FELLIPE AUGUSTO UGLIARA

DESENVOLVIMENTO DE PLATAFORMAS PARA REDES DE SENSORES SEM FIO BASEADAS NO SISTEMA OPERACIONAL TINYOS

LAVRAS – MG
2010

FELLIPE AUGUSTO UGLIARA

DESENVOLVIMENTO DE PLATAFORMAS PARA REDES DE SENSORES SEM FIO BASEADAS NO SISTEMA OPERACIONAL TINYOS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

Orientador:
Dr. João Carlos Giacomin

LAVRAS - MG
2010

FELLIPE AUGUSTO UGLIARA

DESENVOLVIMENTO DE PLATAFORMAS PARA REDES DE SENSORES SEM FIO BASEADAS NO SISTEMA OPERACIONAL TINYOS

Monografia de graduação apresentada ao Departamento de Ciência da Computação da Universidade Federal de Lavras como parte das exigências do curso de Ciência da Computação para obtenção do título de Bacharel em Ciência da Computação.

APROVADO em ____ de _____ de _____.

Dr. João Carlos Giacomin UFLA

Dr. Wilian Soares Lacerda UFLA

Dr. Luiz Henrique Andrade Correia UFLA

Orientador:
Dr. João Carlos Giacomin

LAVRAS - MG
2010

RESUMO

Neste trabalho é apresentada a documentação detalhada dos passos que devem ser seguidos para o projeto e implementação de plataformas de hardware para nós sensores que utilizam o sistema operacional TinyOS. O objetivo é disponibilizar, não somente um manual, mas também uma referência que permita entender de forma fácil como funciona a programação em TinyOS utilizando a linguagem NesC. É feita uma descrição dos componentes do TinyOS mostrando sua estrutura em blocos. A construção de um nó sensor utilizando um rádio que opera na faixa ISM de 2,4 GHz é apresentada como estudo de caso, onde se descrevem as implementações de blocos de código para o TinyOS especificas para esta nova plataforma.

Palavras-chave: Redes de Sensores Sem Fios, TinyOS, NesC, Nós Sensores.

ABSTRACT

This report presents the detailed documentation of the steps that must be followed for the design and implementation of hardware platforms for sensor nodes using TinyOS operating system. The goal is to provide not only a manual but a reference that implementations of blocks of TinyOS code specific to this new platform. It is a description of the components of TinyOS showing its block structure. The construction of a sensor node using a radio that operates in the 2.4 GHz ISM band is presented as a case study which describes the implementations of blocks of code to the TinyOS specific to this new platform.

KEYWORDS: *WIRELESS SENSOR NETWORKS, TINYOS, NESC, SENSOR NODES.*

SUMÁRIO

1 INTRODUÇÃO....4
1.1 Objetivos do trabalho....5
1.2 Motivação....6
1.3 Estrutura do trabalho....7
2 REFERENCIAL TEÓRICO....8
2.1 Hardware e software....9
2.2 Plataformas comerciais....12
2.2.1 Plataforma MICA2....12
2.2.2 Plataforma MICAZ....13
2.3 O Sistema operacional TinyOS....15
3 MATERIAIS E MÉTODOS....17
3.1 Conceitos básicos....17
3.1.1 Ambiente de desenvolvimento....18
3.1.2 Gcc Avr....19
3.1.3 TinyOS e NesC....20
3.1.4 Arquitetura abstrata de hardware....34
3.2 Como criar novos chips e novas plataformas....36
3.2.1 Criando uma nova plataforma....37
3.2.2 Tool-chain....39
3.2.3 Plataforma....42
3.2.4 Chips....46
4 RESULTADOS....47
4.1 Implementação de uma nova plataforma....47
4.2 Escolhendo o hardware e montando o esquema....47
4.3 Passo a passo....50
4.4 Testando a Implementação....56
5 CONCLUSÕES E PROPOSTAS DE CONTINUIDADE....65
REFERÊNCIAS....67
APÉNDICE A – Códigos....68

1 INTRODUÇÃO

Rede de Sensores Sem Fio (RSSF) é uma classe das redes *ad hoc*, a qual tem o objetivo de monitorar algum fenômeno da natureza ou realizar medições em ambientes industriais, domésticos, comerciais e outros. As RSSF são especialmente úteis em aplicações em locais de difícil acesso e em áreas perigosas, como florestas e vulcões.

Uma Rede de Sensores sem Fio (RSSF) consiste de um grande número de dispositivos, denominados nós sensores, os quais têm pouca disponibilidade de energia e se comunicam de uma forma não guiada. Esses dispositivos trabalham de modo colaborativo para realizar medições de forma distribuída em um ambiente. A premissa básica de uma RSSF e fazer um grande número de dispositivos pouco sofisticados, que trocam informação através de uma rede sem fio, trabalhem em conjunto para realizar tarefas que, convencionalmente, dispositivos mais caros e sofisticados fariam. As possíveis vantagens de uma RSSF sobre uma abordagem convencional podem ser resumidas em maior cobertura, precisão, confiabilidade e possivelmente um menor custo.

Há quatro componentes básicos numa RSSF: Nós sensores, uma rede não guiada, uma central de informação e um conjunto de recursos computacionais. (SOHRABY; MINOLI, 2007).

Os principais componentes de um nó sensor são a fonte de energia, o rádio transceptor e o microcontrolador, o qual coordena todo seu funcionamento. O microcontrolador possui internamente uma pequena área de memória que permite a instalação de um programa ou até mesmo de um sistema operacional, desde que este seja pequeno. Alguns sistemas operacionais foram desenvolvidos para RSSF, entre eles o *TinyOS*.

O *TinyOS* é um sistema operacional *open-source* que foi desenvolvido especificamente para redes de sensores sem fio e é largamente utilizado nas

pesquisas nesta área. Suas principais características são:

a) um conjunto de bibliotecas que incluem protocolos de rede, serviços distribuídos, *drivers* para sensores, ferramentas de aquisição de dados;

b) a possibilidade de incorporação rápida de inovações por ter uma arquitetura modular;

c) arquitetura voltada a eventos a fim de promover economia de energia;

d) portabilidade para diferentes plataformas.

Pesquisas na área de RSSF acontecem com foco voltado para os componentes, o sistema ou as aplicações. A pesquisa em nível de componente cuida do *hardware*, nas formas de comunicação e nas capacidades individuais de cada dispositivo. No nível do sistema está concentrada nos protocolos de rede, controle dos componentes para economia de energia e escalabilidade do sistema. O nível das aplicações está focado na aquisição de dados do ambiente e na execução de ações remotas como acionamentos.

O campo de aplicações de uma RSSF é amplo, englobando a área militar, civil, acadêmica e comercial. Muitas aplicações são propostas para esse tipo de sistema. Entre elas estão, coleta de dados, monitoramento de ambientes, vigilância, telemetria médica e controle a distância.

“Mais de 500 grupos de pesquisa e companhias usam ou já usaram o *TinyOS*.” (STOJMENOVIC, 2005, p. 46).

1.1 Objetivos do trabalho

O principal objetivo do trabalho é desenvolver um guia técnico que ensine passo a passo como projetar e implementar novas plataformas para *TinyOS*. Uma plataforma consiste de um conjunto *hardware* e *software* que possibilita o desenvolvimento de aplicações para as RSSF. O hardware são os nós sensores e os demais componentes como placas de gravação entre outros. E o software consiste de um sistema que torne possível a criação de programas para as mais diversas aplicações que serão compatíveis com diferentes tipos de nós sensores.

Outros objetivos:

a) Desenvolver uma referencia para a arquitetura do *TinyOS*;
b) Estudar os componentes presentes no mercado.

1.2 Motivação

A motivação de escrever um guia para o desenvolvimento de novas plataformas está fundamentada na possibilidade de que universidades e outros grupos de pesquisa construção novos modelos de nós sensores que sejam compatíveis com o *TinyOS*. Eliminando um dos problemas encontrados quando são realizadas pesquisas com RSSF, que é o acesso aos equipamentos necessários, como os nós sensores.

E documentar como o *TinyOS* esta organizado, mapeando sua arquitetura. Pois não existe material organizado e atualizado sobre o tema.

O desenvolvimento de novas plataformas mais baratas e acessíveis contribui com a área como um todo visto que ficando mais independente das plataformas comerciais a tecnologia torna-se mais acessível para estudos e inovações.

1.3 Estrutura do trabalho

No capítulo 2 são apresentados os trabalhos existentes na literatura que se relacionam com o problema estudado e com o método a ser desenvolvido. No capítulo 3 são apresentados os conhecimentos básicos para desenvolver novas plataformas e o guia de como projetar e implementar plataformas ou como adicionar componentes. No capítulo 4 estão os resultados descrevendo um uso do guia em um exemplo real. No capítulo 5 são apresentadas as conclusões decorridas da execução do projeto.

2 REFERENCIAL TEÓRICO

Os elementos de uma RSSF, denominados Nós Sensores, devem ter a capacidade de se comunicar através de uma rede sem fio que possui gerenciamento de topologia, uma lógica para o processamento de sinal e formas de manipular a transmissão. Devem ser robustos e de baixo custo, e devem utilizar componentes miniaturizados.

O projeto de um Nó Sensor envolve diferentes áreas de pesquisa, entre elas estão banco de dados, redes, algoritmos, sistemas distribuídos e diversas áreas da engenharia. Por isso a pesquisa em Nós Sensores e RSSF demandam um grupo grande de cientistas e engenheiros.

As funcionalidades típicas de um Nó Sensor e uma RSSF dependem diretamente da aplicação. Exemplos:

a) Determinar o valor de parâmetros do ambiente. Por exemplo, num ambiente controlado determinar temperatura, pressão atmosférica, quantidade de luz solar e umidade;
b) Detectar a ocorrência de eventos de interesse e estimar parâmetros do evento. Exemplo, num tráfego controlado determinar ou identificar se um veículo está passando e qual a sua velocidade;
c) Classificar um objeto que foi previamente detectado. Para o exemplo anterior definir se o veículo é uma moto, um caminhão ou um ônibus;
d) Rastrear objetos. Exemplo, em uma aplicação militar, rastrear um inimigo que está se movendo por uma área monitorada pela Rede de Sensores.

A coleta de dados normalmente necessita de um sistema de tempo real ou quase real, que leva a necessidade de projetar Nós Sensores que atendam a tal

demanda.

Uma nota importante é que os sensores devem ser passivos como sensores de temperatura, sísmicos, acústicos, de força e umidade, e não ativos como radares e sonares, pois consomem muita energia.

Os principais sensores incluem, mas não se limitam a sensores mecânicos, químicos, térmicos, elétricos, magnéticos, biológicos, fluídicos, ópticos, ultra sônicos e de força. Os Nós Sensores devem ser capazes de suportar ambientes com características hostis como altas temperaturas, altas vibrações ou meios corrosivos. Devem ser pequenos, de baixo custo, robustos e confiáveis para que possam realizar medições práticas e econômicas.

Nós Sensores e sensores podem possuir uma variedade grande de configurações dependendo das necessidades da aplicação.

2.1 *Hardware* e *software*

No projeto de um Nó Sensor as seguintes funções devem ser suportadas: processamento digital de sinais, compressão, correção de erros e criptografia; controle e atuação; clusters e computação em rede; automontagem; comunicação; roteamento e encaminhamento; e gerenciamento de conexão de rede. Para suportar essas funções, os componentes de *hardware* devem incluir sensores e unidades de atuação, unidade de processamento, unidade de comunicação, unidade de energia e outras unidades dependendo da aplicação. A Figura 1 mostra os componentes típicos de *hardware* e *software* de um Nó Sensor.

São componentes de *Hardware*:

a) Energia. Uma infra-estrutura de energia apropriada deve fornecer

condição de o sistema operar por algumas horas, meses ou anos, dependendo da aplicação.

b) Unidade de Processamento. É usada para processar e manipular os dados, pelo armazenamento de longo e curto prazo, criptografia, correção de erros, modulação digital e transmissão digital. O requisito computacional típico de um Nós Sensores varia de microcontroladores de 8-bits a microprocessadores de 64-bits. O armazenamento típico varia de 0,01 a 100 gigabytes (GB).

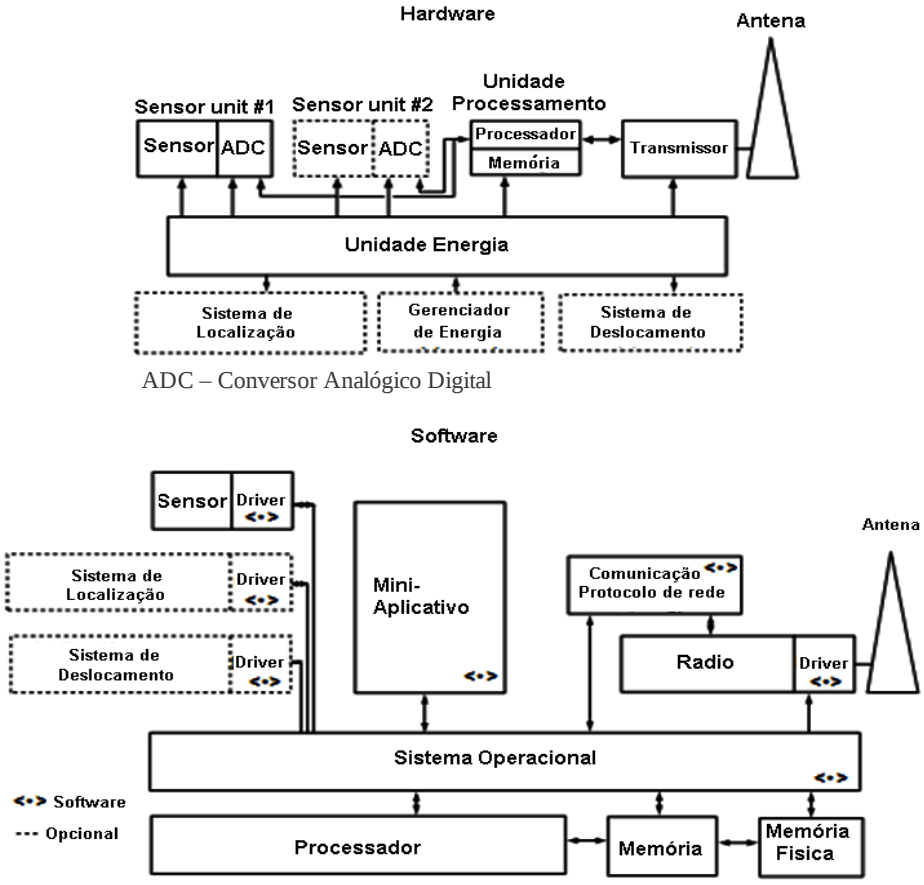

Figura 1 Componentes de *Hardware* e *Software* (SOHRABY; MINOLI, 2007).

c) Sensores. Os principais fenômenos a serem observados com o auxilio de sensores são aceleração, umidade, luz, fluxo magnético, temperatura, pressão e som.
d) Comunicação. O Nós Sensores deve ser capaz de se comunicar através de um sistema baseado em malhas com conectividade de rádio entre múltiplos Nós Sensores, utilizando roteamento dinâmico. A comunicação pode ser ponto a ponto ou multiponto a ponto com conectividade com um único Nó Sensor, utilizando rotas estáticas. A distância de comunicação varia de alguns metros a alguns quilômetros. E a largura de banda varia de 10 a 256 kbps.

São componentes de *Software*:

a) Sistema Operacional. Ele deve ser de código pequeno e garantir funcionalidades básicas ao sistema para que softwares direcionados a aplicações possam ser executados sobre a arquitetura do microcontrolador ou microprocessador.
b) *Driver* de Sensores. É o *software* que gerencia as funções básicas dos sensores.
c) Processos de Comunicação. Este código gerencia as funções de comunicação como rotas, *buffer* de pacotes, encaminhamento de pacotes, manutenção de topologia, controle de acesso ao meio e criptografia.
d) *Driver* de Comunicação. Este código gerencia as funções básicas do canal de transmissão do rádio como sincronização, codificação de sinal, bit de recuperação, bit contador, nível do sinal e modulação.
e) Processamento de Dados. São processamentos como numéricos, de dados, valor e manipulação de sinal e outros processamentos básicos

para aplicações.

