U.P.B. Sci. Bull., Series C, Vol. 75, Iss. 1, 2013 ISSN 1454-234x

OPTIMIZING NODE RESPONSE USING REAL-TIME OPERATING SYSTEM IN WIRELESS SENSORS NETWORKS

Cătălin POPEANGĂ[1], Gabriel IONESCU[2], Radu DOBRESCU[3]

Scopul principal al soluției propuse pentru cercetare şi dezvoltare, într-o versiune experimentală, este implementarea/portarea unor sisteme de operare în timp real OSE[1][2][3] (Operating System Embedded) pe hardware-ul specific unei arhitecturi tipice nodului din rețelele wireless de sensori. Acest studiu de caz va compara implementările proprii ale sistemelor de operare OSE cu cele mai cunoscute sisteme de operare utilizate în rețelele de senzori.

The main goal of the proposed solution for research and development, in an experimental version, is the implementation/porting of real-time operating systems OSE [1][2][3] (Operating System Embedded) on a hardware specific to a typical node architecture in WSNs (wireless sensor networks). This case study will compare our own OSE operating system implementations with the most popular operating systems used in WSNs.

Keywords: real time operating system, wireless sensors networks

1. Introduction

In this paper, we will examine the improvements added to nodes architecture by adapting the OSE operating system to wireless sensors. In particular, it explores the real-time needs in actual embedded applications on different platforms used as sensors nodes. We will consider the optimization of the architecture starts from the beginning of the design (like: the operating system chosen or developed, new algorithm of coverage) until the hardware platforms.

This case study will focus on performances of the operating system in approximately worst real conditions. In the case study, we will not introduce network tests because they can be reduced to any core performance test in the end.

The scope of this case study is to verify if the value-aided (performance, framework for development) by the OSE is significant to port it for this kind of projects.

[1] Eng., Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: catalin.popeanga@gmail.com

[2] Prof., Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail, e-mail: gion@clicknet.com

[3] Prof., Faculty of Automatic Control and Computer Science, University POLITEHNICA of Bucharest, Romania, e-mail: rd_dobrescu@yahoo.com

As we know, WSN nodes are resource constrained and the key features of WSN operating systems should be resource-aware in addition, to be configurable to adapt to each situation. Most of traditional embedded RTOSs (Real-Time Operating Systems): VxWorks [4], QNX [5], µC/OS-II [6], RTLinux [7] are unfit for WSN because they are resource consuming. In addition, they do not meet the requirement of complex hard real-time applications, e.g. TinyOS [8]. Therefore, our objective is to choose and to adapt a configurable real-time dedicated operating system that can adapt to any platform used in a wireless sensor network, starting with the main node (the network manager) and finishing with the low power controller.

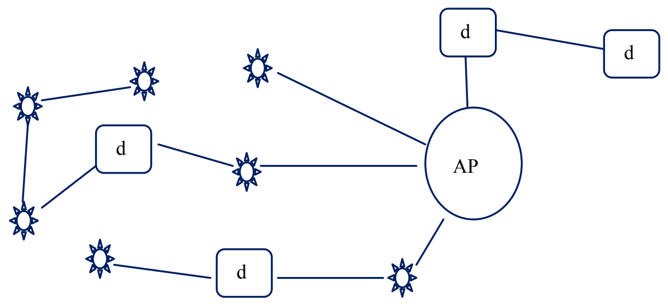
Two main aspects will be discussed regarding the wireless sensor networking technology: first, the implementation of a wireless sensors network with embedded architecture using OSE operating system [1] that offers specific facilities for integration real time applications on different platforms: microcontrollers, DSPs or powerful processors and second, the analysis of the power consumption and response latency to external inputs.

The paper is organized as following: section 2 describes the system architectures; sections 3, 4, 5 and 6 detail the performance evaluation for every architecture. Finally, we present the conclusions and the ongoing work.

2. General architecture of a wireless sensors network [9]

There are many challenges in designing operating system to manage an entire wireless sensor network: resource constraints, limited erasure/write cycles in external nonvolatile storage (e.g., flash), and lack of hardware features (e.g., privileged execution). However, these challenges are not singular in a wireless sensor network.

