# Ασύρματα Δίκτυα Αισθητήρων

Πέππας Κωνσταντίνος

# Προγραμματισμός σε NesC

- Program structure is the most essential and obvious difference between C and nesC.
- C programs are composed of variables, types and functions defined in files that are compiled separately and then linked together.
- nesC programs are built out of components that are connected (“wired”) by explicit program statements;
- the nesC compiler connects and compiles these components as a single unit.
- To illustrate and explain these differences in how programs are built, we compare and contrast C and nesC implementations of two very simple “hello world”-like mote applications, Powerup (boot and turn on a LED) and Blink (boot and repeatedly blink a LED)

# Προγραμματισμός σε NesC

- The closest mote equivalent to the classic “HelloWorld!” program is the “Powerup” application that simply turn on one of the motes LEDs at boot, then goes to sleep.
- A C implementation of Powerup is fairly simple:

```
#include "mote.h"
int main()
{
mote_init();
led0_on();
sleep();
}
```

# Προγραμματισμός σε NesC

- The Powerup application is compiled and linked with a “mote” library which provides functions to perform hardware initialization (mote init), LED control (led0 on) and put the mote in to a low-power sleep mode (sleep).
- The “mote.h” header file simply provides declarations of these and other basic functions. The usual C main function is called automatically when the mote boots.
- The nesC implementation of Powerup is split into two parts. The first, the PowerupC module, contains the executable logic of Powerup (what there is of it. . . ):

# Προγραμματισμός σε NesC

```
module PowerupC
{
        uses interface Boot;
        uses interface Leds;
}
implementation
{
        event void Boot.booted()
        {
                call Leds.led0On();
        }
}
```

# Προγραμματισμός σε NesC

- This code says that PowerupC interacts with the rest of the system via two interfaces, Boot and Leds, and provides an implementation for the booted event of the Boot interface that calls the led0On2 command of the Leds interface.
- Comparing with the C code, we can see that the booted event implementation takes the place of the main function, and the call to the led0On command the place of the call to the led0 on library function.
- This code shows two of the major differences between nesC and C:
- where C programs are composed  of functions, nesC programs are built out of components that implement a particular service (in the case of PowerupC, turning a LED on at boot-time).
- Furthermore, C functions typically interact by calling each other directly, while the interactions between components are specified by interfaces:
- the interface’s user makes requests (calls commands) on the interface’s provider, the provider makes callbacks (signals events) to the interface’s user.

# Προγραμματισμός σε NesC

- Commands and events themselves are like regular functions (they can contain arbitrary C code);
- calling a command or signaling an event is just a function call. PowerupC is a user of both Boot and Leds;
- the booted event is a callback signaled when the system boots, while the led0On is a command requesting that LED 0 be turned on.
- nesC interfaces are similar to Java interfaces, with the addition of a command or event keyword to distinguish requests from callbacks:

# Προγραμματισμός σε NesC

```
interface Boot {
event void booted();
}
interface Leds {
command void led0On();
command void led0Off();
command void led0Toggle();
...
}
```

# Προγραμματισμός σε NesC

The second part of Powerup, the PowerupAppC configuration, specifies how PowerupC is connected to TinyOS's services:

```
configuration PowerupAppC { }
implementation
{
        Components MainC, LedsC, PowerupC;
        MainC.Boot -> PowerupC.Boot;
        PowerupC.Leds -> LedsC.Leds;
}
```

# Προγραμματισμός σε NesC

- This says that the PowerupAppC application is built out of three components (modules or configurations), MainC (system boot), LedsC (LED control), and PowerupC (our powerup module).
- PowerupAppC explicitly specifies the connections (or wiring) between the interfaces provided and used by these components.
- When MainC has finished booting the system it signals the booted event of its Boot interface, which is connected by the wiring in PowerupAppC to the booted event in PowerupC.
- This event then calls the led0On command of its Leds interface, which is again connected (wired) by PowerupAppC to the Leds interface provided by LedsC.
- Thus the call turns on LED 0. The resulting component diagram is shown in Figure 2.1
- — this  diagram was generated automatically from PowerupAppC by nesdoc, nesC's documentation generation tool.

