NanoQplus : A Multi-Threaded Operating System with Memory Protection Mechanism for WSNs

Sang Cheol Kim, Haeyong Kim, JunKeun Song, Misun Yu and Pyeongsoo Mah
Sensor Network Platform Research Team
Electronics and Telecommunications Research Institute (ETRI)
Gajeong-dong 161, Yuseong-gu, Daejeon, S. Korea
{sheart, haekim, jun361, msyu, pmah}@etri.re.kr

Abstract **- Sensor networks are specially constructed networks for collecting data from sensor nodes and delivering it to sink nodes at the end. Each nodes in a sensor network are small embedded systems, which require wireless communication among them with limited hardware constraints. If programmers develop sensor network applications for a large-area sensor network, the development of application will be very difficult without any support of operating system. Therefore, we developed a small operating system, referred to as NanoQplus, to enable to build a large-scale sensor network application rapidly and efficiently. NanoQplus is a multi-threaded, lightweight and low-power sensor network operating system. Although multi-threaded operating systems in sensor nodes need to manage multiple stacks to prevent the stack overflow problem, NanoQplus gives an elegant solution with a memory protection mechanism. This paper describes all about NanoQplus technologies; a philosophy, design issues, architecture, supported software products and NanoQplus programming style.**

Index Terms – Wireless Sensor Networks, Small Embedded Operating System, NanoQplus

I. INTRODUCTION

In recent years, the availability of cheap and small micro sensor nodes and low-power wireless communication enabled the large-scaled deployment of sensor nodes in Wireless Sensor Networks (WSNs). WSNs allow us to control a wide aspect of real-world problems such as monitoring health condition of people at their home, tagging small animals unobtrusively, and tracking endangered species across large remote habitats, etc. For practical use of these applications in real world, making the small-sized sensor nodes is important, and furthermore increasing the life-time of sensor nodes is crucial in WSN, through software algorithms supporting a very efficient low-power mechanism. When application developers build sensor network applications, the development will be very difficult without any operating system. Therefore, we developed a small operating system, referred to as NanoQplus, to support the flexible and convenient programming mechanism of these low-power software algorithms. In the design of NanoQplus, the following issues were considered.

- Performance : Sensor nodes have hardware constraints such as small memory, battery-powered and wireless communication. Performance is one of the important issues under the real hardware constraints. The size of a program image should be quite small (with less than 10KB) and the sensor node with two AA-battery must be operated up to a few months by efficient power management. In addition, sensor data needs to be transferred as fast as possible even in wireless communication.

- Optimization : Sensor nodes have typically small memory. Thus, the kernel size of a sensor operating system should be small, so that there must be data memory optimization in kernel level.

- Energy Efficiency : Sensor network operating systems must provide the information of the amount of energy, currently left of the sensor node. Kernel schedulers and wireless communication modules should manage the energy consumption to ensure long life-time of sensor nodes even with less-durable energy sources like batteries. For example, if there are duplicated sensor nodes in WSN, energy efficiency can be achieved by using such a characteristic of sensor networks.

- Reliable Communication : Wireless networks are highly unreliable. If a sensor node goes around, it makes this situation worse. A reliable protocol for wireless communication is essential.

- Scalability : A sensor network may consist of more than tens of thousands of sensor nodes. Thus, such a network must be constructed easily without critical performance degradation.

- Easy Programming : Most programmers desire to write sensor network applications without learning all the details of the operating system. For this reason, the hardware of embedded systems should be abstracted from user points of view, and a way of efficient debugging must be supported.

NanoQplus, developed in ETRI, is a new multi-threaded, lightweight, and low-power sensor network operating system, integrated with a general-purpose single-board hardware platform to enable flexible and rapid prototyping of WSN. The key design goals of NanoQplus are ease of use, i.e. a small learning curve that encourages novice programmers to rapidly prototype novel sensor network applications, as well as flexibility, so that expert researchers can continue to adapt and extend the hardware/software system to apply the needs of their own advanced research. This paper is organized as

	Dynamic Module Support	Execution Model	Supported Platforms	IP-USN Relevant Modules	Network Modules	Tools
TinyOS	No	Component based Event driven	Atmega128, 3 ARM7, MSP430, PXA271 etc.	6LowPAN (ArchRock)	B-mac, Flooding	TOSSIM, TOSVIS, TinyDB, TinySEC
SOS	Yes	Module based Event driven	ARM7 Atmega128 MSP430	-	UBMAC, Tree Routing	-
MANTIS OS	No	Multi-thread	Atmega128	-	X-MAC, CTP	-
Contiki	Yes	Protothread based Event driven	MSP430 AT91SAM7s	uIP	uIP, Rime	COOJA
Nano-RK	No	Reservation based Multi-thread	Atmega32 Atmega128	SLIPstream	Flooding, DSR, RT-Link, WiDom b-MAC	-
NanoQplus	No	Multi-thread	MSP430 Atmega128 CC2430, S12, PXA27x	IPv4 Gateway, 6LowPAN	IEEE 802.15.4, Zigbee, 6LowPAN, NanoMAC RENO routing	NanoESTO, NanoMON, FOTA
RETOS	Yes	Multi-thread	MSP430, Atmega128 CC2430	-	MLL, NSL, DNL architecture	RMTool

