Luiz Filipe Menezes Vieira
Orientador: Antônio Otávio Fernandes
Co-Orientador: Diógenes Cecílio da Silva

Middleware para Sistemas Embutidos e Redes de Sensores

Dissertação apresentada ao Curso de Pós-graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Belo Horizonte
15 de Abril de 2004

Resumo

Este trabalho descreve uma plataforma de software que facilita o desenvolvimento de aplicações para Rede de Sensores Sem Fio(RSSF). Essas redes consistem de centenas ou milhares de dispositivos autônomos e compactos, chamados nós sensores que são capazes de medir condições ambientais. Um nó sensor é um sistema embutido composto de uma unidade de bateria, processamento, sensoriamento (um ou mais sensores) e uma unidade de comunicação (por exemplo rádio). As aplicações de RSSF variam desde a área militar até a saúde e estão fortemente acopladas com o mundo físico, ao contrário de aplicações convencionais. Além disso, as aplicações desenvolvidas para rede de sensores precisam levar em consideração as restrições inerentes das RSSF, o que dificulta a construção de aplicações para estas redes. A solução para facilitar o trabalho do desenvolvedor é construir um "middleware" que encapsule algumas restrições e modularize o desenvolvimento da aplicação e um sistema operacional(SO) que abstraia as restrições do hardware. A primeira parte deste trabalho consiste no YATOS , um SO dedicado para RSSF e para o projeto SensorNet. A segunda parte consiste do WISDOM, um middleware que facilita o desenvolvimento de aplicações para múltiplas plataformas de redes de sensores.

Abstract

This work describes a software platform that easy the application development for Wireless Sensor Network(WSN). WSN consist of hundred or thousands of autonomous and compact devices denominated sensor nodes. A sensor node is an embedded system formed by a battery unit, processing and sensing units (one or more sensors) and a communication unit (i.e. radio). WSN's applications vary from military to health and they are tightly coupled with the physical world, as opposed to conventional applications. Besides that, the applications developed to wireless sensor networks need to consider the restrictions inherent to WSN. In order to easy the developer work, we construct a middleware that encapsulate some restrictions and modularize the application's development and an operating system(OS) that abstract the hardware's restrictions. The first part of this work consist of YATOS , an operating system dedicated to wireless sensor network and to the SensorNet's project. The second part consist of WISDOM, a middleware that allow the development of applications to multiples platforms of sensor networks.

Agradecimentos

É muito difícil ter que mencionar a todas as pessoas aqui. Espero ter recompensado, durante o dia-a-dia, o imenso apoio que recebi de pessoas tão especiais.

Agradeço ao meu orientador Antônio Otávio, ao meu co-orientador Diógenes, ao meus amigos professores Loureiro e Claudionor, e a doutora Linnyer, que me ajudaram na minha formação e na formação deste trabalho. Com suas contribuições, eles me ajudaram a tornam este trabalho uma realidade.

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Código

Capítulo 1

Introdução

> "I do not fear computers. I fear the lack of them."
>
> Isaac Asimov

Redes de Sensores sem Fio(RSSFs) permitem monitorar as condições ambientais e possuem inúmeras aplicações que variam da área médica a área militar. O desenvolvimento de uma aplicação para estas redes envolve conhecimento e controle dos recursos de hardware e software disponíveis. Além desta tarefa complexa, a construção de uma aplicação para RSSF para múltiplas plataformas exige que a aplicação seje desenvolvida especificamente para cada uma das plataforma. Essas dificuldades justificam a elaboração de um "middleware". Ele facilita a construção de aplicações para RSSF. Junto com o middleware, também é desenvolvido o YATOS , um sistema operacional(SO) dedicado a RSSF que também facilita o desenvolvimento de aplicações para RSSF, uma vez que permite abstrair restrições do hardware.

O recente desenvolvimento da tecnologia MEMS(micro electro-mechanical system) e o avanço na comunicação sem fio, permitiram [28, 26] a criação de Rede de Sensores sem Fio. Estas [56] são formadas por um grande número de pequenos dispositivos dotados de capacidade de processamento, comunicação e sensoriamento. Estes dispositivos têm fortes restrições quanto a memória, capacidade de processamento e principalmente energia. Neste texto eles serão denominados nós sensores. A Figura 1.1 [108] mostra o nó sensor Mica ao lado de uma moeda de US$0,25 que está na figura apenas como parâmetro de comparação de tamanho.

O hardware básico de um nó sensor é composto de transceptor, processador, um ou mais sensores, memória e bateria. Esses componentes permitem ao nó comunicar (enviar e receber

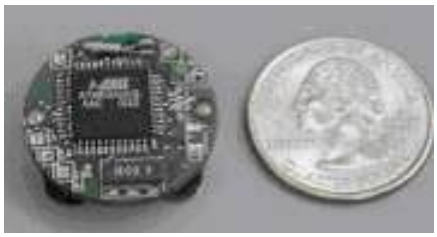
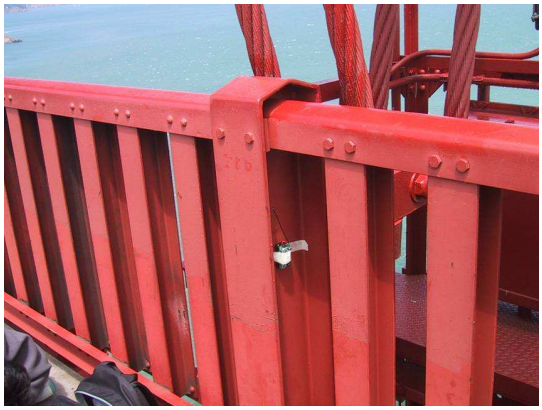


Figura 1.1: *Nó sensor Mica Dot.*

informações), executar tarefas que requerem processamento além de permitir realizar funções de sensoriamento.

O custo dos nós sensores é pequeno e justifica sua utilização em diversas áreas. Diversas aplicações têm sido desenvolvidas utilizando um ou mais tipos de nós sensores. Alguns exemplos são: sensoriamento do meio-ambiente, monitoração em indústrias petroquímicas, fábricas, refinarias, coleta de dados em áreas de difícil acesso como, por exemplo, outros planetas, aplicações militares como detectar inimigos ou presença de material perigoso como gás venenoso ou radiação, detecção, localização e estimativa de danos em estruturas (também conhecido como SHM - Structure Health Monitoring [84]) como mostrado na Figura 1.2(a), monitoração de habitat [58, 57] como mostrado na Figura 1.2(b), etc.

(a) Monitoração de Habitat



(b) Monitoração de Estrutura

Figura 1.2: *Fotos de duas aplicações de Redes de Sensores Fio.*

As aplicações desenvolvidas para rede de sensores precisam levar em consideração as restrições inerentes das redes de sensores. Além disso, diversas plataformas computacionais vem sendo desenvolvidas. Podemos citar [103, 112] Mica [36, 37], Mantis [68, 2, 3], Eyes [25, 66], PushPin [14, 53], Smartdust [107], μAMPS [99], PC104 [71], GNOMES [109], Sensor Webs

do Jet Propulsion Lab da NASA [5, 24], WINS(Wireless Integrated Network Sensors) [110] e o SensorNet [88]. Cada uma possui características próprias. O desenvolvimento de uma aplicação para mais de uma plataforma requer que o mesmo esforço na construção de uma aplicação seja realizado para uma nova plataforma. Visando facilitar o trabalho do desenvolvedor, é possível construir um "middleware"que encapsule algumas restrições e modularize o desenvolvimento da aplicação. Este "middleware" é o trabalho proposto.

O nó sensor é um sistema embutido e nele que será executado o código da aplicação desenvolvido através do "middleware". Com o objetivo de permitir o desenvolvimento de aplicações para a plataforma computacional do projeto SensorNet[1], também foi desenvolvido um sistema operacional dedicado ao hardware deste projeto e que atendesse as necessidades e requisitos das redes de sensores sem fio.

1.1 Objetivos

O objetivo deste trabalho é desenvolver uma plataforma de software que facilite o desenvolvimento de aplicações para redes de sensores. Este trabalho está dividido em duas partes: o desenvolvimento de um sistema operacional(SO) e o do middleware.

A primeira parte deste trabalho é desenvolver um sistema operacional que encapsule os recursos e dissensões do hardware e que forme uma camada de abstração entre o hardware e a aplicação. O sistema operacional deve prover os serviços para as aplicações. A Figura 1.3 mostra que o sistema operacional interage com o hardware e a aplicação.

Figura 1.3: *O Sistema Operacional situa-se entre o hardware e aplicação.*

Conforme [9], qualquer programa embutido irá se beneficiar da inclusão de um sistema operacional. O programador não será mais responsável pelo controle, monitoramento e gerência

[1]SensorNet é a sigla do projeto, financiado pelo CNPq, sob o título "Arquitetura, Protocolos, Gerenciamento e Aplicações em Redes de Sensores Sem Fio". Participam deste projeto as instituições UFMG e UFPE.

do estado do processador e seus periféricos. O sistema operacional facilita o desenvolvimento de aplicações, tornando o código fonte mais simples de entender e manter.

Como descrito em [56], a identificação e implementação de primitivas de sistemas operacionais para redes de sensores sem fio é uma área inicial de pesquisa. O sistema operacional deve ser pequeno para caber na memória, consumir pouca energia, executar sem bloqueio atendendo aos requisitos das redes de sensores que possuem restrições de memória e, principalmente, energia. Junto com o sistema operacional é desenvolvido a API(Application Programming Interface) do sistema que possui os serviços do SO. Alguns desses serviços são implementados diretamente no hardware, mas só são visíveis para as aplicações através do SO.

A segunda parte deste trabalho tem como objetivo desenvolver um middleware que facilite o desenvolvimento de aplicações para redes de sensores, incluindo o SO projetado e construído.

A execução das aplicações devem ser de forma eficiente e isto pode ser mais facilmente alcançado utilizando o middleware. O middleware irá gerar o código da aplicação a partir de uma especificação definida pelo usuário em um modelo de programação intuitivo para as plataformas desejadas. Este código será compilador pelos compiladores específicos de cada plataforma e integrados ao middleware. Esta compilação permitirá que o código executado seje eficiente. Opções de otimização poderão ser acionadas para o hardware específico.

A rede de sensores pode ser formada por vários tipos de nós, e a aplicação deve ser capaz de executar em qualquer um dos nós, e de forma concorrente, com outras aplicações. Dessa forma, o middleware deve gerar código para múltiplas plataformas.

Para o caso da plataforma do SensorNet, o código gerado será carregado junto com o sistema operacional que é desenvolvido especificamente para esta arquitetura. Juntos, todo o processo de desenvolvimento de software estará completo.

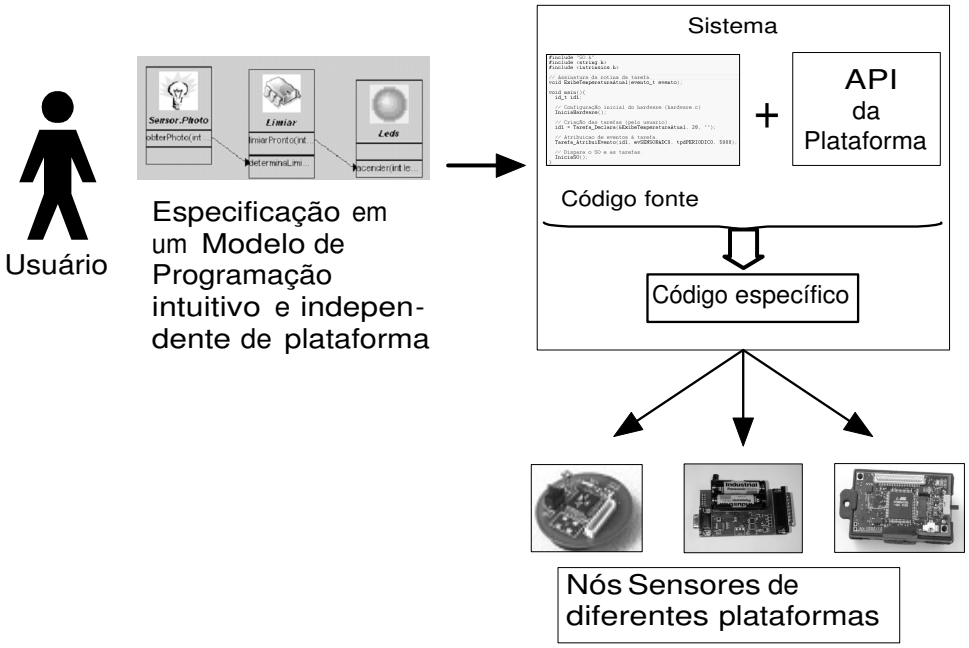


Figura 1.4: *Processo do Middleware*

A Figura 1.4 mostra as várias etapas do processo realizado pelo middleware. O usuário irá desenvolver sua aplicação em um modelo de programação como intuitivo. A caixa API mostrada na Figura 1.5 representa a API previamente definida que permite acesso as funcionalidades providas pelo sistema. O middleware será capaz de gerar o código, utilizando a API para ser executado no nó sensor.

O middleware tem como objetivo agir como uma camada de abstração, entre a aplicação e a plataforma computacional(hardware e software) - por isso o nome middleware.

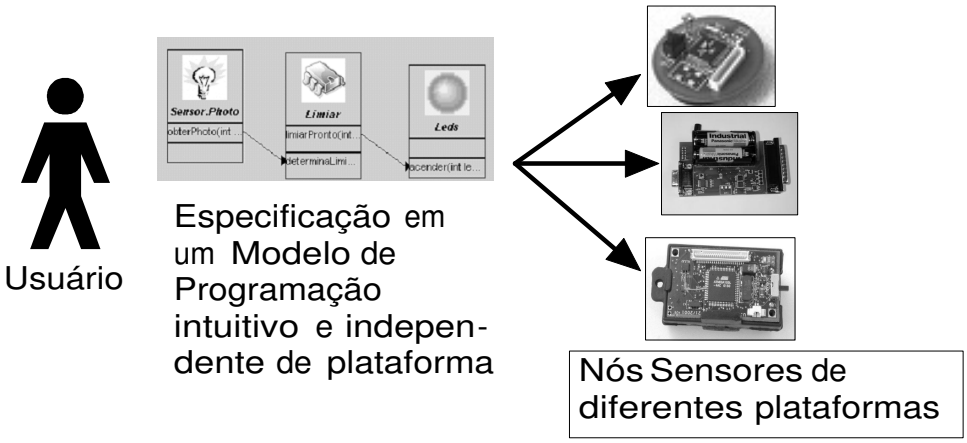


Figura 1.5: *Visão do usuário*

O middleware encapsulará o processo tornando-o mais claro para o usuário que não precisará de se preocupar com alguns detalhes, como projetar a aplicação para ser executada em qualquer nó sensor. A Figura 1.5 mostra a visão do usuário, que é mais simples que a Figura 1.4. O desenvolvedor de aplicações (usuário do middleware) não se preocupa com detalhes da arquitetura de um nó sensor.

Como resultado, espera-se que o usuário seja capaz de desenvolver sua aplicação utilizando o middleware desenvolvido e esta seja executada eficientemente nos nós sensores que são de plataformas previamente suportadas pelo middleware.

Este texto está organizado da seguinte forma. O Capítulo 2 descreve as redes de sensores sem fio, incluindo o hardware de um nó sensor e alguns exemplos de aplicações reais. O Capítulo 3 descreve os trabalhos relacionados, tanto quanto ao middleware quanto ao sistema operacional. O Capítulo 4 apresenta o sistema operacional YATOS (Yet Another Tiny Operating System) e O Capítulo 5 o middleware WISDOM(Wireless Sensor Network Development Code Middleware). No Capítulo 6 são descritos os resultados. Finalmente, no Capítulo 7, o trabalho é resumido, apresentando conclusões e trabalhos futuros.

Capítulo 2

Rede de Sensores Sem Fio

"Nothing shocks me. I'm a scientist."

Harrison Ford

2.1 Visão Geral

Esta seção apresenta uma visão geral de Redes de Sensores Sem Fio(RSSF), que está representada na Figura 2.1. Estas são redes formadas por nós sensores e por pelos menos um ponto de comunicação denominado estação base. Os nós sensores são compostos de sensores que monitoram o ambiente de acordo com a aplicação, e de equipamento rádio para comunicação com a estação base ou com outros nós sensores. Para isso é necessário um microcontrolador para efetuar o processamento requerido. O objetivo destas redes é coletar informações do ambiente. Nós sensores podem ser jogados em uma área que se deseja monitorar, acordam, se testam, estabelecem comunicação dinâmica entre eles, compondo uma rede ad hoc.

Rede de Sensores Sem Fio usualmente não possuem infra-estrutura pré-estabelecida, como redes de celulares ou redes locais sem fio. RSSFs são uma rede ad hoc, uma vez que sua topologia é dinâmica.

A estação base age como meio de comunicação entre a rede de sensores e o usuário final. Nós sensores usualmente não possuem um canal de comunicação direto com a estação base, o que demanda nós intermediários a atuarem como roteadores para enviar mensagens (comunicação multihop).

RSSF possuem recursos limitados, sendo energia o mais importante destes. Cada nó sensor possui uma bateria com capacidade limitada. É praticamente inviável recarregar manualmente

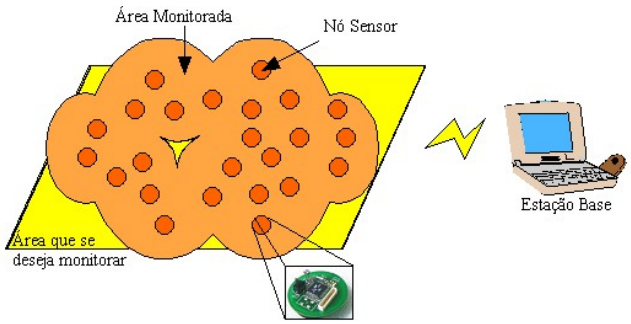


Figura 2.1: *Visão Geral de RSSF*

todas as baterias, uma vez que, RSSF podem ser compostas por milhares de nós sensores e, além disso, estes podem estar em locais inacessíveis. Dessa forma, o foco de projeto em RSSF, do hardware aos protocolos de redes, é o uso eficiente de energia.

RSSF é uma área de pesquisa recente e apresenta diversos temas de pesquisa, como gerência de redes [83], escalonamento de nós sensores na rede [105], obtenção do mapa de energia [104, 113], gerência de energia [22], construção de modelos de consumo de energia [62], posicionamento de nós sensores [101] e, inclusive, plataformas de hardware e software, entre outros.

A próxima seção descreve sucintamente os dispositivos que formam RSSF.

2.2 Descrição do Hardware de um Nó Sensor

Um nó sensor é um elemento computacional com capacidade de processamento, memória, interface de comunicação sem fio, além de um ou mais sensores, como mostrado na Figura 2.2. A capacidade de processamento origina-se da presença de um microprocessador no nó sensor. Este possui uma memória interna ao microcontrolador e, em geral, uma memória externa que age como memória secundária. A comunicação sem fio é realizada através de um transceptor. A bateria é a fonte de alimentação dos componentes de hardware. Os sensores são os responsáveis por mapearem eventos do mundo real em informações que serão coletadas pela rede.

A Figura 2.3 apresenta exemplos de hardware de nós sensores, mostrando algumas características da Família Motes.

Cada nó sensor de uma RSSF pode ser equipado com um ou mais dispositivos sensores [38], tais como acústico, sísmico, infravermelho, vídeo-câmera, estresse mecânico, calor, temperatura, radiação e pressão. A Figura 2.4 mostra alguns tipos de sensores que podem estar presentes em

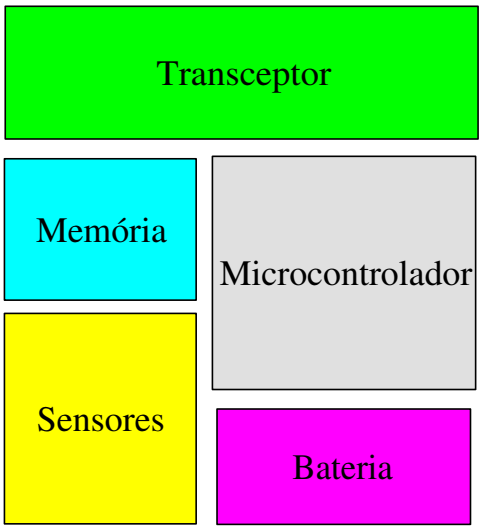


Figura 2.2: *Componentes de hardware de um nó sensor*

um nó sensor, tais como: um microfone (Figura 2.4(a)) que pode ser usado para captar o som ambiente, um fotômetro (Figura 2.4(b)) que é capaz de medir a quantidade de fótons recebidos, um diodo sensível a luz (Figura 2.4(c)), acelerometro (Figura 2.4(d)) que é um sensor capaz de medir a aceleração aplicada a ele, um sensor de umidade é mostrado na Figura 2.4(e), um sensor de radiação na Figura 2.4(f), um sensor de luz ultra-violeta na Figura 2.4(g), um sensor de força na Figura 2.4(h) e um de pressão na Figura 2.4(i). O termistor (mostrado na Figura 2.4(j)) é um sensor capaz de medir a temperatura. E um sensor de infra-vermelho é mostrado na Figura 2.4(k).

Como apontado por [56], a tendência é produzir esses sensores em larga escala, barateando o seu custo,levando a novas melhorias, capacidades e reduzindo o tamanho dos nós sensores como mostra a Figura 2.5. A visão é que redes de sensores sem fio se tornem disponíveis executando as tarefas mais diferentes possíveis e comunicando entre si.

É importante ressaltar que estes nós têm forte restrições quanto a memória, capacidade de processamento e principalmente energia, sendo desejável que possuam mecanismos de economia de energia, tais como desligar componentes de hardware que não estão sendo usados e colocar o processador no modo de mais baixo consumo quando este não estiver sendo usado. Essas funções que devem estar presentes no sistema operacional, para que a aplicação tenha controle do consumo de energia. A maneira como isto é feito em cada hardware e SO é fortemente dependente da aplicação; daí a necessidade do desenvolvimento do middleware que possibilita ao desenvolvedor da aplicação se abstrair dessas particularidades. O potencial desses sistemas

Mote Type	*WeC*	*René*	*René 2*	*Dot*	*Mica*	*MicaDot*
Microcontroller						
Type	AT90LS8535		ATmega163		ATmega128	
Program memory (KB)	8		16		128	
RAM (KB)	0.5		1		4	
Nonvolatile storage						
Chip	24LC256				AT45DB041B	
Connection type	I^2C				SPI	
Size (KB)	32				512	
Default power source						
Type	Lithium	Alkaline	Alkaline	Lithium	Alkaline	Lithium
Size	CR2450	2 x AA	2 x AA	CR2032	2 x AA	3B45
Capacity (mAh)	575	2850	2850	225	2850	1000
Communication						
Radio	TR1000					CC1000
Radio speed (kbps)	10	10	10	10	40	38.4
Modulation type	OOK				ASK	FSK

Figura 2.3: *Exemplos de hardware de nós sensores*

embutidos em rede pode ser visto pelo conjunto de aplicações que podem ser empregados e explicados a seguir.

2.3 Arquitetura de Software

É necessário entender os requisitos computacionais de uma aplicação para Redes de Sensores Sem Fio [52]. Uma aplicação deve atender os seguintes fatores:

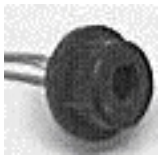
Pequeno O código executável das aplicações devem caber na memória dos nós sensores.

Expressivo um sistema de programação deve permitir um variedade de aplicações.

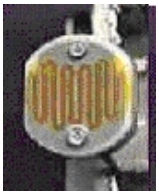
Conciso Aplicações devem ser pequenas, e conservar a banda da rede.

Resilientes Uma aplicação não pode parar de funcionar ou causar danos no nó sensor.

Eficiente Eficiência de energia e de comunicação são essenciais.



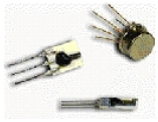
(a) Microfone



(b) Fotômetro

(c) Diodo sensível a luz

(d) Acelerometro

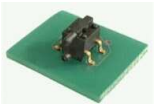


(e) Sensor de umidade

(f) Sensor de Radiação

(g) Sensor de Ultra-violeta

(h) Sensor de Força

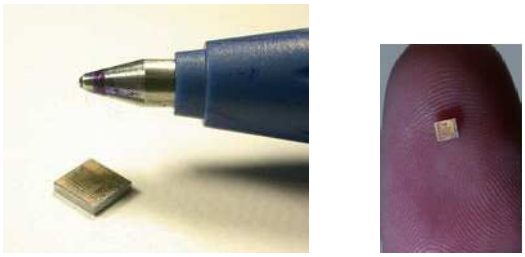


(i) Sensor de Pressão

(j) Termistor

(k) Sensor de Infra-Vermelho

Figura 2.4: *Alguns sensores que podem ser colocados em um nó sensor.*



(a) (b)

Figura 2.5: *A tendência é o tamanho do nó sensor ser reduzido*

2.4 Aplicações

Redes de sensores Sem Fio tem o potencial de serem aplicadas em várias áreas. RSSFs permitem monitorar uma grande variedade de condições ambientais que incluem as seguintes [28]:

- temperatura,
- umidade,
- movimentos veiculares,
- condições de iluminação,
- pressão,
- nível de ruído,
- presença ou ausência de certos tipos de objetos,
- nível de estresse mecânico em objetos atachados,
- características correntes como velocidade, direção e tamanho de um objeto.

