPUTING MACHINES - FCCM'97, 5., 1997. **Proceedings...** Washington: IEEE Computer Society, 1997. p.66.

CALDAS, R. B. *et al*. Low power/high performance self-adapting sensor node architecture. In: IEEE CONFERENCE ON EMERGING TECHNOLOGIES AND FACTORY AUTOMATION - ETFA'05, 10., 2005, Catania, Italy. **Proceedings...** New York: IEEE, 2005. v.2, p.976.

CALLAWAY, E. *et al*. Home networking with IEEE 802.15.4: a developing standard for low-rate wireless personal area networks. **IEEE Communications Magazine**, [S.l.], v.40, n.8, p.70–77, Aug. 2002.

CARNAHAN, L. **Report from the Requirements Group for Real-Time Extensions for the Java Platform**. Disponível em: <http://www.itl.nist.gov/div897/ctg/real-time/rtj-final-draft.pdf/>. Acesso em: dezembro 2008.

CARRO, L. **Projeto e Prototipação de Sistemas Digitais**. 1.ed. Porto Alegre, Brasil: Universidade Federal do Rio Grande do Sul - UFRGS, 2001.

CHANDRAKASAN, A. *et al*. Design considerations for distributed microsensor systems. In: IEEE CUSTOM INTEGRATED CIRCUITS - CICC'99, 1999, San Diego, USA. **Proceedings...** New York: IEEE, 1999. p.279–286.

COMMURI, S.; TADIGOTLA, V. Dynamic data aggregation in wireless sensor networks. In: INTERNATIONAL SYMPOSIUM ON INTELLIGENT CONTROL - ISIC'07, 22., 2007, Suntec City, Singapore. **Proceedings...** New York: IEEE, 2007. p.1–6.

COMPTON, K.; HAUCK, S. Reconfigurable computing: a survey of systems and software. **ACM Comput. Surv.**, New York, NY, USA, v.34, n.2, p.171–210, June 2002.

DEMIRKOL, I.; ERSOY, C.; ALAGOZ, F. MAC protocols for wireless sensor networks: a survey. **IEEE Communications Magazine**, [S.l.], v.44, n.4, p.115–121, abril 2006.

DJENOURI, D.; KHELLADI, L.; BADACHE, A. N. A survey of security issues in mobile ad hoc and sensor networks. **IEEE Communications Surveys and Tutorials**, [S.l.], v.7, n.4, p.2–28, Apr. 2005.

GAISLER, J. **The LEON-2 Processor User's Manual**. [S.l.]: Gaisler Research, 2004.

GARCIA, P. *et al*. An overview of reconfigurable hardware in embedded systems. **EURASIP J. Embedded System**, [S.l.], n.1, p.13–13, 2006.

GOSLING, J.; BOLLELLA, G. **The Real-Time Specification for Java**. Boston, MA, USA: Addison-Wesley Longman Publishing Co. Inc., 2000.

GÖTZ, M.; RETTBERG, A.; PEREIRA, C. E. Towards run-time partitioning of a real time operating system for reconfigurable systems on chip. In: INTERNATIONAL EMBEDDED SYSTEMS SYMPOSIUM - IESS'05, 2005, Manaus, Brazil. **Proceedings...** [S.l.: s.n.], 2005. p.15–17.

GRAY, A. A. *et al*. Object-oriented reconfigurable processing for wireless networks. In: IEEE INTERNATIONAL CONFERENCE ON COMMUNICATIONS - ICC'02, 2002, New York, NY, USA. **Proceedings...** New York: IEEE, 2002. v.1, p.497–501.

HALKES, G. P.; DAM, T. van; LANGENDOEN, K. G. Comparing Energy-Saving MAC Protocols for Wireless Sensor Networks. **Mobile Networks and Applications**, [S.l.], v.10, n.5, p.783–791, Sept. 2005.

HE, T. *et al*. Energy-efficient surveillance system using wireless sensor networks. In: INTERNATIONAL CONFERENCE ON MOBILE SYSTEMS, APPLICATIONS, AND SERVICES - MOBISYS'04, 2., 2004, Boston, MA, USA. **Proceedings...** New York: ACM, 2004. p.270–283.

HINKELMANN, H.; ZIPF, P.; GLESNER, M. A domain-specific dynamically reconfigurable hardware platform for wireless sensor networks. In: INTERNATIONAL CONFERENCE ON FIELD-PROGRAMMABLE TECHNOLOGY - ICFPT'07, 2007, Kokurakita, Kitakyushu, Japan. **Proceedings...** [S.l.: s.n.], 2007. p.313–316.

HOWITT, I.; GUTIERREZ, J. A. IEEE 802.15.4 low rate - wireless personal area network coexistence issues. In: IEEE WIRELESS COMMUNICATIONS AND NETWORKING CONFERENCE - WCNC'03, 2003, New Orleans, Louisiana, USA. **Proceedings...** New York: IEEE, 2003. v.3, p.1481–1486.

IEEE802.15.4. **Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (LR-WPANs)**. [S.l.: s.n.], 2003.

III, T. O. W.; TUMMALA, M.; MCEACHEN, J. Distributed medium access control with flow-based priority for cooperative multi-hop wireless sensor networks. In: ANNUAL HAWAII INTERNATIONAL CONFERENCE ON SYSTEM SCIENCES - HICSS'08, 41., 2008, Honolulu, Hawaii, USA. **Proceedings...** Washington: IEEE Computer Society, 2008. p.493.

INFRARED DATA ASSOCIATION. **Página da associação Infrared Data Association (IrDA)**. Disponível em: <http://www.irda.org/>. Acesso em: abril 2008.

INSTRUMENTS, T. **2.4GHz IEEE802.15.4 / ZigBee-ready RF Transceiver**. [S.l.]: Texas Instruments, 2007. Disponível em: <http://focus.ti.com/lit/ds/symlink/cc2420.pdf/>. Acesso em: dezembro 2008.

ITO, S. A.; CARRO, L.; JACOBI, R. P. Making Java work for microcontroller applications. **IEEE Design and Test of Computers**, Los Alamitos, CA, USA, v.18, n.5, p.100–110, 2001.

JAFARI, R. *et al*. Wireless sensor networks for health monitoring. In: ANNUAL INTERNATIONAL CONFERENCE ON MOBILE AND UBIQUITOUS SYSTEMS: NETWORKING AND SERVICES - MOBIQUITOUS'05, 2., 2005, San Diego, CA, USA. **Proceedings...** [S.l.: s.n.], 2005. p.479–481.

JEONG, W.; NOF, S. Y. Performance evaluation of wireless sensor network protocols for industrial applications. **Intelligent Manufacturing**, [S.l.], v.19, n.3, June 2008.

JOVANOV, E. *et al*. Reconfigurable intelligent sensors for health monitoring: a case study of pulse oximeter sensor. In: ANNUAL INTERNATIONAL CONFERENCE OF THE IEEE ENGINEERING IN MEDICINE AND BIOLOGY SOCIETY - IEMBS'04, 26., 2004, San Francisco, CA, USA. **Proceedings...** [S.l.: s.n.], 2004. v.2, p.4759–4762.

JÚNIOR, F. L. C. **Nó Sensor com Arquitetura Reconfigurável para Redes de Sensores Sem Fio**. 2004. Dissertação (Mestrado em Ciência da Computação) — Instituto de Ciências Exatas, Universidade Federal de Minas Gerais, Belo Horizonte, Brasil.

KAHN, J. M.; BARRY, J. R. Wireless infrared communications. **Proceedings of the IEEE**, [S.l.], v.85, n.2, p.265–298, Feb. 1997.

KHEMAPECH, I.; DUNCAN, I.; MILLER, A. A survey of wireless sensor networks technology. In: ANNUAL POSTGRADUATE SYMPOSIUM ON THE CONVERGENCE OF TELECOMMUNICATIONS, NETWORKING AND BROADCASTING, 6., 2005, Liverpool, UK. **Proceedings...** United Kingdom: EPSRC, 2005.

KOHVAKKA, M. *et al*. High-performance multi-radio WSN platform. In: MULTI-HOP AD HOC NETWORKS: FROM THEORY TO REALITY - REALMAN'06, 2., 2006, Florence, Italy. **Proceedings...** New York: ACM, 2006. p.95–97.

KOUSHANFAR, F.; POTKONJAK, M.; SANGIOVANNI-VINCENTELL, A. Fault tolerance techniques for wireless ad hoc sensor networks. **Proceedings of IEEE Sensors**, [S.l.], v.2, p.1491–1496, 2002.

LEVIS, P.; CULLER, D. Mate: a tiny virtual machine for sensor networks. **Operating Systems Review**, [S.l.], v.36, n.5, p.85–95, 2002.

LO, B. P. *et al*. Body sensor network - a wireless sensor platform for pervasive healthcare monitoring. In: INTERNATIONAL CONFERENCE ON PERVASIVE COMPUTING, 3., 2005, Munich, Germany. **Proceedings...** Berlin: Springer, 2005. p.77–80.

LUO, H.; LIU, Y.; DAS, S. K. Routing correlated data in wireless sensor networks: a survey. **IEEE Network**, [S.l.], v.21, n.6, p.40–47, Nov-Dec 2007.

LYMBEROPOULOS, D.; PRIYANTHA, N. B.; ZHAO, F. mPlatform: a reconfigurable architecture and efficient data sharing mechanism for modular sensor nodes. In: INFORMATION PROCESSING IN SENSOR NETWORKS - IPSN'07, 6., 2007, Cambridge, Massachusetts, USA. **Proceedings...** New York: ACM, 2007. p.128–137.

MAINWARING, A. *et al*. Wireless sensor networks for habitat monitoring. In: WIRELESS SENSOR NETWORKS AND APPLICATIONS - WSNA'02, 1., 2002, Atlanta, Georgia, USA. **Proceedings...** New York: ACM, 2002. p.88–97.

MARTINS, C. A. P. S. *et al*. Computação reconfigurável: conceitos, tendências e aplicações. In: JORNADA DE ATUALIZAçãO EM INFORMáTICA (JAI), 22., 2003, São Paulo, Brasil. **Proceedings...** [S.l.: s.n.], 2003. v.2, p.339–388.

MASSEY, T. *et al*. Towards reconfigurable embedded medical systems. In: JOINT WORKSHOP ON HIGH CONFIDENCE MEDICAL DEVICES, SOFTWARE, AND SYSTEMS AND MEDICAL DEVICE PLUG-AND-PLAY INTEROPERABILITY, 2007, Boston, MA, USA. **Proceedings...** [S.l.: s.n.], 2007. p.178–180.

MELODIA, T.; VURAN, M. C.; POMPILI, D. The state of the art in cross-layer design for wireless sensor networks. In: EURONGI WORKSHOPS ON WIRELESS AND MOBILITY, 2005, Como, Italy. **Proceedings...** Berlin: Springer, 2005.

MENCER, O.; MORF, M.; FLYNN, M. J. Hardware software tri-design of encryption for mobile communication units. In: INTERNATIONAL CONFERENCE ON ACOUSTICS, SPEECH AND SIGNAL PROCESSING, 1998, Seattle, USA. **Proceedings...** New York: IEEE, 1998. v.5, p.3045–3048.

MESQUITA, D. G. **Contribuições para Reconfiguração Parcial, Remota e Dinâmica de FPGAs**. 2002. Dissertação (Mestrado em Ciência da Computação) — Pontifícia Universidade Católica do Rio Grande do Sul - PUCRS, Porto Alegre, Brasil, 2002.

