SenCore: an Embedded Operating System for Wireless Sensor Networks

Chia-Han Li Hsung-Pin Chang
Department of Computer Science,
National Chung Hsing University, Taichung, Taiwan, R.O.C.
s9356053@cs.nchu.edu.tw *hpchang@cs.nchu.edu.tw*

Abstract

Almost previous sensor network kernels can be roughly classified into two categories: event-driven and thread-driven. An event-based kernel is excellent in its tiny code size but fails to support time sensitive applications. In contrast, thread-driven is superior in its flexible programming construct but consumes more memory footprint. This motivates us to develop a new embedded kernel named SenCore.

SenCore provides flexible programming construct by thread-driven architecture and achieves memory efficiency by preemptive three-queue FIFO scheduler. Owing to our scheduling scheme, no matter how many tasks exist, SenCore requires only three task stacks. In contrast, previous thread-driven networked sensor kernels require each task has its own stack. To be a functional operating system, SenCore also provides IPC, synchronization, timer and memory management. Furthermore, to enable portability, all hardware-dependent details are encapsulated in the Hardware Abstraction Layer. Finally, to address the energy constraints, SenCore scheduler sleeps the microcontroller when there is no ready task in the system.

We have implement SenCore on the Mica2 mote and perform lots of performance evaluations. According to the performance results, SenCore is suitable for sensor network applications.

Keywords: Sensor Network Operating Systems, Embedded Systems, Sensor Networks.

1. Introduction

A wireless sensor network (WSN) consists of hundreds or thousands of sensor nodes that self-organize into a multi-hop wireless network. The basic building block of each sensor node includes a microprocessor, memory, RF transceiver, battery and sensor modules. Previously, there have been many embedded operating systems [13,17]. However, they are almost addressed for the ARM platform and are more suited to the word of embedded PC like PDAs, cell phones and set-top boxes, etc. None of them can be directly applied to WSN due to the computing and memory constraints in sensor nodes. For example, MIT' Mica2 Mote [2] only equips with a 128 KB program memory. However, the memory footprint of VxWorks is in the hundreds of kilobytes, which is more than an order of magnitude beyond a sensor node capability.

As a result, there have been many research projects that aims to design and develop their new sensor systems, for example, the U.C. Berkeley's Mote [12], Princeton's Zebranet [15] and MetaCriket [11], MIT's cricket [14], and Europe's Eyes [16] and BTNodes [5].

The newly proposed WSN kernels can be classified into two board categories: event-based and thread-driven. For example, the well-known TinyOS is event-based run-to-completion operating system for WSN [9-10]. Due to its run-to-completion feature, there is no context switch and thus, only one task stack is needed. Consequently, TinyOS is extremely memory efficiency. However, FIFO scheduler may cause applications to be indefinitely blocked and, at worst, may suffer from the bounded buffer problem [3]. In contrast, MANTIS OS (MOS) is a thread-driven architecture and uses the preemptive time-sliced, i.e., round robin (RR) scheduler. As a result, MOS can prevent a CPU-bound task from blocking execution for another time-sensitive task. Nevertheless, due to its RR feature, in MANTIS, each task must have its own stack. Furthermore, RR algorithm causes more context switch overhead. Actually, RR is well suited in the interactive environment to give user prompt response. In a WSN, RR is not appropriate.

Thus, in this paper, we design and implement an embedded operating system for WSNs that eliminates indefinitely blocking while achieving memory efficiency. Our kernel is named SenCore. Like MOS, SenCore also adopt the thread-driven model for its flexibility and friendly programming construct. However, instead of RR, SenCore uses preemptive three-queue FIFO scheduler. There are three priority queues. Tasks belonging to different priority queue are preemptive. However, tasks within the same priority queue are run-to-completion, i.e., by FIFO algorithm. As a result, only three task stacks, one for each priority queue, are needed in SenCore. In addition, SenCore can also prevent a low priority long-lived task from blocking another high priority task by its preemptive scheduler between different priority queues. Consequently, SenCore can achieve memory efficiency while providing prompt response to high priority tasks.

