ver.0.9

TinyOS 프로그래밍

KETI / Ubiquitous Technology Research Center (www.keti.re.kr)
TinyOS Korea Forum (www.tinyos.or.kr)
강정훈, 유준재, 윤명현,이민구, 임호정

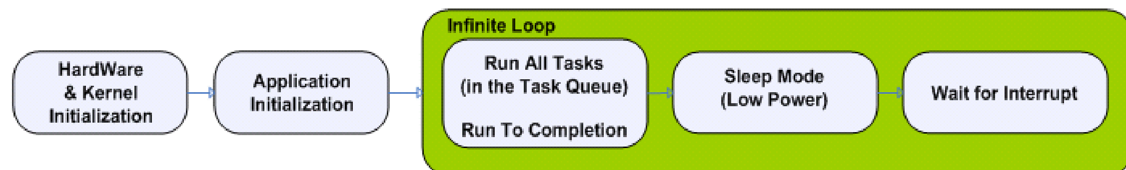
ⓒ TinyOS Korea Forum, Jeonghoon Kang, 2007. All rights reserved.
본 문서는 비상업적 목적으로 수정 없이 재배포할 수 있습니다.
jeonghoon.kang@gmail.com

1. TinyOS Kernel

TinyOS는 기존의 PC용 또는 임베디드 시스템용 OS들과 다르게 한번에 하나의 애플리케이션만을 실행한다. 기본적으로 TinyOS는 센서로부터 데이터를 취득하여 무선 네트워크로 전송하는 간단한 동작을 담당하기 때문에, TinyOS의 설계 단계에서 전원소모와 메모리 소모가 많아지게 할 수 있는 여러 개 애플피케이션이 동시에 동작되는 기능은 제외하였다. 다만, 여러 가지 기능을 동시에 동작시키고자 할 때에는 하나의 애플리케이션에, 아주 작은 단위로 동작을 세분화하여 태스크(Task)형태로 동작 시킨다. 결국, TinyOS 커널은 이런 특징 때문에 하나의 main() 함수로 구성되어 있다.

TinyOS 커널의 동작 순서에 대한 이해에 앞서, nesC로 작성된 소프트웨어의 컴파일 순서를 이해해야 한다. nesC로 작성된 코드는 nesC 컴파일러에 의해 C 언어로 변환된다. C 언어로 변환된 후 GCC(GNU C 컴파일러)에 의해 기계 수준의 바이너리 파일로 컴파일 된다. 컴파일은 make를 이용하여 진행되며(예, make telosb), nesC의 C 변환과 컴파일이 한번에 이루어진다. 컴파일 과정에서 사용되는 nesC 파일 (*.nc)들은 내용에 따라 해당 디렉토리에 분산되어 있는데, make시에 포함되는(include) 옵션을 분석하여 찾을 수 있다.

컴파일된 바이너리 파일을 플랫폼에 다운로드한 후, 전원을 인가하면 하드웨어 초기화와 커널 초기화가 이루어진다. 이후 TinyOS의 커널이 동작하게 된다. TinyOS 커널은 단순한 FIFO 형태의 스케줄러로 동작을 하며, 이 스케줄러는 전원이 꺼질 때 까지 무한 루프(loop)에서 동작한다.



< 부팅 후 플랫폼 동작 순서 >

전원이 켜졌을 때의 동작은 MainM.nc 파일에서 분석할 수 있다. MainM component의 위치는 make 파일을 분석하여 위치를 찾을 수 있다. MainM은 nesC가 C언어로 변환될 때 사용되는 main() 함수를 포함하고 있다.

아래 MainM component는 텔로스B(Telos revision B)용 소프트웨어 컴파일에 사용되는 것이다. 전원이 켜지면 main()의 첫 번째 줄인 call hardwareInit()부터 실행된다. hardwareInit()은 하드웨어, 즉, 센서와 칩들에 대한 초기화를 실행한다. 하드웨어 초기화 후에, 스케줄러 초기화를 위한 TOSH_sched_init()을 호출하여 스케줄러의 변수와 태스크 큐(Task Queue)를 초기화 한다.

call StdControl.init()과 call StdControl.start()는 각 애플리케이션에서 사용하는 component들의 초기화를 위한 command이다. 이 command는 각 component들이 사용되기 위해 필요한 초기화 루틴(routine)을 포함하고 있으며, 애플리케이션에서 사용되는 component가 많은 경우에는 각 component의 StdControl 코드가 모두 선택되어 이 위치에서 호출된다.

애플리케이션 초기화가 StdControl.init(), StdControl.start()를 통해 이루어진 후에는 하드웨

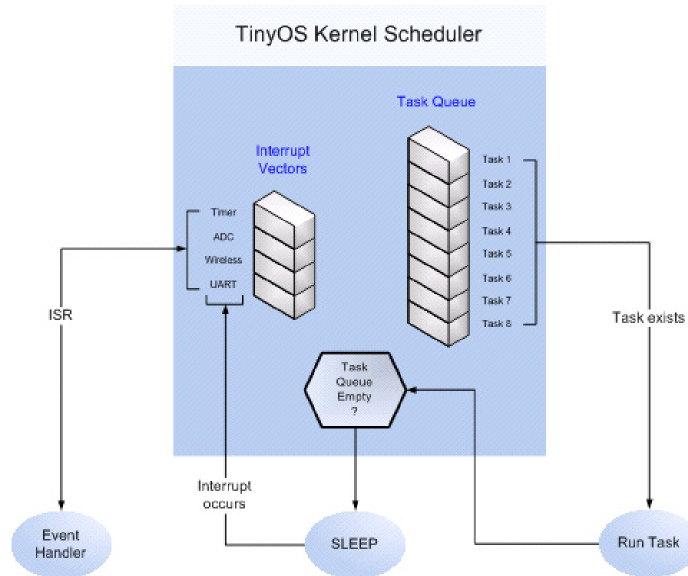
어 인터럽트를 __nesc_enable_interrupt()를 통해 동작시키고 무한 루프에서 TinyOS 커널, 즉, TOSH_run_task()로 TinyOS 태스크 스케줄러를 동작시킨다.

```
module MainM
{
  uses command result_t hardwareInit();
  uses interface StdControl;
}
implementation {
  int main() __attribute__ ((C, spontaneous))   {
    call hardwareInit();
    TOSH_sched_init();
    call StdControl.init();
    call StdControl.start();
    __nesc_enable_interrupt();
    for(;;) { TOSH_run_task(); }
  }
}
```

< MainM 의 main() >

TinyOS 커널(Kernel)의 구조도는 초기화 다음에 무한루프에 빠지면서 하나의 함수를 실행시키고 슬립(Sleep) 상태로 바뀌게 되는 것을 표현하고 있다. MainM component의 main()에서 무한루프로 들어가게 되면, 아래 그림의 " Task Queue Empty? " 부분에서 시작되며, 초기화후에는 Task Queue가 비어 있으므로 바로 슬립(Sleep)상태로 바뀌게 된다.

애플리케이션 동작을 실행시키기 위해 발생하는 것이 하드웨어 인터럽트(Interrupt) 이다. 이런 인터럽트 기반에서 초기화 소프트웨어 동작은 중요한 의미를 가진다. 초기화된 루틴 후에 플랫폼은 슬립상태로 존재하고 하드웨어 인터럽트가 발생함에 따라 하드웨어 인터럽트에 관련된 동작을 수행하며 수행이 완료 되면 다시 슬립 상태가 된다. 이런 반복적인 하드웨어 인터럽트에 따라 동작하는 것이 network embedded system의 가장 큰 특징이다.



< TinyOS Kernel, Task Scheduler >

main() 함수에서 StdControl에 의해 초기화 할 때 설정해 놓은 타이머(Timer)등의 하드웨어 인터럽트들이 발생하면 해당 인터럽트 서비스루틴으로 빠지게 된다. 위 그림에서는 인터럽트 벡터를 통해, 해당 이벤트 핸들러로 동작이 수행된다. 이벤트 핸들러의 동작이 종료 되면 다시 Task Queue Empty ? 부분으로 돌아가 다음 하드웨어 인터럽트를 기다리게 된다.

센서네트워크가 아닌, 지금까지의 PC, 임베디드 시스템에서 OS들은 인터럽트 서비스루틴에서 간단한 작업만을 한다. OS 코어에게 인터럽트가 발생하였다는 표시만 하고 다시 인터럽트를 기다리는 위치로 간다. 이 경우, OS는 정해진 시간마다 인터럽트가 발생했는지 반복해서 점검하게 되고, 인터럽트가 발생했을 경우에는 해당 인터럽트를 처리한다.

TinyOS에서는 nesC 장점을 살려서 인터럽트가 발생하게 되면 실시간(Real-Time)으로 해야 될 동작을 바로 수행 한다. 이런 방식을 통해 대부분 짧은 처리 시간을 갖는 인터럽트 핸들러로 구성된 TinyOS 애플리케이션을 효율적으로 실행 시킨다. 기존 OS들처럼 하드웨어에서 발생한 인터럽트를 처리하게 되면 이벤트 관리 함수가 많아지게 되며, 메모리와 전력 사용이 추가되는 단점이 발생한다.

