International Journal of Multimedia and Ubiquitous Engineering
Vol. 10, No. 1 (2015), pp. 119-126
http://dx.doi.org/10.14257/ijmue.2015.10.1.11

A Case Study on Ubiquitous Sensor Network Programming

Jin-whan Kim

Dept. of Computer Engineering, Youngsan University
San 150 Junam-dong, Yangsan, Gyungnam, 626-790 Korea
kjw@ysu.ac.kr

Abstract

This paper presents research on the TinyOS operating system and associated nesC programming methods. TinyOS is currently being used in many domestic and foreign research institutes, universities and companies. We describe TinyOS and nesC programming methods and their characteristics, comparing with existing programming languages such as C, C++, and MFC window programming. Additionally control methods and analysis activation characteristics of two domestic companies' wireless sensors are introduce. We tried to explain easily and hope to be used in various industry fields.

Keywords: *USN, TinyOS, NesC Programming, Intelligent Wireless Sensor*

1. Introduction

The USN (Ubiquitous Sensor Network), aimed at implementing a communication environment where users freely connect to networks and may transmit and receive wanted information regardless of place and time, is an environment where various wireless sensors, computers and telecommunication networks are connected to utilize data in real time. Specifically an environment which manages and utilizes data in real time through wired and wireless networks by attaching wireless sensors to all matter, people and environment, to search for various datum from the sources or networks. The technology has been widely used in almost all fields including homes, offices, logistics, distribution, environment monitoring, intelligent homes, military, aerospace, ships and vehicles with more in the future. Generally, wireless sensors with embedded wireless communication chips, memory, small operating system (OS) and applications, as in Figure 1, and are called motes or sensor nodes.

Figure 1. Various Intelligent Wireless Sensors

Currently, well-known operating systems for sensor networks include TinyOS, SOS, MANTIS, Contiki, T-kernel and the locally-developed Nano-Qplus [1, 2, 3, 4, 5, 6, 7]. This paper analyzes TinyOS, which is widely used in local research institutes, universities and companies, and explains methods and features of nesC programming. Some important terms used in nesC programming are new to existing programming languages and may cause confusion among current programmers. Therefore, this study seeks to explain through comparison to other existing programming languages (C, C++, MFC Windows programming languages, *etc*). Part 2 takes a look at the structure and features of TinyOS and part 3, through a comparison to existing languages, reviews the structure, terms and usages of nesC programming language. Finally part 4 proposes case studies using and controlling actual wireless sensors in 2 local companies.

ISSN: 1975-0080 IJMUE
Copyright ⓒ 2015 SERSC

2. Structure and Features of the TinyOS

TinyOS is a small-scale OS, developed by UC Berkley, providing open source development to implement a USN environment and features a well modulated and event-driven environment based on components for operation with limited resources. It uses a small amount of memory, requiring a small power and implementing modulation and simultaneous operation [8, 9]. Events and tasks cause sensor nodes to execute in TinyOS. A hardware *event* means an interrupt and the interrupt is generated by the timer, sensor and telecommunication device. A *Task* means a procedure call in general programming languages, however tasks are registered in the task queue to be executed based on an FIFO (First In First Out) concept. This means that a task is executed in a non-preemptive method where the task may not be executed before another one has completed its execution. If there are no tasks to be executed in the task queue, to minimize power consumption the system is switched to the slip mode until a new hardware interrupt or task appears. TinyOS consists of modules of a component basis. Each component is connected to others through the interface and executed by command and event functions. The components have features similar to DLLs (Dynamic Linked Library) of the MFC. *Interface* is a term used in the nesC similar to *class* in C++ or MFC programming. The interface shall be declared to use various interfaces and the command provided by the interface is similar to the *class member function* in MFC to execute any action and an *event* means a function automatically executed by a certain action or environmental change. This is similar to the generation of an automatic message caused by the handling Timer, mouse or keyboard input. Also, the TinyOS performs certain actions through a split—phase method. If the blocking system calls a func 1() with a long action time, as in Figure 2, then other programs must wait until the function returns.

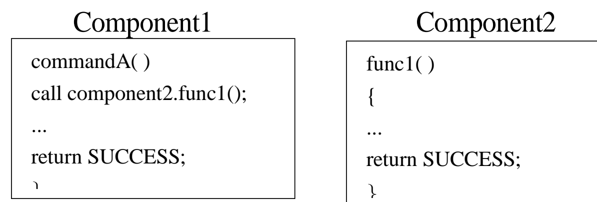


Figure 2. Function Call of Blocking System

However, the split—phase system immediately returns func2() with a long action time, as in Figure 3. The function enters the task queue and signals the event function func3() to secure real time actions which are almost immediately performed.