Fig. 1 Comparison table of characteristics among sensor network operating systems.

follows. Following a review of related works in Section 2, Section 3 describes all about NanoQplus technologies; a philosophy, design issues, architecture, supported software products and NanoQplus programming style. The summary and discussion are given in Section 4.

II. Related works

There are a lot of research activities world-wide for sensor network operating systems. Currently, from a macroscopic view, sensor network operating systems can be classified into two categories. One is those with the event-driven programming model which are TinyOS [1], SOS [2] and Contiki [3]. The event-driven programming model allows a single control flow of programming, and enables to launch event handlers for occurred events. The other category is those with the multi-threaded programming model. Those are MANTIS OS [4], Nano-RK [5], RETOS [6] and NanoQplus. In a sensor network research area, the event-driven programming model has been preferred because of its high-performance and ease of handling flow control of programming. Its design gives a smaller context switching latency and more efficient memory usage based on single stack management [7]. On the other hand, the multi-thread based programming model provides a large degree of preemption by managing multiple tasks based on priority and can lower the complexity of application development. Theoretically, these two programming models have “duality” with respect to each other [8]. The seven operating systems for sensor networks were summarized with comparison of main characteristics in Fig. 1.

TinyOS is a representative event-driven sensor network operating system (event handlers process incoming tasks), which operates based on states. In TinyOS, by the use of FIFO scheduler, preemptivity of tasks is very poor. Although TinyOS has no support for dynamic memory allocation, a new programming language, called NesC (Network Embedded System C), helps to write secure applications by allocating fixed-sized memory at compile-time. The NesC compiler analyzes the whole application and optimizes it using an event-driven concurrent model. Thus, it is a burden for a novice programmer to learn the new language.

SOS is an operating system for mote-class wireless sensor networks [2]. The one important feature is supporting loadable dynamic modules in kernel. This enables reuse of modules and ease of replacing modules without any modification of kernel. This feature is beneficial, particularly for a sensor network operating system, running on limited memory constraints. Similar to traditional operating systems, memory space is strictly separated into user-level and kernel-level.

Contiki is an open source multi-tasking operating system for networked memory-constrained embedded systems [3] [9]. It consists of an event-driven kernel on top of which application programs are dynamically loaded and unloaded at runtime. The processes use light-weight protothreads that provide a linear thread-like programming style on top of the event-driven kernel.

Mantis OS provides a thread-based embedded operating system for wireless sensor networks [4]. It supports C-based APIs and a Linux-style programming environment. It features preemptive multithreading, I/O synchronization by mutual exclusion, device drivers in hardware abstraction and network stacks for wireless communication. MANTIS consumes only 500 bytes of memory to operate kernel and network.

Nano-RK is a preemptive reservation-based real-time operating system for wireless sensor networks [5]. It includes a light-weight embedded resource kernel (RK) with rich functionality and timing support. It supports fixed-priority preemptive multitasking for ensuring that task deadlines are met, along with support for CPU, network, as well as, sensor and actuator reservations. Tasks can specify their resource demands and the operating system provides timely, guaranteed and controlled access to CPU cycles and network packets.