MICROSYSTEMS, S. **Página da empresa Sun Microsystems**. Disponível em: <http://br.sun.com/>. Acesso em: janeiro 2009.

MICROSYSTEMS, S. **SunSPOT**. Disponível em: <http://www.sunspotworld.com/>. Acesso em: janeiro 2009.

MOTOROLA. **M68HC11 reference manual**. [S.l.]: Motorola Inc., 1996.

PARK, S.; MACKENZIE, K.; JAYARAMAN, S. The wearable motherboard: a framework for personalized mobile information processing (pmip). In: DESIGN AUTOMATION CONFERENCE - DAC'02, 2002, New Orleans, LA, USA. **Proceedings...** New York: ACM, 2002. p.170–174.

PATHAN, A.-S. K.; HONG, C. S.; LEE, H.-W. Smartening the environment using wireless sensor networks in a developing country. In: INTERNATIONAL CONFERENCE ADVANCED COMMUNICATION TECHNOLOGY - ICACT'06, 2006, Phoenix Park, Korea. **Proceedings...** New York: IEEE, 2006. v.1, p.709.

PEI, Z. *et al*. Application-oriented wireless sensor network communication protocols and hardware platforms: a survey. In: IEEE INTERNATIONAL CONFERENCE ON INDUSTRIAL TECHNOLOGY - ICIT'08, 2008, Chengdu, China. **Proceedings...** [S.l.: s.n.], 2008. p.1–6.

PLESSL, C. *et al*. The case for reconfigurable hardware in wearable computing. **Personal Ubiquitous Computing**, London, UK, v.7, n.5, p.299–308, Oct. 2003.

PORTILLA, J. *et al*. A modular architecture for nodes in wireless sensor networks. **Journal of Universal Computer Science**, [S.l.], v.12, n.3, p.328–339, March 2006.

PORTILLA, J.; RIESGO, T.; CASTRO, A. de. A reconfigurable FPGA-based architecture for modular nodes in wireless sensor networks. In: SOUTHERN CONFERENCE ON PROGRAMMABLE LOGIC - SPL'07, 3., 2007, Mar del Plata, Argentina. **Proceedings...** [S.l.: s.n.], 2007. p.203–206.

RABAEY, J. *et al*. PicoRadio: ad-hoc wireless networking of ubiquitous low-energy sensor/monitor nodes. In: IEEE COMPUTER SOCIETY WORKSHOP ON VLSI, 2000, Orlando, USA. **Proceedings...** [S.l.: s.n.], 2000. p.9–12.

RABAEY, J. M. Silicon platforms for the next generation wireless systems - what role does reconfigurable hardware play? In: INTERNATIONAL CONFERENCE ON FIELD PROGRAMMABLE LOGIC AND APPLICATIONS - FPL'2000, 10., 2000a, Villach, Austria. **Proceedings...** [S.l.: s.n.], 2000a. p.277–285.

RAHIMI, M. *et al*. Cyclops: in situ image sensing and interpretation in wireless sensor networks. In: EMBEDDED NETWORKED SENSOR SYSTEMS - SENSYS'05, 3., 2005, San Diego, California, USA. **Proceedings...** New York: ACM, 2005. p.192–204.

ROMER, K.; MATTERN, F. The design space of wireless sensor networks. **IEEE Wireless Communications**, [S.l.], v.11, n.6, p.54–61, Dec. 2004.

RTEMS C User's Guide. [S.l.]: On-Line Applications Research Corporation, 2003.

SASHIMI: manual do usuário. Porto Alegre, Brasil: Universidade Federal do Rio Grande do Sul - UFRGS, 2006.

SCHWIEBERT, L.; GUPTA, S. K.; WEINMANN, J. Research challenges in wireless networks of biomedical sensors. In: MOBILE COMPUTING AND NETWORKING - MOBICOM'01, 7., 2001, Rome, Italy. **Proceedings...** New York: ACM, 2001. p.151–165.

SENSICAST. **Página da SensiCast**. Disponível em: <http://www.sensicast.com/>. Acesso em: janiero 2009.

SIKORA, A.; GROZA, V. F. Coexistence of IEEE802.15.4 with other systems in the 2.4 GHz-ISM-Band. In: IEEE INSTRUMENTATION AND MEASUREMENT TECHNOLOGY CONFERENCE - IMTC'05, 2005, Ottawa, Canada. **Proceedings...** [S.l.: s.n.], 2005. v.3, p.1786–1791.

SILVA JÚNIOR, E. T. da. **Middleware Adaptativo para Sistemas Embarcados e de Tempo-Real**. 2008. Tese (Doutorado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2008.

SILVA JÚNIOR, E. T. da *et al*. Java framework for distributed real-time embedded systems. In: IEEE INTERNATIONAL SYMPOSIUM ON OBJECT-ORIENTED REAL-TIME DISTRIBUTED COMPUTING - ISORC'06, 2006, Gyeongju, Korea. **Proceedings...** Los Alamitos: IEEE Computer Society, 2006. v.0, p.85–92.

SMITH, D. J. VHDL and Verilog compared and contrasted: plus modeled example written in vhdl, verilog and c. In: DESIGN AUTOMATION - DAC'96, 33., 1996, Las Vegas, Nevada, United States. **Proceedings...** New York: ACM, 1996. p.771–776.

SU, Z. *et al*. Security mechanisms analysis of wireless sensor networks specific routing attacks. In: INTERNATIONAL SYMPOSIUM ON PERVASIVE COMPUTING AND APPLICATIONS, 1., 2006, Urumchi, China. **Proceedings...** [S.l.: s.n.], 2006. p.579–584.

SZEWCZYK, R. *et al*. Habitat monitoring with sensor networks. **Commununication**, [S.l.], v.47, n.6, p.34–40, 2004.

TAN, Y. *et al*. Design and implementation of a Java processor. In: COMPUTERS AND DIGITAL TECHNIQUES, 2006. **Proceedings...** [S.l.: s.n.], 2006. v.153, n.1, p.20–30.

TANENBAUM, A. S. **Computer Networks**. 4.ed. New Jersey, USA: Prentice Hall, 2003.

TILAK, S.; ABU-GHAZALEH, N. B.; HEINZELMAN, W. A taxonomy of wireless micro-sensor network models. In: MOBILE COMPUTING AND COMMUNICATIONS REVIEW, 2002, Atlanta, USA. **Proceedings...** New York: ACM, 2002. v.6, n.2, p.28–36.

TODMAN, T. *et al*. Reconfigurable computing: architectures and design methods. In: IEEE COMPUTERS AND DIGITAL TECHNIQUES, 2005. **Proceedings...** [S.l.: s.n.], 2005. v.152, n.2, p.193–207.

WEHRMEISTER, M. A. **Framework Orientado a Objetos para Projeto de Hardware e Software Embarcados para Sistemas Tempo-Real**. 2005. Dissertação (Mestrado em Ciência da Computação) — Instituto de Informática, Universidade Federal do Rio Grande do Sul, Porto Alegre, Brasil, 2005.

WEHRMEISTER, M. A.; PEREIRA, C. E.; BECKER, L. B. Optimizing the generation of object-oriented real-time embedded applications based on the real-time specification for Java. In: DESIGN, AUTOMATION AND TEST IN EUROPE - DATE'06, 2006, Munich, Germany. **Proceedings...** Leuven: Belgium: European Design and Automation Association, 2006. p.806–811.

WHEELER, A. Commercial applications of wireless sensor networks using ZigBee. **IEEE Communications Magazine**, [S.l.], v.45, n.4, p.70–77, Apr. 2007.

WILDER, J. L. *et al*. Runtime hardware reconfiguration in wireless sensor networks. In: SOUTHEASTERN SYMPOSIUM ON SYSTEM THEORY - SSST'08, 40., 2008, New Orleans, Louisiana, USA. **Proceedings...** [S.l.: s.n.], 2008. p.154–158.

WILLIG, A. Recent and emerging topics in wireless industrial communications: a selection. **IEEE Transactions on Industrial Informatics**, [S.l.], v.4, n.2, p.102–124, May 2008.

XIA, F. *et al*. Wireless sensor/actuator network design for mobile control applications. **Sensors**, [S.l.], v.7, n.10, p.2157–2173, 2007.

XILINX. **Xilinx University Program Virtex-II Pro Development System**. [S.l.]: Xilinx, 2008a.

XILINX. **Spartan-3A/3AN FPGA starter kit board user guide - UG334**. [S.l.]: Xilinx, 2008b. 140p.

YANG, G. Z. **Body Sensor Networks**. [S.l.]: Berlin, Springer, 2006.

YASUURA, H. *et al*. Embedded system design using soft-core processor and valen-C. **Journal of Information Science and Engineering**, [S.l.], v.14, n.3, p.587–603, Aug. 1998.

ZIGBEE ALLIANCE. **Página da Associação Zigbee Alliance**. Disponível em: <http://www.zigbee.org>. Acesso em: abril 2008.

APÊNDICE A CLASSES E MÉTODOS DA API-WIRELESS

Este apêndice traz a descrição das classes e métodos que contém a API-Wireless.

Package saito.sashimi.Api_Wireless

Class Summary	
Physical_CC2420	
Datalink	
DatalinkWireless	
Network	
Pack	
PackEvent	
PackHandler	

Class Physical_CC2420

```
java.lang.Object
  └ saito.sashimi.API_Wireless.Physical_CC2420
```

```
public class Physical_CC2420
extends java.lang.Object
```

Method Summary

boolean	**configModule**(int regAddr, int data)
void	**flushrx**()
int	**getCommandIn**()
int	**getDataIn**()
void	**module_off**()
void	**module_on**()
void	**setChannel**(int c)
void	**setCommandOut**(int out)
void	**setDataOut**(int out)
void	**setPower**(int p)

Class Datalink

```
java.lang.Object
  └ saito.sashimi.Api_Wireless.Datalink
```

Direct Known Subclasses:

DataLinkWireless

```
public abstract class Datalink
extends java.lang.Object
```

Method Summary

protected abstract int	**addressLength**()
abstract int	**getPackLen**()
protected abstract int	**portLength**()
protected abstract void	**receivePack**()
abstract void	**resetEvent**()
protected abstract boolean	**sendPack**(Pack op, saito.sashimi.realtime.RelativeTime timeOut)
abstract void	**setEvent**()
abstract void	**setLocalMac**()

Class DataLinkWireless

```
java.lang.Object
  └ saito.sashimi.Api_Wireless.Datalink
      └ saito.sashimi.Api_Wireless.DataLinkWireless
```

All Implemented Interfaces:

saito.sashimi.IntrInterface

```
public class DataLinkWireless
extends saito.sashimi.Api_Wireless.Datalink
implements saito.sashimi.IntrInterface
```

Method Summary

protected int	**addressLength**()
int	**getPackLen**()
void	**int0Method**()
void	**int1Method**()
protected boolean	**packListen**()
protected int	**portLength**()
protected void	**receivePack**()
void	**resetEvent**()
protected boolean	**sendPack**(Pack op, saito.sashimi.realtime.RelativeTime timeOut)
void	**setEvent**()
void	**setLocalMac**()
void	**spiMethod**()

Class Network

```
java.lang.Object
  └ saito.sashimi.Api_Wireless.Network
```

```
public class Network
extends java.lang.Object
```

Method Summary

protected int	**addressLength**()
int	**getPackLen**()
boolean	**messageReady**(int con)
boolean	**receiveMsg**(Pack ip)
boolean	**sendMsg**(Pack p, saito.sashimi.realtime.RelativeTime tim)
void	**setBroadcast**(boolean b)
void	**setDataLink**(Datalink dl)
void	**setEventStatus**(boolean stat)
void	**setMsgRdyEvent**(saito.sashimi.realtime.AsyncEvent event)
void	**setResquestEvent**(saito.sashimi.realtime.AsyncEvent event)
private void	**solveData**()