이벤트 핸들러에 따라서 동작들이 수행 되고, 실행되는 동작들 중에 일부 함수는 실행할 때 일반적인 호출 방식이 아닌 태스크(Task)라는 방식으로 실행 시킬 수 있다. 이때, 포스트(post)라는 키워드를 쓰게 된다. 그렇게 되면 여기서 실행되는 함수들이 곧바로 실행되지 않고 Task Queue에 그 함수의 주소를 입력하게 된다. 그렇게 되면 태스크 형태의 함수를 제외한 함수들이 다 실행된 후에, Task Queue에 함수 주소가 있는지 없는지를 확인하게 된다. 태스크 함수가 있다면 이 함수의 주소에 해당하는 함수를 실행을 시켜(Run Task) 주고, Task Queue에 주소가 없어 질 때까지 태스크의 실행을 반복한다. 실행이 완료되면 Task Queue안에 실행해야 할 함수의 주소가 없을 것이고 이때 플랫폼은 Sleep 상태로 빠지게 된다.

어떻게 보면 Event Handler에서 호출하는 태스크 함수는 곧바로 실행이 안 되고, 추후에

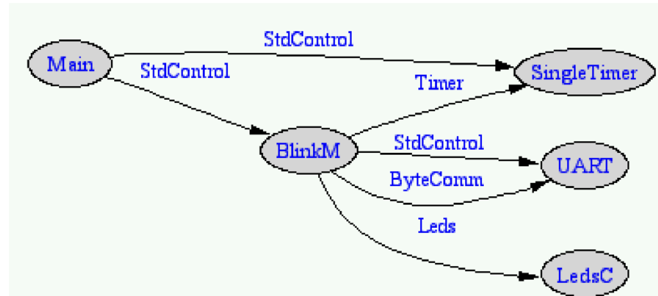
다른 일반 함수들이 다 실행된 후에 동작 되니 이것은 Real-Time을 보장하지 못하는 방법으로 생각될 것이다. 이 TinyOS의 커널은 메인함수에도 봤듯이 초기화하는 과정이 매우 짧은 시간동안 이루어진다. 메인함수에서 많은 하드웨어와 소프트웨어 초기화 명령이 수행되었음에도 불구하고 짧은 시간 안에 실행이 이루어지기 때문에 이벤트 핸들러에서 수행하는 코드의 양이 많아도 전체적으로 걸리는 시간은 짧기 때문에, 함수 하나를 중간 실행하지 않고 태스크 큐에 넣어주고 이것의 실행을 나중으로 미루고 나머지 함수들을 실행해 준 후, Task Queue에 있는 함수를 실행해 주어도 상대적으로 많은 시간 지연이 발생하지 않는다.

2. nesC의 특징

nesC는 component(컴포넌트) 기반의 프로그래밍 언어이기 때문에 여러 가지 장점이 있다. Component를 애플리케이션 소프트웨어를 만드는 부품이라는 의미로 생각하는 것이 이해에 도움이 된다. 부품들을 조립하면 하나의 시스템이 만들어지듯이, 소프트웨어 부품(component)들을 잘 연결하여 조립하면 쉽게 하나의 애플리케이션 소프트웨어를 만들 수 있다는 것이 nesC의 가장 큰 장점이다. 임베디드 네트워크 환경에서의 코드 재 사용성이 많은 프로그래밍 언어이다. 만일, 1초마다 LED가 점등되는 애플리케이션을 만들고 싶으면, 기본 component인 Main, 시간을 관리하는 Timer, LED를 관리하는 LedC 세 개의 component를 조립하면, 하나의 애플리케이션을 만들 수 있다. 조립은 interface(인터페이스)라는 연결 도구를 이용한다.

nesC의 또 하나의 장점은 컴파일 시에 하드웨어 동작 시에 발생할 수 있는 에러를 체크할 수 있도록 만들어져 있다는 것이다. Linux 등의 OS에서는 " ioctl(fd,TCSETS,data); " 방법으로 CPU에 연결되어 있는 디바이스를 제어한다. 이 방법은 디바이스를 지시하는 fd(인자) 내용이, 동작(running)시에 명령을 내리는 디바이스마다 다르게 되며, 시스템의 특징에 따라 소프트웨어 개발자가 다른 이름으로 만들 수 있다. nesC는 이런 방식과 다르게, CPU에 연결되는 각 디바이스들를 동작 시키는 컴포넌트로 만들어 놓고, 이 컴포넌트를 이용하기 때문에 컴파일 할 때에 적절한 디바이스를 사용했고, 이 실제 동작 시에 발생할 수 있는 오류를 컴파일 과정 중에 체크할 수 있다. Linux 소프트웨어를 컴파일 할 때, 플로우차트나 어떤 동작이 일어날 것인가를 컴파일러가 파악하기는 힘들다. fd 라는 인자에 어떠한 디바이스가 지정될지 변수 자체만으로는 판단하기 힘들고, 또 각 디바이스를 제어하는 명령을 내릴 때, 이것이 적절한 명령인지 등을 파악할 수 없다. 센서 네트워크 디바이스처럼 한정된 개수의 간단한 주변 디바이스를 갖는 경우, nesC로 만들어진 컴포넌트는 소프트웨어의 동작상태(running) 상태에서의 오류를 컴파일 시에 체크하고 이를 방지할 수 있는 특징을 지니고 있다.

아래의 그림은 Blink 라는 애플리케이션의 조립상태(구조)를 보여주고 있다. nesC의 컴포넌트들은 인터페이스를 이용해서 조립되는데, 아래 그림에서 Main component는 애플리케이션에 반드시 필요한 최상위 component이며, 이것이 SingleTimer, BlinkM component와 각각의 StdControl interface로 연결되어 있다.



< 컴포넌트 연결 / 조립 상태도 >

3. nesC 프로그래밍

nesC의 함수는 commad와 event라는 두 가지 함수를 사용한다. command는 일반적인 c 언어의 함수와 비슷하고, 명령의 방향이 상위 소프트웨어에서 하위 하드웨어로 호출된다. event는 하드웨어 동작에 대한 이해가 필요한데, command (명령)에 따른 기능 수행 후에 발생하는 이벤트를 호출하는 함수이다.

2-Level Structure

Command (Bottom direction)
Non-time critical
Long running operations
Cannot preempt & can be preempted
Background computation

Events (Top direction)
Time critical
Small running operations
Cannot be preempted
Able to interrupt running Tasks

TinyOS는 non Preemption 방식으로 동작한다. 이는 현재 동작하고 있는 태스크를 중지시키지 못하고, 먼저 실행되고 있는 태스크의 수행이 종료될 때 까지 기다려야 하는 방식을 취하고 있다. 다만 하드웨어 인터럽트는 인터럽트 서비스 부문에서 처리되며, 모든 수행중인 동작보다 우선적으로 처리된다. 즉, 이전에 수행되던 태스크를 정지시키고 인터럽트 핸들러가 처리된다.

Pre-emption의 개념 (www.wikipedia.org)

Pre-emption as used with respect to operating systems means the ability of the operating system to preempt or stop a currently scheduled task in favour of a higher priority task. The scheduling may be one of, but not limited to, process or

I/O scheduling etc.

Non-preemptability arises, for instance, when handling an interrupt. In this case, scheduling is avoided until the interrupt is handled. Making a scheduler preemptible has the advantage of better system responsiveness and scalability.

The schedulers used in most modern operating systems, such as various flavours of Unix, can preempt user processes. This is called preemptive multitasking, and is in contrast to cooperative multitasking wherein a process "gives away" its time by utilizing kernel resources or by specifically calling a kernel routine to allow other processes time to run. Some operating systems' schedulers (including Linux as of the 2.6 series) have the ability to preempt a process while it is processing a system call as well (a preemptible kernel).

Linux, Unix, *BSD, Mac OS X, and Windows NT are all examples of operating systems that utilize preemptive multitasking; Netware, Windows for Workgroups, and Macintosh System 9 are all examples of cooperative multitasking operating systems.

nesC 의 특징은 아래와 같으며, 제약적인 프로그램 메모리와 램 메모리 때문에, 대부분의 경우 static(정적) 메모리 할당을 하여, 컴파일 시에 플랫폼에서 사용될 메모리의 크기를 결정한다.

[nes-si:]
Network Embedded System C Language
Supports TinyOS
Make applications for Network Embedded system
No dynamic memory allocation
Extension of C programming language
Efficient code for micro-controllers
Able to interact with old C code
Many C Programmer
C is little helpful for safe code & structuring applications

nesC와 C는 동일한 문법을 따르고 있고, nesC에 추가된 개념이 Component(컴포넌트)와 Interface(인터페이스) 이다.

Component는 일종의 부품으로써 소프트웨어를 개발할 때 사용하는 부품으로 사용된다. 기능을 모아 놓은 부품으로, RF 통신의 기능이 필요하다면 해당 Component를 사용하면 된다. Component 들을 연결하여 원하는 기능의 소프트웨어를 만들어 가는데, 사용하는 component 들 간에는 interface(인터페이스)를 사용하여 연결한다. Interface는 각 component 들이 제공하는 기능 (command, event)를 정의하는데 사용

된다.

Component는 아래와 같이 *.nc 파일의 형태로 존재하며, 세부적으로 Component는 Module, Configuration 으로 구분된다.

Module
xxxM.nc
Code file

Configuration
xxx.nc/oooC.nc
Wiring of components
Define wiring of Modules

Interfaces
xxx.nc
Contains only definition
Commands/Events

모듈(module)은 실제 실행되는 코드인 함수가 존재하는 파일이다. 확장명은 *.nc 이다. 모듈은 여러 개의 함수를 지원하는 파일이며, 인터페이스를 통해 해당 함수들이 구현된다.