2.2 Plataformas comerciais

São os componentes de hardware desenvolvidos por empresas privadas para comporem a rede de sensores sem fio entre eles estão nós sensores, placas de gravação, dispositivos para saída direta para a internet entre outros.

As duas principais plataformas são MICA2 e MICAZ, fabricados pela Crossbow Technology (CROSSBOW TECHNOLOGY, 2010).

2.2.1 Plataforma MICA2

A arquitetura do MICA2, Figura 2, permite o desenvolvimento de aplicações personalizadas e é otimizada para o baixo consumo de energia. Baseia-se no microcontrolador ATMEGA128L de baixa potência, que pode ser configurado para executar o aplicativo de sensoriamento, o rádio e a pilha de comunicação simultaneamente. Ela também oferece uma interface de 51 pinos onde podem ser conectados dispositivos externos como entradas analógicas, UART (*Universal Asynchronous Receiver/Transmitter*) e uma grande variedade de periféricos. Informações técnicas estão no Quadro 1.

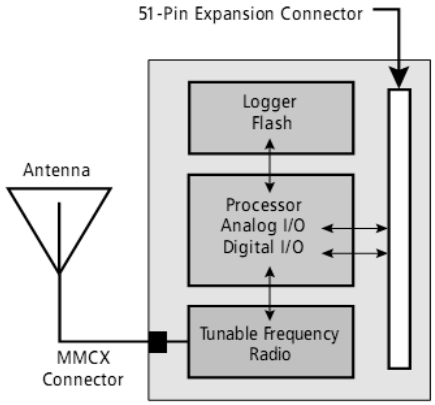

Figura 2 Diagrama de Bloco MICA2 (CROSSBOW TECHNOLOGY, 2010).

Quadro 1 - Especificação do MICA2 (CROSSBOW TECHNOLOGY, 2010).

PROCESSADOR / RÁDIO		
PROCESSADOR		OBSERVAÇÕES
Memória de programa	128 K bytes	
Memória de dados	512 K bytes	>100.000 medições
Configuração EEPROM	4 K bytes	
Comunicação Serial	UART	0-3 V nível de transmissão
Conversor AD	10 bits ADC	8 canais, 0-3 V entrada
Outras Interfaces	DIO, I2C, SPI	
Corrente	8 mA	Ativo
	<15 µA	Hibernando
RÁDIO		
Freqüência	868/916 MHz	Bandas ISM
Número de canais	4/ 50	Configurável, Leis Locais
Velocidade Transmissão	38.4 Kbaud	Codificação Manchester
Potência Tx.	-20 para +5 dBm	Configurável
Sensibilidade Rx.	-98 dBm	
Consumo	27 mA	Máxima Potência
	10 mA	Recepção
	<1 µA	Hibernando

Aplicações:

a) Segurança, vigilância;

b) Monitoramento Ambiental;

c) Rede de grande escala (>1000 nós);

d) Plataforma de computação distribuída;

2.2.2 Plataforma MICAZ

A arquitetura do MICAZ, Figura 3, permite o desenvolvimento de aplicações personalizadas e é otimizada para o baixo consumo de energia.

Baseia-se no microcontrolador ATMEGA128L de baixa potência, que pode ser configurado para executar o aplicativo de sensoriamento, o rádio e a pilha de comunicação simultaneamente. Ela também oferece uma interface de 51 pinos onde podem ser conectados dispositivos externos como entradas analógicas, UART e uma grande variedade de periféricos. Informações técnicas estão no Quadro 2. A maior diferença entre o MICAZ e o MICA2 está no rádio que agora possui uma velocidade mais alta de 250 kbps, comparado com os 38,4 Kbaud do MICA2 possibilitando novas aplicações como sensoriamento de áudio, vídeo e de vibrações devido à velocidade elevada de transmissão de dados.

Quadro 2 - Especificação do MICAZ (CROSSBOW TECHNOLOGY, 2010).

PROCESSADOR / RÁDIO		
PROCESSADOR		OBSERVAÇÕES
Memória programável	128 K bytes	
Medição (Serial)	512 K bytes	>100.000 medições
Configuração EEPROM	4 K bytes	
Comunicação Serial	UART	0-3 V nível de trans
Conversor AD	10 bits ADC	8 canais, 0-3 V entrada
Outras Interfaces	DIO, I2C, SPI	
Corrente	8 mA	Ativo
	<15 µA	Hibernando
RÁDIO		
Freqüência	2400/2483.5 MHz	Bandas ISM
Velocidade Transmissão	250 Kbps	
Potência Tx.	-24 para 0 dBm	
Sensibilidade Rx.	-94/-90 dBm	
Consumo	19.7 mA	Recepção
	11 mA	TX, -10 dbm
	14 mA	TX, -5 dbm
	17,4 mA	TX, 0 dbm
	20 µA	Modo ocioso
	1 µA	Hibernando

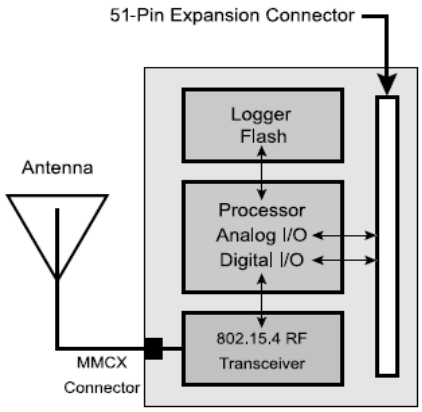

Figura 3 Diagrama de Bloco MICAZ (CROSSBOW TECHNOLOGY, 2010).

2.3 O Sistema operacional *TinyOS*

O *TinyOS* foi projetado para permitir que o *software* aplicativo se comunique diretamente com o hardware quando necessário. Ele tenta abordar duas questões principais: como garantir os fluxos simultâneos de dados entre dispositivos de hardware e como fornecer componentes modulares com pouco processamento e armazenamento.

O sistema utiliza um modelo baseado em eventos para suportar operações simultâneas utilizando uma quantidade pequena de memória. Ele pode criar rapidamente tarefas associadas a um evento, sem bloqueio ou pesquisa. Quando a CPU está ociosa, o processador é mantido dormente para economizar energia.

Ele inclui um agendador de tarefas reduzido e um conjunto de componentes. Cada componente consiste de quatro partes: um manipulador de comandos, um manipulador de eventos, um quadro encapsulado de tamanho fixo e um grupo de tarefas. Comandos e tarefas são executados no contexto de um quadro. Cada componente declara seus comandos e eventos para permitir a modularidade e a interação com outros componentes.

O agendador de tarefas é uma fila simples com uma programação estruturada de dados muito pequena e visa economia de energia visto que o processador fica dormindo quando a fila está vazia ou quando os dispositivos periféricos estão ainda em execução. O quadro é de tamanho fixo e é atribuído estaticamente, tendo os requisitos de memória especificados em tempo de compilação para retirar a sobrecarga da atribuição dinâmica.

Os comandos são feitos pelos componentes de baixo nível, portanto os comandos geralmente não têm que ficar esperando para serem executados, fornecendo um retorno caso sejam bem sucedidos na execução. As ocorrências de eventos de *hardware* invocam os manipuladores de evento e os manipuladores de evento podem armazenar informações de seu quadro, atribuir tarefas e chamar comandos de alto e baixo nível. Como os eventos, as tarefas podem chamar componentes de baixo nível, emitir tarefas de alto nível e atribuir outras tarefas.

Essa arquitetura do *TinyOS* define três tipos estruturas: uma abstração do *hardware*, *hardware* sintéticos e componentes de *software* de alto nível. O primeiro é uma abstração dos componentes de mais baixo nível, eles são o mapeamento do *hardware* físico, tais como dispositivos de I/O, o rádio e sensores. O segundo mapeia o comportamento do hardware avançado um nível acima se apoiando na abstração de *hardware*. O último leva o código já para a abstração da aplicação. Esse modelo gera vantagens como:

a) Diminui o tamanho do código voltado para a aplicação;
b) Facilita a abstração da aplicação final;
c) Permitir modularidade.

3 MATERIAIS E MÉTODOS

Essa pesquisa foi realizada no Departamento de Ciência da Computação no período de agosto de 2009 a junho de 2010. Foi organizada como segue:

a) Estudo do funcionamento do *TinyOS* e do *NesC*;
b) Organização da arquitetura do sistema *TinyOS* e do ambiente de desenvolvimento;
c) Pesquisa das plataformas de hardware existentes;
d) Descrição do processo de adição de novos componentes e criação de novas plataformas;
e) Adição de um novo rádio a um nó sensor.

Este capítulo está dividido em duas partes. A primeira apresenta conceitos básicos sobre o *TinyOS* e a segunda parte é o guia teórico de como implementar uma nova plataforma para este sistema operacional.

3.1 Conceitos básicos

Esta sessão tem o objetivo de apresentar e explicar como a arquitetura do sistema é montada e organizada para que novos componentes possam ser desenvolvidos.

O *TinyOS* é um sistema operacional que foi desenvolvido fazendo uso do compilador *Gcc* e da linguagem *NesC* . Para entender o *TinyOS* primeiro é preciso ter uma visão dessas duas plataformas. Será discutido brevemente o *Gcc* e depois do *NesC* junto com o *TinyOS*. Começando pela instalação do ambiente de desenvolvimento.

3.1.1 Ambiente de desenvolvimento

Esta sessão tem por objetivo descrever como preparar um ambiente de desenvolvimento para o *TinyOS* que funcione corretamente.

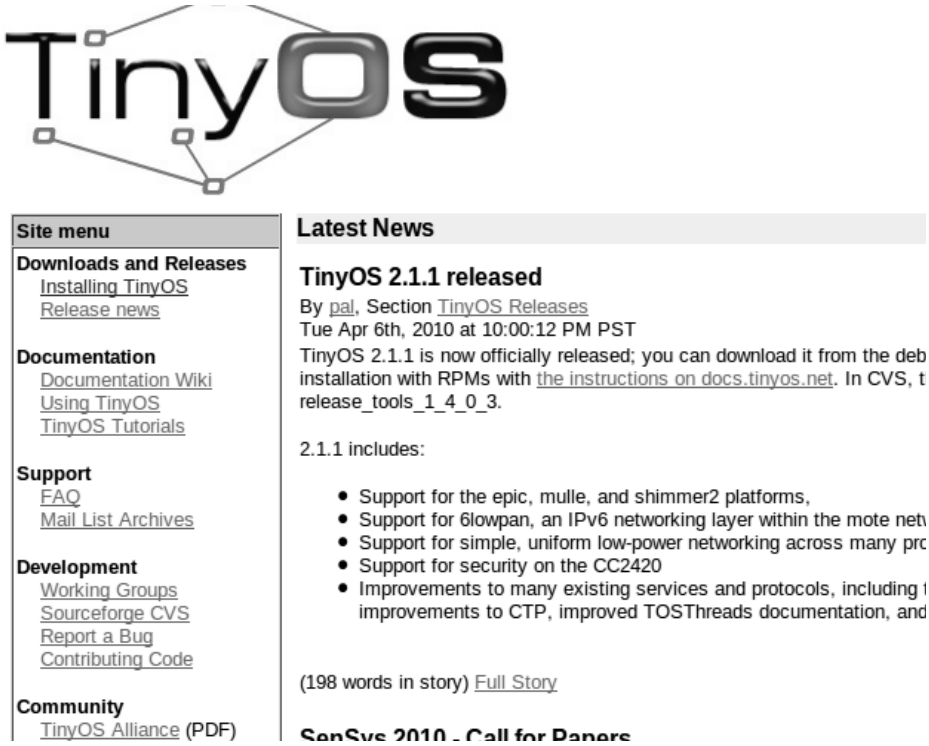

Figura 4 Pagina Inicial do Site do *TinyOS* (TINYOS COMMUNITY FORUM, 2010).

No site do *Tinyos* (http://www.tinyos.net/) podem ser encontradas as ferramentas necessárias para instalar o ambiente de desenvolvimento. O mais importante para dar continuidade são os menus "*install TinyOS*" e "*Documentation Wiki*". No primeiro menu estão os tutoriais para instalar o ambiente de desenvolvimento nas plataformas mais comuns, *Linux*, *Windowns* e *Mac*, o segundo menu tem toda a documentação mais atualizada sobre o *TinyOs* que pode ser encontrada, que vai desde tutoriais básicos até descrições

detalhadas dos módulos do sistema.

Como ambiente de desenvolvimento será usado o *Linux*, onde existem três opções de instalação para o *TinyOS*. Usar uma distribuição de qualquer e instalar os pacotes, emular uma distribuição pronta (*XubunTOS*) para uso em uma máquina virtual ou baixar o *live* CD do XubunTOS. É fundamental que o ambiente funcione corretamente então o *live* CD é uma excelente opção.

3.1.2 Gcc Avr

O *TinyOs* é um sistema que tem por objetivo abstrair o hardware da aplicação fazendo uso de diversas camadas de software, o *Gcc Avr* fica na primeira camada do *TinyOS* é através de suas bibliotecas que o *TinyOS* acessa o hardware.

O *Gcc* é o compilador oficial do projeto GNU e pode compilar código para diversas plataformas como *i386*, *ARM*, *MIPS* e muitas outras incluindo as plataformas da *Avr* como o *Atmega128* ou o *Atmega32*.

O *Gcc* usa uma técnica chamada de compilação cruzada. Quando instalamos o *TinyOS* ele instala junto o *crosscompiler* para as plataformas *Avr.* A compilação cruzada (*crosscompilation*) é feita por um compilador que é executado sobre uma plataforma para gerar código binário para ser executado em outra plataforma.

O *Gcc Avr* possui diversas bibliotecas para a implementação de código, para a família de microcontroladores *Avr* e é sobre ele que o *TinyOS* foi desenvolvido.

No *Linux* os arquivos ficam em `/usr/avr`, os cabeçalhos das funções que podem ser usadas estão em `/usr/avr/include`, e referência a elas vão ser freqüentes nas implementações da camada de *hardware* do *TinyOS*.

3.1.3 *TinyOS* e *NesC*

É difícil falar em *NesC* e não falar em *TinyOS* ou o inverso, pois a influência de um no outro torna complicado separá-los na hora de entender como as coisas funcionam, pois as idéias presentes no *TinyOS* antes mesmo dele ser implementado em *NesC*, influenciou o desenvolvimento do *NesC.*

*M*uito do que era escopo do *TinyOS* já está vinculado ao *NesC*, por isso ambos serão tratados como um só. Sem a divisão encontrada em outras literaturas.

A arquitetura do sistema é baseada em componentes, o sistema de concorrência é baseado em tarefas e eventos e a execução é dividida em fases.

3.1.3.1 Organização e componentes

Os programas escritos em *NesC* são organizados em três tipos de arquivos: interfaces, módulos e configurações.

a) Interface: é o invólucro do módulo que define os serviços providos e usados pelos módulos que as chamam. Modelo Figura 5.

```
interface <name> {
}
```

Figura 5 Modelo de código para Interface.

b) Módulo: é a implementação das interfaces, o código que será executado quando um serviço for usado. Modelo na Figura 6. O módulo usa interfaces e fornece interfaces para ligação com outros módulos.

```
module <nome> {
      provides {
            interface <nome>;
      }
      uses {
            interface <nome>;
      }
}
implementation {
}
```

Figura 6 Modelo de código para o modulo.

```
configuration <name> {
      provides {
            interface <name>;
      }
      uses {
            interface <name>;
      }
}
implementation {
      components <name>, <name>, <name>;
}
```

Figura 7 Modelo de código para a configuração.

c) Configuração: organiza os módulos e outras configurações, ligando suas interfaces para criar os componentes de serviços e abstrair os blocos do programa. As configurações não possuem blocos de código como os módulos. Apenas indicam ligações entre as interfaces dos componentes. Modelo na Figura 7.