Nós sensores podem ser usados para monitoração contínua, detecção de eventos, localização e controle local de atuadores. As áreas de aplicação das RSSFs são proeminentes e se destacam a área militar, meio-ambiente, saúde, automação residencial, monitoração de estruturas e aplicações comerciais. Cada uma dessas aplicações é exemplificada a seguir.

2.4.1 Aplicações na Área Militar

RSSF podem ser integradas em sistemas militares de comando, controle, comunicação, computação, vigilância e reconhecimento. Algumas aplicações de rede de sensores seriam: monitoração de forças amigas, equipamentos e munição; vigilância do campo de batalha, reconhecimento de terreno e das forças opostas; detecção de alvos; avaliação de estragos causados em uma batalha e detecção de ataques biológicos, químicos ou nucleares.

2.4.2 Aplicações no Meio-ambiente

Algumas aplicações de meio-ambiente de redes de sensores incluem o rastreamento do movimento dos pássaros, pequenos animais, e insetos; monitoração de condições ambientais que afetam colheitas e plantio; irrigação; detecção de componentes químicos ou biológicos; agricultura de precisão; monitoração da água, solo e atmosfera; detecção de fogo em florestas; pesquisa meteorológica ou geofísica; detecção de inundação; mapeamento da bio-complexidade ambiental e estudo da poluição[6, 10, 12, 32, 15, 17, 18, 20, 33, 42, 43, 28, 97, 91, 107, 29]. Embora satélites e sensores presentes em aviões possam observar grandes biodiversidades, como espécies de plantas dominantes em uma região, eles não possuem granularidade suficiente para observar pequenas biodiversidades que compõe a maior parte de ecossistemas [48]. Um exemplo de mapeamento de biodiversidade acontece na James Reserve, Califórnia, Estados Unidos [18], onde protótipos de RSSF foram construídos para coletar dados biológicos, mapear a diversidade de espécies e a estrutura dos ecossistemas.

Um exemplo de estudo do comportamento de animais utilizando RSSF ocorre na Great Duck Island, Maine [57, 73, 58]. Em agosto de 2002 [74], pesquisadores da UCB/Intel Research Laboratory posicionaram nós sensores para estudar o comportamento de uma espécie de ave[1] mostrada na Figura 2.6. O fato dos nós sensores poderem ser dispostos estrategicamente, aleatoriamente e densamente em florestas, permite que redes de sensores detectem a exata localização de incêndios e informem ao usuário antes que o fogo se espalhe incontrolavelmente.

O grupo de pesquisa da Intel e do Agri-Food Canada (Departamento de Agricultura do Canada) estão usando uma rede de sensores para medir temperatura em um vinhedo de 50 acres no sul de British Columbia [65]. A leitura de temperatura feita pela rede ajuda a identificar alvos para o combate a geada, permitindo utilizar de medidas como molhar as plantas para evitar os danos da geada.

[1] Storm Petrel, genus *Oceanodroma* [52]

Figura 2.6: *Ave Petrel*

O projeto Sensor Web (da NASA) colocou nós sensores ”Sensorwebs” no Parque Nacional Sevilleta National Wildlife Refuge na região central do Novo México. O deserto árido da região provê não só um ambiente extremo para testar a tecnologia, mas também permite realizar pesquisa ecológica significativa na área. A Figura 2.7 mostra uma foto deste experimento, um nó sensor entre árvores Juniper[2].

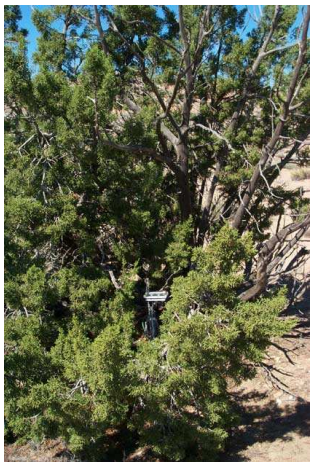


Figura 2.7: *Árvore Juniper com Sensor Web.*

Uma companhia de San Diego, CA,[93] desenvolveu um sistema de irrigação com nó sensores que medem a umidade do solo e iniciam a irrigação quando o gramado precisa de água. O

[2]Juniperus monosperma

sistema, denominado S.Sense, é mostrado na Figura 2.8. Os sensores são colocados no solo e possuem comunicação sem fio com o controlador.

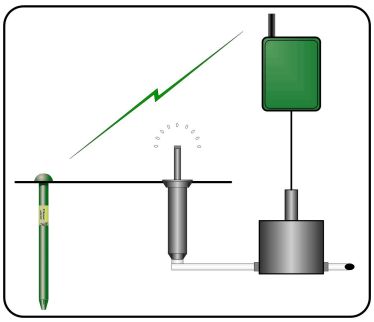


Figura 2.8: *Sistema S.Sense*

No parque nacional Hawaii Volcanoes, foi desenvolvida um aplicação que utiliza uma rede de sensores sem fio para ajudar na recuperação e preservação de espécies em extinção. Os nós sensores foram dispostos em posições pré-determinadas para coletar dados sobre a temperatura do ar, luz, vento e umidade relativa. Estes dados são importantes para a monitoração do desenvolvimento de espécies e, particularmente, para o entendimento dos fatores que alteram o crescimento destes. O uso de redes de sensores diminui a interferência humana e o risco de acidente com qualquer uma das espécies [11].

2.4.3 Aplicações na Saúde

Na área da saúde, redes de sensores podem ser utilizadas em algumas aplicações como prover interface para deficientes; monitorar pacientes; diagnósticar distúrbios; administrar drogas em hospitais; monitorar o movimento de pequenos animais ou insetos; controle, monitoração e localização de pacientes e médicos em hospitais [15, 46, 67, 80, 107].

Loren et al. [86] descreve uma aplicação biomédica que está sendo desenvolvida, a retina artificial. No projeto SSIM(Smart Sensors and Integrated Microsystems), um chip de prótese de retina composto de 100 microsensores é construído e implementado no olho humano. Isto permite pacientes com nenhuma visão ou visão limitada possam enxergar a um nível aceitável. A comunicação sem fio é necessária para controle do “feedback”, identificação da imagem e validação que está ilustrada na Figura 2.9.

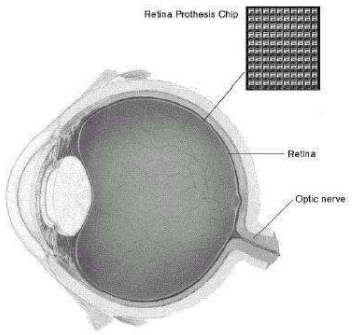


Figura 2.9: *Localização de uma rede de sensores dentro do olho.*

2.4.4 Aplicações em Automação Residencial

Em automação residencial, nós sensores podem ser embutidos em eletrodomésticos como aspirador de pó, microondas, geladeiras, etc[72]. Estes sensores dentro de dispositivos domésticos podem interagir entre si e com uma rede externa via Internet ou Satellite. Elas permitiriam ao usuário gerenciar eletrodomésticos localmente ou remotamente [38].

2.4.5 Monitoração de Estruturas

SHM (Structure Health Monitoring) é outro domínio importante de aplicações de rede de sensores. Só no Canadá e Estados Unidos o valor estimado da estrutura civil é de US$25 trilhões [51]. A tecnologia SHM trabalha com a monitoração e identificação de falhas em estruturas como pontes e prédios e tem evoluído desde que foi proposta no anos 1990s, e recentemente, a abordagem baseada em rede de sensores é promissora e apresenta algumas vantagens como o baixo custo para disposição e de manutenção, grande área de cobertura, boa resolução espacial.

Um projeto em andamento é o uso de redes de sensores para avaliar a vibração da ponte Golden Gate em São Francisco, Califórnia [50]. Uma placa de sensores foi desenvolvida contendo um acelerometro e um conversor ADC de 16 bits (mostrada na Figura 2.10) e esta foi conectada ao Mica Motes e colocada na ponte para avaliar as vibrações da ponte devido aos ventos e/ou possíveis terremotos. A Figura 2.11 mostra a proposta da arquitetura da rede. Um conjunto de nós sensores é posicionado manualmente na ponte e os dados são coletados e enviados a um computador que armazena e processa os dados coletados.

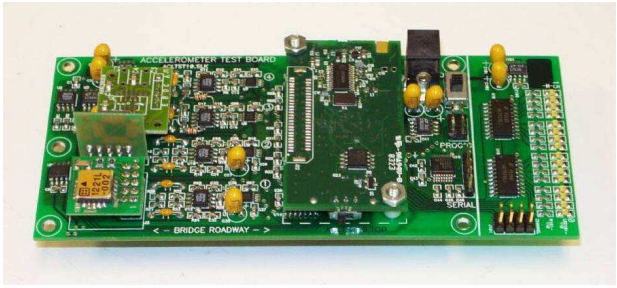


Figura 2.10: *Placa de sensor desenvolvida com acelerômetro.*

Figura 2.11: *Redes de Sensores na ponte de São Francisco.*

2.4.6 Aplicações Comerciais

RSSF também podem ser utilizadas em aplicações comerciais. Automação de vendas em supermercados, monitoração de fatiga do material; criação de teclados virtuais; manutenção de inventário; monitoração de qualidade de produtos; construção de escritórios inteligentes; controle de robôs; museus interativos; controle e automação de processos industriais; monitoração de áreas de desastres; diagnósticos de máquinas; controle local de atuadores; detecção e monitoração de carros roubados; detecção e vigilância de veículos são alguns exemplos [6, 15, 20, 28, 27, 46, 77, 79, 80, 76, 89, 107].

2.4.7 Outras aplicações

Locher [55] propõem o uso de redes de sensores em crianças no jardim de infância. Um crachá composto por um nó sensor com capacidade de coleta de dados e comunicação, denominado iBadge e mostrado na Figura 2.12, é colocado na roupa dos alunos. Isto permite monitorar os

alunos e acompanhar a interação entre alunos.

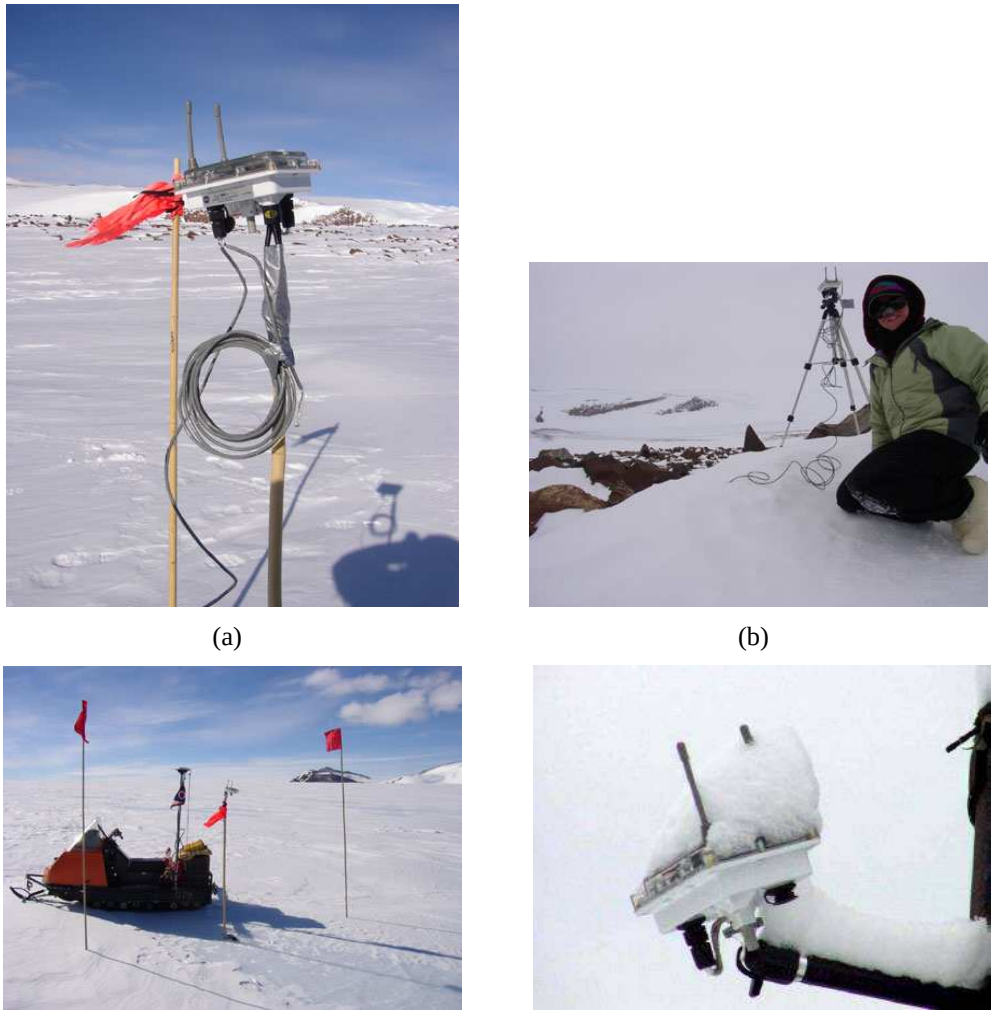
Figura 2.12: *iBadge.*

Redes de sensores tem sido testadas nos mais diferentes ambientes [4]. Um time de pesquisadores, em cooperação com pesquisadores procurando por meteoritos na Antártica [8], montou uma rede com Sensor Webs [5] em uma remota região no lado leste da Antártica por três semanas (Dezembro de 2002 a Janeiro de 2003) [23]. Esta aplicação mostrou que sistemas Sensor Web podem ser dispostos e funcionarem confiavelmente em condições ásperas (baixa temperatura, ventos constantes e ar seco). Quatorze nós sensores do Sensor Web foram distribuídos perto de uma estação de pesquisa por uma distância de 2 quilômetros. A temperatura típica da região onde os nós sensores foram colocados era de -10 °C, e chegavam a -20 °C. A Figura 2.13 mostra fotos deste experimento. Devido ao forte vento, os nós foram amarrados a bambus.

Outro experimento com Sensor Webs [4] envolve monitoração das condições afetando o crescimento de plantas em áreas que não são facilmente controladas, como estufas. Estudando o efeito de microclimas nas plantas irá permitir cientistas a compreender melhor maneiras de como otimizar a qualidade. Cada nó sensor mede níveis de luz, temperatura e umidade do ar, com a opção de medir a temperatura e umidade do solo. A Figura 2.14 mostra fotos do experimento em uma plantação de alface.

Outras aplicações em outras áreas também podem utilizar RSSF, como por exemplo, exploração espacial, processamento químico, ajuda em resgates, monitoração do tráfego de veículos em rodovias.

Esta seção apresentou uma variada aplicabilidade de rede de sensores sem fio, demonstrando a necessidade de um middleware que facilite o desenvolvimento de aplicações. O próximo capítulo mostra os trabalhos relacionados, tanto do middleware quanto do sistema operacional.



(a) (b)

(c) (d)

Figura 2.13: *Fotos da aplicação de uma RSSF na Antártica.*

(a) (b)

Figura 2.14: *Fotos da aplicação de uma RSSF na plantação de alface.*

Capítulo 3

Trabalhos Relacionados

> "Intellectuals solve problems, geniuses prevent them."
>
> Albert Einstein

Este capítulo apresenta os trabalhos relacionados ao middleware (Seção 3.1) e o sistema operacional(Seção 3.2).

3.1 Middleware

Várias estratégias já foram descritas na literatura para uma implementação de um *middleware* para redes de sensores sem fio [81]. Entretanto, essas abordagens são muito recentes e estão sujeitas a um estudo mais aprofundado.

Nas seções seguintes encontra-se o resultado da avaliação dos trabalhos atuais. Na seção 3.1.1, traça-se uma análise dos *middlewares* existentes, o que resulta numa classificação dos mesmos. Na seção 3.1.2, verifica-se a estrutura de programas simples para RSSF's expressados em linguagem de alto nível, analisando-se questões como a relação expressividade versus custo computacional. Na última seção, compara-se esses trabalhos ao novo *middleware*, mostrando-se como esse trabalho tenta superar as limitações apresentadas.

3.1.1 Análise dos *middlewares* existentes

Os *middlewares* voltados para RSSF's existentes na literatura foram enquadrados neste trabalho em duas categorias: *programação holística* e *scripting*. Após a descrição detalhada dessas cate-

gorias, o *middleware* deste trabalho será comparado às mesmas, de modo a enfatizar as diferenças dessa nova abordagem.

3.1.1.1 Programação holística

Quando o *middleware* é capaz de apoiar um programa que descreve o comportamento da rede de sensores como um todo, atribuindo tarefas aos grupos de sensores que farão parte dessa rede, esse *middleware* é capaz de oferecer uma programação holística das RSSF's. Mais especificamente, os nós podem ser agrupados em função de sua posição geográfica ou de suas características, como por exemplo pelo tipo de sensores que possuem. Esses grupos, então, interagem entre si de acordo com as funções declaradas para cada um. Esse tipo de programação foi proposta na linguagem c@t [87], para sistemas massivamente distribuídos.

A linguagem c@t é capaz de representar programas relativamente complexos com muita simplicidade e isso reduz bastante o tempo de desenvolvimento desses programas. O *middleware* possui recursos para invocação de procedimentos em um outro grupo e para seleção de grupos com base em critérios determinados em tempo de execução. Esses dois conjuntos de funcionalidades são embutidos em todo programa gerado pelo *middleware*.

Apesar da proposta ser interessante, a programação holística não foi utilizada e testada em experimentos reais. Da forma como foi concebido, esse *middleware* não é apropriado para a implementação de aplicações para controle da infraestrutura da rede.

3.1.1.2 *Scripting*

Para certos tipos de algoritmos, como os de rastreamento de alvo, e para uma customização do *software* da rede em tempo de execução, identificou-se que a capacidade de migração do código pela rede seria interessante. Essa característica é denominada mobilidade de código, e já vem sendo estudada no contexto da Internet. Essa pesquisa mostra que, para atingir a segurança e portabilidade necessária para os scripts poderem funcionar em um ambiente heterogêneo, é útil escrever esses programas (denominados *agentes*) como *scripts*. Os *scripts*, por sua vez, são executadas em máquinas virtuais, as quais são capazes de interpretar esses *scripts* e fornecer um acesso limitado aos recursos de hardware, que não comprometam a segurança do mesmo. Sobre esse estudo, surgiram os *middlewares* baseados em *scripting*.

Um *software* construído com essa idéia foi o Maté [52]. Ele é uma máquina virtual sobre o sistema operacional TinyOS [37] e provê mobilidade de código a um baixo custo de transmissão.

Além disso, o modelo de programação define que todo o endereçamento é feito a partir de pilhas, sendo muito semelhante aos bytecodes Java [54], oferecendo proteção de memória. Ele foi desenhado para lidar com problemas de sincronismo de acesso a dados compartilhados existentes na linguagem nesC [21], simplificando a programação dos nós sensores.

Sensorware [13] utiliza a mesma idéia, mas com a linguagem Tcl [69]. Este middleware utiliza o conceito de rede pró-ativa, na qual as aplicações operam de forma autônoma de acordo com eventos detectados como estímulos físicos e chegada de mensagens. O Sensorware, entretanto, não pode ser utilizado em nós sensores com severas restrições de memória. Ele foi desenvolvido para a arquitetura iPAQ 3670, a qual possui muito mais recursos de hardware que os nós sensores no estado da arte [21].

Os *scripts* são mais seguros e possuem menor tamanho de código. Os comandos dos *scripts* possuem maior semântica se comparados aos de uma linguagem imperativa como C. Ao serem interpretados, os *scripts* são desdobrados em várias instruções de baixo nível. Esse conjunto de comandos disponíveis é controlado, o qual não oferece acesso direto à memória da máquina, por exemplo. Os interpretadores são robustos, de tal forma a não violarem a segurança da máquina.

Uma grande desvantagem dos *scripts* é que a sua interpretação consome mais tempo e energia do que a execução de códigos binários. Algoritmos mais complexos, que demandem maior processamento, ou programas que executem por um longo tempo nos nós, vêem-se prejudicados nesse ambiente. Entretanto, os *middlewares* de *scripting* são desenhados para aplicações que exijam mobilidade de código, e a argumentação gira em torno de que os *scripts* impõem um gasto bem menor de energia na sua transmissão do que os códigos executáveis correspondentes, o que compensaria o dispêndio de energia na interpretação.

3.1.2 Linguagens de programação para RSSF's

Durante a criação dos primeiros nós sensores (o primeiro da família Motes foi lançado em 1999 [21]), a programação desses nós era feita através da linguagem C com rotinas específicas para o *hardware* alvo. Esse tipo de programação lembra muito a construção de *drivers* para dispositivos, e é muito próxima à linguagem de montagem.

Surgiram, então, várias abordagens para programação de redes de sensores sem fio. A mais difundida atualmente é a linguagem nesC [21], que é uma variante da linguagem C com recursos de modularidade e suporte à declaração de seções atômicas. Outra característica peculiar é a não-alocação dinâmica de memória, o que facilita a otimização do programa. Os módulos expressam

tanto a interface entre software e hardware como as aplicações em si. A modularidade permite o reuso de código em várias aplicações, trazendo menor tempo de desenvolvimento e maior robustez. O código 3.1 mostra um programa que escreve o valor atual de um contador tanto para os leds como para a pilha de comunicação do rádio.

Código 3.1 Código-fonte de um programa em nesC.

```
1  configuration CntToLedsAndRfm {
2  }
3  implementation {
4    components Main, Counter, IntToLeds, IntToRfm, TimerC;
5
6    Main.StdControl -> Counter.StdControl;
7    Main.StdControl -> IntToLeds.StdControl;
8    Main.StdControl -> IntToRfm.StdControl;
9    Main.StdControl -> TimerC.StdControl;
10   Counter.Timer -> TimerC.Timer[unique("Timer")];
11   IntToLeds <- Counter.IntOutput;
12   Counter.IntOutput -> IntToRfm;
13 }
```

A programação em nesC é dirigida por eventos, no sentido em que os procedimentos são executados em resposta a interrupções do hardware ou por outros módulos. Computações mais complexas são delegadas a abstrações denominadas tarefas, as quais são executadas quando não há eventos a serem tratados. Esse modelo de programação é bastante diferente das linguagens comuns, nas quais existem um fluxo de execução definido. Respondendo de forma assíncrona aos eventos, a seqüência de chamadas aos comandos só é conhecida em tempo de execução. Isso exige maior cuidado do programador, pois essa imprevisibilidade pode gerar erros difíceis de detectar e corrigir.

É muito importante lembrar que essas linguagens estão muito ligadas ao sistema operacional ou ao *middleware* que as suporta. Threads, por exemplo, são suportadas pelo sistema operacional Mantis [2] e utilizadas largamente em seus programas. Programação por threads já é usada em outros contextos, o que facilita o entendimento e a portabilidade de trechos de código para essa plataforma. O programa 3.2 testa os *leds* do nó Nymph.

Uma linguagem totalmente diferente das apresentadas acima é a c@t. Baseada na linguagem Scheme [49], ela permite uma programação holística das RSSF's, como discutido na seção 3.1.1.1. O exemplo de código 3.3 ilustra o alto poder de expressão dessa linguagem. O programa liga um grupo de sensores equipados com ventiladores toda vez que a temperatura ultrapassa 70

Código 3.2 Código-fonte de um programa em Mantis.

```
#include <inttypes.h>
#include <stdio.h>
#include "scheduler.h"
#include "led.h"

void blink_leds() {
  uint16_t i;

  while(1) {
    for(i = 0; i < 0xffff; i++);
      led_red_toggle();
      led_green_toggle();
      led_yellow_toggle();
  }
}

void start(void) {
  thread_new(blink_leds, 128, PRIORITY_NORMAL);
}
```

graus Celsius.

Código 3.3 Código-fonte de um programa em C@t language.

```
(declare-device sensor ((processor ‘‘16F628’’)))
(declare-device fan ((processor‘‘18F2320’’)))
(declare-cluster temp-control ((sensor 160) (fan 90)))
(define (@ (= device sensor). . .) float temperature 0)
(define (@ (= device sensor). . .) void (monitor)
(if (>= temperature 70)
(activate (@ (grammar relational)
(filter (and (= type fan) (> battery 0.5) ))))))
(define (@ (= device fan). . .) void (activate)
(set! RB7 #x80))
```

As linguagens oferecidas pelo Maté e SensorWare (códigos 3.4 e 3.5) também são bastante expressivas, mas aqui o objetivo é diferente da linguagem c@t: possibilitar a implementação de algoritmos distribuídos. Através de comandos embutidos na linguagem, é simples acessar os sensores ou enviar mensagens para os nós vizinhos. Para expandir as funcionalidades dessas linguagens, é preciso modificar a máquina virtual ou o interpretador que as executa, o que reduz a expansibilidade. Outro problema dessa abordagem é a falta de uma abstração para módulo, o que

dificulta a colaboração entre os *scripts*. O programa 3.4 mantém um contador que é incrementado a cada unidade de tempo do relógio. Os *leds* exibem os três bits de baixa ordem desse contador. O programa 3.5 implementa um algoritmo de agregação de dados, no qual cada nó recebe as leituras dos sensores dos seus vizinhos e, em seguida, envia a um nó pai o máximo entre essas leituras e a sua própria.