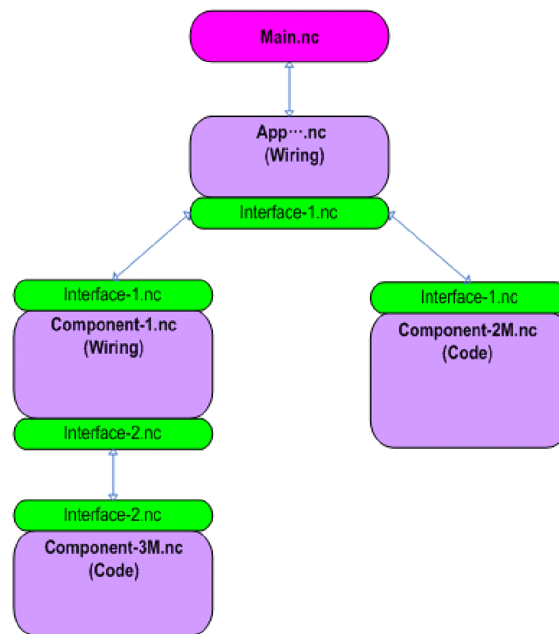
여기서 인터페이스를 통해 함수들이 제공(provides)되거나 사용(uses)되는 방법은 주의해서 이해해야 한다. 일반적으로 함수를 코드로 구현해서 제공하고, 원하는 곳에서 호출하여 사용하는 것이 C 언어에서 함수를 사용하는 방법이다. nesC에서는 일반 C와 다르게 그냥 함수를 구현하여 제공만 하지 않는다. 인터페이스라는 것은 일반 C와 다르게 함수를 두 가지로 분류한다. 커맨드(command)와 이벤트(event)이다.

모듈은 provides 키워드로 제공할 커맨드 함수를 정할 수 있다. 제공된 커맨드 함수는 다른 모듈에서 호출하여 사용할 수 있다. provides interface Leds 는 Lesd.nc 인터페이스 파일에 정의되어 있는 커맨드 함수들을 모듈 파일 내에서 구현하여, 다른 모듈에서 호출하여 사용할 수 있다.

반대로 모듈은 uses 키워드로 다른 모듈에서 제공하는 커맨드 함수를 호출하여 사용할 수 있다. 위에서 예를 든, provides interface Leds; 에 대응되는 코드는 uses interface Leds; 이다. uses 키워드는 인터페이스 Leds.nc 파일에 정의되어 있는 커맨드 함수를 호출할 수 있다는 의미이다.

uses 키워드는 한 가지 의미를 더 포함하고 있다. 이것은 nesC 프로그래밍 언어의 문법을 이해하는 초기에 가장 이해하기 힘든 부분이다. uses interface Leds; 는 Leds.nc 에 정의되어 있는 이벤트(event) 함수를 구현해야 함을 의미한다. 만약, Leds.nc 파일에 정의되어 있는 이벤트가 없는 경우도 있다.

provides 와 uses 의 사용방법은 아래 그림과 같이, 하나의 컴포넌트에서 provides 키워드로 인터페이스를 제공하고 다른 쪽 컴포넌트에서는 uses 키워드를 사용하여 해당 인터페이스에 정의된 커맨드와 이벤트를 사용한다. 사용한다는 의미는, 제공되는 커맨드를 호출할 수 있다는 것이며, 이벤트를 구현하는 것이다.



<그림> 컴포넌트와 인터페이스 연결 예

위 그림에서, App.nc 는 Component-1.nc 와 인터페이스 Interface-1.nc 로 연결되어 있다. Component-1.nc에서 provides interface Interface-1; 로 인터페이스를 제공하며, App.nc에서 uses interface Interface-1; 로 Interface-1을 사용한다.

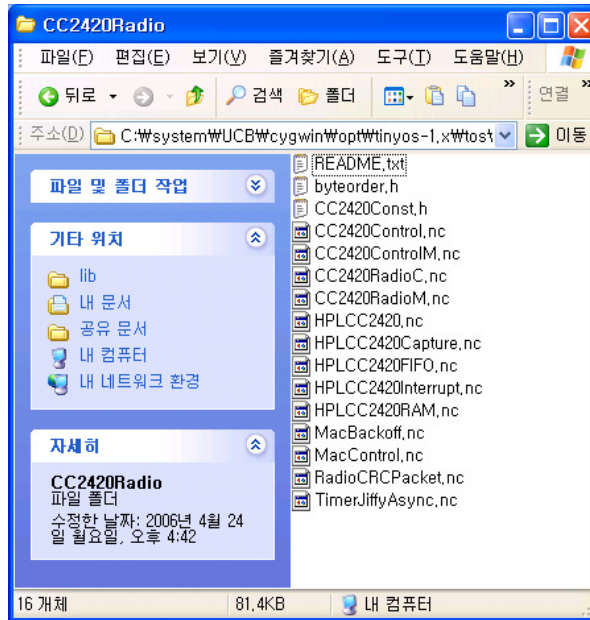
이를 통해서, App.nc에서는 Interface-1.nc 에 정의되어 있는 커맨드 함수를 호출할 수 있으며, Interfacd-1.nc 에 정의되어 있는 이벤트를 구현한다.

커맨드 함수를 호출하는 것은 일반 C 언어에서의 함수 호출과 동일하다. 이벤트는 기존의 C 언어에서의 함수 중 함수 포인터를 이용한 Callback 함수 호출과 동일한 기능을 제공한다. 하드웨어에서 인터럽트가 발생하면 그에 해당하는 이벤트 함수가 실행된다.

어떤 모듈에서 uses interface Timer; 코드를 이용하여 Timer 인터페이스를 사용하

면, Timer.nc에 정의되어 있는 커맨드 함수를 호출할 수 있으며, Timer.nc에 정의된 이벤트 함수를 구현해야 한다.

모듈과 인터페이스는 각 커맨드, 이벤트 함수를 구현하고 호출, 실행시키기 위한 방법에 대해 설명 하였다. nesC에는 이런 모듈을 그룹화 하여 추상화하기 위해서는 컨피규레이션 컴포넌트를 사용할 수 있다. 컨피규레이션은 여러 모듈을 서로 연결(wiring)하여 그룹화하고, 외부 컴포넌트가 기능을 사용할 수 있도록 해당 인터페이스를 제공(provides) 한다.



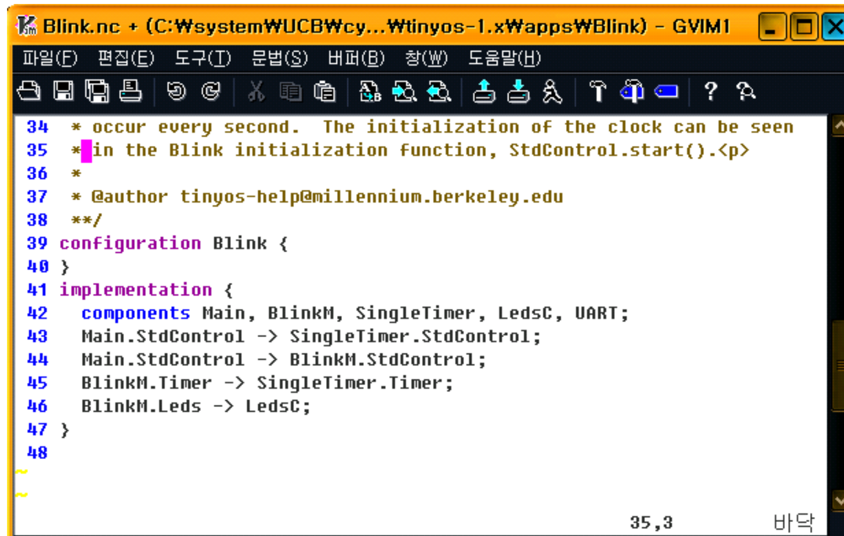
<그림> 컴포넌트, 인터페이스 파일

위 그림에서와 같이 컴포넌트인 컨피규레이션, 모듈과 인터페이스는 확장명이 *.nc이다. 파일이 어떤 종류인지는 파일 내부를 확인해 봐야한다. 반드시 정확하지는 않지만, CC2420RadioC.nc 는 컨피규레이션을 의미하고, CC2420RadioM.nc 는 모듈임을 의미한다.

주요 키워드

- Component
  - Configuration
  - Module
    - Provides
    - Uses
      - Interface
        - Command
        - Event
- Task
  - Post

Interface 파일은 Command / Event를 정의하고 있으며,
커맨드(command)는 인터페이스를 제공하는 컴포넌트에 구현되고,
이벤트(event)는 인터페이스를 사용하는 컴포넌트에 구현되어야 한다.



<그림> 컴포넌트, 컨피규레이션

위 그림은 컴포넌트의 예를 보여준다. 이 컴포넌트는 이름이 Blink인 컨피규레이션이다. 39줄처럼 선언되어 있고, 42줄과 같이 Main, BlinkM, SingleTimer, LedsC, UART 컴포넌트를 사용한다. 이 컴포넌트들은 컨피규레이션 또는 모듈이며, Main.nc, BlinkM.nc, SingleTimer.nc, LedsC.nc, UART.nc 의 파일로 존재한다. 이 파일들은 현재 디렉토리에 위치하거나, 또는 컴파일러가 사용하는 경로 중에 존재한다.