A arquitetura baseada em componentes permite a criação de código reusável e independente, pois ao fechar um componente ele vai fornecer algumas interfaces e usar outras ficando independente das demais implementações, podendo cada aplicação escolher quais componentes pretende usar eliminando

os componentes que não são importantes para ela. Entendendo melhor.

Exemplo como implementar:

A interface *Timer,* Figura 8, implementada em *NesC*.

```
interface Timer {
     command result_t start();
     command result_t stop();
     event result_t fired();
}
```

Figura 8 Interface *Timer*.

Ela possui dois comandos e um evento que serão detalhados posteriormente. Por hora basta saber que o modulo que usar a interface deve implementar o evento no caso o `fired()`, pois durante a execução quando esse modulo usar os comandos de `Timer` recebera sinais de retorno para executar eventos. Quando receber um sinal para `fired()` executa o que estiver implementado no evento.

O módulo que fornecer `Timer` deve implementar os comandos no caso o `start()` e o `stop()` e os sinais para quem usa a interface.

Implementação dos módulos:

Um módulo pode fornecer serviços a outros módulos ou requisitar serviços de outros módulos. Quando um serviço é requisitado o modulo executa o comando correspondente. Ao final da execução, normalmente é enviado um sinal a quem requisitou, informando a conclusão do serviço. Quem requisitou deverá executar um evento para tratar o sinal recebido.

```
module Y {
      uses {
            interface Timer;
      }
}
implementation {
      call Timer.start();
      event result_t Timer.fired(){
      /* Aqui o Modulo Y recebe o sinal enviado pelo
      modulo X e executa as funções necessárias para
      tratar o evento */
      }
}
```

Figura 9 Modulo Y.

```
module X {
      provides {
            interface Timer;
      }
}
implementation {
      command result_t Timer.start(){
            signal Timer.fired();
            /* Quando o comando start é requisitado
            pelo modulo X, o modulo Y apenas envia
            um sinal de confirmação */
      }
      command result_t Timer.stop(){}
}
```

Figura 10 Modulo X.

Os modulo Y, Figura 9, e o modulo X, Figura 10, demostram a forma como uma interface deve ser usada ou provida.

O fluxo de execução ocorre da seguinte forma. Quando o módulo `Y` chamar o comando `start()` do modulo `X`, o código implementado no comando sera executado e um sinal sera enviado para o evento em `Y` no caso o `fired()` e o código que foi implementado em `fired()` é executado, quando

terminar a execução volta para `X`.

Agora como dizer para `Y` que o comando `start()` deve vir de `X` e que `X` deve mandar o sinal para `Y`? Usando uma configuração que irá ligar os dois componentes.

Implementação da configuração:

```
configuration Z {
      provides {
            interface P;
      }
      uses {
            interface D;
      }
}
implementation {
      components X, Y, T;
      T.P = P;
      T.D = D;
      Y.Timer -> X.Timer;
}
```

Figura 11 Configuração Z

Esta configuração Z, Figura 11, define como os módulos X e Y estão ligados. Componentes listam os módulos e as configurações que serão usados aqui. Os componentes das configurações podem ser outras configurações que serão ligadas a outros módulos ou outras configurações, criando camadas para o sistema. As ligações são feitas de forma que quem usa é ligado a quem provê. Então o Timer de Y deve ser ligado ao Timer de X, portanto, na declaração Y.Timer -> X.Timer, o símbolo -> representa a ligação entre Y e X de quem usa para quem provê.

Temos também o componente `T` que no caso provê uma interface P e usa uma interface `D`. Como saber isso sem ter o módulo de `T`? E possível porque a configuração `Z` provê `P` e usa `D` e iguala essas interfaces as interfaces `P` e `D` de `T`,

no caso `T.P=P` e `D=T.D`. As configurações também podem prover e usar interfaces mas não as implementam como os módulos pois o papel da configuração é de abstrair os módulos e ligá-los. Aqui `Z` abstrai `T` e liga `X` com `Y` mas poderia criar relações mais complexas como prover interfaces de dois módulos diferentes unindo os dois em uma mesma abstração.

Na Figura 12 abstrai-se os módulos `Q` e `W` em uma configuração `R` criando um componente. Isso pode ser feito por vários motivos como unir módulos diferentes para obter uma função mais geral, ou unir um módulo que recebe mensagens com um que envia para termos um componente rádio. Quando alguma aplicação for usar o rádio, basta acrescentar o componente rádio que fornece interfaces apropriadas para o uso do rádio.

```
configuration R {
     provides {
          interface A;
          interface B;
     }
}
implementation {
     components Q, W;
     Q.A = A;
     W.B = B;
}
```

Figura 12 Exemplo de encapsulamento de módulos.

3.1.3.2 Escalonamento

O *TinyOS* possui um sistema de escalonamento baseado em duas estruturas, Eventos e Tarefas. O escalonador possui também uma fila de tarefas que envia uma tarefa para a execução sempre que o processador fica disponível.

a) Evento: os eventos são executados até terminar, mas podem intercalar sua execução com a execução de tarefas e outros eventos.

b) Tarefas: são estruturas de execução adiáveis. Tarefas não interferem entre si, não são como os eventos que podem dividir execução com outros eventos ou outras tarefas. Uma vez em execução, a tarefa divide o processador com os eventos que tem execução prioritária. Só quando a tarefa termina, uma nova tarefa pode ser chamada. Quando uma tarefa é chamada ela vai para a fila de tarefas para esperar sua vez de executar. Como as tarefas executam do inicio ao fim elas devem ser curtas, não devem bloquear recursos nem entrar em espera ocupada para não bloquear as demais tarefas. As tarefas também podem enviar sinais.

Exemplo:

Os eventos e as tarefas são implementados em *NesC* dentro dos módulos. Primeiro implementaremos uma soma sem o uso de *task* e seguiremos sua execução. Em seguida faremos a mesma coisa só que com o uso de *task* e seguiremos sua execução. Usaremos uma interface, dois módulos e uma configuração.

A interface, Figura 13, possui um serviço de soma com um comando onde são passados os valores a serem somados e um evento que receberá o resultado da soma.

```
interface A {
     command soma(int a, int b);
     event void resultado(int resposta);
}
```

Figura 13 Interface A.

O módulo `B`, Figura 14, provê a interface `A` e realiza a soma quando o

comando `soma` da interface `A` for chamado. Quando termina a execução do comando, é sinaliza da chamada para o evento resultado.

```
module B {
      provides {
            interface A;
      }
}
implementation {
      command A.soma(int a,int b) {
            int temp;
            temp = a + b;
            signal resultado(temp);
      }
}
```

Figura 14 Modulo B.

```
module C {
      uses {
            interface A;
            interface Boot;
      }
}
implementation {
      int soma;

      event void Boot.booted() {
            call A.soma(2,5);
            soma = 0;
      }

      event void A.resultado(int resposta) {
            soma = resposta;
      }
}
```

Figura 15 Modulo C

O módulo `C`, Figura 15, tem o evento `Boot.booted` que chama o

comando `soma`, a execução passa para o comando `soma`, realiza a soma e chama o evento resultado que executa. A execução volta para soma que não tem mais nada a fazer e encerra voltando para `boot` que chama a atribuição (`soma = 0;`) em seguida.

A configuração, Figura 16, cria a ligação entre os módulos `C` e `B`. E a suas interfaces `A`.

```
configuration L {}
implementation {
	components Main,B, C;

	C -> MainC.Boot;
	C.A -> B.A;
}
```

Figura 16 Configuração L

```
module B {
	provides {
		interface A;
	}
}
implementation {

	int valor1,valor2,soma;

	task void Tsoma() {
		soma = valor1 + valor2;
		signal resultado(soma);
	}

	command A.soma(int a,int b) {
		valor1 = a;
		valor2 = b;
		post Tsoma();
	}
}
```

Figura 17 Módulo B alterado.

Agora vêm as tarefas que são executadas de forma diferente, para isso serão feitas algumas alterações no módulo `B`, Figura 17. As tarefas são implementadas como *task* e são declaradas como funções do tipo *void*, pois não retornam valores. Sua chamada é feita com *post* e não recebem argumentos, pois devem ser independentes e desvinculadas do resto da execução, visto que vão para uma fila e não é determinado o instante em que serão executadas. As *task* usam variáveis globais apenas, as quais são declaradas no início da implementação.

Com essa alteração a execução passa a ter o seguinte formato. O evento `boot.booted` inicia a execuçao do modulo C e chama o comando `A.soma` que será executado no módulo B. O comando `A.soma` realiza duas atribuições e coloca a *task* `Tsoma` na fila de tarefas a serem executadas. Aqui a execução começa a mudar, a task `Tsoma` vai para a fila de tarefas e a execução continua no comando `A.soma` até ele terminar e volta para `boot.booted`, que termina a execução chamando a atribuição.

Mas e a *task* `Tsoma`? Como uma tarefa `Tsoma` está na fila de tarefas e agora o processador está desocupado, ela é posta em execução. A `Tsoma` é realizada e sinaliza o evento resultado que passa a ser executado devolve a soma e termina a execução.

A diferença sutil é muito importante que seja entendida visto que o sistema todo faz uso dela, para poder criar a divisão de fases.

É importante observar que retornos de comandos são recebidos por quem chama o comando, e retorno de eventos são recebidos por quem sinaliza o evento.

O *TinyOS* faz uso das *task* para dividir longas execuções em fases. Por exemplo, um comando simples como acender um *led* não é dividido em fases, pois executa de forma rápida e libera o processador. Algo mais complicado como enviar uma mensagem pelo rádio é dividida em várias fases pois assim

podem ser executada aos poucos e deixar que o processador atenda outras requisições. O comando *send* é chamado, ele coloca uma *task* na fila e vai em frente com a execução. Quando a *task* for resolvida ela coloca a outra fase na fila até que o envio seja feito e um sinal de término enviado.

3.1.3.3 Concorrência

No *TinyOS* existem duas formas de concorrência, a que ocorre durante a execução normal do código entre os eventos e as tarefas e as geradas por interrupções de hardware. A diferença está na forma como ocorrem, as geradas por eventos e tarefas são síncronas enquanto as geradas por interrupções são assíncronas.

Para proteger partes do código que não podem sofrer concorrência com interrupções usamos a diretiva *atomic*, Figura 18, exemplo:

```
command A.soma(int a,int b) {
     atomic {
          valor1 = a;
          valor2 = b;
     }
     post soma();
}
```

Figura 18 Exemplo de uso do *atomic*.

3.1.3.4 Estrutura de diretório do *TinyOS*

A árvore de diretórios do *TinyOS* está dividida em três diretórios principais dentro do diretório `/tinyos-2.x`, que é a raiz do sistema: `/apps`, `/support`, `/tos`. As funções de seus diretórios e subdiretórios principais serão descritas a seguir.

a) `/apps`, `/apps/demos`, `/apps/tests`, `/apps/tutorials` – contém as aplicações divididas em diferentes propósitos como aprendizado, testes e demonstração;
b) `/support/make` – aqui estão os *scripts* que fazem a compilação, a geração de documentos e outras funções disponíveis pelo *TinyOS*;
c) `/tos/system` – aqui está o núcleo do *TinyOS* . Os principais componentes para o *TinyOS* funcionar estão aqui;
d) */tos/interface* – as interfaces que criam a abstração do hardware estão aqui e são usadas em todas as camadas do sistema;
e) `/tos/platforms` – contém a especificação dos nós sensores independente da forma como os chips foram implementados;
f) `/tos/chips` – contém a implementação dos chips e a abstração para ser usada pelas plataformas;
g) `/tos/lib` – contém as interfaces e componentes que estendem o *TinyOS*, mas que não é essencial para seu funcionamento.

3.1.3.5 Aprendendo a usar o ambiente *TinyOS*

Será descrito como compilar uma aplicação, como instalar uma aplicação em um nó e como usar a ferramenta de recepção de pacotes pela porta serial. Usando dois nós sensores MICAZ, Figura 20, uma placa para programação Mib520, Figura 19, e as aplicações *RadioCountToLeds* e *BaseStation* será realizada a tarefa.

Figura 19 Mib520 (CROSSBOW TECHNOLOGY, 2010).

Figura 20 MICAZ (CROSSBOW TECHNOLOGY, 2010).

a) Liga-se o MICAZ na porta USB de um computador com o ambiente de desenvolvimento *TinyOS* configurado, a placa de programação e comunicação mib520;

b) Abre-se o diretório da aplicação que fica na instalação do *TinyOS* no *Linux* em `/opt/tinyos-2.x/apps/ RadioCountToLeds`;

c) Agora a aplicação é compilada para a plataforma, Figura 21. "`make micaz`". Modelo: "`make <plataforma>`". OBSERVAÇÃO: um diretório *build* será criado e dentro dele ficam os diretórios com as plataformas para o qual o exemplo foi compilado (no caso MICAZ) e dentro dele estão os arquivos frutos da compilação que serão detalhados mais a frente;

```
Setting up for TinyOS 2.1.1
fellipe@fellipe-laptop:/opt/tinyos-2.1.1/apps/RadioCountToLeds$ make micaz
mkdir -p build/micaz
    compiling RadioCountToLedsAppC to a micaz binary
ncc -o build/micaz/main.exe  -Os -fnesc-separator=__ -Wall -Wshadow -Wnesc-all -
OUP=0x22 --param max-inline-insns-single=100000 -DIDENT_APPNAME=\"RadioCountToLe
USERHASH=0x1c9bed60L -DIDENT_TIMESTAMP=0x4be6f63cL -DIDENT_UIDHASH=0xf420d027L -
ed(interfacedefs, components)' -fnesc-dumpfile=build/micaz/wiring-check.xml Radi
/opt/tinyos-2.1.1/tos/chips/cc2420/lpl/DummyLplC.nc:39:2: warning: #warning "***
    compiled RadioCountToLedsAppC to build/micaz/main.exe
          11630 bytes in ROM
            311 bytes in RAM
avr-objcopy --output-target=srec build/micaz/main.exe build/micaz/main.srec
avr-objcopy --output-target=ihex build/micaz/main.exe build/micaz/main.ihex
    writing TOS image
fellipe@fellipe-laptop:/opt/tinyos-2.1.1/apps/RadioCountToLeds$
```

Figura 21 Tela de uma compilação bem sucedida com o TinyOS.

d) Se a compilação estiver OK é gravada a aplicação no nó, Figura 22. "`make micaz install mib510,/dev/ttyUSB0`". Modelo: "`make <plataforma> install mib510,<device>`". Quando a placa é ligada à porta USB ela monta dois *device-drives* no *Linux*: `/dev/ttyUSB0` e `/dev/ttyUSB1`. O primeiro é usado para gravação e o segundo para comunicação (operação). Durante a gravação o *led* vermelho ficará piscando;

```
Setting up for TinyOS 2.1.1
fellipe@fellipe-laptop:/opt/tinyos-2.1.1/apps/RadioCountToLeds$ make micaz insta
mkdir -p build/micaz
    compiling RadioCountToLedsAppC to a micaz binary
ncc -o build/micaz/main.exe  -Os -fnesc-separator=__ -Wall -Wshadow -Wnesc-all -
OUP=0x22 --param max-inline-insns-single=100000 -DIDENT_APPNAME=\"RadioCountToLe
USERHASH=0x1c9bed60L -DIDENT_TIMESTAMP=0x4be6f6b7L -DIDENT_UIDHASH=0x9635ea02L -
ed(interfacedefs, components)' -fnesc-dumpfile=build/micaz/wiring-check.xml Radi
/opt/tinyos-2.1.1/tos/chips/cc2420/lpl/DummyLplC.nc:39:2: warning: #warning "***
    compiled RadioCountToLedsAppC to build/micaz/main.exe
          11630 bytes in ROM
            311 bytes in RAM
avr-objcopy --output-target=srec build/micaz/main.exe build/micaz/main.srec
avr-objcopy --output-target=ihex build/micaz/main.exe build/micaz/main.ihex
    writing TOS image
cp build/micaz/main.srec build/micaz/main.srec.out
    installing micaz binary using mib510
uisp -dprog=mib510 -dserial=/dev/ttyUSB0 --wr_fuse_h=0xd9 -dpart=ATmega128  --wr
Firmware Version: 1.8
Atmel AVR ATmega128 is found.
Uploading: flash
Verifying: flash

Fuse High Byte set to 0xd9

Fuse Extended Byte set to 0xff
rm -f build/micaz/main.exe.out build/micaz/main.srec.out
fellipe@fellipe-laptop:/opt/tinyos-2.1.1/apps/RadioCountToLeds$
```

