Relatório sobre o Sistema Operacional TinyOS e a Linguagem NesC

Marcus Vinícius de Sousa Lemos[1], Líliam Barroso Leal[1]

[1]Mestrado em Informática Aplicada – Universidade de Fortaleza (UNIFOR)
Av. Washington Soares, 1321, Edson Queiroz
CEP 60.811-905 Fortaleza-Ce, Brasil

{marvin, liliam}@edu.unifor.br

Abstract. *Wireless Sensor Networks, considered a part of ubiquitous computing, bring new challenges most of all due to the great difficulty of adapting existing protocols caused by the many restrictions attached to the WSN (eg, memory, processing capacity). These restrictions also affect significantly the development of operating systems. The main goal of this report is to present the TinyOS, an operating system specially designed for the WSN, and the programming language NesC, depicting its main feature addition to developing a simple application in order to demonstrate the programming model.*

Resumo. *Redes de Sensores sem fio, consideradas uma vertente da computação ubíqua, trazem novos desafios devido à grande dificuldade de adaptação dos protocolos existentes ocasionado pelas inúmeras restrições inerentes às RSSF (por exemplo, memória, capacidade de processamento). Essas restrições também influenciam bastante o desenvolvimento de sistemas operacionais. O objetivo deste relatório é apresentar o TinyOS, um Sistema Operacional especialmente projetado para as RSSF, e a linguagem de programação NesC, ilustrando suas principais característica além de desenvolver uma simples aplicação com o intuito de demonstrar o modelo de programação.*

1. Introdução

Considerada uma vertente da computação Ubíqua, as Redes de Sensores sem Fio (RSSF) são formadas por um grande número de pequenos *sensores inteligentes* e com o objetivo de detectar e transmitir alguma característica do meio-físico [Marluce et al, 2004]. Por sensores inteligentes referimos a um pequena placa de circuito integrado contendo um ou mais sensores com capacidade de processamento de sinais e comunicação de dados [Loureiro et al, 2004]. Outras denominações também são encontradas na literatura, como, por exemplo, nós sensores, nós, nodos, ou motes (um termo muito comum na literatura de língua inglesa). Dentre os vários nós existentes em uma rede de sensores, existe um, denominado nó sorvedouro (ou *sink-node*), com a função de agregar as informações da rede e transmitir para uma estação-base ou até mesmo para um gateway com a internet. A figura 1 ilustra um cenário bastante comum nas RSSF:

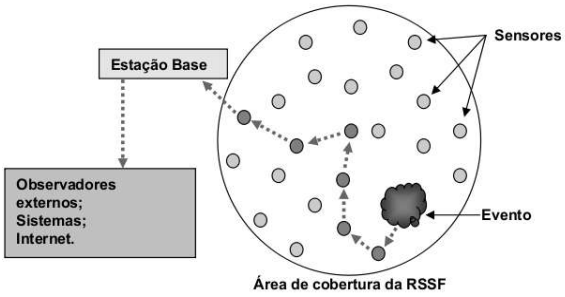

Figura 1 – Cenário típico de uma RSSF

Contudo, as RSSF impõe desafios que não são encontrados nas redes tradicionais. Principalmente no que diz respeito à energia. Diferentemente dos computadores de grande porte que são ligados a uma fonte de alimentação contínua, os nós sensores geralmente dispõe apenas de uma bateria com vida limitada, dando uma vida bastante curta ao nó. Muitas vezes esses mesmos nós são espalhados em regiões geográficas tornando difícil a reposição de uma possível bateria esgotada ou mesmo de um nó danificado.

Percebemos que essa nova variável torna praticamente impossível a reutilização de diversos algorítimos desenvolvidos para os sistemas computacionais tradicionais, uma vez que o projeto de qualquer solução para redes de sensores deve levar em consideração o consumo de energia. Conseqüentemente, o desenvolvimento de Sistemas Operacionais também deve ser pensado de forma a economizar o máximo de energia possível. O SO deve ser eficiente em termos de consumo de memória , processador e, claro, de energia. Deve ser ágil o suficiente para permitir que várias aplicações usem simultaneamente os recursos de comunicação.

O objetivo deste relatório é apresentar o TinyOS, um sistema operacional bastante utilizado em RSSF e a Linguagem de programação NesC, criada para facilitar o desenvolvimento de aplicações para o TinyOS. Além disso, pretendemos demonstrar a construção de uma aplicação simples com o intuito de ilustrar o modelo de programação do NesC.

Escrever software robusto e sólido não é uma tarefa fácil, e quando o cenário muda para RSSF a tarefa torna-se mais difícil ainda. Recursos limitados (como 4KB de RAM) e pouca disponibilidade de energia tem levado os desenvolvedores a escreverem versões de softwares específicas para cada aplicação. Uma conseqüência bastante direta desse fato é a dificuldade no uso de padrões de projeto com o objeto de desenvolver código reusável. Em [Gay et al, 2007] vimos como os projetista do TinyOS trataram essas dificuldades, ilustrando os principais padrões de projeto que influenciaram o desenvolvimento do TinyOS.

Em [Gay et al, 2007] temos ilustrado os principais desafios na construção de aplicações para redes de sensores:

- Robustez: uma vez instalado no ambiente, a rede de sensores deve ser executada sem intervenção humana por meses ou anos.