컨피규레이션에서는 컴포넌트의 이름은 명시적으로 표시되어 있지만, 어떤 것이 인터페이스인지에 대해서는 해당 파일에 명시적으로 표현되고 있지 않다. 따라서 컴포넌트들의 이름과 와이어링 상태를 확인하여, 해당 파일을 일일이 열어서 코드를 읽어보면 알 수 있다. 43줄 Main.StdControl 의 경우는 Main 은 42줄에 따라 컴포넌트임을 알 수 있다. 그러나 StdControl에 대한 선언은 없다. 컨피규레이션에서 컨포넌트와 인터페이스를 구분하는 “ . ”의 사용을 이해하면, 추가적인 파일을 읽지 않아도 쉽게 컴포넌트와 인터페이스를 구분할 수 있다. C 언어는 “ . ” 의 역할이 구조체의 멤버 변수를 구분할 때 사용한다. nesC에서도 비슷하게 컴포넌트에서 제공 또는 사용하는 인터페이스를 소속을 개념으로 파악하면 비슷한 의미를 갖는다. “ . ”은 소속의 의미를 갖는다고 할 수 있다. 컨피규레이션에서 “ . ” 앞쪽은 컴포넌트를 의미하고, 뒤쪽은 컴포넌트에 소속된 인터페이스를 의미한다. “ . ” 의 사용은 추후 설명할 모듈에서는 동일한 소속의 의미를 갖으나, 인터페이스에 소속된 커맨드 또는 이벤트 함수를 의미하기 때문에 코드 분석시에 주의해야 한다.

43줄 Main.StdControl -> SingleTimer.StdControl; 은 Main 컴포넌트와 SingleTimer 컴포넌트가 어떻게 연결(wiring) 되어 있는지를 표현한다. Main과 SingleTimer는 StdControl 이라는 인터페이스로 연결된 것이며, SingleTimer가 StdControl 인터페이스를 제공하고 Main이 StdControl 인터페이스를 사용한다. SingleTimer.nc 파일에는 provides interface StdControl; 이라는 코드가 포함되어 있고, Main.nc 에는 uses interface StdControl; 이라는 코드가 포함되어 있다. 따라서 StdControl.nc 파일에 정의되어 있는 커맨드 함수들이 SingleTimer.nc 파일에 구현된다. StdControl 은 이벤트 함수가 없지만, 만약에 있다면, 해당 이벤트들이 Main.nc에 구현될 것이다.

“ -> ”(wiring)의 의미는 제공(provides)와 사용 (uses)의 관계를 표현한다. 제공이라는 의미와는 다르게 화살표 방향이 제공과는 반대로 되어있다. 이 기호의 의미를 해석할 때 실제 커맨드 함수가 구현된 코드가 어느 곳에 있는지 코드의 위치를 지시하고 있다고 해석해야 한다. 와이어링은 “ -> ”과 “ = ”, 두가지 기호가 있다. “ = ” 기호에 대해서는 추후 설명한다.

46줄은 인터페이스를 생략한 경우다. 42줄처럼 LedsC는 컴포넌트라는 것을 알 수 있고, 일반적인 경우에는 컴포넌트 뒤에 “ . ” 이 찍히고 이어서 인터페이스 이름이 적혀있어야 하지만, 여기서는 생략되어 있다. 컴포넌트들은 동일한 인터페이스로 연결되어야 하기 때문에 생략하여도 컴파일러는 해당 인터페이스를 찾을 수 있다.

```
BlinkM.nc (C:₩system₩UCB₩cygw...₩tinyos-1.x₩apps₩Blink) - GVIM1
파일(F) 편집(E) 도구(T) 문법(S) 버퍼(B) 창(W) 도움말(H)
35  **/
36 module BlinkM {
37   provides {
38     interface StdControl;
39   }
40   uses {
41     interface Timer;
42     interface Leds;
43     interface StdControl as ByteControl;
44     interface ByteComm;
45   }
46 }
47 implementation {
48
49   /**
50    * Initialize the component.
51    *
52    * @return Always returns <code>SUCCESS</code>
53    **/
54   command result_t StdControl.init() {
55     call Leds.init();
56     call ByteControl.init();
57     return SUCCESS;
58   }
59
                                                59,0-1        39%
```

```
BlinkM.nc (C:₩system₩UCB₩cygw...₩tinyos-1.x₩apps₩Blink) - GVIM1
파일(F) 편집(E) 도구(T) 문법(S) 버퍼(B) 창(W) 도움말(H)
65    **/
66   command result_t StdControl.start() {
67     // Start a repeating timer that fires every 1000ms
68 //     call Leds.redOff();
69 //   call Leds.yellowOff();
70 //   call Leds.greenOff();
71     call Leds.set(0x7);
72     call ByteControl.start();
73     return call Timer.start(TIMER_REPEAT, 1000);
74   }
75
76   /**
77    * Halt execution of the application.
78    * This just disables the clock component.
79    *
80    * @return Always returns <code>SUCCESS</code>
81    **/
82   command result_t StdControl.stop() {
83     return call Timer.stop();
84   }
85
86
87   /**
88    * Toggle the red LED in response to the <code>Timer.fired</code> eve
nt.
                                                86,0-1        72%
```

BlinkM.nc (C:₩system₩UCB₩cygw...₩tinyos-1.x₩apps₩Blink) - GVIM1

파일(F) 편집(E) 도구(T) 문법(S) 버퍼(B) 창(W) 도움말(H)

```
89     *
90     * @return Always returns <code>SUCCESS</code>
91     **/
92    event result_t Timer.fired()
93    {
94      call ByteComm.txByte(0x41);
95      call Leds.redToggle();
96      return SUCCESS;
97    }
98
99
100   async event result_t ByteComm.rxByteReady(uint8_t data, bool error, u
   int16_t strength){
101     return SUCCESS;
102   }
103
104
105   async event result_t ByteComm.txByteReady(bool success){
106     return SUCCESS;
107   }
108
109   async event result_t ByteComm.txDone(){
110     return SUCCESS;
111   }
112 }
                                                    112,1          바닥
```

<그림> 컴포넌트, 모듈

모듈은 인터페이스를 제공 또는 사용하며, 커맨드와 이벤트가 실행될 수 있도록 구현한다. 즉, C 언어의 함수처럼 실행되는 코드를 제공한다.

36줄은 BlinkM이라는 모듈을 정의하고 있으며, 이 모듈은 인터페이스 StdControl 을 제공하고, Timer, Leds, StdControl, ByteComm 인터페이스를 사용한다. 36줄부터 45줄까지는 모듈 BlinkM에 대해 속성을 선언한 것으로, 제공하는 인터페이스의 커맨드들이 구현될 것이며, 다른 컴포넌트에서 제공하는 인터페이스들을 사용하여 호출하고 이벤트를 사용할 것을 표현한다.

37, 38줄은 StdControl 인터페이스를 제공(provides)하고 있음 나타내며, 다른 컴포넌트에서는 이 인터페이스를 사용(uses)할 수 있다. 제공하는 StdControl.nc 인터페이스 파일에 정의된 커맨드 함수들이 BlinkM.nc 파일 하단에 코드로 구현된다. StdConrol.nc 파일을 확인해 보면, command StdControl.init(), command StdControl.start(), command StdControl.stop(), 세 가지 커맨드를 정의하고 있다.

40에서 45줄 까지는 사용(uses)하는 인터페이스를 의미한다. 사용하는 인터페이스는 다른 컴포넌트에서 제공하는 인터페이스의 커맨드 함수를 호출하여 실행할 수 있고, 그 인터페이스에 이벤트 함수가 정의 되어 있다면, 이벤트 함수를 구현하여 실행시킨다는 의미를 갖고 있다.

41줄의 Timer.nc 인터페이스 파일은 commnad Timer.start(type, sec), command

Timer.stop()의 두 커맨드 함수와 event Timer.fired()의 한 개 이벤트 함수가 정의 되어 있다. BlinkM 모듈에서는 다른 컴포넌트에서 제공하는 Timer 인터페이스의 커맨드 함수를 호출하여 실행 시킬 수 있으며, Timer.fired()를 반드시 구현하여 해당 이벤트 발생시 실행할 수 있도록 해야 한다. 다만, 이벤트 발생시에 실행할 것이 없으면, 함수의 중괄호 안을 공란으로 비워두면 된다.

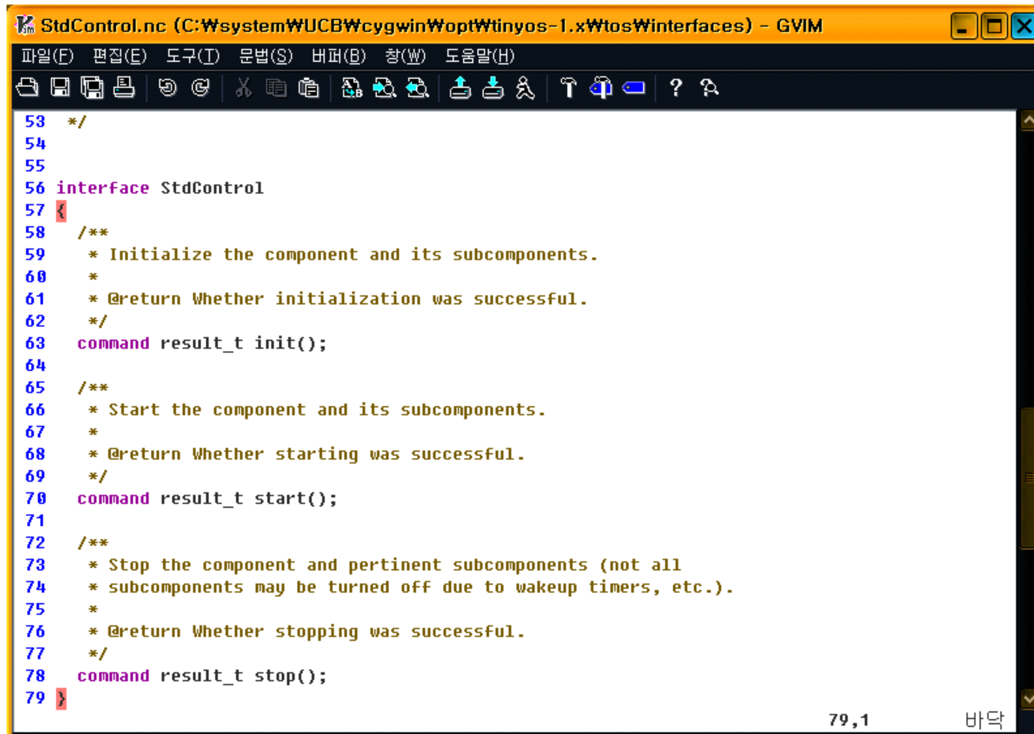
BlinM 컴포넌트에서 사용되고 있는 다른 인터페이스인 Leds, ByteControl, ByteComm 도 동일한 방법으로 커맨드 함수, 이벤트 함수를 사용한다.