Código 3.4 Código-fonte de um programa em Maté.

```
1 pushc 1 # Push one onto operand stack
2 add # Add the one to the stored counter
3 copy # Copy the new counter value
4 pushc 7
5 and # Take bottom three bits of copy
6 putled # Set the LEDs to these three bits
7 halt
```

3.1.3 Resumo dos Trabalhos Relacionados

Scripting e *Programação holística* foram analisados. Apesar de *Scripting* ser um mecanismo que permite o desenvolvimento de aplicações independente de plataforma, visto o custo computacional da abordagem de *scripting*, ela não foi adotada pois é imprescindível que a aplicação execute de forma eficiente no nó sensor. A *Programação holística* têm o objetivo de especificar aplicações para RSSFs como um todo, mas não para cada nó sensor na rede, não sendo apropriada, além de não evitar o retrabalho no caso de múltiplas plataformas.

O *middleware* proposto nesse trabalho não se encaixa em nenhuma das duas categorias existentes. Ele não especifica a aplicação para a rede como um todo, mas para cada um dos nós sensores. Também não utiliza *scripts*, mas abstrações na especificação da aplicação. As abstrações são essenciais para facilitar e agilizar o desenvolvimento de software para RSSF's, ao mesmo tempo que código eficiente pode ser gerado para cada plataforma.

A diferença principal desse *middleware* para os demais reside na especificação de aplicações para nós sensores através de um modelo de programação estruturado, utilizando-se o conceito de módulos. Como essa especificação é feita fora do nó sensor, dispomos de poder computacional maior para apresentar ao desenvolvedor esse modelo de programação, o qual se utiliza de uma interface gráfica. Quando a especificação da aplicação é finalizada, o *middleware* é responsável por gerar código próximo à linguagem de máquina, em C ou nesC, que utilize as rotinas disponíveis no sistema operacional escolhido pelo desenvolvedor.

Código 3.5 Código-fonte de um programa em SensorWare.

```
set need_reply_from [ replicate -m]
set maxvalue [ query sensor value ]
if {$need_reply_from == ""} { send $parent $maxtemp; exit }
else { set return_reply_to $parent }
set first_time 1
while {1} {
  wait anyRadioPck // "anyRadioPck" is a predefined eventID
  if { $msg_body ==add_user } {
    if { $first_time == 1 } {
      send $parent $msg_body
      set first_time 0
    }
    set return_reply_to "$return_reply_to $msg_sender"
  }else {
    set maxvalue [expr {($maxvalue<$msg_body) ? $maxvalue :
                                                $ msg_body }]
    set n [lsearch $need_reply_from $ msg_sender]
    set need_reply_from [lreplace $need_reply_from $n $n]
  }
  foreach node $return_reply_to {
    if { ($need_reply_from=="")||($need_reply_from==$node)} {
      send $node $maxvalue
      set n [lsearch $return_reply_to $node]
      set return_reply_to [lreplace $return_reply_to $n $n]
    }
  }
  if {$return_reply_to==""} {exit}
}
```

3.2 Sistema Operacional

A princípio é apresentado e justificado porque sistemas operacionais para sistemas embutidos em geral não são adequados para redes de sensores sem fio. A seguir, são mostrados os sistemas operacionais voltados para RSSFs.

3.2.1 Sistemas Operacionais para Sistemas Embutidos

Os atuais sistemas operacionais de tempo real não atendem aos requisitos impostos pelas restrições do hardware dos nós sensores. A maioria deles possui o tamanho e performance voltados para sistemas com mais de uma ordem de grandeza acima dos sistemas embutidos que formam as re-

des de sensores sem fio. Sistemas Operacionais tradicionais de tempo real incluem VxWorks [1], WinCE [16], PalmOS [70], QNX [34], μLinux [60] e mais alguns [35] que provêm um conjunto de serviços não necessários as RSSFs.

Nome	Tamanho ROM	Plataforma Alvo
pOSEK	2K	Microcontrolador
VxWorks	286K	Strong ARM
QNX Neutrino	$>$100K	Pentium II NEC chips
QNX Realtime	100K	Pentium II 386's
Ariel	19K	ARM Thumb
CREEM	560	ATMEL 8051

Tabela 3.1: *Tabela comparativa de alguns SO.*

A Tabela 3.1 mostra as características de um conjunto desses sistemas. Muitos são baseados em micro Kernels que permitem que capacidades sejam adicionadas ou removidas de acordo com a necessidade do sistema. Eles provêem um ambiente de execução que é similar aos desktops tradicionais. Esses sistemas são escolhas comuns para PDA, telefones celulares e set-top-boxes. Entretanto, eles não chegam perto de atender os requisitos de RSSF; eles são convenientes para os PCs embutidos. Por exemplo, uma troca de contexto no QNX requer mais de 2400 ciclos em um processador 386EX a 33Mhz [35]. Além disso, consome centenas de kilobytes em memória. Estes dois aspectos são mais de uma ordem de magnitude além dos limites do sistema.

Existe também uma coleção de sistemas operacionais menores que incluem Creem [47], pOSEK [75] e Ariel [7], que são projetados para sistemas embutidos específicos, como controlador de motor e microondas. Enquanto eles permitem a preempção de tarefas, eles possuem severas restrições no modelo de execução. Por exemplo, o modelo do pOSEK é estaticamente configurado para atender os requisitos de uma aplicação específica. O que se deseja é uma arquitetura de sistema que permita um grupo heterogêneo de dispositivos (diferentes placas de sensores, interfaces de comunicação,etc.) a funcionarem eficientemente tanto em termos de tempo de execução e consumo de energia. Até mesmo o pOSEK, que atende a necessidade de caber na memória, excede os limites que existem no tempo da troca de contexto.

3.2.2 Sistemas Operacionais para Redes de Sensores

Para atender às necessidades de dispositivos com severas restrições quanto ao consumo de energia, espaço em memória e poder computacional, recentemente foram propostos e/ou desenvolvidos diferentes SOs, que utilizam diferentes abordagens ao problema. *Bertha* [53] e *Sen-*

sorWare [13] atendem algumas das demandas específicas de RSSFs, sendo descritos a seguir. *TinyOS*(Tiny Operating System) [35] e *PeerOS*(Pre-Emptive Eyes Real-Time Operating System) [64] são específicos para RSSFs. Os dois são dirigidos a eventos, o que representa um ganho na economia de energia. TinyOS [94] é parte do projeto WEBS [95] da UC Berkeley. Como o nome sugere, ele é um pequeno sistemas operacional. PeerOS é um SO desenvolvido pelo grupo europeu EYES [66] (Universidade de Twente - Holanda) para o projeto de seu nó sensor.

Salvo [78] é um produto comercial voltado para o mercado crescente de sistemas embutidos. Ele está disponível para muitas plataformas (família 8051, Atmel AVR, Motorola M68HC11, TI MSP430, Microchip PIC12) como um produto comercial, compradores podem ter suporte técnico. A desvantagem de um produto comercial é que tudo deve ser comprado e uma licença ser paga. A seguir são descritos os sistemas operacionais Bertha, SensorWare, PeerOS e TinyOS.

3.2.2.1 Bertha

Bertha OS [53] é um SO para redes de sensores de nós idênticos, utilizando a arquitetura de hardware *Pushpin* [14]. É desenvolvido pelo Responsive Environment Group, no MIT Media Lab. Seu modelo de programação é baseado em um sistema de agentes móveis [46] muito simples, com três componentes: fragmentos de processos (PFrags), um sistema de quadro de boletins (BBS) e um vigia de vizinhança (NW). A Figura 3.1 mostra um conjunto de *Pushpin*.

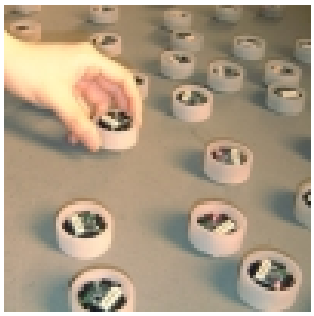


Figura 3.1: *Foto de um conjunto de PushPin, executando o Bertha.*

Fragmentos de processos, ou *PFrags*, são entidades de programação autônomas e móveis, escritas em C, que podem interagir com seu ambiente local, podendo migrar de um nó para outro. Para isso, eles possuem código executável e estado persistente e podem se comunicar através do BBS do nó. A granularidade de execução é definida em turnos, que equivalem a uma

execução do método update de um PFrag. Todos os PFrags residentes em um nó num dado momento são executados, um de cada vez, em “round-robin”. Apesar da concorrência não ter sido implementada, um método de interrupção por um temporizador de “watchdog” foi sugerido pelos autores.

Um sistema de quadro de boletins, ou *BBS*, está disponível para os PFrags em cada nó da rede. Ele é usado para permitir a comunicação entre PFrags, que postam mensagens nele. Elas podem ser lidas por qualquer PFrag, mas só podem ser alteradas ou removidas pelo seu autor.

Sinopses do BBS dos nós vizinhos são espelhados no vigia de vizinhança, ou *NW*, do nó local. Quando um PFrag posta alguma mensagem em um BBS, aquele pode escolher que uma sinopse daquela postagem esteja disponível no NW de todos os nós vizinhos. Através de NWs, cada PFrag pode, por exemplo, identificar por quais nós ele já passou. A Figura 3.2 mostra a organização da memória, divida em Fragmentos de processos, sistema de quadro de boletins e sinopses do BBS.

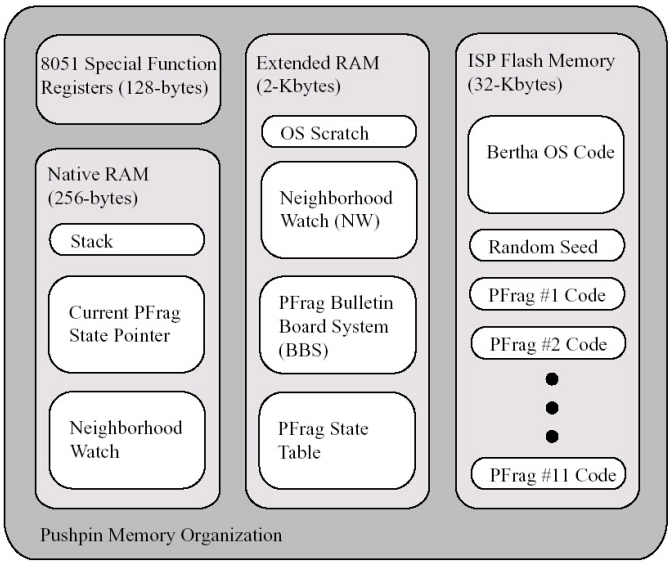


Figura 3.2: *Organização da Memória do Pushpin executando Bertha.*

Infelizmente, Bertha é voltado especificamente para a plataforma PushPin, não provê mecanismos de comunicação sem fio, não é portado para o MSP430, e limita o tamanho de código dos programas em 2K.

3.2.2.2 SensorWare

Desenvolvido na Universidade da Califórnia, *SensorWare* [13] é um arcabouço para RSSFs. Suas características principais são: mobilidade de código através de scripts móveis e separação de funcionalidades correlatas em unidades distintas.

Scripts móveis são entidades do SensorWare que podem se deslocar de um nó para outro numa RSSF. Usam o modelo de mobilidade fraca. Isso significa que um script executa até um certo ponto num nó e, ao transferir-se para outro, não continua a execução do ponto onde parou. Em vez disso, cada script móvel possui pontos de entrada bem definidos.

A maioria dos comandos e funções são agrupadas em APIs distintas. Elas possuem funcionalidades que fornecem acesso a um recurso ou serviço do nó sensor. Comandos e funções que não possuam um tema específico ou que sejam de uso mais geral localizam-se no módulo-núcleo. As principais funcionalidades desenvolvidas para o SensorWare são:

Rede: provê a comunicação entre os scripts.

Sensoriamento: acesso e compartilhamento de dados dos sensores.

Temporização: inicia e reinicia temporizadores de tempo real.

Mobilidade: provê funções de mobilidade aos scripts, acesso a seus dados e à memória do nó no qual esteja residindo.

SensorWare utiliza recursos que consomem memória (scripts), não possui um sistema operacional dedicado a rede de sensores, não é portado para o MSP430, sendo inapropriado para uso no Projeto. A idéia de mobilidade de código é interessante e, provavelmente, deve ser adicionado aos serviços implementados no SO.

3.2.2.3 PeerOS

PeerOS [25, 64] foi desenvolvido na Universidade de Twente - Holanda. Projetado para o nó sensor EYES [100, 66] que utiliza o microcontrolador MSP430, rádio TR1000, possui porta serial de comunicação, uma memória EEPROM e um potenciometro digital para ajuste da potência do sinal do rádio.

Seu funcionamento baseia-se nos seguintes princípios: operação dirigida a eventos, divisão da execução em tarefas e separação de funcionalidades correlatas em unidades distintas. A Figura 3.3 mostra o nó sensor EYES, que executa o *PeerOS*.

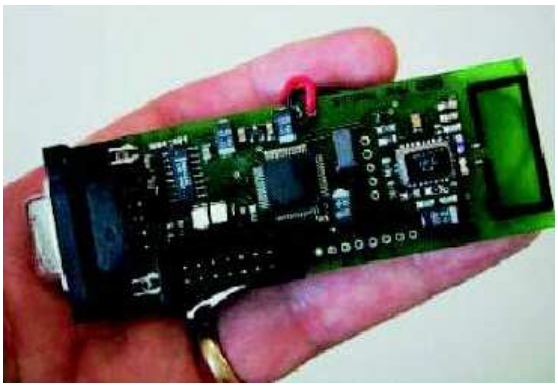


Figura 3.3: *Nó sensor EYES.*

Usando o modelo de execução dirigido por eventos, obtém-se código pequeno e consome-se pouca energia. Assim, o código é estruturado em módulos, que serão executados em resposta a eventos externos. Quando não há mais desses eventos para cuidar, o sistema passa a operar em um modo de economia de energia.

Os serviços fornecidos pelo sistema são agrupados em APIs distintas. As funcionalidades principais das APIs existentes no EYES são as seguintes:

Informação Local: fornece acesso aos dados obtidos do nó sensor.

Rede: provê as funções básicas para enviar e receber pacotes de dados.

Sistema Operacional: fornece informações sobre a localização do nó e ativa, durante sua inicialização, a recepção de módulos de software para programação dinâmica do nó.

Apesar do PeerOS ser voltado para o mesmo micro-controlador, sua API, principalmente o rádio, difere significativamente do Projeto, além de não atender completamente os requisitos descritos na Seção 4.1, como por exemplo, oferecer multi-tarefa baseado em prioridade.

3.2.2.4 TinyOS

O TinyOS, desenvolvido inicialmente na Universidade da Califórnia, em Berkeley [35], é um sistema operacional muito simples e compacto, baseado em eventos, desenvolvido para redes de sensores sem fio. Ele é utilizado por uma grande comunidade de usuários [82]. O *TinyOS* [37] provê execução concorrente para redes de sensores embutidos, com recursos de hardware escassos usando a arquitetura Motes [18] que utiliza o microprocessador ATMega da Atmel©. O

TinyOS também é uma plataforma de software de código aberto projetada para apoiar operações concorrentemente intensivas, usando requisitos mínimos de hardware. A seguir, uma sucinta descrição das características do TinyOS são apresentadas.

TinyOS é um sistema operacional simples com código aberto. O TinyOS possui um escalonador de tarefas, que é uma simples FIFO, utilizando uma estrutura de dados fixa e limitada. O escalonador é não-preemptivo e não possui mecanismos mais sofisticados como fila de prioridades. A simplicidade do TinyOS significa ausência de gerenciamento de processos, inexistência de memória virtual e alocação dinâmica de memória, mas ela permite que o tamanho do TinyOS em memória seje pequeno, atendendo as necessidades das redes de sensores sem fio. O código fonte é aberto e atualmente as modificações no código e nos componentes do TinyOS são feitos pelo grupo Intel-Berkeley Research Lab. O TinyOS é implementado em nesC [21], uma extensão da linguagem emphC projetada para incorporar os conceitos estruturais do TinyOS e seu modelo de execução. Os conceitos básicos deste modelo são:

Separação entre construção e composição: aplicativos são formados por componentes, os quais são combinados para criar aplicativos mais complexos.

Especificação de funcionalidade através de interfaces: interfaces podem ser fornecidas ou usadas pelos componentes. As interfaces fornecidas representam a funcionalidade que o componente provê ao aplicativo; as interfaces usadas representam a funcionalidade necessária ao componente para executar seu trabalho.

Interfaces são bi-direcionais: interfaces especificam um conjunto de funções que serão implementadas pelo componente provedor da interface (i.e., comandos) e outro conjunto que será implementado pelo componente usuário da interface (i.e., eventos).

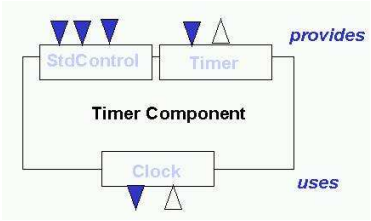
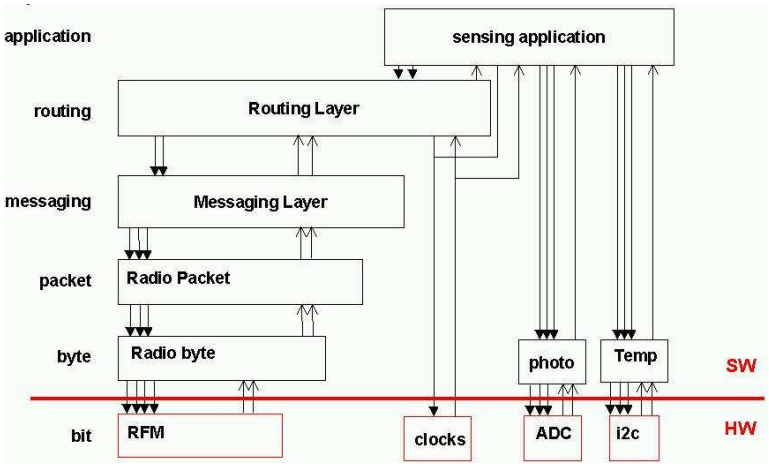


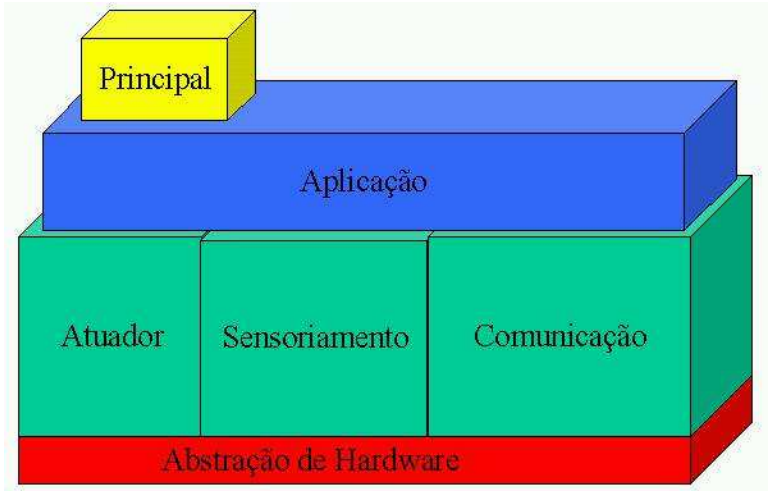
Figura 3.4: *Interface especifica funções de um componente.*

TinyOS possui uma arquitetura baseada em componentes. Uma configuração completa do sistema no TinyOS consiste de um programa composto de uma aplicação e dos componentes

do TinyOS. Uma aplicação é na verdade um grafo de componentes agrupados hierarquicamente como mostra a Figura 3.5. Os componentes interagem através das interfaces, que são bi-direcionais.



(a)



(b)

Figura 3.5: *Uma aplicação no TinyOS é um grafo de componentes.*

Todo aplicativo possui pelo menos um arquivo de configuração e um de implementação. O primeiro especifica o conjunto de componentes do aplicativo e como eles se invocam. No segundo estão listadas as interfaces fornecidas e as usadas por um componente.

Um aplicativo usa um ou mais componentes, sendo possível reutilizar alguns componentes mais simples para criar outros mais elaborados. Isso cria uma hierarquia de camadas, na qual componentes em níveis altos na hierarquia originam comandos para componentes em níveis

baixos, onde estes sinalizam eventos para aqueles. Os componentes de nível mais baixo representam o próprio hardware.

TinyOS possui um modelo de concorrência baseado em eventos. Concorrência é obtida através do uso de eventos e tarefas.

Tarefas de um componente são atômicas entre si, executando até seu término, mas podem ser interrompidas por eventos externos. Podem executar comandos de componentes em níveis baixos na hierarquia, sinalizar eventos para componentes em níveis altos e agendar a execução de outras tarefas em um componente.

Eventos são generalizações de tratadores de interrupção, propagando processamento para cima na hierarquia (através da sinalização de outros eventos) ou para baixo (por meio da execução de comandos). Eventos são executados quando sinalizados, interrompendo a execução de uma tarefa ou outro evento.

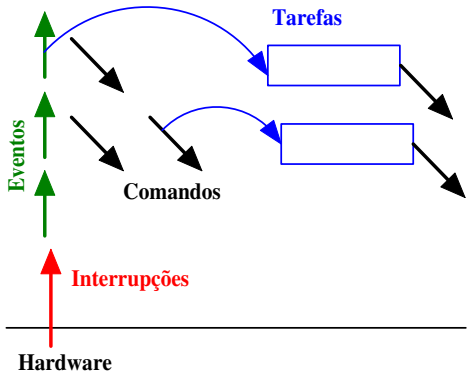


Figura 3.6: *Fluxo de execução entre tarefas e eventos.*

Seu projeto modular permite aos usuários a mantê-lo o menor possível ao custo de funcionalidades. TinyOS foi construído para uma plataforma muito específica e iria requerer um trabalho substancial de re-escrita de código para executar na plataforma SensorNet. Além de algumas modificações na estrutura do código.

3.2.2.5 Necessidade de um Sistema Operacional

Muito pode ser feito com os sistemas operacionais e hardware disponíveis, porém a combinação de características desejadas não existe em nenhum produto ainda. A Tabela 3.2 resume os trabalhos relacionados.

	Bertha	SensorWare	PeerOS	TinyOS
Plataforma	PushPin	SensorWare	EYES	Mica Motes
Processador	Cygnal 8051	StrongARM 1100	MSP430	ATMega128

Tabela 3.2: *Tabela de trabalhos relacionados.*

Além disso, o desenvolvimento da própria tecnologia é um dos objetivos da pesquisa em si, da qual faz parte este trabalho. O Capítulo 4 apresenta o YATOS - o sistema operacional desenvolvido. Conforme visto nos trabalhos relacionados, o TinyOS já apresenta o nome de sistema operacional pequeno, por isso, o sistema desenvolvido foi nomeado YATOS (Yet Another Tiny Operating System). No entanto, o nome Yet Another não implica que é apenas mais um sistema operacional, sem inovação. Ele é o primeiro a atender os requisitos descritos na seção 4.1 como ser dirigidos a eventos, oferecendo multi-tarefas baseada em prioridade e fácil de usar.

Capítulo 4

Sistema Operacional YATOS

> "When solving a "panic" you must first ask yourself what you were doing that could possibly frighten an operating system."
>
> Peter van der Linden

Um sistema operacional é um programa que gerência os recursos do hardware. Seu objetivo é facilitar o desenvolvimento de aplicativos em uma dada plataforma, evitando do programador ter de se preocupar com detalhes da arquitetura dos nós e cuidando dos problemas e limitações explicitados anteriormente. O kernel desses nós deve fornecer ao programador rotinas para seu acesso e controle, provendo primitivas que permitam desenvolver seus aplicativos. Além disso, deve ser pequeno para caber na memória, consumir pouca energia e executar sem bloqueio. O sistema operacional desenvolvido é voltado para RSSFs cujos nós são estáticos. Ou seja, nenhuma funcionalidade foi acrescida para o caso de serem móveis.

