OPERATING SYSTEM FOR WIRELESS SENSOR NETWORKS AND AN EXPERIMENT OF PORTING CONTIKIOS TO MSP430 MICROCONTROLLER

Thang Vu Chien[1], Hung Nguyen Chan[2], and Thanh Nguyen Huu[2]

[1]Thai Nguyen University of Information and Communication Technology, Thai Nguyen, province Vietnam, 23999, Vietnam
[2]Hanoi University of Science and Technology, Số 1 Đại Cồ Việt, Hai Bà Trưng, Hanoi, 71000, Vietnam

E-mail: vcthang@ictu.edu.vn

Abstract

Wireless Sensor Networks (WSNs) consist of a large number of sensor nodes, and are used for various applications such as building monitoring, environment control, wild-life habitat monitoring, forest fire detection, industry automation, military, security, and health-care. Each sensor node needs an operating system (OS) that can control the hardware, provide hardware abstraction to application software, and fill in the gap between applications and the underlying hardware. In this paper, researchers present OS for WSNs and an experiment of porting contikiOS to MSP430 microcontroller which is very popular in many hardware platforms for WSNs. Researchers begin by presenting the major issues for the design of OS for WSNs. Then, researchers examine some popular operating systems for WSNs including TinyOS, ContikiOS, and LiteOS. Finally, researchers present an experiment of porting ContikiOS to MSP430 microcontroller.

Keywords: *operating system, porting contikiOS to MSP430, wireless sensor networks*

Abstrak

Wireless Sensor Networks (WSNs) terdiri dari sejumlah besar *sensor nodes*, dan digunakan untuk berbagai aplikasi seperti pemantauan gedung, pengendalian lingkungan, pemantauan kehidupan habitat liar, deteksi kebakaran hutan, otomatisasi industri, militer, keamanan, dan kesehatan. Setiap *sensor node* memerlukan sistem operasi (SO) yang dapat mengontrol *hardware*, menyediakan abstraksi *hardware* untuk aplikasi perangkat lunak, dan mengisi kesenjangan antara aplikasi dan *hardware*. Dalam penelitian ini, peneliti menyajikan SO untuk WSNs dan percobaan dari *port* contikiOS untuk MSP430 mikrokontroler yang sangat populer di *platform hardware* untuk WSNs. Peneliti memulai dengan menghadirkan isu utama yaitu desain SO untuk WSNs. Lalu, peneliti memeriksa beberapa sistem operasi populer untuk WSNs, termasuk TinyOS, ContikiOS, dan LiteOS. Akhirnya peneliti menyajikan sebuah percobaan dari *port* ContikiOS untuk MSP430 mikrokontroler.

Kata Kunci: *sistem operasi, port ContikiOS untuk MSP430, wireless sensor networks*

1. Introduction

A WSN is generally composed of a centralized station (sink) and tens, hundreds, or perhaps thousands of tiny sensor nodes. With the integration of information sensing, computation, and wireless communication, these devices can sense the physical phenomenon, (pre-)process the raw information, and share the processed information with their neighboring nodes.

Typical sensor nodes are equipped with a sensor, a microprocessor or microcontroller, a memory, a radio transceiver, and a battery. Therefore, these hardware components should be organized in a way that makes them work correctly and effectively without a conflict in support of the specific applications for which they are designed. Each sensor node needs an OS that can control the hardware, provide hardware abstraction to application software, and fill in the gap between applications and the underlying hardware.

The basic functionalities of an OS include resource abstractions for various hardware devices, interrupt management and task scheduling, concurrency control, and networking support. Based on the services provided by the OS, application programmers can conveniently use high-level application programming interfaces (APIs) independent of the underlying hardware.

The traditional OS is system software that operates between application software and

This paper is the extended version from paper titled "A Comparative Study on Operating System for Wireless Sensor Networks" that has been published in Proceeding of ICACSIS 2012.

hardware and is often designed for workstations and PCs with plenty of resources. This is usually not the case with sensor nodes in WSNs. There are also embedded operating systems such as VxWorks [1] and WinCE [2], none of which is specially designed for data-centric WSNs with constrained resources. Sensors usually have a slow processor and small memory, different from most current systems. These parameters should be kept in mind in the process of OS design for WSN nodes.