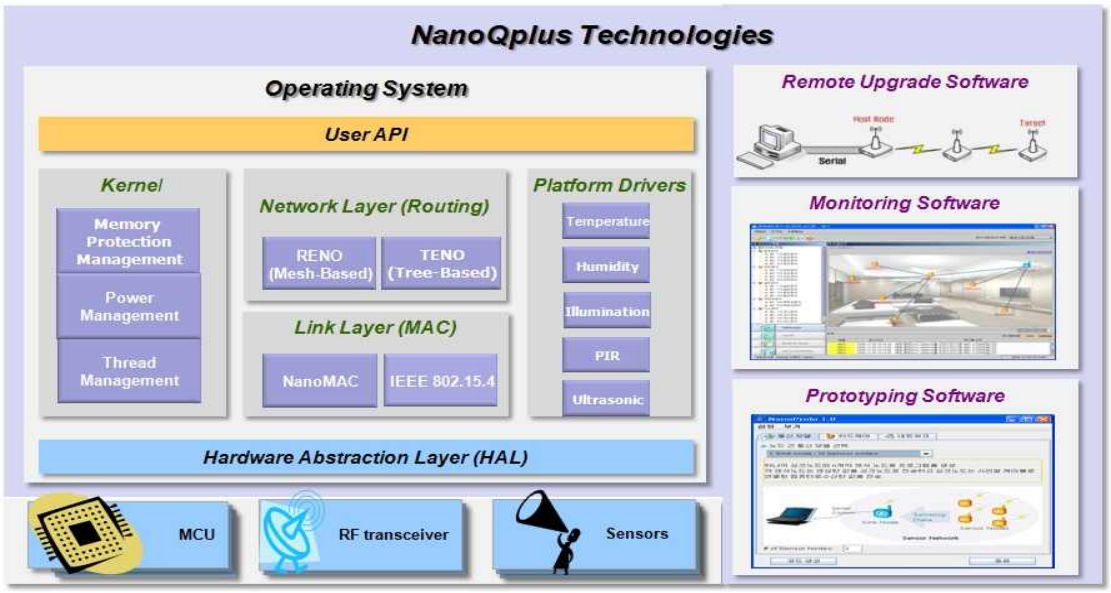

Fig. 2 Overall view of NanoQplus technologies.

RETOS is the operating system for sensor nodes from Yonsei University [6]. It provides a multithreaded programming interface, system resiliency, kernel expandability with dynamic reconfiguration, and wireless sensor network oriented network abstraction. It features the dual-mode kernel that allows kernel mode and user mode in a separate memory space. Further, RETOS supports to insert dynamic checking-codes into application. This is for detecting the intrusion of data into unallowable memory region. However, the dynamic checking-code mechanism may increase the overhead of kernel and code size.

III. NanoQplus

NanoQplus is a sensor network operating system with the thread programming model, handling a variety of issues that arise in operating systems for sensor network applications, such as hardware abstraction, task management, power management, RF message handling, routing, sensing and actuating.

Fig. 2 depicts list of NanoQplus technologies. The NanoQplus technologies adopts a modular and layered design, in which software modules are classified into hardware part, operating system part, and auxiliary software part, respectively. The hardware part is a set of real hardware components on a sensor node platform. It includes MCUs(micro-controller units) (e.g. ATmega128, MSP430, and 8051), RF transceiver, for transmitting and receiving sensor node data, which can be CC2420 or other vendor products for wireless communication, and Sensors/Actuators for collecting sensor data and actuating a device to operate. The operating system part involves kernel, network protocol stack for handling RF messages, platform driver modules for handling data from platform-specific sensors, and hardware abstraction layer (called as HAL). HAL is layered of several components such as LED, CLOCK, POWER, RFM (RF Module), UART, and ADC (Analog to Digital Converter). This part offers system APIs for convenient development of WSN applications. The auxiliary software part is a package of software supplements that ease to develop sensor network applications, or maintain sensor networks already constructed. They correspond to remote-upgrading, monitoring, and prototyping software in Fig. 2.

NanoQplus supports a rich set of sensor hardware platforms, ranging from 8-bit to 32-bit microprocessors. Sensor node platforms supported by NanoQplus are shown in Fig. 3. All NanoQplus applications can be executed on these different sensor node platforms without any modification. This is possible due to efficient hardware abstraction of NanoQplus. Because of that, it is also quite easy to port NanoQplus on another hardware platform. Thus, it is expected that more sensor node platforms will be supported in NanoQplus.

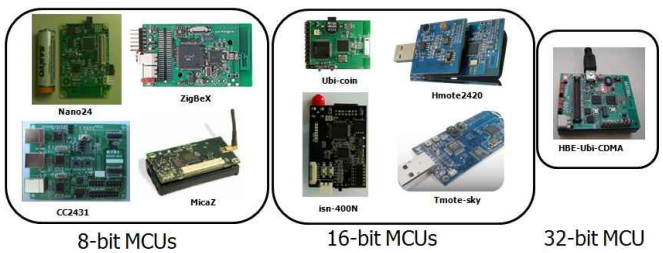

Fig. 3 Collection of hardware platforms supported by NanoQplus.

A. Kernel

The NanoQplus kernel, as shown in Fig. 4, consists of thread management, memory management and power

management. The thread management initiates a task scheduler to perform context-switching to handle thread operations. It provides a set of thread functions that can control threads. A task scheduler in NanoQplus kernel adopts a preemption round-robin scheduling method in which a thread is selected based on the priority of the thread. For the equal priorities among threads, the sequence of execution follows a round-robin fashion. In NanoQplus, two kernel objects, message queue and semaphore, are given for synchronization and communication among threads. With these kernel objects, coordinated management of threads is possible in programming. Although not shown in the figure, there is the user timer (derived from a system timer) that enables to invoke a user-defined function once or periodically. This user timer is very useful in sensor network programming because much of the works in sensor networks are sensing and sending data periodically.