- Pouca disponibilidade de recursos: Nós sensores possuem memória RAM e uma bateria com vida útil limitada.
- Diversas implementações de serviços: as aplicações deveriam ser capazes de escolher entre múltiplas implementações de, por exemplo, roteamento *multihop*.
- Evolução do Hardware: os *hardwares* dos nós estão em constante evolução; aplicações e muitos serviços de sistemas devem ser portáveis entre gerações de *hardwares*.
- Adaptabilidade aos requerimentos das aplicações: As aplicações possuem requerimentos muito diferentes em termos de tempo de vida, comunicação, sensibilidade, etc.

O NesC [Gay et al, 2003], a linguagem de implementação do TinyOS, foi desenvolvido tendo esses desafios como prioridade. A principal característica do NesC é a visão holística do sistema. As aplicações dos nós sensores são bastante dependentes do hardware, e cada nó roda apenas uma aplicação por vez. [Gay et al, 2003].

NesC é uma linguagem orientada a componentes com um modelo de execução baseado em eventos. Os componentes NesC possuem uma certa familiaridade com os objetos de uma linguagem orientadas a objetos no sentido em que ambos encapsulam estado e interagem entre-si através de interfaces bem definidas [Levis, 2006]. Contundo, há algumas diferenças bastante significativas. NesC não possui certas características encontradas tipicamente em linguagens orientadas a objetos tais como herança, alocação dinâmica e *Dynamic Dispatch*. Isto se deve basicamente ao fato de que o conjunto de componentes e suas interações são determinadas em tempo de compilação ao invés de tempo de execução. Esse modelo de alocação estática é bastante interessante para dispositivos computacionais com poucos recursos disponíveis pois evita o desperdício de recursos alocados e não usados ou mesmo falhas que porventura venha a ocorrer em tempo de execução. NesC tenta empurrar para tempo de compilação tantas decisões quanto for possível.

Uma grande vantagem desse modelo orientado a componentes é o fato do desenvolvedor poder construir a aplicação utilizando um conjunto de componentes existentes e adicionando algum código extra necessário à execução do objetivo final da aplicação. Assim, ao invés de ser um SO de propósito geral, TinyOS comporta-se como uma *framework* que permite a construção de um novo TinyOS específico, evitando o uso de componentes que não são necessários para a execução da aplicação.

Outra característica que influencia bastante o NesC é o fato das aplicações serem capazes de responder a eventos do ambiente. Diferentemente dos computadores tradicionais, os nós sensores são usados basicamente para coletar dados e propagar esses dados para uma estação base, ao invés de uma computação de propósito geral [Gay et al, 2003]. Percebemos, então, que as aplicações devem ser dirigidas a eventos, reagindo a eventos do ambiente (chegada de uma mensagem, leitura de uma informação do ambiente) ao invés de uma modelo interativo ou um processamento em lote. Isso nos faz lembrar de outro detalhe importante que direcionou o desenvolvimento do NesC. Eventos e processamento de dados são atividades que devem ser executadas concorrentemente, exigindo, assim, um gerenciamento de concorrência. Contundo, gerenciamento de atividades concorrentes é fonte conhecida de potenciais bugs, tais como condição de corrida (*race conditions*), inversão de prioridade, *starvation*, etc.

Todos problemas clássicos no desenvolvimento de qualquer sistema operacionais moderno.

Pelo fato dos nós sensores possuírem recursos computacionais bastante limitado, o modelo de concorrência do NesC deve ser simples o suficiente de forma que esses problemas não ocorram, com isso não será necessário adicionar código extra na aplicação para tratar potencias *bugs*. De fato, o compilador NesC é capaz de detectar condições de corrida em tempo de compilação [Gay et al, 2003] permitindo, assim, um grau satisfatório de concorrência mesmo com recursos limitados. Além da detecção de condição de corrida, o compilador realiza ainda a criação de componentes estaticamente e eliminação de código morto. NesC proíbe qualquer recurso que impeça a análise estática como ponteiros para função e alocação de memória dinâmica, mas ainda assim é capaz de suportar complexas aplicações além de possuir um extenso apoio da comunidade [Gay et al, 2003].

O resto do trabalho está descrito conforme a seguir: (2) Descrição do Sistema Operacional TinyOS, (3) Apresentação da linguagem de programação NesC, (4) Instalação do ambiente de desenvolvimento no Ubuntu Linux, (5) Criação de uma aplicação de exemplo, (6) Conclusões e (7) Referências Bibliográficas.

2. TinyOS

TinyOS é um sistema Operacional dirigido a eventos projetado para redes de sensores sem fio que possui recursos muito limitados (por exemplo, 8KB de memória de programa, 512 bytes de RAM).

Programar em TinyOS não é difícil, mas um tanto desafiador uma vez que o modelo de programação adotado prioriza fortemente o tratamento dessas restrições em detrimento da simplicidade oferecida para o desenvolvimento de aplicações.

TinyOS atualmente é usado em vários projetos científicos e comercias. No site oficial, http://www.tinyos.net, podemos encontrar uma relação com mais de 60 projetos.

Duas das principais motivações dos projetistas do NesC foi a criação de uma linguagem que fosse mais de acordo com o modelo de programação do TinyOS e reimplementar o próprio TinyOS em NesC [Gay et al, 2003]. Esses objetivos foram atendidos, de forma que hoje o código-fonte do TinyOS está disponibilizado em NesC.