In this paper, researchers identify several major issues for the design of OS for WSNs. By examining some existing operating systems for WSNs, researchers hope this research may allow research community to know the strengths and weaknesses a number of different operating systems.

2. Methodology

Traditional operating systems are system software, including programs that manage computing resources, control peripheral devices, and provide software abstraction to the application software. Traditional OS functions are therefore to manage processes, memory, CPU time, file system, and devices. This is often implemented in a modular and layered fashion, including a lower layer of kernels and a higher layer of system libraries. Traditional operating systems are not suitable for WSNs because WSNs have constrained resources and diverse data-centric applications, in addition to a variable topology. WSNs need a new type of operating system, considering their special characteristics. There are several issues to consider when designing sensor network OS.

First, process management and scheduling. The traditional OS provides process protection by allocating a separate memory space (stack) for each process. Each process maintains data and information in its own space. But this approach usually causes multiple data copying and context switching between processes. This is obviously not energy efficient for WSNs. Sensor network operating systems should provide efficient resource management mechanisms in order to allocate microprocessor time and limited memory. The CPU time and limited memory must be scheduled and allocated for processes carefully to guarantee fairness (or priority if required).

Second, memory management. Memory is often allocated exclusively for each process/task in traditional operating systems, which is helpful for protection and security of the tasks. Since sensor nodes have small memory, another approach, sharing, can reduce memory requirements.

Third, kernel model. The event-driven and finite state machine (FSM) models have been used to design microkernels for WSNs. The event-driven model may serve WSNs well because they look like event-driven systems. An event may comprise receiving a packet, transmitting a packet, detection of an event of interest, alarms about energy depletion of a sensor node, and so on. The FSM-based model is convenient to realize concurrency, reactivity, and synchronization.

Fourth, energy efficiency. Sensor nodes provide very limited battery lifetime. On the other hand, guaranteeing sensor networks to operate for 3 to 5 years is a very desirable objective. Sensor network OS should support power management, which helps to extend the system lifetime and improve its performance. For example, the operating system may schedule the process to sleep when the system is idle, and to wake up with the advent of an incoming event or an interrupt from the hardware.

Fiveth, application program interface. Sensor nodes need to provide modular and general APIs for their applications. The APIs should enable applications access the underlying hardware. This may allow access and control of hardware directly, to optimize system performance.

Sixth, code upgrade and reprogramming. Since the behavior of sensor nodes and their algorithms may need to be adjusted either for their functionality or for energy conservation, the operating system should be able to reprogram and upgrade.

Seventh, small footprint. The limited memory of only a few kilobytes on a sensor node necessitates the OS to be designed with a very small footprint. It is a fundamental characteristic of a sensor network OS and is the primary reason why so many sophisticated embedded operating systems can not be easily ported to sensor nodes.

Eighth real-time guarantee. As most sensor network applications such as surveillance tend to be time-sensitive in nature where packets must be relayed and forwarded on a timely basis, real-time guarantee is a necessary requirement for such applications.

The last is reliability. In most applications, sensor networks are deployed once and intended to operate unattended for a long period of time. OS reliability is of great importance to facilitate developing complex WSN software, ensuring the correct functioning of WSN systems.

Over the years, researchers have seen many operating systems for wireless sensor networks such as TinyOS, Contiki, SOS, Mantis OS, Nano-RK, RETOS and LiteOS [3]. In this paper,

researchers present only TinyOS, Contiki (many interested OS users), and LiteOS (the newest sensor network OS).

TinyOS is developed in UC Berkeley [4]. The design of TinyOS allows application software to access hardware directly when required. TinyOS is a tiny microthreaded OS that attempts to address two issues: how to guarantee concurrent data flows among hardware devices, and how to provide modularized components with little processing and storage overhead. These issues are important since TinyOS is required to manage hardware capabilities and resources effectively while supporting concurrent operation in an efficient manner. TinyOS uses an event-based model to support high levels of concurrent application in a very small amount of memory. Compared with a stack-based threaded approach, which would require that stack space be reserved for each execution context, and because the switching rate of execution context is slower than in an event-based approach, TinyOS achieves higher throughput. It can rapidly create tasks associated with an event, with no blocking or polling. When CPU is idle, the process is maintained in a sleep state to conserve energy.