In addition to the memory constraint, WSNs are also well-known for its energy constraint since they rely on batteries as power source. Thus, when there is no ready

task in the system, SenCore will sleeps the microprocessor to save energy. Finally, SenCore hides all hardware dependent details under the Hardware Abstraction Layer (HAL). As a result, it is easy to port SenCore to other wireless sensor network platforms.

To be a complete functional kernel, SenCore also supports task management, synchronization, inter-process communication, and management of memory, timer, and peripheral devices. We have implemented SenCore on Mica2 Mote sensor node. The size of the kernel image is about 6Kbytes, which is small enough for resource-limited sensor nodes.

The rest of the paper is organized as follows. Section 2 describes the previous work on networked sensor operating system. Section 3 presents the design and implementation of our SenCore kernel. The experiment results are shown in Section 4. Finally, Section 5 gives conclusions and future work.

2. Related Work

In this section, we describe the related work on embedded operating systems for WSNs. In Section 2.1, we first present the work based on even-driven model while the work on thread-driven model is shown in Section 2.2.

2.1 Event-Driven Model

TinyOS is a famous operating system for WSNs. It uses the even-driven model. The event means a hardware event, which is an hardware interrupt caused by a timer, sensor modules, or communication devices etc. Once an event comes in, it triggers the execution of an event handler. Event handler can post a task into the task queue for deferred processing. Tasks are scheduled by FIFO. Thus, tasks are atomic with respect to each other but can be preempted by event handlers. Thus, TinyOS is memory efficient since only one task stack is needed. However, owing to the FIFO algorithm, a time-sensitive task would be blocked by a long-lived task. As a result, TinyOS can cause the classic bounded buffer producer-consumer problem [3]. For example, a number of packet processing tasks may be blocked by a long-lived CUP-bound task. As a result, the network stack's bounded buffer could quickly be overflowed and results in packet losses. This problem is more serious in sensor networks since the limited memory capacity.

SOS is also an event-driven kernel for WSNs [7]. Although SOS adopts two-level task queues, however, SOS does not allow preemptive scheduling. As a result, a higher priority task still has to wait for the execution of a lower priority task. Only after the lower priority task finishes its execution, the higher priority task then has the chance to begin execution.

Our SenCore avoids the problems of event-based run-to-completion kernels since SenCore uses the preemptive three-level priority queues. A higher priority time-sensitive task can preempt a lower priority long-lived task to avoid indefinitely blocking.

2.2 Thread-Driven Model

MANTIS OS (MOS) is a thread-driven kernel for WSNs. It uses the preemptive time-sliced multithreading scheduling algorithm. MOS supports five task priorities and allows more than one tasks in the same priority. Higher priority tasks can preempt lower priority tasks. Furthermore, tasks within the same priority are also preemptive and are scheduled by RR algorithm. As a result, MOS is a Unix-like operating system for WSNs.

Contiki is an event-based kernel but it also supports preemptive multi-threading in the form of a library [6]. Tasks require the multi-threading capability can explicitly link with the library.

However, both of MOS and Contiki multi-threading model require each task has its own stack. In addition, the RR scheduling in MOS causes excessively context switch overhead. These memory and computing overheads are much more serious in the resource limited sensor nodes. In contrast, our SenCore kernel uses three-level FIFO priority queues. Tasks in the same queue are scheduled by FIFO and are run-to-completion. As a result, no matter how many tasks in the system, it only requires three task stacks, one for each task queue. Furthermore, the context switch overhead is extremely reduced since context switch is only occurred between tasks within different task queues.