Class Pack

java.lang.Object
  └ **saito.sashimi.Api_Wireless.Pack**

---

public class **Pack**
extends java.lang.Object

Method Summary

int	**getAddrDest**()
int	**getAddrSrc**()
int	**getData**()
int	**getPanAddr**()
int	**getType**()
void	**setAddrDest**(int addr)
void	**setAddrSrc**(int addr)
void	**setData**(int data)
void	**setPanAddr**(int pan)
void	**setRSSI**(int rssi)
void	**setType**(int type)

Class PackEvent

```
java.lang.Object
  └ saito.sashimi.realtime.AsyncEvent
      └ saito.sashimi.Api_Wireless.PackEvent
```

```
public class PackEvent
extends saito.sashimi.realtime.AsyncEvent
```

Methods inherited from class saito.sashimi.realtime.AsyncEvent
addHandler, fire, handledBy, removeHandler, setHandler

Class PackHandler

```
java.lang.Object
  └ saito.sashimi.realtime.AsyncEventHandler
      └ saito.sashimi.Api_Wireless.PackHandler
```

```
public class PackHandler
extends saito.sashimi.realtime.AsyncEventHandler
```

Method Summary

protected void	**handleAsyncEvent**()
protected void	**init**(Datalink dlink)

Methods inherited from class saito.sashimi.realtime.AsyncEventHandler
attachAsyncEvent, deattachAsyncEvent, hasAsyncEventAttached

APÊNDICE B CÓDIGO FONTE VHDL DO MÓDULO WIRELESS

Este apêndice traz o código VHDL do Módulo Wireless do nó-sensor FemtoNode.