# Προγραμματισμός σε NesC

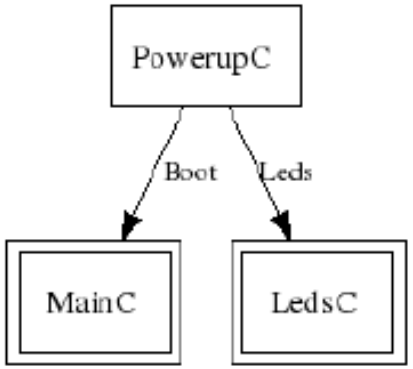


Figure 2.1: Wiring Diagram for Powerup application

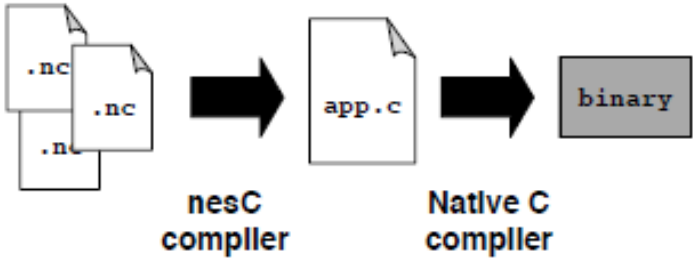


Figure 2.2: The nesC compilation model. The nesC compiler loads and reads in nesC components, which it compiles to a C file. This C file is passed to a native C compiler, which generates a mote binary.

# Προγραμματισμός σε NesC

- PowerupAppC illustrates the third major difference between C and nesC:
- wiring makes the connections expressed by linking the C version of Powerup with its “mote” library explicit.
- In the C version, Powerup calls a global function named led0 on which is connected to whatever library provides a function with the same name;
- if two libraries provide such a function then (typically) the first one named on the linker command line “wins”.
- Using a nesC configuration, the programmer instead explicitly selects which component’s implementation of the function to use.

# Προγραμματισμός σε NesC

- The nesC compiler can take advantage of this explicit wiring to build highly optimized binaries.
- Current implementations of the nesC compiler (nesc1) take nesC files describing components as input and output a C file.
- The C file is passed to a native C compiler that can compile to the desired microcontroller or processor.
- Figure 2.2 shows this process.
- The nesC compiler carefully constructs the generated C file to maximize the optimization abilities of the C compiler. For example, since it is given a single file, the C compiler can freely optimize across call boundaries, inlining code whenever needed.
- The nesC compiler also prunes dead code which is never called and variables which are never accessed:
- since there is no dynamic linking in nesC, it has a complete picture of the application call graph. This speeds the C compilation and reduces program size in terms of both RAM and code

# Προγραμματισμός σε NesC

- The three essential differences between C and nesC — components, interfaces and wiring — all relate to naming and organizing a program's elements (variables, functions, types, etc).
- In C, programs are broken into separate files which are connected via a global namespace:
- a symbolX declared in one file is connected by the linker to a symbol X defined in another file.

# Προγραμματισμός σε NesC

- For instance, if file1.c contains:

```
extern void g(void); /* declaration of g */
int main() /* definition of main */
{
g(); g();
}
```

and file2.c contains:

```
void g(void)
{
printf("hello world!");
}.
```

then compiling and linking file1.c and file2.c connects the calls to g() in main to the definition of g in file2.c. The resulting program prints “hello world!” twice.

# Προγραμματισμός σε NesC

- Organizing symbols in a global namespace can be tricky.
- C programmers use a number of techniques to simplify this task, including header files and naming conventions.
- Header files group declarations so they can be used in a number of files without having to retype them, e.g. a header file file1.h for file1.c would normally contain:

```
#ifndef FILE1_H
#define FILE1_H
extern void g(void); /* declaration of g */
#endif
```

# Προγραμματισμός σε NesC

- Naming conventions are designed to avoid having two different symbols with the same name.
- For instance, types are often suffixed with t guaranteeing that a type and function won’t have the same name.
- Some libraries use a common prefix for all their symbols, e.g. Gtk and gtk for the GTK+ graphical toolkit.
- Such prefixes remind users that functions are related and avoid accidental name collisions with other libraries, but make programs more verbose.