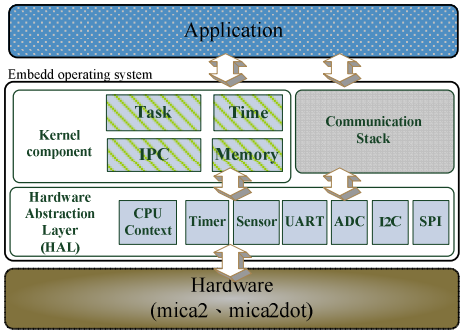

Figure 1. The System Architecture of SenCore

3. Design and Implementation

3.1 SenCore Architecture

Figure 1 shows the SenCore architecture. SenCore kernel includes the task management, timing management, IPC, and memory management. The HAL layer includes saving/restoring CPU context and the management of peripheral devices, like timer, sensor, UART etc. In the following sections, we detail the various components of our SenCore.

3.2 Task Management

An application consists of one or more tasks. Each task is managed by the Task Control Block (TCB). TCP maintains the task information including stack pointer, task priority, task state, and task ID etc. According to the characteristics and applications in a sensor network,

tasks in SenCore are given one of following three priorities.

1. High priority tasks/Deferred ISRs: these tasks have the highest priority in SenCore. They usually have short execution time. In our current design, high priority tasks are often periodic tasks that are triggered by hardware interrupts. For example, in SenCore, we have a high priority task that periodically captures the sensor module data including light, temperature, acoustic, seismic, and magnetometer. This is similar to post a task by event handlers in TinyOS and deferred ISRs in Linux [4].

2. Normal tasks: these tasks have the middle priority. For example, tasks that perform the analysis/processing of the sensed data, routing algorithm, etc., are belonging to the normal tasks.

3. Low priority tasks: these tasks have the lowest priority task. Tasks require very long computing time are given the lowest priority to prevent them from delaying the execution of higher priority tasks. For example, tasks that carry out data compression/decompression or encryption/decryption are given the lowest priority.

Note that, in addition to the following three kinds of tasks, in SenCore kernel, we also have an idle task. When there is no ready task in the kernel, SenCore invokes the idle task. However, unlike the idle task in the Linux kernel that performs iterative for-loop, the idle task in the SenCore just puts the CPU into the sleep mode to save the energy usage.

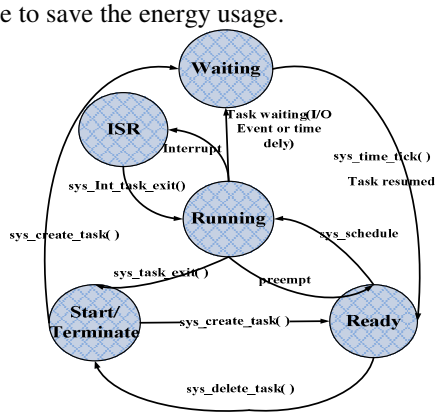

Figure 2. The task transition diagram in SenCore

Figure 2 shows the task state transition diagram. In our SenCore kernel, tasks can be transferred between the initial/termination, wait, ready, running, and ISR running states.

- Initial/Termination state: this is a pseudo state. When a task is not yet created or has been terminated, it is in this pseudo state.
- Ready state: when a task is ready to run but is not currency running (another higher priority task is currency occupying the CPU), it is in the ready state and is queued in the task ready queue.
- Waiting state: tasks waiting for some resource or suspending itself for a time interval are in the waiting state.
- ISR state: when the current running task is preempted by an interrupt to execute the IRS routine, it is in the ISR state.
- Ready state: the current running task's state.

3.3 Scheduling Algorithm

Since SenCore supports three task priorities, thus, the scheduling algorithm used by SenCore is a three-level priority queue scheme. Moreover, SenCore combines both preemptive and non-preemptive scheduling algorithms. First, Tasks in different queues (or priorities) are preemptive. This is used to provide prompt response to the time critical high priority tasks so that they will not be delayed by a lower priority tasks.