43줄의 " as " 키워드는 하나의 컴포넌트 안에서 동일한 인터페이스가 여러 번 다른 목적으로 사용될 때, 중복사용으로 인한 혼동을 막기 위해 사용하는 키워드이다. 실제는 StdControl 이지만, 38줄의 인터페이스와 구분하기 위해 as로 사용 이름만 변경한 것이다.

사용하는 인터페이스의 커맨드 함수의 구현된 코드를 보기 위해서는, 상위 컨피규레이션에서의 연결 관계를 찾아 실제 코드가 위치를 파악해야 한다. 실제 실행되는 코드를 찾기 위해서는 상위 컨피규레이션 파일을 확인해야 한다. 41줄은 Blin.nc 컨피규레이션을 통해 SingleTimer.nc에서 제공되는 Timer 인터페이스를 사용하고 있다.

```
Timer.nc (C:₩system₩UCB₩cygwin₩opt₩tinyos-1.x₩tos₩interfaces) - GVIM
파일(F)  편집(E)  도구(T)  문법(S)  버퍼(B)  창(W)  도움말(H)
40  */
41 includes Timer; // make TIMER_x constants available
42 interface Timer {
43 
44   /**
45    * Start the timer.
46    * @param type The type of timer to start. Valid values include
47    *  'TIMER_REPEAT' for a timer that fires repeatedly, or
48    *  'TIMER_ONE_SHOT' for a timer that fires once.
49    *  @param interval The timer interval in <b>binary milliseconds</b> (1/1024
50    *  second). Note that the
51    *    timer cannot support an arbitrary range of intervals.
52    *    (Unfortunately this interface does not specify the valid range
53    *    of timer intervals, which are specific to a platform.)
54    *  @return Returns SUCCESS if the timer could be started with the
55    *    given type and interval. Returns FAIL if the type is not
56    *    one of TIMER_REPEAT or TIMER_ONE_SHOT, if the timer rate is
57    *    too high, or if there are too many timers currently active.
58    */
59   command result_t start(char type, uint32_t interval);
60 
61   /**
62    * Stop the timer, preventing it from firing again.
63    * If this is a TIMER_ONE_SHOT timer and it has not fired yet,
64    * prevents it from firing.
65    * @return SUCCESS if the timer could be stopped, or FAIL if the timer
66    * is not running or the timer ID is out of range.
67    */
68   command result_t stop();
69 
70   /**
71    * The signal generated by the timer when it fires.
72    */
73   event result_t fired();
74 }
75 
                                                        43,0-1        바닥
```

<그림> 인터페이스



<그림> 인터페이스

인터페이스는 컴포넌트와 컴포넌트간의 커맨드와 이벤트의 제공, 구현을 명시하는 역할을 한다. 인터페이스 파일에 정의된 커맨드와 이벤트 함수만이 컴포넌트 간에 연결될 수 있으며, 인터페이스에 정의되지 않은 함수는 해당 파일에서만 개별적으로 사용된다. 즉, 개발자가 어떤 기능을 구현하여 외부 다른 컴포넌트에 제공하고자 한다면, 인터페이스 파일을 구현하고, 그에 맞도록 컴포넌트를 작성해서 제공해야 한다. 반대로, 어떤 기능을 사용하고자 할 때는, 그 기능이 구현된 컴포넌트에서 제공하는 인터페이스를 사용(uses) 한다는 것을 명시적으로 표현해야 한다.

위 그림 중, Timer.nc 인터페이스는 41줄에서 헤더파일인 Timer.h를 include 하고 있다. interface Timer는 command result_t start(char type, uint32_t interval); 59줄, command result_t result_t stop(); 68줄, event result_t rired(); 73줄을 정의하고 있다. 여기서 command 들은 Timer 인터페이스를 제공하는 컴포넌트에서 구현되며, event 는 Timer 인터페이스를 사용하는 컴포넌트에서 구현된다.

```
module PTimer{
      provides interface Timer;
}

implementation{

      command result_t Timer.start(){
            dosomething();
            dosomething();
            return SUCCESS;
      }

      command result_t Timer.stop(){
            dosomething();
            dosomething();
            return SUCCESS;
      }

}
```

위 예제와 같이 PTimer 모듈은 Timer 인터페이스를 제공(provides)하고, Timer.nc 인터페이스 파일에 정의된 command인 Timer.start()와 Timer.stop()을 반드시 제공해야 한다.

```
module UTimer{
      provides interface GoUse;
      uses interface Timer;
}

implementation{

      command result_t GoUse.dosomething(){
            call Timer.start();
            call Timer.stop();
            return SUCCESS;
      }

      command result_t Timer.fired(){
            dosomething();
            dosomething();
            return SUCCESS;
      }

}
```

반대로 Timer 인터페이스를 사용(uses)하는 모듈에서는 위 예제와 같이 Timer.nc 에 정의되어 있는 command 함수를 call 키워드를 이용해서 호출, 실행 시킨다. 이 command 함수들은 연결(wiring)되어 있는 컴포넌트에 구현되어 있다. 연결(wiring)을 확인하가 위해서는 UTimer의 연결 상태를 표현하고 있는, 컨피규레이션 파일을 확인해야 한다.

```
configuration UApp{
}

implementation{
      components Main, UTimer, PTimer;

      Main.StdControl -> UTimer.StdControl;
      Main.StdControl -> PTimer.StdControl;

      UTimer.Timer -> PTimer.Timer;

}
```

configuration 연결(wiring) 상태는 상위 컨피규레이션을 확인해야 한다. 위 내용처럼 UTimer에서 사용(uses)하는 Timer 인터페이스는 PTimer에서 제공(provides)되었다는 연결(wiring)을 해주어야 한다.

```
task void sendData() { // send sensorReading
  adcPacket.data = sensorReading;
  call Send.send(&adcPacket, sizeof adcPacket.data);
  return SUCCESS;
}

event result_t ADC.dataReady(uint16_t data) {
  sensorReading = data;
  post sendData();
  return SUCCESS;
}
```

<그림> 태스크

태스크의 정의는 위 예제처럼 task 키워드로 정해주며, 리턴타입은 void로 해준다. 태스크의 실행을 위해서는 post 키워드를 사용하여 호출하며, 이 태스크의 실행은 바로 실행되지 않고, 일반 command 와 함수들이 실행된 후로 순서가 뒤쪽으로 스케줄링 된다. 이런 태스크의 실행은 리소스가 한정된 무선 플랫폼에서 중요한 의미를 갖는다. 빠르게 실행되는 스케줄이 정해진 동작에 비해, 스케줄에 자유로운 태스크들을 post 키워드를 통해 플랫폼 리소스를 효율적으로 할당할 수 있다.

atomic 키워드는 atomic { } 범위 안에서는 하드웨어 인터럽트가 발생되지 않도록 정의하는 것이며, 이벤트(event) 함수에서 전역 변수를 사용할 때, 컴파일러가 경고(warning)를 표시한다. 이 경우는 컴파일러가 변수의 race condition을 경고 하는 것이다. 이벤트 함수는 언제나 발생할 수 있기 때문에, 여러 이벤트 함수에서 동일한 변수를 처리하면, 변수의 변경 순서를 보장할 수 없다. 만약, 개발자가 변수의 race condition을 확신하고 여러 개 이벤트의 실행에 따른 변수 변화를 허용한다면, 변수 선언시에 no_race 라는 키워드로 컴파일러 경고(warning)을 제거할 수 있다.

4. VMPlayer 개발 환경

VMPlayer를 이용한 TinyOS 개발 환경 구축은 Linux, Windows, MACOS 등의 다양한 플랫폼에서 일관된 개발 환경을 구축할 수 있다는 장점 때문에, 많이 사용되고 있다. 그러나 Linux / Unix 개발 환경에 익숙하지 않은 개발자들에겐 몇 가지 기본적인 사용방법이 제공되어야 한다. 이 장에서는 VMPlayer(2.0 이상)과 TinyOS 개발환경 Fedora 이미지 (Kmote Release) 소프트웨어를 이용한 설치, 사용방법에 대해 설명한다.