In addition, nodes with more complex operations needs exist in a network and from this point of view the necessity of hardware DSP (Digital Signal Processor) and an OS that can make these operations is required. In addition, the network will include an access point (AP) or the main node that manages his network, this node will need more computing power and memory manger for different applications. With these challenges, it is impossible to use traditional OS design techniques to implement OS protection, virtual memory, preemptions and to run the same high-end module application in a single OS that can run on all of these platforms. To cover all these challenges, we will select a commercial Real Time Operating System – OSE to compare with the freeware operation system used in academic environment.[11]



✹ Sensor node with low power consumption and low computation power.

d Sensor node with high computation power.

AP - Access point very high computation power, used to manage the network

Fig.1 WSN –general architecture

Three releases depending on the need of the complexity are distributed for OSE:

- High level RISC CPU (like PowerPC, MIPS, ARM and x86) usually dedicated to high-end application with the ***OSE Delta*** kernel;
- small and medium micro controllers (like Infineon, C166, NEC V850, ARM, Motorola M-Core, ColdFire, Hitachi) dedicated to medium range and highly embedded applications with the ***OSE Epsilon*** kernel;
- Digital Signal Processors (like TI TMSC5x, C6x, Motorola, Analog Devices) with the ***OSEck*** kernel.

An important point in this selection is that an OSE Epsilon application runs with no modification on OSE Delta and OSEck. Moreover, what is more important, the developer implements only once the protocol stack and it will run on all platforms.

OSE is fully pre-emptive (a process can be pre-empted between two assembly instructions), scheduling is based on the priority assigned to the processes (tasks) and the interrupt latency is extremely low. The OSE is a powerful platform for the design of real-time embedded systems. OSE's message based architecture instantly and seamlessly achieves simplicity in complex and distributed systems.

3. Performance evaluation

The most important evaluation will be on the OSE Epsilon, and the reason for this are the number of nodes, the challenge added by the limited resources available in hardware, like energy, memory and compute power.

The most used dedicated operating system in academic development is TinyOS[13]. This OS, more like task manager, will be the reference operating system in our evaluation.

The OS architecture [10] is exemplified in the next figure:

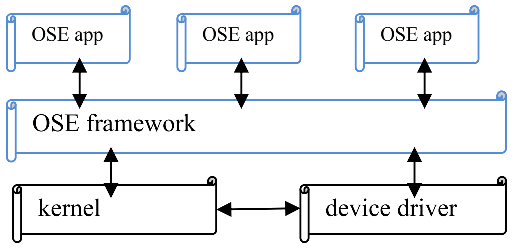


Fig.2 OS architecture

In the comparison, we will use several important parameters to determine if OSE is the right choice [11]:

- *Interrupt Latency:* time from assertion of hardware interrupts though start of ISR execution. Interrupt latency represents the time between the assertion of signal and the execution of the first instruction of the interrupt service routine assigned to handle corresponding events;

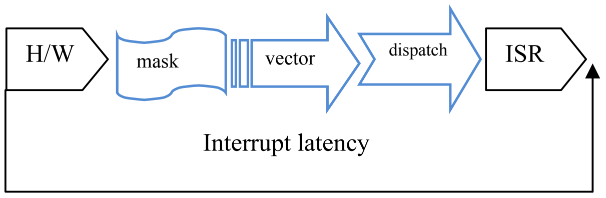


Fig.3 Latency in system

- *Worst Case:* The longest response latency time observed.

Context switch: This operation happens between two processes of the SO when the second one gets in a run able state and the first one is pulled from running to a wait state.

4. OSE Epsilon and Tiny OS

OSE Epsilon is a fast, small, low-cost RTOS optimized for resource constrained embedded microcontroller applications. This fully pre-emptive real-time kernel, written entirely in assembler, is optimized for each target processor, employs efficient system calls to reduce application code size, and occupies just 4 kbytes of memory.

OSE Epsilon speeds application development by combining simple yet powerful system calls with high-performance inter process communications services. In fact, with just eight simple system calls, most designers will have all they need to write the bulk of their application.

OSE Epsilon employs a simple, intuitive message passing programming model that makes it easy to break complex applications into simpler concurrent processes, each communicating via a high-speed, transparent, direct message passing protocol.

It is implemented in assembler and it is tuned for each processor, delivering the highest performance and lowest interrupt latency possible.