Figure 1 illustrates the basic architecture of TinyOS. TinyOS includes a tiny scheduler and a set of components.

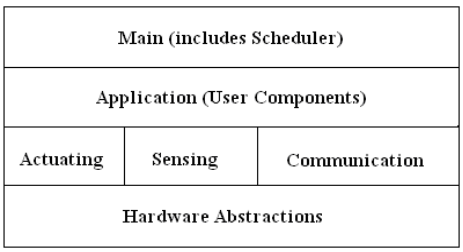

Figure 1. TinyOS architecture.

The scheduler schedules operation of those components. Each component consists of four parts: command handlers, event handlers, an encapsulated fixed-size frame, and a group of tasks. Commands and tasks are executed in the context of the frame and operate on its state. Each component will declare its commands and events to enable modularity and easy interaction with other components. The current task scheduler in TinyOS is a simple FIFO mechanism whose scheduling data structure is very small, but it is power efficient since it allows a processor to sleep when the task queue is empty and while the peripheral devices are still running. The frame is fixed in size and is assigned statically. It specifies the memory requirements of a component at compile time and removes the overhead from dynamic assignment. Commands are nonblocking requests made to the low-level components. Therefore, commands do not have to wait a long time to be executed. A command provides feedback by returning status indicating whether it was successful (e.g., in the case of buffer overrun or of timeout). A command often stores request parameters into its frame and conditionally assigns a task for later execution. The occurrence of a hardware event will invoke event handlers. An event handler can store information in its frame, assign tasks, and issue high-level events or call low-level commands. Both commands and events can be used to perform a small and usually fixed amount of work as well as to preempt tasks. Tasks are a major part of components. Like events, tasks can call low-level commands, issue high-level events, and assign other tasks. Through groups of tasks, TinyOS can realize arbitrary computation in an event-based model. The design of components makes it easy to connect various components in the form of function calls. In order to provide a better support for the component architecture and execution model of TinyOS, the nesC language [5] was designed for programming based on TinyOS. TinyOS has a component-based programming model, codified by the nesC language.

This WSN operating system defines three types of components: hardware abstractions, synthetic hardware, and high-level software components. Hardware abstraction components are the lowest-level components. They are actually the mapping of physical hardware such as Input/Output (I/O) devices, a radio transceiver, and sensors. Each component is mapped to a certain hardware abstraction. Synthetic hardware components are used to map the behavior of advanced hardware and often sit on the hardware abstraction components. TinyOS designs a hardware abstract component called the Radio-Frequency Module (RFM) for the radio transceiver, and a synthetic hardware component called radio byte, which handles data into or out of the underlying RFM.

TinyOS supports a wide range of hardware platforms and has been used on several generations of sensor nodes. Supported processors include the Texas Instruments MSP430 and the Atmel AVR. TinyOS applications may be compiled to run on any of these platforms without modification.

The Contiki operating system is an open source operating system for networked embedded systems in general, and wireless sensor nodes in particular. It is developed by a team of developers

from the industry and academia [6]. The Contiki project is lead by Adam Dunkels.

Typically, a running Contiki system consists of the kernel, libraries, the program loader, and a set of processes. Communication between processes always goes through the kernel, which does not provide a hardware abstraction layer, but lets device drivers and applications communicate directly with the hardware.

A process is defined by an event handler function and an optional poll handler function. The process state is held in the process' private memory and the kernel only keeps a pointer to the process state. All processes share the same address space and do not run in different protection domains. Interprocess communication is done by posting events.

Looking at it from a higher perspective, the Contiki system is partitioned into two parts: the core and the loaded programs as shown in figure 2. Typically, the core consists of the Contiki kernel, the program loader, the most commonly used parts of the language run-time and support libraries, and a communication stack with device drivers for the communication hardware. The core is compiled into a single binary image and is usually not modified after deployment, although it is possible to use a special boot loader to overwrite or patch the core. Programs are loaded into the system by the program loader. The program loader is in charge of loading or unloading the programs into the system either by using the communication stack or directly attached storage (such as EEPROM).