4.1 Requisitos

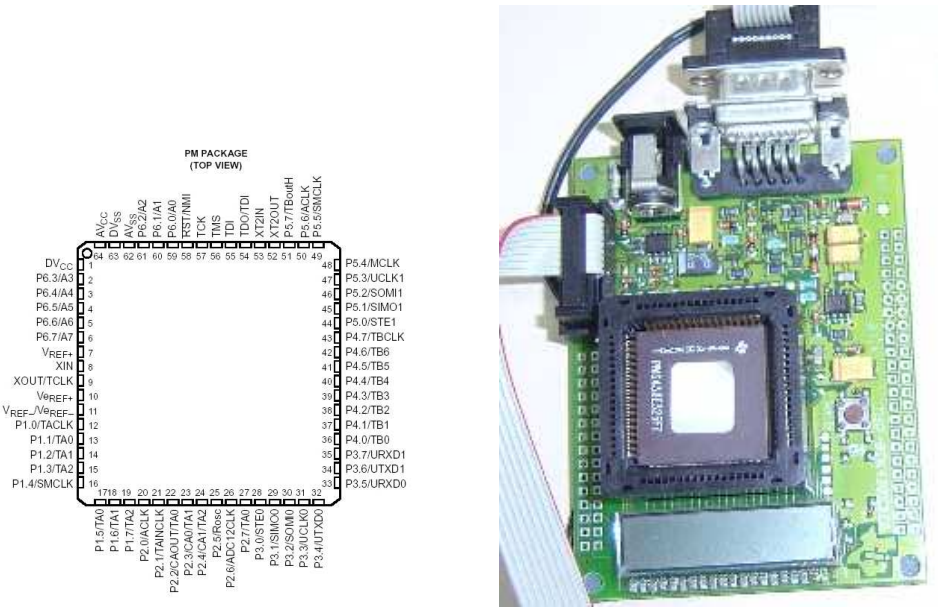
Na seção 3.2.2, pôde ser visto que existem diferentes sistemas operacionais que podem ser aplicáveis ao paradigma de RSSFs. No entanto, há algumas características que não são encontradas em nenhum deles ao mesmo tempo, quais sejam:

Ser dirigido por eventos: Um sistema dirigido por eventos, pode entrar no modo de menor consumo de energia sempre que possível. Além disso, elimina a espera ocupada. A tarefa

esperando por algum recurso, ao invés de realizar espera ocupada, pode “dormir” e ser acordada por um evento que notifica a liberação do recurso. Conforme [92], um sistema dirigido a eventos chega a economizar até 12 vezes mais energia que um sistema tradicional e propósito geral.

Ocupar pouca memória de um nó sensor: O hardware de um nó sensor possui memória limitada. Neste espaço, além do código do sistema operacional, tem o código das aplicações e os dados coletados. Por isso, o espaço ocupado pelo sistema operacional deve ser o menor possível. Para a plataforma do SensorNet, há 2 Kbytes de memória RAM e 60Kbytes de ROM.

Voltado para a plataforma SensorNet: O objetivo principal do sistema operacional é formar uma plataforma de software para o projeto. O microprocessador utilizado é o MSP430-F149 [41], e por esta razão, o código foi desenvolvido para ele. A Figura 4.1(a) mostra o esquemático deste micro-controlador, e a Figura 4.1(b) uma foto.



(a) (b)

Figura 4.1: *Esquemático e foto do Microcontrolador.*

Consumir pouca energia: Energia é um recurso precioso em uma Rede de Sensores Sem Fio e seu consumo deve ser eficiente. O sistema operacional deve, sempre que possível, minimizar o consumo de energia dos componentes. Por exemplo, quando apenas uma

computação estiver sendo realizada, o rádio pode ser colocado no modo de mais baixo consumo de energia de modo que não interfira na computação sendo realizada. A eficiência de uma rede de sensores depende da aplicação desta. O sistema operacional deve prover os mecanismos, como permitir o processador mudar de modo de operação de energia, para que a aplicação possa gerenciar a rede eficientemente.

Oferecer multi-tarefa baseada em prioridade: Mais de uma tarefa podem querem executar em um mesmo dado intervalo de tempo. Ainda mais, elas podem ter prioridades diferentes, e/ou, terem que ser executadas em uma ordem específica (neste caso elas são atribuídas prioridades que forcem a execução na ordem definida).

O sistema operacional deve ser modular: Um sistema modular facilita a manutenção e o desenvolvimento de novos componentes. Além disso, componentes não necessários podem ser excluídos.

O sistema operacional deve ser fácil de usar: A facilidade de uso é um requisito importante. De nada adianta um sistema operacional que ninguém consegue usar ou integrar com a sua aplicação. Além disso, o tempo gasto para aprender a usá-lo não deve atrapalhar o tempo de desenvolvimento da aplicação. Nem mesmo o tempo necessário para aprender uma nova linguagem de programação e paradigma de programação. Por esta razão, o YATOS foi desenvolvido em *C* [85], uma linguagem de programação bem difundida entre programadores [45] e desenvolvedores de sistemas embutidos. Para os casos em que eficiência era requerida e o código não estaria visível externamente, o desenvolvimento foi feito em linguagem de máquina.

Esta seção descreve o YATOS , suas principais estruturas (tarefas, escalonador de tarefas, eventos, lista de tarefas associadas a eventos) e funcionalidades.

4.2 Características

4.2.1 Hardware

O sistema operacional está diretamente relacionado ao hardware que ele gerencia. Aqui uma breve descrição do hardware presente no nó sensor é apresentada, contendo as restrições impostas pelo hardware ao sistema operacional. Para maiores detalhes, leia a especificação do componente do hardware ou a descrição do BEAN [102].

4.2.1.1 Energia

Os nós sensores são alimentados por pilhas ou baterias e devem ser conservadores se quiserem que sua energia durem por um tempo.

4.2.1.2 Rádio

O nó do projeto SensorNet usa um rádio CC1000 [19] . O rádio dispõe de freqüência programável (300 - 1000 MHz) com granularidade de 250 Hz, baixo consumo de corrente (Rx: 7,4 mA) e baixa tensão (2,1 - 3,6 V).

4.2.1.3 Microcontrolador

O BEAN possui um microcontrolador fabricado pela Texas Instruments, o MSP430F149 [40]. O processador MSP430F149 foi escolhido por causa do seu preço além de possuir algumas funcionalidades úteis e modo de baixo consumo de energia (low power consumption). O consumo de energia varia de $0.6mA$ no modo ativo a $0.5/muA$ no modo "idle". O processador tem uma arquitetura de 16-bits e alguns componentes de hardware que tornam mais fácil a comunicação e controle de periféricos. Ele inclui:

- 2 UARTs que podem transladar um byte em uma série de bits que podem ser enviados em intervalos regulares através de dispositivos de comunicação;
- Conversores Analógicos-Digitais (ADC) que convertem sinais analógicos em sinais digitais;
- comparadores analógicos no chip que continuamente checam se algum valor medido passa de um limiar e gera um aviso;
- dois temporizadores programáveis;
- um watchdog timer programável que pode acordar o processador em tempos regulares;
- cinco modos de economia de energia que permitem especificar partes do processador a serem desligados quando estes não forem necessários.

O MSP430 é um processador RISC cujo o modo de baixo consumo de energia é seu maior atrativo que especifica partes do chip que podem ser desligadas ou ligadas quando desejado. Isto

significa que um chip configurado apropriadamente não consome mais energia que o necessário. Quando parte do processador é desligada, ele é entra no modo Low Power(LPM). LPM0 tem o menor número de partes desligadas e é o modo que consome mais energia, enquanto LPM4 tem virtualmente tudo desligado e consome menos energia. A Figure 4.2 mostra a relação entre o de consumo de energia e os LPMs. YATOS usa LPM1 quando comunicação precisa estar ativa, LPM3 quando a comunicação está desativada e LPM4 quando desligado.

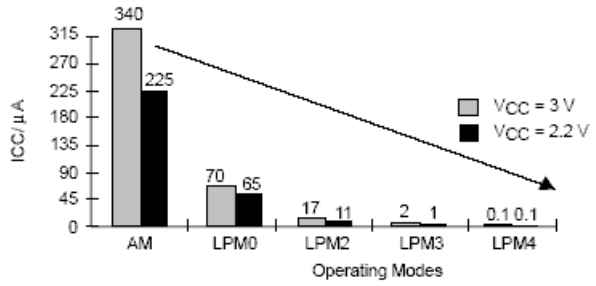


Figura 4.2: *Consumo de corrente nos diferentes modos de operação do MSP430.*

Maiores informações do processador podem ser encontradas em: [39, 41] e [40].

4.2.1.4 Memória

Uma memória FLASH externa (M25P40 [59]) faz parte do nó sensor, e funciona como uma memória secundária.

4.2.2 Tarefa

Tarefas representam as rotinas que são executados no sistema. O sistema operacional é responsável por decidir qual tarefa irá utilizar o processador em um dado momento. Sendo que ele mantém o estado de cada tarefa [9].

No YATOS , tarefas são não-preemptivas. Uma tarefa não interrompe outra tarefa. Isto implica em um ganho significativo, porque não há necessidade de troca de contexto. Uma tarefa é executada até o fim. Isto requer disciplina do programador mas garante um ganho em processamento, memória e reduz tempo de latência. Ciclos do processador não são gastos na troca de contexto, e apenas uma pilha é necessária. Não há o perigo de estourar a memória com sucessivas trocas de contexto. Além disso, o middleware pode ser usado para abstrair a necessidade de disciplina do programador provendo um conjunto de componentes.

Campo	Significado
id_t id	Identificador da tarefa
estado_t estado	Estado da tarefa
rotina_t * rotina	Rotina executada pela tarefa.
prioridade_t prioridade	Nível de prioridade da tarefa
nome_t nome	Nome da tarefa.
tipoPeriodo_t tipoPeriodo	Tipo de período de execução da tarefa
evento_t ultimoEventoDisparado	Último evento que disparou a execução da tarefa

Tabela 4.1: *Tabela com estrutura de uma Tarefa.*

Interrupções podem parar a execução de tarefas. Isto permite que o sistema seja dirigido a eventos, o que implica em um ganho de energia.

4.2.3 Escalonador de Tarefa

O escalonador de tarefas é o núcleo do sistema operacional. Ele decide qual tarefa irá executar e quando isto ocorrerá. A operação é o mais simple possível, porque simplicidade significa rapidez e tamanho otimizado. Ambos são importantes para o sistema operacional. O escalonador de tarefas trabalha com uma fila de prioridade que armazena uma lista de ponteiros de tarefas, e um flag que indica o status da tarefa.

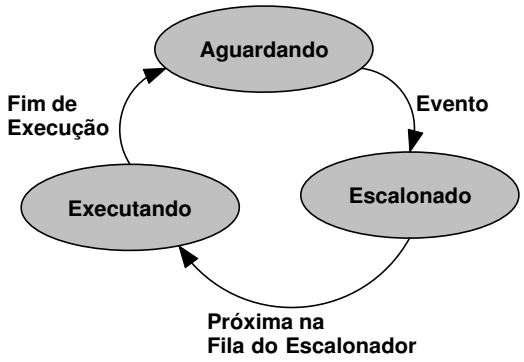


Figura 4.3: *Estados das tarefas.*

Uma tarefa pode estar executando (estado Executando), pode estar esperando na fila do escalonador para ser executada (estado Escalonado) ou pode simplesmente não estar na fila do escalonador (estado Aguardando). A Figura 4.3 mostra os estados e as transições entre estes estados. Além disso, a tarefa pode estar aguardando por um evento para ser colocada na fila do escalonador, ou não está esperando por nenhum evento. Uma tarefa que não aguarda por

eventos pode ser executada quando for postada por outra tarefa, como ocorre nos casos em que um procedimento grande é dividido em tarefas seqüenciais. Para diferenciar entre estes dois tipos de tarefas (as que aguardam por eventos e as que não aguardam), a informação de que uma tarefa está esperando por um evento ou não foi incorporada ao estado da tarefa. Dessa forma, os possíveis estados válidos de uma tarefa são:

- AGUARDANDO
  A tarefa aguarda por um evento para ser colocada na fila do escalonador.
- ESCALONADA
  A tarefa está na fila do escalonador, mas não está executando no momento.
- EXECUTANDO
  A tarefa está sendo executada e possui controle do processador.
- NÃO AGUARDANDO
  A tarefa foi criada mas não aguarda por nenhum evento.
- NA_ESCALONADA
  A tarefa está na fila do escalonador, não está executando no momento e não aguarda por nenhum evento.
- NA_EXECUTANDO
  A tarefa está executando, ela possui controle do processador e não aguarda por nenhum evento.

4.2.4 Eventos

Eventos são quaisquer interrupção que possam ocorrer. Eles podem ser usados para acordar tarefas. Um evento pode ser classificado quanto a sua periodicidade em 3 tipos: aperiódicos, periódicos e tiro único.

- aperiódicos: ocorrem sem o uso de temporizadores, através do disparo de interrupções pelo hardware.
- periódicos: ocorrem em intervalos de tempo definidos, sendo necessário o uso de temporização.
- tiro único: são eventos programados para ocorrer uma única vez.

4.2.5 Lista de tarefas associadas a eventos

A cada evento que pode ocorrer no sistema, existe uma lista com os ponteiros das tarefas que estão esperando por aquele evento. Quando o evento ocorre, as tarefas presentes na lista são colocadas na fila de prioridades do escalonador. Isto é uma otimização presente no sistema. Apesar de gastar alguns bytes a mais de memória, a freqüência e o tempo economizado evitando de percorrer a lista com todas as tarefas justifica esta implementação. Vale ressaltar que esta característica ainda existe no TinyOS como trabalho futuro [35]. Este processo é ilustrado na Figura 4.4.

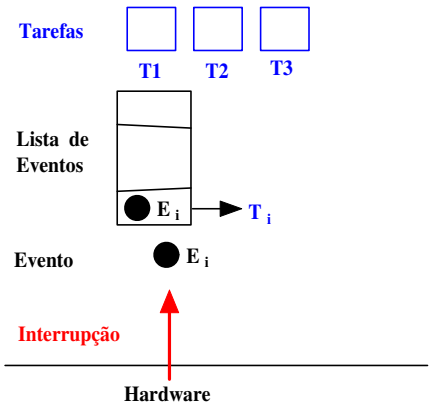


Figura 4.4: *Lista de tarefas associadas a eventos.*

4.3 Definição da API

O sistema operacional provê todas as funções básicas e serviços que um são necessários pelas camadas acima, que são descritas pela sua API. Esta foi construída de acordo com as características do hardware de um nó sensor e das funções necessárias para as aplicações. A API é formada por funções que gerenciam tarefas, os sensores, o processador, a memória, o rádio, a bateria, os temporizadores e por funções de inicialização.

A API *Tarefas* é responsável pela criação de tarefas, sua postagem e atribuição de eventos a elas. *Rádio* permite o envio e a recepção de dados através do rádio, além de configurar seu alcance. A API *Sensor* permite a obtenção de leituras dos sensores disponíveis no nó. *Memória* possibilita apagar, ler e escrever na memória Flash. A API *Portas de entrada/saída* lê e escreve dados nas portas de entrada/saída do nó. *Bateria* informa o diferencial de potencial na bateria que

serve para estimar o nível de energia disponível. Finalmente, a API *Microcontrolador* permite configurar o modo de operação do nó, além de iniciar e fechar seções críticas, que são regiões onde as interrupções são desabilitadas.

A descrição detalhada, contendo inclusive os parâmetros e significados dos tipos dos parâmetros, se encontra no Apêndice C.

4.4 Como usar o YATOS

Nessa seção, se encontra uma explicação de como um usuário poderia usar o YATOS junto com seus aplicativos. A linguagem para desenvolvimento dos aplicativos é a linguagem *C*, acrescida dos serviços do sistema operacional e da API.

4.4.1 Inicialização

Para criar seu aplicativo, o usuário deve primeiramente incluir as seguintes linhas de código (excetuando os comentários):

Código 4.1 Exemplo de código de Inicialização

```
1  #include "SO.h"    // APIs e rotinas do SO.
2  void main(){
3    /* Regiao declaracao de variáveis.
4     * (nao mostrado)
5     * Fim Regiao declaracao de variáveis. */
6
7    // Inicia dispositivos de hardware.
8    Microcontrolador_IniciaHardware();
9
10   /* Regiao declaracao de tarefas.
11    * (nao mostrado)
12    * Fim Regiao declaração de tarefas.*/
13
14   /* Regiao atribuicao de eventos.
15    * (nao mostrado)
16    * Fim Regiao datribuicao de eventos.*/
17
18   Microcontrolador_IniciaSO(); // Inicia o sistema operacional.
19 }
```

4.4.2 Declaração de tarefas

Para declarar as tarefas, será necessário criar as rotinas que serão executadas quando a tarefa for escalonada para execução. Essas rotinas não podem retornar valor e devem ter como único parâmetro *evento_t evento*, que informará à rotina em execução qual foi o evento que motivou seu disparo. Além disso, é necessário criar variáveis do tipo *id_t*, que armazenarão os identificadores das tarefas retornados pelo sistema. Para declarar as tarefas, será usado o serviço de API *Tarefa_Declara*. Esse serviço recebe como parâmetros: o ponteiro para a rotina, uma prioridade (de 1=mais baixa ... 255=mais alta) e um nome (opcional).

Código 4.2 Exemplo de declaração de Tarefa

```
1  #include "SO.h"    // APIs e rotinas do SO.
2  void rotina1(evento_t evento){
3    /* Código da rotina (nao mostrado) */
4  }
5  void rotina2(evento_t evento){
6    /* Código da rotina (nao mostrado) */
7  }
8  void main(){
9    /* Declaracao de variáveis.*/
10   id_t id1, id2;
11   // Inicia dispositivos de hardware.
12   Microcontrolador_IniciaHardware();
13
14   /* Declaracao de tarefas.*/
15   id1 = Tarefa_Declara(&rotina1, 28, "RotinaA");
16   id2 = Tarefa_Declara(&rotina2, 1, "");  // tarefa anônima.
17
18   /* Regiao atribuicao de eventos.
19    * (nao mostrado)
20    * Fim Regiao atribuicao de eventos.*/
21
22   Microcontrolador_IniciaSO(); // Inicia o sistema operacional.
23 }
```

4.4.3 Atribuição de eventos às tarefas

As tarefas devem, normalmente, executar em resposta a eventos do sistema. Assim, deve-se atribuir à uma tarefa o(s) evento(s) a que deve responder quando ocorrer(em). Além disso, algumas tarefas devem ser executadas em intervalos de tempo regulares, outras devem executar uma

única vez e, ainda, outras que são completamente aperiódicas. O serviço responsável por essas incumbências é *Tarefa_AtribuiEvento*.

Uma tarefa pode não executar, necessariamente, em resposta a um evento de sistema. Nesse caso, a tarefa que não depende do evento pode ser postada por outra tarefa, que responde a um evento de sistema. Isto é realizado pelo serviço *Tarefa_Posta*.

Utilizando os conceitos de eventos e tarefas, e a atribuição de eventos a tarefas, além da ausência de troca de contexto e o uso de uma fila de prioridades no escalonador, fizeram com que o YATOS atende-se as necessidades do Projeto.

Código 4.3 Exemplo de atribuição de eventos a tarefa

```
#include "SO.h"    // APIs e rotinas do SO.
void rotina1(evento_t evento){
  /* Código da rotina (nao mostrado) */
  return;
}
void rotina2(evento_t evento){
  /* Código da rotina (nao mostrado) */
  switch(evento){
    case evRADIO:      // dado recebido via rádio.
      /* Código (nao mostrado) */
      break;
    case evTEMPORIZADOR0:  // varredura em ADC0.
      /* Código (nao mostrado) */
      if(condicao)
        Tarefa_Posta("RotinaA");  // Postagem de tarefa.
      break;
  }
  return;
}
void main(){
  /* Declaracao de variáveis.*/
  id_t id1, id2;
  // Inicia dispositivos de hardware.
  Microcontrolador_IniciaHardware();

  /* Declaracao de tarefas.*/
  id1 = Tarefa_Declara(&rotina1, 28, "RotinaA");
  id2 = Tarefa_Declara(&rotina2, 1, "");  // tarefa anônima.

  /* Atribuicao de eventos.*/
    // aperiodica.
  Tarefa_AtribuiEventos(id2, evRADIO, tpdNULO, pdNULO);
    // periodica: 200 ms.
  Tarefa_AtribuiEventos(id2, evTEMPORIZADOR0, tpdPERIODICO, 200);

  Microcontrolador_IniciaSO(); // Inicia o sistema operacional.
}
```

Capítulo 5

Middleware

"Science is organized knowledge.
Wisdom is organized life."

Immanuel Kant

O middleware é uma camada de abstração entre a aplicação e o sistema operacional que facilita o desenvolvimento de aplicações, tipicamente distribuídas [61]. O middleware mascara a heterogeneidade de arquiteturas de computadores, sistemas operacionais, linguagens de programação, tecnologias de redes e facilita o desenvolvimento de aplicações [31]. Este capítulo apresenta a definição da arquitetura do middleware (Seção 5.1), os requisitos do middleware (Seção 5.2), o modelo de programação, uma descrição para construir aplicações (Seção 5.4) e um exemplo de uso do WISDOM(Wireless Sensor Development Code Middleware) (Seção 5.3). Algumas das telas da ferramenta WISDOM construída são apresentadas no Apêndice A. Estudos de casos de aplicações desenvolvidas através do WISDOM e que exemplificam o modelo de programação visual estabelecido são demonstrados no Capítulo 6.

5.1 Definição da Arquitetura

O objetivo do middleware é facilitar o desenvolvimento de aplicações para redes de sensores, de preferência, independente de plataforma. A Figura 5.1 mostra a definição da arquitetura. O middleware está situado entre a aplicação e as plataformas de RSSFs. Cada plataforma está representada por um retângulo. No nível mais baixo se encontra o hardware da plataforma e, acima deste, o sistema operacional. Na figura estão presentes a plataforma de UC Berkeley,

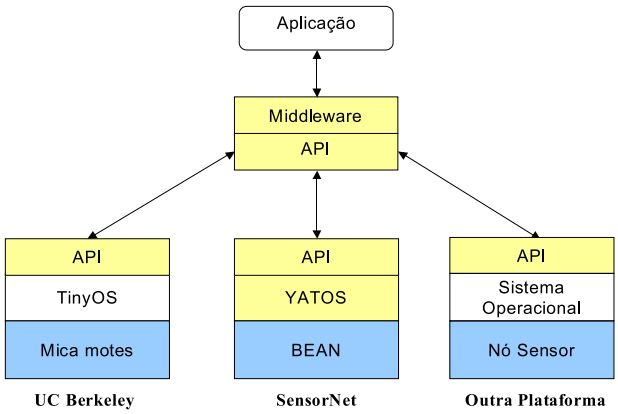


Figura 5.1: *Definição da Arquitetura*

formada pelo nó sensor Mica 2 Motes / Mica 2 dot [108] e sistema operacional TinyOS [94], a plataforma do grupo SensorNet, formada pelo nó sensor BEAN(Brazilian Energy-Efficient Architectural Node) [102] e o sistema operacional YATOS desenvolvido neste trabalho. Há também um retângulo que representa que qualquer plataforma de redes de sensores pode ser incorporada a ferramenta desde que implemente a API utilizada na geração de código. Dessa forma, outras plataformas como a do grupo europeu EYES [66] ou a do grupo MANTIS (Colorado) [68] podem ser adicionadas. Ou, inclusive, uma outra plataforma do próprio grupo SensorNet [88] formada por um nó sensor com FPGA e μLinux [60].

O uso de uma API permite ao middleware interagir com as mais diferentes plataformas, sistemas operacionais e hardwares. A Figura 5.1 também mostra o que foi implementado neste trabalho: o próprio middleware e sua API, o sistema operacional YATOS e uma API para o TinyOS. O YATOS é descrito mais detalhadamente no Capítulo 4 e a API do TinyOS no Apêndice B.

A arquitetura proposta não só gera código independente de plataforma, desde que esta implemente a API, como também gera código independente de versão.

5.2 Requisitos do middleware

O primeiro passo para a definição do *middleware* foi estudar um conjunto de requisitos aos quais ele deve atender. Esse conjunto de requisitos baseia-se na literatura, englobando aspectos já identificados como essenciais a um *middleware* eficaz, além de atender a alguns problemas identificados em abordagens existentes.

Em resumo, o *middleware* deve apoiar:

Aplicações dirigidas a eventos: esta é a característica essencial de qualquer aplicação de RSSFs. Devido ao tamanho reduzido dos nós sensores, eles são capazes de coletar dados através de sensores. Portanto, facilitar a especificação desse tipo de aplicação é o foco principal do *middleware*.

Plataformas computacionais restritas: o código-fonte gerado deve ser otimizado de forma a ocupar o mínimo de memória possível. Devido à restrição de energia, o *middleware* deve oferecer um modelo de programação que leve a programas eficientes, que efetuem computações somente quando necessário.

Diferentes sistemas operacionais: o modelo de programação do middleware deve abstrair características específicas dos sistemas operacionais para RSSFs. Essa abstração, além de tornar o programa portável, exigirá do desenvolvedor menos tempo para aprender como o acesso ao hardware é efetuado. Isso porque o modelo deverá simplificar os métodos de acesso aos dispositivos do nó sensor.

Aplicações comuns e de infra-estrutura: uma deficiência detectada nos middlewares existentes é que eles não conseguem ser genéricos o suficiente para permitir o desenvolvimento tanto de aplicações comuns, como coleta e fusão de dados, como os de infra-estrutura, como algoritmos de roteamento. Visto que esse último tipo de aplicação ainda está sendo muito estudada, a comunidade científica pode ser beneficiada com esse middleware, além dos usuários, que poderão construir aplicações de forma bastante simplificada.

Os avanços tecnológicos: a tendência dos circuitos integrados e do MEMS é de continuar a reduzir de tamanho, e de incluir tecnologias de baixo consumo de energia (“low power”). O middleware deve acompanhar este desenvolvimento e utilizar de uma API que forneça os requisitos necessários para acompanhar a evolução das redes de sensores de sensores sem fio.

Programabilidade: No nível de software o problema principal é a programabilidade [30], que se refere a habilidade de descrever um problema para um máquina em uma maneira conveniente e eficiente, independente da configuração da máquina. O middleware deve facilitar o desenvolvimento de aplicações.