TinyOS possui ainda algumas característica únicas que influenciaram bastante o modelo de programação do NesC. Dentre essas característica, podemos citar [Gay et al, 2003]: (1) uma arquiterura baseada em componentes, (2) um modelo de concorrência simples baseado em eventos e (3) operações divididas em fases (*split-phase operations*).

- **Arquitetura baseada em componentes:**

  Seguindo essa arquitetura, TinyOS disponibiliza um conjunto de componentes de sistemas reusáveis que a aplicação deve conectar através de uma especificação(falaremos com mais detalhes adiante). A especificação é independente da implementação do componente e cada aplicação escolhe os componentes que necessita. Dividir os serviços do TinyOS em componentes traz uma grande vantagem pois permite que serviços não usados sejam excluídos da aplicação.

- **Concorrência baseada em Tarefas e Eventos:**

  TinyOS provê um modelo simples de concorrência baseado em Tarefas (*tasks*) e eventos. Tarefas são mecanismos computacionais adiados. Tarefas não interferem entre si, ou seja, não há preempção entre as mesmas. Uma vez escalonada, uma tarefa é executada até terminar (*Run to Completion*). Componentes podem postergar (*post*) a execução de tarefas. A operação de *post*, realizada através de comandos, termina imediatamente, requisitando uma tarefa a ser executada posteriormente pelo processador.

  Como não há preempção entre as tarefas, o código executado por elas deve ser curto, de forma a evitar que uma tarefa postergue indefinidamente a execução de outras tarefas. Além disso, para garantir que uma tarefa sempre termine, ela não pode bloquear recursos nem entrar em espera ocupada.

  O escalonador de tarefas, implementado no laço principal de uma aplicação TinyOS é configurado para escalonar as tarefas para execução sempre que o processador torna-se disponível. A política FIFO (*First In First Out*) é a padrão.

  Eventos também são executados até terminar, mas podem fazer preempção de tarefas ou outros eventos. Um **evento** sinaliza o término de um serviço, como por exemplo, o envio de uma mensagem, ou a ocorrência de um evento do ambiente. Existem componentes de baixo nível que são conectados diretamente às interrupções de hardware através de "tratadores". Um tratador de evento pode depositar informações no ambiente, escalonar tarefas, sinalizar outros eventos ou chamar comandos. Os tratadores de eventos são executados quando ocorre uma interrupção de hardware e executam até terminar, podendo interromper as tarefas e outros tratadores de interrupção".

- **Operações divididas em fase (Split-phase operations):**

  Vimos anteriormente que as tarefas no TinyOS não são preemtivas, fazendo com que não haja operações de bloqueio. Dessa forma, todas as operação de longa duração devem ser dividas em fases. Há uma clara distinção entre a requisição e o término de uma de uma tarefa. As tarefas são comumente requisitadas por comandos. O comando escalona a tarefa e retorna imediatamente. O término da tarefa sera sinalizado para a aplicação através de um evento (*callback*). Operações que não são divididos em fase não geram eventos, como, por exemplo, acender um *LED*. Como exemplo de uma operação divida em fase, podemos citar o envio de um pacote. Um componente pode invocar o comando *send* para iniciar a transmissão de uma mensagem de rádio e o componente de comunicação, ao término da transmissão, gera o evento *sendDone*. "Cada componente implementa uma metade da operação divida em fase e chama a outra" (Gay et al., 2003).

Segundo [Gay et al, 2003], o modelo de concorrência simples do TinyOS permite um alto nível de concorrência com pouco *overhead*, em contraste com um modelo de concorrência baseado em *threads* no qual a pilha de *threads* consomem memória enquanto ficam bloqueadas disputando recursos do sistema. Ainda segundo [Gay et al, 2003], como em qualquer sistema de concorrência, concorrência e não-determinismo pode ser a fonte de *bugs* complexos, incluindo *deadlocks* e condição de corrida.

A partir da versão 2.0 do TinyOS, as decisões de projeto por trás do desenvolvimento do SO são documentados em TEP's [10] (TinyOS Enhacement Protocols). É uma excelente fonte para quem deseja aprender mais sobre o TinyOS.

Além dos TEP's, podemos citar também o "*TinyOS Documentation Wiki"* (http://docs.tinyos.net/index.php/Main_Page) e o próprio site ofical (http://www.tinyos.net).

3. A linguagem NesC

Programar para TinyOS pode ser um tanto desafiador, uma vez que será necessário aprender uma nova linguagem chamada NesC, [Levis, 2006] uma extensão da linguagem C. Não obstante ser uma nova linguagem, NesC foi criada para incorporar os conceitos e o modelo de execução do TinyOS. Assim todo um novo paradigma de programação deve ser aprendido. TinyOS é orientado a eventos, dessa forma, NesC deve fornecer um modelo de programação orientado a eventos. Os programas em NesC são formados por um conjunto de unidades denominados componentes. Cada componente especifica interfaces bem definidas que são usadas para liga-los a outros componentes formando, assim, programas completos.

Componentes dividem algumas semelhanças com objetos, uma vez que encapsulam estados e interagem através de interfaces bem definidas. Entretanto, diferentemente dos objetos, não há herança entre componentes nem alocação dinâmica.

A complexidade inerente ao desenvolvimento dos componentes não é escrever o código em si, afinal, nesC é bastante parecido com a linguagem C. O desafio está em conectar os vários componentes de forma a obtermos uma aplicação completa.