Figura 22 Tela de uma instalação bem sucedida com o TinyOS no micaZ.

e) Apos gravar o primeiro nó, vá ao diretório *app/BaseStation* e repita o processo para gravar o outro nó;

f) Deixe o nó gravado com o *BaseStation* ligado ao computador e execute, "`java net.tinyos.tools.Listen -comm serial@/dev/ttyUSB1:micaz`";

g) Um programa de leitura da porta serial será carregado, Figura 23, e ficara esperando dados que chegam pela porta serial;

```
Setting up for TinyOS 2.1.1
fellipe@fellipe-laptop:/opt/tinyos-2.1.1/apps/BaseStation$ java net.tinyos.tools
serial@/dev/ttyUSB1:57600: resynchronising
```

Figura 23 Tela de recepção de pacotes via serial.

h) Ligando o outro Nó sensor e as mensagens vão começar a chegar,

Figura 24.

```
Setting up for TinyOS 2.1.1
fellipe@fellipe-laptop:/opt/tinyos-2.1.1/apps/BaseStation$ java net.tiny
serial@/dev/ttyUSB1:57600: resynchronising
00 FF FF 00 01 02 00 06 00 01
00 FF FF 00 01 02 00 06 00 02
00 FF FF 00 01 02 00 06 00 03
00 FF FF 00 01 02 00 06 00 04
00 FF FF 00 01 02 00 06 00 05
00 FF FF 00 01 02 00 06 00 06
00 FF FF 00 01 02 00 06 00 07
```

As demais aplicações também podem ser testadas. Cada uma possui um arquivo *readme* que ajuda a entender como a aplicação deve funcionar. As que não possuem, basta ler o código, pois são mais simples que as com o arquivo explicativo. Além disso, a *TinyOS* também gera documentação da aplicação, basta executar o comando "`make <plataforma> docs`".

3.1.4 Arquitetura abstrata de hardware

Essa arquitetura, Figura 25, fornece mais portabilidade ao sistema que não fica dependente de *hardware*, o que simplifica o desenvolvimento das aplicações, escondendo o *hardware* do resto do sistema. A arquitetura está dividida em três níveis:

a) Camada da Apresentação de *hardware* (HPL): esse bloco promove o acesso direto ao hardware fazendo uso da memória e do mapa de portas de E/S (os pinos). Fornecendo funções mais simples para manipular o *hardware* e escondendo seus detalhes de operação e acesso das demais camadas do sistema. Para promover essa integração são criados:

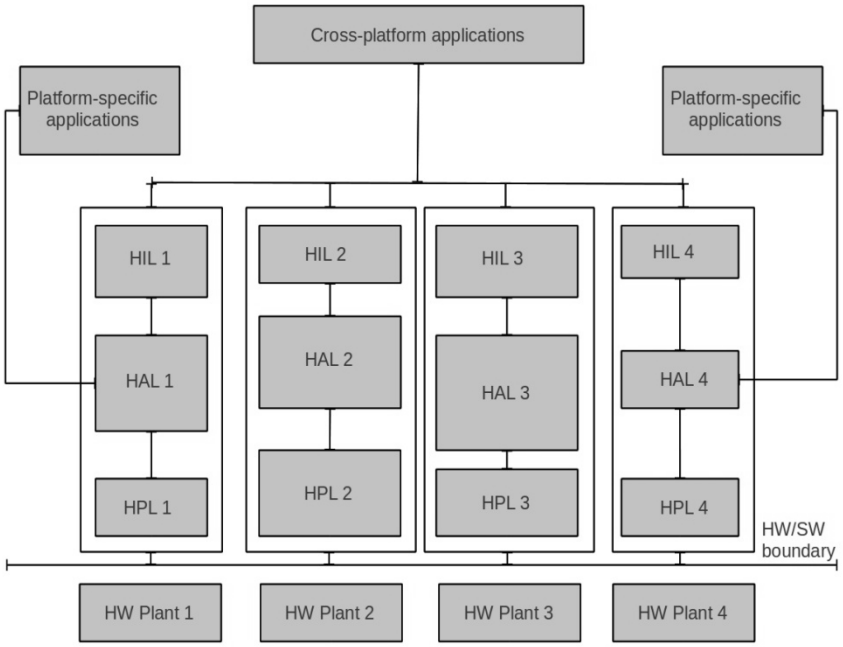

Figura 25 Arquitetura abstrata de *Hardware* (TINYOS COMMUNITY FORUM, 2010).

i) Comandos de inicialização, de *start*, *stop*;

ii) Comandos *Get* e *set* para configuração de alguns registradores;

iii) Comandos para habilitar e desabilitar interrupções e outros.

b) Camada da Adaptação de *hardware* (HAL): representa o núcleo da arquitetura, ela fica responsável por fazer a junção dos diversos recursos de *hardware* existentes, como o rádio e o microcontrolador, intermediando e controlando os recursos, escondendo as características de uso individuais das partes.

c) Camada da Interface de *Hardware* (HIL): esta camada unifica as diversas plataformas permitindo que uma aplicação compilada para um nó sensor possa ser compilada para os demais sem alteração na

aplicação. A complexidade desta camada está diretamente ligada ao que é fornecido pela camada de apresentação. Então ela só mudará quando novas características de hardware alcançar a camada de apresentação e só se essas novas características forem gerais para todas as plataformas.

3.2 Como criar novos *chips* e novas plataformas

Na Figura 26 é apresentado um resumo de como estão organizadas as plataformas no *TinyOS*. No topo está representada a camada de aplicação, o software independente da Plataforma alvo. Na transição está o sistema operacional que é usado para implementar a aplicação e o compilador usado para gerar o código para a maquina alvo. Em baixo está a a representação do hardware que é programado com o código alvo gerado.

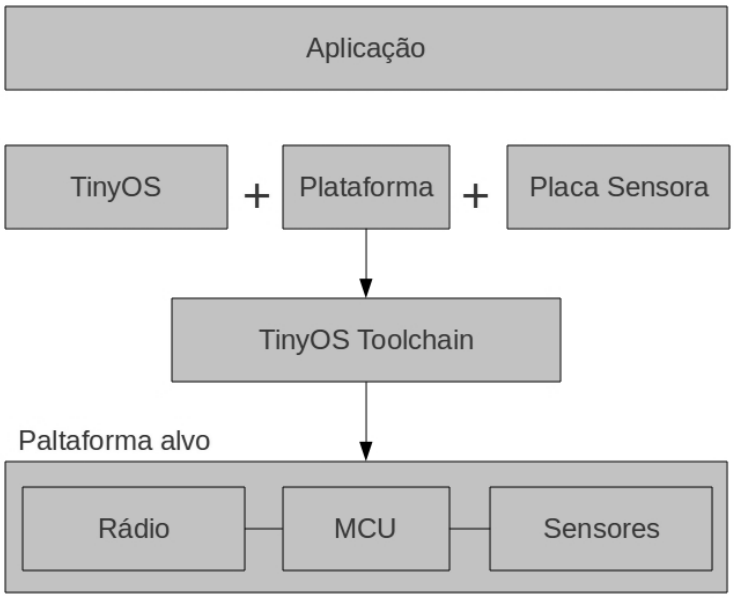

Figura 26 Organização do sistema (TINYOS COMMUNITY FORUM, 2010).

O *TinyOS* fornece um conjunto de códigos prontos e um conjunto de ferramentas para construir aplicações para os nós sensores. Uma plataforma no *TinyOS* expõe só as características gerais do nó escondendo as particularidades físicas de cada dispositivo (*chip*). Uma plataforma é uma junção de diversos dispositivos (*chips*) que forma um nó sensor.

3.2.1 Criando uma nova plataforma

Para definir uma nova plataforma alguns passos devem ser seguidos:

a) Modificar a *toolchain* para compilar para a nova plataforma;

b) definir ou implementar os chips que compõem a plataforma;

c) definir ou implementar como esses chips são combinados para montar a plataforma.

Para isso os arquivos certos devem ser adicionados ou modificados. Abaixo estão os locais onde serão feitas modificações e adições a árvore do *TinyOS*. Daqui para frente diremos plataformaX, MCUX para a nova plataforma e para a nova MCU (microcontrolador) que estaremos projetando.

As partes foram divididas e explicadas em separado:

a) Chips: implementa o acesso direto a um determinado hardware como rádios, microcontroladores, sensores e outros. Cada chip fica em um diretório separado em tos/chips. O diretório contém as camadas HPL e HAL da arquitetura do *TinyOS*. Se uma plataforma contém características particulares de um chip elas ficam localizadas em `tos/platforms/platformX/chips/chipX` para a plataformaX.

b) Plataforma: é a parte do quebra-cabeça que junta os chips para formar

a implementação do nó sensor. A codificação da plataforma deve ficar em `tos/platforms/plataformaX`. A plataforma *Null* é uma plataforma vazia que serve de exemplo para o desenvolvimento de novas plataformas.

c) *Tool-chain* (*make system*): fica localizado em `support/make` e é responsável por compilar, gerar documentação e outras tarefas do sistema; é o ambiente de desenvolvimento. Para construirmos uma nova plataforma temos que no mínimo criar ou modificar os seguintes arquivos:

a) criar um diretório para o nova plataforma em `tos/platforms/PlataformaX` e inserir nele:

ai) um arquivo de definição *.platform*

aii) um `hardware.`*h header*

b) Um `platformaX.target` em `tos/support/make`

c) `MCUX.rules` em `tos/support/make/MCUX`

d) *MCUX* diretório em `tos/chips/MCUX`, que contém

di)*McuSleepC* (define interrupções)

dii) `mcuxhardware.h` (controla início e fim de seções *atomic*)

Árvore de diretório tos, Figura 27:

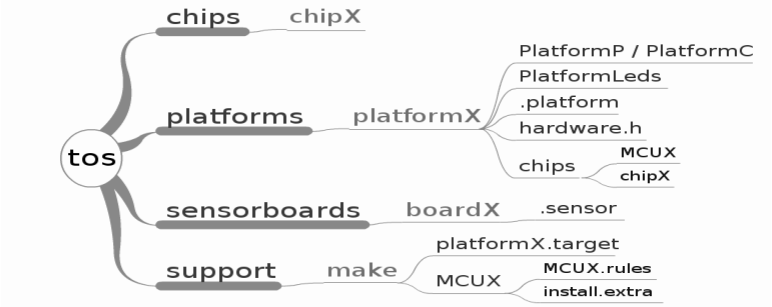

Figura 27 Árvore do `/tos` (TINYOS COMMUNITY FORUM, 2010).

3.2.2 *Tool-chain*

Esta sessão está dividida em duas partes. Primeiro o compilador e segundo o sistema que usa o compilador para gerar o arquivo binário e gravá-lo no nó sensor. Que é construído sobre do *make* do *Linux*.

3.2.2.1 Compilando usando *NesC*

O sistema recebe como entrada um código que é transformado em um arquivo binário, Figura 28, que é carregado para a plataforma. O compilador *Nesc* pré-compila o arquivo de entrada com duas ferramentas: o ncc e o *Nesc*. Essas duas ferramentas são usadas para converter o arquivo de entrada *NesC* em um único arquivo do tipo C que será compilado usando um compilador C padrão, o *Gcc*.

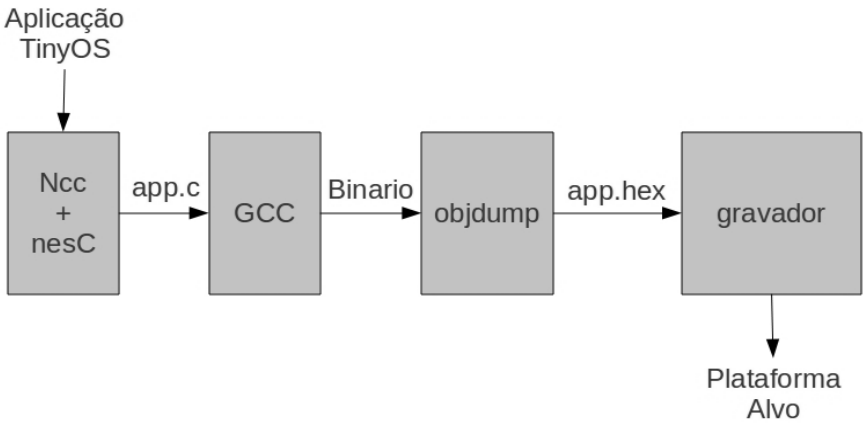

Figura 28 Processo de compilação (TINYOS COMMUNITY FORUM, 2010).

Como o *TinyOS* usa o *Gcc Avr,* o código escrito em *NesC* deve ser traduzido para um código em C que o *Gcc Avr* entenda para depois ele gerar o arquivo binário que será transformado em um `app.hex` que é usado pelo programa que grava o nó sensor, o arquivo `app.hex` é o mesmo binário só que

com informações adicionais para a gravação em hardware como endereços de memória e outros dados o programa que faz a gravação é o *avrdude*.

Estes arquivos são criados na pasta *build* que é gerada dentro do diretório da aplicação que foi compilada. É muito importante olhar o arquivo `app.c` que é o arquivo único em C gerado pelo *NesC*. Abra o `main.exe` (precisará de um programa para abrir arquivos binários, esse binário não pode ser executado no PC, pois ele é para arquitetura Avr) que é o arquivo binário gerado pelo Gcc.

3.2.2.2 Sistema "*make*" que controla a compilação

A compilação e outras funções são automatizadas usando o *make* do *Linux*. Um arquivo *make* global procura o *make* da plataforma específica, portanto não precisa alterar o *make* global, mas acrescentar os arquivos certos aos locais certos para que sejam buscados na hora da compilação. O que precisamos são os arquivos `platformaX.target` e `MCUX.rules` citados acima.

O *make* ao ser chamado procura pelo `.target` em `support/make` e nos diretórios listados na variável de ambiente `TOSMAKE_PATH`. Esse arquivo não contém as regras de como proceder com a compilação e a construção dos binários mas ele leva ao arquivo .rules apropriado que contém tais informações, apropriada a arquitetura *MCU* do nó. Muitas plataformas dividem o mesmo arquivo .rules mas tem arquivos `.target` individuais. Arquivos .extras podem ser usados para definir funções como *clean* e *install* para as plataformas de nós sensores. Para criar o sistema *make* para uma nova plataforma seguimos dois passos:

a) Criar o arquivo `plataform.target`. `BUILD_DEPS` é o montador

e as regras serão trazidas do .rules relacionado.

b) Criar o arquivo .rules que fica em um subdiretório de `tos/support/make`, aqui estão indicados como os binários e os *hex* serão criados, o montador é definido em `BUILD_DEPS`.