Representação Gráfica: Uma representação gráfica de uma informação facilita a leitura e o entendimento de uma aplicação, e reduz o tempo de desenvolvimento. Também ajuda evitando erros, como palavras chaves escritas incorretamente.

Descrição Baseada em Componentes: Um desenho baseado em componentes permite particionar a aplicação em vários blocos. Isto aumenta a modularização da aplicação, facilita a depuração visto que os módulos podem ser testados separadamente, permite a criação de blocos reusáveis de código que são criados uma vez e utilizados várias vezes.

Extensibilidade: facilidade de inserção de novos módulos ou propriedades no sistema, inclusive de novas plataformas computacionais de redes de sensores.

5.3 Modelo de Programação

Observando os serviços básicos da API e os componentes do hardware, nota-se uma relação entre eles. Esta relação pode ser utilizada para especificar a aplicação e pode representar um novo modelo de programação formado por cinco componentes: Sensoriamento, Processamento, Comunicação, Memória e Energia.

Sensoriamento

São tarefas relacionadas a leitura de dados dos sensores. Elas podem estar conectadas a temporizadores e podem ser classificadas em [96]: contínuo, dirigido a eventos, iniciado pelo observador e híbrido. No modo contínuo, os sensores comunicam seus valores numa taxa pré-especificada. No modo dirigido a eventos, sensores reportam informação apenas quando um evento de interesse ocorrer. No modo iniciado pelo observador, os sensores apenas reportam seus resultados em resposta a uma requisição explícita de um observação. Finalmente, os três modos podem coexistir.

Processamento

Tarefas de processamento de dados, como fusão de dados, filtro de mensagens, compressão [111].

Comunicação

Tarefas de envio e recepção de mensagens, como protocolos multihop.

Memória

Tarefas de armazenamento de dados. A memória externa age como uma memória secundária, armazenando dados ou log.

Energia

A quantidade de energia em um nó sensor pode ser usada para modificar a atuação deste nó sensor. Uma tarefa pode ter a sua funcionalidade modificada devido a quantidade de energia restante no nó sensor. Por exemplo, um nó sensor com pouca energia pode deixar de participar do roteamento de pacotes.

Uma aplicação para RSSF pode ser definida como um conjunto de componentes do tipo Sensoriamento, Processamento, Comunicação, Memória e Energia. Uma definição baseada em componentes permite particionar a aplicação em vários componentes com interfaces bem definidas. Isto facilita o desenvolvimento da aplicação, bem como a sua depuração [106]. Além disso, cada componente, além do código, pode ter uma representação gráfica também.

Este modelo de programação pode ser visual também. Uma programação utilizando um modelo gráfico facilita a programação e reduz o tempo de desenvolvimento. Mais importante, este modelo é ortogonal, permite que qualquer tipo de aplicação seja descrita usando os cinco componentes. Inicialmente vamos trabalhar apenas com três componentes. Os outros dois serão incorporados futuramente, assim como um repositório de código.

Tipicamente, uma aplicação de RSSF irá coletar o dado de um sensor, processá-lo e enviá-lo. A Figura 5.2 ilustra a modelagem de uma aplicação. Esta aplicação relata os dados coletados de um sensor. Ela pode ser estendida para ler dados de mais de um sensor, realizar outras tarefas de processamento ou enviar mais de uma mensagem. A cada componente da modelagem (sensor, processamento, envio de mensagem) está associado um módulo que implementa as funcionali-

dades usando a API e que é utilizado para gerar o código fonte para uma determinada plataforma especificada. A modelagem é independente de plataforma e de versão.

Figura 5.2: *Modelagem de uma aplicação que relata os dados coletados do sensor*

Outra modelagem de aplicação típica de redes de sensores são as tarefas de roteamento que é mostrada na Figura 5.3. Um nó sensor pode receber uma mensagem, processá-la, e enviá-la.

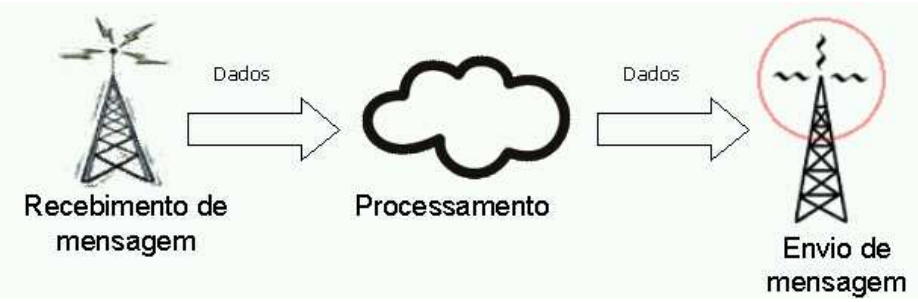


Figura 5.3: *Modelagem de uma aplicação que recebe mensagens, processa e envia mensagem*

5.4 Construção de Aplicações

O componente básico de uma aplicação é um módulo. Um módulo encapsula funcionalidades correlatas, sendo composto de campos e métodos. Ele possui apenas uma instância em uma dada aplicação. Enquanto os métodos são exportados para o acesso por outros módulos, os campos são acessíveis apenas internamente ao módulo. Um módulo de rádio, por exemplo, encapsula métodos de recebimento e envio de mensagens, mudança da freqüência do rádio e do nível de energia utilizado. A aplicação é construída a partir da interação entre os módulos utilizados.

Existem dois tipos de módulos: sistema e de usuário. Os módulos de sistema são implementados com o auxílio do *middleware*, pois acessam funções específicas do sistema operacional utilizado. Essas funções envolvem todo o acesso ao hardware do nó sensor. Esses módulos possuem código específico para cada sistema operacional suportado pelo middleware, e portanto eles não podem ser modificados pelo desenvolvedor. Já os módulos de usuário são independentes do sistema operacional utilizado, e sua funcionalidade é especificada pelo desenvolvedor. Ambos os módulos podem ser armazenados, para utilização posterior em outras aplicações.

As aplicações construídas pelo *middleware* são *dirigidas a eventos*. Isso significa que só há computação em resposta a um evento ocorrido. Para computações que requerem que ocorram constantemente, basta associar um evento de um temporizador a ela e configurar o temporizador para uma freqüência que atenda a necessidade. Para oferecer esse modelo assíncrono de programação, o *middleware* se vale do conceito de métodos sinalizadores e receptores.

5.5 Métodos sinalizadores e receptores

O conceito de métodos sinalizadores e receptores surgiu com a necessidade de especificar aplicações que respondessem a eventos. Isso significa que uma função da aplicação pode ser chamada de forma assíncrona, na ocorrência de eventos. Como será descrito a seguir, o conceito descrito nessa seção pode ser utilizado para oferecer esse paradigma, sendo de simples entendimento para o desenvolvedor. Os módulos de usuário também podem oferecer “eventos” a serem tratados por outros módulos.

Baseando-se em uma metodologia de desenvolvimento de aplicações gráficas utilizada pela ferramenta Qt Designer [98], foi criado o conceito de *métodos sinalizadores e receptores*. Um método sinalizador, na ocorrência de um evento associado ao mesmo, é capaz de disparar a execução de um ou mais métodos receptores, os quais são responsáveis por tratar aquele evento de maneira adequada. Por outro lado, um mesmo método receptor pode tratar diferentes métodos sinalizadores. Esse relacionamento é exibido na Figura 5.4.

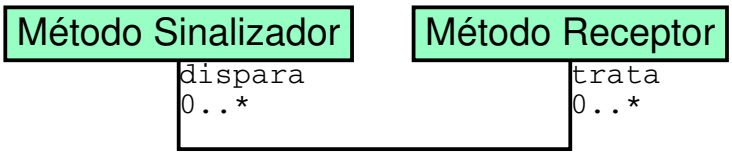


Figura 5.4: *Relacionamento entre os métodos sinalizadores e receptores.*

Cada método sinalizador de um módulo corresponde a um evento disparado pelo mesmo. Esse evento pode ser a recepção de uma mensagem via rádio, a leitura de algum sensor ou a finalização de algum método receptor. Quando um evento ocorre, há a chamada do método sinalizador correspondente. Quando o módulo é do sistema, o próprio middleware se encarrega de fazer a chamada. Se o módulo for de usuário, um método sinalizador pode ser disparado em qualquer parte do código na qual se queira alertar outros módulos do acontecimento de um evento. Esse disparo é feito através de uma simples chamada de procedimento. No *middleware*, o usuário pode indicar quais métodos de um módulo são sinalizadores. Essa informação faz parte da especificação do módulo.

Os métodos receptores, por sua vez, são aqueles encarregados de tratar o evento notificado por um método sinalizador. Qualquer função declarada no módulo que não seja sinalizadora pode ser um possível método receptor. A única diferença para uma função comum é que um método receptor é executado toda vez que um evento associado a ele é disparado.

Definidos esses métodos, é possível construir *conexões* entre os métodos sinalizadores e receptores. Essas são definidas estaticamente, através do *middleware*, em tempo de desenvolvimento. As conexões indicam quais métodos receptores serão executados quando algum método sinalizador for chamado. É possível conectar um mesmo método sinalizador a vários métodos receptores. Nesse caso, todos os métodos receptores associados são executados em ordem arbitrária. Também é permitido um método receptor esteja conectado a vários métodos sinalizadores. Dessa maneira, é possível tratar de forma homogênea eventos distintos.

Conexões só podem ser construídas se os métodos sendo conectados possuem parâmetros do mesmo tipo e o mesmo retorno. Dessa forma, um método sinalizador `void obterTemp(int leitura)` só pode ser conectado a um método receptor que retorne `void` e tenha como um parâmetro do tipo `int`, como em `void coletarDados(int dadoLido)`, ou nenhum parâmetro, como em `void terminarColetaDados()`. Obrigatoriamente, um método sinalizador deve ser um procedimento.

A conexão entre módulos através de métodos sinalizadores e receptores favorece a construção de módulos reutilizáveis. O método sinalizador não tem conhecimento, a priori, de quais métodos receptores aos quais ele está conectado. Da mesma forma, os métodos receptores não conhecem seus métodos sinalizadores. Portanto, o acoplamento entre módulos é fraco, o que permite a reutilização desses módulos em outros contextos para os quais ele não foi construído. Outro aspecto interessante é que os métodos sinalizadores podem ser utilizados para notificar condições de exceção durante a execução do módulo, retirando o tratamento de erros para fora do módulo.

Tudo isso promove a reutilização efetiva de módulos, tornando a construção de aplicações mais simples e rápida.

5.6 Geração de código

A partir da escolha dos módulos e a conexão apropriada dos mesmos, o *middleware* está apto a gerar código. Como a execução da aplicação é dirigida por eventos, cria-se uma tarefa para cada evento de módulos de sistema. Uma tarefa é uma porção de código que executa de forma síncrona, podendo ser escalonada para execução pelo sistema operacional. Todos os sistemas operacionais para RSSF's suportam esse conceito, seja através de linhas (*threads*) ou a postagem explícita de um procedimento para execução posterior.

A partir das tarefas criadas nesse primeiro passo, são criadas novas tarefas para cada método receptor que estiver conectado a um ou mais métodos sinalizadores.O *middleware*, neste momento, implementa os métodos sinalizadores escalonando tarefas para cada método receptor vinculado a esses, utilizando chamadas específicas do sistema operacional utilizado. Como as tarefas não podem receber parâmetros, esses parâmetros são criadas como variáveis globais e sua atribuição se dá através de uma seção crítica. Essa situação é exibida na Figura 5.5

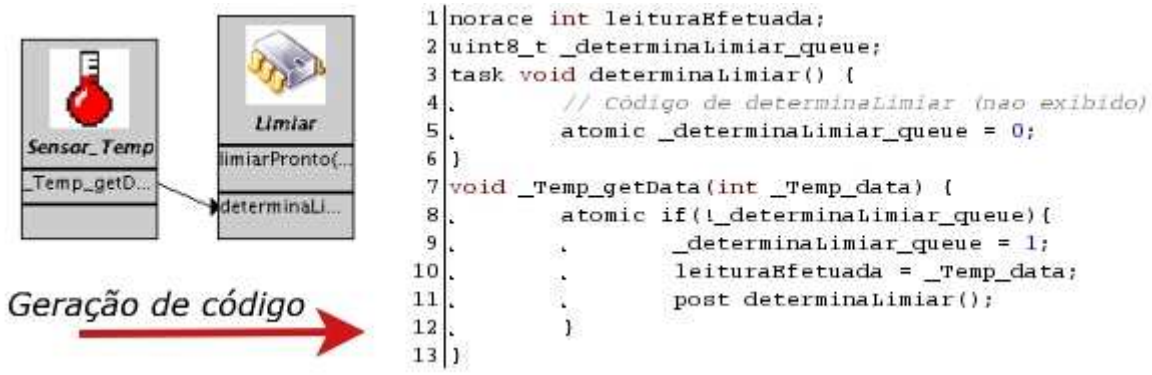


Figura 5.5: *Geração do código-fonte de um módulo de sistema, para o sistema operacional TinyOS.*

O *middleware* também se encarrega de incluir código independente de aplicação. Esse código involve a inicialização do *hardware* ou a geração de quaisquer cabeçalhos que sejam necessários.

5.7 Exemplo de uso do WISDOM

Esta seção demonstra a aplicabilidade do modelo de programação da seção 5.3, bem como a independência de plataforma do código gerado através da ferramenta WISDOM.

Os passos para se gerar uma aplicação utilizando o WISDOM são:

1. Escolher os módulos do sistema. Algumas funções, como as presentes na API do sistema, já possuem código implementado. O usuário precisa apenas escolher aquelas que vai usar. A Figura 5.6 mostra o primeiro passo, a escolha dos módulos do sistema. Observe que é mostrado uma descrição textual do módulo e também a descrição dos métodos sinalizadores deste módulo, além de um ícone.

Figura 5.6: *Passo 1: Escolha dos módulos de sistema.*

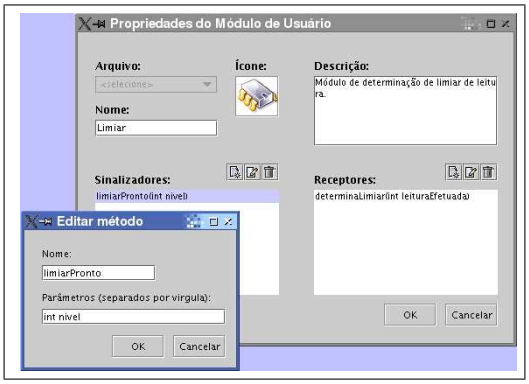


Figura 5.7: *Passo 2: Escolha dos módulos de usuário.*

2. Escolher os módulos do usuário. O usuário pode incluir no sistema funções que ele implementou, criar novos módulos ou editar os existentes no próprio WISDOM. Estas funções podem ser adicionados ao repositório de código. Na próxima vez que ele for precisar da mesma função, está já estará presente na ferramenta, não precisando implementá-las novamente. A Figura 5.7 mostra que o nome da função, bem como sua descrição, e até mesmo o código podem ser editados.

Dessa forma, o WISDOM permite desenvolver aplicações para RSSFs através de um modelo de programação que permite gerar código independente de plataforma, e que, ao mesmo tempo, é eficiente pois o código é gerado para a plataforma específica.

3. Conectar os métodos dos módulos. Após a inclusão dos módulos que serão utilizados, é preciso definir a interação entre os módulos. A Figura 5.8 mostra a ligação entre métodos receptores com métodos sinalizadores dos módulos. Esta ligação define a ordem de processamento e de chamada das tarefas.

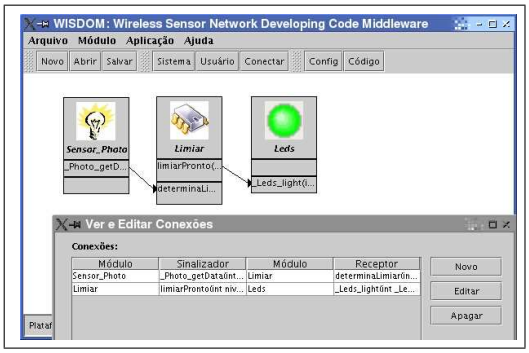
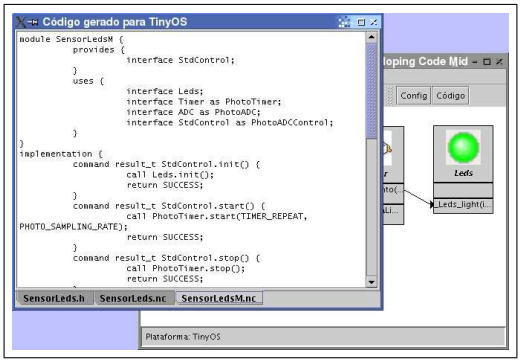
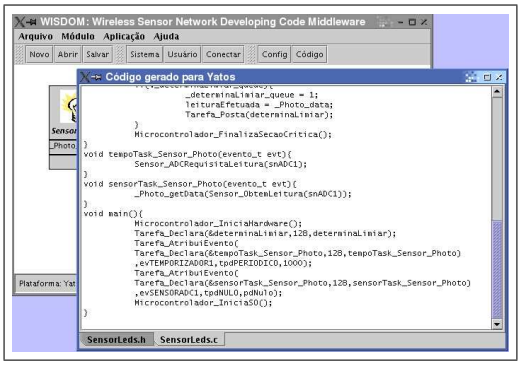


Figura 5.8: *Passo 3: Conexão entre módulos.*



(a) Código para o TinyOS



(b) Código para o YATOS

Figura 5.9: *Passo 4: Gerar o código.*

4. Gerar o código. O usuário escolhe a plataforma para a qual o código deve ser gerado, e, aciona o comando para gerar o código. A Figura 5.9 mostra o código de uma aplicação gerado para o TinyOS (Figura 5.9(a)) e para o YATOS (Figura 5.9(b)).

5. Carregar o código no nó sensor utilizando as ferramentas próprias de cada plataforma (compiladores, linkers, loaders). O código gerado pelo WISDOM está pronto para ser carregado e executado no nó sensor. A Figura 5.10 ilustra o procedimento para carregar um programa na plataforma Mica. O código fonte é compilado e o arquivo gerado se encontra no formato s-rec [63] para ser carregado no nó sensor.

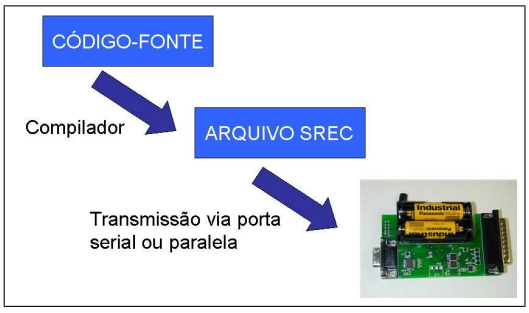


Figura 5.10: *Passo 5: Carregar o código no nó sensor.*

Capítulo 6

Resultados

> "The important thing is not to stop questioning."
>
> Albert Einstein

6.1 WISDOM

Como resultados, apresentamos dois estudos de casos de aplicações desenvolvidas com o WISDOM e que utilizam o modelo de programação elaborado, e que geram código para o YATOS e o TinyOS.

6.1.1 Leitura de dados de um sensor

Neste estudo de caso é desenvolvido uma aplicação simples, a leitura de um sensor. A aplicação é composta por três módulos, como mostra a Figura 6.1. O módulo Sensor_Temp é responsável por adquirir a temperatura do ambiente periodicamente. As leituras obtidas são transmitidas ao módulo Limiar, que sinaliza uma leitura somente quando essa ultrapassa uma constante (definida internamente). A leitura sinalizada é posteriormente exibida pelos Leds.

6.1.1.1 Código gerado para o TinyOS

O código gerado para o TinyOS é mostrado no Apêndice D, iniciando na página 105. Foram gerados dois arquivos, que representam, respectivamente, o arquivo de configuração e o arquivo de implementação em nesC. Um trecho do programa é mostrado no código 6.1 na página 63.

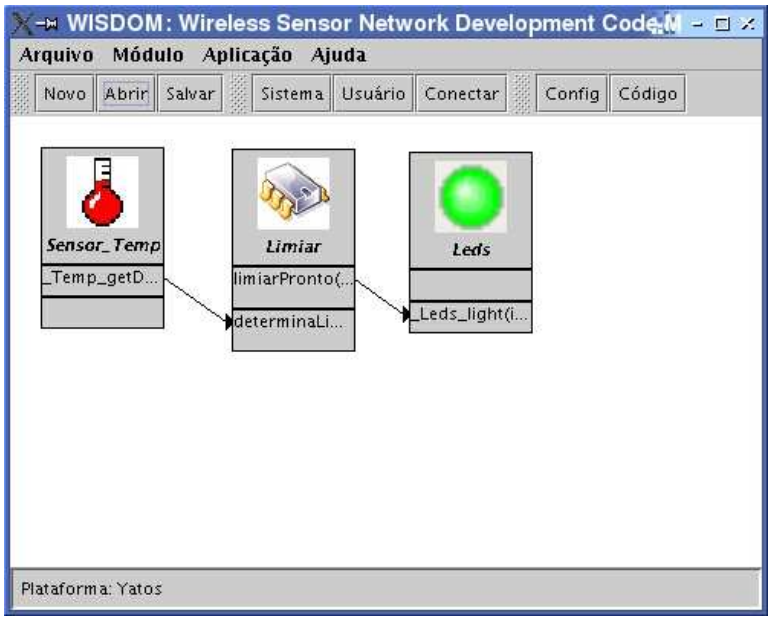


Figura 6.1: *Aplicação no WISDOM: Leitura de dados de um sensor.*

6.1.1.2 Código gerado para o YATOS

O código gerado para o YATOS é mostrado no Apêndice D, código D.1.2 na página 108. Um trecho do programa é mostrado no código 6.2.

6.1.2 Fusão de dados

Neste estudo de caso é desenvolvido uma aplicação mais complexa, que implementa uma fusão de dados simples. A aplicação é composta por cinco módulos, como mostra a Figura 6.2. O módulo Sensor_Photo é responsável por adquirir as leituras do sensor de luz a cada 1s (definido pela constante PHOTO_SAMPLING_RATE). Essas leituras são enviadas para o módulo Fusao-Media, que continuamente recebe esses valores e acumulá-os em um arranjo até que o mesmo seja completamente preenchido. Quando isso ocorre, faz-se a média dos valores desse arranjo e repassa-se esse valor através do sinalizador fusaoPronta, juntamente com o endereço do nó que deverá receber esse valor. Como FusaoMedia está conectado ao módulo Radio, os dados da fusao sao enviados via rádio para o nó destino. Os dados que alcançam o nó destino são tratados pelo módulo Base, o qual exibe os três bits mais significativos desses dados nos seus leds.

Código 6.1 Código da aplicação leitura de dados gerado para o TinyOS.

```
1  module SensorLedsM {
2    provides {
3      interface StdControl;
4    }
5    uses {
6      interface Leds;
7      interface Timer as TempTimer;
8      interface ADC as TempADC;
9      interface StdControl as TempADCControl;
10   }
11 }
12 implementation {
13   command result_t StdControl.init() { ...  }
14   command result_t StdControl.start() { ...  }
15   command result_t StdControl.stop() { ... }
16   ...
17   void limiarPronto(int nivel) { ...  }
18   task void determinaLimiar() { ...  }
19   void _Temp_getData(int _Temp_data) { ...  }
20   event result_t TempTimer.fired() { ...  }
21   async event result_t
22     TempADC.dataReady(uint16_t data) { ...  }
23 }
24
```

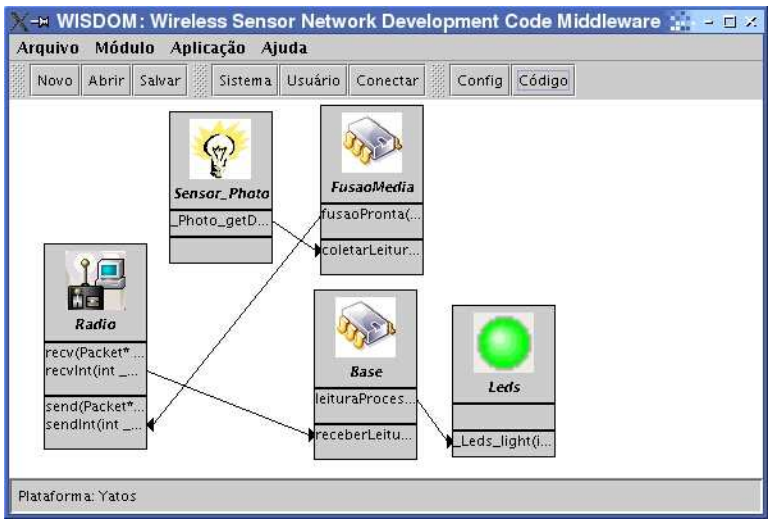


Figura 6.2: *Aplicação no WISDOM: Fusão de dados*

Código 6.2 Código da aplicação leitura de dados gerado para o YATOS .

```
#include "SensorLeds.h"
#include "SO.h"
int _Leds_value;
uint8_t __Leds_light_queue;
void _Leds_light(evento_t evt) { ...  }
void limiarPronto(int nivel) { ...  }
void determinaLimiar(evento_t evt) { ...  }
void _Temp_getData(int _Temp_data) { ...  }
void tempoTask_Sensor_Temp(evento_t evt){ ...  }
void sensorTask_Sensor_Temp(evento_t evt){ ...  }
void main(){
  Microcontrolador_IniciaHardware();
  Tarefa_Declara(&determinaLimiar,128,determinaLimiar);
  Tarefa_AtribuiEvento(
    Tarefa_Declara(&tempoTask_Sensor_Temp,128,
      tempoTask_Sensor_Temp)
  ,evTEMPORIZADOR0,tpdPERIODICO,1000);
  Tarefa_AtribuiEvento(
    Tarefa_Declara(&sensorTask_Sensor_Temp,128,
      sensorTask_Sensor_Temp)
  ,evSENSORADC0,tpdNULO,pdNulo);
  Microcontrolador_IniciaSO();
}
```

6.1.2.1 Código gerado para o TinyOS

O código gerado para o TinyOS é mostrado no Apêndice D, iniciando na página 109. Foram gerados dois arquivos, que representam, respectivamente, o arquivo de configuração e o arquivo de implementação em nesC. Um trecho do programa é mostrado no código 6.3.

