Design and Implementation of a Lightweight Operating System for Wireless Sensor Networks

Jang-Ping Sheu*, Bai-Kuan Hsu, Po-Chuan Lin, and Chia-Jen Chang
Department of Computer Science and Information Engineering
National Central University
**Email:sheujp@csie.ncu.edu.tw*

ABSTRACT

Wireless sensor networks are composed of large numbers on tiny networked devices, which are called sensor nodes. To implement a sensor network application, we need an operating system that provides a complete hardware control libraries and schemes to interactive with the sensor node hardware. In this paper, we design and implement a lightweight operating system (LOS) for wireless sensor networks. We implement four applications in our LOS and TinyOS, respectively. The code sizes used in our LOS can save about 42% to 67% memory space for the four applications compared to TinyOS.

1: INTRODUCTION

Wireless sensor networks (WSNs) are composed by large number of sensor nodes and can be used in a variety of applications such as military surveillance, air-conditioner control, building security, health monitor, and scientific investigations in harsh physical environments. To realize the above applications, it is very difficult for users directly programming to control sensor node hardware. Thus, several researchers developed the operating systems focused on WSNs. TinyOS [2] and Sensor Network Operating System (SOS) [4] provide powerful interface between users and the hardware. TinyOS is featured with NesC programming language, which was focused on network programming issues [1]. SOS is featured with a dynamic micro kernel [5] concept which saves the cost of memory space and wireless power consumption on kernel update.

In this paper, we design and implement a lightweight operating system (LOS) for WSNs. We implement our LOS in C language, which is already known and learned world widely. Without learning a new programming language, users can implement their new ideas quickly by using our LOS. The users can also have the standard ANSI C functions included in their programs, which save a lot time for them to complete their work. LOS uses static memory mapping that has optimized performance and code size. Event-driven is the concurrency mechanism of LOS which is suitable for WSNs. Kernel architecture adopted in LOS is monolithic which has better performance and compact size compared to micro kernel approach. The main design concept of LOS is to keep the kernel as lightweight as possible and the characteristics of highly flexible and easy modification. The LOS also can lower the cost of sensor nodes with a small memory space. The LOS is designed to be lightweight but it still supports enough functions for applications. The applications written in LOS are smaller than TinyOS in our experiments.

The rest of this paper is organized as follows. Section 2 presents the preliminary of WSN OS. Section 3 describes the system architecture and implementation of our LOS. Section 4 evaluates the system performance of LOS. Section 5 concludes this paper.

2: PRELIMINARY

In the following, we review the programming languages, memory mapping, concurrency control, and kernel architecture which are adapted by the existing operating systems of WSNs.

(a) Programming Languages:

The programming languages are directly influence on user's developing experience and efficiency. Since the low-level system implementation on WSN OS is usually needed, it is desirable if only one programming language is used through the entire system. Programming language also effect the performance and size of the system. NesC is focused on networked embedded systems [1]. TinyOS uses NesC for users programming language. However, in its kernel still uses C for system programming language. During the compiling procedure, the NesC interpreter will transfer the NesC program codes into C codes. If we use the C programming language instead of the NesC language, the code size is more compact than that generated by NesC. C is compact and performs better than NesC, but it lacks extendibility. We suggest users to write the application code in structure based formation, which can enhance the extendibility of applications.

(b) Memory Mapping:

The memory control system is essential for operating systems. Memory control system has to manage the whole memory related operations such as

memory allocation, memory protection, and garbage collection. There are two kinds of memory mapping: dynamic and static. In static memory mapping system, the compiler displays the memory usage after compiling. So, if the OS adapts static memory mapping, memory control system can be ignored or reduced which makes the kernel much smaller. Static memory mapping lacks of extendibility on kernel function update, recompiling the whole kernel is necessary if there needs functional update. The memory size of dynamic memory mapping is bigger than that of static one because the kernel needs to include the memory control system and increase the kernel size by a certain amount. The memory control system also brings extendibility for user to have functional update on system kernel without compiling the whole kernel. In order to minimize the kernel size, LOS adapts static memory mapping mechanism.