modulo_wireless.vhd

```
1  ----------------------------------------------
2  -- Company:       UFRGS - Univ. Fed. do RS
3  -- Engineer:      Rodrigo Schmidt Allgayer
4  --
5  -- Module Name:    modulo_wireless - Behavioral
6  -- Project Name:   FemtoNode
7  ----------------------------------------------
8
9  library IEEE;
10 use IEEE.STD_LOGIC_1164.ALL;
11 use IEEE.STD_LOGIC_ARITH.ALL;
12 use IEEE.STD_LOGIC_UNSIGNED.ALL;
13
14 library work;
15   use work.fj_types.all;
16   use work.modulo_wireless_types.all;
17   use work.spi_com_package.all;
18   use work.divisor_clock_package.all;
19   use work.lcd_package.all;
20
21 entity modulo_wireless is
22     Port (
23     --Interface com o processador
24   system_clk   : in STD_LOGIC; -- clock do processador
25   system_rst   : in STD_LOGIC; -- reset do sistema
26     --Portas para interface com processador
27   uc_data_in   : in STD_LOGIC_VECTOR(UC_DATA_WIDTH-1 downto 0); --Dados de entrada (saida do processador)
28   uc_data_out  : out STD_LOGIC_VECTOR(UC_DATA_WIDTH-1 downto 0); --Dados de saída (entrada no processador)
29   uc_addr_in   : in STD_LOGIC_VECTOR (UC_ADDR_WIDTH-1 downto 0); -- Endereçamento do uC
30   uc_rw_en     : in std_logic;                          -- enable do processador
31     --Interrupção para interface com processador
32   int1_fj : out STD_LOGIC :='0'; --Interrupção 0 do processador FJ
33     --Porta Transceiver
34         FIFO : in  STD_LOGIC;
35         FIFOp : in  STD_LOGIC;
36         CCA : in  STD_LOGIC;
37         SFD : in  STD_LOGIC;
38         RESETn : out  STD_LOGIC;
39         VREG_en : out  STD_LOGIC;
40     --SPI
41         MISO : in  STD_LOGIC;
42         CSn : out  STD_LOGIC :='1';
43         MOSI : out  STD_LOGIC :='Z';
44         SCLK : out  STD_LOGIC :='Z';
45     --Led Teste
46   LED1 : out  STD_LOGIC :='0';
47   LED2 : out  STD_LOGIC :='0';
48   LED3 : out  STD_LOGIC :='0';
49   LED4 : out  STD_LOGIC :='0'
50         );
51
52 end modulo_wireless;
53
54 architecture structure of modulo_wireless is
55
56   type mod_state is (INICIALIZACAO, IDLE, FRAME_TX, FRAME_RX, CONFIGURACAO);
57   signal module_state : mod_state;
58
59   type state_coord is (START, INIT_MODULE, WAIT_MODULE_INIT, WAITING, DECODE_COMMAND, SEND_FRAME, MOD_CONFIGURATION,
         SEND_CONFIG_STATUS, RECEIVE_FRAME, SEND_FRAME_FJ, SFLUSHRX, ERROR);
60   signal state_coordenador : state_coord;
61
62   signal state_send_fj : INTEGER := FRAME_TYPE;
63
64 --MOD_WIRELESS
65   signal clock : STD_LOGIC;
66   signal reset : STD_LOGIC;
67   signal inic_module : STD_LOGIC := '0';
68   signal module_ready : STD_LOGIC;
```

```
69    signal module_busy : STD_LOGIC := '0';
70
71  --FSM
72    signal data_count : INTEGER := (ZIGBEE_PAYLOAD/FJ_DATA_WIDTH);
73    signal first_time : BIT := '1';
74
75  --SPI
76    signal spi_clk    : STD_LOGIC;
77    signal spi_dataTx    : STD_LOGIC_VECTOR (SPI_DATA_WIDTH - 1 downto 0);
78    signal spi_dataRx    : STD_LOGIC_VECTOR (SPI_DATA_WIDTH - 1 downto 0);
79    signal spi_start : STD_LOGIC;
80    signal spi_reset : STD_LOGIC := '1';
81    signal spi_ready : STD_LOGIC;
82    signal spi_Length : INTEGER := 24;
83    signal spi_wait : STD_LOGIC;
84
85  --uC
86    signal uc_data_out_command : STD_LOGIC_VECTOR(UC_DATA_WIDTH-1 downto 0);
87    signal uc_data_out_data : STD_LOGIC_VECTOR(UC_DATA_WIDTH-1 downto 0);
88    signal uc_data_in_command : STD_LOGIC_VECTOR(UC_DATA_WIDTH-1 downto 0);
89    signal uc_data_in_data : STD_LOGIC_VECTOR(UC_DATA_WIDTH-1 downto 0);
90    signal flag_command_received : STD_LOGIC := '0';
91    signal flag_data_received : STD_LOGIC := '0';
92    signal flag_command_send : STD_LOGIC := '0';
93    signal flag_data_send : STD_LOGIC := '0';
94    signal flagCommandreceived : STD_LOGIC := '0';
95    signal flagDataReceived : STD_LOGIC := '0';
96    signal flagCommandSend : STD_LOGIC := '0';
97    signal flagDataSend : STD_LOGIC := '0';
98
99  --MOD_INIT
100   signal module_init : STD_LOGIC := '0';
101   signal spi_Reset_init : STD_LOGIC;
102   signal spi_Start_init : STD_LOGIC;
103   signal spi_DataTx_init : STD_LOGIC_VECTOR (SPI_DATA_WIDTH - 1 downto 0);
104
105 --MOD_CONFIG
106   signal module_config_init : STD_LOGIC := '0';
107   signal data_config : STD_LOGIC_VECTOR(SPI_DATA_WIDTH - 1 downto 0);
108   signal spi_Start_Config : STD_LOGIC;
109   signal spi_DataTx_Config : STD_LOGIC_VECTOR (SPI_DATA_WIDTH - 1 downto 0);
110   signal spi_Reset_Config : STD_LOGIC;
111   signal module_config_status : STD_LOGIC;
112   signal data_config_status : STD_LOGIC_VECTOR (SPI_DATA_WIDTH - 1 downto 0);
113
114 --FRAME_TX
115   signal payload_frameTx : STD_LOGIC_VECTOR ( ZIGBEE_PAYLOAD-1 downto 0);
116   signal addr_source_frameTx : STD_LOGIC_VECTOR ( ADDR_LENGTH_16 - 1 downto 0);
117   signal addr_dest_frameTx : STD_LOGIC_VECTOR ( ADDR_LENGTH_16 - 1 downto 0);
118   signal addr_pan_frameTx : STD_LOGIC_VECTOR ( PAN_ADDR_WIDTH - 1 downto 0);
119   signal frame_type_frameTx : STD_LOGIC_VECTOR ( FRAME_TYPE_LENGTH - 1 downto 0);
120   signal data_length_frameTx : INTEGER;
121   signal busy_frameTx : STD_LOGIC := '0';
122   signal send_frameTx : STD_LOGIC := '0';
123   signal spi_Reset_frameTx : STD_LOGIC;
124   signal spi_Start_frameTx : STD_LOGIC;
125   signal spi_DataTx_frameTx : STD_LOGIC_VECTOR (SPI_DATA_WIDTH - 1 downto 0);
126
127 --FRAME_RX
128   signal payload_frameRx : STD_LOGIC_VECTOR ( ZIGBEE_PAYLOAD-1 downto 0);
129   signal addr_source_frameRx : STD_LOGIC_VECTOR ( ADDR_LENGTH_16 - 1 downto 0);
130   signal addr_dest_frameRx : STD_LOGIC_VECTOR ( ADDR_LENGTH_16 - 1 downto 0);
131   signal frame_type_frameRx : STD_LOGIC_VECTOR ( FRAME_TYPE_LENGTH - 1 downto 0);
132   signal frame_seq_frameRx : STD_LOGIC_VECTOR ( FRAME_SEQ_FIELD - 1 downto 0);
133   signal frame_rssi_frameRx : STD_LOGIC_VECTOR ( FRAME_RSSI_FIELD - 1 downto 0);
134   signal data_length_frameRx : INTEGER;
135   signal status_error_frameRx : STD_LOGIC;
136   signal busy_frameRx : STD_LOGIC := '0';
137   signal spi_DataTx_frameRx : STD_LOGIC_VECTOR (SPI_DATA_WIDTH - 1 downto 0);
138   signal spi_DataRx_frameRx : STD_LOGIC_VECTOR (SPI_DATA_WIDTH - 1 downto 0);
139   signal receive_frameRx : STD_LOGIC := '0';
140   signal spi_Reset_frameRx : STD_LOGIC;
141   signal spi_Start_frameRx : STD_LOGIC;
142
143 begin
144
145   clock <= system_clk;
146   reset <= system_rst;
147   flag_command_send <= flagCommandSend;
148   flag_data_send <= flagDataSend;
149   flag_data_received <= flagDataReceived;
150   flag_command_received <= flagCommandReceived;
151
152   LED1 <= FIFO;
153    LED2 <= FIFOp;
154    LED3 <= CCA;
155    LED4 <= SFD;
156
157 ----VIRTEX II - clock 32MHz
158   spi_clock : divisor_clock
159     generic map(
160       DIVIDER_RATIO => "00001"   -- clock = 2*clk
161       )
162     port map(
163       in_clk => clock,
164       out_clk => spi_clk
165       );
166 ----------------
167
```

```
168 ----SPARTAN-3AN - clock 50MHz
169 --  spi_clock : divisor_clock
170 --    generic map(
171 --      DIVIDER_RATIO => "00010"   -- clock = 6*clk
172 --    )
173 --    port map(
174 --      in_clk => clock,
175 --      out_clk => spi_clk
176 --     );
177 ----------------
178
179   mod_tx : entity work.frame_tx
180     port map(
181       clock => clock,
182       payload => payload_frameTx,
183       addr_source => addr_source_frameTx,
184       addr_dest => addr_dest_frameTx,
185       addr_pan => addr_pan_frameTx,
186       frame_type => frame_type_frameTx,
187       frame_data_length => data_length_frameTx,
188       frame_send => send_frameTx,
189       CCA_pin => CCA,
190       SFD_pin => SFD,
191       frame_busy => busy_frameTx,
192       spi_Reset => spi_Reset_frameTx,
193       spi_DataTx => spi_DataTx_frameTx,
194       spi_Start => spi_Start_frameTx,
195       spi_DataRx => spi_dataRx,
196       spi_Ready => spi_ready
197       );
198
199
200   mod_rx : entity work.frame_rx
201     port map(
202       clock => clock,
203       payload => payload_frameRx,
204       addr_source => addr_source_frameRx,
205       addr_dest => addr_dest_frameRx,
206       frame_type => frame_type_frameRx,
207       frame_data_length => data_length_frameRx,
208       frame_seq => frame_seq_frameRx,
209       frame_rssi => frame_rssi_frameRx,
210       frame_receive => receive_frameRx,
211       FIFO_pin => FIFO,
212       SFD_pin => SFD,
213       frameRx_busy => busy_frameRx,
214       status_error => status_error_frameRx,
215       spi_DataRx => spi_DataRx_frameRx,
216       spi_Ready => spi_ready,
217       spi_Reset => spi_Reset_frameRx,
218       spi_DataTx => spi_DataTx_frameRx,
219       spi_Start => spi_Start_frameRx
220
221     );
222
223
224   mod_init : entity work.mod_init
225     port map(
226       clock => clock,
227       spi_DataRx => spi_dataRx,
228       spi_Ready => spi_ready,
229       module_init => module_init,
230       module_status => module_ready,
231       spi_Reset => spi_Reset_init,
232       spi_DataTx => spi_DataTx_init,
233       spi_Start => spi_Start_init,
234       RESETn => RESETn,
235       VREG_en => VREG_en
236     );
237
238
239   mod_config : entity work.mod_config
240     port map(
241       clock => clock,
242       data_config => data_config,
243       spi_DataRx => spi_dataRx,
244       spi_Ready => spi_ready,
245       module_init => module_config_init,
246       spi_Reset => spi_Reset_Config,
247       spi_DataTx => spi_DataTx_Config,
248       spi_Start => spi_Start_Config,
249       module_status => module_config_status,
250       data_status => data_config_status
251
252     );
253
254   spi : spi_com   -- cuidar a restrição do CC2420 de 10MHz em SCLK
255     generic map(
256       DATA_WIDTH => SPI_DATA_WIDTH
257       )
258     port map (          -- completar a descrição
259       reset => spi_reset,
260       clk => spi_clk, --spi_clk < 20MHz
261       SCLK => SCLK,  --SCLK = clk/2
262       CSn => CSn,
263       MOSI => MOSI,
264       MISO => MISO,
265       DataTx => spi_dataTx,
266       DataRx => spi_dataRx,
```

```
267         Start => spi_start,
268          Status => spi_ready,
269         DataLength => spi_Length
270         );
271
272 ----------------------------
273 --ATRIBUICOES ASSINCRONAS
274 ----------------------------
275     spi_dataTx <=   spi_DataTx_Init     when  module_state = INICIALIZACAO  else
276                 spi_DataTx_frameTx  when  module_state = FRAME_TX      else
277                 spi_DataTx_frameRx  when  module_state = FRAME_RX      else
278                 spi_DataTx_Config   when  module_state = CONFIGURACAO else
279                 "00000000000000000000000";
280
281
282     spi_start <=  spi_Start_Init    when  module_state = INICIALIZACAO  else
283                 spi_Start_frameTx   when  module_state = FRAME_TX      else
284                 spi_Start_frameRx   when  module_state = FRAME_RX      else
285                 spi_Start_Config     when   module_state = CONFIGURACAO else
286                 '0';
287
288     spi_reset <=  spi_Reset_Init    when  module_state = INICIALIZACAO  else
289                 spi_Reset_frameTx   when  module_state = FRAME_TX      else
290                 spi_Reset_frameRx   when  module_state = FRAME_RX      else
291                 spi_Reset_Config    when  module_state = CONFIGURACAO else
292                 '0';
293
294 ------------------------------------
295 -- PROCESSO PARA GERENCIAR PROCESSOS
296 ------------------------------------
297 MANAGER : process(clock)
298
299   variable cont : integer :=0;
300
301 begin
302
303   if(clock'event and clock = '1') then
304
305     case state_coordenador is
306       when START =>
307         state_coordenador <= INIT_MODULE;
308
309       when INIT_MODULE =>
310           module_state <= INICIALIZACAO;
311           module_init <= '1';
312           state_coordenador <= WAIT_MODULE_INIT;
313
314       when WAIT_MODULE_INIT =>
315         if(module_ready = '1') then
316           state_coordenador <= WAITING;
317           module_init <= '0';
318         end if;
319
320       when WAITING =>
321         if(flag_command_received = '1') then
322           state_coordenador <= DECODE_COMMAND;
323
324         elsif(FIFO='1' and FIFOp='1') then
325           state_coordenador <= RECEIVE_FRAME;
326
327         elsif(FIFO='0' and FIFOp='1') then
328           state_coordenador <= SFLUSHRX;
329
330         elsif(reset = '1') then
331           state_coordenador <= START;
332
333         else
334           state_coordenador <= WAITING;
335         end if;
336
337
338       when DECODE_COMMAND =>
339         if(flag_data_received = '1') then
340           flagCommandReceived <= '0';
341           flagDataReceived <= '0';
342
343           case conv_integer(uc_data_in_command) is
344
345             when SOURCE_ADDR =>
346               addr_source_frameTx <= uc_data_in_data(ADDR_LENGTH_16-1 downto 0);
347               state_coordenador <= WAITING;
348
349             when DEST_ADDR =>
350               addr_dest_frameTx <= uc_data_in_data(ADDR_LENGTH_16-1 downto 0);
351               state_coordenador <= WAITING;
352
353             when PAN_ADDR =>
354               addr_pan_frameTx <= uc_data_in_data(PAN_ADDR_WIDTH-1 downto 0);
355               state_coordenador <= WAITING;
356
357             when FRAME_TYPE =>
358               frame_type_frameTx <= uc_data_in_data(FRAME_TYPE_LENGTH-1 downto 0);
359               state_coordenador <= WAITING;
360
361             when DATA =>
362               if(data_count = 0) then
363                 state_coordenador <= ERROR;
364                 data_count <= (ZIGBEE_PAYLOAD/FJ_DATA_WIDTH);
365               else
```

```
366                 payload_frameTx(((FJ_DATA_WIDTH*data_count)-1) downto (FJ_DATA_WIDTH*(data_count-1))) <=
                          uc_data_in_data; --AJEITAR--
367                 state_coordenador <= WAITING;
368                 data_count <= data_count - 1;
369               end if;
370
371             when SEND =>
372               state_coordenador <= SEND_FRAME;
373               module_busy <= '1';
374               data_count <= (ZIGBEE_PAYLOAD/FJ_DATA_WIDTH);
375               data_length_frameTx <= (ZIGBEE_PAYLOAD-data_count)*(FJ_DATA_WIDTH/8);
376
377             when CONFIG_MODULE =>
378               data_config <= uc_data_in_data(SPI_DATA_WIDTH - 1 downto 0);
379               state_coordenador <= MOD_CONFIGURATION;
380               module_busy <= '1';
381
382             when others =>
383               state_coordenador <= WAITING;
384
385           end case;
386         end if;
387
388
389       when SEND_FRAME =>
390         if(first_time = '1') then
391           module_state <= FRAME_TX;
392           send_frameTx <='1';
393           state_coordenador <= SEND_FRAME;
394           first_time <= '0';
395         else
396           if(busy_frameTx = '1') then
397             state_coordenador <= SEND_FRAME;
398           else
399             state_coordenador <= WAITING;
400             send_frameTx <= '0';
401             module_busy <= '0';
402             first_time <= '1';
403           end if;
404         end if;
405
406
407       when MOD_CONFIGURATION =>
408         if(first_time = '1') then
409           module_state <= CONFIGURACAO;
410           module_config_init <='1';
411           state_coordenador <= MOD_CONFIGURATION;
412           first_time <= '0';
413         else
414           if(module_config_status = '1') then
415             state_coordenador <= MOD_CONFIGURATION;
416           else
417             state_coordenador <= SEND_CONFIG_STATUS;
418             module_config_init <= '0';
419             module_busy <= '0';
420             first_time <= '1';
421           end if;
422         end if;
423
424       when SEND_CONFIG_STATUS =>
425         uc_data_out_command <= conv_std_logic_vector( CONFIG_MODULE, FJ_DATA_WIDTH);
426         uc_data_out_data <= X"00000000";
427         uc_data_out_data(SPI_DATA_WIDTH-1 downto 0) <= data_config_status;
428         int1_fj <= '1';
429         state_coordenador <= SEND_CONFIG_STATUS;
430         if(flag_data_send = '1') then
431           int1_fj <= '0';
432           flagCommandSend <= '0';
433           flagDataSend <= '0';
434           state_coordenador <= WAITING;
435         end if;
436
437       when RECEIVE_FRAME =>
438         if(first_time = '1') then
439           module_busy <= '1';
440           module_state <= FRAME_RX;
441           receive_frameRx <= '1';
442           state_coordenador <= RECEIVE_FRAME;
443           first_time <= '0';
444         else
445           if(busy_frameRx = '1') then
446             receive_frameRx <= '0';
447             state_coordenador <= RECEIVE_FRAME;
448           else
449             state_coordenador <= WAITING;
450             module_busy <= '0';
451             first_time <= '1';
452           end if;
453         end if;
454
455       when SEND_FRAME_FJ =>
456           case state_send_fj is
457
458             when FRAME_TYPE =>
459               uc_data_out_command <= conv_std_logic_vector( FRAME_TYPE, FJ_DATA_WIDTH);
460               uc_data_out_data <= X"00000000";
461               uc_data_out_data(FRAME_TYPE_LENGTH-1 downto 0) <= frame_type_frameRx;
462               int1_fj <= '1';
463               state_send_fj <= DEST_ADDR;
```

```
464                 state_coordenador <= SEND_FRAME_FJ;
465
466               when DEST_ADDR =>
467                 state_send_fj <= DEST_ADDR;
468                 state_coordenador <= SEND_FRAME_FJ;
469                 if(flag_data_send = '1') then
470                   flagCommandSend <= '0';
471                   flagDataSend <= '0';
472                   int1_fj <= '0';
473                   uc_data_out_command <= conv_std_logic_vector( DEST_ADDR, FJ_DATA_WIDTH);
474                   uc_data_out_data <= X"00000000";
475                   uc_data_out_data(ADDR_LENGTH_16-1 downto 0) <= addr_dest_frameRx;
476                   state_send_fj <= SOURCE_ADDR;
477                 end if;
478
479               when SOURCE_ADDR =>
480                 state_send_fj <= SOURCE_ADDR;
481                 state_coordenador <= SEND_FRAME_FJ;
482                 if(flag_data_send = '1') then
483                   flagCommandSend <= '0';
484                   flagDataSend <= '0';
485                   uc_data_out_command <= conv_std_logic_vector( SOURCE_ADDR, FJ_DATA_WIDTH);
486                   uc_data_out_data <= X"00000000";
487                   uc_data_out_data(ADDR_LENGTH_16-1 downto 0) <= addr_source_frameRx;
488                   state_send_fj <= DATA;
489                 end if;
490
491
492               when DATA =>
493                 state_send_fj <= DATA;
494                 state_coordenador <= SEND_FRAME_FJ;
495                 if(flag_data_send = '1') then
496                   flagCommandSend <= '0';
497                   flagDataSend <= '0';
498                   uc_data_out_command <= conv_std_logic_vector( DATA, FJ_DATA_WIDTH);
499                   uc_data_out_data <= X"00000000";
500                   uc_data_out_data <= payload_frameRx(31 downto 0);
501                   state_send_fj <= RSSI;
502                 end if;
503
504               when RSSI =>
505                 state_send_fj <= RSSI;
506                 state_coordenador <= SEND_FRAME_FJ;
507                 if(flag_data_send = '1') then
508                   flagCommandSend <= '0';
509                   flagDataSend <= '0';
510                   uc_data_out_command <= conv_std_logic_vector( RSSI, FJ_DATA_WIDTH);
511                   uc_data_out_data <= X"00000000";
512                   uc_data_out_data(FRAME_RSSI_FIELD-1 downto 0) <= frame_rssi_frameRx;
513                   state_send_fj <= FRAME_TYPE;
514                   state_coordenador <= WAITING;
515                 end if;
516
517               when others =>
518                 state_coordenador <= WAITING;
519
520             end case;
521
522         when SFLUSHRX =>
523           module_busy <= '1';
524           data_config <= SFLUSHRX_COMMAND;
525           state_coordenador <= MOD_CONFIGURATION;
526
527         when ERROR =>
528           -- avisar de erro ao enviar
529
530         when OTHERS =>
531           state_coordenador <= WAITING;
532
533       end case;
534     end if;
535
536 end process;
537
538 ------------------------------------------------
539 ---- Decodificação de escrita e leitura do uC---
540 ------------------------------------------------
541
542 -- Processo para leitura do microcontrolador --
543
544 uc_read : process (uc_rw_en)
545   begin
546
547     if uc_rw_en'event and uc_rw_en='0' then
548       case uc_addr_in is
549
550         when WIRELESS_OUT_COMMAND =>
551           uc_data_in_command <= uc_data_in;
552           flag_command_received <= '1';
553
554         when WIRELESS_OUT_DATA =>
555           uc_data_in_data <= uc_data_in;
556           flag_data_received <= '1';
557
558         when OTHERS =>
559
560       end case;
561     end if;
562 end process;
```

```
563
564
565 -- Processo para escrita no microcontrolador --
566
567 uc_write : process (uc_rw_en)
568
569   variable cont : bit := '1';
570
571     begin
572
573       if uc_rw_en'event and uc_rw_en='1' then
574           case uc_addr_in is
575
576             when WIRELESS_IN_COMMAND =>
577               uc_data_out <= uc_data_out_command;
578               flag_command_send <= '1';
579
580             when WIRELESS_IN_DATA =>
581               uc_data_out <= uc_data_out_data;
582               flag_data_send <= '1';
583             when OTHERS =>
584
585           end case;
586       end if;
587 end process;
588
589 ------------------------------------------------
590 ------------------------------------------------
591
592 end structure;
```