Abaixo temos listado os conceitos básico da linguagem [Gay et all, 2003]:

- Separação de construção e composição: programas são construídos à partir de elementos denominados **componentes,** que são ligados (*"wired"*) para formar programas completos.
- Especificação do comportamento de componentes através de um conjunto de interfaces. Interfaces podem ser fornecidas ou usadas pelos componentes. Quando um componente fornece uma interface, ele está representando a funcionalidade que ele deseja fornecer para seu usuário. Quando um componente usa uma interface, ele está seguindo uma funcionalidade que precisa para realizar sua tarefa.
- Interfaces são bidirecionais: Eles especificam um conjunto de funções que devem ser implementadas pelo fornecedor da interfaces (comados) e pelo usuário da interfaces (eventos). Com isto, uma simples interface pode ser usada para representar uma interação complexa entre componentes. (Operações divididas em fase). Como vimos anteriormente, as operações no TinyOS (como por exemplo, o envio de um pacote) não podem bloquear outras tarefas. Assim, o encerramento de uma operação é sinalizada através dos eventos (por exemplo, quando o envio é finalizado). Especificando interfaces, temos a certeza que o componente não pode chamar o comando **send** sem antes fornecer uma implementação para o evento **sendDone**.

- Componentes são ligados (“wired”) através das interfaces. Esse recurso melhora a eficiência em tempo de execução além de permitir uma melhor análise estática dos programas.
- O modelo de concorrência no NesC é baseado em tarefas *“run-to-completion”*; e gerenciadores de interrupção que podem interromper tarefas bem como eles próprios. O compilador nesC sinaliza eventuais condições de corrida causada por gerenciadores de interrupção.

3.1 Componentes e Interfaces

NesC define dois tipos de componentes:

- **Módulos:** Componentes implementados com código NesC.
- **Configurações:** Componentes implementados através da ligação de componentes uns aos outros.

Programas em NesC são montados a partir de componentes conectados através de interfaces que eles fornecem ou usam [Gay et al, 2007]. Figura 2 ilustra 6 componentes conectados através das interfaces *Sense* e *Initialize*. No exemplo, *Main* (um componente de “*boot”* do sistema), *LightM*, *TempM* e *AppM* são módulos, enquanto *AppC* e *SensorsC* são configurações.

Módulos e Configurações possuem um nome, especificação e implementação. Veja o exemplo abaixo:

```
module AppM {
    provides interface Initialize as Init;
    uses interface Sense as Sensor1;
    uses interface Sense as Sensor2;
}
implementation { ... }
```

O código acima declara que AppM é um módulo que provê uma interface chamada *Init* e usa duas interfaces, *Sensor1* e *Sensor2*. Cada interface tem um tipo, neste exemplo foram usados os tipos *Initialize* e *Sense*. Vale destacar que um nome de componente denota um componente único ou *singleton*. Qualquer referência ao componente Main em diferentes trechos da aplicação farão referência ao mesmo componente.

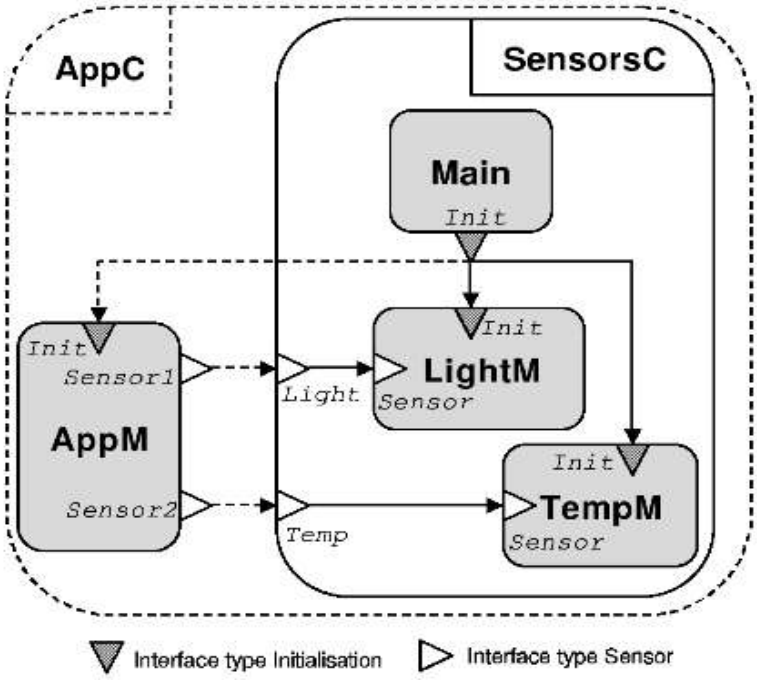

Figura 2 – Componentes conectados através de Interfaces

As interfaces definem as interações entre um componente provedor de serviço e um componente que usará esse serviço:

```
interface Initialize {
        command void init();
}

interface Sense {
        command void sense();
        event void senseDone();
}
```

Essas interações são bidirecionais. Comandos são invocações que o usuário pode fazer para o componente provedor do serviço e eventos são do provedor ao usuário. As interfaces geralmente são utilizadas para representar uma operação dividida em fases (*split-phase operations*). A interface *Sense* é um bom exemplo. O componente que prover a interface *Sense* deve implementar o comando *send* (por exemplo, requisitar a leitura de um sensor), enquanto os usuários devem implementar o evento *sendDone*, o qual será usado pelo provedor para avisar quando a leitura do sensor terminar. No NesC eventos são sinalizados (*signaled*) enquanto os comandos são chamados (*called).* De qualquer forma, em ambos os casos a interação dá-se através de uma chamada de função.

