**Tinyos and nesC:**

TinyOS is an open-source operating system designed for wireless embedded sensor networks. It features a component-based architecture. TinyOS is not an operating system in the traditional sense; it is a programming framework for embedded systems that enables building an application-specific OS into each application. The reason for this is to ensure that the application code has an extremely small memory footprint. TinyOS is designed to have no file system, supports only static memory allocation, implements a simple task model, and provides minimal device and networking abstractions.

**nesC (network embedded systems C)** is a component-based, event-driven programming language used to build applications for the TinyOS embedded operating system. nesC is built as an extension to the C programming language with components "wired" together to run applications on TinyOS.

The basic concepts behind nesC are:

1. **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.
2. **Specification of component behavior in terms of a 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.
3. **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.
4. **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.
5. **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.

## Introduction to Components

Every program in nesC is formed by ***Components***. There are two kinds of *Components*: **Modules** and **Configurations**. A *Component* is always a file written in nesC, that presents two sections, containing its specification and its implementation. However, the implementation section differs depending whether we are talking about a Module or a Configuration:

* Module implementation sections consist of nesC code that looks like C. Module code declares variables and functions, calls functions, and compiles to assembly code.
* Configuration implementation sections consist of nesC *wiring* code, which connects components together. Configurations are the major difference between nesC and C (and other C derivatives).

All components have therefore two code blocks. The first block describes their signature, and the second block describes their implementation:

module testP {

 // signature

}

implementation {

 // implementation

}

configuration testC {

 // signature

}

implementation {

 // implementation

}

Let's take a closer look at this with an example:

There are two types of files in nesC programming, Components and Interfaces.

### **Main Applications**

Any application in nesC requires at least the following:

* Makefile
* Configuration
* Module

Each one of them corresponds to an actual file, and it has to be stored in the same folder. Example:



While it is possible to use any kind of name, it is always recommended to use this naming convention for your files. This and other practices are better described in TinyOS:[TEP 3](http://www.tinyos.net/tinyos-2.x/doc/html/tep3.html). Therefore, from now on, our programs will add *C* at the end of configuration file names and *P* at the end of modules.

Nevertheless, within TinyOS example applications, we will find that occasionally this naming convention is not used.

#### **Makefile**

The basic structure for a Main Application Makefile is:

COMPONENT=testC

include $(MAKERULES)

In *COMPONENT* we indicate the name of the configuration file, in this case *testC*. The compiler will search in the folder for a file named *testC.nc*.

#### **Configuration**

As indicated in the Makefile, the **Configuration** file is *testC.nc*. A *Configuration* is a public component, which means that other components may instantiate it in order to use the provided interfaces. However, in this case, no components can wire to this Configuration as it is a main file, which is not providing any interfaces. The lines included therefore refer to other components that are used:

configuration testC

{

}

implementation

{

 components MainC;

 components testP as App;

 App -> MainC.Boot;

}

The Configuration presents two sections, *"configuration"* and *"implementation"*:

* **configuration:** N/A in Main Applications.
* **implementation:** Links to other components. Contrary to the rest of nesC components, Main Applications instantiate the component **MainC**. It is also mandatory for every configuration to always instantiate their module, in this case *testP*. In this example we are telling the compiler that MainC's interface Boot will be used in the module *testP*.

**Note:** The keyword "as" is useful to rename components within the code. By assigning the alias *App* to the component's module testP, we can easily reuse code in other applications without renaming.

#### **Module**

The **Module** file is *testP.nc*. The Module is the private part of the component, and it is where the actual implementation of the code takes place:

module testP

{

 uses interface Boot;

}

implementation

{

 event void Boot.booted()

 {

 }

}

As with the Configuration, the Module also presents two sections, *"module"* and *"implementation"*:

* **module:** With the keyword *"uses"* we indicate what interfaces we are going to use. If we check the system provided interface Boot (*TOSROOT/tos/interfaces/Boot.nc*) we can see that this particular interface provides the event **booted**.
* **implementation:** In Main Applications, this is where commands will be called and events will happen. In this simple example, we have a unique event happening (*booted*). The syntax for events is always the same, first of all we must indicate the interface name (Boot) followed by a full-stop and then the event name.

### **Header Files**

As with a pure C program, nesC allows the use of header files in your code. This can be especially useful on those occasions where many constants or variable types have to be defined, in order to make it easy to quickly edit a program. An example header file could be as follows:

#ifndef TESTCM5000\_H

#define TESTCM5000\_H

enum{

 DEFAULT\_TIMER = 10240, // 10 seconds

 MAX\_SENSORS = 5, // Number of sensors (Vref, Temp, Hum, Light, TSR)

 TestCM5000\_AM\_ID = 0x01, // TestCM5000 AM ID

};

typedef nx\_struct THL\_msg {

 nx\_uint16\_t vref;

 nx\_uint16\_t temperature;

 nx\_uint16\_t humidity;

 nx\_uint16\_t photo;

 nx\_uint16\_t radiation;

} THL\_msg\_t;

#endif

This header file has been taken from CM5000 Example Application. To use the header file, just include the line:

#include TestCM5000.h

at the beginning of *testP.nc*.