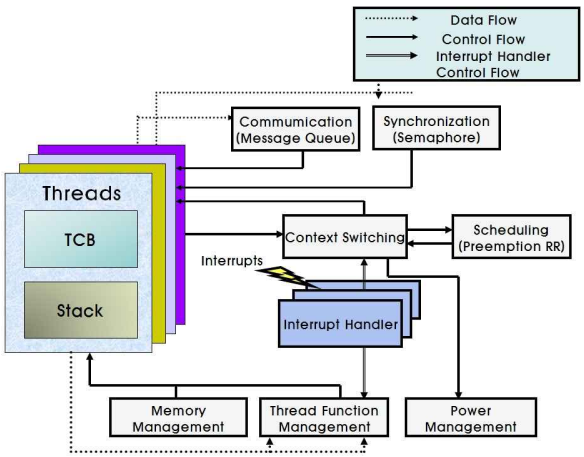

Fig. 4 Kernel architecture of NanoQplus.

The memory management handles memory-related operations such as memory allocation or de-allocation. Because of shortage of memory in a small sensor node, it is important to prevent a possible memory leakage. In NanoQplus, the memory allocating algorithm is different from platform to platform. The power management handles the power modes depending on power-consuming chips such as microprocessors or RF transceivers. It continuously attempts to minimize the power consumption of the sensor node by sleeping microprocessors whenever possible.

Fig. 5 depicts the programming style of NanoQplus. It is a typical programming style using standard C APIs. All NanoQplus applications begin with a header file “nos.h”, and the nos_init() function should be placed after the main function call. The figure shows that two threads are created, each of which is supposed to execute task1 and task2 functions, respectively. After the sched_start() function calls, the two threads will run in a round-robin fashion if the priorities of the two threads are identical.

```
#include “nos.h”

void task1()
{
      while (1) { uart_printf(“hello task1\n”); }
}

void task2()
{
      while (1) { uart_printf(“hello task2\n”); }
}

int main (void)
{
      nos_init();

      thread_create(task1, NULL, 0, …
      thread_create(task2, NULL, 0, …

      /* two threads starts */
      sched_start();

      return 0;
}
```

Fig. 5 Application programming style in NanoQplus.

Thread-driven OS has some picky issues such as the cost of context switching overhead, memory for each thread’s stack, and the possibility of stack overflow. Of those issues, the possibility of stack overflow is a representative one. For small embedded sensor nodes, microprocessors have no hardware mechanism for memory protection, which may be fatal to the reliability of system.

Some techniques for resolving the memory corruption problem in a MMU-less system were proposed such as bounding thread stack sizes [11] or dynamic stack allocation [7]. Torgerson [11] proposed a method of bounding worst-case thread stack sizes from assembly code by constructing a limited control flow graph and statically examining all possible execution flow within each thread. This approach performs a static analysis to detect ‘loops and recursion’ and ‘jumping and calls’ in the assembly code. After analysis, a complete flow graph is produced. Using the flow graph, the worst-case thread stack sizes can be statically determined at compile-time. Yi et al. [7] proposed a stack-less thread scheme for sensor operating systems. In this approach, there are no conventional thread stacks. Instead, whenever functions are invoked, this approach allocates extra memory to serve the functions in the heap memory region. The allocated memory is returned to the heap memory whenever each function completes. How much memory is needed when a function is called, is determined at compile-time by parsing assembly codes. But, this technique also uses the compiler assistance, code modification in the assembly language level. Kim et al. [12] proposed a dynamic stack allocating method that adjusts stack sizes at run-time. In this method, the prediction of stack sizes is too complicated, and thus it is expected that it is not practical in real embedded systems.

In this paper, we introduce a stack protection mechanism without any extra hardware assistance such as MMU, to prevent possible stack overflow in a thread-driven sensor network operating system. The advantage of this method is no need of any compiler assistance or code modification in

the assembly language level. It works by simply adding platform-independent codes to a scheduler part in NanoQplus kernel.

As shown in Fig. 6, the growing stack of a task may corrupt the stack data of other task. If the corruption happens, it can crash the whole system immediately after the corrupted task is about to run. This is because the stack size is fixed at compile time and cannot be altered when needed. Prediction of stack size at runtime is very difficult because a task may have a lot of control flows under different conditions and situations.