Table 1

OS performances

Operations	TinyOS ATMega128 6 MHz	OSE MSP430x5xx 8 MHz
Context Switch	23 µs	20 µs
Interrupt Latency	21 µs	17 µs
Worst Case Interrupt Latency	150 µs	50 µs

The worst-case scenario is the most significant measure in a benchmark. In our measurements, we observed that the value measured on OSE is less than the one from TinyOS and it depends only by the time spent in the dispatch routine. In the TinyOS, not fully preempted the latency depends also by the time spent in the current running task.

In the memory footprint, there are some differences gained by the TinyOS. It is more simple, does not have protection, is not fully preemptive and it does not offer the same framework for high-end processors.

Table 2

Occupied memory

Memory size (kernel)	TinyOS bytes	OSE bytes
Code	2145	4133
Data	50	210

The main difference between OSE and TinyOS can be described in few words.

OSE's architecture based on message signals achieves powerful simplicity, instantly responses in real-time systems. The dynamic runtime configuration enables faster and more reliable system deployment and maintenance. In addition, OSE has a sophisticated framework for creating error handlers at the system level. The error events during run-time will automatically activate an error handler. The error handler is called from either the application process or the OSE real-time kernel for microcontroller itself. As plus the OSE Epsilon with this error handler adds an important safety issue in user applications.

In OSE, there is no difference in creation of global or local visibility of private process into a user application. A local process can become globally visible to the whole system without re-creation.

5. OSE and Linux

The next benchmark shows the performances measured on the access point. The access point is a powerful processor used to monitor and to manage the whole network. The access point will have the OSE delta operating system.

The interrupt latency is measured on an ARM board and the concurrent operating system for OSE will be, of course, Linux 2.6.22 with real-time feature.

Table 3

Hardware device

Manufacturer/model	Freescale i.Mx21
CPU	ARM926EJ-S
Clock	266MHz
I/D cache	16K/16K
System memory	128MB SDRAM
	32 MB flash

Table 4

OS performances

Operations	Linux	OSE
Interrupt Latency (average)	7.1 μs	6.4 μs
Worst Case Interrupt Latency	25 μs	21 μs

In this case, there are many processes that execute in the same time so at the interrupt latency, they will add a new latency with a big implication in modifying the response time. This is the preemption latency. The preemption latency is defined as the time from ISR to the first instruction from the process destination.

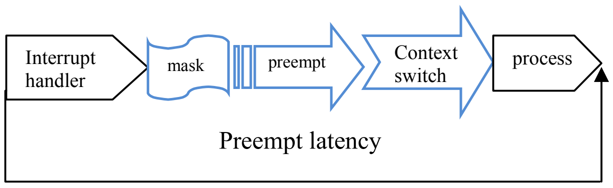


Fig.4 Preempt latency

In others words the interrupt latency represents the time needed to say "I know you are here", while preemption latency constitutes the time needed before "Now, let's work". In bare systems with couple of interrupting devices, the response time is less than microseconds or even nanosecond in worst cases. In theory, interrupt latency is independent of load; but in practice the system is overloaded with many queries of system resources and the latency will increase very fast. Our results are measured with the operating system in idle to get the fastest response it can get if it is in a wait state.

Table 5

OS performances

Operations	Linux	OSE
Preemption Latency	9.1 us	7.6 us

6. OSEck

For the special nodes with specific functions and with greater calculation power, another specialized operating system will be used and this is OSEck.

Like all members of the OSE family, OSEck employs a high-level message passing programming model that makes it easy to break complex applications into simpler concurrent processes, each communication via high-speed direct messages. This high level of abstraction makes complex applications easier to conceptualize, model, partition, and debug. It also provides transparency that separates applications from the details of the underlying hardware and physical topology, thereby making the resulting code more scalable and easier to migrate.

OSEck provides a simple yet powerful API that offers a high level of abstraction, typically enabling programmers to code the bulk of their application with just eight system calls. This versatile API, together with OSEck's high-level messaging protocol, reduces application size and complexity, and makes programs easier to maintain, read and understand. OSEck provides a subset of the full-featured OSE API, making easy to migrate applications between OSEck, OSE, and OSE Epsilon with few changes to the application code.

Communication throughput, overhead, and time response have always been concerning DSP applications that handle data streams on which the DSP performs real-time processing. These nodes will be used in wireless networks to transmit some alert messages or messages with little payload. Nevertheless, the data rates that it needs to handle are increasing dramatically through, for example, HD video streams and number of voice channels [11].