Código 6.3 Código de configuração gerado para o TinyOS.

```
1  module SensorRadioM {
2    provides {
3      interface StdControl;
4    }
5    uses {
6      interface Timer as PhotoTimer;
7      interface ADC as PhotoADC;
8      interface StdControl as PhotoADCControl;
9      interface ReceiveMsg;
10     interface SendMsg;
11   }
12 }
13 implementation {
14   command result_t StdControl.init() { ... }
15   command result_t StdControl.start() { ... }
16   command result_t StdControl.stop() { ... }
17   task void _Base_recebeDados() { ... }
18   event TOS_MsgPtr
19     ReceiveMsg.receive(TOS_MsgPtr _Radio_recv_packet){ ... }
20   task void sendInt() { ... }
21   event result_t
22     SendMsg.sendDone(TOS_MsgPtr msg, bool success) { ... }
23   void fusaoPronta(int dadosFusao, int endereco) { ... }
24   task void coletarLeitura() { ... }
25   void _Photo_getData(int _Photo_data) { ... }
26   event result_t PhotoTimer.fired() { ... }
27   async event result_t PhotoADC.dataReady(uint16_t data) { ... }
28 }
29
```

6.1.2.2 Código gerado para o YATOS

O código gerado para o YATOS é mostrado no Apêndice D, código D.2.2. Um trecho do programa é mostrado no código 6.4.

Código 6.4 Código da aplicação fusão de dados gerado para o YATOS .

```
#include "SensorRadio.h"
#include "SO.h"
int _Leds_value;
uint8_t __Leds_light_queue;
void _Leds_light(evento_t evt) { ...  }
void leituraProcessada(int _Base_leituraProcessada) { ...  }
void receberLeitura(evento_t evt) { ...  }
void recvInt(int _Radio_recvInt_data,
  int _Radio_recvInt_address) { ...  }
void taskRadioRecvInt(evento_t evt){ ...  }
void sendInt(evento_t evt) { ...  }
void fusaoPronta(int _Fusao_dadosFusao,
  int _Fusao_endereco) { ...  }
void coletarLeitura(evento_t evt) { ...  }
void _Photo_getData(int _Photo_data) { ...  }
void tempoTask_Sensor_Photo(evento_t evt){ ...  }
void sensorTask_Sensor_Photo(evento_t evt){ ...  }
void main(){ ...  }
```

6.2 YATOS

YATOS foi desenvolvido para o BEAN [102] atendendo aos requisitos das redes de sensores (pouco consumo de memória, economia de energia quando ausência de tarefas no escalonador, mapeamento de eventos em tarefas, não-preemptivo, ausência de troca de contexto, dirigido a eventos).

A Tabela 6.1 mostra o consumo de memória do YATOS .

Memória de código	3.144 bytes
Memória de dados	1.984 bytes
Memória de constantes	530 bytes

Tabela 6.1: *Tabela com consumo de memória.*

Capítulo 7

Conclusão

> "People do not like to think. If one thinks, one must reach conclusions. Conclusions are not always pleasant."
>
> Helen Keller

Redes de sensores sem fio tem o potencial de serem aplicadas em várias áreas, desde áreas militares a saúde, e apresentam várias oportunidades. Elas são formadas por nós sensores, que são sistemas embutidos com capacidade de processamento, memória, interface de comunicação sem fio, além de um ou mais sensores que permitem medir dados físicos do mundo real como temperatura, pressão, umidade, etc.

Inúmeras aplicações podem ser criadas para redes de sensores sem fio. Estas devem considerar as características específicas de cada nó sensor e devem atender aos requisitos impostos por estas redes, como restrição de memória e energia. Visando facilitar o desenvolvimento de aplicações para estas redes foi possível desenvolver uma plataforma de software composta por um sistema operacional e um middleware que facilitam o trabalho do desenvolvedor. O sistema operacional facilita o desenvolvimento de aplicações, tornando o código fonte mais simples de entender e manter. O middleware é uma camada de abstração entre a aplicação e o sistema operacional que facilita o desenvolvimento de aplicações.

Este trabalho apresenta como maiores contribuições a definição de um arquitetura, a especificação dos requisitos e a implementação de um sistema operacional dedicado a redes de sensores sem fio, o YATOS . Também, a construção de um middleware, chamado WISDOM, que facilita o desenvolvimento de aplicações, gerando código independente de plataforma ou versão, desenha-

do para acompanhar os avanços tecnológicos, e ainda possui um novo modelo de programação visual. Este modelo facilita a programação, reduzindo o tempo de desenvolvimento.

O YATOS é o sistema operacional desenvolvido neste trabalho que preencheu os requisitos do projeto SensorNet e os impostos pelas Redes de Sensores de Sem Fio como pouco consumo de memória e economia de energia. O YATOS é voltado para a plataforma BEAN e aproveita os conceitos "low-power" presentes nesta plataforma. Comparado com o TinyOS, o YATOS é maior em espaço de memória, mas compensa isto tendo mais funcionalidades, sendo mais fácil de usar e não requerendo o aprendizado de uma nova linguagem de programação como nesC [21]. O YATOS foi testado em um número diferente de aplicações. Ele garante que o processador esteja no modo de baixo consumo de energia "LPM" quando não existe tarefas a serem executadas. Quando no modo "LPM", qualquer evento pode acordar o nó um instante para atender a requisição do evento.

O WISDOM é middleware desenvolvido, assim denominado por situar-se entre a aplicação e o sistema operacional. Ele atendeu os seus objetivos, facilitando o desenvolvimento de aplicações e obtendo resultados esperados. Ele mostrou ser multi-plataforma, gerando código para as plataforma YATOS /BEAN e TinyOS/Mica Motes, conforme mostrou os estudos de casos na Seção Resultados. O primeiro estudo de caso mostrou uma aplicação simples que coletava dados de um sensor, já a segunda implementou uma aplicação mais complexa que fazia uma fusão de dados antes de transmitir os dados coletados. Na ausência do middleware, o desenvolvimento de cada uma das aplicações para diferentes plataformas teria que ser feito separadamente, exigindo esforço dobrado (no caso de apenas duas plataformas) ou até mais (no caso de mais plataformas).

O WISDOM ainda permite uma programação visual, o que reduz erros como erros de digitação, baseada em módulos. De fato, uma aplicação é composta por módulos. Os módulos podem ser escritos em C, uma linguagem difundida entre programadores de sistemas embutidos. A especificação da plataforma é independente de plataforma, mas o código gerado é específico. Isso é uma importante vantagem, porque aumenta a eficiência da aplicação. O WISDOM é capaz de gerar código independente de plataforma, desde que esta implemente a API definida. O WISDOM também acompanha modificações, gerando código independente de versão, desde que as versões mantenham a API podendo mudar a implementação. Outra característica importante do WISDOM é que ele é uma ferramenta independente de plataforma também, uma vez que foi desenvolvido na linguagem de programação Java.

Como trabalho futuro, podemos criar um repositório de código no WISDOM, facilitando ainda mais o desenvolvimento de aplicações. O usuário precisará apenas de selecionar os módulos

e conexões entre eles. Alguns módulos podem ser criados uma vez e usados por várias aplicações. Entre eles podemos citar módulos de compressão de dados.

Futuramente, pode ser incluído no WISDOM e no YATOS apoio a programação dos nós sensores remotamente, ou "programação via ar", como ocorre no SensorWare, mas sem o uso de scripting. Uma mensagem identificada como código pode ser carregada na memória de programa e reiniciar o sistema para executar a partir do endereço em que o código foi carregado, como está sendo estudado no projeto Mica [44]. Essa área também irá requerer um estudo de segurança e de mecanismos de criptografia que impeçam a invasão da rede ou execução de um código que não pertence a rede.

Também há espaço nas redes de sensores sem fio para interagirem com robôs, adicionando mobilidade a estas redes. Robomote [90] é um projeto em desenvolvimento que colocou nós sensores em pequenos robôs móveis. O modelo de programação e o WISDOM podem futuramente incorporar mecanismos que permitam adicionar mobilidade aos nós sensores.

O WISDOM ainda pode adquirir algumas funcionalidades extras, como internacionalização e integração com outras ferramentas relacionadas ao desenvolvimento de aplicações para redes de sensores, como por exemplo ncc (o compilador da linguagem nesC), TOSSIM (o simulador de uma rede de sensores executando o TinyOS), TinyViz (o visualizador da simulação no TOSSIM), gcc (o compilador da linguagem de programação C).

Espera-se que o WISDOM possa ser usado por outros grupos de pesquisa e que novas plataformas além do Mica [36, 37] e do BEAN possam ser adicionadas, como por exemplo o Mantis [68, 2, 3], o Eyes [25, 66], o PushPin [14, 53], o Smartdust [107], o μAMPS [99], o PC104 [71], o GNOMES [109], o Sensor Webs do Jet Propulsion Lab da NASA [5, 24] ou o WINS(Wireless Integrated Network Sensors) [110].

Além da programação remota de nós e dependendo da aplicação, pode ser interessante incluir outras funcionalidades no YATOS , como por exemplo um sistema de arquivos. Algumas aplicações podem se beneficiar deste tipo de abstração, que permite mapear regiões de memória em arquivos, simplificando o trabalho do desenvolvedor e o código produzido, porém adicionando um overhead no sistema - consumindo mais memória.

Em conclusão, YATOS é fácil de usar e atuou bem nas aplicações testadas. Possivelmente, ele pode ser uma alternativa para os grupos de pesquisa que usam o TinyOS. O modelo de programação criado neste trabalho mostrou ser eficiente e atendeu as necessidades para programação de nós sensores. Finalmente, o WISDOM permitiu o desenvolvimento de aplicações simples , como leitura de dados dos sensores, a mais complexas, como fusão de dados, indepen-

dentemente da plataforma utilizando a API construída.

Referências Bibliográficas

[1] VxWorks 5.4. Datasheet. http://www.windriver.com/products/html/vxwks54_ds.html.

[2] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. Mantis: System support for multimodal networks of in-situ sensors. In *2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, pages 50–59, 2003.

[3] H. Abrach, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, and R. Han. Earlier version mantis: System support for multimodal networks of in-situ sensors. Technical report, Department of Computer Science, University of Colorado, April 2003. Techinal Report CU-CS-950-03.

[4] National Aeronautics and Space Administration (NASA). Sensor webs deployments. http://sensorwebs.jpl.nasa.gov/resources/deployments.shtml.

[5] National Aeronautics and Space Administration (NASA). Sensor webs, jet propulsion lab. http://sensorwebs.jpl.nasa.gov.

[6] Jon Agre and Loren Clare. An integrated architecture for cooperative sensing networks. *Computer*, 33(5):106–108, 2000.

[7] Ariel. Microware ariel technical overview. http://www.microware.com/ProductsServices/Technologies/ariel_technology%_brief.html.

[8] ASMET. The antarctic search for meteorites. http://geology.cwru.edu/~ansmet/, June 2003.

[9] Michael Barr. *Programming Embedded System in C and C++*. O'Reilly & Associates Inc., 1 edition, January 1999.

[10] M. Bhardwaj, A. Chandrakasan, and T. Garnett. Upper bounds on the lifetime of sensor networks, 2001.

[11] Edoardo S. Biagioni and Kent Bridges. The application of remote sensor technology to assist the recovery of rare and endangered species. *Special Issue on Distributed Sensor Networks for the International Journal of High Performance Computing Applications*, 16, August 2002.

[12] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Querying the physical world, October 2000.

[13] A. Boulis and M. B. Srivastava. A framework for efficient and programmable sensor networks. In *Proceedings of OPENARCH 2002*, June 2002.

[14] M. Broxton, J. Lifton, D. Seetharam, and J. Paradiso. Pushpin computing system overview: a platform for distributed, embedded, ubiquitous sensor networks. In *Pervasive Computing 2002*, August 2002.

[15] Lewis Girod Nirupama Bulusu and Deborah Estrin. Scalable coordination for wireless sensor networks: Self-configuring localization systems, July 2001.

[16] Microsoft Windows CE. `http://www.microsoft.com/windowsce/embedded/`.

[17] A. Cerpa and D. Estrin. Ascent: Adaptive self-configuring sensor networks topologies. Technical report, UCLA Computer Science Department, May 2001. Techinal Report UCLA/CSDTR-01-0009.

[18] Alberto Cerpa, Jeremy Elson, Michael Hamilton, Jerry Zhao, Deborah Estrin, and Lewis Girod. Habitat monitoring: application driver for wireless communications technology. In *Workshop on Data communication in Latin America and the Caribbean*, pages 20–41, Costa Rica, April 2001. ACM Press.

[19] Chipcon. Smartrf cc1000 preliminary datasheet (rev. 2.1), `http://www.chipcon.com/files/CC1000\_Data\_Sheet\_2\_1.pdf`, 2002.

[20] Seong-Hwan Cho and A. Chandrakasan. Energy efficient protocols for low duty cycle wireless microsensor networks. *Procedings of the 33rd Annual Hawaii International Conference on System Sciences*, 2, 2000.

[21] D. Culler, D. Gay, P. Levis, R. von Behren, M. Welsh, and E. Brewer. The nesc language: A holistic approach to networked embedded systems. In *Conference on Programming Language Design and Implementation of ACM SIGPLAN 2003*, 2003.

[22] Ana Luiza de Almeida Pereira Zuquim, Antonio Alfredo F. Loureiro, Claudionor N. Coelho. Jr., Marcos Augusto M. Vieira, Luiz Filipe Menezes Vieira, Alex Borges Vieira, Antônio Otávio Fernandes, Diógenes Cecilio da Silva Jr., José Monteiro da Mata, José Augusto Nacif, and Hervaldo Sampaio Carvalho. Efficient power management in real-time embedded systems. In *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, Lisboa, Portugal, September 2003.

[23] K.A. Delin, R.P. Harvey, N.A. Chabot, S.P. Jackson, Mike Adams, D.W. Johnson, and J.T. Britton. Sensor web in antarctica: Developing an intelligent, autonomous platform for locating biological flourishes in cryogenic environments. In *34th Lunar and Planetary Science Conference*, Houston, Texas, March 2003.

[24] Kevin A. Delin, Shannon P. Jackson, Scott C. Burleigh, David W. Johnson, Richard R. Woodrow, and Joel T. Britton. The jpl sensor webs project: Fielded technology. In *Space Mission Challenges for Information Technology*, Pasadena, Califórnia, July 2003.

[25] S. Dulman and P. Havinga. Operating system fundamental for the eyes distributed sensor network. In *Progress 2002, Utrecht - Netherlands*, October 2002.

[26] Deborah Estrin. Embedded networked sensing research: Emerging systems challenges. In *Workshop on Disributed Communications and Signal Processing*, December 2002.

[27] Deborah Estrin, Ramesh Govindan, and John Heidemann. Embedding the internet: introduction. *Communications of the ACM*, 43(5):38–41, 2000.

[28] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next century challenges: scalable coordination in sensor networks. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 263–270. ACM Press, 1999.

[29] Food and Agriculture Organization Of The United Nations. Forest fire management, special: Remote sensing for decision-makers. `http://www.fao.org/sd/EIdirect/EIre0074.htm`, May 1999.

[30] Jean-Luc Gaudiot and Lubomir Bic. *Advanced Topics in Data-Flow Computing*. Prentice-Hall, 1991.

[31] Kurt Geihs. Middleware challenges ahead. *IEEE Computer*, 34(6):24–31, June 2001.

[32] B. Halweil. Study finds modern farming is costly. World Watch 14 (1), 2001.

[33] Wendi Rabiner Heinzelman, Joanna Kulik, and Hari Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 174–185. ACM Press, 1999.

[34] Dan Hildebrand. An architectural overview of qnx. `http://www.qnx.com/literature/whitepapers/archoverview.html`.

[35] Jason Hill. A software architecture supporting networked sensors. Master's thesis, University of California, Berkeley, December 2000.

[36] Jason Hill and David Culler. A wireless embedded sensor architecture for system-level optimization. Technical report, Computer Science Department, University of California at Berkeley, 2002. Techinal Report.

[37] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *ACM SIGMOD, Architectural Support for Programming Languages and Operating Systems*, pages 93–104, San Diego, CA, June 2000.

[38] I.F.Akyildiz, W.Su, Y.Sankarasubramaniam, and E. Cayirci. Wireless sensor networks: A survey. *IEEE Communications Magazine*, 40(8):102–14, August 2002.

[39] Texas Instruments. Msp430 data sheet, `http://wwws.ti.com/sc/psheets/slas272c/slas272c.pdf`, 2001.

[40] Texas Instruments. Mixed signal microcontroller (rev. e), `http://www-s.ti.com/sc/ds/msp430f149.pdf`, 2003.

[41] Texas Instruments. Msp430x1xx family user's guide, `http://www-s.ti.com/sc/psheets/slau049d/slau049d.pdf`, 2004.

[42] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 56–67. ACM Press, 2000.

[43] C. Jaikaeo, C. Srisathapornphat, and C. Shen. Diagnosis of sensor networks. IEEE International Conference on Communications ICC'01, June 2001.

[44] Jaein Jeong. Projects: Node-level representation and system support for network programming. `http://www.cs.berkeley.edu/~jaein/cs294_1/`, October 2003.

[45] Nigel Jones. Beginner's corner. Embedded Systems Programming, July 2001.

[46] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: mobile networking for s̈mart dust.̈ In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 271–278. ACM Press, 1999.

[47] Barry Kauler. Creem, concurrent realtime embedded executive for microcontrollers. `http://members.dodo.net.au/~void/old/creem.htm`.

[48] Timothy H. Keitt, Dean L. Urban, and Bruce T. Milne. Detecting critical scales in fragmented landscapes. Conservation Ecology 1(1)(1997) 4. Disponível em `http://www.consecol.org/vol1/iss1/art4`.

[49] Richard Kelsey, William Clinger, and Jonathan Rees (eds.). Revised5 report on the algorithmic language scheme. *Higher-Order and Symbolic Computation*, 11(1), August 1998.

[50] Sukun Kim. Structural health monitoring of the golden gate bridge. `http://www.cs.berkeley.edu/~binetude/ggb/`, January 2004.

[51] H. Kinawi, M.M. Reda Taha, and N. El-Sheimy. Gpsr: greedy perimeter stateless routing for wireless networks. In *27th Annual IEEE Conference on Local Computer Networks (LCN'02)*, 2002.

[52] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *International Conference on Architectural Support for Programming Languages and Operating Systems, San Jose, CA, USA*, October 2002. To appear.

[53] Joshua Lifton, Deva Seetharam, Margo Seltzer, and Joseph Paradiso. Bertha: The os for pushpin computers. Technical report, MIT Media Lab, May 2002.

[54] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[55] I. Locher, S. Park, M.B. Srivastava, A. Chen, R.R. Muntz, and S. Yuen. Design of a wearable sensor badge for smart kindergarten. In *Proceedings of the 6th International Symposium on Wearable Computers*, 2002.

[56] Antonio Alfredo F. Loureiro, Jose Marcos S. Nogueira, Linnyer Beatriz Ruiz, and Raquel Aparecida de Freitas Mini. Redes de sensores sem fio. *Curso da XXI Jornada de Atualização em Informática, XXII Congresso da SBC*, July 2002.

[57] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. *First ACM Workshop on Wireless Sensor Networks and Applications.*, September 2002.

[58] Alan Mainwaring, Robert Szewczyk, John Anderson, and Joseph Polastre. Habitat monitoring on great duck island. `http://www.greatduckisland.net`.

[59] ST Microeletronics. Low voltage, serial flash memory. http://www.st.com/stonline/books/pdf/docs/7737.pdf.

[60] microLinux. Survey about micro linuxes. `http://www.tldp.org/HOWTO/LaptopHOWTO18.html`.

[61] Dejan Milojicic. Middleware rules, today and tomorrow. *IEEE Concurrency*, April 1999.

[62] Raquel A. F. Mini, Badri Nath, and Antonio A. F. Loureiro. A probabilist approach to predict the energy consumption in wireless sensor networks. In *IV Workshop de Comunicação sem Fio e Computação Móvel*, São Paulo-SP, Brasil, October 2002.

[63] Motorola. S-record file format. `http://www.cs.net/lucid/moto.htm`.

[64] Job Mulder. Peeros preemptive eyes real time operating system. Technical report, University of Twente, April 2003.

[65] Wired News. Making wines finer with wireless. `http://www.wired.com/news/wireless/0,1382,58312,00.html`, April 2003.

[66] EYES Energy Efficient Sensor Nodes. Eyes website. `http://eyes.eu.org/`.

[67] N. Noury, T. Herve, V. Rialle, G. Virone, E. Mercier, G. Morey, A. Moro, and T. Porcheron. Monitoring behavior in home using a smart fall sensor. IEEE-EMBS Special Topic Conference on Microtechnologies in Medicine and Biology, October 2000.

[68] University of Colorado at Boulder. Mantis: Multimodal networks of in-situ sensors. `http://mantis.cs.colorado.edu`.

[69] John K. Ousterhout. Scripting: Higher-level programming for the 21st century. *Computer*, 31(3):23–30, 1998.

[70] PalmOS. Software 3.5 overview. `http://www.palm.com/devzone/docs/palmos35.html`.

[71] PC104. `http://www.isi.edu/scadds/pc104testbed/guideline.html`.

[72] E.M. Petriu, N.D. Georganas, D.C. Petriu, D. Makrakis, and V.Z. Groza. Sensor-based information appliances. IEEE Instrumentation and Measurement Magazine, December 2000.

[73] Joseph Polastre. Design and implementation of wireless sensor networks for habitat monitoring. Master's thesis, University of California at Berkeley, 2003.

[74] Joseph Polastre, Robert Szewczyk, Alan Mainwaring, David Culler, and John Anderson. *Wireless Sensor Networks*, chapter Analysis of Wireless Sensor Networks for Habitat Monitoring. Kluwer Academic Publishers, 2004.

[75] pOSEK. A super-small, scalable real-time operating system for high-volume, deeply embedded applications. `http://www.isi.com/products/posek/index.htm`.

[76] G. J. Pottie and W. J. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43(5):51–58, May 2000.

[77] Nissanka B. Priyantha, Anit Chakraborty, and Hari Balakrishnan. The cricket location-support system. *Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 32–43, August 2000.

[78] Pumpkininc. Salvo, the rtos that runs in tiny places. `http://www.pumpkininc.com/`.

[79] J. Rabaey, J. Ammer, J.L. da Silva Jr., and D. Patel. Pico-radio: ad-hoc wireless networking of ubiquitous low-energy sensor/monitor nodes. *Proceedings of the IEEE Computer Society Annual Workshop on VLSI (WVLSI'00)*, pages 9–12, April 2000.

[80] Jan M. Rabaey, M. Josie Ammer, Julio L. da Silva, Danny Patel, and Shad Roundy. Picoradio supports ad hoc ultra-low power wireless networking. *Computer*, 33(7):42–48, 2000.

[81] Kay Römer, Oliver Kasten, and Friedemann Mattern. Middleware challenges for wireless sensor networks. *ACM SIGMOBILE Mobile Computing and Communications Review*, 6(4):59–61, October 2002.

[82] Linnyer Beatrys Ruiz, Luiz Henrique Correia, Luiz Filipe Menezes Vieira, Daniel F. Macedo, Eduardo Nakamura, Carlos Mauricio Figueiredo, Marcos Augusto Menezes Vieira, Eduardo Habib Mechelane, Daniel Camara, Antonio Alfredo Ferreira Loureiro, José Marcos Nogueira, and Diógenes Cecilio da Silva Jr. Arquitetura para redes de sensores sem fio. *Minicurso, XXII Congresso da SBC*, May 2004.

[83] Linnyer Beatrys Ruiz, José Marcos Nogueira, and Antonio A. F. Loureiro. Manna: A management architecture for wireless sensor networks. In *IEEE Communication Magazine*, volume 41, February 2003.

[84] Anders Rytter. *Vibration based inspection of civil engineering structures*. PhD thesis, Dept. of building technology and structural engineering, Aalborg University, April 1993.

[85] Herbert Schildt. *C Completo e Total*. MAKRON Books , McGraw-Hill, 1990.

[86] Loren Schwiebert, Sandeep K. S. Gupta, and Jennifer Weinmann. Research challenge in wireless networks of biomedical sensors. In *Mobile Computing and Networking*, pages 151–165, 2001.