modulo_wireless_types.vhd

```
1  ----------------------------------------------------------------
2  -- Company:        UFRGS - Univ. Fed. do RS
3  -- Engineer:       Rodrigo Schmidt Allgayer
4  --
5  -- Module Name:     modulo_wireless_types - Behavioral
6  -- Project Name:    FemtoNode
7  ----------------------------------------------------------------
8  library ieee;
9    use ieee.std_logic_1164.all;
10
11 package modulo_wireless_types is
12
13 constant BYTE : INTEGER:= 8;
14
15 --:: BUS SIZING
16   constant WIRELESS_DATA_WIDTH  : INTEGER:= 32;
17   constant FJ_DATA_WIDTH      : INTEGER:= 32;
18   constant FJ_DATA_ADDR_WIDTH : INTEGER:=16;
19
20 --:: Wireless Module
21   constant SPI_DATA_WIDTH    : INTEGER := 24;
22   constant ZIGBEE_PAYLOAD    : INTEGER := 96;
23   constant PAN_ADDR_WIDTH    : INTEGER := 16;
24   constant ADDR_LENGTH_16    : INTEGER := 16;
25   constant ADDR_LENGTH_64    : INTEGER := 64;
26   constant FRAME_ACK_LENGTH   : INTEGER := 48;
27   constant FRAME_CRC_LENGTH   : INTEGER := 16;
28   constant FRAME_MAX_LENGTH   : INTEGER := 192;
29
30 --:: FemtoJava ADDRESS
31   constant WIRELESS_IN_COMMAND  : STD_LOGIC_VECTOR (FJ_DATA_ADDR_WIDTH-1 downto 0):= X"0013";
32   constant WIRELESS_IN_DATA    : STD_LOGIC_VECTOR (FJ_DATA_ADDR_WIDTH-1 downto 0):= X"0014";
33   constant WIRELESS_OUT_COMMAND   : STD_LOGIC_VECTOR (FJ_DATA_ADDR_WIDTH-1 downto 0):= X"0015";
34   constant WIRELESS_OUT_DATA   : STD_LOGIC_VECTOR (FJ_DATA_ADDR_WIDTH-1 downto 0):= X"0016";
35   constant UC_DATA_WIDTH   : INTEGER := 32;  -- Tamanho do barramento de dados do uC
36   constant UC_ADDR_WIDTH   : INTEGER := 16;  -- Tamanho do barramento de endereços do uC
37
38 --:: FemtoJava COMMANDS
39   constant SOURCE_ADDR   : INTEGER:= 1;
40   constant DEST_ADDR   : INTEGER:= 2;
41   constant DATA       : INTEGER:= 3;
42   constant PAN_ADDR    : INTEGER:= 4;
43   constant DATA_LENGTH   : INTEGER:= 5;
44   constant SEND       : INTEGER:= 6;
45   constant CONFIG_MODULE   : INTEGER:= 7;
46   constant STATUS_MODULE   : INTEGER:= 8;
47   constant ERROR      : INTEGER:= 9;
48   constant FRAME_TYPE    : INTEGER:= 10;
49   constant ADDR_64   : INTEGER:= 11;
50   constant ADDR_16   : INTEGER:= 12;
51   constant RSSI    : INTEGER:= 13;
52
53 --:: IEEE802.15.4 Frame TYPES
54   constant BEACON_FRAME    : INTEGER := 0;
55   constant DATA_FRAME    : INTEGER := 1;
56   constant ACK_FRAME   : INTEGER := 2;
57   constant MAC_COMMAND_FRAME : INTEGER := 3;
58
59 --:: IEEE802.15.4 Frame Length FIELDS
60   constant FRAME_LENGTH_FIELD    : INTEGER := 1*BYTE;
61   constant FRAME_CONTROL_FIELD   : INTEGER := 2*BYTE;
62   constant FRAME_SEQ_FIELD   : INTEGER := 1*BYTE;
63   constant FRAME_RSSI_FIELD    : INTEGER := 1*BYTE;
64   constant FRAME_CHECK_FIELD   : INTEGER := 2*BYTE;
65   constant FRAME_PAN_ADDR_16_FIELD : INTEGER := 6*BYTE;
66   constant FRAME_TYPE_LENGTH   : INTEGER := 2*BYTE;
67
68 --:: IEEE802.15.4 Frame CONTROL Field
69   constant FRAME_TYPE_BEACON   : STD_LOGIC_VECTOR (FRAME_TYPE_LENGTH-1 downto 0) :=  b"0000000000000000";
70   constant FRAME_TYPE_DATA   : STD_LOGIC_VECTOR (FRAME_TYPE_LENGTH-1 downto 0) :=  b"0000000000000001";
71   constant FRAME_TYPE_ACK    : STD_LOGIC_VECTOR (FRAME_TYPE_LENGTH-1 downto 0) :=  b"0000000000000010";
72   constant FRAME_TYPE_MAC_COMMAND : STD_LOGIC_VECTOR (FRAME_TYPE_LENGTH-1 downto 0) := b"0000000000000011";
73   constant FRAME_TYPE_MASK   : STD_LOGIC_VECTOR (FRAME_TYPE_LENGTH-1 downto 0) :=  b"0000000000000111";
74   constant ACK_REQUEST     : STD_LOGIC_VECTOR (FRAME_CONTROL_FIELD-1 downto 0)   := b"0000000000100000";
75   constant INTRA_PAN      : STD_LOGIC_VECTOR (FRAME_CONTROL_FIELD-1 downto 0)     := b"0000000001000000";
76   constant SECURITY       : STD_LOGIC_VECTOR (FRAME_CONTROL_FIELD-1 downto 0)     := b"0000000000001000";
77   constant FRAME_PENDING     : STD_LOGIC_VECTOR (FRAME_CONTROL_FIELD-1 downto 0) := b"0000000000010000";
78   constant DST_ADDR_16     : STD_LOGIC_VECTOR (FRAME_CONTROL_FIELD-1 downto 0)   := b"0000100000000000";
79   constant DST_ADDR_64     : STD_LOGIC_VECTOR (FRAME_CONTROL_FIELD-1 downto 0)   := b"0000110000000000";
80   constant SRC_ADDR_16     : STD_LOGIC_VECTOR (FRAME_CONTROL_FIELD-1 downto 0)   := b"1000000000000000";
81   constant SRC_ADDR_64     : STD_LOGIC_VECTOR (FRAME_CONTROL_FIELD-1 downto 0)   := b"1100000000000000";
82
83 --:: CC2420 COMMAND
84   constant SFLUSHRX_COMMAND    : STD_LOGIC_VECTOR (SPI_DATA_WIDTH-1 downto 0) := X"080000";
85
86 end modulo_wireless_types;
```