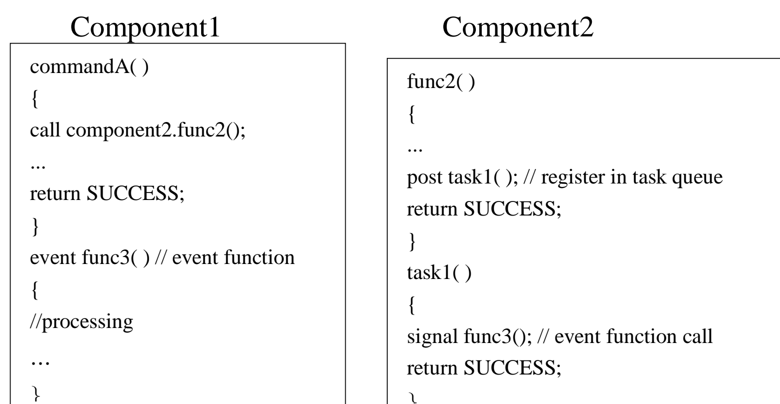


Figure 3. Function Call of Split-phase System

 Copyright ⓒ 2015 SERSC

Application developers use components as libraries and interfaces to connect the components to each other for application development.

3. nesC Programming

TinyOS uses a programming language called the nesC to develop applications. The nesC language is an expanded form of C and was developed for embedded systems like sensor networks. The developing environment supports Linux or Windows 2000/XP using Cygwin. The language takes features of C, C++ and MFC (Microsoft Foundation Class) programming languages. Software developed by the nesC is converted to C through a conversion process, compiled to execution codes through a GCC compiler, then downloaded to and executed in the sensor nodes.

3.1. Component

nesC is designed with component-based structures. All the programs consist of components and the components may be composed of configurations, connecting modules and other components [10, 11, 12]. The module implements components using similar codes to C and the configuration clarifies relationships between components to be used and interfaces. A nesC program shall contain the main component as the main() function included in a C language program. This is because the main component calls the system initialization routine and operates scheduler.

3.1.1. Configuration

The configuration clarifies components to be used, displays mutual connection (->, <-, =) among interfaces, shows relations between uses and provides and expresses logical writing among different components. The configuration consists of top-level with an aaa.nc file and general configuration with bbbC.nc. Each application shall contain at least one aaa.nc file. There are 3 connection symbols (->, <-, =) to express wires among components in nesC.

· interface aa -> interface bb: aa uses a function implemented and used by bb.
  bb and aa mean the supplier and user interface, respectively.
· interface aa <- interface bb: bb uses a function implemented and used by aa.
  aa and bb mean the supplier and user interface, respectively.
· interface aa = interface bb: Both aa and bb interfaces are the same.
  aa and bb mean user or provider interfaces.

With connections as above, the user may use commands, events and interfaces of the components. Hereafter codes for apps/Blink.nc. are explained:

```
configuration Blink
//Declaring configuration in the Blink component
{
}
implementation  {
  components Main, BlinkM, SingleTimer, LedsC;
  // Declaring components to be used
  Main.StdControl -> BlinkM.StdControl;
 // The BlinkM component provides the StdControl
   //  interface to Main
  Main.StdControl -> SingleTimer.StdControl;
  // The SingleTimer component provides the StdControl
   // interface to Main
  BlinkM.Timer -> SingleTimer.Timer;
  // The SingleTimer component provides the Timer
   // Interface to BlinkM
```

Copyright ⓒ 2015 SERSC

```
    BlinkM.Leds -> LedsC;
    // The LedsC component provides the Leds interface to
    // BlinkM
  }
  The StdControl is defined as below in tos/interfaces/StdControl.nc.
  interface StdControl {
    command result_t init();
    command result_t start();
    command result_t stop();
```

3.1.2. Module

The module uses or provides pre-defined interfaces and writes execution codes during the implementation. The module has files formed with aaaM.nc and writes command(event) codes in the implementation{}. Each module calls commands and signals events. It signals required components through uses and provides implementation using provided commands to other components. All the commands and events declared in the module are written as events and codes in the implementation. Analyzing the example of apps/Blinkm.nc structure helps understand the module.

```
module BlinkM {
  provides {
    interface StdControl; //Provide StdControl to Blink.nc
  }
  uses {
    interface Timer; // An interface used by BlinkM
    interface Leds;  // Shall be declared in Blink.nc
    }
}

implementation { // Write a code executed here
      command result_t StdControl.init() {
      // Component initialization
      // CreateWindow() while creating a window in the MFC
      // A command function similar to
      // automatically-executed OnCreate() message handler
    call Leds.init();
    return SUCCESS;
  }
   command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 1000) ;
    // Cause Timer to generate signals at 1000ms interval
    // A function similar to SetTimer() of the MFC
   }

   command result_t StdControl.stop() {
    // command function of StdControl
    return call Timer.stop();
    // command function of Timer
    // A function similar to KillTimer() of the MFC
   }

   event result_t Timer.fired()
```