[87] Devasenapathi P. Seetharamakrishnan. c@t: A language for programming massively distributed embedded system. Master's thesis, Massachusetts Institute of Technology, September 2002.

[88] SensorNet. Sensornet website. `http://www.sensornet.dcc.ufmg.br/index.html`.

[89] Eugene Shih, Seong-Hwan Cho, Nathan Ickes, Rex Min, Amit Sinha, Alice Wang, and Anantha Chandrakasan. Physical layer driven protocol and algorithm design for energy-efficient wireless sensor networks. *Proceedings of the 7th annual international conference on Mobile computing and networking*, pages 272–287, July 2001.

[90] Gabriel T. Sibley, Mohammad H. Rahimi, and Gaurav S. Sukhatme. Robomote: A tiny mobile robot platform for large-scale sensor networks. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA2002)*, volume 2, pages 1143–1148, Washington, DC, May 2002.

[91] S. Slijepcevic and M. Potkonjak. Power efficient organization of wireless sensor networks. IEEE International Conference on Communications ICC'01, June 2001.

[92] Roy Sutton Suet-Fei Li and Jan Rabaey. Low power operating system for heterogeneous wireless communication systems. In *Workshop on Compilers and Operating Systems for Low Power 2001*, September 2001.

[93] Digital Sun. S.sense wireless sensors. `http://www.digitalsun.com/html/products.html`.

[94] Berkeley WEBS: Wireless Embedded Systems. Tinyos website. `http://www.tinyos.net`.

[95] Berkeley WEBS: Wireless Embedded Systems. Webs website. `http://webs.cs.berkeley.edu/index.html`.

[96] S. Tilak, N. Abu-Ghazaleh, and W. Heinzelman. A taxonomy of wireless microsensor network models. *ACM Mobile Computing and Communications Review*, 2002.

[97] New York Times. With Glacier Park in Its Path, Fire Spreads to 40,000 Acres, New York Times, Vol. 150, Issue 51864, p. 24, 0p, 1 map, 4c, 9/2/2001.

[98] Trolltech. Signals and slots. *Qt Reference Documentation*, 2003.

[99] uAMPS. u-adaptive multi-domain power aware sensors. `http://www-mtl.mit.edu/research/icsystems/uamps/`.

[100] L.F.W. van Hoesel, S.O. Dulman, P.J.M. Havinga, and H.J. Kip. Design of a low-power testbed for wireless sensor networks and verification. Technical report, Centre for Telematics and Information Technology , University of Twente, September 2003. Techinal Report.

[101] Luiz Filipe Menezes Vieira, Marcos Augusto Menezes Vieira, Linnyer Beatrys Ruiz, Antonio Alfredo Ferreira Loureiro, Antônio Otávio Fernandes, Diógenes Cecilio da Silva Jr., and José Marcos S. Nogueira. Efficient incremental sensor network deployment algorithm. In *XXII Simpósio Brasileiro de Redes de Computadores (SBRC)*, Gramado-RS, Brasil, May 2004.

[102] Marcos Augusto Menezes Vieira. Embedded system for wireless sensor network. Master's thesis, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte-MG, Brasil, April 2004.

[103] Marcos Augusto Menezes Vieira, Diógenes Cecilio da Silva Jr., and Claudionor Nunes Coelho. Survey on wireless sensor network devices. *IEEE International Conference on Emerging Technologies and Factory Automation (ETFA) 2003*, September 2003.

[104] Marcos Augusto Menezes Vieira, Luiz Filipe Menezes Vieira, Antonio Alfredo Ferreira Loureiro, Antônio Otávio Fernandes Linnyer Beatrys Ruiz and, Diógenes Cecilio da Silva Jr., and José Marcos S. Nogueira. Como obter o mapa de energia em rede de sensores sem fio? uma abordagem tolerante a falhas. In *Workshop de Comunicação Sem Fio*, São Lourenço-MG, October 2003.

[105] Marcos Augusto Menezes Vieira, Luiz Filipe Menezes Vieira, Antonio Alfredo Ferreira Loureiro, Antônio Otávio Fernandes Linnyer Beatrys Ruiz and, Diógenes Cecilio da Silva Jr., and José Marcos S. Nogueira. Scheduling nodes in wireless sensor network: A voronoi approach. In *The 28th Annual IEEE Conference on Local Computer Networks (LCN)*, Bonn/Königswinter, Alemanha, September 2003.

[106] Peter Volgyesi, Miklos Maroti, Sebestyen Dora, Esteban Osses, Akos Ledeczi, and Tamas Paka. Embedded software composition and verification. Technical report, Institute for Software Integrated Systems, Vanderbilt University, February 2004.

[107] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 34(1):44–51, 2001.

[108] Motes website. Motes. `http://kingkong.me.berkeley.edu/~nota/RunningMan/Mote.htm`.

[109] Erik Welsh, Walt Fish, and J. Patrick Frantz. Gnomes: A testbed for low power heterogeneous wireless sensor networks.

[110] WINS. Wireless integrated network sensors. `http://www.janet.ucla.edu/WINS/`, 2002.

[111] Ning Xu. Implementation of data compression and fft on tinyos, 2004. Disponível em `http://enl.usc.edu/~ningxu/papers/lzfft.pdf`.

[112] Ning Xu. A survey of sensor network applications, 2004. Disponível em `http://enl.usc.edu/~ningxu/papers/survey.pdf`.

[113] Jerry Zhao, Ramesh Govindan, and Deborah Estrin. Residual energy scans for monitoring wireless sensor networks. In *IEEE Wireless Communications and Networking Conference (WCNC 02)*, Orlando, FL, March 2002.

Apêndice A

Telas do WISDOM

Neste capítulo são mostradas algumas telas da ferramenta WISDOM.

A Figura A.1 mostra a tela inicial do WISDOM, que permite abrir uma aplicação já construída ou iniciar uma nova.

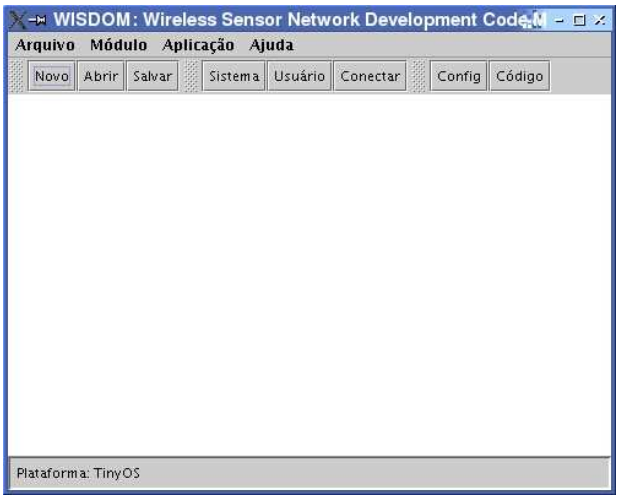


Figura A.1: *Tela Inicial do WISDOM.*

Uma aplicação é construída desenvolvendo módulos, que podem ser módulos de sistema (conforme Figura A.3) ou de usuário (conforme Figura A.2).

Os módulos escolhidos são conectados utilizando a tela mostrada na Figura A.4.

Antes de gerar o código, é necessário escolher a plataforma de desenvolvimento, e isto é feito através da tela mostrada na Figura A.5.

É possível editar as propriedades das conexões, conforme mostra a Figura A.6.

Escolhendo um módulo de sistema, é possível visualizar o código. Escolhendo um módulo de usuário também é possível editá-lo conforme mostra a Figura A.9.

Figura A.2: *Criação de módulos do usuário no WISDOM.*

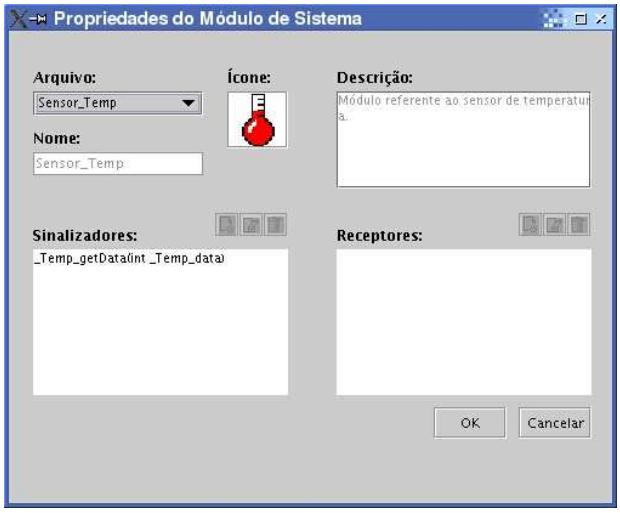


Figura A.3: *Criação de módulos do usuário no WISDOM.*

O código gerado também é mostrado no WISDOM, como mostra a Figura A.8.

Além de abrir e salvar, é possível imprimir e visualizar a impressão. Esta é mostrada na Figura A.7.

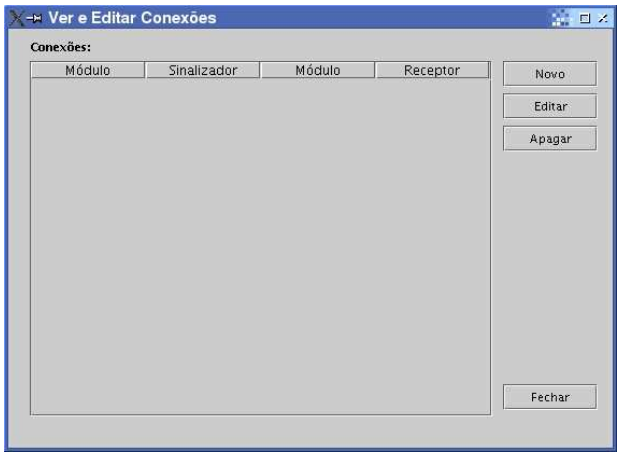


Figura A.4: *Estabelecimento das conexões entre módulos.*

Figura A.5: *Configuração da Aplicação.*

Figura A.6: *Propriedades Conexão.*

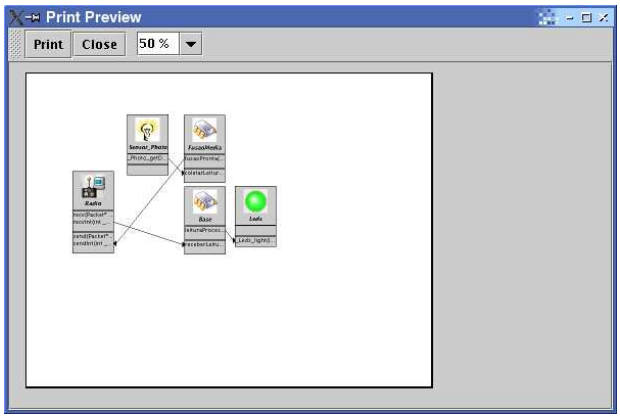


Figura A.7: *Visualizar Impressao.*

Figura A.8: *Tela com código gerado no WISDOM.*

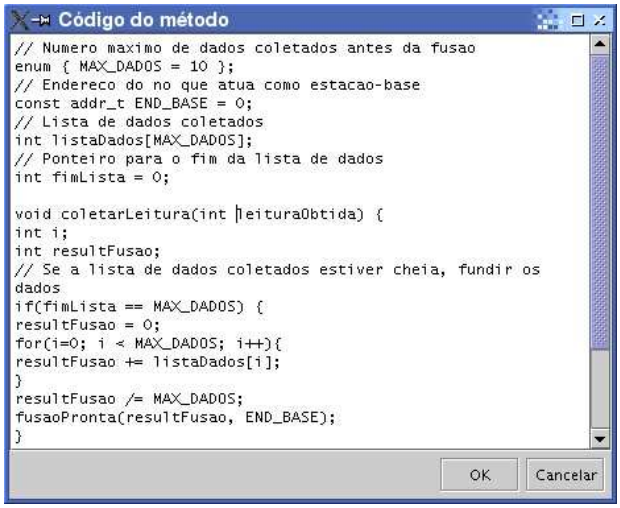


Figura A.9: *Código do Módulo.*

Apêndice B

API para o TinyOS

A API desenvolvida para que o código gerado pelo WISDOM funciona-se com o TinyOS é descrita a seguir.

B.1 API Radio

- Objetivo: prover acesso e controle do dispositivo de rádio do nó.
- Serviços:

  command result_t *init*(): prepara o rádio e seus sub-componentes para operação.

  Parâmetros: não aplicável.

  Estado:

  – SUCCESS: preparação bem-sucedida.

  – FAIL: falha na preparação.

  Particularidades: não aplicável.

  command result_t *start*(): inicia o funcionamento do rádio e de seus sub-componentes.

  Parâmetros: não aplicável.

  Estado:

  – SUCCESS: início de funcionamento bem-sucedido.

  – FAIL: falha no início de funcionamento.

  Particularidades: não aplicável.

  command result_t *stop*(): pára o funcionamento do rádio e de seus sub-componentes.

Parâmetros: não aplicável.

Estado:

- SUCCESS: parada bem-sucedida.
- FAIL: falha na parada.

Particularidades: sub-componentes aguardando eventos de temporização não são desligados.

command result_t *send*(uint16_t address, uint8_t length, TOS_MsgPtr msg): requisita o envio de mensagem para outro nó através do rádio.

Parâmetros:

- address: endereço do nó de destino.
- length: comprimento, em bytes, da mensagem.
- msg: ponteiro para mensagem a ser enviada.

Estado:

- SUCCESS: requisição aceita.
- FAIL: requisição recusada.

Particularidades: para o caso de requisição aceita, após o envio da mensagem será sinalizado o evento *sendDone* (descrito abaixo).

event result_t *sendDone*(TOS_MsgPtr msg, result_t success): evento que é sinalizado ao término do envio de mensagem para outro nó através do rádio.

Parâmetros:

- msg: ponteiro para mensagem enviada.
- success: informa se o envio foi bem-sucedido (i.e. SUCCESS) ou mal-sucedido (i.e. FAIL).

Particularidades: não aplicável.

event TOS_MsgPtr *receive*(TOS_MsgPtr m): evento sinalizado após a recepção de mensagem via rádio.

Parâmetros:

- m: ponteiro para mensagem recebida.

Particularidades: não aplicável.

command result_t *SetListeningMode*(uint8_t power): altera o modo de escuta do rádio.

Parâmetros:

- power: modo de consumo de energia. Pode assumir os seguintes valores: 0 = sempre ligado ... 3 = menor período de escuta.

Estado:

- SUCCESS: alteração bem-sucedida.
- FAIL: falha na alteração.

Particularidades: não aplicável.

command uint8_t *GetListeningMode*(): obtém o modo de escuta do rádio.

Parâmetros: não aplicável.

Estado:

- 0 = sempre ligado ... 3 = menor período de escuta.

Particularidades: não aplicável.

command result_t *SetTransmitMode*(uint8_t power): altera o modo de transmissão do rádio.

Parâmetros:

- power: modo de transmissão. Pode assumir os seguintes valores: 0 = sempre ligado (throughput efetivo: 12,364kbps) ... 7 = menor período de transmissão (throughput efetivo: 0,258kbps).

Estado:

- SUCCESS: alteração bem-sucedida.
- FAIL: falha na alteração.

Particularidades: o nó de transmissão do nó emissor deve ser o mesmo do nó receptor para que este receba a mensagem corretamente.

command uint8_t *GetTransmitMode*(): obtém o modo de transmissão do rádio.

Parâmetros: não aplicável.

Estado:

- 0 = sempre ligado (throughput efetivo: 12,364kbps) ... 7 = menor período de transmissão (throughput efetivo: 0,258kbps).

Particularidades: não aplicável.

command result_t *set*(uint8_t setting): altera a resistência do potenciômetro, permitindo com isso a regulagem do alcance do rádio.

Parâmetros:

- setting: nível de resistência do potenciômetro. Pode assumir os seguintes valores: 0 = grande potência, baixa resistência do potenciômetro ... 99 = baixa potência, alta resistência do potenciômetro.

Estado:

- SUCCESS: alteração bem-sucedida.
- FAIL: falha na alteração.

Particularidades: o alcance efetivo dependerá da antena empregada: embutida = de 3 cm a 4,5 m aproximadamente; externa = 33 cm a 330 m aproximadamente.

command result_t *increase*(): incrementa a potência de transmissão do potenciômetro de 1 unidade.

Parâmetros: não aplicável.

Estado:

- SUCCESS: incremento bem-sucedida.
- FAIL: falha no incremento.

Particularidades: não aplicável.

command result_t *decrease*(): decrementa a potência de transmissão do potenciômetro de 1 unidade.

Parâmetros: não aplicável.

Estado:

- SUCCESS: decremento bem-sucedida.
- FAIL: falha no decremento.

Particularidades: não aplicável.

command uint8_t *get*(): obtém a potência de transmissão do potenciômetro.

Parâmetros: não aplicável.

Estado:

- 0 = grande potência, baixa resistência do potenciômetro . . . 99 = baixa potência, alta resistência do potenciômetro.

Particularidades: o alcance efetivo dependerá da antena empregada: embutida = de 3 cm a 4,5 m aproximadamente; externa = 33 cm a 330 m aproximadamente.

B.2 API Sensor

Nessa API, foi implementado o mesmo conjunto de comandos para sensores de: temperatura, luz, som, acelerômetro e magnetômetro. A escolha de qual comando será executado será determinada por qual das interfaces fornecidas será usada.

- Objetivo: prover acesso e controle dos dispositivos sensores do nó.
- Serviços:

command result_t *init*(): prepara o componente sensor e seus sub-componentes para operação. Idem comando homônimo da API Radio, descrito em B.1

command result_t *start*(): inicia o funcionamento do componente sensor e de seus sub-componentes. Idem comando homônimo da API Radio, descrito em B.1

command result_t *stop*(): pára o funcionamento do componente sensor e de seus sub-componentes. Idem comando homônimo da API Radio, descrito em B.1

async command result_t *getData*(): requisita uma leitura do sensor, usando o conversor ADC.

Parâmetros: não aplicável.

Estado:

- SUCCESS: ADC livre e disponível para aceitar a requisição de serviço.
- FAIL: ADC indisponível.

Particularidades: quando a conversão estiver concluída, o evento *dataReady* (descrito abaixo) será sinalizado, passando o valor da leitura como parâmetro.

async command result_t *getContinuousData*(): requisita uma série de leituras do sensor, usando o ADC. Para os demais itens, idem comando *getData*, acima.

async event result_t *dataReady*(uint16_t data): evento que é sinalizado ao término da leitura do sensor e sua conversão usando o ADC.

Parâmetros:

– data: valor obtido pelo ADC.

Particularidades: não aplicável.

B.3 API Memory

- Objetivo: prover acesso e controle da memória Flash.
- Serviços:

command result_t *init*(): prepara o componente de memória e seus sub-componentes para operação. Idem comando homônimo da API Radio, descrito em B.1

command result_t *start*(): inicia o funcionamento do componente de memória e de seus sub-componentes. Idem comando homônimo da API Radio, descrito em B.1

command result_t *stop*(): pára o funcionamento do componente de memória e de seus sub-componentes. Idem comando homônimo da API Radio, descrito em B.1

command result_t *request*(uint32_t numBytesReq): requisita a alocação de uma seção de memória.

Parâmetros:

– numBytesReq: tamanho em bytes da seção de memória requisitada.

Estado:

– SUCCESS: requisição aceita.

– FAIL: argumentos inválidos ou não há memória disponível.

Particularidades: essa requisição deve ser feita em tempo de inicialização do componente. Para o caso de requisição aceita, após a alocação da seção será sinalizado o evento *requestProcessed* (descrito abaixo).

command result_t *requestAddr*(uint32_t byteAddr, uint32_t numBytesReq): requisita a alocação de uma seção específica de memória.

Parâmetros:

- byteAddr: deslocamento inicial em bytes, que deve ser na fronteira de uma página (a constante TOS_BYTEEEPROM_PAGESIZE informa o tamanho de uma página).
- numBytesReq: tamanho em bytes da seção de memória requisitada.

Para os demais itens, idem comando *request*, acima.

event result_t *requestProcessed*(result_t success): evento que é sinalizado ao término da alocação da seção de memória.

Parâmetros:

- success: informa se a alocação foi bem-sucedida (i.e. SUCCESS) ou mal-sucedida (i.e. FAIL).

Particularidades: não aplicável.

command result_t *read*(int32_t offset, uint8_t* buffer, uint32_t numBytesRead): requisita uma leitura da memória.

Parâmetros:

- offset: deslocamento a partir do qual será feita a leitura.
- buffer: onde colocar os dados lidos.
- numBytesRead: número de bytes a serem lidos.

Estado:

- SUCCESS: requisição aceita.
- FAIL: requisição recusada.

Particularidades: para o caso de requisição aceita, após a leitura dos dados será sinalizado o evento *readDone* (descrito abaixo).

event result_t *readDone*(uint8_t* buffer, uint32_t numBytesRead, result_t success): evento que é sinalizado ao término da leitura da memória.

Parâmetros:

- buffer: onde os dados lidos foram colocados.

- numBytesRead: número de bytes lidos.
- success: informa se a leitura foi bem-sucedida (i.e. SUCCESS) ou mal-sucedida (i.e. FAIL).

Particularidades: não aplicável.

command result_t *write*(uint32_t offset, uint8_t *data, uint32_t numBytesWrite): requisita uma escrita na memória.

Parâmetros:

- offset: deslocamento a partir do qual será feita a escrita.
- data: dados a serem escritos.
- numBytesWrite: número de bytes a serem escritos.

Estado:

- SUCCESS: requisição aceita.
- FAIL: requisição recusada.

Particularidades: para o caso de requisição aceita, após a escrita dos dados será sinalizado o evento *writeDone* (descrito abaixo).

event result_t *writeDone*(uint8_t *data, uint32_t numBytesWrite, result_t success): evento que é sinalizado ao término da escrita de memória.

Parâmetros:

- data: endereço dos dados escritos.
- numBytesWrite: número de bytes escritos.
- success: informa se a escrita foi bem-sucedida (i.e. SUCCESS) ou mal-sucedida (i.e. FAIL).

Particularidades: não aplicável.

command result_t *erase*(): requisita o apagamento da região de memória alocada.

Parâmetros: não aplicável.

Estado:

- SUCCESS: requisição aceita.
- FAIL: requisição recusada.

Particularidades: para o caso de requisição aceita, após o apagamento da região será sinalizado o evento *eraseDone* (descrito abaixo). Chamadas desse comando não devem ser entremeadas com chamadas ao comando *write* (descrito acima), a menos que seja executado o comando *sync* primeiro (descrito abaixo).

event result_t *eraseDone*(result_t success): evento que é sinalizado ao término do apagamento da região de memória.

Parâmetros:

- success: informa se o apagamento foi bem-sucedido (i.e. SUCCESS) ou mal-sucedido (i.e. FAIL).

Particularidades: não aplicável.

command result_t *append*(uint8_t* data, uint32_t numBytes): requisita a anexação (append) de bytes a uma região de memória.

Parâmetros:

- data: dados a serem anexados.
- numBytes: número de bytes a serem anexados.

Estado:

- SUCCESS: requisição aceita.
- FAIL: requisição recusada.

Particularidades: para o caso de requisição aceita, após a anexação dos dados será sinalizado o evento *appendDone* (descrito abaixo). Chamadas desse comando não devem ser entremeadas com chamadas ao comando *write* (descrito acima), a menos que seja executado o comando *sync* primeiro (descrito abaixo).

event result_t *appendDone*(uint8_t* data, uint32_t numBytes, result_t success): evento que é sinalizado ao término da anexação de dados à região de memória.

Parâmetros:

- data: endereço dos dados escritos.
- numBytesWrite: número de bytes escritos.
- success: informa se a anexação foi bem-sucedida (i.e. SUCCESS) ou mal-sucedida (i.e. FAIL).

Particularidades: não aplicável.

command uint32_t *currentOffset()*: informa o endereço de memória onde os próximos dados serão anexados.

Parâmetros: não aplicável.

Estado:

- (uint32_t)-1: anexações não são permitidas.
- demais valores: endereço de memória onde os proximos dados serão anexados.

Particularidades: anexações não são permitidas após a execução do comando *sync* (descrito abaixo) ou antes do comando *erase* (descrito acima).

command result_t *sync*(): assegura que todos os dados anexados foram gravados na região de memória.

Parâmetros: não aplicável.

Estado:

- SUCCESS: requisição aceita.
- FAIL: requisição recusada.

Particularidades: para o caso de requisição aceita, após a anexação dos dados será sinalizado o evento *syncDone* (descrito abaixo). Uma vez que esse comando é chamado, não são permitidas mais anexaçòes.

event result_t *syncDone*(result_t success): evento que é sinalizado ao término da sincronização dos dados da região de memória.

Parâmetros:

- success: informa se a sincronização foi bem-sucedida (i.e. SUCCESS) ou mal-sucedida (i.e. FAIL).

Particularidades: não aplicável.

B.4 API IOPorts

- Objetivo: prover envio/recepção de mensagens através da porta serial.

- Serviços:

  command result_t *init*(): prepara o componente de porta serial e seus sub-componentes para operação. Idem comando homônimo da API Radio, descrito em B.1

  command result_t *start*(): inicia o funcionamento do componente de porta serial e de seus sub-componentes. Idem comando homônimo da API Radio, descrito em B.1

  command result_t *stop*(): pára o funcionamento do componente de porta serial e de seus sub-componentes. Idem comando homônimo da API Radio, descrito em B.1

  command result_t *send*(TOS_MsgPtr msg): requisita o envio de mensagem através da porta serial.