Voltando ao nosso exemplo, AppM, por ser um módulo, deve fornecer implementações para os comandos das interfaces que ele fornece e para os eventos das

interfaces que ele usa. Ele pode chamar ou sinalizar qualquer um dos comandos e eventos e pode também postergar a execução de tarefas (*tasks*). Assim:

```
module { ... }
implementation {
        int sum = 0;
        task void startSensing() {
                call Sensor1.sense();
        }
        command void Init.init(){
                post startSensing();
        }
        event void Sensor1.senseDone(int val) {
                sum += val;
                call Sensor2.sense();
        }
        event void Sensor2.senseDone(int val){
                sum += val;
        }
}
```

Pelo exemplo acima, podemos perceber a nomenclatura dos comandos e eventos. Um comando ou evento *f* de uma interface *I* é chamado *I.f*. Módulos encapsulam seu estado. Dessa forma, todas as suas variáveis são privadas.

3.2 Configurações

Configurações (*Configurations*) são componentes cuja função é a de ligar (*wire*) componentes uns aos outros de acordo com as interfaces fornecidas e usadas por esses componentes. Dois componentes podem apenas interagir entre si se alguma configuração tiver ligado os dois:

```
configuration SensorsC{
        provides interface Sense as Light;
        provides interface Sense as Temp;
}
implementation {
        components Main, LightM, TempM;

        Main.Init -> LightM.Init;
        Main.Init -> TempM.init;
```

```
    Light = LightM.Sensor;
    Temp = TempM.Sensor;
}
```

O componente SensorsC é formado por três componentes, LightM e TempM e Main. SensorsC provê interfaces para os componente sensores (LightM e TempM). A interface Light é associada à interface Sensor de LightM e a interface Temp é associada à interface Sensor de TempM.

Por fim, a configuração AppC junta todas as peças desse quebra-cabeça.

```
Configuration AppC { }
implementation {
    components Main, AppM, SensorsC;

    Main.Init → AppM.Init;
    AppM.Sensor1 -> SensorsC.Light;
    AppM.Sensor2 -> SensorsC.Temp;
}
```

4. Instalação do Ambiente de Desenvolvimento

O Ambiente de desenvolvimento TinyOS é composto basicamente por:

- Código-fonte do TinyOS
- Compilador nesC
- Compilador gcc específico para a plataforma de Hardware (por exemplo o microcontrolador AVR Atmega).
- Bibliotecas
- Aplicações de demonstração

O processo de compilação de uma aplicação ocorre de forma automática através do comando *make,* utilitário muito popular nos ambientes *unix-like* para automatizar tarefas. *Make*, primeiramente, executa o compilador **nesC** o qual gerará um código **C** a partir do código-fonte **nesC**. Em seguida, será executado o compilador **gcc** especifico para a plataforma para qual a aplicação será executada. Nos nossos testes estamos usando o modelo *mica2* que utiliza o micro-processador AVR Atmega, sendo, então, necessário o **gcc** compilado para a essa plataforma.

Em nossos trabalhos utilizamos a distribuição Ubuntu Linux [7] como base para o desenvolvimento das aplicações TinyOS. A instalação do ambiente de desenvolvimento TinyOS pode ser feita de forma rápida e prática em sistemas Ubuntu através da ferramenta Apt-get [8]. A versão do Ubuntu utilizada neste artigo foi a 8.04, denominada *Hardy Heron.* A instalação do ambiente pode ser realiza em três ou quatro passos:

1. Remover qualquer repósitorio TinyOS antigo do arquivo /etc/apt/sources.list e adicionar a seguinte linha:

   deb http://tinyos.stanford.edu/tinyos/dists/ubuntu feisty main

2. Realizar a atualização do cache do repositório

   sudo apt-get update

3. Instalar os pacotes necessários

   sudo apt-get install tinyos tinyos-avr

   Obs: Caso o modelo do sensor utilize o micro-controlador MPS430, é necessário o pacote **tinyos-msp430** ao invés do **tinyos-avr**

4. Por último, se você estiver usando o nó sensor da família *Telos* será necessário remover qualquer driver *braille*, pois o Ubuntu pensa que os sensores *Telos* são entradas *braille*. Você pode remover os drivers com o seguinte comando:

   sudo apt-get remove brltty

Ao final desse processo, o ambiente de desenvolvimento será instalado no seguinte diretório:

/opt/tinyos-2.x/

É necessário ainda possuir uma versão do JDK instalado. A documentação oficial do TinyOS recomenda versão 1.4 ou a 1.5. Em nossos testes, utilizamos a versão 1.6, a qual não demonstrou nenhuma incompatibilidade. É necessário que as variáveis de ambiente PATH e CLASSPATH estejam devidamente configuradas. A versão mais recente da JDK pode ser encontrada no site oficial da SUN.

Recomendamos ainda a instalação da aplicação Graphviz [13] através do comando:

sudo apt-get install graphviz

Graphviz é uma aplicação open-source (http://www.graphviz.org), que permite a visualição de gráficos. A instalação do Graphviz não é obrigatória, contudo é altamente recomendada, pois o mesmo é utilizado para a geração da documentação das aplicações TinyOS.