(c) Concurrency Control:

Concurrency is a property of systems which consist of multiple tasks executing simultaneously. Two concurrency control mechanisms [9][12] were discussed here. In time-sharing systems, each running task requires the hardware resources from the process scheduler. Process scheduler will dispatch timeslots for each task. A task may end up its computation and then leave the time slot. Otherwise, the scheduler will force the task to quit in the end of time slot. On the other hand, event-driven systems have their own control flow. CPU executes the main control flow mostly, and it is largely driven by external events.

Time-sharing systems have to include a scheduler and process handler for process creation and termination, which increase the kernel size but are extendable for many processes. Event-driven systems execute user defined event handlers and routine tasks, which save much kernel space without the scheduler and process handler. Since the user has to define each event handler, event-driven system suits only those have few events, like the WSNs. In general, an application WSN will have less than ten events. Comparing time-sharing system and event-driven system, the later has less memory request and more appropriate for WSN applications.

(d) Kernel Architecture:

Kernel is a piece of software responsible for the communication between hardware components and software components. There are two kernel architectures: micro kernel [5] and monolithic kernel [7]. The concept of micro kernel is to build a minimal kernel providing the most important system calls. The minimal set of services required in a micro kernel is memory space management, inter-module processing (IMP), and timer management. By linking optional kernel modules, users can have more functions to enhance the kernel. Micro kernel concept has better upgradeability. The communications between kernel modules depend on IMP. Since the IMP happens quite frequently, it will slow down the whole system performance [5]. The IMP also making the sensor network system may fail for some real-time constrained applications. The concept of monolithic kernel is to compile all the required services set into a larger kernel. The compiled kernel does not need IMP since the kernel itself contained all the required functions. So the performance of monolithic kernel does better than micro kernel if the two kernels contain the same service set.

These four aspects have great influence to the system performance and kernel size of operating systems. TinyOS uses static memory mapping, event-driven mechanism, and monolithic kernel. NesC uses component-based architecture, with many components statically linked with the kernel to a complete image of the system. The advantage of module-based design concept is easy of inheriting developed components, but on the other hand, the program grows large faster than traditional C program language. SOS [4] is the acronym of Sensor Network Operating System, developed by UCLA. SOS uses C as the programming language, dynamic memory mapping, and event-driven mechanism. SOS features in micro kernel architecture, which can reduce the system image size. It saves the time and power on modules update. The modules are dynamically linked and the memory is dynamically allocated. Therefore, the main kernel has to induce memory allocation and protection system, which also increase the risk of run-time error possibility.

Middleware consists of software acting as an intermediary between application components. Taking an example for quicker understanding, JAVA virtual machine is a middleware. Middleware have to adopt dynamic memory mapping method since the dynamic attribute of agent. The concurrency control mechanism of middleware is time-sharing because the support of multiple agents inside a node. The middleware Mate [10] and Agilla [11] are based on TinyOS and with their own script format. The major benefit of middleware concept is the high mobility of software agents which are positive on WSN applications. Middleware concept also brings many other drawbacks. First, integrated middleware makes the operating system image quite large. Second, middleware systems have their own script standards, users need to learn new script languages. Third, the sensor node will spend much computation power on middleware script interpreter which needs many cycles on a single operation string compare computation.

According to the above considerations, our LOS adapts C programming language, static memory mapping mechanism, event-driven concurrent control and monolithic kernel architecture to achieve lightweight OS.

3. IMPLEMENTATION OF LOS

In this section, we describe the hardware platform, software platform, and implementation details of our lightweight operating system (LOS). The hardware

platform consists of hardware programmer and Octopus sensor node. Hardware programmer is the bridge between development platform (a desktop or a laptop) and sensor node device. Octopus is our self-designed IEEE 802.15.4 compliant wireless sensor node, which is composed of AVR Atmega128 [6] and Chipcon's CC2420 [3], as shown in Fig. 3.1. The AVR Atmega128 is an 8-bit RISC processor. It can be programmed with 128 K bytes internal flash memory, 4 K bytes EEPROM, and 4 Kbytes internal SRAM. Chipcon's CC2420 is the industry's 2.4 GHz IEEE 802.15.4 compliant RF transceiver.