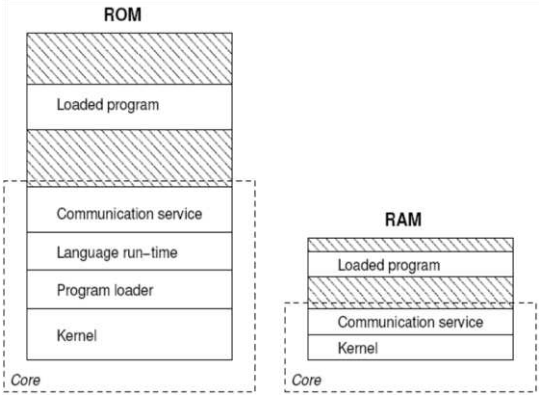

Figure 2. Contiki system.

The Contiki kernel consists of a lightweight event scheduler that dispatches events to running processes and periodically calls processes' polling handlers. All program execution is triggered either by events dispatched by the kernel or through the polling mechanism. The kernel does not preempt an event handler once it has been scheduled. The kernel supports two kinds of events: asynchronous and synchronous events. In addition to the events, the kernel provides a polling mechanism. Polling can be seen as high priority events that are scheduled in-between each asynchronous event.

Contiki was the first operating system for wireless sensor nodes that provided IP communication with the uIP TCP/IP stack. In 2008, the Contiki system incorporated uIPv6, the world's smallest IPv6 stack. The footprints of the uIP and uIPv6 stacks are small: less than 5 kB for the uIP stack and approximately 11 kB for uIPv6. This makes them suitable for use in the constrained environment of a wireless sensor node.

Both the Contiki system and applications for the system are implemented in the C programming language. Because Contiki is implemented in C, it is highly portable. Contiki has been ported to a number of microcontroller architectures, including the Texas Instruments MSP430 and the Atmel AVR.

LiteOS [7], developed in the University of Illinois at Urbana Champaign, is designed to provide a traditional Unix-like environment for programming WSN applications. It includes: a hierarchical file system and a wireless shell interface for user interaction using UNIX-like commands; kernel support for dynamic loading and native execution of multithreaded applications; and online debugging, dynamic memory, and file system assisted communication stacks. LiteOS also supports software updates through a separation between the kernel and user applications, which are bridged through a suite of system calls.

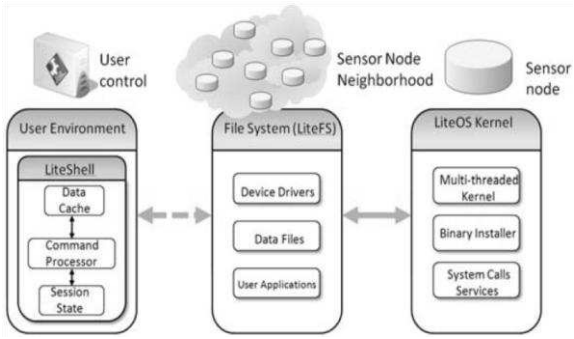

Figure 3. LiteOS architecture.

Figure 3 shows the overall architecture of the LiteOS operating system, partitioned into three subsystems: LiteShell, LiteFS, and the Kernel. Implemented on the base station PC side, the LiteShell subsystem interacts with sensor nodes (motes) only when a user is present. Therefore,

LiteShell and LiteFS are connected with a dashed line in this figure.

The LiteShell subsystem provides Unix-like commandline interface to motes. This shell runs on the base station PC side. Therefore, it is a front-end that interacts with the user. The motes do not maintain command-specific state, and only respond to translated messages (represented by compressed tokens) from the shell, which are sufficiently simple to parse.

The interfaces of LiteFS provide support for both file and directory operations. The APIs of LiteFS can be exploited in two ways; either by using shell commands interactively, or by using application development libraries.

The kernel subsystem of LiteOS takes the thread approach, but it also allows user applications to handle events using callback functions for efficiency. It implements both priority-based scheduling and round-robin scheduling in the kernel. It also support dynamic loading and un-loading of user applications, as well as a suite of system calls for the separation between kernel and applications.

The LiteOS 2.0 is the latest version of LiteOS. It runs on the following platforms: MicaZ as target board, and MIB510/MIB520 as programming boards. Unlike 1.0, LiteOS 2.0 is closely integrated with AVR Studio 5.0. This brings multiple advantages, such as IDE editing, debugging, and built-in JTAG support. Due to a problem of compatiability between IRIS and AVR Studio, IRIS mote support will be added in version 2.1.