Caso você não utilize o Ubuntu Linux, no site oficial do TinyOS encontram-se disponíveis tutoriais para instalação do ambiente de desenvolvimento não só para outras distribuições Linux como também para o sistema Windows©.

5. Desenvolvimento de uma aplicação

Neste tópico, abordaremos o desenvolvimento de uma aplicação TinyOS simples, mas interessante o suficiente para verificarmos todos os passos necessários para se ter uma aplicação completa e instalada no nó sensor.

Os exemplos utilizados nessa sessão foram adaptados livremente do *Wiki* oficial do TinyOS (http://docs.tinyos.net/index.php/Main_Page).

Nos nossos testes utilizaremos como plataforma de desenvolvimento o modelo de sensor *mica2* da empresa Crossbow (http://www.xbow.com). Abaixo, temos uma imagem do *mica2*:

Figura 3 - Imagem do Mica2

As principais característica do *mica2* são:

- Memória Flash Programável: 128Kbytes
- Comunicação Serial: UART
- Microprocessador: AVR ATmega 128L (8 bits)
- Rádio: CC1000 da Chipcon com modulação FSK na faixa de 315, 433, 868, 915 Mhz.

5.1 Conhecendo a aplicação Blink

Para conhecer como escrever uma aplicação nesC do zero, utilizaremos uma aplicação de demonstração que acompanha o ambiente de demonstração. Blink é uma aplicação bastante simples, mas interessante o suficiente para conhecermos melhor a estrutura de um programa desenvolvido em NesC. Basicamente, tudo o que ele faz é piscar os três LED's do equipamento em uma ordem pré-definida: LED0 pisca a uma freqüência de 25Hz, LED1 a uma freqüência de 5Hz enquanto que o LED2 pisca a uma freqüência de 1Hz. O efeito visual resultante é a impressão de que os três LED's estão "imprimindo" a contagem de 0 até 7 em binário à cada 2 segundos.

Blink é composto de dois componentes: um **módulo** chamado "BlinkC.nc" e uma configuração chamada "BlinkAppC.nc". Lembre que toda aplicação requer uma configuração *top-level*. Neste caso, BlinkAppC.nc será usado para ligar o módulo BlinkC.nc a todas os outros módulos que a aplicação precisa.

A grande vantagem dessa divisão entre módulos e configurações é a possibilidade de um programador criar novos programas apenas ligando objetos existentes. Imagine que você tem uma coleção de blocos "*lego ©* " e vai arrumando as peças que você quer até atingir a formação que você deseja. Voltando aos componentes, caso os existentes

não sejam suficientes para completar sua aplicações, o programador tem a liberdade de desenvolver novos.

Não há uma regra fixa para a nomenclatura dos arquivos. Mas geralmente, segue-se o padrão abaixo:

Tipo de Nome	Tipo do Arquivo
Foo.nc	Interface
Foo.h	Header File
FooC.nc	Public Module
FooP.nc	Private Module

5.1.1 O Arquivo de Configuração BlinkAppC.nc

Conforme visto anteriormente, toda aplicação necessita de uma configuração *top-level*. Esse arquivo fica definido dentro de um arquivo localizado no diretório da aplicação chamado de *Makefile*.

Vamos analisar a estrutura do arquivo BlinkAppC.nc:

```
configuration BlinkAppC
{
}
implementation
{
  components MainC, BlinkC, LedsC;
  components new TimerMilliC() as Timer0;
  components new TimerMilliC() as Timer1;
  components new TimerMilliC() as Timer2;

  BlinkC -> MainC.Boot;
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
  BlinkC.Leds -> LedsC;
}
```

Perceba a palavra ***configuration*** no começo do arquivo, ela é usada para definir um arquivo do tipo configuração:

```
configuration BlinkAppC
{
}
```

Perceba as chaves em branco. Dentro delas é possível especificar as clausulas *uses* e *provides*, caso fosse necessário usar ou prover alguma interface.

A configuração de fato é implementada dentro do escopo ***implementatios***. As linhas com ***componentes*** especificam o conjunto de componentes que serão conectados dentro dessa configuração. Neste caso temos os componentes *MainC*, *BlinkC, LedsC e* três instâncias do componente *TimerMilliC* que serão referenciados pelos nomes *Timer0, Timer1* e *Timer2*.

As quatros últimas linhas ligam as interfaces que o componente BlinkC usa com as interfaces que os componentes LedsC e TimerMilliC prover.

Vamos olhar o componente BlinkC para entendemos melhor como toda essa estrutura funciona.

5.1.2 O Módulo BlinkC.nc

```
module BlinkC
{
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
        //Código omitido
}
```

A primeira parte indica que este arquivo é um módulo chamado BlinkC além de definir as interfaces que ele usa e prover. BlinkC está utilizando três interfaces *Timer<TMilli>*, além das interfaces *Leds* e *Boot*. Perceba que deve haver uma congruências entre as interfaces definidas aqui e as ligações feitas no arquivo de configuração. Tomaremos como exemplo a interface *Leds*. BlinkC ao usar (*uses)* a interface *Leds* indica que pretende chamar algum comando definido por essa interface. Entretanto, lembre que interfaces não definem implementações. É aí onde entra o arquivo de configuração. Nele indicaremos qual componente, dentre os vários possíveis que implementa a interface *Leds*, será conectado ao componente BlinkC. Verificando no arquivo de configuração BlinkAppC.nc, percebemos que a interface Leds de BlinkC está conectada (*wired*) à interface Leds do componente LedsC:

BlinkC.Leds -> LedsC;

NesC usa a flecha (->) para ligar as interfaces entre componentes. A flecha para a direita (A -> B) indica A está conectado a B ou A usa B. Nomear uma interface é importante quando um componente usa ou prover várias interfaces. No nosso exemplo, BlinkC usar três instâncias de Timer: *Timer0*, *Timer1* e *Timer2*. Quando um componente tem somente uma instância de uma interface, pode-se omitir o nome da interface. Dessa forma:

BlinkC.Leds -> LedsC;

seria o mesmo que:

BlinkC.Leds -> LedsC.Leds;

5.1.3 Visualizando um gráfico de Componentes

Programar em TinyOS não é algo realmente difícil, contudo, em uma aplicação robusta é comum ter-se várias configurações conectando vários componentes, que por sua vez podem ser outras configurações. Percebe-se que há uma certa dificuldade em visualizar o sistema como um todo. Felizmente dispõe-se de uma ferramenta, chamada de *Nescdoc*, para auxiliar esse processo. *Nesdoc* gera uma documentação automática a partir do código-fonte. Para gerar a documentação basta digitar o comando abaixo dentro do diretório da aplicação:

$ make *platform* docs

Se ocorrer o seguinte erro:
sh: dot: command not found

Então deve-se instalar o programa Graphviz.

5.2 Compilando a aplicação

Antes de prossegui é interessante checar se o ambiente está configurando corretamente execução o comando *tos-check-env*. Em um terminal, simplesmente digite:

$ tos-check-env

O comando irá checar praticamente tudo que o ambiente TinyOS necessita. As mensagem geralmente são auto-explicativas de forma que se você está acostumado ao ambiente Unix-like não terá dificuldades em compreender.

Para possibilitar a compilação de suas aplicações será necessário uma variável de ambiente chamada MAKERULES. Digite o comando abaixo para confirmar se a mesma não está definida:

$ echo $MAKERULES

Caso a resposta seja apenas uma linha em branco então a variável não está definida. Digite o comando abaixo para cria-la, supondo que o TinyOS foi instalado no diretório padrão:

$ export MAKERULES=/opt/tinyos-2.x/support/make/Makerules

Caso o TinyOS tenha sido instalado em algum diretório diferente do padrão, troque **/opt/tinyos-2.x** pelo diretório da instalação.

Executando o comando acima, a variável será criada com sucesso, contudo, ficará visível apenas enquanto durar a sessão. Para torna-la permanente é necessário defini-la no arquivo **/etc/environment**. Caso utilize outra distribuição, procure o arquivo de configuração do shell bash correspondente. Utilizando algum editor de texto, deve-se abrir o arquivo **/etc/environment**. Em nosso exemplo, foi utilizado o *vim*. Assim:

$ sudo vim /etc/environment

Adicione a seguinte linha ao final do arquivo:

MAKERULES=/opt/tinyos-2.x/support/make/Makerules

Em seguida, o arquivo deve ser salvo. Vale lembrar que para editar o referido arquivo é necessário possuir poderes administrativos.

Para compilar uma aplicação TinyOS é necessário a execução do comando *make* dentro do diretório da aplicação. A sintaxe do comando para a compilação da aplicação é **make *[platform]***. *Platform* representa o hardware para o qual a aplicação será compilada. Caso você não possua nenhum sensor disponível, pode-se compilar a aplicação para o TOSSIM, o simulador do TinyOS.

Nosso primeiro teste será compilar uma aplicação chamada Blink que acompanha a instalação padrão do TinyOS. Para compilar Blink, acesse o diretório apps/Blink e, de acordo com sua plataforma, digite *make micaz*, *make mica2*, *make telosb* ou, para simulação, *make micaz sim*. Em nossos testes, estamos usando a plataforma mica2, assim:

$ make mica2

Deverá ser impresso na tela a seguinte saída:

mkdir -p build/mica2

compiling BlinkAppC to a mica2 binary

ncc -o build/mica2/main.exe -Os -finline-limit=100000 -Wall -Wshadow -Wnesc-all -target=mica2 -fnesc-cfile=build/mica2/app.c -board=micasb -DIDENT_PROGRAM_NAME=\"BlinkAppC\" -DIDENT_USER_ID=\"marvin\" -DIDENT_HOSTNAME=\"skyship\" -DIDENT_USER_HASH=0x3b3023baL -DIDENT_UNIX_TIME=0x481a6a0eL -DIDENT_UID_HASH=0xe1b7fae7L -fnesc-dump=wiring -fnesc-dump='interfaces(!abstract())' -fnesc-

dump='referenced(interfacedefs, components)' -fnesc-dumpfile=build/mica2/wiring-check.xml BlinkAppC.nc -lm

compiled BlinkAppC to build/mica2/main.exe

7130 bytes in ROM

52 bytes in RAM

avr-objcopy --output-target=srec build/mica2/main.exe build/mica2/main.srec

avr-objcopy --output-target=ihex build/mica2/main.exe build/mica2/main.ihex

writing TOS image

5.3 Instalando a aplicação

Instalaremos a aplicação Blink em um mica2. Para isso, deve-se conectar o equipamento a uma placa de programação (por exemplo, *mib510*). A placa de programação, por sua vez, deve ser conectada ao computador através de uma comunicação serial. Para instalar sua aplicação, digite o seguinte comando:

$ make mica2 reinstall mib510,*serialport*

Onde *serialport* é o nome do dispositivo que representa a porta serial ao qual está conectado a placa de programação. Em sistemas Windows©, a porta serial é representada por **COM*n*** ao passo que em ambiente Linux é representado por **/dev/ttyS*n***, onde ***n*** identifica o número da porta.