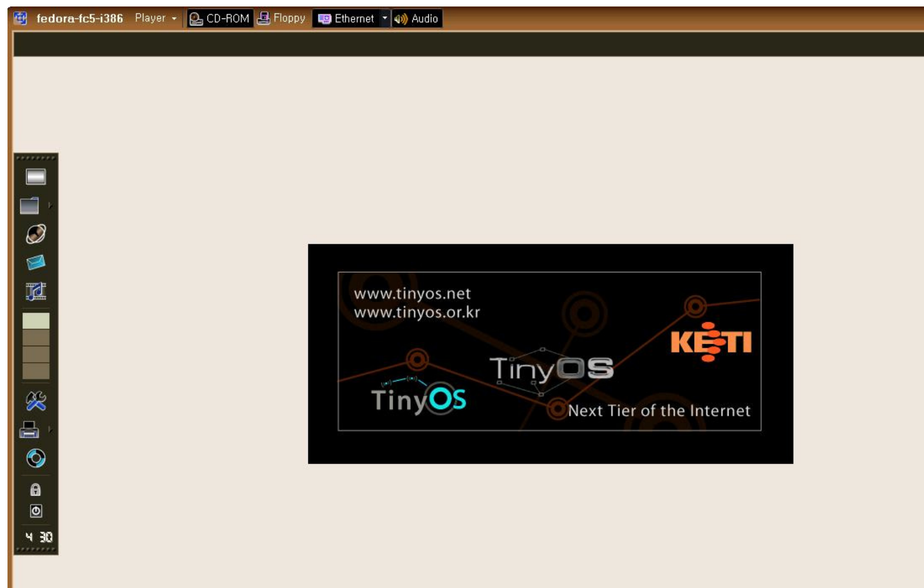
VMPlayer를 설치하고, TinyOS 개발환경 Fedora 이미지를 하드디스크에 복사한다. VMPlayer를 실행하고, virtual machine configuration file의 위치를 지정한다.

위 같은 창이 나타나면, Create를 선택한다. 일반적으로 컴퓨터가 변경되면 나타난다. 그 외의 질문들에 대해서는 최소 또는 확인을 선택한다.

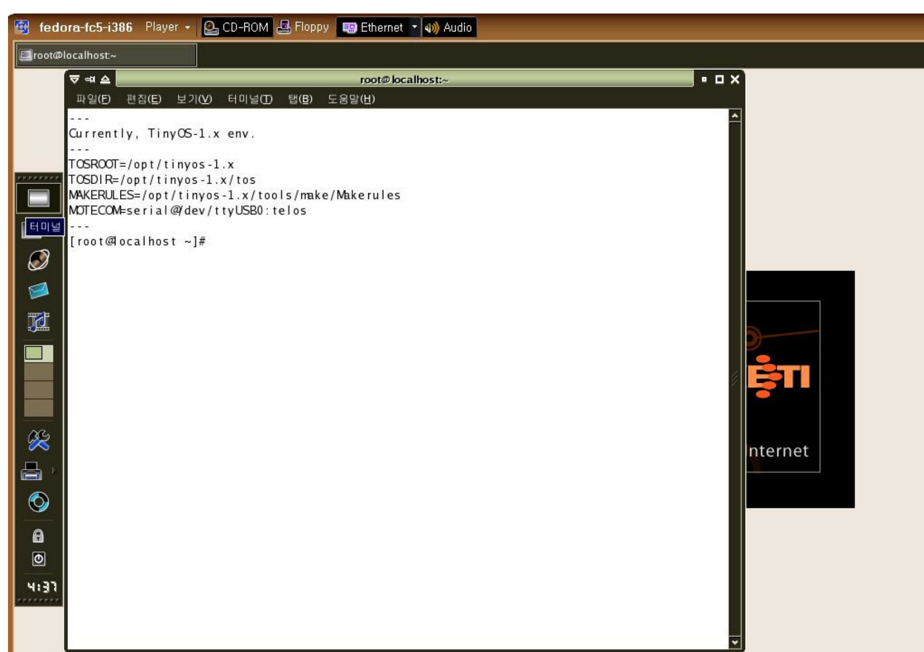
VMPlayer가 리눅스를 부팅하는 화면이 지나가면 로그인 issue 메시지를 출력하며, 로그

인을 기다린다. issue 메시지를 살펴보면, root / tinyos 로 로그인할 수 있음을 알 수 있다. VMPlayer 창으로 들어가려면 마우스로 클릭하고, VMPlayer에서 빠져 나오려면 ALT-CTRL 키를 동시에 누른다. 로그인 후에는 tinyos를 입력하여 로그인 메시지를 볼 수 있다.

일반적으로는 startxfce4를 실행하여 가벼운 윈도우를 실행시키며, gnome 환경을 실행시키기 위해서는 startx를 쉘 프롬프트에서 실행하면 된다. 앞으로는 XFCE 환경을 기본으로 설명한다.

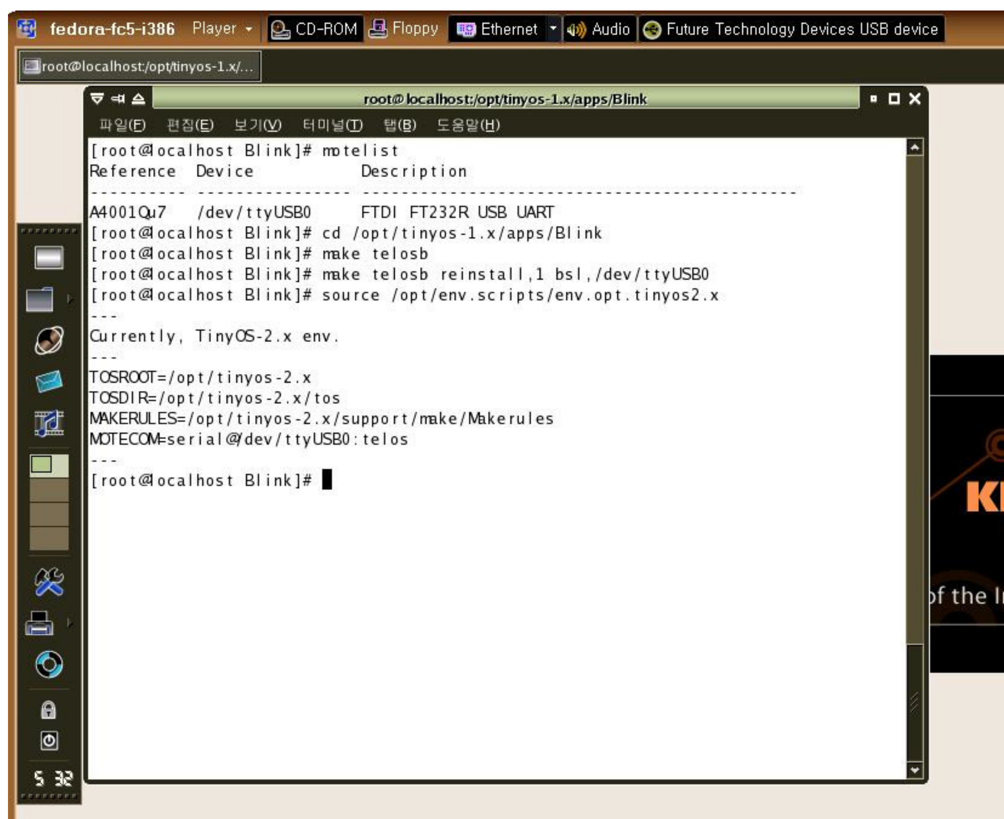


위와 같은 그림은 윈도우가 실행된 상태이며, 좌측 최상단의 아이콘을 이용해서 터미널을 실행시킨다.



초기에는 TinyOS-1.x 개발환경으로 설정되어 있다. 위처럼 VMPlayer가 동작하고 키보드의 제어권이 VMPlayer으로 넘어간 상태에서 Kmote를 USB 포트에 연결하면, 아래 그림처럼 VMPlayer의 제목표시줄에 Future Technology Devices USB Device라는 탭이 생긴다. 이 버튼은 토글되며, 눌려 있는 경우는 VMPlayer가 제어권을 갖고, 안 눌려 있는 경우는 Windows가 제어권을 갖는다.

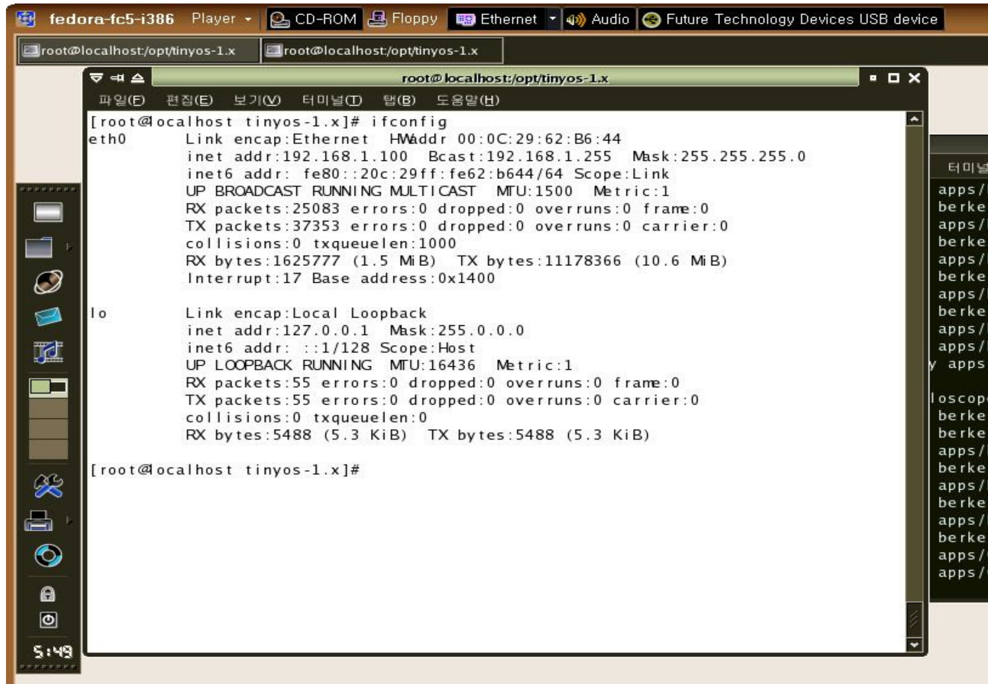
아래 그림들은 Fedora Linux 상태에서 TinyOS 개발을 위해 입력한 몇 개의 명령어 예제이다.