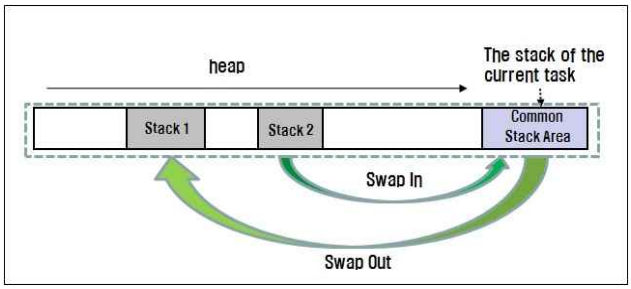

Fig. 6 Swapping mechanism for memory protection in NanoQplus.

The policy of fixed stack size cannot give an efficient memory management. Our solution to avoid this situation is to have a boundless common stack that can be shared by all tasks at runtime. When a task is about to run, the common stack is used as a stack of the task since then. When the task is preempted, the current stack data of the task is copied to somewhere in the heap region, with the exact amount of stack memory used for the task at preemption-time. Supposing that heap grows from low to high addresses, the kernel places the stack data to as low address region as possible. There is no strict boundary between heap memory and common stack memory. Thus, the stored stacks of tasks are compacted into the lower heap memory region, and in turn the common stack memory area is larger.

Immediately after copying the stack data of the preempted task, the kernel places the stack data of the new scheduled task that was stored in the heap region to the common stack region. Implementation of this mechanism needs the swap-in/out algorithm to store and restore pieces of stack data of each task between common stack area and heap memory area. Suppose that there are N bytes of total memory unused, and n tasks that requires K bytes of stack memory to be stored at preemption. Then, in this simple assumption, the stack size of the running task is ideally $N - (n\text{-}1)K$ bytes.

Although this is a good policy adopted in NanoQplus kernel, it is important to note that, even in this case, if the common stack size is too small due to lack of memory, the kernel can go into a panic. The bottom line of stack overflow problem is how to provide as large a stack as possible, given a constant memory size, so that stacks cannot be overflowed.

The proposed swapping mechanism ensures that many tasks can run in a small sensor node memory.

The disadvantage of this mechanism is the copying overhead of swapping stack memory. In our experiment, the overhead was identified as 500us for copying 500 bytes of stack. This experiment was carried out by running NanoQplus application on Octacomm's Nano-24 wireless sensor platform (whose CPU is running in 8MHz), which is similar to the Berkeley's MicaZ sensor platform. From the kernel's perspective, this cannot be simply ignored for critical sensor network applications. But the overhead can be further reduced by using a hardware device like DMA (direct memory access).

B. Network

The network part of NanoQplus is divided into link layer and routing layer. The link layer manages one-hop distance wireless communication, whereas the routing layer is responsible for transferring the data through multiple hops. If the data sent was not delivered to the destination node, the source node should deal with this situation properly. This is one of the responsibilities of the network layer.

1) MAC (Link Layer)

NanoMAC is a very small-sized MAC. This MAC implementation provides most of basic MAC functions, such as data retransmission, auto-acknowledgement and CCA for wireless communication. In NanoMAC, the MAC data is retransmitted up to 3 times, if the ACK message was not received within 12 symbol periods at the sending node. If all the three times fail, it is decided that the destined node has a problem, e.g. died. Complex MAC functions are excluded from NanoMAC to optimize the size of MAC module. NanoMAC consumes less than only 1KB of static memory. For developers who want more advanced MAC functions, NanoQplus provides IEEE standard 802.15.4 MAC.

2) Routing (Network Layer)

NanoQplus provides two kinds of routing implementations, RENO (Reactive Routing Algorithm for ETRI NanoQplus OS) and TENO (Tree-Based Routing Algorithm for ETRI NanoQplus OS). These are explained as given below in detail.

RENO is a routing protocol supporting bidirectional communication for mesh-networks. In the RENO routing protocol, each sensor node keeps a table of next-hop neighbor nodes, to find a data path to the destination node. This is performed by an on-demand manner. Specifically, a sensor node executes a process to find a data path to another node when it needs to send a data to the destined node and records the result in the table. The routine to find a data path is similar to AODV protocol, a well-known routing protocol in mobile ad-hoc networks. However, RENO is more suitable than AODV in sensor networks, because RENO has fewer number of control messages to find the data path. Current RENO implementation can cover up a few hundreds of sensor nodes,

because the limited memory capacity supplied by a small sensor node cannot maintain a large table list.

