CSC2026 TinyOS Guide

Getting started

1. Introduction

Wireless sensor networks are composed of large numbers of tiny resource-limited devices (motes). Typical applications of these networks include data collection in uncontrolled environments, such as nature reserves or seismically threatened structures. Motes interact with the local environment using sensors and communicate with each other via a wireless network. Network lifetime requirements of months to a year are not uncommon.

Although very resource constrained, motes must be very reactive and participate in complex distributed algorithms, such as data aggregation or spatial localization. This combination of requirements makes traditional operating systems and programming models inappropriate for sensor networks.

TinyOS is an operating system specifically designed for network embedded systems. TinyOS has a programming model tailored for event-driven applications as well as a very small footprint (the core OS requires only 400 bytes of code and data memory, combined).

TinyOS provides a set of reusable system components. An application connects components using a wiring specification that is independent of component implementations; each application customizes the set of components it uses. Although most OS components are software modules, some are thin wrappers around hardware; the distinction is invisible to the developer. Decomposing different OS services into separate components allows unused services to be excluded from the application, keeping it small.

The nesC programming language supports and reflects TinyOS's design through the concept of components and directly supports TinyOS's event-based concurrency model.

2. The programming environment

All the PCs contain a Xubuntos virtual machine pre-loaded with all the software necessary to develop mote applications. The virtual machine is started by selecting:

Start -> Virtualisation -> TinyOS

Or try to **search** → Type in **TinyOS**

Then double click o Kubuntu_10.04, the **password** is: **tinyos**

When the virtual machine is terminated, all changes you make are overwritten to ensure that Xubuntos always starts in a known configuration.

The VM provides access to your Windows Documents folder (network H drive) via the folder /TinyOS/. Changes made under this folder are not overwritten when the virtual machine is terminated.

Tasks:

1. From within the Virtual Machine, go to **Places**, then **Computer**, then **File System** and navigate to **/opt/tinyos-2.1.1** and copy the **apps** folder to **File System/Media/sf_TinyOS**
2. Take a look at **H:/TinyOS** under Windows now to check that the **apps** folder has been copied there. Any changes you make under **H:/TinyOS** are reflected in your **File System/Media/sf_TinyOS/apps** folder.
3. You will use the VM and the files under **File System/Media/sf_TinyOS/** for the remainder of this tutorial.
4. You can edit your files from windows **H:/TinyOS** only. You cannot do that from **File System/Media/sf_TinyOS/apps** in the VM.

Task:

Open a terminal window from the **Applications/Accessories** menu. You can check the contents of your VM drive by typing in the following commands:

tinyos@sownet-tinyos-sdk: ~$ df
tinyos@sownet-tinyos-sdk:~$ cd /media/sf_TinyOS
tinyos@sownet-tinyos-sdk: */media/sf_TinyOS* $ cd apps
tinyos@sownet-tinyos-sdk:~$ ls

3. Writing Applications

3.1. Compiling and installing your first application

A mote should already be inserted into a USB socket and with the Windows device driver software loaded.

Task:

Make the mote visible to the virtual machine by selecting **Devices** pull-down menu then **USB Devices** then **FTDI FT232R USB UART**. Wait for the virtual machine device driver software to load.

Open a terminal from the **Applications/Accessories** menu. Check that the virtual machine can now see the mote. You can do this by entering the following command into a terminal window:

```
motelist
```

Make a note of the device name assigned to the mote (e.g.: /dev/ttyUSB0).

Navigate to the application directory:

```
cd  /media/sf_TinyOS/apps/Blink
```

Compile the application:

```
make gnode
```

Install the compiled application on the mote:

```
make gnode reinstall.0 bsl.0
```

If three colored lights (red, green, yellow) turn on and off in a binary sequence, then you have successfully compiled and installed your first application.

When you have finished using the mote, make the mote invisible to the virtual machine by selecting the ***Devices*** pull-down menu then ***USB Devices*** then ***FTDI FT232R USB UART***.

3.2.Components and Interfaces

Now that you've installed Blink, let's look at how it works. Blink, like all TinyOS code, is written in nesC, which is C with some additional language features for components and concurrency.

A nesC application consists of one or more components assembled, or wired, to form an application executable. Components define two scopes: one for their specification which contains the names of their interfaces, and a second scope for their implementation. A component provides and uses interfaces. The *provided* interfaces are intended to represent the functionality that the component provides to its user in its specification; the *used* interfaces represent the functionality the component needs to perform its job in its implementation.

Interfaces are bidirectional: they specify a set of commands, which are functions to be implemented by the interface's provider, and a set of events, which are functions to be implemented by the interface's user. 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.

The set of interfaces which a component provides together with the set of interfaces that a component uses is considered that component's *signature*.

3.3.Modules and Configurations

There are two types of components in nesC: *modules* and *configurations*. Modules provide the implementations of one or more interfaces. Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by others. Every nesC application is described by a top-level configuration that wires together the components inside.