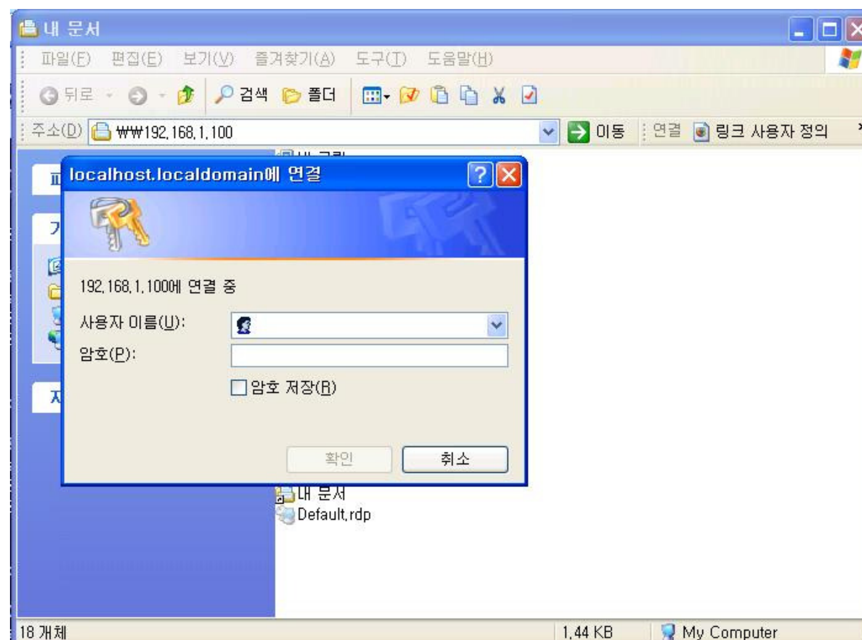
motelist 는 현재 어떤 USB 포트에 mote가 연결되어 있는지를 확인할 수 있으며, make telosb는 telosb 플랫폼에 맞게 애플리케이션 코드를 컴파일 하라는 명령이며, make telosb reisntall,1 bsl,/dev/ttyUSB0 는 USB0 포트로 컴파일된 telosb 실행 바이너리 이미지를 다운로드 하라는 명령이다. source /opt/env.scripts/env.opt.tinyos2.x 는 tinyos-2.x 개발 환경으로 변경하라는 명령이다.

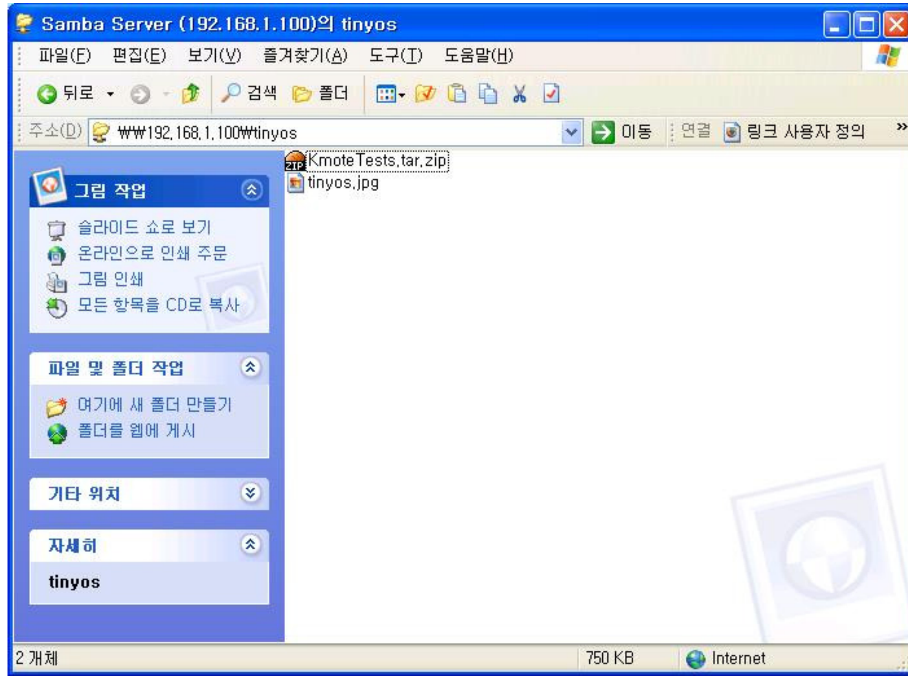
VMPlayer를 사용하여, 윈도우에서 작업할 경우에 가장 문제시 되는 것이 윈도우(호스

트 OS)와 리눅스(게스트 OS) 간의 파일 공유이다. Samba를 이용하여, 네트워크 공유로 윈도우와 리눅스간의 파일 공유하는 방법은 윈도우즈 탐색기에서 해당 주소를 \ \ 192.168.1.100 입력하면 된다. 리눅스의 IP 주소를 얻기 위해서는 아래 그림처럼, 프롬프트 쉘에서 ifconfig 명령을 입력한다.



윈도우에서 \ \ 192.168.1.100을 입력하면, 아래와 같은 창이 나타난다. 이에 ID, Password를 tinyos / tinyos 로 입력하면, 리눅스의 /share 디렉토리에 직접 접근할 수 있다.




Samba Server (192.168.1.100)의 tinyos
파일(F) 편집(E) 보기(V) 즐겨찾기(A) 도구(T) 도움말(H)
뒤로
검색
폴더
주소(D)
₩₩192.168.1.100₩tinyos
이동
연결
링크 사용자 정의
그림 작업
슬라이드 쇼로 보기
온라인으로 인쇄 주문
그림 인쇄
모든 항목을 CD로 복사
파일 및 폴더 작업
여기에 새 폴더 만들기
폴더를 웹에 게시
기타 위치
자세히
tinyos
KmoteTests.tar.zip
tinyos.jpg
2 개체
750 KB
Internet

5. 그 외 사항들

· TinyOS 개발 환경 DVD

http://www.tinyosmall.co.kr/shop/shopdetail.html?brandcode=016002000001&search=&sort=brandname

· Xubuntos의 경우는 아래 링크 참조

http://sing.stanford.edu/klueska/running_xubuntos_vm.html

· VMPlayer 1.03 은 USB 사용시에 에러 발생함

Fedora ID - root / PASSWORD - tinyos

주요 명령어 (shell command)

cd , ls, ls -al, mkdir, rmdir, cp, env, env | grep TOS, source, grep -rn tinyos *, find ./ -name good, clear, exit, printenv

윈도우즈에서 공유디렉토리 설정

공유 directory에 tinyos-1.x source tree copy

ifconfig

mount.cifs //192.168.1/1/tinyos /opt/mnt.win

umount /opt/mnt.win

/etc/profile.d/tinyos.sh 생성

export TOSROOT=/opt/mnt.win/tinyos-1.x

export TOSDIR=$TOSROOT/tos

export MAKERULES=$TOSROOT/tools/make/Makerules

echo $TOSDIR

echo $MAKERULES

export MOTECOM=serial@/dev/ttyUSB0:telos

윈도우즈에서 생성한 파일(스크립트) 가 동작을 안할 경우

envT1 파일을 공유중인 윈도우즈 디렉토리에 만들었을 경우

반드시 dos2unix envT1 이라는 명령으로 unix 파일 형태로 변환 시켜줘어야 함

변환하지 않았을 때에는 source envT1 이라고 해도 제대로 동작을 안함
윈도우상에서 envT1을 만들고 리눅스 쉘에서 dos2unix 명령을 한번 실행해 주면 됨
make telosb reinstall.2 라는 명령을 내렸을때, make가 동작을 안하면

/opt/tinyos-1.x/tools/make 안의 모든 파일들의 모드를 777로 변경
예) chmod 777 *

채널, 그룹아이디, RF 세기 바꾸기
기본적으로는 /opt/tinyos-1.x/tools/make/Makelocal 을 고칩니다.
Makelocal 내용을 아래와 같이 바꾸면 됩니다.
또는 /opt/tinyos-1.x/apps/Blink 내에 있는 Makefile 에 추가해 줘도 됩니다.

DEFAULT_LOCAL_GROUP=0x01
PFLAGS += -DCC2420_DEF_CHANNEL=12
PFLAGS += -DCC2420_DEF_RFPOWER=31
PFLAGS += -DDEFAULT_BAUDRATE=38400
채널은 11~26 / RF Power는 1~31

컴파일 및 다운로드 명령
make telosb
make telosb reinstall
make telosb install
make telosb reinstall.1
make telosb install.1 bsl,/dev/ttyUSB0

Windows에서 공유 디렉토리 생성
ifconfig 로 현재 windows의 IP 확인
mount.cifs //192.168.12.1/tinyos /mnt
파일의 경우 chmod 777 , dos2unix 로 상태 변경 필요

삼바 이용방법
ifconfig 로 현재 Linux IP 확인

윈도우즈의 폴더 주소표시줄에 \ \ 192.168.111.11 과 같이 주소 입력

id : tinyos / password : tinyos

문제 발생시

vi /etc/samba/smb.conf 에 있는 IP Allow 부분에

ifconfig 에서 확인하 IP가 접근 가능하도록 설정되었는지 확인

----

nesDoc 만드는 방법

/opt/tinyos-1.x/apps 내부의 application 중에 한 곳에서

make telosb docs

cd /opt/tinyos-1.x/doc/nesdoc/telosb

firefox /opt/tinyos-1.x/doc/nesdoc/telosb/index.html

OscilloscopeRF 의 동작센서를 TelosB에 있는 빛센서로 변경 하기

InternalTempC.nc 컴포넌트 file 찾는 방법 :

find {찾기시작할 디렉토리} -name {찾고자하는 파일이름}

예, find / -name *InternalTempC.nc*

HamamatsuC.nc 컴포넌트로 변경한다.