The software platform consists of toolkits and our designed operating system LOS. The toolkits include WinAVR library, AVR Studio, and GNU C Compiler. AVR programmer can load the compiled kernel image into our sensor nodes. The AVR Studio is the software to control AVR programmer. WinAVR is a suite of open source software development toolkit. It is designed for AVR RISC micro-controllers. The most important component inside WinAVR is WinAVR library. It is a set of C function library for lower level operation on AVR micro-controllers. It is essential for LOS since we use many functions from WinAVR library. GNU C Compiler is a suite of famous open source software (AKA GCC). Here we use AVR instruction compatible and Windows compatible version.

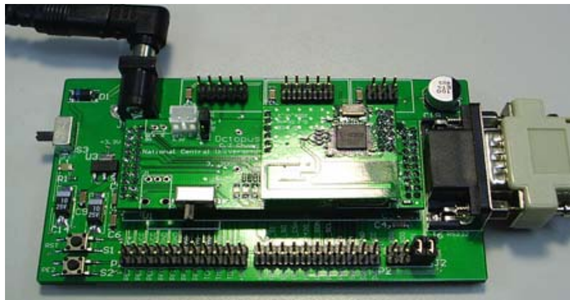

Figure 3.1 A photo snap of Octopus

Currently, the development platform is on Microsoft Windows XP 32-bit version. The development of the LOS system is focused on four main components: main program, Makefile, system code, and system header. In the main program, user can design process routine by his own algorithm and data structure. The main program eventually becomes a combination set of process routines including main schedule, event handler, and routine task. The system overview is shown in Fig. 3.2. Main schedule is the entrance of the whole process. In most situations, main schedule does not have many instructions to execute, but only a few routine maintain instructions. In some cases, the main schedule can even be an idle schedule. Another importance of main schedule is the declaration of other schedules such as event handler and routine task.

Event handler is the key idea of event-driven system. In time-sharing system, the RF device will hold a process in the process scheduler. The process scheduler waits for CPU to poll the data inside the RF device frequently. In event-driven system, the arrival of RF packet will trigger an interrupt to MCU. When the interrupt is triggered, the MCU will sense it and hold the current process, then jump into the event handler which was declared in main scheduler. After the event handler is executed, MCU will jump back to the original process.

Basically, only those acceptable interrupts can have their own event handlers. Here, we can find out the system actually will not complex due to the number of event handler will never exceed the number of acceptable interrupts. The event handlers must be declared in header files or in the main schedule. Routine task is another important and useful component in WSN operating systems. In most WSN applications, routines like polling data from a sensor device are highly required. For instance, if you want a series of temperature recording, you may create a task that polls the data from the temperature sensor every 10 minutes, and stores the value into an array. And then create another routine task, which sends the recorded values to sink node every 6 hours. These two routines formed a simple WSN application. The routine task was triggered by timer interrupt.

Makefile is a configuration file that defines many detail project specifications and compiler strategies. Compiler will read the Makefile before each compiling action is triggered. Each project will need an individual Makefile for configuration definition. In the Makefile we used, there exist 88 kinds of specifications and compiler strategies. Here, we list five specifications: MCU name, processor frequency, target filename, included C source files, and output format.

In our LOS, the system code was split into six main modules including delay control module, EEPROM control module, I/O control module, timer control module, USART control module, and RF control module. Delay control module holds an instruction for a while is tricky in WSN application programming. For example, if we want to display some information on the LED, the state should be hold for a while for human eyes. It is difficult to control the delay time precisely. It is also very important to confirm the delay control parameter inside the system is accurate. The mechanism we adopted for confirming delay period is to program an I/O pin and measure the delay time through oscilloscope. Users just need to input a counter for desired delay time.