mod_init.vhd

```
1  ----------------------------------------------------
2  -- Company:        UFRGS - Univ. Fed. do RS
3  -- Engineer:       Rodrigo Schmidt Allgayer
4  --
5  -- Module Name:     mod_init - Behavioral
6  -- Project Name:    FemtoNode
7  ----------------------------------------------------
8  library IEEE;
9  use IEEE.STD_LOGIC_1164.ALL;
10 use IEEE.STD_LOGIC_ARITH.ALL;
11 use IEEE.STD_LOGIC_UNSIGNED.ALL;
12
13 library work;
14   use work.modulo_wireless_types.all;
15
16 entity mod_init is
17
18   port(
19     clock : in STD_LOGIC;
20     spi_DataRx : in STD_LOGIC_VECTOR(SPI_DATA_WIDTH - 1 downto 0);
21     spi_Ready : in STD_LOGIC;
22     module_init : in STD_LOGIC;
23     module_status : out STD_LOGIC;
24     spi_Reset : out STD_LOGIC;
25     spi_DataTx : out STD_LOGIC_VECTOR(SPI_DATA_WIDTH - 1 downto 0);
26     spi_Start : out STD_LOGIC;
27     RESETn : out STD_LOGIC;
28     VREG_en : out STD_LOGIC
29   );
30
31 end mod_init;
32
33 architecture Behavioral of mod_init is
34
35   signal module_ready : STD_LOGIC := '0';
36     signal spi_Start_Inic : STD_LOGIC;
37
38   type state_begin is (START, SET_VREG, WAIT_VREG, PULSE_RESET_OFF, WAIT_RESET_OFF, PULSE_RESET_ON, WAIT_RESET_ON,
         STROBE_SXOSCON, WAIT_SXOSCON, VERIF_OSCILLATOR, WAIT_OSCILLATOR, OSCILLATOR_OK, DISABLE_ADR, WAIT_DISABLE,
         WAIT_CHANNEL, CHANGE_CHANNEL, THR_FIFOP, STROBE_SRXON, WAIT_STROBE, READY);
39   signal state_inic : state_begin;
40
41 begin
42
43 spi_Start <= spi_Start_Inic;
44
45 -- PROCESSO PARA INICIALIZACAO DO MODULO CC2420
46
47 MODULE_INICIALIZACAO : process(clock, module_init)
48
49     variable cont : INTEGER :=0;
50     variable enviando : BIT := '0';
51     variable dataRXBuffer : STD_LOGIC_VECTOR (SPI_DATA_WIDTH - 1 downto 0);
52
53   begin
54
55     if(clock'event and clock = '1') then
56      case state_inic is
57         when START =>
58           if(module_init = '1') then
59             state_inic <= SET_VREG;
60             module_status <= '0';
61           else
62             state_inic <= START;
63           end if;
64
65         when SET_VREG =>
66           VREG_en <= '1';
67           state_inic <= WAIT_VREG;
68
69         when WAIT_VREG =>
70           if(cont > 320000) then
71             state_inic <= PULSE_RESET_OFF;
72             cont := 0;
73           else
74             cont := cont+1;
75           end if;
76
77         when PULSE_RESET_OFF =>
78             RESETn <= '0';
79             state_inic <= WAIT_RESET_OFF;
80
81         when WAIT_RESET_OFF =>
82           if(cont > 320000) then
83             state_inic <= PULSE_RESET_ON;
84             cont := 0;
85           else
86             cont := cont+1;
87           end if;
88
89         when PULSE_RESET_ON =>
90           RESETn <= '1';
91           state_inic <= WAIT_RESET_ON;
92
93         when WAIT_RESET_ON =>
94           if(cont > 30) then
95             state_inic <= STROBE_SXOSCON;
```

```
96              cont := 0;
97            else
98              cont := cont+1;
99            end if;
100
101         when STROBE_SXOSCON =>
102           if(spi_Ready='1' and spi_Start_Inic='0') then
103             spi_DataTx <= X"010000";
104             spi_Reset <= '0';
105             spi_Start_Inic  <= '1';
106             enviando := '1';
107           elsif(enviando='1' and spi_Ready='0') then
108             spi_Start_Inic <= '0';
109             state_inic <= WAIT_SXOSCON;
110             enviando := '0';
111           end if;
112
113         when WAIT_SXOSCON =>
114           if(spi_Ready='1') then
115             state_inic <= VERIF_OSCILLATOR;
116             spi_Reset <= '1';
117           end if;
118
119         when VERIF_OSCILLATOR =>
120           if(spi_Ready='1' and spi_Start_Inic='0') then
121             spi_DataTx <= X"000000";
122             spi_Reset <= '0';
123             spi_Start_Inic <= '1';
124             enviando := '1';
125           elsif(enviando='1' and spi_Ready='0') then
126             spi_Start_Inic <= '0';
127             state_inic <= WAIT_OSCILLATOR;
128             enviando := '0';
129           end if;
130
131         when WAIT_OSCILLATOR =>
132           if(spi_Ready='1') then
133             state_inic <= OSCILLATOR_OK;
134             spi_Reset <= '1';
135              dataRXBuffer := spi_DataRx;
136           end if;
137
138         when OSCILLATOR_OK =>
139           if(dataRXBuffer(22)='1') then
140             state_inic <= DISABLE_ADR;
141           else
142             state_inic <= VERIF_OSCILLATOR;
143           end if;
144
145         when DISABLE_ADR =>
146           if(spi_Ready='1' and spi_Start_Inic='0') then
147             spi_DataTx <= X"1102E2";
148             spi_Reset <= '0';
149             spi_Start_Inic <= '1';
150             enviando := '1';
151           elsif(enviando='1' and spi_Ready='0') then
152             spi_Start_Inic <= '0';
153             state_inic <= WAIT_DISABLE;
154             enviando := '0';
155           end if;
156
157         when WAIT_DISABLE =>
158           if(spi_Ready='1') then
159             state_inic <= CHANGE_CHANNEL;
160             spi_Reset <= '1';
161           end if;
162
163         when CHANGE_CHANNEL =>
164           if(spi_Ready='1' and spi_Start_Inic='0') then
165             spi_DataTx <= X"1841B0";
166             spi_Reset <= '0';
167             spi_Start_Inic <= '1';
168             enviando := '1';
169           elsif(enviando='1' and spi_Ready='0') then
170             spi_Start_Inic <= '0';
171             state_inic <= WAIT_CHANNEL;
172             enviando := '0';
173           end if;
174
175         when WAIT_CHANNEL =>
176           if(spi_Ready='1') then
177             state_inic <= STROBE_SRXON;
178             spi_Reset <= '1';
179           end if;
180
181
182         when STROBE_SRXON =>
183           if(spi_Ready='1' and spi_Start_Inic='0') then
184             spi_DataTx <= X"030000";
185             spi_Reset <= '0';
186             spi_Start_Inic <= '1';
187             enviando := '1';
188           elsif(enviando='1' and spi_ready='0') then
189             spi_Start_Inic <= '0';
190             state_inic <= WAIT_STROBE;
191             enviando := '0';
192           end if;
193
194         when WAIT_STROBE =>
```

```
195             if(spi_Ready='1') then
196               state_inic <= READY;
197               spi_Reset <= '1';
198             end if;
199
200           when READY =>
201             module_status <= '1';
202             state_inic <= START;
203
204           when others =>
205             state_inic <= READY;
206
207         end case;
208       end if;
209
210 end process;
211 end Behavioral;
```

mod_config.vhd

```
1  ----------------------------------------------------
2  -- Company:        UFRGS - Univ. Fed. do RS
3  -- Engineer:       Rodrigo Schmidt Allgayer
4  --
5  -- Module Name:     mod_config - Behavioral
6  -- Project Name:    FemtoNode
7  ----------------------------------------------------
8  library IEEE;
9  use IEEE.STD_LOGIC_1164.ALL;
10 use IEEE.STD_LOGIC_ARITH.ALL;
11 use IEEE.STD_LOGIC_UNSIGNED.ALL;
12
13 library work;
14   use work.modulo_wireless_types.all;
15
16 entity mod_config is
17
18   port(
19     clock : in STD_LOGIC;
20     data_config : in STD_LOGIC_VECTOR(SPI_DATA_WIDTH - 1 downto 0);
21     spi_Ready : in STD_LOGIC;
22     module_init : in STD_LOGIC;
23     spi_DataRx : in STD_LOGIC_VECTOR(SPI_DATA_WIDTH - 1 downto 0);
24     data_status : out STD_LOGIC_VECTOR(SPI_DATA_WIDTH - 1 downto 0);
25     module_status : out STD_LOGIC := '0'; --Busy = 1 , Free = 0
26     spi_Reset : out STD_LOGIC;
27     spi_DataTx : out STD_LOGIC_VECTOR(SPI_DATA_WIDTH - 1 downto 0);
28     spi_Start : out STD_LOGIC
29   );
30
31 end mod_config;
32
33 architecture Behavioral of mod_config is
34
35   signal spiStart : STD_LOGIC := '0';
36   signal spiReset : STD_LOGIC := '1';
37
38   type state_conf is (IDLE, SEND_CONFIG, WAIT_CONFIG, DATA_RECEIVED);
39   signal state_config : state_conf;
40
41 begin
42
43   spi_Start <= spiStart;
44   spi_Reset <= spiReset;
45
46 CONFIGURACAO : process( clock, module_init )
47
48     variable enviando : BIT := '0';
49
50 begin
51
52     if(clock'event and clock='1') then
53       case state_config is
54
55         when IDLE =>
56           if(module_init = '1') then
57             module_status <= '1';
58             state_config <= SEND_CONFIG;
59           else
60             state_config <= IDLE;
61           end if;
62
63         when SEND_CONFIG =>
64           if(spi_Ready='1' and spiStart='0') then
65             spi_DataTx <= data_config;
66             spiReset <= '0';
67             spiStart <= '1';
68             enviando := '1';
69           elsif(enviando='1' and spi_Ready='0') then
70             spiStart <= '0';
71             state_config <= WAIT_CONFIG;
72             enviando := '0';
73           end if;
74
75         when WAIT_CONFIG =>
76           if(spi_Ready='1') then
77             spiReset <= '1';
78             data_status <= spi_DataRx;
79             state_config <= DATA_RECEIVED;
80           end if;
81
82         when DATA_RECEIVED =>
83           module_status <= '1';
84           state_config <= IDLE;
85
86         when others =>
87           state_config <= IDLE;
88
89       end case;
90     end if;
91
92 end process;
93 end Behavioral;
```