Second, we use FIFO to schedule tasks with the same priority. By FIFO's run-to-completion characteristic, we can save memory usage since only three task stacks, one for each priority queue, are required. Furthermore, we can also eliminate lots of context switch overhead compared to the RR scheme in MOS.

Scheduling Algorithm

```
for(i=TaskPrioHigh;i<=TaskPrioLow;i++){
    if(TCBPriolevel[i]->TCBStat==STAT_RDY)
        PrioHighRdy=TCBPriolevel[i]->taskid;
        break;
}
if(TCBPrioTbl[OSPrioHighRdy]->taskid != TCBPrioTbl[OSPrioCur]->taskid)
{
  if(TCBPrioTbl[OSPrioHighRdy]->tasklevel==TCBPrioTbl[OSPrioCur]->tasklevel){
    S_Task_SW();   /* task with the same priority，pass stkpointer，clear CPU Register*/
  }else {
    D_Task_SW();   /* task with the different priority， context switch*/
  }
}
```

Figure 3. The scheduling algorithm in SenCore

Figure 3 shows the SenCore scheduling algorithm. When the scheduler is called, it will first search the highest priority task in the system. Then, it checks whether the new highest priority ready task is just the previous executing task or not to avoid unnecessary context switch. After that, the scheduler checks whether the selected highest priority task has the same priority with the current task or not. If both tasks have the same priority, the scheduler calls S_TASK_SW(). Otherwise, the scheduler invokes D_TASK_SW(). Both S_TASK_SW() and D_TASK_SW() are provided in the HAL and will be introduced in Section 3.6.

Note that, a key feature of SenCore is that, no matter how many tasks in the systems, it can achieve constant-time scheduling cost. SenCore maintains a data structure called TCBPrioQueue[]. TCBPrioQueue[]has three elements with each element keeping track of whether the corresponding task queue has ready task or not. For example, if TCBPrioQueue[0] equals to 1 means that the first priority queue has ready task enqueued. Thus, to search the highest priority tasks, the scheduler first check the TCBPrioQueue[] array to find a nonzero element. After that, since tasks in the same priority queue are scheduled by FIFO, the scheduler then dequeues the first element and marks it as the highest

priority ready task. Furthermore, to facilitate to insert a ready task into the corresponding priority queue, we maintain a pointer that keeps track of the last element currently in the priority queue. Consequently, the new ready task can be easily inserted into the rear of the priority queue. As a result, the cost of the task scheduling is fixed no matter how many tasks in the system.

3.4 Timer Management

This component provides all the timing facilities in SenCore. We distinguish two main kinds of timing management:

- Maintaining the current time and date: so that user applications can query and set the current time and data through sys_time_get() and sys_time_set() APIs.
- Maintaining task timers: each task has a built-in timer, which allows application to suspend for a specified time interval. When the timer expires, the task is marked ready and put into the corresponding priority queue.

To implement the task timer, we follow the approach used in the Linux kernel [4]. All tasks timer are sorted from the largest amount to the least value. As shown in Figure 4, the amount of each task timer is maintained by the TCPCounter, which is an element in the Task's TCB data structure. However, TCPCounter is a relative value which is compared to its previous element. Thus, upon each time tick, we only have to update the first element and remove it from the queue when its value becomes zero.

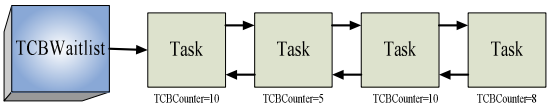

Figure 4. The waiting list in SenCore

3.5 Inter-Process Communication (IPC)

To save memory usage, we only provide counting semaphore and shared memory in SenCore. The value of each semaphore ranges from 0 to 256. If a task fails to obtain the semaphore (i.e., the value of semaphore is zero), the task is suspended on the waiting list until the semaphore is again available.