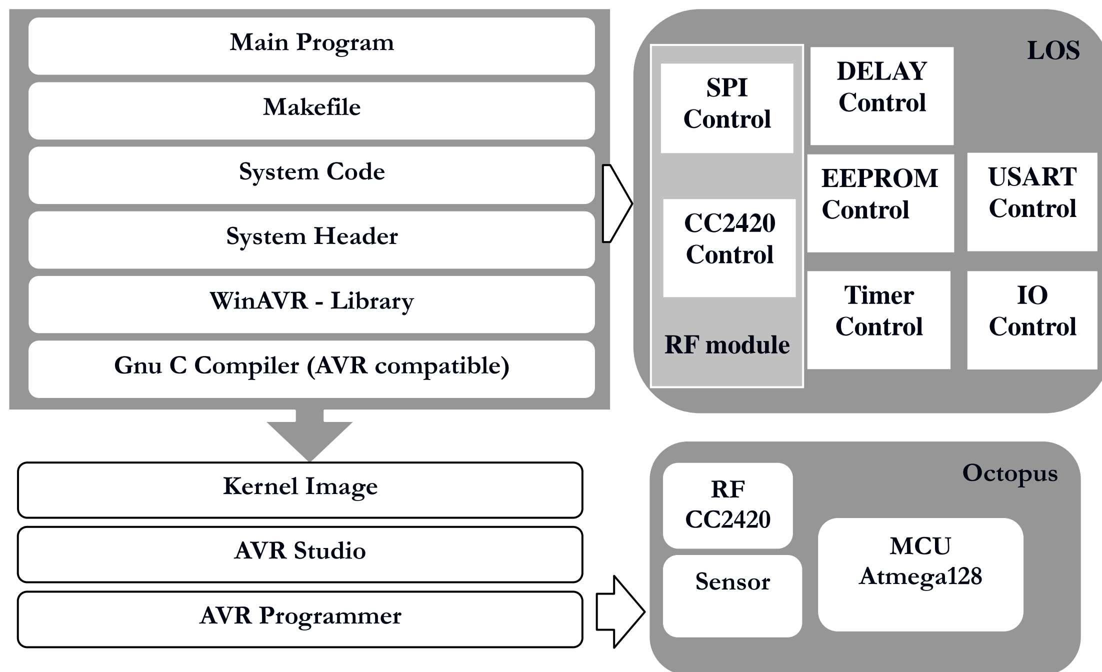

Figure 3.2 System diagram overview

EEPROM Control Module is a kind of non-volatile memory. The characteristic of non-volatile memory is that the data stored inside will not cleared if power is lost. User can store critical data inside EEPROM in case the power supply is lost or drained. When the power regained, the critical data can be restored from EEPROM. To be noticed, access EEPROM takes more time and power than RAM. EEPROM operations are not very complex. First, access the EEPROM when it is idle by checking the EECR register. Second, make sure the EEPROM operations were not being split by disable the interrupts. Third, write the data (EEDR = inByte) into the desired address (EEAR = adrEEPROM). Finally, execute the operation for MCU to write the EEDR register into address EEAR.

I/O control module in LOS is dedicated for easy I/O signal control for users in just one line of C program code. Users can access the IO pins of the MCU by a redefined variable name. For example, the green surface mounted LED pin is defined as GREENLED. This is because LOS integrated difficult control procedure into a single function call. By define the MCU name in the Makefile, the compiler will read the MCU information for each MCU hardware specifications. The direct address information is based on WinAVR library definitions, and then refined in LOS system headers. Timer control module is an important component in event-driven system. Atmega128 MCU provides two 8-bit timers and three 16-bit timers. Usually one timer will be taken as the system timer. The others can be assigned for different routine tasks like sensor data polling. Each timer can be triggered in three different ways. First, the input capture trigger. This means each tick of the timer will trigger the MCU to execute the service routine of this timer. Second, the compare match trigger. This kind of trigger is useful for periodic routines like sensor data polling. Third, the overflow capture trigger. Some functions like watchdog will need this kind of timer trigger.