É interessante destacar que o mesmo comando acima vale para a placa de programação *mib520*.

Em sistemas Linux, se houver problema em acessar a porta, é bem provável que o usuário não tenha permissão de escrita no arquivo. Para resolver esse problema, digite o seguinte comando como super-usuário:

chmod 666 *serialport*

Se ocorrer tudo bem com a instalação, deverá ser gerada a seguinte saída:

cp build/mica2/main.srec build/mica2/main.srec.out
installing mica2 binary using mib510
uisp -dprog=mib510 -dserial=/dev/ttyUSB1 --wr_fuse_h=0xd9 -dpart=ATmega128 --wr_fuse_e=ff --erase --upload if=build/mica2/main.srec.out
Firmware Version: 2.1
Atmel AVR ATmega128 is found.
Uploading: flash
Fuse High Byte set to 0xd9

Fuse Extended Byte set to 0xff
rm -f build/mica2/main.exe.out build/mica2/main.srec.out

6. Conclusões

Redes de Sensores Sem Fio é um tema relativamente novo e excitante, contudo, impõe novos desafios uma vez que praticamente é impossível o reaproveitamento de conceitos e algorítimos existentes na computação tradicional, devido as várias restrições impostas por essa nova tecnologia (energia, recursos computacionais limitados, etc). Não poderia ser diferente com sistemas Operacionais e Linguagens de programação. A natureza *multithread* dos sistemas operacionais tem se mostrado ineficiente nos equipamentos sensores devido à grande limitação de memória e processamento dos mesmos. Além disso, temos que lembrar da natureza orientada a eventos das aplicações em RSSF.

Neste relatório, apresentamos o TinyOS, um sistemas operacional projetado especialmente para as Redes de Sensores e a linguagem de programação NesC que facilita o desenvolvimento de aplicações para o TinyOS.

TinyOS possui algumas característica que o tornam ideal para as RSSF como um modelo de programação orientado a componentes e eventos, programação estática dentre outros.

NesC tem se tornado uma linguagem de fato para programação TinyOS. Podemos citar vários bons projetos desenvolvidos em NesC como TinyDB [Madden et al, 2003] e Maté .

No Brasil, vários são os projetos que utilizam o TinyOS. Em [Fernandes et al, 2004] temos PROC: Um protocolo Pró-Ativo com Coordenação de Rotas em Redes de Sensores sem Fio, enquanto [Rosseto, 2006] implementou gerência cooperativa de tarefas no TinyOS.

Como trabalhos futuros, pretendemos investigar com mais detalhes o modelo de programação NesC, desenvolvendo uma aplicação completa capaz de fazer sensoriamento em um determinado ambiente e transmitir essas informações para uma estação base, o qual será capaz de extrair as informações dos pacotes recebidos e armazená-los em um banco de dados.

Além disso, integraremos essa aplicações com o TOSSIM, ambiente de simulação do TinyOS, para verificar como o mesmo se comportaria em vários cenários de forma que possamos analisar o seu desempenho.

7. Referências

Pereira , Marluce R., Amorim, Cláudio L. De e Castro, Maria Clicia Stelling de. Tutorial sobre Redes de Sensores

Loureiro, Antônio A. F; Nogueira, José Marcos S.; Ruiz, Linnyer Beatrys; Mini, Raquel Aparecida de Freitas; Nakamura, Eduardo Freire; Figuereiro , Carlos Maurício Seródio. Redes de Sensores sem Fio. Em XXI Simpósio Brasileiro de Redes de Computadores. Pag. 179-226.

Gay, David, Levis, Philip e Culler, David David. Software Design Patterns for TinyOS. Em ACM Trans. Embedd. Comput Syst. 6, 4, Article 22 (September 2007), 39 páginas.

Gay, David, Levis, Philip, Behren, Robert Von, Welsh, Matt, Brewer, Eric e Culler, David. The NesC Language: A Holistic Approach to Networked Embedded System. San Diego, California, USA, 2003.

Gay, David; Levis, Philip; Culler, David e Brewer, Eric. NesC 1.1 Language Reference Manual. 2003.

Levis, Philip. TinyOS Programming. 2006

Madden, Sam; Hellerstein, Joe e Hong, Wei. TinyDB: In-Network Query Processing in TinyOS. 2003

Levis, Philip; Lee, Nelson; Welsh, Matt e Culler, David: TOSSIM: Acurate and Scalable Simulation for Entire TinyOS Applications. Em *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003).*

Macedo, Daniel Fernandes; Correia, Luiz Henrique Andrade; Loureiro, Antônio Alfredo Ferreira; Nogueira, José Marcos. PROC: Um protocolo Pró-Ativo com Coordenação de Rotas em Redes de Sensores sem Fio. Em 22. Simpósio Brasileiro de Redes de Computadores, 2004.

Silvana Rosseto. Integrando comunicação assíncrona e gerência cooperativa de tarefas em ambientes de computação distribuída. Tese de Doutorado, 2006.