Table I provides a comparison between TinyOS, Contiki, and LiteOS by examining some important OS features.

3. Results and Analysis

This section is devided into three sections. First, introduction to msp430 microcontroller. The Texas Instruments MSP430 family of ultralow power microcontroller consists of several devices featuring different sets of peripherals targeted for various applications. The architecture, combined with five low power modes is optimized to achieve extended battery life in portable measurement applications. The device features a powerful 16-bit RISC CPU, 16-bit registers, and constant generators that contribute to maximum code efficiency. The digitally controlled oscillator (DCO) allows wake-up from low-power modes to active mode in less than 6 μs [8]. The typical operating conditions of the MSP430 microcontroller are presented in detail in table II.

The MSP430 family microcontrollers are used in many hardware platforms for wireless sensor network such as Eyes, EyesIFX, TelosB, Tmote Sky, Shimmer, Zolertia [9]. They permit wireless sensor nodes to operate in the low power mode. So wireless sensor nodes can run for years on a single pair of AA batteries. In this paper, researchers present an experiment of porting ContikiOS to MSP430F1611 microcontroller.

Second, designing hardware with msp430f1611. Researchers design a hardware for porting ContikiOS to MSP430F1611. Figure 4 illustrates the functional block diagram of the hardware and figure 5 illustrates the printed circuit board of the hardware.

TABLE I
A COMPARISON BETWEEN TINYOS, CONTIKIOS AND LITEOS

Features	TinyOS	ContikiOS	LiteOS
Publication (Year)	ASPLOS (2000)	EmNets (2004)	IPSN (2008)
Website	www.tinyos.net	www.sics.se/contiki	www.liteos.net
Static/Dynamic System	Static	Dynamic	Dynamic
Monolithic/Modular System	Monolithic	Modular	Modular
Networking Support	Active Message	uIP. uIPv6, Rime	File-Assisted
Real-Time Guarantee	No	No	No
Language Support	nesC	C	LiteC++
Event Based Programming	Yes	Yes	Yes (through callback functions)
Multi-Threading Support	Partial (through TinyThreads)	Yes (also supports Protothreads)	Yes
Wireless reprogramming	Yes	Yes	Yes
File Sytem	Single level (ELF, Matchbox)	Coffee	Hierarchical Unix-like
Platform Support	Mica, Mica2, MicaZ, TelosB, Tmote, XYZ, IRIS, Tinynode, Eyes, Shimmer	Tmote, TelosB, ESB, AVR MCU, MSP430 MCU	MicaZ, IRIS, AVR MCU
Simulator	TOSSIM, Power Tossim	Cooja, MSPSim, Netsim	Through AVRORA

The main aim of this design is researchers can program MSP430F1611 microcontroller with using ContikiOS. The program after compiling with using ContikiOS that will create a *.hex file. This file can be written to the ROM of MSP430F1611 microcontroller through JTAG connector. The result of the program can be

observed through LED display including 8 single LEDs.

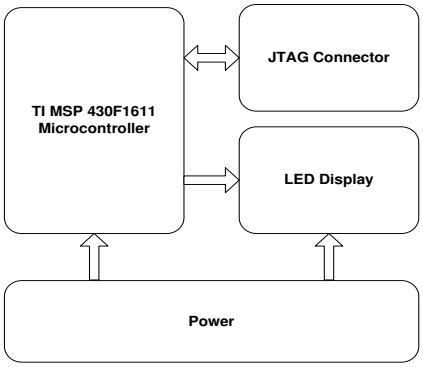

Figure 4. Functional block diagram of the hardware.

