# TinyOS and nesC

## TinyOS

TinyOS is an event-driven operating system designed for sensor network nodes that have very limited resources (e.g., 8K bytes of program memory, 512 bytes of RAM). It is an open-source operating system designed for wireless embedded sensor networks. It features a component-based architecture which enables rapid innovation and implementation while minimizing code size as required by the severe memory constraints inherent in sensor networks. [1] Sensor networks and embedded systems needs such operating systems, applications and languages which can efficiently utilize their limited resources and work better in these minimum resourced areas. TinyOS's component library includes network protocols, distributed services, sensor drivers, and data acquisition tools. All of which can be used as-is or be further refined for a custom application. TinyOS's event-driven execution model enables fine-grained power management yet allows the scheduling flexibility made necessary by the unpredictable nature of wireless communication and physical world interfaces. [1]

TinyOS defines a number of important concepts that are expressed in nesC. First, nesC applications are built out of components with well-defined, bidirectional interfaces. Second, nesC defines a concurrency model, based on tasks and hardware event handlers, and detects data races at compile time. [6]

### Components

***Specification***

- A nesC application consists of one or more components linked together to form an executable. A component provides and uses interfaces. These interfaces are the only point of access to the component and are bi-directional. An interface declares a set of functions called commands that the interface provider must implement and another set of functions called events that the interface user must implement. For a component to call the commands in an interface, it must implement the events of that interface. A single component may use or provide multiple interfaces and multiple instances of the same interface.
- ***Implementation***
  There are two types of components in nesC: modules and configurations. Modules provide application code, implementing one or more interface. Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by others. This is called wiring. Every nesC application is described by a top-level configuration that wires together the components inside.
  NesC uses the filename extension ".nc" for all source files -- interfaces, modules, and configurations. Please see TinyOS Coding and Naming Conventions [7] for more information on naming conventions.

### Concurrency Model

TinyOS executes only one program consisting of selected system components and custom components needed for a single application. There are two threads of execution:

tasks and hardware event handlers. Tasks are functions whose execution is deferred. Once scheduled, they run to completion and do not preempt one another. Hardware event handlers are executed in response to a hardware interrupt and also run to completion, but may preempt the execution of a task or other hardware event handler. Commands and events that are executed as part of a hardware event handler must be declared with the async keyword.
Because tasks and hardware event handlers may be preempted by other asynchronous code, nesC programs are susceptible to certain race conditions. Races are avoided either by accessing shared data exclusively within tasks, or by having all accesses within atomic statements. The nesC compiler reports potential data races to the programmer at compile-time. It is possible the compiler may report a false positive. In this case a variable can be declared with the norace keyword. The norace keyword should be used with extreme caution.

## nesC

nesC is a programming language designed to build applications for the TinyOS platform. nesC is built as an extension to the C_Programming_Language with components "wired" together to run applications on TinyOS. [2]
The basic concepts behind nesC are:

- Separation of construction and composition: programs are built out of components, which are assembled ("wired") to form whole programs. Components have internal concurrency in the form of tasks. Threads of control may pass into a component through its interfaces. These threads are rooted either in a task or a hardware interrupt.
- Specification of component behavior in terms of set of interfaces. Interfaces may be provided or used by components. The provided interfaces are intended to represent the functionality that the component provides to its user; the used interfaces represent the functionality the component needs to perform its job.
- Interfaces are bidirectional: they specify a set of functions to be implemented by the interface's provider (commands) and a set to be implemented by the interface's user (events). This allows a single interface to represent a complex interaction between components (e.g., registration of interest in some event, followed by a callback when that event happens). This is critical because all lengthy commands in TinyOS (e.g. send packet) are non-blocking; their completion is signaled through an event (send done). By specifying interfaces, a component cannot call the send command unless it provides an implementation of the sendDone event. Typically commands call downwards, i.e., from application components to those closer to the hardware, while events call upwards. Certain primitive events are bound to hardware interrupts.
- Components are statically linked to each other via their interfaces. This increases runtime efficiency, encourages robust design, and allows for better static analysis of programs.
- nesC is designed under the expectation that code will be generated by whole-program compilers. This should also allow for better code generation and analysis.