3.4.Blink: An Example Application

Let's look at a concrete example: Blink in the TinyOS tree. As you saw, this application displays a counter on the three mote LEDs. In actuality, it simply causes the LEDs0 to turn on and off at 4Hz, LED1 to turn on and off at 2Hz, and LED2 to turn on and off at 1Hz. The effect is as if the three LEDs were displaying a binary count of zero to seven every two seconds.

Blink is composed of two components: a module, called "`BlinkC.nc`", and a configuration, called "`BlinkAppC.nc`". Remember that all applications require a top-level configuration file, which is typically named after the application itself. In this case `BlinkAppC.nc` is the configuration for the Blink application and the source file that the nesC compiler uses to generate an executable file. `BlinkC.nc`, on the other hand, actually provides the implementation of the Blink application. As you might guess, `BlinkAppC.nc` is used to wire the `BlinkC.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 build applications out of existing implementations. For example, a designer could provide a configuration that simply wires together one or more modules, none of which they actually designed. Likewise, another developer can provide a new set of library modules that can be used in a range of applications.

3.5.The BlinkAppC.nc Configuration

Let's look at `BlinkAppC.nc`, the configuration for this application:

```
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;
}
```

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

```
configuration BlinkAppC {
}
```

The actual configuration is implemented within the pair of braces following the key word `implementation`. The `components` lines specify the set of components that this configuration references. In this case those components are `Main`, `BlinkC`, `LedsC`, and three instances of a timer component called `TimerMilliC` which will be referenced as `Timer0`, `Timer1`, and `Timer2` respectively. This is accomplished via the `as` keyword which is simply an alias.

As we continue reviewing the application, keep in mind that the `BlinkAppC` component is not the same as the `BlinkC` component. Rather, the `BlinkAppC` component is composed of the `BlinkC` component along with `MainC`, `LedsC` and the three timers.

The remainder of the `BlinkAppC` configuration consists of lines connecting interfaces used by components to interfaces provided by others. The `MainC.Boot` interface is part of TinyOS's boot sequence and will be covered in detail later. Suffice it to say that these wirings enable the LEDs and Timers to be initialized. The last four lines wire interfaces that the `BlinkC` component `uses` to interfaces that the three instances of `TimerMilliC` and the `LedsC` components `provide`. To fully understand the semantics of these wirings, it is helpful to look at the `BlinkC` module's definition and implementation.

3.6. The BlinkC.nc Module

```
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
{
  // implementation code omitted
}
```

The first part of the module code states that this is a module called `BlinkC` and declares the interfaces it `uses`. The `BlinkC` module uses three instances of the interface `Timer<TMilli>` using the names `Timer0`, `Timer1` and `Timer2` (the `<TMilli>` syntax simply supplies the generic Timer interface with the required timer precision). Lastly, the `BlinkC` module also uses the `Leds` and `Boot` interfaces. This means that `BlinkC` may call any command declared in the interfaces it uses and must also implement any events declared in those interfaces.

After reviewing the interfaces used by the `BlinkC` component, the semantics of the last four lines in the `BlinkAppC.nc` configuration should become clearer. The line `BlinkC.Timer0 -> Timer0` wires the first `Timer<TMilli>` interface used by `BlinkC` to the `Timer<TMilli>` interface provided by the first `TimerMilliC` component. The second and third lines wire the second and third timers. The `BlinkC.Leds -> LedsC` line wires the `Leds` interface used by the `BlinkC` component to the `Leds` interface provided by the `LedsC` component.

The nesC language uses arrows to bind interfaces to one another. The right arrow (A->B) means "A wires to B". The left side of the arrow (A) `uses` the interface, while the right side of the arrow (B) `provides` the interface. The full wiring syntax is A.a->B.b, which means "interface `a` of component `A` wires to interface `b` of component `B`." Naming the interface is important when a component uses or provides multiple instances of the same interface. For example, `BlinkC` uses three instances of

Timer: Timer0, Timer1 and Timer2. When a component only has one instance of an interface it is implicitly defined and you can omit the interface name. For example, in the configuration file BlinkAppC.nc, the interface name LedsC does not have to be called LedsC.Leds.

4. Visualizing Applications

Carefully engineered TinyOS systems often have many layers of configurations, each of which refines the abstraction in simple way, building something robust with very little executable code. Getting to the modules underneath -- or just navigating the layers -- with a text editor can be laborious. To aid in this process, TinyOS and nesC have a documentation feature called nesdoc, which generates documentation automatically from source code. In addition to comments, nesdoc displays the structure and composition of configurations.