TABLE II
TYPICAL OPERATING CONDITIONS OF MSP430 MICROCONTROLLER

	Min	Nom	Max	Unit
Supply voltage during program execution	1.8		3.6	V
Supply voltage during flash memory programming	2.7		3.6	V
Operating free air temperature	-40		85	°C
Low frequency crystal frequency		32.768		kHz
Active current at Vcc = 3V, 1MHz		500	600	µA
Sleep current in LPM3 Vcc = 3V, 32.768KHz active		2.6	3.0	µA
Wake up from LPM3 (low power mode)			6	µs

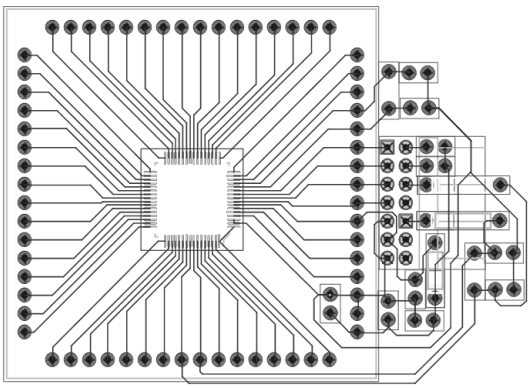

Figure 5. Printed circuit board of the hardware.

Third, porting contikios to msp430f1611. Researchers write a contiki program (blink.c) that controls LED effect with researchers hardware. Figure 6 illustrates this program. In this program, LEDs are turned on respectively. After all 8 LEDs are bright, the program turns off them. This process is regularly repeated.

This program is only a experiment which helps us to know how to write a contiki program. After the program is compiled, a hex file (blink.hex) is created. Finally, this file is written to the ROM of MSP430F1611. The result shows that 8 LEDs are controlled correctly.

However, researchers did not use communication stack in this program. In the future work, researchers will manufacture some wireless sensor nodes which have radio modules. Researchers hope that researchers can program and test with communication stack in ContikiOS for this future hardware.

```
PROCESS(blink, "blink");
AUTOSTART_PROCESSES(&blink);
static void show_leds(int v)
{
  LEDS_PxOUT = ~v;
}
PROCESS_THREAD(blink, ev, data)
{
  PROCESS_BEGIN();
  LEDS_PxDIR = 0xff;
  LEDS_PxOUT = 0xff;
  while(1)
  {
    static struct etimer e;
    static int i = 1;
    etimer_set(&e, CLOCK_SECOND);
    PROCESS_WAIT_EVENT();
    i = i << 1;
    show_leds(i);
    if(i == 0)
    {
      show_leds(i);
      i =1;
    }
  }
  PROCESS_END();
}
```

Figure 6. Contiki program controlling LED effect.

4. Conclusion

In this paper, researchers present operating system for wireless sensor networks and several major design issues with sensor network operating system. By examining some existing sensor network operating systems, researchers know the strengths and weaknesses of a number of different operating systems. An experiment of porting ContikiOS to MSP430 microcontroller are also provided. Researchers hope this research may allow research community to know the features of a number of different operating systems. So they can select a sensor network operating system that is the most appropriate for their applications.

References

[1] Vxwork, The RTOS That Powers More Than 1 Billion Embedded Systems Around the Globe, http://www.windriver.com/products/vxworks/, retrieved December 2, 2011.

[2] WinCE, http://www.microsoft.com/windowsembedded/en-us/windows-embedded.aspx, retrieved December 2, 2011.

[3] D. Wei & L. Xue, "Providing OS Support for Wireless Sensor Networks: Challenges and Approaches," *IEEE communications surveys & tutorials*, vol. 12, pp. 519-530, 2010.

[4] TinyOS, http://tinyos.net/, retrieved December 2, 2011.

[5] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, & D. Culler, "The nesC language: A holistic approach to networked embedded systems" *In Proceeding ACM PLDI*, pp. 1-11, 2003.

[6] A. Dunkels, B. Grönvall, & T. Voigt, "Contiki – a lightweight and flexible operating system for tiny networked sensors" *In Proceeding EmNets*, pp. 455-462, 2004.

[7] Q. Cao, A. Tarek, & J.A. Stankovic, "The LiteOS operating system: towards Unix-like abstractions for wireless sensor networks" *In Proceeding ACM/IEEE IPSN*, pp. 233-244, 2008.

[8] Texas Instruments MSP430x1xx Family User's Guide. http://ti.com/msp430, retrieved December 3, 2011.

[9] V.C. Thang, N.C. Hung, & N.H. Thanh, "A Comparative Study on Hardware Platforms for Wireless Sensor Networks," *International Journal on Advanced Science Engineering Information Technology*, vol. 2, pp. 70-74, 2012.