USART control module is used to connect the sensor node and the desktop through the 9-pin D-shell connector (DE-9 connector). The programmability of the whole system is great enhanced. USART can send and receive data packets. User can transmit any variables and characters as he wants. Regarding to the poor information display ability on sensor node, this function turns a desktop computer into a powerful debug system. Receive data packets is also useful on sensor hardware. For example, by coordination through a desktop computer and a sensor node, user can build the desktop computer into an interactive communication interface.

RF control module is composed of two sub-modules, the SPI control module and CC2420 control module. These two sub-modules have dependency for each other so they must be included together. SPI control module is the module for controlling digital electronics that accepts clocked serial stream of bits. CC2420 control module is the collection of operation for CC2420 RF chip. Based on the SPI control module, the CC2420 control module provides all kinds of RF transmitter functions for WSN applications. The operations of RF transmitter can be sorted into three types: packet transmission, packet receiving, and status control. Packet transmission operation is not complex. It waits for the idle state and then sends the data and command into CC2420. Packet receiving is a rather complex operation. CC2420 has 128 K bytes packet buffer. If the packets arrive, but the MCU did not fetch in time, they will be stored in the buffers If the buffer overflows, the latest packet will overwrite the oldest

one. LOS provides 128 x *N* K bytes ring-queue buffer. Default value of *N* is set to 4. This means that packet buffer has been expend to 128 * 5 K bytes, and the packet inside CC2420 will be fetch into LOS system buffer automatically if the LOS buffer is not full.

In LOS, each module has one header file. Each header file includes several compiling essentials such as function declaration, variables definition, parameters explanation, function dependency, and function example, which is a brief example of the function call for users' reference.

The whole system has been explained in above. To create a new project, users need to lookup the manual to compose a new Makefile and a main program. In the Makefile users can define variables such as the project name, included modules, and library path. In the main program, users can create main process, periodical routines, and interrupt service routines for several different events.

4: KERNEL SIZE EVALUATION

In order to evaluate the performance of our lightweight operating system LOS, we developed four applications on LOS and TinyOS, respectively. These applications achieve the same goals by the same algorithms. We will compare their compiled kernel image sizes in LOS and TinyOS.

First, we implement an I/O control application, which controls a LED blinking and an USART control demo application, which control the USART on the sensor node. These two simple demo applications can be found inside TinyOS under application demonstration directory (/opt/tinyos1.x/apps/). The LED blinking application is to control one or many LEDs blinking. The LOS kernel image takes only 1.55 K bytes and TinyOS takes 4.7K bytes. The USART control application sends the value of an integer through USART. The LOS kernel image takes 2.89K bytes and TinyOS takes 5.55K bytes. Thus, our LOS can save 67% and 48% memory cost compared to that of TinyOS under the two simple applications.

Second, we implement a program that transmits data packets through wireless communication and displays the received packet information on LED. The result shows that the RF transmission application we implemented in TinyOS takes 26.6K bytes, and in LOS takes only 11.2 K bytes which saves 58% of memory cost. The above experimental results are drawn in Fig. 4.1.

Finally, we implement a time synchronization protocol (RSP), which is proposed in [8]. To perform this protocol, we need to implement three kinds of applications for the master node, slave node, and broadcast node, respectively. The image size of broadcast node in LOS takes 11.8 K bytes and in TinyOS takes 29.7 K bytes. The image size of master node in LOS takes 12.0 K bytes and in TinyOS takes 30.3 K bytes. The image size of slave node takes 26.7 K bytes in LOS and takes 45.7 K bytes in TinyOS. The experimental results are shown in Fig 4.2. Besides, studies have shown that the power consumption of memory scales roughly as the square root of the memory capacity [13]. This implies that LOS achieves further reduction in power.

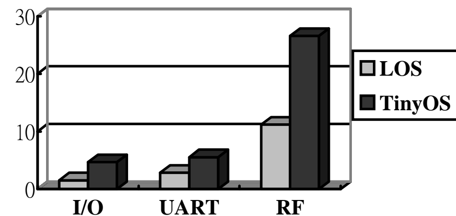

Figure 4.1 Kernel size comparison diagram for small size applications.