frame_tx.vhd

```
1  ----------------------------------------------
2  -- Company:       UFRGS - Univ. Fed. do RS
3  -- Engineer:      Rodrigo Schmidt Allgayer
4  --
5  -- Module Name:    frame_tx - Behavioral
6  -- Project Name:   FemtoNode
7  ----------------------------------------------
8  library IEEE;
9  use IEEE.STD_LOGIC_1164.ALL;
10 use IEEE.STD_LOGIC_ARITH.ALL;
11 use IEEE.STD_LOGIC_UNSIGNED.ALL;
12 use IEEE.NUMERIC_STD.ALL;
13
14 library work;
15   use work.modulo_wireless_types.all;
16
17 entity frame_tx is
18
19   port(
20     clock : in STD_LOGIC;
21     payload : in STD_LOGIC_VECTOR ( ZIGBEE_PAYLOAD-1 downto 0 );
22     addr_source : in STD_LOGIC_VECTOR ( ADDR_LENGTH_16 - 1 downto 0);
23     addr_dest : in STD_LOGIC_VECTOR ( ADDR_LENGTH_16 - 1 downto 0);
24     addr_pan : in STD_LOGIC_VECTOR ( PAN_ADDR_WIDTH - 1 downto 0);
25     frame_type: in STD_LOGIC_VECTOR ( FRAME_TYPE_LENGTH - 1 downto 0);
26     frame_data_length : in integer;
27     frame_send : in STD_LOGIC;
28     spi_DataRx : in std_logic_vector(SPI_DATA_WIDTH - 1 downto 0);
29     spi_Ready : in STD_LOGIC;
30     CCA_pin : in STD_LOGIC;
31     SFD_pin : in STD_LOGIC;
32     frame_busy : out STD_LOGIC; -- 1:Busy  0:Free
33     spi_Reset : out STD_LOGIC;
34     spi_DataTx : out STD_LOGIC_VECTOR(SPI_DATA_WIDTH - 1 downto 0);
35     spi_Start : out STD_LOGIC
36   );
37
38 end frame_tx;
39
40 architecture Behavioral of frame_tx is
41
42   type state_send is (IDLE, START, SEND_DATA_FRAME, SEND_BEACON_FRAME, SEND_ACK_FRAME, SEND_MAC_COMMAND_FRAME,
       DATA_TXFIFO, WAIT_TXFIFO, TEST_CCA, WAIT_TIME_CCA, STROBE_STXON, WAIT_STROBE, SFD_LOW_HIGH, SFD_HIGH_LOW,
       WAIT_CYCLES, DATA_SEND);
43   signal state_tx : state_send;
44
45   signal spiStart : STD_LOGIC := '0';
46   signal spiReset : STD_LOGIC := '1';
47
48 begin
49
50   spi_Start <= spiStart;
51   spi_Reset <= spiReset;
52
53 SEND_FRAMETX : process(clock, frame_send)
54
55     variable cont : INTEGER :=0;
56     variable enviando : BIT := '0';
57     variable dataTXBuffer : STD_LOGIC_VECTOR (SPI_DATA_WIDTH - 1 downto 0);
58     variable frameTX : STD_LOGIC_VECTOR ( FRAME_MAX_LENGTH-1 downto 0 );
59     variable frameDataSeq : INTEGER := 0;
60     variable frameLength : STD_LOGIC_VECTOR ( FRAME_LENGTH_FIELD-1 downto 0) := X"00";
61     variable frameControl : STD_LOGIC_VECTOR ( FRAME_CONTROL_FIELD-1 downto 0) := X"0000";
62     variable frameCheck : STD_LOGIC_VECTOR ( FRAME_CRC_LENGTH-1 downto 0) := X"0000";
63     variable frameControl_tmp : STD_LOGIC_VECTOR ( FRAME_CONTROL_FIELD-1 downto 0);
64     variable frameDataSeq_tmp : STD_LOGIC_VECTOR ( FRAME_SEQ_FIELD-1 downto 0);
65     variable addr_pan_tmp : STD_LOGIC_VECTOR ( PAN_ADDR_WIDTH-1 downto 0);
66     variable addr_dest_tmp : STD_LOGIC_VECTOR ( ADDR_LENGTH_16-1 downto 0);
67     variable addr_source_tmp : STD_LOGIC_VECTOR ( ADDR_LENGTH_16-1 downto 0);
68     variable frameCont : INTEGER;
69
70 begin
71
72   if(clock'event and clock='1') then
73       case state_tx is
74
75         when IDLE =>
76           if(frame_send='1') then
77             frame_busy <= '1';
78             state_tx <= START;
79           else
80             frame_busy <= '0';
81             state_tx <= IDLE;
82           end if;
83
84         when START =>
85           case conv_integer(frame_type) is
86             when BEACON_FRAME =>
87               state_tx <= SEND_BEACON_FRAME;
88             when DATA_FRAME =>
89               state_tx <= SEND_DATA_FRAME;
90             when ACK_FRAME =>
91               state_tx <= SEND_ACK_FRAME;
92             when MAC_COMMAND_FRAME =>
93               state_tx <= SEND_MAC_COMMAND_FRAME;
94             when OTHERS =>
95               state_tx <= IDLE;
```

```
96              end case;
97
98            when SEND_DATA_FRAME =>
99              frameDataSeq := frameDataSeq + 1;
100             frameLength := conv_std_logic_vector(((FRAME_CONTROL_FIELD + FRAME_SEQ_FIELD + FRAME_PAN_ADDR_16_FIELD +
                    ZIGBEE_PAYLOAD + FRAME_CHECK_FIELD)/BYTE) , FRAME_LENGTH_FIELD);
101             frameControl := FRAME_TYPE_DATA or INTRA_PAN or DST_ADDR_16 or SRC_ADDR_16;
102             frameControl_tmp := frameControl((FRAME_CONTROL_FIELD/2)-1 downto 0) & frameControl(FRAME_CONTROL_FIELD-1
                    downto FRAME_CONTROL_FIELD/2);
103             frameDataSeq_tmp := conv_std_logic_vector( frameDataSeq , FRAME_SEQ_FIELD);
104             addr_pan_tmp := addr_pan((PAN_ADDR_WIDTH/2)-1 downto 0) & addr_pan(PAN_ADDR_WIDTH-1 downto PAN_ADDR_WIDTH/2)
                    ;
105             addr_dest_tmp := addr_dest((ADDR_LENGTH_16/2)-1 downto 0) & addr_dest(ADDR_LENGTH_16-1 downto ADDR_LENGTH_16
                    /2);
106             addr_source_tmp := addr_source((ADDR_LENGTH_16/2)-1 downto 0) & addr_source(ADDR_LENGTH_16-1 downto
                    ADDR_LENGTH_16/2);
107             frameTX(FRAME_MAX_LENGTH-1 downto 0) := frameLength & frameControl_tmp & frameDataSeq_tmp & addr_pan_tmp &
                    addr_dest_tmp & addr_source_tmp & payload & frameCheck;
108             frameCont := FRAME_LENGTH_FIELD + FRAME_CONTROL_FIELD + FRAME_SEQ_FIELD + FRAME_PAN_ADDR_16_FIELD +
                    ZIGBEE_PAYLOAD + FRAME_CHECK_FIELD;
109             state_tx <= DATA_TXFIFO;
110
111
112           when SEND_ACK_FRAME =>
113             frameLength := conv_std_logic_vector(((FRAME_CONTROL_FIELD + FRAME_SEQ_FIELD + FRAME_CHECK_FIELD)/BYTE) ,
                    FRAME_LENGTH_FIELD);
114             frameControl := FRAME_TYPE_ACK or INTRA_PAN or DST_ADDR_16 or SRC_ADDR_16;
115             frameControl_tmp := frameControl((FRAME_CONTROL_FIELD/2)-1 downto 0) & frameControl(FRAME_CONTROL_FIELD-1
                    downto FRAME_CONTROL_FIELD/2);
116             frameTX(FRAME_ACK_LENGTH-1 downto 0) := frameLength & frameControl_tmp & frameDataSeq_tmp & frameCheck;
117             frameCont := FRAME_LENGTH_FIELD + FRAME_CONTROL_FIELD + FRAME_SEQ_FIELD + FRAME_CHECK_FIELD;
118             state_tx <= DATA_TXFIFO;
119
120           when SEND_BEACON_FRAME =>
121             --not implemented
122
123           when SEND_MAC_COMMAND_FRAME =>
124             --not implemented
125
126           when DATA_TXFIFO =>
127             if(spi_Ready='1' and spiStart='0') then
128               frameCont := frameCont - 16;
129               dataTXBuffer(15 downto 0) := frameTX((frameCont+15) downto frameCont);
130               spi_DataTx <= X"3E0000" or dataTXBuffer;
131               spiReset <= '0';
132               spiStart <= '1';
133               enviando := '1';
134             elsif(enviando='1' and spi_Ready='0') then
135               spiStart <= '0';
136               state_tx <= WAIT_TXFIFO;
137               enviando := '0';
138               dataTXBuffer := X"000000";
139               frameDataSeq := frameDataSeq + 1;
140             end if;
141
142           when WAIT_TXFIFO =>
143             if(spi_ready='1') then
144               spiReset <= '1';
145               if(frameCont>0) then
146                 state_tx <= DATA_TXFIFO;
147               else
148                 state_tx <= TEST_CCA;
149               end if;
150             end if;
151
152           when TEST_CCA =>
153             if(CCA_pin = '1') then
154               state_tx <= STROBE_STXON;
155             else
156               state_tx <= WAIT_TIME_CCA;
157             end if;
158
159           when WAIT_TIME_CCA =>
160             if(cont > 32000 ) then
161               state_tx <= TEST_CCA;
162               cont := 0;
163             else
164               cont := cont+1;
165             end if;
166
167           when STROBE_STXON =>
168             if(spi_Ready='1' and spiStart='0') then
169               spi_DataTx <= X"040000";
170               spiReset <= '0';
171               spiStart <= '1';
172               enviando := '1';
173             elsif(enviando='1' and spi_Ready='0') then
174               spiStart <= '0';
175               state_tx <= WAIT_STROBE;
176               enviando := '0';
177             end if;
178
179           when WAIT_STROBE =>
180             if(spi_Ready='1') then
181               state_tx <= SFD_LOW_HIGH;
182               spiReset <= '1';
183             end if;
184
185           when SFD_LOW_HIGH =>
```

```
186             if(SFD_pin = '1') then
187               state_tx <= SFD_HIGH_LOW;
188             end if;
189
190           when SFD_HIGH_LOW =>
191             if(SFD_pin = '0') then
192               state_tx <= WAIT_CYCLES;
193             end if;
194
195           when WAIT_CYCLES =>
196             if(cont > (192*32) ) then
197               state_tx <= DATA_SEND;
198               frame_busy <= '0';
199               cont := 0;
200             else
201               cont := cont+1;
202             end if;
203
204           when DATA_SEND =>
205             state_tx <= IDLE;
206
207           when others =>
208             state_tx <= IDLE;
209
210       end case;
211   end if;
212
213 end process;
214 end Behavioral;
```