TENO is a tree-based routing protocol designed to cover up thousands of sensor nodes. The main idea is to configure multiple tree networks to reduce control packets in routing. Suppose that there is a sensor network with a few sink nodes and the application monitors sensor data continuously. In this case, the flow of communication is performed in one direction. In such a situation, it is beneficial to form a multiple tree networks to reduce control messages. Each sensor node has only to remember its parent sensor node to find a data path to a sink node (root) in tree networks. Further, it is very simple to find a data path, because exchange of data packets is carried out only with 1-hop distance neighbor nodes. Thus, TENO routing protocol can save more control packets than RENO routing protocol.

3) Example of Network Programming

An example of TENO routing is shown in Fig. 7. The scenario is data communication among two traffic lights and eight sensor nodes (e.g. attached to cars). The sensor nodes are deployed in a tree-based topology proportional to the distance from the traffic lights. Only one-hop distance communication is performed between sensor nodes. Data communication between traffic lights and sensor nodes is still maintained even if an intermediate node fails. In this example, the traffic lights can be considered as sink nodes. The rest nodes are sensor nodes. Thus, two kinds of node exist, which implies that two application types are needed. Two application types are shown in Figs. 8 and 9.

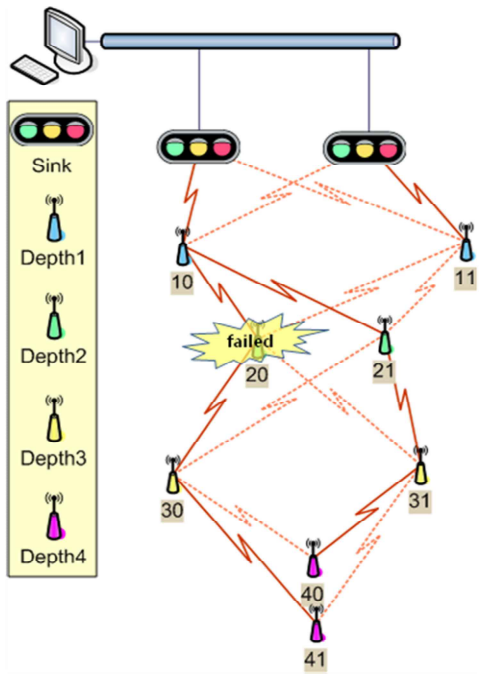

Fig. 7 Example of tree based routing protocol

```
#include "nos.h"

#define MY_ID            10        //short address: 0x0000~0xfffe
#define CHANNEL          26        //0x0B~0x1A
#define PAN_ADDR         312       //0x0000~0xfffe
#define TX_SLEEP_PERIOD  1         //sec

UINT16 tx_data[NWK_MAX_PAYLOAD_SIZE/sizeof(UINT16)];
UINT16 rx_data[NWK_MAX_PAYLOAD_SIZE/sizeof(UINT16)];

void rx_callback(void)
{
      UINT16 rx_src_id, parent_id, dest_id;
      UINT8 rx_data_length, rx_port;
      teno_recv_from_nwk(&rx_src_id, &rx_port, &rx_data_length,
rx_data, &parent_id, &dest_id);
      uart_printf("\n\t\tRX : %u(%u)", rx_src_id, rx_data[0]);
}
void task1(void* args)
{
      UINT8 i;
      tx_data[0]=0;   // dummy data
      while (TRUE)
      {
            ++tx_data[0];
            if (teno_send_to_sink(1, NWK_MAX_PAYLOAD_SIZE,
tx_data))
                  uart_printf("\nTX(%u) done to sink.", tx_data[0]);
            else
                  uart_printf("\nTX(%u) FAIL.", tx_data[0]);
            thread_sleep_sec(TX_SLEEP_PERIOD);
      }
}

int main(void)
{
      nos_init();
      teno_init(CHANNEL, PAN_ADDR, MY_ID);
      teno_set_rx_cb(rx_callback);
      uart_printf("RF Channel : %d\nPAN address: %u\nID : %d",
CHANNEL, PAN_ADDR, MY_ID);
      thread_create(task1, NULL, 0, PRIORITY_NORMAL);
      sched_start();

      return 0;
}
```

Fig. 8 Example : TENO sensor node programming

In each node, the main function calls the teno_init() function for setup the channel, PAN id and short address of the node. The teno_set_rx_cb() function registers the callback function that will be invoked as data is arrived at the node. The callback function handles the received data properly in teno_recv_from_nwk() function. However, when a sensor node receives data from their neighbour nodes as a router node, the received data does not call the callback function. Instead, the data is automatically forwarded to next neighbour node towards a sink node. The callback function is called only when the receiver node is the destination node of the data. The main function in sensor nodes creates a thread that runs the task1 function. The task1 function sends data to the sink node periodically in teno_send_to_node() function. Since the function is called in network layer, this actually performs a