To achieve communication between tasks, SenCore provides shared memory mechanism. First, the sender application needs to dynamically allocate a memory block. The mechanism of dynamic memory allocation is mentioned in Section 3.5. After that, the sender application just sends *pointer* to the shared memory to the receiver application to avoid large data copying. Furthermore, to prevent this shared memory from race condition, both tasks must use semaphore to synchronize their accesses.

3.6 Memory Management

Figure 5 shows the memory management scheme in Atmega128L microprocessor used in Mica2. The Atmega128L consists of three kinds of memory: 128 KB flash ROM, 4KB EEPROM, and 4KB SRAM. The *.text* section is stored in Flash ROM and is executed in place (XIP). The boot loader is also stored in the Flash ROM. The *.data* and *.bss* section (initialized and uninitialized global data) are stored in SRAM. Furthermore, the *.stack* section are also in the SRAM that follows below .bss section. EERPROM has not yet been used in our current SenCore kernel.

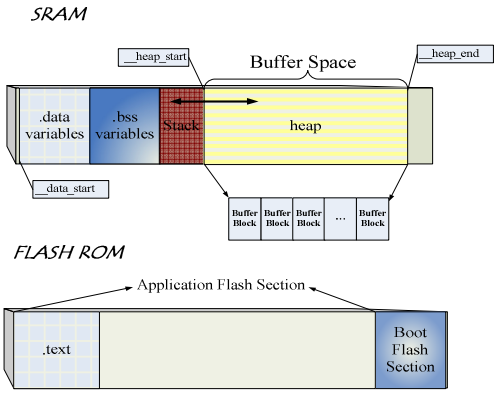

Figure 5. The Memory management in SenCore

SenCore supports dynamic memory allocation by the heat area, which is called buffer space in SenCore. We obtain the start of buffer space by reading the *__heap_start* symbol provided by the linker. We provide a partition-based memory management mechanism. This mechanism follows the same approach in μ C/OS-II [8] and Nucleus [1].

3.7 Hardware Abstraction Layer (HAL)

HAL consists of all hardware dependent details to facilitate portability. In SenCore, we partition HAL into CPU-related and peripheral-related. In CPU-related HAL, we support the following routines: *task_stk_init()*, *D_task_SW()*, *S_Task_SW()*, and *sys_int_task_SW()*. *task_stk_init()* is used to initialize task stack. *D_task_SW()* takes care of the context switch between tasks with different priorities. It pushs the CPU status and register contents into the current task's stack and then pops the previous saved CPU status and register contents from the new task's stack. In contrast, *S_TASK_SW()* is used to for tasks have the same priority. Since tasks having the same priority are scheduled by FIFO, thus, the current task must have run-to-completion. As a result, in *S_TASK_SW()*, we just pass the current task's stack pointer to the new task and re-initialize the stack content. Consequently, in SenCore, we require only one stack in each priority queue. Finally, *sys_int_task_SW()* is invoked when the ISR is finished and wants to return to the task. This routine is similar to the *D_task_SW()*, however, since the CPU and ISR has

save the CPU status and register contents when the interrupt occurs. Thus, we don't have to save it again.

In our current version of SenCore, we have been support the following peripherals: timer, UART, ADC, sensor module, and LEDs. All of these device drivers are contained in the peripheral-related HAL.

4. Performance Evaluation

We have implemented SenCore in the Mica2 sensor node. It consists of the Atmega128 microprocessor, 512 KB on-board flash, radio from Chipcon CC10000 FSK. The ATmega128 is an 8-bit RISC microprocessor from Atmel's AVR series. Furthermore, the sensor module can be connected to the Mica2 via the expansion connector.

4.1 Code Size

After compiling the kernel source code (without device drivers) for AVR microcontroller by avr-gcc, we derive the code size the *read-elf* utility. The results are shown in Table 1. The *Code size* column shows the size of the compiled object code (*.data section*), and the *Data size* column shows the size of *.data* section and *.bss* section. The total code size is about 6K bytes and the total data size is about 634 bytes. Hence, Seed kernel is very small and is very suitable for WSNs.