  Parâmetros:

  – msg: ponteiro para mensagem a ser enviada.

  Estado:

  – SUCCESS: requisição aceita.

  – FAIL: requisição recusada.

  Particularidades: para o caso de requisição aceita, após o envio da mensagem será sinalizado o evento *sendDone* (descrito abaixo).

  event result_t *sendDone*(TOS_MsgPtr msg, result_t success): evento que é sinalizado ao término do envio de mensagem através da porta serial.

  Parâmetros:

  – msg: ponteiro para mensagem enviada.

  – success: informa se o envio foi bem-sucedido (i.e. SUCCESS) ou mal-sucedido (i.e. FAIL).

  Particularidades: não aplicável.

  event TOS_MsgPtr *receive*(TOS_MsgPtr m): evento sinalizado após a recepção de mensagem através da porta serial. Idem comando homônimo da API Radio, descrito em B.1

B.5 API Microcontroller

- Objetivo: prover acesso e controle do microcontrolador e temporizadores.
- Serviços:

  command void *enable*(): ativa o nó, de acordo com o nível de energia disponível.

  Parâmetros: não aplicável.

  Estado: não aplicável.

  Particularidades: não aplicável.

  command void *disable*(): desativa o nó, mantendo apenas os serviços básicos funcionando.

  Parâmetros: não aplicável.

  Estado: não aplicável.

  Particularidades: não aplicável.

  command result_t *start*(char type, uint32_t interval): inicia o temporizador.

  Parâmetros:

  - type: tipo do temporizador. Valores válidos incluem:
    - TIMER_REPEAT: temporizador dispara repetidamente.
    - TIMER_ONE_SHOT: temporizador dispara uma única vez.
  - interval: intervalo em *milisegundos binários* (1/1024 de segundo) entre cada evento de temporização.

  Estado:

  - SUCCESS: temporizador foi iniciado com o tipo e intervalo fornecidos.
  - FAIL: o tipo ou o intervalo informados são inválidos, ou já existem muitos temporizadores ativos.

  Particularidades: não aplicável.

  command result_t *stop*(): pára o temporizador.

  Parâmetros: não aplicável.

  Estado:

- SUCCESS: temporizador foi parado.
- FAIL: temporizador não estava executando.

Particularidades: se o temporizador for do tipo TIMER_ONE_SHOT e ele não foi disparado ainda, esse comando o impede de disparar.

event result_t *fired*(): evento que é sinalizado ao término do intervalo de temporização.

Parâmetros: não aplicável.

Particularidades: não aplicável.

Apêndice C

API do YATOS

Este capítulo descreve a API do YATOS , o sistema operacional desenvolvido. As APIs descritas abaixo integram o núcleo do microkernel, devendo estar sempre presentes no nó. Outras, específicas dos tipos de sensoriamento a ser executado, serão desenvolvidas a partir das APIs desse núcleo. Quanto aos métodos internos, eles são serviços providos pelos TADs respectivos, não sendo usados pelo usuário, e sim pelo microkernel para o seu funcionamento.

C.1 Formato de descrição das APIs

A descrição de cada API obedece à seguinte estrutura:

- Objetivo: informa a finalidade da API e/ou casos em que seu uso é recomendado.

  Assinatura: cabeçalho em formato similar a C.

  Descrição: explicação da funcionalidade do serviço de SO.

  Parâmetros: lista dos eventuais parâmetros de entrada e saída.

  Estado: lista dos possíveis valores de retorno, incluindo condições de erro.

  Particularidades: eventuais considerações e/ou restrições relacionadas à utilização do serviço de SO.

C.2 API Tarefa

Os nomes de todos os serviços abaixo são precedidos por *Tarefa_*.

- Objetivo: criação e postagem de tarefas no escalonador, atribuição de eventos às tarefas.

- Serviços:

id_t *Declara*(rotina_t rotina,prioridade_t prioridade, nome_t nome): cria uma tarefa para ser executada pelo SO.

Parâmetros:

- rotina: procedimento cujo parâmetro é *evento_t evento*.
- prioridade: prioridade da tarefa (1=mais baixa ... 255=mais alta).
- nome: string representando o nome da tarefa. Esse dado é opcional, podendo ser atribuído uma string vazia (“”). É usado para que uma tarefa possa postar outra.

Estado:

- identificador único da tarefa.

Particularidades: não aplicável.

sucesso_t *AtribuiEvento*(id_t idTarefa, evento_t evento, tipoPeriodo_t tipoPeriodo, periodo_t periodo):

Atribui eventos a ser aguardados por uma dada tarefa, o tipo de período (tiro único, periódico ou não-aplicável) e um periodo em ms, se aplicável. Se o evento não é temporizável, deve-se informar tipo de período *tpdNULO* e período *pdNULO*. Se o evento é disparado uma única vez, deve-se informar tipo de período *tpdTIROUNICO* e o período em ms. Se o evento é periódico, deve-se informar tipo de período *tpdPERIODICO* e o período em ms.

Parâmetros:

- idTarefa: identificador da tarefa.
- evento: evento ao qual a tarefa deve responder .
- tipoPeriodo: tipo do período de execução da tarefa .
- periodo: período em ms de execução entre tarefas, ou *pdNULO* se não aplicável.

Estado:

- SUCESSO: atribuição bem-sucedida.
- FALHA: falha na atribuição.

Particularidades: não aplicável.

sucesso_t *Posta*(nome_t nomeTarefa): posta uma tarefa para posterior execução no escalonador.

Parâmetros:

– nomeTarefa: string com o nome da tarefa.

Estado:

– SUCESSO: postagem bem-sucedida.

– FALHA: falha na postagem.

Particularidades: não aplicável.

C.3 API Rádio

Os nomes de todos os serviços abaixo são precedidos por *Radio_*.

- Objetivo: envio e recepção de dados através do rádio (interface *USART1* [40]), além da configuração de seu alcance.
- Serviços:

sucesso_t *Envia*(int envio): envia um dado pelo rádio.

Parâmetros:

– envio: dado a ser enviado.

Estado:

– SUCESSO: envio bem-sucedido.

– FALHA: falha no envio.

Particularidades: não aplicável.

int *Obtem*(): retorna o último dado recebido via rádio, que ainda não foi removido do buffer de recepção.

Parâmetros: não aplicável.

Estado:

– último dado recebido via rádio, ainda não removido do buffer de recepção.

Particularidades: a recepção do byte ocorre no tratador de interrupção, que armazena os dados recebidos em um buffer de recepção.

sucesso_t *AlteraAlcance*(int metros): altera o alcance do rádio.

Parâmetros:

– metros: novo alcance do rádio em metros.

Estado:

– SUCESSO: alteração bem-sucedida.

– FALHA: falha na alteração.

Particularidades: não aplicável.

C.4 API Sensor

Os nomes de todos os serviços abaixo são precedidos por *Sensor_*.

- Objetivo: sensoriamento do nó.
- Serviços:

sucesso_t *Sensor_ADCRequisitaLeitura*(sensor_t sensor): requisita uma leitura do sensor ligado ao conversor informad..

Parâmetros:

– sensor: sensor ADC ao qual será obtido a leitura.

Estado:

– SUCESSO: se requisição bem sucedida.

– FALHA: caso contrário.

Particularidades: não aplicável.

int *ObtemLeitura*(sensor_t sensor): retorna a leitura de um sensor.

Parâmetros:

– sensor: sensor que será lido.

Estado:

– Leitura do sensor.

Particularidades: Para sensores ADC, deve-se fazer primeiramente uma requisição para uma nova leitura, a menos que se deseje acessar um valor antigo. Para sensores DL, a leitura é feita instantaneamente, não sendo necessária requisição

de leitura. Para os sensores INT, eles apenas informam que um dado limiar foi ultrapassado, não podendo ser lidos.

C.5 API Memória

Os nomes de todos os serviços abaixo são precedidos por *Memoria_*.

- Objetivo: possibilitar leitura e escrita e apagamento da memória Flash do nó.
- Serviços:

  int * *Le*(int endereco, int comprimento): lê dados da memória Flash no endereço especificado.

  Parâmetros:

  – endereco: endereço da 1a. posição de memória de onde os dados serão lidos.

  – comprimento: quantidade de dados a serem lidos.

  Estado:

  – NULL: não foi possível ler os dados no endereço informado;

  – ponteiro para lista com os dados lidos.

  Particularidades: não aplicável.

  sucesso_t *Escreve*(int endereco, int comprimento, int * dados): escreve dados na memória Flash no endereço especificado.

  Parâmetros:

  – endereco: endereço da 1a. posição de memória de onde os dados serão escritos.

  – comprimento: quantidade de dados a serem escritos.

  Estado:

  – SUCESSO: escrita bem-sucedida.

  – FALHA: falha na escrita.

  Particularidades: não aplicável.

  sucesso_t *Apaga*(int endereco, int comprimento): apagada dados da memória Flash.

Parâmetros:

- endereco: endereço da 1a. posição de memória Flash a partir da qual os dados serão apagados.
- comprimento: quantidade de dados a serem apagados.

Estado:

- SUCESSO: apagamento bem-sucedido.
- FALHA: falha no apagamento.

Particularidades: não aplicável.

C.6 API Microcontrolador

void *Microcontrolador_IniciaHardware*(void): Inicia os componentes de hardware.

void *Microcontrolador_IniciaSecaoCritica*(void): Inicia uma seção critica, desabilitando interrupcoes do hardware.

void *Microcontrolador_FinalizaSecaoCritica*(void): Encerra uma seção critica, reabilitando interrupcoes do hardware.

void *Microcontrolador_Dorme*(void): Entra em modo de economia de energia.

void *Microcontrolador_IniciaSO*(void): Inicia o funcionamento do Sistema Operacional (SO).

void *Microcontrolador_ObtemId*(void): Informa o identificador unico do no.

Apêndice D

Código gerado nos Estudos de Caso

Este capítulo contêm o código gerado para o YATOS e o TinyOS, nos estudos de caso do Capítulo 6.

D.1 Leitura de dados de um sensor

D.1.1 Para o TinyOS

SensorLeds.nc

```
includes SensorLeds;
configuration SensorLeds {
  // this module does not provide any interface
}
implementation {
  components Main, SensorLedsM, LedsC, TimerC, SensorApi;
  Main.StdControl -> SensorLedsM;
  SensorLedsM.Leds -> LedsC;
  Main.StdControl -> TimerC;
  Main.StdControl -> SensorApi.TempADCControl;
  SensorLedsM.TempADC -> SensorApi.TempADC;
  SensorLedsM.TempADCControl -> SensorApi.TempADCControl;
  SensorLedsM.TempTimer -> TimerC.Timer[unique("Timer")];
}

```

SensorLedsM.nc

```
module SensorLedsM {
  provides {
    interface StdControl;
  }
  uses {
    interface Leds;
```

```
7       interface Timer as TempTimer;
8       interface ADC as TempADC;
9       interface StdControl as TempADCControl;
10    }
11  }
12  implementation {
13    command result_t StdControl.init() {
14      call Leds.init();
15      return SUCCESS;
16    }
17    command result_t StdControl.start() {
18      call TempTimer.start(TIMER_REPEAT, TEMP_SAMPLING_RATE);
19      return SUCCESS;
20    }
21    command result_t StdControl.stop() {
22      call TempTimer.stop();
23      return SUCCESS;
24    }
25    norace int _Leds_value;
26    uint8_t __Leds_light_queue;
27    task void _Leds_light() {
28      call Leds.set(_Leds_value);
29      atomic __Leds_light_queue = 0;
30    }
31    void limiarPronto(int nivel) {
32      atomic if(!__Leds_light_queue){
33        __Leds_light_queue = 1;
34        _Leds_value = nivel;
35        post _Leds_light();
36      }
37    }
38    norace int leituraEfetuada;
39    uint8_t _determinaLimiar_queue;
40    task void determinaLimiar() {
41      if(leituraEfetuada >= 0x80){
42        limiarPronto(leituraEfetuada >> 7);
43      }
44      atomic _determinaLimiar_queue = 0;
45    }
46    void _Temp_getData(int _Temp_data) {
47      atomic if(!_determinaLimiar_queue){
48        _determinaLimiar_queue = 1;
49        leituraEfetuada = _Temp_data;
50        post determinaLimiar();
51      }
```

```
52    }
53    event result_t TempTimer.fired() {
54      return call TempADC.getData();
55    }
56    async event result_t TempADC.dataReady(uint16_t data) {
57      _Temp_getData(data);
58      return SUCCESS;
59    }
60  }
61
```

D.1.2 Para o YATOS

SensorLeds.c

```
1  #include "SensorLeds.h"
2  #include "SO.h"
3  int _Leds_value;
4  uint8_t __Leds_light_queue;
5  void _Leds_light(evento_t evt) {
6    Microcontrolador_IniciaSecaoCritica();
7    __Leds_light_queue = 0;
8    Microcontrolador_FinalizaSecaoCritica();
9  }
10 void limiarPronto(int nivel) {
11   Microcontrolador_IniciaSecaoCritica();
12   if(!__Leds_light_queue){
13     __Leds_light_queue = 1;
14     _Leds_value = nivel;
15     Tarefa_Posta(_Leds_light);
16   }
17   Microcontrolador_FinalizaSecaoCritica();
18 }
19 int leituraEfetuada;
20 uint8_t _determinaLimiar_queue;
21 void determinaLimiar(evento_t evt) {
22   if(leituraEfetuada >= 0x80){
23     limiarPronto(leituraEfetuada >> 7);
24   }
25   Microcontrolador_IniciaSecaoCritica();
26   _determinaLimiar_queue = 0;
27   Microcontrolador_FinalizaSecaoCritica();
28 }
29 void _Temp_getData(int _Temp_data) {
30   Microcontrolador_IniciaSecaoCritica();
```

```
31    if(!_determinaLimiar_queue){
32      _determinaLimiar_queue = 1;
33      leituraEfetuada = _Temp_data;
34      Tarefa_Posta(determinaLimiar);
35    }
36    Microcontrolador_FinalizaSecaoCritica();
37  }
38  void tempoTask_Sensor_Temp(evento_t evt){
39    Sensor_ADCRequisitaLeitura(snADC0);
40  }
41  void sensorTask_Sensor_Temp(evento_t evt){
42    _Temp_getData(Sensor_ObtemLeitura(snADC0));
43  }
44  void main(){
45    Microcontrolador_IniciaHardware();
46    Tarefa_Declara(&determinaLimiar,128,determinaLimiar);
47    Tarefa_AtribuiEvento(
48      Tarefa_Declara(&tempoTask_Sensor_Temp,128,
49        tempoTask_Sensor_Temp)
50      ,evTEMPORIZADOR0,tpdPERIODICO,1000);
51    Tarefa_AtribuiEvento(
52      Tarefa_Declara(&sensorTask_Sensor_Temp,128,
53        sensorTask_Sensor_Temp)
54      ,evSENSORADC0,tpdNULO,pdNulo);
55    Microcontrolador_IniciaSO();
56  }
57
```

D.2 Fusão de dados

D.2.1 Para o TinyOS

SensorRadio.nc

```
1  includes SensorRadio;
2  configuration SensorRadio {
3    // this module does not provide any interface
4  }
5  implementation {
6    components Main, SensorRadioM, TimerC, SensorApi, RadioApi;
7    Main.StdControl -> SensorRadioM;
8    Main.StdControl -> SensorApi.PhotoADCControl;
9    SensorRadioM.PhotoADC -> SensorApi.PhotoADC;
10   SensorRadioM.PhotoADCControl -> SensorApi.PhotoADCControl;
11   SensorRadioM.PhotoTimer -> TimerC.Timer[unique("Timer")];
```

```
12    Main.StdControl -> TimerC;
13    Main.StdControl -> RadioApi;
14    SensorRadioM.ReceiveMsg -> RadioApi.ReceiveMsg[AM_RADIO];
15    SensorRadioM.SendMsg -> RadioApi.SendMsg[AM_RADIO];
16  }
17
```

SensorRadioM.nc

```
1   module SensorRadioM {
2     provides {
3       interface StdControl;
4     }
5     uses {
6       interface Timer as PhotoTimer;
7       interface ADC as PhotoADC;
8       interface StdControl as PhotoADCControl;
9       interface ReceiveMsg;
10      interface SendMsg;
11    }
12  }
13  implementation {
14
15    command result_t StdControl.init() {
16      return SUCCESS;
17    }
18    command result_t StdControl.start() {
19      call PhotoTimer.start(TIMER_REPEAT, PHOTO_SAMPLING_RATE);
20      return SUCCESS;
21    }
22    command result_t StdControl.stop() {
23      call PhotoTimer.stop();
24      return SUCCESS;
25    }
26    uint8_t __Base_recebeDados_queue;
27    norace int _Base_dados;
28    norace uint16_t _Base_enderecoFonte;
29    task void _Base_recebeDados() {
30        dbg(DBG_USR1, "EstacaoBase: Recebi leitura %d do no sensor %d\n",
31          _Base_dados, _Base_enderecoFonte);
32      atomic __Base_recebeDados_queue = 0;
33    }
34    void recvInt(int _Radio_recvInt_data,
35      uint16_t _Radio_sendInt_address) {
36      atomic if(!__Base_recebeDados_queue){
37        __Base_recebeDados_queue = 1;
38        _Base_dados = _Radio_recvInt_data;
```

```
39        _Base_enderecoFonte = _Radio_sendInt_address;
40        post _Base_recebeDados();
41      }
42    }
43    event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr
44      _Radio_recv_packet){
45      recvInt(*((int *)_Radio_recv_packet->data),
46       _Radio_recv_packet->addr);
47      return _Radio_recv_packet;
48    }
49    uint8_t _sendInt_queue;
50    norace int _Radio_sendInt_data;
51    norace int _Radio_sendInt_address;
52
53    typedef TOS_Msg Packet;
54
55    Packet _Radio_intPacket;
56    task void sendInt() {
57      *((int *)_Radio_intPacket.data) = _Radio_sendInt_data;
58      call SendMsg.send(_Radio_sendInt_address, sizeof(int),
59        &_Radio_intPacket);
60      atomic _sendInt_queue = 0;
61    }
62    event result_t SendMsg.sendDone(TOS_MsgPtr msg, bool success) {
63      return success;
64    }
65    void fusaoPronta(int dadosFusao, int endereco) {
66      atomic if(!_sendInt_queue){
67        _sendInt_queue = 1;
68        _Radio_sendInt_data = dadosFusao;
69        _Radio_sendInt_address = endereco;
70        post sendInt();
71      }
72    }
73    uint8_t _coletarLeitura_queue;
74    norace int leituraColetada;
75    // Numero maximo de dados coletados antes da fusao
76    enum {
77      MAX_DADOS = 10
78    };
79    // Endereco do no que atua como estacao-base
80    const int END_BASE = 0;
81    // Lista de dados coletados
82    int listaDados[MAX_DADOS];
83    // Ponteiro para o fim da lista de dados
```

```
84    int fimLista = 0;
85    task void coletarLeitura() {
86        int i;
87        int resultFusao;
88        // Se a lista de dados coletados estiver cheia, fundir os dados
89        if(fimLista == MAX_DADOS) {
90            resultFusao = 0;
91            for(i=0; i < MAX_DADOS; i++){
92                resultFusao += listaDados[i];
93            }
94            resultFusao /= MAX_DADOS;
95            fusaoPronta(END_BASE, resultFusao);
96        }
97        else {
98            listaDados[fimLista++] = leituraColetada;
99        }
100     atomic _coletarLeitura_queue = 0;
101   }
102   void _Photo_getData(int _Photo_data) {
103     atomic if(!_coletarLeitura_queue){
104       _coletarLeitura_queue = 1;
105       leituraColetada = _Photo_data;
106       post coletarLeitura();
107     }
108   }
109   event result_t PhotoTimer.fired() {
110     return call PhotoADC.getData();
111   }
112   async event result_t PhotoADC.dataReady(uint16_t data) {
113     _Photo_getData(data);
114     return SUCCESS;
115   }
116 }
117
```

D.2.2 Para o YATOS

SensorRadio.c

```
1  #include "SensorRadio.h"
2  #include "SO.h"
3  int _Leds_value;
4  uint8_t __Leds_light_queue;
5  void _Leds_light(evento_t evt) {
6    Microcontrolador_IniciaSecaoCritica();
```

```
7     __Leds_light_queue = 0;
8     Microcontrolador_FinalizaSecaoCritica();
9   }
10  void leituraProcessada(int _Base_leituraProcessada) {
11    Microcontrolador_IniciaSecaoCritica();
12    if(!__Leds_light_queue){
13      __Leds_light_queue = 1;
14      _Leds_value = _Base_leituraProcessada;
15      Tarefa_Posta(_Leds_light);
16    }
17    Microcontrolador_FinalizaSecaoCritica();
18  }
19  int _Base_enderecoFonte;
20  int _Base_dadosRecebidos;
21  uint8_t _receberLeitura_queue;
22  void receberLeitura(evento_t evt) {
23    leituraProcessada(_Base_dadosRecebidos >> 7);
24    Microcontrolador_IniciaSecaoCritica();
25    _receberLeitura_queue = 0;
26    Microcontrolador_FinalizaSecaoCritica();
27  }
28  void recvInt(int _Radio_recvInt_data, int _Radio_recvInt_address) {
29    Microcontrolador_IniciaSecaoCritica();
30    if(!_receberLeitura_queue){
31      _receberLeitura_queue = 1;
32      _Base_dadosRecebidos = _Radio_recvInt_data;
33      _Base_enderecoFonte = _Radio_recvInt_address;
34      Tarefa_Posta(receberLeitura);
35    }
36    Microcontrolador_FinalizaSecaoCritica();
37  }
38  void taskRadioRecvInt(evento_t evt){
39  }
40  int _Radio_sendInt_address;
41  int _Radio_sendInt_data;
42  uint8_t _sendInt_queue;
43  void sendInt(evento_t evt) {
44    Microcontrolador_IniciaSecaoCritica();
45    _sendInt_queue = 0;
46    Microcontrolador_FinalizaSecaoCritica();
47  }
48  void fusaoPronta(int _Fusao_dadosFusao, int _Fusao_endereco) {
49    Microcontrolador_IniciaSecaoCritica();
50    if(!_sendInt_queue){
51      _sendInt_queue = 1;
```

```
52      _Radio_sendInt_data = _Fusao_dadosFusao;
53      _Radio_sendInt_address = _Fusao_endereco;
54      Tarefa_Posta(sendInt);
55    }
56    Microcontrolador_FinalizaSecaoCritica();
57  }
58  int leituraObtida;
59  uint8_t _coletarLeitura_queue;
60  // Numero maximo de dados coletados antes da fusao
61  const int MAX_DADOS = 10;
62  // Endereco do no que atua como estacao-base
63  const int END_BASE = 0;
64  // Lista de dados coletados
65  int listaDados[MAX_DADOS];
66  // Ponteiro para o fim da lista de dados
67  int fimLista = 0;
68  void coletarLeitura(evento_t evt) {
69    int i;
70    int resultFusao;
71    // Se a lista de dados coletados estiver cheia, fundir os dados
72    if(fimLista == MAX_DADOS) {
73      resultFusao = 0;
74      for(i=0; i < MAX_DADOS; i++){
75        resultFusao += listaDados[i];
76      }
77      resultFusao /= MAX_DADOS;
78      fusaoPronta(resultFusao, END_BASE);
79    }
80    else {
81      listaDados[fimLista++] = leituraColetada;
82    }
83    );
84    Microcontrolador_IniciaSecaoCritica();
85    _coletarLeitura_queue = 0;
86    Microcontrolador_FinalizaSecaoCritica();
87  }
88  void _Photo_getData(int _Photo_data) {
89    Microcontrolador_IniciaSecaoCritica();
90    if(!_coletarLeitura_queue){
91      _coletarLeitura_queue = 1;
92      leituraObtida = _Photo_data;
93      Tarefa_Posta(coletarLeitura);
94    }
95    Microcontrolador_FinalizaSecaoCritica();
96  }
```

```
97  void tempoTask_Sensor_Photo(evento_t evt){
98    Sensor_ADCRequisitaLeitura(snADC1);
99  }
100 void sensorTask_Sensor_Photo(evento_t evt){
101   _Photo_getData(Sensor_ObtemLeitura(snADC1));
102 }
103 void main(){
104   Microcontrolador_IniciaHardware();
105   Tarefa_Declara(&receberLeitura,128,receberLeitura);
106   Tarefa_Declara(&coletarLeitura,128,coletarLeitura);
107   Tarefa_AtribuiEvento(
108   Tarefa_Declara(&tempoTask_Sensor_Photo,128,tempoTask_Sensor_Photo)
109   ,evTEMPORIZADOR1,tpdPERIODICO,1000);
110   Tarefa_AtribuiEvento(
111   Tarefa_Declara(&sensorTask_Sensor_Photo,128,sensorTask_Sensor_Photo)
112   ,evSENSORADC1,tpdNULO,pdNulo);
113   Tarefa_AtribuiEvento(
114   Tarefa_Declara(&taskRadioRecvInt,128,taskRadioRecvInt)
115   ,evRadio,tpdNULO,pdNULO);
116   Microcontrolador_IniciaSO();
117 }
```