# Προγραμματισμός σε NesC

- nesC's components provide a more systematic approach for organizing a program's elements.
- A component (module or configuration) groups related functionality (a timer, a sensor, system boot) into a single unit, in a way that is very similar to a class in an object-oriented language.
- For instance, TinyOS represents its system services as separate components such as LedsC (LED control, seen above), ActiveMessageC (sending and receiving radio messages), etc.
- Only the service (component) name is global, the service's operations are named in a per-component scope: ActiveMessageC.SplitControl starts and stops the radio, ActiveMessageC.AMSend sends a radio message, etc.

# Προγραμματισμός σε NesC

- Interfaces bring further structure to components: components are normally specified in terms of the set of interfaces (Leds, Boot, SplitControl, AMSend) that they provide and use, rather than directly in terms of the actual operations.
- Interfaces simplify and clarify code because, in practice, interactions between components follow standard patterns: many components want to control LEDs or send radio messages, many services need to be started or stopped, etc.
- Encouraging programmers to express their components in terms of common interfaces also promotes code reuse: expressing your new network protocol in terms of the AMSend message transmission interface means it can be used with existing applications, using AMSend in your application means that it can be used with any existing or future network protocol

# Προγραμματισμός σε NesC

- Rather than connect declarations to definitions with the same name, nesC programs use wiring to specify how components interact:

PowerupAppC wired PowerupC's Leds interface to that provided by the LedsC component, but a two-line change could switch that wiring to the NoLedsC component (which just does nothing):

components PowerupC, NoLedsC;

PowerupC.LedsC -> NoLedsC.Leds;

- without affecting any other parts of the program that wish to use LedsC.
- In C, one could replace the "mote" library used by Powerup by a version where the LED functions did nothing, but that change would affect all LED users, not just Powerup.

# Προγραμματισμός σε NesC

- Leaving the component connection decisions to the programmer does more than just simplify switching between multiple service implementations.
- It also provides an efficient mechanism for supporting callbacks, as we show through the example of timers.
- TinyOS provides a variable number of periodic or deadline timers; associated with each timer is a callback to a function that is executed each time the timer fires.
- We first look at how such timers would be expressed in C, by modifying Powerup to blink LED 0 at 2Hz rather than turn it on once and for all:

# Προγραμματισμός σε NesC

- Rather than connect declarations to definitions with the same name, nesC programs use wiring to specify how components interact:

PowerupAppC wired PowerupC's Leds interface to that provided by the LedsC component, but a two-line change could switch that wiring to the NoLedsC component (which just does nothing):

components PowerupC, NoLedsC;

PowerupC.LedsC -> NoLedsC.Leds;

- without affecting any other parts of the program that wish to use LedsC.
- In C, one could replace the "mote" library used by Powerup by a version where the LED functions did nothing, but that change would affect all LED users, not just Powerup.

# Προγραμματισμός σε NesC

```
#include "mote.h"
timer_t mytimer;
void blink_timer_fired(void)
{
leds0_toggle();
}
int main()
{
mote_init();
timer_start_periodic(&mytimer, 250, blink_timer_fired);
sleep();
}
```

# Προγραμματισμός σε NesC

- In this example, the Blink application declares a global mytimer variable to hold timer state, and calls timer start periodic to set up a periodic 250ms timer.
- Every time the timer fires, the timer implementation performs a callback to the blink timer fired function specified when the timer was set up.
- This function simply calls a library function that toggles LED 0 on or off.
- The nesC version of Blink is similar to the C version, but uses interfaces and wiring to specify the connection between the timer and the application:

# Προγραμματισμός σε NesC

```
module BlinkC {
uses interface Boot;
uses interface Timer;
uses interface Leds;
}
implementation {event void Boot.booted() {
call Timer.startPeriodic(250);
}
event void Timer.fired() {
call Leds.led0Toggle();
}
}
```