Exemplos, `mica2.target`, Figura 29:

```
PLATFORM = mica2
SENSORBOARD ?= micasb
PROGRAMMER_PART ?= -dpart=ATmega128 --wr_fuse_e=ff
PFLAGS += -finline-limit=100000
AVR_FUSE_H ?= 0xd9
$(call TOSMake_include_platform,avr)
mica2: $(BUILD_DEPS)
        @:
```

Figura 29 `mica2.target`.

E `avr.rules` para o Atmega128L do MICA2, Figura 30:

```
...
BUILD_DEPS = srec tosimage

srec: exe FORCE
        $(OBJCOPY) --output-target=srec $(MAIN_EXE)
$(MAIN_SREC)
tosimage: ihex build_tosimage FORCE
        @:

exe: builddir $(BUILD_EXTRA_DEPS) FORCE
        @echo "    compiling $(COMPONENT) to a
$(PLATFORM) binary"
        $(NCC) -o $(MAIN_EXE) $(OPTFLAGS) $(PFLAGS)
$(CFLAGS)
                  $(COMPONENT).nc $(LIBS) $(LDFLAGS)
...
```

Figura 30 `avr.rules`.

3.2.3 Plataforma

Uma plataforma, Figura 31, é um conjunto de componentes ligados de forma correta. Para definir uma plataforma existem três elementos principais que ficam em `tos/platform/PlataformaX`:

a) Definir arquivo `.platform`, que contém os caminhos e outros argumentos usados pelo *NesC*;
b) Processo de *Boot* da Plataforma: `PlatformP/PlatformC`;
c) O código fonte com as especificações de plataforma, o cabeçalho de hardware (`hardware.h`) e o código que combina os chips de forma correta para formar a plataforma.

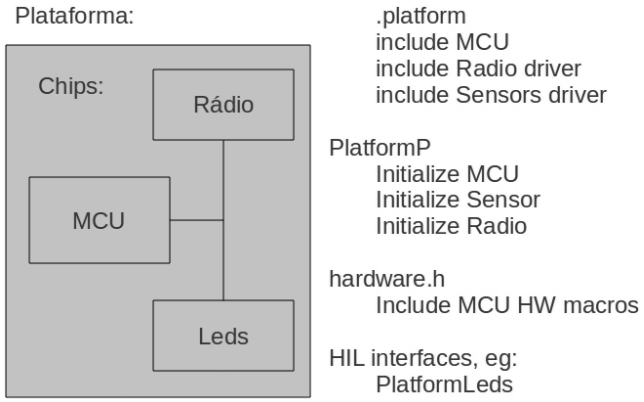

Figura 31 Organização da plataforma em arquivos (TINYOS COMMUNITY FORUM, 2010).

3.2.3.1 O arquivo `.platform`

Todas as plataformas têm um arquivo desses. Este arquivo define onde os códigos dos componentes usados na plataforma estão. O caminho do diretório em si. Este arquivo será usado pelo *make* na hora de compilar para saber onde deve procurar os arquivos necessários para a compilação. Opções adicionais

podem ser passadas neste arquivo para a compilação como o tipo de escalonamento, opção de *debug* entre outras.

Exemplo, Figure 32, MICA2:

```
push( @includes, qw(
  %T/platforms/mica
  %T/platforms/mica2/chips/cc1000
  %T/chips/cc1000
  %T/chips/atm128
  %T/chips/atm128/adc
  %T/chips/atm128/pins
  %T/chips/atm128/spi
  %T/chips/atm128/timer
  %T/lib/timer
  %T/lib/serial
  %T/lib/power
) );

@opts = qw(
  -gcc=avr-gcc
  -mmcu=atmega128
  -fnesc-target=avr
  -fnesc-no-debug
  -fnesc-scheduler=TinySchedulerC,
TinySchedulerC.TaskBasic,
TaskBasic,TaskBasic,runTask,postTask
);
```

Figura 32 `.platform` do MICA2.

3.2.3.2 `PlatformP e PlatformC`

Estes componentes são responsáveis pelo *boot* da plataforma chamando as rotinas iniciais para ajustar o relógio, iniciar os pinos (I/O) do microcontrolador e outras configurações.

Exemplo `PlatformP`, Figura 33,e `PlatformC`, Figura 34:

```
module PlatformP
{
      provides interface Init;
            uses interface Init as MoteInit;
            uses interface Init as MeasureClock;

}
implementation
{
      void power_init() {
      }

      command error_t Init.init()
      {
            error_t ok;

            ok = call MeasureClock.init();
            ok = ecombine(ok, call MoteInit.init());
            return ok;
      }
}
```

Figura 33 `PlataformaP`.

```
#include "hardware.h"
configuration PlatformC {
      provides {
            interface Init;
            interface Atm128Calibrate;
      }
      uses interface Init as SubInit;
} implementation {
      components PlatformP, MotePlatformC,
MeasureClockC;
      Init = PlatformP;
      Atm128Calibrate = MeasureClockC;

      PlatformP.MeasureClock -> MeasureClockC;
      PlatformP.MoteInit -> MotePlatformC;
      MotePlatformC.SubInit = SubInit;
}
```

Figura 34 `PlataformaC`.

A interface `init` é a primeira a ser executada quando um aplicação *TinyOS* começa a operar e ela pode ser adicionada a qualquer módulo.

3.2.3.3 Código especifico de plataforma

É o bloco de código que liga os componentes entre si de forma correta e implementa as funções necessárias para que o nó funcione com todas as suas características.

3.2.3.4 Arquivos de Cabeçalho

Existem arquivos de cabeçalho que o *TinyOS* precisa que estejam presentes no diretório da plataforma.

O arquivo `platform_message.h` é um exemplo de arquivo de cabeçalho:

```
typedef union message_header {
     cc1000_header_t cc1k;
     serial_header_t serial;
} message_header_t;

typedef union message_footer {
     cc1000_footer_t cc1k;
} message_footer_t;

typedef union message_metadata {
     cc1000_metadata_t cc1k;
} message_metadata_t;
```

Figura 35 Exemplo de estrutura genérica.

Os dois principais arquivos de cabeçalho são:

a) `hardware.h`: este cabeçalho é incluído no arquivo `tos/system/MainC.nc`.
b) `platform_message.h`: Figura 35, que é usado para definir o *buffer* de mensagens. E para que a interface genérica de mensagem possa ser definida.

3.2.3.5 *Chips* e Plataformas

Por último existem os *chips* que precisam ser ligados para compor a plataforma. Eles devem ficar em um diretório específico dentro da árvore da plataforma, devem ficar em: `platforms/PLATFORMX/chips/CHIPX`.

3.2.4 *Chips*

Estão localizados em `tos/chips` e são a implementação que faz acesso direto ao *hardware.* Como exemplo o chip `atm128` cuida do acesso direto ao microcontrolador Atmega128.

4 RESULTADOS

Este capítulo demonstra o uso do guia implementando um novo componente para o *TinyOS*. O rádio TRF24G e o integrando a plataforma MICAZ.

4.1 Implementação de uma nova plataforma

Finalmente será implementada a nova plataforma. Para realizar a tarefa, primeiro será escolhido qual o hardware do nó sensor. Como esse guia não tem por objetivo disponibilizar uma plataforma final será usado um nó sensor já existente e acrescentar a ele um novo componente, um dos mais importantes para um nó sensor: o rádio transmissor/receptor. Esta utilizará um microcontrolador Atmega128 e um rádio TRF24G.

4.2 Escolhendo o hardware e montando o esquema

Como plataforma base será usado o MICAZ que foi descrito anteriormente e o rádio usado será o TRF24G, Figura 36, que descreveremos agora. O rádio escolhido para a construção do Nó sensor foi o TRF-2.4G, que é similar ao rádio da plataforma MICAZ. Aqui serão listadas suas características e os motivos de utilizá-lo para a nova plataforma.

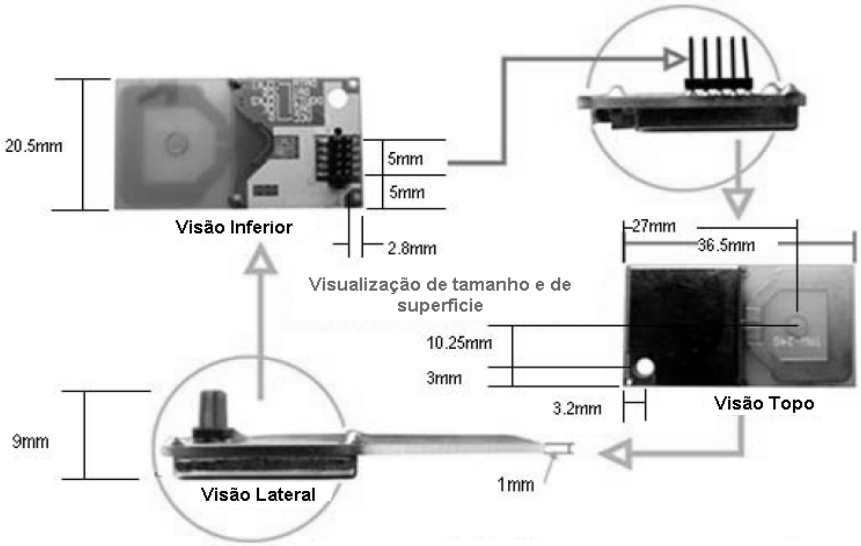

Figura 36 Visualização do tamanho e superfície do TRF-24G (RF-24G_DATASHEET, 2010).

Características:

a) Frequência: 2.4~2.524 GHz;

b)Velocidade: 1Mbps; 250Kbps;

c) Alimentação de 1,9 a 3,6V;

d) Possui modo de economia de energia (Hibernação).

Motivos para utilizá-lo:

a) Alta velocidade de transmissão como o rádio do MICAZ;

b) Modo econômico de energia, para quando não estiver em uso;

c) Similar ao rádio do MICAZ o que o torna portável para o *TinyOS* mais facilmente.

A ligação física do rádio TRF25G ao MICAZ é feita conforme indicado no Quadro 3. Os códigos da esquerda indicam os terminais do rádio e os da direita os terminais do microcontrolador Atmega128L. Eles são ligados diretamente através de fios metálicos.

Quadro 3 Esquema de ligação do radio.

Esquema Rádio/Atmega128			Observações
CE	←	PortC5	Chip Enable
CS	←	PortC6	Chip Select
CLK1	←	PortC2	Clock
DATA	↔	PortC4	Data
VCC	←	PortC0	Vcc
DRI	→	Int1	Interrupção

Todos estes são ligados ao microcontrolador Atmega128L que está no nó sensor MICAZ, conforme visto na Figura 37. Os primeiros terminais são ligados a portas de entrada e saída digital de uso comum.

O terminal DR1 muda de nível lógico sempre que uma mensagem for recebida pelo rádio. Este sinal será usado para acionar uma interrupção no Atmega128L. As setas do Quadro 3 definem o fluxo de sinais.

Na Figura 37 foi usada uma placa adicional a mda100cb para facilitar as ligações que são feitas diretamente dos pinos do rádio para os pinos do microcontrolador.

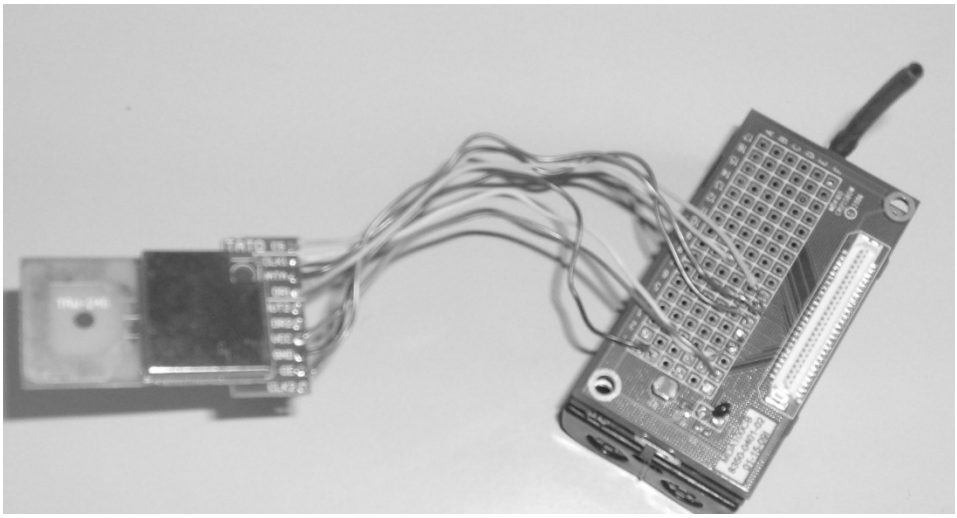

Figura 37 Radio TRF24G ligado ao Micaz.

4.3 Passo a passo

Com o *hardware* definido basta seguir os passos definidos anteriormente:

a) Modificar a *toolchain* para compilar para a nova plataforma (Passo 1);

b) definir ou implementar os chips que compõem a plataforma (Passo 2);

c) definir ou implementar como esses chips são combinados para montar a plataforma (Passo 3).

Passo 1:

O protótipo da plataforma uflaNode será construído sobre o nó sensor MICAZ, aproveitando toda a sua estrutura, menos a do rádio CC2420. Algumas modificações devem ser feitas na *toolchain* do MICAZ:

a) Criar o arquivo `uflaNode.target`;

b) Criar o arquivo `.rules`.

Para o arquivo `.rules` será usado o `avr.rules`, pois está sendo usado o mesmo *MCU* do MICAZ que usa esse arquivo de regras de compilação.

A Figura 38 apresenta um exemplo de arquivo .target para o uflaNode.

Passo 2:

Primeiro deve-se saber quais os chips serão usados, o que é simples pois como o MICAZ é tido como base serão os mesmo que ele usa então estes já estão implementados. Como o MICAZ usa o CC2420 como rádio e não existe uma implementação prévia para o TRF24G, é necessário implementar uma camada de software para acessá-lo. Caso houve-se outros chips que forem ser usados e que não estivessem implementados seria necessário implementá-los também.

```
#-*-Makefile-*- vim:syntax=make
#$Id: uflaNode.target,v 1.6 2008/07/09 15:36:50
sallai Exp $

PLATFORM = uflaNode
SENSORBOARD ?= micasb
PROGRAMMER ?= uisp
ifeq ($(PROGRAMMER),avrdude)
  PROGRAMMER_PART ?= -pm128  -U efuse:w:0xff:m
endif

ifeq ($(PROGRAMMER),uisp)
  PROGRAMMER_PART ?= -dpart=ATmega128  --wr_fuse_e=ff
endif

AVR_FUSE_H ?= 0xd9

ifdef TRF24G_CHANNEL
PFLAGS += -DCC2420_DEF_CHANNEL=$(TRF24G_CHANNEL)
endif
$(call TOSMake_include_platform,avr)
uflaNode: $(BUILD_DEPS)
     @:
```

Figura 38 `uflaNode.target`.

Seguindo o que foi visto anteriormente em chips deve-se implementar as camadas HPL e HAL. Então seguindo as instruções anteriores da sessão 3.2.4 devemos criar uma pasta em tos/chips com o nome do nosso chip (no caso trf24g) e dentro dela ficará a implementação.

Para garantir a nítida divisão das camadas, criaremos dois módulos `TRF24GActiveMessageP.nc` e `TRF24GModuleP.nc`, duas configurações `TRF24GActiveMessageC.nc` e `TRF24GModuleC.nc` e uma interface `TRF24GModule.nc`, e um cabeçalho `trf24g.h` com as constantes fixas para o rádio.

Os códigos dos arquivos podem ser encontrados no APÊNDICE A.

O módulo TRF24GModuleP faz a implementação da camada HPL do rádio pois ele acessa diretamente os pinos de I/O , ele liga e desliga a interrupção e controla as características próprias do rádio.

Acima dele tem-se o módulo TRF24GActiveMessageP que faz o papel da camada HAL onde a segunda abstração do hardware é feita quando as funções de envio e recepção recebem uma camada que as vincula as funções padrão de envio e recepção do *TinyoOS* escondendo o padrão de envio e recepção típicos do novo rádio. A função de envio do *TinyOS* podendo enviar pelo rádio CC2420 ou pelo TRF24G sem precisar pensar em suas características especificas.