Java 실행 방법

- SerialForwarder

  java net/tinyos/sf/SerialForwarder &

  실행되는 윈도우 창에서 serial@/dev/ttyUSB0:telos 로 자신의 시스템에 맞게 정확히 변경해 준다

- Oscilloscope

  java net/tinyos/oscope/oscilloscope &

화면에 그래프가 안 그려지면, scroll 또는 zoom out Y, 버튼을 눌러준다.

Surge 실행 방법

java net/tinyos/surge/MainClass 0x7d &

Samba를 이용한 windows와 Linux 간의 파일 공유

사용법

ifconfig 를 쳐서, Linux의 IP 확인

windows 탐색기에서 \ \ 192.168.11.1 이라고 치면 공유 폴더가 보임

id : tinyos , password : tinyos

문제 발생시

vi /etc/samba/smb.conf 에 있는 IP Allow 부분에

ifconfig 에서 확인하여 현재 작동중인 IP가 접근 가능하도록 설정되었는지 확인

Linux에서 Windows를 마운팅 하는 방법 (되도록이면 위에 설명한 samba 사용을 권장합니다.)

mount.cifs //192.168.11.1/tinyos /mnt

TinyOS 2.0 설치 방법

http://www.tinyos.net/tinyos-2.x/doc/html/install-tinyos.html

TinyOS의 사용을 위해서는 크게 두가지 단계로 설치를 진행하여야 한다. 첫 번째는 개발환경 (컴파일러, 리눅스 환경 등)의 설치이고, 두 번째는 TinyOS 소스코드 설치이다.

TinyOS 개발환경으로는 Linux 실행환경을 위한 Cygwin 설치, 호스트 소프트웨어를 위한 JAVA 설치, Mote용 소프트웨어를 위한 컴파일러 설치, 환경변수 설정이 필요하다. TinyOS 코드 설치는 CVS(http://en.wikipedia.org/wiki/Concurrent_Versions_System) 서버로부터 소스코드를 다운받아 설치하는 것이 필요하다.

1. Java 1.5 JDK 설치

Download and install Sun's 1.5 JDK from http://java.sun.com

TinyOS 1.x의 Java 실행환경을 위한 Javax Comm API 는 www.tinyos.re.kr 의 files 게시판에서 찾아 설치한다.

2. Linux 환경 설치 - Cygwin 설치 (Linux 사용하는 경우 필요 없음)

http://www.tinyos.net/dist-1.2.0/tools/windows/cygwin-1.2a.tgz

3. Mote 용 컴파일러 설치

TI MSP430 용 컴파일러

Cygwin 또는 Linux Shell에서 아래와 같이 실행하여 설치

rpm -ivh msp430tools-base-0.1-20050607.cygwin.i386.rpm

만일 os 또는 dependancy 문제가 발생하면 아래와 같은 옵션으로 실행

rpm -ivh --ignoreos --nodeps --force msp430tools-base-0.1-20050607.cygwin.i386.rpm

아래 rpm 파일들을 순서대로 설치 (Windows 경우 5개, Linux 경우 7개)

Tool	Windows/Cygwin	Linux
base	msp430tools-base-0.1-20050607.cygwin.i386.rpm	msp430tools-base-0.1-20050607.i386.rpm
python tools	msp430tools-python-tools-1.0-1.cygwin.noarch.rpm	msp430tools-python-tools-1.0-1.noarch.rpm
binutils	msp430tools-binutils-2.16-20050607.cygwin.i386.rpm	msp430tools-binutils-2.16-20050607.i386.rpm
gcc	msp430tools-gcc-3.2.3-20050607.cygwin.i386.rpm	msp430tools-gcc-3.2.3-20050607.i386.rpm
libc	msp430tools-libc-20050308cvs-20050608.cygwin.i386.rpm	msp430tools-libc-20050308cvs-20050608.i386.rpm
jtag	Not yet available	msp430tools-jtag-lib-20031101cvs-20050610.i386.rpm
gdb	Not yet available	msp430tools-gdb-6.0-20050609.i386.rpm

4. TinyOS Toolchain (nesC, tinyos-tools)

rpm -Uvh nesc-1.2.8a-1.cygwin.i386.rpm

또는 rpm -Uvh --ignoreos --nodeps --force nesc-1.2.8a-1.cygwin.i386.rpm 로 설치

nesC : nesc-1.2.8a-1.cygwin.i386.rpm

tinyos-tools : tinyos-tools-1.2.3-1.cygwin.i386.rpm

5. Graphviz 설치

nesC document를 생성하기 위한 Graphviz를 http://www.graphviz.org/Download..php 에서 다운로드하여 설치한다.

6. 환경변수 설정

cygwin, Linux shell 에서는 export, unset 명령으로 환경변수를 추가, 제거할 수 있다. TinyOS의 개발환경에 작성되어 있는 Makefile은 아래와 같은 환경변수가 설정되어야 컴파일 등의 작업을 수행할 수 있다.

TinyOS 2.0 의 경우는 아래와 같이 환경 변수가 설정되어야 한다.

TOSROOT=/opt/tinyos-2.x

TOSDIR=$TOSROOT/tos

MAKERULES=$TOSROOT/support/make/Makerules

CLASSPATH=`cygpath -w $TOSROOT/support/s아/java/tinyos.jar`

CLASSPATH="$CLASSPATH;."

PATH=/opt/msp430/bin:$PATH

각 환경변수를 추가하기 위해서는 아래와 같이 export 명령을 이용해 수행해야 한다.

export TOSROOT=/opt/tinyos-2.x

env | grep TOS 같은 명령으로 현재 설정되어있는 환경변수를 확인할 수 있다.

환경 변수를 하나씩 매번 등록하는 것은 비효율적이므로, 하나의 파일, 예를 들면 envT2 라는 파일에 6줄의 환경 설정 내용을 작성해 놓고, shell 상에서 source envT2 라고 실행하면 환경 변수가 모두 적용된다.

TinyOS 1.0의 경우는 TOSROOT=/opt/tinyos-1.x,

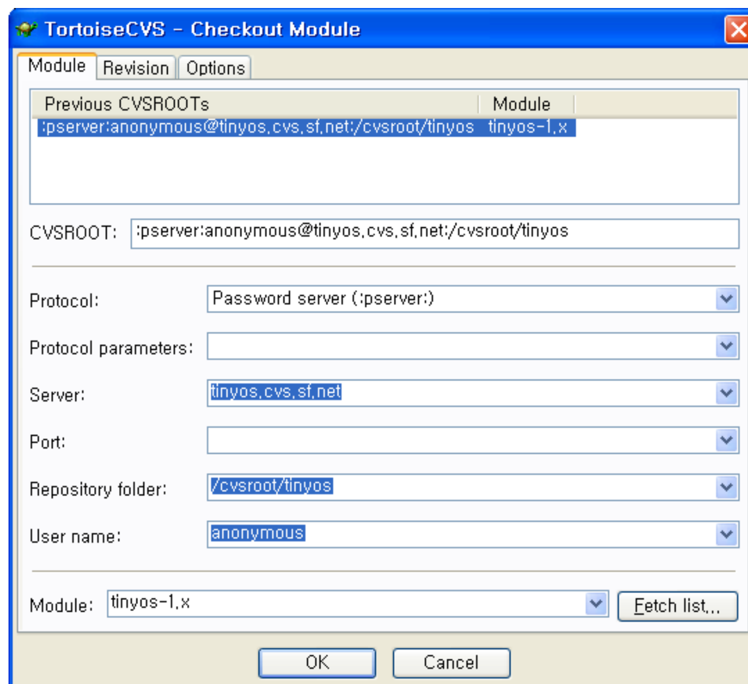
MAKERULES=/opt/tinyos-1.x/tools/make/Makerules 로 변경해 주면 된다. 그리고 해당하는 JAVA 경로로 변경하면 된다.

[TinyOS Source code Download]

TinyOS 환경변수 설정에 맞게 TinyOS source code가 위치해야 한다. 일반적인 방법으로는 TinyOS source tree rpm 파일을 다운로드 받아 설치하지만, 항상 업데이트 되는 상태를 유지하기 위해서는 cvs 도구를 이용하여 cvs 서버로부터 다운로드 받고, 필요시마다 업데이트 하는 것이 좋다.

CVS 클라이언트는 www.tortoisecvs.org 를 사용하는 것이 간편하다.

설치후 바탕화면에서 마우스 오른쪽 버튼을 누르면 아래와 같이 pop-up 메뉴에 CVS checkout 이라는 항목이 생성된다.



Protocol은 pserver, Server는 tinyos.cvs.sourceforge.net (또는 tinyos.cvs.sf.net), Repository folder는 /cvsroot/tinyos, user name은 anonymous, module은 tinyos-1.x 또는 tinyos-2.x 로 적어주면, tinyos-1.x / tinyos-2.x 소스코드를 다운로드 받을 수 있다.