### **How to compile an application**

We now have the most basic nesC application. To proceed, we should now learn how to compile our program and install it in a mote. To do so, place all three files, *testC.nc*, *testP.nc* and the *Makefile,* in the same folder, and open a console at that location. Type ***make telosb***and enter. With this instruction, we are telling the compiler to compile our application taking telosb as target platform. Most of Advanticsys® products, like the [CM5000](https://www.advanticsys.com/wiki/index.php?title=CM5000), are based on the telosb design. Your output should be something like:

advantic@advantic-desktop:~/test$ make telosb

mkdir -p build/telosb

 compiling testC to a telosb binary

ncc -o build/telosb/main.exe -Os -fnesc-separator=\_\_ -Wall -Wshadow -Wnesc-all -target=telosb

-fnesc-cfile=build/telosb/app.c -board= -DDEFINED\_TOS\_AM\_GROUP=0x22 -DIDENT\_APPNAME=\"testC\"

-DIDENT\_USERNAME=\"mfst\" -DIDENT\_HOSTNAME=\"mfst-desktop\" -DIDENT\_USERHASH=0xc19fcdb5L

-DIDENT\_TIMESTAMP=0x5028e1d8L -DIDENT\_UIDHASH=0xf985058fL testC.nc -lm

 compiled testC to build/telosb/main.exe

 1320 bytes in ROM

 6 bytes in RAM

msp430-objcopy --output-target=ihex build/telosb/main.exe build/telosb/main.ihex

 writing TOS image

advantic@advantic-desktop:~/test$

Let's take a look at the files that have been created. Browse to the ***test/build/telosb***folder that has been created. In your console type:

advantic@advantic-desktop:~/test$ ls -l build/telosb/

total 120

-rw-r--r-- 1 advantic advantic 96639 2012-08-13 13:15 app.c

-rw-r--r-- 1 advantic advantic 173 2012-08-13 13:15 ident\_flags.txt

-rwxr-xr-x 1 advantic advantic 8107 2012-08-13 13:15 main.exe

-rw-r--r-- 1 advantic advantic 3856 2012-08-13 13:15 main.ihex

-rw-r--r-- 1 advantic advantic 4211 2012-08-13 13:15 tos\_image.xml

advantic@advantic-desktop:~/test$

You should see a list of files. TinyOS® uses the following compilation model:



Essentially, whenever we compile our application by typing *make telosb*, we are first of all compiling using the nesC compiler. The result of this first process is the ***app.c***file, which is a pure C file. This C file is then compiled with the native gcc compiler to produce a binary file, which is what will be finally installed in the actual mote. In the case of the telosb platform, the compiler used is ***msp430-gcc***, and the binary output in always called ***main.exe***.

### **How to install an application**

Now that we have an application binary, it is time to install it in the mote. Connect a mote, such as CM5000, to the USB port of your PC. To install the compiled binary, it is first needed to know the port to which the CM5000 is mapped. To find out the port, type ***motelist***on your console:

advantic@advantic-desktop:~/test$ motelist

Reference Device Description

---------- ---------------- ---------------------------------------------

MFV60NW2 /dev/ttyUSB0 FTDI MTM-CM5000MSP

advantic@advantic-desktop:~/testCM5000$

As we can see, the CM5000 is mapped to the /dev/ttyUSB0 port, which is the default port in TinyOS®. Now, type ***make telosb reinstall.0x0042 bsl,/dev/ttyUSB0***,where *0x0042*indicates the mote's id (in this case, 0x0042 in *hex*format) and */dev/ttyUSB0*is the port where the mote is connected. The Mote id is therefore configurable, and it accepts any value between 0x0000 and 0xFFFF, that is, any value between 0 and 65535. The global variable **TOS\_NODE\_ID** will always adopt this value. In this example program, this variable is not used, but this will not always be the case as in radio-based application this id is vital to distinguish one mote from the rest:

advantic@advantic-desktop:~/test$ make telosb reinstall.0x0042 bsl,/dev/ttyUSB0

tos-set-symbols --objcopy msp430-objcopy --objdump msp430-objdump --target ihex build/telosb/main.ihex

build/telosb/main.ihex.out-0x0042 TOS\_NODE\_ID=0x0042 ActiveMessageAddressC\_\_addr=0x0042

Could not find symbol ActiveMessageAddressC\_\_addr in build/telosb/main.exe, ignoring symbol.

Could not find symbol TOS\_NODE\_ID in build/telosb/main.exe, ignoring symbol.

 installing telosb binary using bsl

tos-bsl --telosb -c /dev/ttyUSB0 -r -e -I -p build/telosb/main.ihex.out-0x0042

MSP430 Bootstrap Loader Version: 1.39-telos-8

Mass Erase...

Transmit default password ...

Invoking BSL...

Transmit default password ...

Current bootstrap loader version: 1.61 (Device ID: f16c)

Changing baudrate to 38400 ...

Program ...

1352 bytes programmed.

Reset device ...

rm -f build/telosb/main.exe.out-0x0042 build/telosb/main.ihex.out-0x0042

advantic@advantic-desktop:~/test$

Alternatively, the command ***install***could be used and both the compilation and installation would be done in the same step.

The instruction would in this case be ***make telosb install.0x0042bsl,/dev/ttyUSB0***.

Congratulations, the program is now installed in the mote! However, the program is so simple that it doesn't really do anything.

Re-open the example of Blink and try once again to see how it works and analyze the code based on what you learned here.