Task:

To generate documentation for your application, type:

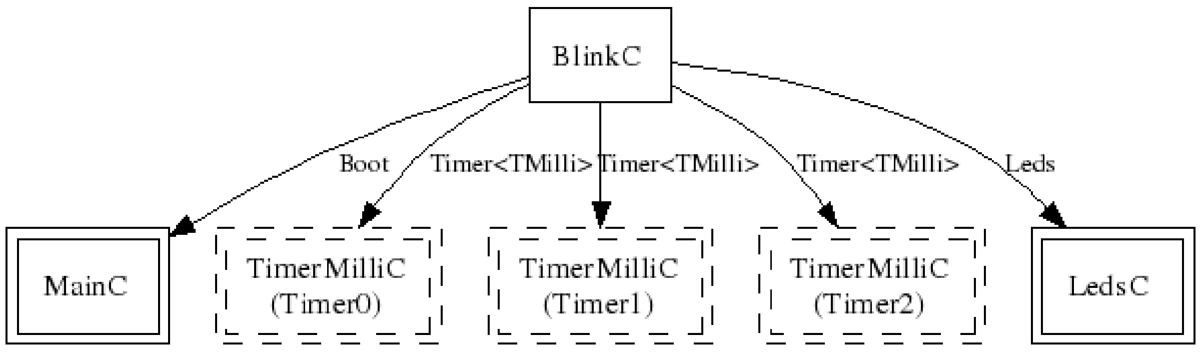
```
make gnode docs
```

You should see a long list of interfaces and components stream by.

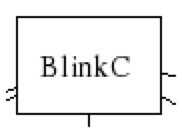
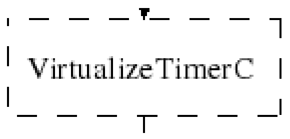
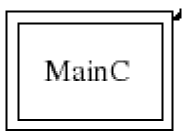
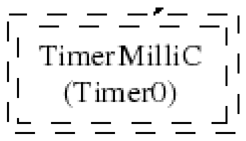
Once you've generated the documentation, within Windows, go to:

```
H:/TinyOS/doc/nesdoc/gnode
```

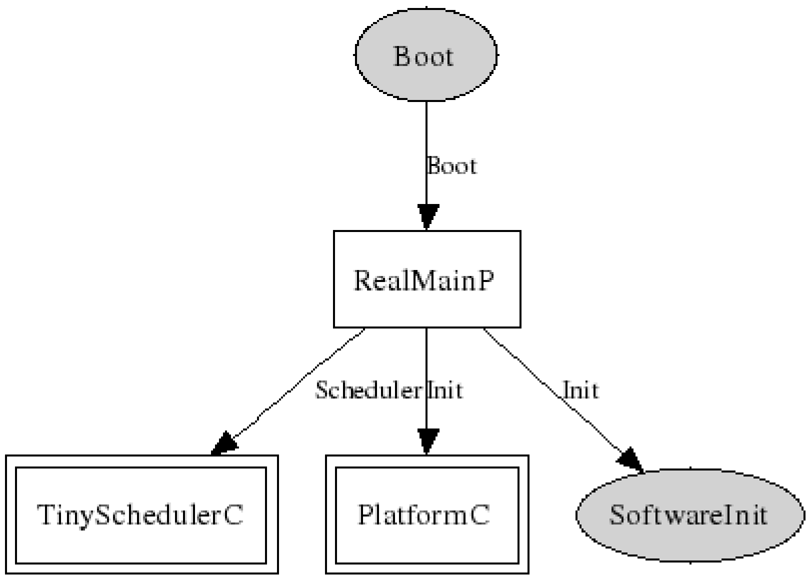
Open index.html in a browser, and in the lower left panel, you'll see a list of the interfaces and components for which documentation has been generated. Scroll down until you see the BlinkAppC component and click on it. You should now see a graphical representation of your application and all its components (BlinkC, LedsC etc).



In nesdoc diagrams, a single box is a module and a double box is a configuration. Dashed border lines denote that a component is a generic:

	Singleton	Generic
Module		
Configuration		

Lines denote wirings, and shaded ovals denote interfaces that a component provides or uses. You can click on the components in the graph to examine their internals. Click on `MainC`, which shows the wirings for the boot sequence:



Shaded ovals denote wireable interfaces. Because `MainC` provides the `Boot` interface and uses the `Init` (`as SoftwareInit`) interface, it has two shaded ovals. Note the direction of the arrows: because it uses `SoftwareInit`, the wire goes out from `RealMainP` to `SoftwareInit`, while because it provides `Boot`, the wire goes from `Boot` into `RealMainP`. The details of `MainC` aren't too important here, but looking at the components you can get a sense of what it does: it controls the scheduler, initializes the hardware platform, and initializes software components.

5. Experimenting with NesC

Look at the `implementation` code in `BlinkC.nc`. Notice that there's no `main()`, only events. The processor can sleep between events, thereby conserving energy.

Task:

Try changing the code so the three LEDs all flash at 2 second intervals.

If you accidently break the Blink application, copy the original code again from:

/opt/tinyos-2.1.1/apps/Blink

6. References

[1] nesC reference manual at http://nescc.sourceforge.net/papers/nesc-ref.pdf

[2] TinyOS Programming Guide at http://csl.stanford.edu/~pal/pubs/tinyos-programming-1-0.pdf