Passo 3:

Como a plataforma MICAZ é a base, deve-se ir ao diretório de plataformas em `tos/platforms` e replicar o diretório do MICAZ. Mudando o nome para nossa plataforma uflaNode, então temos `tos/platforms/uflaNode` com o conteúdo do MICAZ.

E agora o que deve ser modificado para transformar o MICAZ em uflaNode e adicionar nosso novo rádio no lugar do CC2420.

Seguindo a receita anterior:

a) Definir o arquivo `.platform`, que contém os caminhos e outros argumentos usados pelo *NesC*;
b) Definir o *Boot* da Plataforma :`PlatformP/PlatformC`;
c) E definir código fonte com as especificações de plataforma, o cabeçalho de hardware (`hardware.h`) e o código que combina os chips de forma correta para formar a plataforma.

Passo 3.a:

Aqui é necessário definir todos os diretórios que fazem parte da nossa plataforma e acrescentá-los ao arquivo `.platform`, usando o do MICAZ como base.

Na Figura 32 é possível ver que estão listados vários diretórios em *chips*, alguns em *lib* e outros em *platforms*. Esses caminhos de diretórios devem ser alterados para atender a realidade da nova plataforma indicando os novos locais onde os *chips*, as *libs* e outras estão localizados.

Então, mudando o que é necessário para acrescentar nosso novo chip e acertando o nome da plataforma para uflaNode o arquivo fica igual a Figura 39.

```
push( @includes, qw(
  %T/platforms/mica
  %T/platforms/uflaNode/chips/cc2420
  %T/chips/cc2420
  %T/chips/cc2420/alarm
  %T/chips/cc2420/control
  %T/chips/cc2420/csma
  %T/chips/cc2420/interfaces
  %T/chips/cc2420/link
  %T/chips/cc2420/lowpan
  %T/chips/cc2420/lpl
  %T/chips/cc2420/packet
  %T/chips/cc2420/receive
  %T/chips/cc2420/spi
  %T/chips/cc2420/transmit
  %T/chips/cc2420/unique
  %T/platforms/uflaNode/chips/trf24g
  %T/chips/trf24g
  %T/platforms/mica2/chips/at45db
  %T/platforms/mica/chips/at45db
  %T/chips/at45db
  %T/chips/atm128
  %T/chips/atm128/adc
```

```
  %T/chips/atm128/pins
  %T/chips/atm128/spi

  %T/chips/atm128/i2c
  %T/chips/atm128/timer
  %T/lib/timer
  %T/lib/serial
  %T/lib/power

) );

@opts = qw(
  -gcc=avr-gcc
  -mmcu=atmega128
  -fnesc-target=avr
  -fnesc-no-debug
);
push @opts, "-fnesc-scheduler=TinySchedulerC,
TinySchedulerC.TaskBasic,
TaskBasic,TaskBasic,runTask,postTask"
if !$with_scheduler_flag;
push @opts, "-mingw-gcc" if $cygwin;

$ENV{'CIL_MACHINE'} =
    "version_major=3 " .
    "version_minor=4 " .
    "version=avr-3.4.3 " .
    "short=2,1, " .
"long=4,1 " .
    "long_long=8,1 " .
    "pointer=2,1 " .
    "enum=2,1 " .
    "float=4,1 " .
    "double=4,1 " .
    "long_double=4,1 " .
    "void=1,1 " .
    "fun=1,1 " .
    "wchar_size_size=2,2 " .
    "alignof_string=1 " .
"max_alignment=1 " .
    "char_wchar_signed=true,true " .
    "const_string_literals=true " .
```

```
    "big_endian=false " .
    "underscore_name=false " .
    "__builtin_va_list=true " .
    "__thread_is_keyword=true";
```

Figura 39 `.platform` do uflaNode.

O passo 3.b será pulado pois o *boot* é o mesmo do MICAZ que está sendo usado como base e o *boot* não precisa ser alterado.

Então começa o passo 3.c:

Aqui é definida a forma como será ligado o microcontrolador com o novo rádio e a terceira camada de abstração: a HIL.

Primeiro liga-se o rádio ao microcontrolador. Como depende de características do rádio e do Atmega128, então os arquivos de configurações `HplTRF24GPinsC.nc` e `TRF24GInterruptsC.nc` deve ficar em `tos/platforms/uflaNode/chips/trf24g`.

Os códigos dos arquivos podem ser encontrados no APÊNDICE A.

No primeiro arquivo ligamos os pinos que vamos usar e monitorar e no segundo arquivo ligamos a interrupção externa que avisa quando uma nova mensagem chega.

E para terminar só resta a ultima camada de abstração. No MICAZ, a HIL do CC2420 fica na raiz de diretórios da plataforma em `ActiveMessageC.nc`, então é esse arquivo que vamos alterar para trocar os rádios .

O código do arquivo pode ser encontrado no APÊNDICE A.

Agora tem-se uma Nova plataforma uflaNode com o rádio TRF24G que respeita os padrões de abstração do *TinyOS*. Onde é possível compilar e usar os demais comandos do ambiente de desenvolvimento, o *make* e gerar binários para o nó uflaNode a partir do *TinyOS*. Para testar, compile a aplicação *RadioCountToLeds* com "`make uflaNode`".

4.4 Testando a Implementação

Para testar a implementação do novo rádio sera usado dois nós sensores modificados com o novo rádio como na Figura 37 e dois computadores com o ambiente *TinyOS* instalado. O teste consiste em ligar um nó sensor a cada computador e realizar uma transmissão de um computador para o outro usando os nós sensores e os rádios TRF24G.

Como programa auxiliar é usado uma aplicação Java que recebe dados pela porta serial e envia dados pela mesma porta.

Então em um dos computadores será enviada uma mensagem pelo programa Java que direciona a mensagem para a porta serial aonde o nó sensor esta ligado. O nó sensor usa o rádio TRF24G para transmitir ao outro nó sensor que recebe a mensagem pelo rádio TRF24G conectado a ele e a encaminha para a porta serial em que esta conectado. A aplicação Java no outro computador recebe a mensagem da porta serial e imprime na tela.

Primeiro os nós sensores devem ser programados com o código das Figuras 40, 41 e 42. Que são respectivamente a configuração, o módulo e um cabeçalho para a aplicação:

```
configuration BaseStationC {
}

implementation {
```

```
  components MainC, BaseStationP, LedsC;
        components    ActiveMessageC    as    Radio,
SerialActiveMessageC as Serial;

  MainC.Boot <- BaseStationP;

  BaseStationP.RadioControl -> Radio;
  BaseStationP.SerialControl -> Serial;

  BaseStationP.RadioSend -> Radio.AMSend;
  BaseStationP.RadioReceive -> Radio.Receive;

  BaseStationP.UartSend -> Serial.AMSend;
  BaseStationP.UartReceive -> Serial.Receive;

  BaseStationP.Leds -> LedsC;
}
```

Figura 40 Configuração da aplicação teste.

```
#include "AM.h"
#include "Serial.h"
#include "SerialMsgs.h"

module BaseStationP {
  uses {
    interface Boot;
    interface SplitControl as SerialControl;
    interface StdControl as RadioControl;

    interface AMSend as RadioSend;
    interface Receive as RadioReceive;

    interface AMSend as UartSend[am_id_t id];
    interface Receive as UartReceive[am_id_t id];

    interface Leds;
  }
}

implementation
```

```
{
  bool uartBusy, radioBusy;
  serial_msg_t* out_payload;
  message_t out;

  void uartSendTask(message_t *msg,  uint8_t len);
  void radioSendTask(message_t *msg,  uint8_t len);

  event void Boot.booted() {
    out_payload = (serial_msg_t*)out.data;
    uartBusy = FALSE;
    radioBusy = FALSE;

    call RadioControl.start();
    call SerialControl.start();
  }

   event  void  SerialControl.startDone(error_t  error)
{}
  event void SerialControl.stopDone(error_t error) {}

   message_t*  ONE  receive(message_t*  ONE  msg,  void*
payload, uint8_t len);

    event  message_t  *RadioReceive.receive(message_t
*msg, void *payload, uint8_t len) {
    return receive(msg, payload, len);
  }

   message_t*  receive(message_t  *msg,  void  *payload,
uint8_t len) {
    if(uartBusy == FALSE) {
      atomic {
        uartBusy = TRUE;
     uartSendTask(msg,len);
      }
    }
    return msg;
  }

  void uartSendTask(message_t *msg,  uint8_t len) {
```

```
    atomic {
      call UartSend.send[0](0xFFFF, msg, len);
    }
  }

  event void UartSend.sendDone[am_id_t id](message_t*
msg, error_t error) {
    uartBusy = FALSE;
    call Leds.led2Toggle();
  }

   event message_t *UartReceive.receive[am_id_t id]
(message_t *msg, void *payload, uint8_t len) {
    uint8_t i;
    serial_msg_t* in = (serial_msg_t*)payload;
    for(i=0;i<14;i++) {
      out_payload->value[i] = in->value[i];
    }

    atomic {
      if(radioBusy == FALSE) {
        radioBusy = TRUE;
        radioSendTask(&out, len);
      }
    }

    return msg;
  }
  void radioSendTask(message_t *msg,  uint8_t len) {
    atomic {
      call RadioSend.send(0xFFFF, msg, len);
    }
  }

    event  void  RadioSend.sendDone(message_t*  msg,
error_t error) {
    radioBusy = FALSE;
  }
}
```

Figura 41 Módulo da aplicação teste.

```
#ifndef SERIALMSGS_H
#define SERIALMSGS_H

#include "message.h"

enum {
  AM_SERIAL_MSG         = 6, //0x92
};

typedef struct serial_msg {
  char value[16];
} serial_msg_t;

#endif //SERIALMSGS_H
```

Figura 42 Cabeçalho da aplicação teste.

Agora cada nó sensor é ligado a um computador. Como nas Figuras 43 e 44.

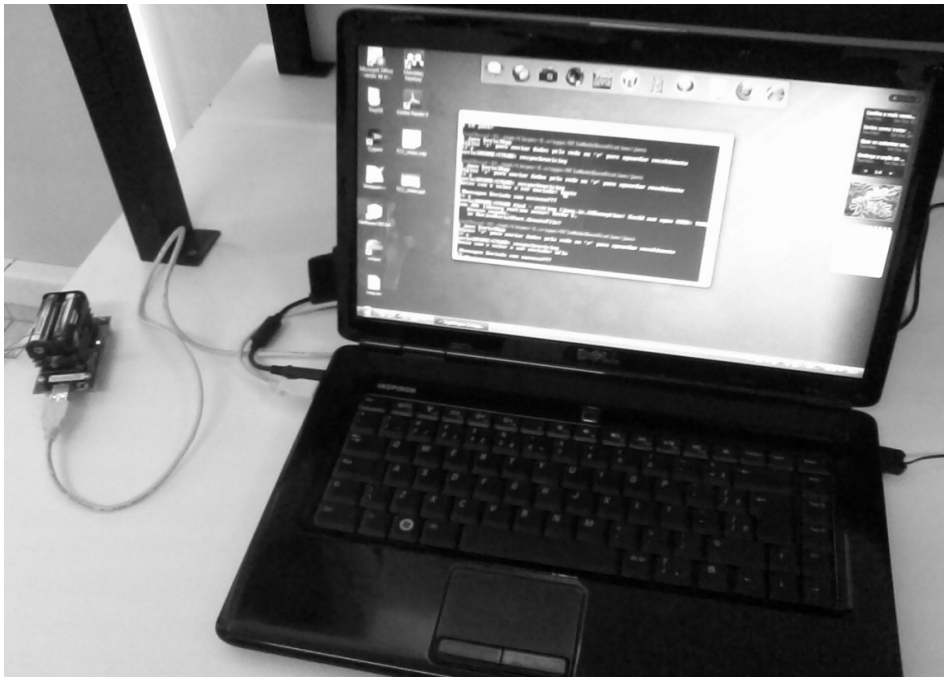

Figura 43 Nó Sensor ligado a um dos computadores.

Figura 44 Segundo Nó Sensor ligado ao outro computador.

Com os nós sensores devidamente preparados e conectados aos respectivos computadores, é preciso montar o programa auxiliar em Java que é a interface de saída para o teste.

As Figuras 45. Contem Makefile para compilar o programa java, os códigos do programa estão no APÊNDICE A , são os arquivos `main.java` e `mensage.java`:

```
GEN=SerialMsg.java

all: SerialApp.jar

SerialApp.jar: SerialApp.class
	jar cf $@ *.class
```

```
SerialMsg.java: ../SerialMsgs.h
	mig -target=null \
-java-classname=SerialMsg  java  ../SerialMsgs.h  serial_msg
-o $@

SerialApp.class: $(wildcard *.java) $(GEN)
	javac *.java

clean:
	rm -f *.class $(GEN)

veryclean: clean
	rm -f SerialApp.jar
```

Figura 45 Makefile.

Com o programa Java devidamente construído o teste será realizado. Nos dois computadores inicie o programa Java, como na Figura 46 usando o comando "`java SerialApp`".

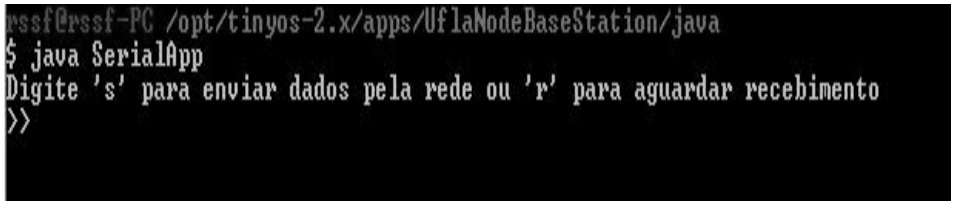

Figura 46 Programa Java executando.

Em um dos computadores entre com o comanda "`r`" para entra no modo de escuta como na Figura 47.

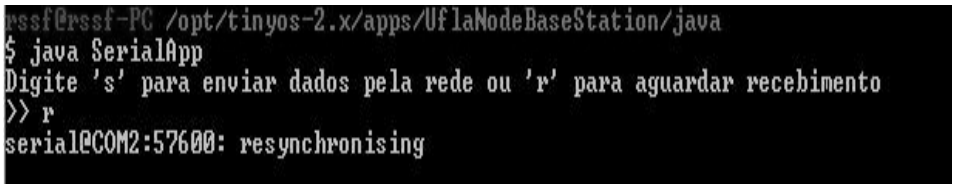

Figura 47 Modo de escuta.

O outro computador fará o envio da mensagem, usando o comando “s” envie a palavra ufla como na Figura 48.

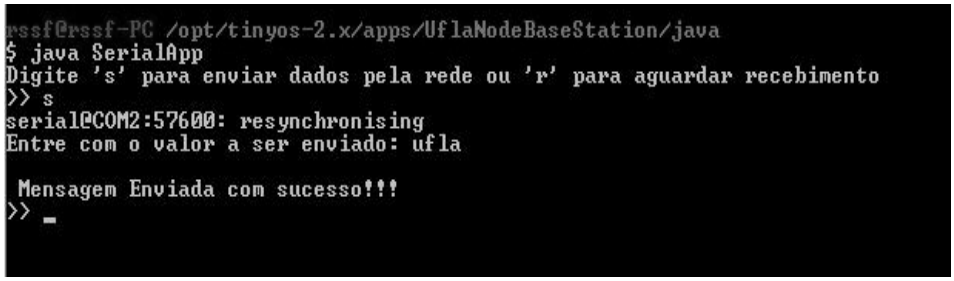

Figura 48 Envio da palavra ufla.

O computador que esta escutando recebera uma mensagem em 24 números no formato hexadecimal, como mostrado na Figura 49.