In Table 1, we also show the code and data size of the associated device drivers. Our currently supported device drivers have about 6.7 K bytes code size and 1.5 Kbytes data size.

4.2 Kernel Performance

Table 2 shows the execution time of the primary functions in Seed. To measure the execution of dynamic memory allocation functions, we partition a 1K bytes SRAM into 20 blocks with each block consists of 50 bytes. The results shown in Table 2 can be used as a reference to estimate task execution time when supporting soft real-time applications.

Table 3 gives the performance of interrupt overhead. We divide the interrupt handling into two parts. The first part is *interrupt response time*, which is defined as the time between the interrupt occurs and system takes to start running the ISR code. Actually, the interrupt response time can be further divided into two units: one is the hardware interrupt latency and the other is the time to save the CPU context of the current task and branch to the ISR. Note that, the interrupt latency counts the hardware delay that represents the time between an interrupt occurs and CPU jump to the first interrupt handling code (usually the interrupt vector table). The second part is the *interrupt recovery*. First, the kernel tries to find the highest priority task in the *sys_interrupt_task_exit()*. If the highest ready task is different from the current task, then SenCore invokes *sys_interrupt_task_SW()* to restore the new task's CPU context. Note that, as mentioned before, we don't need to save the context of current task since it has been saved before entering interrupt handler.

Table 1. Code Size of SenCore Kernel

Function	Code size (Byte)	Data size (Byte)
Task Management	4168	(.bss+.data) 634
Semaphore	510	
HAL	860	
Time Management	90	
Mem. Management	412	
Kernel Total without Drivers	Kernel Code Size ：6040 bytes Data size：634 bytes	
Drivers	6768	1518
Kernel Total with Drivers	Code size：12808 bytes Data size： 2152 bytes	

Table 2. Performance of Seed Primary Functions

Function	Time (Cycles)	Time (us)
sys_create _task	271	36.76
sys _delete_task	166	22.515
sys_task_exit	157	21.294
sys_scheduler	99	13.427
sys_create_semaphore	115	15.597
sys_pend_semaphore	66	8.951
sys_post_semaphore	62	8.409
sys_delete_semaphore	68	9.223
sys_create_buffer	240	32.52
sys_allocate_buffer	75	10.172
sys_free_buffer	66	8.951
D_Task_SW	176	23.871
S_Task_SW	123	16.682
sys_int_task_SW	108	14.648

Table 3. Performance of Interrupt Handling

Function	Cycles	Time(us)
(1) Interrupt Latency (hw)	11	1.491
(2) Time to save CPU context	97	13.156
Interrupt Response Time= (1)+(2)	108	14.647
Function	Cycles	Time(us)
(1) sys_interrupt_task_exit()	104	14.105
(2) sys_interrupt_task_SW()	108	14.648
Interrupt Recovery = (1)+(2)	212	28.754

Table 4. Context switch overhead

Cost (Cycles)	Time (u-sec)
Context Switch between the same priority (*sys_task_exit() + sys_scheduler() + S_Task_SW()*)	
366	49.641
Context Switch between different priorities (*sys_task_exit() + sys_scheduler() + D_Task_SW()*)	
422	57.237

Furthermore, we also measure the context switch overhead. Before giving the performance results, we first define the context switch time in our measurement.

- The context switch time between the same priority tasks: Since tasks in the same priority are run-to-completion. Thus, after performing its job, the current task must calls the *sys_task_exit()* to let the kernel to remove it from the task queue. After that, kernel invokes *sys_scheduler()* to find the highest priority task. Finally, if both tasks have the same priority, kernel calls *S_TASK_SW()* to pass the stack pointer to the new task. Thus, the context switch time between the same priority tasks = *sys_task_exit()* + *sys_scheduler()* + *S_TASK_SW()*
- The context switch time between different priority tasks: as stated in Section 3.7, similar to the above discussion but finally the kernel invokes *D_TASK_SW()* instead of *S_TASK_SW()*.