frame_rx.vhd

```
1  ----------------------------------------------------
2  -- Company:        UFRGS - Univ. Fed. do RS
3  -- Engineer:       Rodrigo Schmidt Allgayer
4  --
5  -- Module Name:     frame_rx - Behavioral
6  -- Project Name:    FemtoNode
7  ----------------------------------------------------
8  library IEEE;
9  use IEEE.STD_LOGIC_1164.ALL;
10 use IEEE.STD_LOGIC_ARITH.ALL;
11 use IEEE.STD_LOGIC_UNSIGNED.ALL;
12
13 library work;
14   use work.modulo_wireless_types.all;
15
16 entity frame_rx is
17
18   port(
19     clock : in STD_LOGIC;
20     payload : out STD_LOGIC_VECTOR ( ZIGBEE_PAYLOAD-1 downto 0);
21     addr_source : out STD_LOGIC_VECTOR ( ADDR_LENGTH_16 - 1 downto 0);
22     addr_dest : out STD_LOGIC_VECTOR ( ADDR_LENGTH_16 - 1 downto 0);
23     frame_type : out STD_LOGIC_VECTOR ( FRAME_TYPE_LENGTH - 1 downto 0);
24     frame_seq : out STD_LOGIC_VECTOR ( FRAME_SEQ_FIELD - 1 downto 0);
25     frame_rssi : out STD_LOGIC_VECTOR ( FRAME_RSSI_FIELD - 1 downto 0);
26     frame_data_length : out INTEGER;
27     frame_receive : in STD_LOGIC;
28     spi_DataRx : in STD_LOGIC_VECTOR(SPI_DATA_WIDTH - 1 downto 0);
29     spi_Ready : in STD_LOGIC;
30     FIFO_pin : in STD_LOGIC;
31     SFD_pin : in STD_LOGIC;
32     status_error : out STD_LOGIC; -- 1:Error  0:OK
33     frameRx_busy : out STD_LOGIC; -- 1:Busy  0:Free
34     spi_Reset : out STD_LOGIC;
35     spi_DataTx : out STD_LOGIC_VECTOR(SPI_DATA_WIDTH - 1 downto 0);
36     spi_Start : out STD_LOGIC
37   );
38
39 end frame_rx;
40
41 architecture Behavioral of frame_rx is
42
43    type state_receive is (IDLE, TEST_FIFO, RECEIVE_LENGTH_CONTROL_SEQ_FIELD, WAIT_RECEIVE_LENGTH_CONTROL_SEQ_FIELD,
          RECEIVE_FRAME, WAIT_RECEIVE_FRAME, DECODE_CONTROL_FIELD, STROBE_SFLUSHRX, WAIT_SFLUSHRX, DATA_RECEIVED);
44    signal state_rx : state_receive;
45
46   signal spiStart : STD_LOGIC := '0';
47   signal spiReset : STD_LOGIC := '1';
48
49 begin
50
51   spi_Start <= spiStart;
52   spi_Reset <= spiReset;
53
54 RECEBER_DADO : process(clock, frame_receive)
55
56     variable frame_count : INTEGER :=0;
57     variable frame_length : INTEGER := 0;
58     variable enviando : BIT := '0';
59     variable first_time : BIT := '1';
60     variable frameLength : STD_LOGIC_VECTOR ( FRAME_LENGTH_FIELD-1 downto 0 );
61     variable frameControl : STD_LOGIC_VECTOR ( FRAME_CONTROL_FIELD-1 downto 0 );
62     variable frameSeq : STD_LOGIC_VECTOR ( FRAME_SEQ_FIELD-1 downto 0 );
63     variable frameType : INTEGER;
64     variable dataLength : INTEGER;
65     variable frameRx : STD_LOGIC_VECTOR (FRAME_MAX_LENGTH-1 downto 0);
66
67 begin
68
69     if(clock'event and clock='1') then
70       case state_rx is
71
72         when IDLE =>
73           if(frame_receive = '1') then
74             frameRx_busy <= '1';
75             state_rx <= TEST_FIFO;
76           else
77             state_rx <= IDLE;
78           end if;
79
80         when TEST_FIFO =>
81           if(FIFO_pin='1' and SFD_pin='0') then
82             state_rx <= RECEIVE_LENGTH_CONTROL_SEQ_FIELD;
83           else
84             state_rx <= DATA_RECEIVED;
85           end if;
86
87         when RECEIVE_LENGTH_CONTROL_SEQ_FIELD =>
88           if(spi_Ready='1' and spiStart='0') then
89             spi_DataTx <= X"7F0000";
90             spiReset <= '0';
91             spiStart <= '1';
92             enviando := '1';
93           elsif(enviando='1' and spi_Ready='0') then
94             spiStart <= '0';
95             state_rx <= WAIT_RECEIVE_LENGTH_CONTROL_SEQ_FIELD;
96             enviando := '0';
```

```
97              end if;
98
99            when WAIT_RECEIVE_LENGTH_CONTROL_SEQ_FIELD =>
100             if(spi_Ready='1') then
101               if(first_time = '1') then
102                 frameLength := spi_DataRx(15 downto 8);
103                 frameControl(7 downto 0) := spi_DataRx(7 downto 0);
104                 first_time := '0';
105                 state_rx <= RECEIVE_LENGTH_CONTROL_SEQ_FIELD;
106               else
107                 frameControl(15 downto 8) := spi_DataRx(15 downto 8);
108                 frameSeq := spi_DataRx(7 downto 0);
109                 spiReset <= '1';
110                 first_time := '1';
111                 frame_length := conv_integer(frameLength);
112                 frame_count := 3;
113                 state_rx <= RECEIVE_FRAME;
114               end if;
115             end if;
116
117           when RECEIVE_FRAME =>
118             if(spi_Ready='1' and spiStart='0') then
119               spi_DataTx <= X"7F0000";
120               spiReset <= '0';
121               spiStart <= '1';
122               enviando := '1';
123             elsif(enviando='1' and spi_Ready='0') then
124               spiStart <= '0';
125               state_rx <= WAIT_RECEIVE_FRAME;
126               enviando := '0';
127               frame_length := frame_length + 2;
128             end if;
129
130           when WAIT_RECEIVE_FRAME =>
131             if(spi_Ready='1') then
132               spiReset <= '1';
133               frameRx((FRAME_MAX_LENGTH-frame_length-6) downto (FRAME_MAX_LENGTH-frame_length-22)) := spi_DataRx;
134               if(frame_length >= frame_count) then
135                 state_rx <= DECODE_CONTROL_FIELD;
136               else
137                 state_rx <= RECEIVE_FRAME;
138               end if;
139             end if;
140
141           when DECODE_CONTROL_FIELD =>
142             frameType := conv_integer(frameControl and FRAME_TYPE_MASK);
143
144             case frameType is
145               when DATA_FRAME =>
146                 frame_type <= frameControl and FRAME_TYPE_MASK;
147                 addr_dest <= frameRx(FRAME_MAX_LENGTH-17 downto FRAME_MAX_LENGTH-32);
148                 addr_source <= frameRx(FRAME_MAX_LENGTH-33 downto FRAME_MAX_LENGTH-48);
149                 dataLength := conv_integer(frameLength)-11;
150                 payload((dataLength*BYTE)-1 downto 0) <= frameRx(FRAME_MAX_LENGTH-49 downto FRAME_MAX_LENGTH-((
                      dataLength*BYTE)+48));
151                 frame_rssi <= frameRx(FRAME_MAX_LENGTH-((dataLength*BYTE)+49) downto FRAME_MAX_LENGTH-((dataLength*BYTE)
                      +56));
152                 frame_data_length <= dataLength;
153                 state_rx <= STROBE_SFLUSHRX;
154
155               when ACK_FRAME =>
156                 frame_rssi <= frameRx(FRAME_MAX_LENGTH-1 downto FRAME_MAX_LENGTH-8);
157                 state_rx <= STROBE_SFLUSHRX;
158
159               when BEACON_FRAME =>
160                 --not implemented
161                 state_rx <= STROBE_SFLUSHRX;
162
163               when MAC_COMMAND_FRAME =>
164                 --not implemented
165                 state_rx <= STROBE_SFLUSHRX;
166
167               when others =>
168                 state_rx <= STROBE_SFLUSHRX;
169
170             end case;
171
172           when STROBE_SFLUSHRX =>
173             if(spi_Ready='1' and spiStart='0') then
174               spi_DataTx <= X"080000";
175               spiReset <= '0';
176               spiStart <= '1';
177               enviando := '1';
178             elsif(enviando='1' and spi_Ready='0') then
179               spiStart <= '0';
180               state_rx <= WAIT_SFLUSHRX;
181               enviando := '0';
182             end if;
183
184           when WAIT_SFLUSHRX =>
185             if(spi_Ready='1') then
186               spiReset <= '1';
187               state_rx <= DATA_RECEIVED;
188             end if;
189
190           when DATA_RECEIVED =>
191             frameRx_busy <= '0';
192             state_rx <= IDLE;
193
```

```
194           when others =>
195             state_rx <= IDLE;
196
197         end case;
198     end if;
199
200 end process;
201 end Behavioral;
```

spi_com.vhd

```
1  ----------------------------------------------------
2  -- Company:        UFRGS - Univ. Fed. do RS
3  -- Engineer:       Rodrigo Schmidt Allgayer
4  --
5  -- Module Name:     spi_com - Structure
6  -- Project Name:    FemtoNode
7  ----------------------------------------------------
8  library ieee;
9  use ieee.std_logic_1164.all;
10 use ieee.numeric_std.all;
11
12 package spi_com_package is
13
14   component spi_com is
15     generic(DATA_WIDTH : INTEGER := 24);
16     port (
17       reset     : in  STD_LOGIC;
18       clk       : in  STD_LOGIC;
19       SCLK      : out STD_LOGIC;
20       CSn       : out STD_LOGIC;
21       MOSI      : out  STD_LOGIC;
22       MISO      : in STD_LOGIC;
23       DataTx    : in STD_LOGIC_VECTOR(DATA_WIDTH - 1 downto 0);
24       DataRx    : out STD_LOGIC_VECTOR(DATA_WIDTH - 1 downto 0);
25       Start     : in STD_LOGIC;
26       Status    : out  STD_LOGIC;
27       DataLength : in INTEGER
28       );
29   end component;
30 end spi_com_package;
31
32 library ieee;
33 use ieee.std_logic_1164.all;
34 use ieee.numeric_std.all;
35
36 entity spi_com is
37   generic(DATA_WIDTH : INTEGER := 24);
38   port (
39     reset     : in  STD_LOGIC;
40     clk       : in  STD_LOGIC;
41     SCLK      : out STD_LOGIC;
42     CSn       : out STD_LOGIC;
43     MOSI      : out STD_LOGIC;
44     MISO      : in STD_LOGIC;
45     DataTx    : in STD_LOGIC_VECTOR(DATA_WIDTH - 1 downto 0);
46     DataRx    : out STD_LOGIC_VECTOR(DATA_WIDTH - 1 downto 0);
47     Start     : in STD_LOGIC;
48    Status   : out  STD_LOGIC;       -- 1:Free   0:Busy
49    DataLength : in INTEGER
50    );
51 end spi_com;
52
53 architecture behavorial of spi_com is
54     type state_type is (idle, txBit, checkFinished);
55     signal state : state_type;
56
57 begin
58   process(clk, reset, Start)
59
60     variable index : INTEGER := (DataLength-1);
61     variable dataLen : INTEGER := DataLength;
62
63   begin
64     if reset = '1' then
65       DataRx <= (others => '0');
66       SCLK <= '1';
67       CSn <= '1';
68     Status <= '1';
69       MOSI <= 'Z';
70       dataLen := DataLength;
71       index := (DataLength-1);
72     else
73       if(clk'event and clk = '1') then
74         case state is
75           when idle =>
76         SCLK <= '0';
77             if(Start = '1') then
78               state <= txBit;
79               CSn <= '0';
80           Status <='0';
81             else
82               state <= idle;
83               CSn <= '1';
84           Status <='1';
85               index := (DataLength-1);
86             end if;
87
88           when txBit =>
89             SCLK <= '0';
90             MOSI <= DataTx(index);
91             state <= checkFinished;
92
93           when checkFinished =>
94             if(index = 0) then
95               state <= idle;
96               SCLK <= '1';
97           DataRx(index) <= MISO;
```

```
98                else
99              DataRx(index) <= MISO;
100                 state <= txBit;
101                 index := index - 1;
102                 SCLK <= '1';
103               end if;
104
105             when others => null;
106           end case;
107         end if;
108       end if;
109
110     end process;
111
112 end behavorial;
```

modulo_wireless.ucf

```
#PACE: Start of Constraints generated by PACE
#PACE: Start of PACE I/O Pin Assignments

#SPARTAN3AN Starter Kit
#NET "clock"   LOC = "E12";
#NET "reset"   LOC = "V8";
#NET "CCA"     LOC = "AA19";
#NET "CSN"     LOC = "AA21";
#NET "FIFO"    LOC = "AB19";
#NET "MISO"    LOC = "V15";
#NET "MOSI"    LOC = "W16";
#NET "SCLK"    LOC = "V16";
#NET "SFD"     LOC = "AB21";
#NET "VREG_en" LOC = "V14";
#NET "FIFOp"     LOC = "W18";
#NET "RESETn"  LOC = "Y18";
#NET "spartan_led1"  LOC = "R20";
#NET "spartan_led2"  LOC = "T19";
#NET "spartan_led3"  LOC = "U20";
#NET "spartan_led4"  LOC = "U19";
#NET "spartan_led5"  LOC = "V19";
#NET "spartan_led6"  LOC = "V20";
#NET "spartan_led7"  LOC = "Y22";
#NET "spartan_led8"  LOC = "W21";

#VIRTEXII
#NET "clock"  LOC = "AH15" | IOSTANDARD = LVCMOS25;
#NET "clock" PERIOD = 31.25ns HIGH 50%;
#NET "CCA"     LOC = "N3";
#NET "CSn"     LOC = "P1";
#NET "FIFO"    LOC = "R7";
#NET "FIFOp"   LOC = "P2";
#NET "MISO"    LOC = "N2";
#NET "MOSI"    LOC = "R9";
#NET "RESETn" LOC = "L4";
#NET "SCLK"    LOC = "M3";
#NET "SFD"     LOC = "P7";
#NET "VREG_en" LOC = "N5";
#NET "LED1"   LOC = "AC4";
#NET "LED2"   LOC = "AC3";
#NET "LED3"   LOC = "AA6";
#NET "LED4"   LOC = "AA5";

#PACE: Start of PACE Area Constraints
#PACE: Start of PACE Prohibit Constraints
#PACE: End of Constraints generated by PACE
```