For the same hardware control modules such as I/O control and RF control, the experimental results show that our LOS indeed saves much memory space as comparing to TinyOS. Since the WSNs applications always need to access these hardware control modules, the code sizes of applications developed on LOS will be smaller than TinyOS.

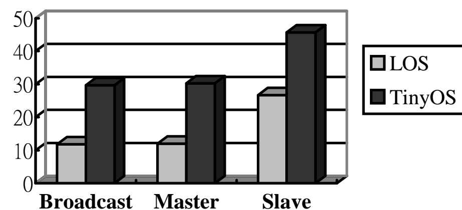

Figure 4.2 Kernel size comparison diagram for time synchronization application.

5: CONCLUSIONS

In this paper, we review the related work of WSNs operating systems in four aspects, programming language, memory mapping mechanism, concurrency handle mechanism, and kernel architecture. The lightweight OS was considered as a proper solution for WSNs since sensor devices provide only a little computation power, limited wireless communication bandwidth and battery energy. Lightweight OS also has more flexibility and easier for users to design applications. To achieve the lightweight feature, our LOS adapts C programming language, static memory mapping method, event-driven handler, and monolithic kernel architecture.

The LOS achieves the lightweight kernel sizes with available functionalities. The sensor nodes adapt LOS use less power and memory cost than TinyOS when executes the same applications. Users can experience longer battery maintaining periods, faster application developing on cheaper sensor nodes by using our LOS.

REFERENCES

[1] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC Language: A Holistic Approach to Networked Embedded Systems," *Proceedings of Programming Language Design and Implementation*, pp. 1-11, San Diego, CA, USA, June 2003.

[2] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System Architecture Directions for Networked Sensors", *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 93-104, Cambridge, MA, USA, November 2000.

[3] Chipcon IEEE 802.15.4 Compliant RF Transmitter Single Chip, http://www.chipcon.com/

[4] C. -C. Han, R. Kumar, R. Shea, E. Kohler and M. Srivastava, "A Dynamic Operating System for Sensor Nodes," *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services,* pp. 163-176, New York, NY, USA, 2005.

[5] D. L Black, D. B Golub, D. P. Julin, R.F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, and D. Bohman , "Micro Kernel Operating System Architecture and Mach," *Proceedings of the Workshop on Micro Kernels and Other Kernel Architectures*, pp. 11-30, April 1992.

[6] Atmel 8-bit RISC Processors, http://www.atmel.com/products/avr/

[7] M. Zec, "Implementing a Clonable Network Stack in the FreeBSD Kernel," *Proceedings of the USENIX 2003 Annual Technical Conference*, pp. 137-150, February 2003.

[8] J. P. Sheu, W. K. Hu, and J. C. Lin "Design and Implementation of a Ratio- ased Time Synchronization Protocol for Wireless Sensor Networks", *Proceedings of the 2nd Workshop on Wireless, Ad Hoc, and Sensor Networks*, pp. 65-72, Jhongli Taiwan, August 2006.

[9] D. C. Schmidt, "Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching," *in Pattern Languages of Program Design, Reading, MA: Addison-Wesley,* 1995.

[10] P. Levis and D. Culler, "Mate': A Tiny Virtual Machine for Sensor Networks," *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 85-95, San Jose, CA, USA, October 2002.

[11] C. -L. Fok, G.-C. Roman, and C. Lu, "Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications, "*Technical Report WUCSE-04-59*, Department of Computer Science and Engineering Washington University, St. Louis, USA, 2004.

[12] F. Zhao, L. J. Guibas, "Wireless Sensor Networks: An Information Processing Approach," *Morgan Kaufmann publishers*, 2003

[13] R. Evans & P. Franzon, "Energy Consumption Modeling and Optimization for SRAM's", Journal of Solid-State Circuits, Vol. 30, No. 5, May 1995.