series of MAC layer communication with one-hop distance nodes.

```
#include "nos.h"

#define MY_ID           1       //short address: 0x0000~0xfffe
#define CHANNEL         26      //0x0B~0x1A
#define PAN_ADDR        312     //0x0000~0xffffe

UINT16 rx_data[NWK_MAX_PAYLOAD_SIZE/sizeof(UINT16)];
UINT16 tx_data[NWK_MAX_PAYLOAD_SIZE/sizeof(UINT16)];
INT8 input_str[6];

//Receives string and toggles LEDs.
void rx_callback(void)
{
    UINT16 rx_src_id, parent_id, dest_id;
    UINT8 rx_data_length, rx_port;
    teno_recv_from_nwk(&rx_src_id, &rx_port, &rx_data_length,
rx_data, &parent_id, &dest_id);
    uart_printf("RX : %u(%u,%u)\n", rx_src_id, parent_id,
rx_data[0]);
}

void task1(void* args)
{
    UINT8 i, data_length, dest_addr;
    tx_data[0]=0;       // dummy data
    data_length = 1;

    while (TRUE)
    {
        ++tx_data[0];
        uart_printf("\nInput destination address : ");
        uart_gets(input_str, sizeof(input_str));
        dest_addr = atoi(input_str);
        teno_send_to_node(dest_addr, 2, data_length, tx_data);
    }
}

int main(void)
{
    nos_init();

    teno_init(CHANNEL, PAN_ADDR, MY_ID);
    teno_set_rx_cb(rx_callback);
    teno_role_as_sink();
    uart_printf("RF Channel : %d\nPAN address: %u\nID : %d\n",
CHANNEL, PAN_ADDR, MY_ID);

    thread_create(task1, NULL, 0, PRIORITY_NORMAL);
    sched_start();

    return 0;
}
```

Fig. 9 Example : TENO sink node programming

The sink nodes receive data from sensor nodes and print the data. In the application of sink node, task1 can send data to a sensor node designated by a user terminal.

C. Auxiliary Software

The auxiliary software is additional software product aside from the operating system. These are important portion in sensor networks because they help to construct efficient development of sensor network applications and management of sensor networks. The followings are currently supported auxiliary software.

1) Flash update over the air programming (FOTA) is remote upgrading software that upgrades flash memory in a distant sensor node via RF transmission, without any direct wired-connection of a computer. The FOTA is easy to use with GUI interface, faster than ISP upgrade via serial port, supports upgrade of multiple nodes at the same time, and further, enables to upgrade a multi-hop distance node (compatible with RENO routing protocol).

2) Rapid Prototyping Software (NanoProto) is a GUI tool that helps to generate NanoQplus application prototype-codes very fast. This tool is especially useful in developing typical sensor network programs, with least efforts and errors. Programmers should select parameters to generate application codes for their own purposes, in the graphical user interface of the rapid prototyping tool.

3) Network Monitoring (NanoMon) is flexible sensor network monitoring software with plug-in capability and dynamic GUI configuration (Fig. 10) [13]. It visualizes sensor network topology and displays sensor values by various charts (line chart, bar chart and custom charts) registered by users. It also supports the recorded history of sensor data readings and some data filtered by generation time, node IDs and value range. If sensor nodes are interfaced with NanoMON, their behaviors (e.g. transferring sensor data) are monitored in GUI.

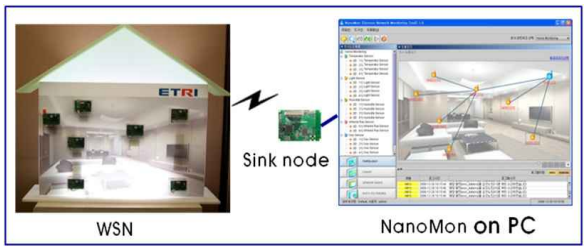

Fig. 10 Home monitoring system with NanoMon.

4) NanoEsto (Development Tool)

NanoEsto is an advanced software development tool for NanoQplus (Fig. 11) [14]. This tool can improve developers' productivity by providing a point-and-click environment for building and executing WSN applications, reducing the amount of time spent on manual development. NanoEsto runs on Linux as well as Windows with the same look and feel of the Eclipse GUI style. NanoEsto has the following features.

IDE (Integrated Development Environment) provides an eclipse-based GUI environment, which enables to create and build a project, edit C/C++ source codes and view messages. The kernel configuration tool visualizes and handles kernel modules. EEPROM management, fusing and restoring tool handles EEPROM data to download data image separately from kernel image. The JTAG debugger helps to find software

bugs by providing breakpoints, view of memory, variables, and registers.