 Copyright ⓒ 2015 SERSC

```
// Automatically called whenever a signal is generated based on configured Timer interval
// Generate signals with pre-defined 1000ms interval
// Event function of the Timer
// A function similar to OnTimer() message handler of the MFC programming

 call Leds.redToggle();
 // Toggle red-colored LED
 return SUCCESS;
}
```

Table 1 explains how to implement actual codes of command and event functions in the component and calls (or signals) when uses and provides are declared.

Table 1. Usage of Uses and Provides

components	commands	events
uses	Call possibility	Implement execution codes
provides	Implement execution codes	Signal possibility

The Timer is defined as below in tos/interfaces/Timer.nc.

```
interface Timer
  command result_t start(char type, unit32_t interval);
  command result_t stop();
  event result_t fired();
}
```

3.1.3. Interface

The Interface has bidirectional features in nesC and plays a role as a connector between provider and user components. The provider module implements codes for all the commands and the user module implements all the event codes. Table 2 shows interfaces used in the nesC programming.

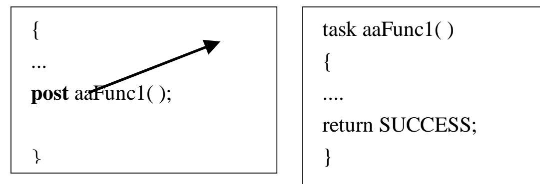
Table 2. Interfaces in nesC

Interface Type	Usage
ADC	Analog-digital conversion interface
I2C	I2C serial bus protocol interface
Leds	LED control interface
LogData	Interface for log data management
Radio	Interface for wireless communication
Receive	Interface for receiving messages
Send	Interface for transmitting messages
StdControl	Standard control interface
BitVector	Interface for handling bit vectors
Timer	Interface for controlling timer

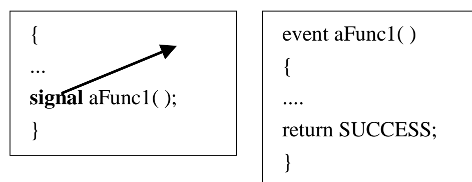
Copyright ⓒ 2015 SERSC

3.1.4. Task, Event, Command

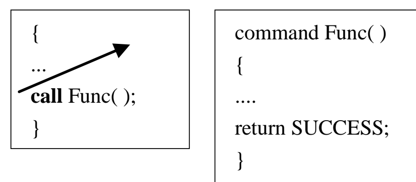
Tasks to be used shall be declared in advance. The Task has no factors and a function with a return value of void. 'Post' keyword shall be used to call a Task and placed it in the task queue.



TinyOS tasks are consecutively performed based on FIFO (First In First Out) schedule by the task scheduler. It is interrupted by the Event and the 'signal' keyword shall be used to call an Event function implemented by user interfaces.



Command functions defined in the interface shall be called by using 'call' keywords.



3.1.5. Major Terms used in the nesC

O components : Module + Configuration
O module : Basic element of the component
O interface : Declare events and commands functions
A "connector" among components
O provides : Component provider
O uses : Component user
O as : Define other name
O command : Command function defined in the interface
O event : Event function defined in the interface
O implementation : Where program variables and actual codes are written
O configuration : Connection with other components
O call: Execute command functions
O signal: Execute event functions
O post: Put task functions to be executed into the task queue
O task: Functions to be consecutively executed in the task queue
O includes: Declare headers to be use

4. Case Studies of Wireless Sensor Control

Cygwin, TinyOS and Java SDK are installed in the Windows XP environment to establish the development and test environment, and USB drivers are installed to control base motes connected to the PC for communication with the wireless sensors in Figure 4.

Copyright ⓒ 2015 SERSC

Figure 4. Base Mote and Wireless Sensor

Serial telecommunication is performed with the PC as in Figure 5 and the base mote exchanges Zigbee wireless communication with the wireless sensors.