# Προγραμματισμός σε NesC

- The BlinkC module starts the periodic 250ms timer when it boots.
- The connection between the startPeriodic command that starts the timer and the fired event which blinks the LED is implicitly specified by having the command and event in the same interface:

```
interface Timer {
command void startPeriodic(uint32_t interval);
event void fired();
...
}
```

# Προγραμματισμός σε NesC

- Finally, this Timer must be connected to a component that provides an actual timer.
- BlinkAppC wires BlinkC.Timer to a newly allocated timer MyTimer:

```
configuration BlinkAppC { }
implementation {
components MainC, LedsC, new TimerC() as MyTimer, BlinkC;
BlinkC.Boot -> MainC.Boot;
BlinkC.Leds -> LedsC.Leds;
BlinkC.Timer -> MyTimer.Timer;
}
```

# Προγραμματισμός σε NesC

```
module BlinkC {
uses interface Boot;
uses interface Timer;
uses interface Leds;
}
implementation {event void Boot.booted() {
call Timer.startPeriodic(250);
}
event void Timer.fired() {
call Leds.led0Toggle();
}
}
```

# Προγραμματισμός σε NesC

- In the C version the callback from the timer to the application is a runtime argument to the timer start periodic function.
- The timer implementation stores this function pointer in the mytimer variable that holds the timer's state, and performs an indirect function call each time the timer fires.
- Conversely, in the nesC version, the connection between the timer and the Blink application is specified at compile-time in BlinkAppC.
- This avoids the need to store a function pointer (saving precious RAM), and allows the nesC compiler to perform optimizations (in particular, inlining) across callbacks.

# Προγραμματισμός σε NesC

- A nesC program is a collection of components.
- Every component is in its own source file, and there is a 1-to-1 mapping between component and source file names.
- For example, the file LedsC.nc contains the  nesC code for the component LedsC, while the component PowerupC can be found in the file PowerupC.nc.
- Components in nesC reside in a global namespace:
- there is only one PowerupC definition, and so the nesC compiler loads only one file named PowerupC.nc.
- There are two kinds of components: modules and configurations. Modules and configurations can be used interchangeably when combining components into larger services or abstractions.

# Προγραμματισμός σε NesC

- The two types of components differ in their implementation sections.
- 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 two code blocks. The first block describes its signature, and the second block describes its implementation:

# Προγραμματισμός σε NesC

```
module PowerupC {
// signature
}
implementation {
// implementation
}

configuration LedsC {
// signature
}
implementation {
// implementation
}
```

# Προγραμματισμός σε NesC

- Signature blocks in modules and configurations have the same syntax.
- Component signatures contain zero or more interfaces.
- Interfaces define a set of related functions for a service or abstraction.
- For example, there is a Leds interface for controlling node LEDs, a Boot interface for being notified when a node has booted, and an Init interface for initializing a component’s state.
- A component signature declares whether it provides or uses an interface.
- For example, a component that needs to turn a node’s LEDs on and off uses the Leds interface, while the component that implements the functions that turns them on and off provides the Leds interface.
- Returning to the two examples, these are their signatures

# Προγραμματισμός σε NesC

```
module PowerupC {
uses interface Boot;
uses interface Leds;
}

configuration LedsC {
provides interface Leds;
}
```

# Προγραμματισμός σε NesC

- PowerupC is a module that turns on a node LED when the system boots.
- It uses the Boot interface for notification of system boot and the Leds interface for turning on a LED.
- LedsC, meanwhile, is a configuration which provides the abstraction of three LEDs that can be controlled through the Leds interface.
- A single component can both provide and use interfaces.
- For example, this is the signature for the configuration MainC:

# Προγραμματισμός σε NesC

```
configuration MainC {
provides interface Boot;
uses interface Init;
}
```

# Προγραμματισμός σε NesC

- MainC is a configuration which implements the boot sequence of a node.
- It provides the Boot interface so other components, such as PowerupC, can be notified when a node has fully booted.
- MainC uses the Init interface so it can initialize software as needed before finishing the boot sequence.
- If PowerupC had state that needed initialization before the system boots, it might provide the Init interface