Figura 49 Mensagem recebida.

A mensagem recebida tem 24 números no formato hexadecimal, os 10 primeiros são o cabeçalho e os demais a palavra enviada.

Como a palavras envida, ufla, tem 4 caracteres o campo para a palavra tem 14 o restante é preenchido com zeros.

Então a palavra é formada como mostrado no Quadro 4:

Quadro 4 Composição da palavra recebida.

Hexadecimal	**Caractere**
75	u
66	f
6C	l
61	a

Assim o teste é realizado e fica provado que implementações baseadas no guia são corretas e compatíveis com o sistema *TinyOS*.

5 CONCLUSÕES E PROPOSTAS DE CONTINUIDADE

Neste trabalho foi apresentada uma forma organizada e bem definida de como se deve proceder para criar novas plataformas e componentes para o *TinyOS*.

Foram utilizado conhecimentos de diversas áreas da ciência da computação como sistemas digitais, compiladores, sistemas operacionais, linguagem de programação, redes de computadores entre outras.

Foram mostrandas algumas características de programação para sistemas embarcados e de microcontroladores.

E principalmente mostrou como são construídos os equipamentos para controle e comunicação digital para uma Rede de Sensores Sem Fio.

Entender como funciona o projeto e implementação de uma nova plataforma para o *TinyOS* ajuda a esclarecer muitas dúvidas comuns e prepara o campo para aprender maiores detalhes do sistema. Como começar a desenvolver não só aplicações, mas novos componentes internos que um dia poderão fazer parte do núcleo do sistema.

Para o aprendizado uma abordagem *botton-up* como a que foi mostrada aqui é mais gratificante e permite um aprendizado mais rápido.

Ao contrário para um desenvolvimento com fins mais profissionais como novos módulos reais é mais indicado uma abordagem *top-down* para garantir a abstração, visto que quem pretende desenvolver algo mais robusto já tem uma visão mais ampla do *TinyOS* e de como ele está organizado.

Como trabalhos futuros é proposto:

a) Estudar os modelos de hardware existentes para propor novos modelos;

b) Estudar os componentes presentes no mercado e avaliar custo-benefício;

c) Desenvolver uma nova plataforma de baixo custo;

d) Construir um Nó Sensor com o microcontrolador Atmega64 que é mais fácil de manusear;

e) Propor melhorias para os componentes já existentes do *TinyOS*.

REFERÊNCIAS

ATMEL CORPORATION - INDUSTRY LEADER IN THE DESIGN AND MANUFACTURE. **8-bit Microcontroller ARV with 128K Bytes In-System Programmable Flash.** Disponível em: <www.atmel.com>. Acesso em: 24/05/2010.

CROSSBOW TECHNOLOGY, INC. **MPR/MIB Mote User Manual**: Revision A. 52p. Disponível em: <www.xbow.com>. Acesso em: 15/03/2010.

LI, Y.; WU, W. **Wireless Sensor Networks and Applications.** Springer, 2008.

SHOREY, R. **Mobile, Wireless, And Sensor Networks Technology, Applications, And Future Directions.** Wiley, 2006.

SOHRABY, K.; MINOLI, D. **Wireless Sensor Networks Technology, Protocols, And Applications.** Wiley, 2007.

STOJMENOVIC, I. **Handbook Of Sensor Networks Algorithms And Architectures.** Wiley, 2005.

SWAMI, A.; ZHAO, Q. Wireless Sensor Networks Signal Processing and Communications Perspectives. Wiley, 2007.

TATO. **RF-24G_datasheet.** Disponível em: <www.tato.ind.br>. Acesso em: 24/05/2010.

TINYOS COMMUNITY FORUM. **Documentation.** Disponível em: <www.tinyos.net>. Acesso em: 24/05/2010.

APÉNDICE A – Códigos

Arquivo trf24g.h

```
/*************************************************************/
// Ufla - 17/04/2010
// Fellipe Augusto Ugliara
/*************************************************************/
#ifndef TRF24G_H
#define TRF24G_H
/*************************************************************/
// Defines
/*************************************************************/
// Default configuration settings
// Default data length (bits) -- 16 bytes

#define TRF24G_DEFAULT_DATA2_W 128
#define TRF24G_DEFAULT_DATA1_W 128

// Default address length (bits) -- use all 5 bytes possible
//  to decrease chance of bad packets

#define TRF24G_DEFAULT_ADDR_W 40

// Default CRC length (0 = 8bit, 1 = 16bit)

#define TRF24G_DEFAULT_CRC_L 1

// Default CRC enable - yes

#define TRF24G_DEFAULT_CRC_EN 1

// Default not to enable receive on #2?

#define TRF24G_DEFAULT_RX2_EN 0

// Communication mode (1 = "shockburst").  Currently this package
only
//  supports shockburst communications.

#define TRF24G_DEFAULT_CM 1

// Default data rate (for shockburst) - 1 = 1Mb, 0 = 250K

#define TRF24G_DEFAULT_RFDR_SB 1

// Crystal freq (3 = 16 Mhz; that's how the trf24g comes)

#define TRF24G_DEFAULT_XO_F 3

// Default power level 0 = -20db, 1 = -10, 2 = -5, 3 = 0
```

```
#define TRF24G_DEFAULT_RF_PWR 1

// Communications channel (0-125)

#define TRF24G_DEFAULT_RF_CH 0x55

#define TRF24G_DEFAULT_RXEN 1

// Defines to help accessing bits in configuration

#define TRF24G_ADDR_W_SHIFT 2
#define TRF24G_ADDR_W_MASK 0xfc

#define TRF24G_CRC_L_SHIFT 1
#define TRF24G_CRC_L_MASK 0x02

#define TRF24G_CRC_EN_SHIFT 0
#define TRF24G_CRC_EN_MASK 0x01

#define TRF24G_RX2_EN_SHIFT 7
#define TRF24G_RX2_EN_MASK 0x80

#define TRF24G_CM_SHIFT 6
#define TRF24G_CM_MASK 0x40

#define TRF24G_RFDR_SB_SHIFT 5
#define TRF24G_RFDR_SB_MASK 0x20

#define TRF24G_XO_F_SHIFT 2
#define TRF24G_XO_F_MASK 0x1c

#define TRF24G_RF_PWR_SHIFT 0
#define TRF24G_RF_PWR_MASK 0x3

#define TRF24G_RF_CH_SHIFT 1
#define TRF24G_RF_CH_MASK 0xfe

#define TRF24G_RXEN_SHIFT 0
#define TRF24G_RXEN_MASK 1
/*************************************************************/
// Typedefs
/*************************************************************/
typedef struct {
       uint8_t data2_w;
       uint8_t data1_w;
       uint8_t addr2[5];
       uint8_t addr1[5];

       uint8_t addr_w_crc;  // addr_w:6 crc_l:1 crc_en:1 (hi-lo)
       uint8_t misc;        //  rx2_en:1 cm:1 rfdr_sb:1 xo_f:3
```

```
rf_pwr:2 (hi-lo)
      uint8_t rf_ch_rxen;  // rf_ch:7 rxen:1 (hi-lo)
} __attribute__ ((packed)) trf24g_config_t;

extern trf24g_config_t trf24g_config;
/*************************************************************/
// Basic spin delay functions
/*************************************************************/
# define delay_500ns() {__asm__ __volatile__ ("     nop\r\n
      nop\r\n      nop\r\n      nop\r\n"); }

#define MCU_CLK 7370000

# define LOOPS_PER_MS ((MCU_CLK/1000)/4)

# if (MCU_CLK >= 4000000)

# define LOOPS_PER_US ((MCU_CLK/1000000)/4)

// Spin for about (us) microseconds
// The inner loop takes 4 cycles per iteration

static inline void us_spin(unsigned short us)
{
      if (us) {
            __asm__ __volatile__ (
                  "outer_%=: "
                  "      ldi r26, %2     \n"
                  "inner_%=: "
                  "      dec r26         \n"
                  "      brne inner_%=   \n"
                  "      sbiw %0, 1      \n"
                  "      brne outer_%=   \n"
                  : "=w" (us)
                  : "w" (us), "i" (LOOPS_PER_US)
                  : "r26"
            );
      }
}

# else // MCU_CLK < 4000000

# define LOOPS_PER_8US (MCU_CLK / 500000)

/* Spin for about (us) microseconds (slow CPU clock version)
 *    -- This actually does it in 8 us increments
 * the inner loop takes 4 cycles per iteration
 *
 * The us divide will actually add a few cycles extra...
 */
```

```
static inline void us_spin(unsigned short us) {
	if (us/=8) {
		__asm__ __volatile__ (
			"outer_%=: "
			"	ldi r26, %2	\n"
			"inner_%=: "
			"	dec r26		\n"
			"	brne inner_%=	\n"
			"	sbiw %0, 1	\n"
			"	brne outer_%=	\n"
			: "=w" (us)
			: "w" (us), "i" (LOOPS_PER_8US)
			: "r26"
		);
	}
}

# endif // if (MCU_CLK >= 4000000)
/*
 * Spin for about (ms) milliseconds
 *
 */
static inline void ms_spin(unsigned short ms) {
	if (ms) {
		__asm__ __volatile__ (
			"outer_%=:"
			"	ldi r26, %3	\n"
			"	ldi r27, %2	\n"
			"inner_%=:"
			"	sbiw r26, 1	\n"
			"	brne inner_%=	\n"
			"	sbiw %0, 1	\n"
			"	brne outer_%=	\n"
			: "=w" (ms)
			: "w" (ms),
			  "i" (LOOPS_PER_MS >> 8),
			  "i" (0xff & LOOPS_PER_MS)
			: "r26", "r27"
		);
	}
}
#endif
/*************************************************************/
// Fim do Arquivo trf24g.h
/*************************************************************/
```

Arquivo TRF24GActiveMessageC.nc

```
/*****************************************************************/
// Ufla - 01/05/2010
// Fellipe Augusto Ugliara
/*****************************************************************/
configuration TRF24GActiveMessageC {
  provides {
    interface StdControl;
    interface AMSend;
    interface Receive;
  }
}
implementation {

  components TRF24GModuleC,
             TRF24GActiveMessageP;

  StdControl = TRF24GModuleC.StdControl;
  AMSend = TRF24GActiveMessageP.AMSend;
  Receive = TRF24GActiveMessageP.Receive;

  TRF24GActiveMessageP.TRF24GModule ->
TRF24GModuleC.TRF24GModule;
}
/*****************************************************************/
// Fim do Arquivo TRF24GActiveMessageC.nc
/*****************************************************************/
```

Arquivo TRF24GActiveMessageP.nc

```
/*****************************************************************/
// Ufla - 01/05/2010
// Fellipe Augusto Ugliara
/*****************************************************************/
#define SIZE_ADDR 2*sizeof(uint8_t)
#define SIZE_DATA 14*sizeof(uint8_t)

module TRF24GActiveMessageP {
  provides {
    interface AMSend;
    interface Receive;
  }

  uses {
    interface TRF24GModule;
  }
}
implementation {
/*****************************************************************/
// Variables
/*****************************************************************/
  message_t temp_send;
  message_t temp_recv;
/*****************************************************************/
// Comands
/*****************************************************************/
  command error_t AMSend.send(am_addr_t addr, message_t* msg,
uint8_t len) {
    char msg_array[16];
    if(len > SIZE_ADDR + SIZE_DATA) {
      return FAIL;
    }
    else {
      memcpy(msg_array, &addr, sizeof(uint16_t));
      memcpy(msg_array + sizeof(uint16_t), msg->data, len);

      temp_send = *msg;
      call TRF24GModule.trf24g_send(msg_array);
      return SUCCESS;
    }
  }
//-------------------------------------------------------
  command error_t AMSend.cancel(message_t* msg) {
  }
//-------------------------------------------------------
  command uint8_t AMSend.maxPayloadLength() {
    return SIZE_ADDR + SIZE_DATA;
  }
//-------------------------------------------------------
```

```
  command void* AMSend.getPayload(message_t* msg, uint8_t len) {
  }
/************************************************************/
// Events
/************************************************************/
  event void TRF24GModule.trf24g_recv_done(char *recv) {
    memcpy(temp_recv.data, recv, SIZE_ADDR + SIZE_DATA);
    signal Receive.receive(&temp_recv, temp_recv.data, SIZE_ADDR
+ SIZE_DATA);
  }
//-------------------------------------------------------
  event void TRF24GModule.trf24g_send_done() {
    signal AMSend.sendDone(&temp_send, SUCCESS);
  }
/************************************************************/
// End
/************************************************************/
}
/************************************************************/
// Fim do Arquivo TRF24GActiveMessageP.nc
/************************************************************/
```

Arquivo TRF24GModule.nc

```
/*****************************************************************/
// Ufla - 17/04/2010
// Fellipe Augusto Ugliara
/*****************************************************************/
interface TRF24GModule {
  command void trf24g_send(char *data);

  event void trf24g_recv_done(char *recv);
  event void trf24g_send_done();
}
/*****************************************************************/
// Fim do Arquivo TRF24GModule.nc
/*****************************************************************/
```

Arquivo TRF24GModuleC.nc

```
/*****************************************************************/
// Ufla - 17/04/2010
// Fellipe Augusto Ugliara
/*****************************************************************/
configuration TRF24GModuleC {
  provides {
    interface StdControl;
    interface TRF24GModule;
  }
}

implementation {
  components HplTRF24GPinsC as Pins,
             TRF24GInterruptsC as InterruptsC,
             LedsC,
             TRF24GModuleP;

  TRF24GModule = TRF24GModuleP.TRF24GModule;
  StdControl   = TRF24GModuleP.StdControl;

  TRF24GModuleP.DATA  ->  Pins.DATA;
  TRF24GModuleP.CLK1  ->  Pins.CLK1;
  TRF24GModuleP.CS    ->  Pins.CS;
  TRF24GModuleP.CE    ->  Pins.CE;
  TRF24GModuleP.VCC   ->  Pins.VCC;

  TRF24GModuleP.DR1   ->  InterruptsC;
  TRF24GModuleP.Leds  ->  LedsC;
}
/*****************************************************************/
// Fim do Arquivo TRF24GModuleC.nc
/*****************************************************************/
```