4.3 Application Performance

Finally, we measure the superior of our scheduling algorithm to application performance. The workload consists of two high priority tasks, two low priority tasks and one middle priority task. The high priority tasks are invoked periodically to capture the light and temperature from the sensor module. When the number of sample is beyond 130, two low priority tasks are invoked to encode all of the sensed data. Finally, one middle priority task is responsible to calculate the average, maximum, and minimum value of all sensed data. We measure the time delay between the invocation and the execution of middle priority tasks. From the experimental results, the time delay is about 130 *u*s to 300 *u*s. As a result, by our scheduling scheme, the middle priority task will not be indefinite blocked by the low priority tasks. Note that, the low priority tasks have their execution time around 300 *m*s. Thus, in TinyOS-like run-to-completion model, the middle priority task would be blocked at least 300 *m*s.

5. Conclusions and Future Work

In this paper, we design and implement SenCore: an embedded kernel for WSNs. It supports three-level priority task queues and combines both preemptive and nonpreemptive scheduling. Tasks in different priorities are preemptive to give high priority task shorter response time. In contrast, tasks in the same priority are nonpreemptive and scheduled by FIFO to save memory usage and to reduce context switch overhead. SenCore also be power efficient. When there is no ready task, the kernel sleeps the CPU to save energy. Finally, SenCore has a hardware abstraction layer to ease the porting effort.

We have implemented SenCore on Mica2 sensor node. The results shown in experiments demonstrate that Seed is flexible, energy efficient and suitable for WSNs. Our feature work will extend SenCore to support modules and to be fail-safe.

6. References

[1] Accelerated technology Inc., “Nucleus RTOS - Nucleus Plus”, available at http://www.acceleratedtechnology.com/embedded/plus.php, May 2004.

[2]. Berkeley mica motes, http://www.xbow.com/

[3] S. Bhatti, et. al. "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms, " ACM/Kluwer Mobile Networks & Applications, vol. 10, no. 4, August ,2005

[4] D. P. Bovet and M. Cesati, “Understanding the Linux Kernel (3nd Edition),” O'Reilly, 2005

[5] BTnodes- A Distributed Environment for Prototyping Ad Hoc Networks , http://www.btnode.ethz.ch/

[6]. Adam Dunkels, et. al., “Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors,” IEEE Workshop on Embedded Networked Sensors, November 2004.

[7] C. C. Han, et. al., "SOS: A dynamic operating system for sensor networks", the Third International Conference on Mobile Systems, Applications, And Services (Mobisys 2005), Seattle, June 2005.

[8] J. J. Labrosse, MicroC/OS II: The Real Time Kernel, CMP Books, June 2002.

[9] Philip Levis, et. al. “The Emergence of Networking Abstractions and Techniques in TinyOS,” USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI), 2004

[10] Philip Levis, et. al. “System Architecture Directions for Network Sensors,” ASPLOS, 2000

[11] F.Martin, B.Mikhak. and B.silverman, “MetaCricket: A designer’s kit for making computational devices”,IBM Systems Journal ,2000

[12] NEST Project http://webs.cs.berkeley.edu/nest-index.html

[13] Redhat Inc., “eCos RTOS Reference Manual,” available at http://ecos. sourceware.org/docs-latest/ref/ecos-ref.html, Apr. 2004.

[14] N.B Priyantha, et.”The Cricket Location-support System ”, Proc.of the sixth Annual ACM International Conference on Mobil Computing and Networking (MoBICOM),August 2000

[15] The ZebraNet Wildlife Tracker,http://www.princeton.edu/~mrm/zebranet.html

[16] The Eyes Project http://eyes.eu.org

[17] Windriver Inc, VxWorks homepage, available at http://www.windriver.com/products/device_technologies/os/vxworks5/, May 2004.