The throughput and delay allowed by the software needs to be kept good to handle both the wireless communication and data processing. In addition, the memory footprint is still a concern as it was for the simplest nodes. Total used memory adds material costs so it is better to have the executed real-time processing code and data placed in internal memory instead of external.

Another advantage of this is that it saves cycles, but can decrease the number of cache misses if the code is placed in external memory to be more compact.

These advantages and disadvantage have to be balanced against keeping overhead at a minimum to allow the DSP to spend as much time as possible doing real-time processing. It is better to compile the kernel with the specific tools for that architecture and used by the manufacturer to get the footprint as small as possible.

The OSEck kernel has, in the normal run, a routine for runtime error checks, an extended framework API for system calls unavailable with standard system calls. The performance will not be compared with any operating system because these devices depends of the application used and for wireless sensor network it has enough power to deal with the messages manager.

The most important thing in using the proposed operating system is to maintain a low footprint, as it shows the tests, to have the same framework easily to develop applications for working on the three architectures and also to have a good support if something is getting wrong in the applications development. An example of a board used in wireless networks is EVM6488 with TI C6488 DSP, which is mainly a DSP for wireless infrastructure baseband.

Table 6

OS memory footprint

Memory footprint	OSE (bytes)
Code	4380
Data	992

7. Conclusions

RTOSes evolved over time performing more application-specific functions. Unlike most traditional RTOSes, OSE was designed specifically with distributed, fault-tolerant telecommunications systems in mind. OSE's message-passing architecture and general approach to process and memory management, process scheduling, error handling, and distributed communications have made it the choice for millions of telecommunications applications worldwide.

Our own ported OSE operating systems on the specific architectures with the real-time feature and a common framework can be included in a wireless sensors network system becoming an alternative to other operating systems available in this industry.

The ongoing work will concern to implement a network stack in OSE operating system with a powerful coverage algorithm in parallel with selecting different power modes in processors, to decrease the power consumption and maintain the monitoring and connections in certain quality levels. Another aspect to keep in mind when trying to develop big projects is the support offered for using a certain operating system. This is an advantage for commercial operating systems but this implies also a bigger price. Depending on the destination of the project this is the most important one and it is one of the decisive attribute in choosing the operating system.

R E F E R E N C E S

[1] **** OSE – high-level operating system, http://www.enea.com/ , 2009*;

[2] *** *OSE Epsilon – microprocessor operating system, http://www.enea.com/, 2009;*

[3] *** *OSEck – DSP operating system, http://www.enea.com/ , 2009*;

[4] *J.J. Labrosse*, "MicroC/OS-II, The Real-time Kernel, R & D Books", Technical Document, Oct. 1998;

[5] *** VxWorks/x86 5.3.1 evaluation, Dedicated Systems Magazine, http://www.dedicated-systems.com , 2000;

[6] *** QNX4.25 Evaluation Executive Summary, Dedicated Systems Magazine, http://www.dedicated-systems.com , 2000 ;

[7] *V. Yodaiken*, "An Introduction to RTLinux", Technical Document, New Mexico Institute of Technology, Oct. 1997;

[8] *** *PSOS 2.2.6 Evaluation Executive Summary,* Dedicated Systems Magazine, http://www.dedicated-systems.com , 2000;

[9] *R. Dobrescu*, *M. Nicolae*, *F. Stoica* and *R. Varbanescu*, "Design of an Intelligent Sensor Network Node", in Proceedings of the 10thInternational Conference on Optimization of Electrical and Electronic Equipments OPTIM'06, Brasov, 2006;

[10] *K. Singh*, “Design and Evaluation of an Embedded Realtime Microkernel”, Master thesis, Faculty of the Virginia Polytechnic Institute and State University, 2002;
[11] *M. Barr*, “Choosing an RTOS”, Embedded Systems Programming, January 2003;
[12] *Maximilian Nicolae, Radu Dobrescu, Matei Dobrescu, Dan Popescu* - “Embedded Node around a DSP core for Mobile Sensor Networks over 802.11 infrastructure” - IEEE Proceedings of the 6th Symposium on Communication Systems, Networks and Digital Signal Processing, Graz 2008;
[13] *** TinyOS, User Guide, www.tinyos.net , 2009.