Arquivo TRF24GModuleP.nc

```
/*****************************************************************/
// Ufla - 17/04/2010
// Fellipe Augusto Ugliara
/*****************************************************************/
#include "trf24g.h"

module TRF24GModuleP {
  provides {
    interface StdControl;
    interface TRF24GModule;
  }
  uses {
    interface GeneralIO as CE;
    interface GeneralIO as CS;
    interface GeneralIO as VCC;
    interface GeneralIO as DATA;
    interface GeneralIO as CLK1;

    interface GpioInterrupt as DR1;
    interface Leds;
  }
}

implementation {
/*****************************************************************/
// Variables
/*****************************************************************/
  trf24g_config_t trf24g_config;
  char trf24g_recv_payload_msg[16];
/*****************************************************************/
// Prototypes
/*****************************************************************/
  void TRF24G_MODE_CONFIG();
  void TRF24G_END_MODE_CONFIG();
  void TRF24G_ACTIVATE();
  void TRF24G_DEACTIVATE();

  void TRF24G_SEND_BIT(uint8_t bit);
  uint8_t TRF24G_RECV_BIT();

  void TRF24G_INIT();
  void TRF24G_LOAD_CONFIG();
  void TRF24G_RECV(char *packet);
  void TRF24G_TX_WAIT();
  void TRF24G_RX_WAIT();

  void TRF24G_RECONFIG(uint8_t channel, uint8_t rxen);
/*****************************************************************/
// Comands
```

```
/*************************************************************/
  command error_t StdControl.start() {
    atomic {
      call CE.makeOutput();
      call CS.makeOutput();
      call CLK1.makeOutput();
      call DATA.makeInput();
      call VCC.makeOutput();

      us_spin(10);
      call VCC.set();
      us_spin(10);

      TRF24G_INIT();
    }
    return SUCCESS;
  }
//-------------------------------------------------------
  command error_t StdControl.stop() {
    TRF24G_DEACTIVATE();
    call DR1.disable();
    call VCC.clr();
    return SUCCESS;
  }
//-------------------------------------------------------
  command void TRF24GModule.trf24g_send(char *data) {
    uint8_t i;
    uint8_t j;

    char address[5] = {00, 00, 00, 0xaa, 0x55};

    //novo
    call DR1.disable();
    TRF24G_RECONFIG(0x55,0);
    //novo

    TRF24G_ACTIVATE();
      us_spin(10);

      // Clock out address
      for (i = 0; i < ((trf24g_config.addr_w_crc &
TRF24G_ADDR_W_MASK) >> TRF24G_ADDR_W_SHIFT) / 8; i++) {
        for (j = 0x80; j; j>>=1)
         TRF24G_SEND_BIT(address[i] & j);
      }
      // Clock out data
      for (i = 0; i < trf24g_config.data1_w / 8; i++) {
        for (j = 0x80; j; j>>=1)
         TRF24G_SEND_BIT(data[i] & j);
      }
```

```
    TRF24G_DEACTIVATE();

    //espera a mensagem ser enviada
    TRF24G_TX_WAIT();
    // Dispara sinal para o evento send Done
    signal TRF24GModule.trf24g_send_done();

    //novo
    TRF24G_RECONFIG(0x55,1);
    TRF24G_RX_WAIT();
    //novo
  }
/*****************************************************************/
// Events
/*****************************************************************/
  async event void DR1.fired() {
    TRF24G_RECV( trf24g_recv_payload_msg );
  }
/*****************************************************************/
// Functions
/*****************************************************************/
  void TRF24G_MODE_CONFIG() {

    call DATA.makeOutput();

    call CE.clr();
    us_spin(10);

    call CS.set();
    ms_spin(3);
  }
//-------------------------------------------------------
  void TRF24G_END_MODE_CONFIG() {
    call DATA.makeInput();
    us_spin(10);

    call CE.clr();
    call CS.clr();
    ms_spin(3);
  }
//-------------------------------------------------------
  void TRF24G_ACTIVATE() {
    call CS.clr();
    us_spin(10);
    call CE.set();
    ms_spin(3);
  }
//-------------------------------------------------------
  void TRF24G_DEACTIVATE() {
    call CE.clr();
    call CS.clr();
```

```
  }
//-------------------------------------------------------
  void TRF24G_SEND_BIT(uint8_t bit) {

    bit ? call DATA.set() : call DATA.clr();
    //controle visual
          //if(bit != 0)
          //{
          //  call Leds.led1Toggle();
          //  ms_spin(100);
          //}else{
          //  call Leds.led0Toggle();
          //  ms_spin(100);
          //}
    //controle visual
    us_spin(500);
    call CLK1.set();
    us_spin(500);
    call CLK1.clr();
  }
//-------------------------------------------------------
  uint8_t TRF24G_RECV_BIT() {
    uint8_t bit;

    us_spin(500);
    call CLK1.set();
    us_spin(500);
    bit = call DATA.get();
    //controle visual
          //if(bit != 0)
          //{
          //  call Leds.led1Toggle();
          //  ms_spin(100);
          //}else{
          //  call Leds.led0Toggle();
          //  ms_spin(100);
          //}
    //controle visual
    us_spin(500);
    call CLK1.clr();

    return bit;
  }
//-------------------------------------------------------
  void TRF24G_INIT() {
    char myaddress[5] = {00, 00, 00, 0xaa, 0x55};
    TRF24G_MODE_CONFIG();
      // Monta palavras de config inicial
      memcpy(trf24g_config.addr1, myaddress,
sizeof(trf24g_config.addr1));
      trf24g_config.data2_w = TRF24G_DEFAULT_DATA2_W;
```

```
      trf24g_config.data1_w = TRF24G_DEFAULT_DATA1_W;
      trf24g_config.addr_w_crc = TRF24G_DEFAULT_ADDR_W <<
TRF24G_ADDR_W_SHIFT
                                  | TRF24G_DEFAULT_CRC_L <<
TRF24G_CRC_L_SHIFT
                               | TRF24G_DEFAULT_CRC_EN <<
TRF24G_CRC_EN_SHIFT;
      trf24g_config.misc = TRF24G_DEFAULT_RX2_EN <<
TRF24G_RX2_EN_SHIFT
                             | TRF24G_DEFAULT_CM << TRF24G_CM_SHIFT
                           | TRF24G_DEFAULT_RFDR_SB <<
TRF24G_RFDR_SB_SHIFT
                             | TRF24G_DEFAULT_XO_F <<
TRF24G_XO_F_SHIFT
                           | TRF24G_DEFAULT_RF_PWR <<
TRF24G_RF_PWR_SHIFT;
      trf24g_config.rf_ch_rxen = TRF24G_DEFAULT_RF_CH <<
TRF24G_RF_CH_SHIFT
                                | TRF24G_DEFAULT_RXEN;
      // Fim da Montagem da palavras de config inicial
      TRF24G_LOAD_CONFIG();
    TRF24G_END_MODE_CONFIG();
    //novo
    TRF24G_RECONFIG(0x55,1);
    TRF24G_RX_WAIT();
    //novo
  }
//-------------------------------------------------------
  void TRF24G_LOAD_CONFIG() {
    uint8_t i;
    uint8_t *cbyte = (uint8_t *)&trf24g_config;
    us_spin(10);
    while (cbyte < (((uint8_t *)&trf24g_config) +
sizeof(trf24g_config))) {
      for (i = 0x80; i; i>>=1) {
        TRF24G_SEND_BIT(i & *cbyte);
      }

      cbyte++;
    }
  }
//-------------------------------------------------------
  void TRF24G_RECV(char *packet) {
    uint8_t i;
    uint8_t j;
    us_spin(10);
    // Clock in the bits...
    for (i = 0; i < TRF24G_DEFAULT_DATA1_W / 8; i++) {
      packet[i] = 0;
      for (j = 0x80; j; j>>=1)
        packet[i] |= (TRF24G_RECV_BIT() ? j:0);
```

```
    }
    TRF24G_DEACTIVATE();
    signal TRF24GModule.trf24g_recv_done( packet );
    //novo
    TRF24G_RX_WAIT();
    //novo
  }
//-----------------------------------------------------
  void TRF24G_TX_WAIT() {
    us_spin(195 + 8
              + ((trf24g_config.addr_w_crc & TRF24G_ADDR_W_MASK)
>> TRF24G_ADDR_W_SHIFT)
              + trf24g_config.data1_w
              + 16);
    ms_spin(1000);
  }
//-----------------------------------------------------
  void TRF24G_RX_WAIT() {
    call DR1.enableRisingEdge();
    TRF24G_ACTIVATE();
  }
//-----------------------------------------------------
  void TRF24G_RECONFIG(uint8_t channel, uint8_t rxen) {
    int i;
    TRF24G_MODE_CONFIG();
    us_spin(10);
      // Set RX enable bit
      if (rxen) {
        trf24g_config.rf_ch_rxen = (channel <<
TRF24G_RF_CH_SHIFT) | 1;
      } else {
        trf24g_config.rf_ch_rxen = (channel <<
TRF24G_RF_CH_SHIFT) | 0;
      }
      // Send channel/RX enable bit
      for (i = 0x80; i; i>>=1)
        TRF24G_SEND_BIT(trf24g_config.rf_ch_rxen & i);

    TRF24G_END_MODE_CONFIG();

    if (!rxen) {
      call DATA.makeOutput();
    }
  }
/*************************************************************/
// End
/*************************************************************/
}
/*************************************************************/
// Fim do Arquivo TRF24GModuleP.nc
/*************************************************************/
```

Arquivo HplTRF24GPinsC.nc

```
/*****************************************************************/
// Ufla - 01/05/2010
// Fellipe Augusto Ugliara
/*****************************************************************/
configuration HplTRF24GPinsC {
  provides {
    interface GeneralIO as CE;
    interface GeneralIO as CS;
    interface GeneralIO as CLK1;
    interface GeneralIO as DATA;
    interface GeneralIO as VCC;
  }
}

implementation {

  components HplAtm128GeneralIOC as IO;//modificar ainda

  //mudar pinos para as portas corretas
  CE    = IO.PortC5;
  CS    = IO.PortC6;
  CLK1  = IO.PortC2;
  DATA  = IO.PortC4;
  VCC   = IO.PortC0;
}
/*****************************************************************/
// Fim do Arquivo HplTRF24GPinsC.nc
/*****************************************************************/
```

Arquivo TRF24GInterruptsC.nc

```
/*****************************************************************/
// Ufla - 01/05/2010
// Fellipe Augusto Ugliara
/*****************************************************************/
configuration TRF24GInterruptsC {

  provides interface GpioInterrupt as InterruptMsg;

}

implementation {

  components new Atm128GpioInterruptC() as Interrupt128;
  components HplAtm128InterruptC as Interrupts;
  InterruptMsg = Interrupt128;
  Interrupt128.Atm128Interrupt -> Interrupts.Int1;//no mda100cb é
o pino C9

}
/*****************************************************************/
// Fim do Arquivo TRF24GInterruptsC.nc
/*****************************************************************/
```

Arquivo ActiveMessageC.nc

```
/*****************************************************************/
// Ufla - 01/05/2010
// Fellipe Augusto Ugliara
/*****************************************************************/
configuration ActiveMessageC {
  provides {
    interface StdControl;
    interface AMSend;
    interface Receive;
  }
}
implementation {
  components TRF24GActiveMessageC;

  StdControl = TRF24GActiveMessageC.StdControl;
  AMSend = TRF24GActiveMessageC.AMSend;
  Receive = TRF24GActiveMessageC.Receive;
}
/*****************************************************************/
// Fim do Arquivo ActiveMessageC.nc
/*****************************************************************/
```

Arquivo Mensage.java do Programa Java

```
public class SerialMsg extends net.tinyos.message.Message {

    /** The default size of this message type in bytes. */
    public static final int DEFAULT_MESSAGE_SIZE = 2;

    /** The Active Message type associated with this message. */
    public static final int AM_TYPE = 6;

    /** Create a new SerialMsg of size 2. */
    public SerialMsg() {
        super(DEFAULT_MESSAGE_SIZE);
        amTypeSet(AM_TYPE);
    }

    /** Create a new SerialMsg of the given data_length. */
    public SerialMsg(int data_length) {
        super(data_length);
        amTypeSet(AM_TYPE);
    }

    /**
     * Create a new SerialMsg with the given data_length
     * and base offset.
     */
    public SerialMsg(int data_length, int base_offset) {
        super(data_length, base_offset);
        amTypeSet(AM_TYPE);
    }

    /**
     * Create a new SerialMsg using the given byte array
     * as backing store.
     */
    public SerialMsg(byte[] data) {
        super(data);
        amTypeSet(AM_TYPE);
    }

    /**
     * Create a new SerialMsg using the given byte array
     * as backing store, with the given base offset.
     */
    public SerialMsg(byte[] data, int base_offset) {
        super(data, base_offset);
        amTypeSet(AM_TYPE);
    }

    /**
     * Create a new SerialMsg using the given byte array
```

```
       * as backing store, with the given base offset and data
length.
     */
         public  SerialMsg(byte[]  data,  int  base_offset,  int
data_length) {
        super(data, base_offset, data_length);
        amTypeSet(AM_TYPE);
    }

    /**
     * Create a new SerialMsg embedded in the given message
     * at the given base offset.
     */
         public  SerialMsg(net.tinyos.message.Message  msg,  int
base_offset) {
        super(msg, base_offset, DEFAULT_MESSAGE_SIZE);
        amTypeSet(AM_TYPE);
    }

    /**
     * Create a new SerialMsg embedded in the given message
     * at the given base offset and length.
     */
         public  SerialMsg(net.tinyos.message.Message  msg,  int
base_offset, int data_length) {
        super(msg, base_offset, data_length);
        amTypeSet(AM_TYPE);
    }

    /**
     /* Return a String representation of this message. Includes
the
     * message type name and the non-indexed field values.
     */
    public String toString() {
      String s = "Mensagem Recebida\n";
      try {
        s += "Valor: 0x"+Long.toHexString(get_value())+"]\n\n";
       } catch (ArrayIndexOutOfBoundsException aioobe) { /* Skip
field */ }
      return s;
    }

    /**
     * Return whether the field 'value' is signed (false).
     */
    public static boolean isSigned_value() {
        return false;
    }

    /**
```

```
     * Return whether the field 'value' is an array (false).
     */
    public static boolean isArray_value() {
        return false;
    }

    /**
     * Return the offset (in bytes) of the field 'value'
     */
    public static int offset_value() {
        return (0 / 8);
    }

    /**
     * Return the offset (in bits) of the field 'value'
     */
    public static int offsetBits_value() {
        return 0;
    }

    /**
     * Return the value (as a int) of the field 'value'
     */
    public int get_value() {
        return (int)getUIntBEElement(offsetBits_value(), 16);
    }

    /**
     * Set the value of the field 'value'
     */
    public void set_value(byte[] value) {
        dataSet(value);
    }

    /**
     * Return the size, in bytes, of the field 'value'
     */
    public static int size_value() {
        return (16 / 8);
    }

    /**
     * Return the size, in bits, of the field 'value'
     */
    public static int sizeBits_value() {
        return 16;
    }
}
```

Arquivo do Main.java do Programa Java

```
import net.tinyos.packet.*;
import net.tinyos.message.*;
import net.tinyos.util.*;
import java.io.*;
/**
*/

public class SerialApp implements MessageListener
{
  MoteIF mote;
  static byte msgout[];

  /* Main entry point */
  void run() {
    mote = new MoteIF(PrintStreamMessenger.err);
       mote.registerListener(new SerialMsg(), this);
  }

   //Imprime mensagem recebida na tela, caso esteja aguardando
recebimento
  synchronized public void messageReceived(int dest_addr, Message
msg) {
    if (msg instanceof SerialMsg) {
      System.out.print(msg.toString());
    }
  }

  //Envio via serial para o no conectado na porta USB
  synchronized public void enviar(byte[] value) {
    SerialMsg msg = new SerialMsg(msgout);
       msg.set_value(value);

    try {
      mote.send(MoteIF.TOS_BCAST_ADDR, msg);
         System.err.println("\n Mensagem Enviada com sucesso!!!");
    }
    catch (IOException e) {
      System.err.println("Cannot send message!!!");
    }
  }

  public static void main(String[] args) {
    SerialApp me = new SerialApp();
    msgout = new byte[14];

    PacketSource reader;


    InputStreamReader cin = new InputStreamReader(System.in);
```

```
    BufferedReader in = new BufferedReader(cin);
    String input1 = "";

     System.out.print("Digite 's' para enviar dados pela rede ou
'r' para aguardar recebimento\n");
    System.out.print(">> ");
       for(;;) {
         try {
              input1 = in.readLine();
              if(input1.equals("s")) {
                me.run();
                System.out.print("Entre com o valor a ser enviado:
");
                input1 = in.readLine();
                //me.enviar(Integer.parseInt(input1));
                me.enviar(input1.getBytes());
              }
              else if(input1.equals("r")){
                     reader = BuildSource.makePacketSource();
                     try {
                            reader.open(PrintStreamMessenger.err);
                            for(;;){
                                   byte[]          packet          =
reader.readPacket();
                                   Dump.printPacket(System.out,
packet);
                                   System.out.println();
                                   System.out.flush();
                            }
                     }
                     catch (IOException e) {
                            System.err.println("Error   on   "   +
reader.getName() + ": " + e);
                     }
              }
              else System.out.println("Entrada invalida!!!!: ");
              System.out.print(">> ");
         }
         catch (IOException e) {
              System.out.print("Erro na entrada dos dados!!");
         }
       }
  }
}
```