Now we will follow a step wise procedure to run a simple “HelloWorld” program written in nesC in the TinyOS platform. The installation looks very complicated, but it is sure that at the end of the installation you will be able to program for the sensor and embedded systems in efficient way using the serial and parallel ports.

## Downloading and Installing TinyOS

Download TinyOS from the following URL.

http://www.xbow.com/Support/Support_pdf_files/tinyos-1.1.0-1is.zip

The executable file sizes 142MB. If someone have problem in downloading it you can take it from me.

Execute the file. This installation package includes Cygwin, Java 1.4, JavaComm, and TinyOS 1.1.11. Choose the default settings. If you already have Cygwin or Java installed, it will display an error and tell you to uncheck it from the install list. Do it and continue with the install.

**Cygwin**

Cygwin is a collection of free software tools originally developed by Cygnus Solutions to allow various versions of Microsoft Windows to act somewhat like a UNIX system. It aims mainly at porting software that runs on POSIX systems (such as Linux systems, BSD systems, and UNIX systems) to run on Windows with little more than a recompilation. Programs ported with Cygwin work best on Windows NT, Windows 2000, Windows XP, and Windows Server 2003, but some may run acceptably on Windows 95 and Windows 98.

**JavaComm**

Sun has produced the Javacomm package to allow Java to access these serial and parallel ports, but it's not part of the core Java download. This means you have to download and install it yourself, in TinyOS it is comes with the package.

Once done you will have a directory named “tinyos” in the drive where you have installed the application. By default the drive is C. The hierarchy of the “tinyos” folder is:

- ATT
- Cygwin
- Cygwin-installationfiles
- Jdk1.4.1_02

The path “C:\tinyos\cygwin\opt\tinyos-1.x\apps” have some example programs written in nesc. We will follow the “Blink.nc” application here. The application causes the red LED on the mote to turn on and off at 1Hz.

## Running sample application in Nesc

Blink application is composed of two components: a module, called "BlinkM.nc", and a configuration, called "Blink.nc". Remember that all applications require a top-level configuration file, which is typically named after the application itself. In this case Blink.nc is the configuration for the Blink application and the source file that the nesC compiler uses to generate an executable file. BlinkM.nc, on the other hand, actually

provides the implementation of the Blink application. As you might guess, Blink.nc is used to wire the BlinkM.nc module to other components that the Blink application requires.
The reason for the distinction between modules and configurations is to allow a system designer to quickly "snap together" applications. For example, a designer could provide a configuration that simply wires together one or more modules, none of which she actually designed. Likewise, another developer can provide a new set of "library" modules that can be used in a range of applications.
Sometimes (as is the case with Blink and BlinkM) you will have a configuration and a module that go together. When this is the case, the convention used in the TinyOS source tree is that Foo.nc represents a configuration and FooM.nc represents the corresponding module. While you could name an application's implementation module and associated top-level configuration anything, to keep things simple we suggest that you adopt this convention in your own code. There are several other naming conventions used in TinyOS code.

**Blink.nc Configuration**
The nesC compiler, ncc, compiles a nesC application when given the file containing the top-level configuration. Typical TinyOS applications come with a standard Makefile that allows platform selection and invokes ncc with appropriate options on the application's top-level configuration.
Let's look at Blink.nc, the configuration for this application first:

```
configuration Blink
 {
 }
implementation {
  components Main, BlinkM, SingleTimer, LedsC;

  Main.StdControl -> BlinkM.StdControl;
  Main.StdControl -> SingleTimer.StdControl;
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

**Figure 1:** Blink.nc

The first thing to notice is the key word configuration, which indicates that this is a configuration file. The first two lines,

```
configuration Blink
  { }
```

simply state that this is a configuration called Blink. Within the empty braces here it is possible to specify uses and provides clauses, as with a module. This is important to keep in mind: a configuration can use and provide interfaces!
The actual configuration is implemented within the pair of curly bracket following key word implementation. The components line specifies the set of components that this configuration references, in this case Main, BlinkM, SingleTimer, and LedsC. The

remainder of the implementation consists of connecting interfaces used by components to interfaces provided by others.

Main is a component that is executed first in a TinyOS application. To be precise, the Main.StdControl.init() command is the first command executed in TinyOS followed by Main.StdControl.start(). Therefore, a TinyOS application must have Main component in its configuration. StdControl is a common interface used to initialize and start TinyOS components. Let us have a look at tos/interfaces/StdControl.nc:

```
interface StdControl {
  command result_t init();
  command result_t start();
  command result_t stop();
}
```

**Figure 2:** StdControl.nc

We see that StdControl defines three commands, init(),start(), and stop(). init() is called when a component is first initialized, and start() when it is started, that is, actually executed for the first time. stop() is called when the component is stopped, for example, in order to power off the device that it is controlling. init() can be called multiple times, but will never be called after either start() or stop are called. Specifically, the valid call patterns of StdControl are init*(start | stop)*. All three of these commands have "deep" semantics; calling init() on a component must make it call init() on all of its subcomponents. The following 2 lines in Blink configuration

```
Main.StdControl -> SingleTimer.StdControl;
Main.StdControl -> BlinkM.StdControl;
```

Wire the StdControl interface in Main to the StdControl interface in both BlinkM and SingleTimer. SingleTimer.StdControl.init()andBlinkM.StdControl.init() will be called by Main.StdControl.init(). The same rule applies to the start() and stop() commands.
Concerning used interfaces, it is important to note that subcomponent initialization functions must be explicitly called by the using component. For example, the BlinkM module uses the interface Leds, so Leds.init() is called explicitly in BlinkM.init().
nesC uses arrows to determine relationships between interfaces. Think of the right arrow (->) as "binds to". The left side of the arrow binds an interface to an implementation on the right side. In other words, the component that uses an interface is on the left, and the component provides the interface is on the right.
The line

```
BlinkM.Timer -> SingleTimer.Timer;
```

is used to wire the Timer interface used by BlinkM to the Timer interface provided by SingleTimer. BlinkM.Timer on the left side of the arrow is referring to the interface called Timer (tos/interfaces/Timer.nc), while SingleTimer.Timer on the right side of the arrow is referring to the implementation of Timer (tos/lib/SingleTimer.nc). Remember that the arrow always binds interfaces (on the left) to implementations (on the right).

nesC supports multiple implementations of the same interface. The Timerinterface is such a example. The SingleTimer component implements a single Timer interface while another component, TimerC, implements multiple timers using timer id as a parameter. Further discussions on timers can be found in [9].
Wirings can also be implicit. For example,
BlinkM.Leds -> LedsC;
is really shorthand for
BlinkM.Leds -> LedsC.Leds;
If no interface name is given on the right side of the arrow, the nesC compiler by default tries to bind to the same interface as on the left side of the arrow.

**BlinkM.nc Module**
Now let's look at the module `BlinkM.nc`:

```
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface Leds;
  }
}
implementation {

  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }

  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 1000) ;
  }

  command result_t StdControl.stop() {
    return call Timer.stop();
  }

  event result_t Timer.fired()
  {
    call Leds.redToggle();
    return SUCCESS;
  }
}
```

**Figure3:** `BlinkM.nc`

The first part of the code states that this is a module called BlinkMand declares the interfaces it provides and uses. The BlinkM module provides the interface StdControl. This means that BlinkM implements the StdControl interface. As explained above, this is necessary to get the Blink component initialized and started. The BlinkM module also uses two interfaces: Leds and Timer. This means that BlinkM may call any command

declared in the interfaces it uses and must also implement any events declared in those interfaces.
The Leds interface defines several commands like redOn(),redOff(), and so forth, which turn the different LEDs (red, green, or yellow) on the mote on and off. Because BlinkM uses the Leds interface, it can invoke any of these commands. Keep in mind, however, that Leds is just an interface: the implementation is specified in the Blink.nc configuration file.
Timer.nc is a little more interesting:

```
interface Timer {
  command result_t start(char type, uint32_t interval);
  command result_t stop();
  event result_t fired();
}
```

**Figure 4:** Timer.nc

Here we see that Timer interface defines the start() and stop() commands, and the fired() event.
The start() command is used to specify the type of the timer and the interval at which the timer will expire. The unit of the interval argument is millisecond. The valid types are TIMER_REPEAT and TIMER_ONE_SHOT. A one-shot timer ends after the specified interval, while a repeat timer goes on and on until it is stopped by the stop() command.
How does an application know that its timer has expired? The answer is when it receives an event. The Timer interface provides an event:
event result_t fired();
An event is a function that the implementation of an interface will signal when a certain event takes place. In this case, the fired() event is signaled when the specified interval has passed. This is an example of a bi-directional interface: an interface not only provides commands that can be called by users of the interface, but also signals events that call handlers in the user. Think of an event as a callback function that the implementation of an interface will invoke. A module that uses an interface must implement the events that this interface uses.
This is simple enough. As we see the BlinkM module implements the StdControl.init(), StdControl.start(), and StdControl.stop() commands, since it provides the StdControl interface. It also implements the Timer.fired() event, which is necessary since BlinkM must implement any event from an interface it uses.
The init() command in the implemented StdControl interface simply initializes the Leds subcomponent with the call to Leds.init(). The start() command invokes Timer.start() to create a repeat timer that expires every 1000 ms. stop() terminates the timer. Each time Timer.fired() event is triggered, the Leds.redToggle() toggles the red LED.

**Compiling the Blink.nc Application**

TinyOS supports multiple platforms. Each platform has its own directory in the tos/platform directory. In this tutorial, we will use the mica platform as an example. If

you are in the TinyOS source tree, compiling the Blink application for the Mica mote is as simple as typing

```
make mica
```

in the apps/Blink directory. Before this please copy the C:\tinyos\cygwin\opt\tinyos-1.x\apps directory into the C:\tinyos\cygwin\home\[user] directory. All the commands will be run in Cygwin. Of course this doesn't tell you anything about how the nesC compiler is invoked.

nesC itself is invoked using the ncc command which is based on gcc. For example, you can type

```
ncc -o main.exe -target=mica Blink.nc
```

to compile the Blink application (from the Blink.nc top-level configuration) to main.exe, an executable file for the Mica mote. Before you can upload the code to the mote, you use

```
avr-objcopy --output-target=srec main.exe main.srec
```

to produce main.srec, which essentially represents the binary main.exe file in a text format that can be used for programming the mote. You then use another tool (such as uisp) to actually upload the code to the mote, depending on your environment. In general you will never need to invoke ncc or avr-objcopy by hand, the Makefile does all this for you, but it's nice to see that all you need to compile a nesC application is to run ncc on the top-level configuration file for your application. ncc takes care of locating and compiling all of the different components required by your application, linking them together, and ensuring that all of the component wiring matches up.

If you actually deploy the application on a mote then it will show you some result. Till now if all goes without any error then you are going perfectly right.

## TOSSIM:

TOSSIM, the TinyOS simulator, compiles directly from TinyOS code. Built with make pc, the simulation runs natively on a desktop or laptop. TOSSIM can simulate thousands of nodes simultaneously. Every mote in a simulation runs the same TinyOS program.

TOSSIM provides run-time configurable debugging output, allowing a user to examine the execution of an application from different perspectives without needing to recompile. TinyViz is a Java-based GUI that allows you to visualize and control the simulation as it runs, inspecting debug messages, radio and UART packets, and so forth. The simulation provides several mechanisms for interacting with the network; packet traffic can be monitored, packets can be statically or dynamically injected into the network.

For further working with TOSSIM please refer to the resources [10], [11],[13] and [14].

## References:


1. http://www.tinyos.net/special/mission
2. http://en.wikipedia.org/wiki/NesC
3. http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson1.html
4. http://leitl.org/docs/intel/IR-TR-2004-60-ai-bookchapter-tinyos.pdf#search=%22filetype%3Apdf%20tinyOS%22
5. http://www.tinyos.net/tinyos-1.x/doc/tutorial/
6. http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson1.html
7. http://www.tinyos.net/tinyos-1.x/doc/tutorial/naming.html
8. http://www.tinyos.net/tinyos-1.x/doc/nesc/ref.pdf
9. http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson2.html
10. http://www.tinyos.net/tinyos-1.x/doc/tutorial/lesson5.html
11. http://www.cs.berkeley.edu/~pal/pubs/nido.pdf#search=%22how%20to%20use%20tossim%20for%20running%20nesc%20program%22
12. http://cs-people.bu.edu/gtw/motes/
13. http://www.cs.berkeley.edu/~pal/pubs/tossim-sensys03.pdf#search=%22how%20to%20use%20tossim%20for%20running%20nesc%20program%22
14. http://www.tinyos.net/nest/doc/tutorial/tossim-lesson.html