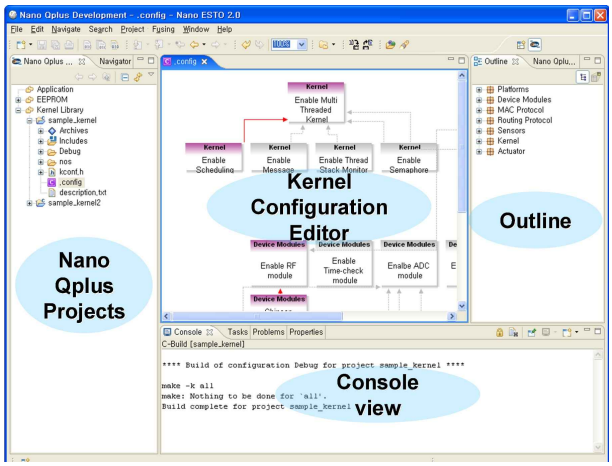

Fig. 11 Kernel configuration tool of NanoEsto.

IV. Summary and Discussion

In this paper, we introduced a new multi-threaded, lightweight, and dynamic operating system, NanoQplus[1], for WSNs. NanoQplus technologies are a rich set of software products needed for sensor network applications, including an embedded operating system, and thus an effective solution in real sensor network application. The advantages of our NanoQplus platform are intuitiveness, ease of use and development, comprehensibility, extensibility, conforming to C standard. They are very important characteristics from the developer's point of view.

In NanoQplus, the stack overflow problem, which arises with adopting a multi-thread programming approach, is elegantly solved by providing a solution of swapping thread stacks at run-time. In our experiments, the solution performed well in the latest NanoQplus kernel. In the future, we plan to provide a large-scale sensor network simulator that can produce real NanoQplus codes. The simulator is useful in a way that it helps to observe the performance characteristics of NanoQplus applications more closely. We hope that this tool is to be invaluable in designing sensor networks, and further deploying sensor nodes in large-area sensor networks.

References

[1] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler, "The emergence of networking abstractions and techniques in TinyOS," in *FirstUSENIX/ACM Symposium on Networked Systems Design and Implementation*, 2004.

[2] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *MobiSys.*, pp. 163–176, 2005.

[3] A. Dunkels, "Protothreads: Simplifying event-driven programming of memory-constrained embedded system," in *ACM Sensys 2006*, 2006.

[4] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han, "MANTIS OS : An embedded multithreaded operating system for wireless micro sensor platforms," *ACMKluwer MobilNetworks and Applications (MONET) Journal, Special Issue on Wireless Sensor Networks*, Aug. 2005.

[5] A. Eswaran, A. Rowe, and R. Rajkumar, "Nano-RK: An energy-aware resource-centric RTOS for sensor networks," in *26th IEEE International Real-Time Systems Symposium*, pp. 256–265, Dec. 2005.

[6] H. Cha, S. Choi, I. Jung, H. Shin, J. Yoo, and C. Yoon, "RETOS: Resilient, expandable, and threaded operating system for wireless sensor networks," in *The Sixth International Conference on Information Processing in Sensor Networks*, Apr. 2007.

[7] S. Yi, B. Gu, H. Min, J. Heo, Y. Kim, and Y. Cho, "Stackless thread scheme for space constrained sensor operating systems," in *Korea Computer Congress 2006*, June 2006.

[8] H. C. Lauer and R. M. Needham, "On the duality of operating system structures," in *Proceedings of 2nd International Symposium on Operating Systems*, pp. 3–19, Oct. 1978.

[9] F. Osterlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt, "Cross-level sensor network simulation with cooja," in *Proceedings 2006 31st IEEE Conference on Local Computer Networks*, pp. 641–648, Nov. 2006.

[10] G. Lin and S. J. A, "t-kernel : Provide reliable OS support for wireless sensor networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, pp. 1–14, 2006.

[11] A. Torgerson, "Automatic thread stack management for resource-constrained sensor operating systems," unpublished.

[12] S. C. Kim, H. Kim, J. Song, and P. S. Mah, "A dynamic stack allocating method in multi-threaded operating systems for wireless sensor network platforms," in *IEEE International Symposium on Consumer Electronics*, vol. 2, pp. 1–6, June 2007.

[13] M. Yu, H. Kim, and P. S. Mah, "NanoMon: An adaptable sensor network monitoring software," in *IEEE International Symposium on Consumer Electronics*, vol. 2, pp. 1–6, June 2007.

[14] I. Chun and C. Lim, "NanoEsto Debugger: The tiny embedded system debugger," in *The 8th International Conference on Advanced Communication Technology*, vol. 1, pp. 715–718, Feb. 2006.

[1] available at the site, http://www.qplus.or.kr