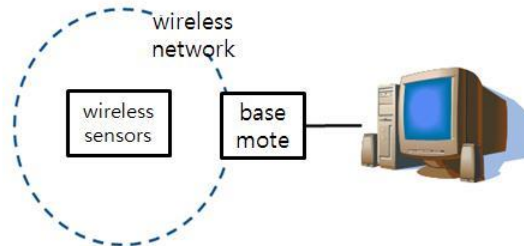


Figure 5. Control Environment for the Wireless Sensor

Microphone (sound), temperature, intensity, humidity and IR signal tests are performed for wireless sensors manufactured by 2 local companies [13, 14] and the procedure is almost similar except for controlling sensors for each manufacturer (the compile process, a process putting execution files into the mote). The study of receiving distance with the base mote showed that there were no signal transmission and receiving issues within a 15m distance. However, the signal sensitivity reduced between 15 and 20m and stopped operation at farther than 20m. Also, in measurements using hurdles, like concrete walls, showed that the receiving distance became shorter even though it was possible to transmit signals. Transmitting and receiving signals were not done in an elevator but worked in a refrigerator. Also, the system worked at a normal walking speed but the signal became weaker while running. Executing an oscilloscope program written by Java showed in Figure 6 that signals were transmitted through wireless sensors.

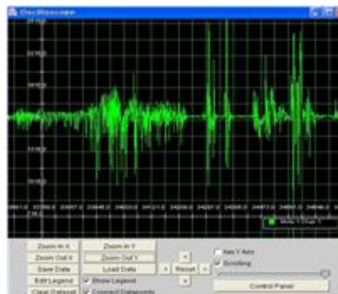


Figure 6. Oscilloscope with Wireless Sensor Signal

5. Conclusion

The wide penetration of computers and the internet have had a positive impact on society, including upgrading efficiency and productivity and the easy exchange of information among people at a long distance. Recently, studies have focused on attaching wireless sensors to all materials, people, robots and environment and various fields (all fields of industry like homes, offices, logistics, distribution, monitoring environment, intelligent homes, military, aerospace, ships and vehicles, *etc.*), integrating various networks like computers, wired and wireless internet service. Many countries compete by participating in researching and developing fundamental and applied technologies in hopes of them emerging as future core technologies.

Copyright ⓒ 2015 SERSC

Developing USN-related technologies is going to accelerate and it is imperative to develop and prepare technologies and business models.

Acknowledgement

This thesis was supported by the research funding of Youngsan University.

References

[1] L. H. Diakite, L. Yu and A. Halidou, "Improvement of Energy Efficiency in Wireless Sensor Network (WSN)", JDCTA, vol. 8, no. 2, **(2014)**, pp. 119-125.
[2] M. Alwadi and G. Chetty, "Feature Selection and Energy Management for Wireless Sensor Networks", International Journal of Computer 46 Science and Network Security, vol. 12, no. 6, **(2012)**, pp. 46-51.
[3] G. H. Gao and T. Yang, "Implementation of the Wireless Sensor Network Routing Protocols Based on TinyOS", JCIT, vol. 8, no. 5, **(2013)**, pp. 474-483.
[4] J. Song, P. Ma and S. Park, "Micro sized OS for Sensor Networks", Telecommunication Software, **(2007)**, pp. 26-35.
[5] Y. H. Kim, "Geometry-Based Sensor Selection for Large Wireless Sensor Networks", Journal of Information and Communication Convergence Engineering, vol. 12, no. 1, **(2014)**, pp. 8-13.
[6] T. S. Jin, "Position Estimation of Mobile Robots using Multiple Active Sensors with Network", International Journal of Fuzzy Logic and Intelligent Systems, vol. 11, no. 4, **(2011)**, pp. 217-309.
[7] S. Naveed and N. Y. Ko, "Analysis of Indoor Robot Localization Using Ultrasonic Sensors", International Journal of Fuzzy Logic and Intelligent Systems, vol. 14, no. 1, **(2014)**, pp. 75-72.
[8] http://docs.tinyos.net/index.php/TinyOS_Tutorials.
[9] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer and D. Culler, "TinyOS: An operating system for sensor networks", In Ambient Intelligence, Springer Berlin Heidelberg, **(2005)**, pp. 115-148.
[10] http://www.tinyos.net/tinyos-1.x/doc/nesc/ref.pdf.
[11] http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf.
[12] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer and D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems", In Proceedings of Programming Language Design and Implementation (PLDI), **(2003)**.
[13] http://www.hanback.co.kr.
[14] http://www.hybus.net.

Author

Jin-whan Kim received a BS degree in computer and statistics from Pusan National University in 1989, and MS and Ph.D. in computer and science from Yonsei University, Seoul, Korea, in 1992 and Pusan National University, Pusan, Korea, in 2006, respectively. He is an associate professor in Youngsan University and as a CEO in MMiGroup Co., Ltd. His research areas are dynamic signature verification, on-line character recognition, voice processing, multi-modal biometric system, ubiquitous computing and wired/wireless Internet security.

Phone: +82-55-380-9331
E-Mail: kjw@ysu.ac.kr

 Copyright ⓒ 2015 SERSC