# Deterministic Multi-dimensional Task Scheduling Algorithms for Wearable Sensor Devices

**Jong-Jin Won[1,2], Cheol-Oh Kang[2], Moon-Hyun Kim[1] and Moon-Haeng Cho[2]**
[1] Department of Computer Engineering, Sungkyunkwan University
Suwon, 440-746 - Korea
[e-mail: wonjj@skku.edu, mhkim@ece.skku.ac.kr]
[2] The Attached Institute of Electronics and Telecommunications Research Institute
Daejeon, 305-600 - Korea
[e-mail: wonjj@ensec.re.kr, cyberkan@ensec.re.kr, root4567@ensec.re.kr]
*Corresponding author: Moon-Haeng Cho

*Received March 7, 2014; revised May 23, 2014; revised July 8, 2014; revised August 13, 2014; accepted August 27 2014; published October 31, 2014*

## Abstract

In recent years, wearable sensor devices are reshaping the way people live, work, and play. A wearable sensor device is a computer that is subsumed into the personal space of the user, and is always on, and always accessible. Therefore, among the most salient aspects of a wearable sensor device should be a small form factor, long battery lifetime, and real-time characteristics. Thereby, sophisticated applications of a wearable sensor device use real-time operating systems to guarantee real-time deadlines. The deterministic multi-dimensional task scheduling algorithms are implemented on ARC (Actual Remote Control) with relatively limited hardware resources. ARC is a wearable wristwatch-type remote controller; it can also serve as a universal remote control, for various wearable sensor devices. In the proposed algorithms, there is no limit on the maximum number of task priorities, and the memory requirement can be dramatically reduced. Furthermore, regardless of the number of tasks, the complexity of the time and space of the proposed algorithms is *O(1)*. A valuable contribution of this work is to guarantee real-time deadlines for wearable sensor devices.

***Keywords:*** Wearable sensor device, time determinism, task scheduling, real-time operating systems

**http://dx.doi.org/10.3837/tiis.2014.10.008**

## 1. Introduction

Computation and communication systems have been rapidly moving toward wearable sensor devices. A wearable sensor device is a computer that is subsumed into the personal space of the user, controlled by the user, and has both operational and interactional constancy, i.e. it is always on, and always accessible [1]-[4]. It is a wearable sensor device that is always with the user, and into which the user can always enter commands and execute a set of such entered commands, and in which the user can do so while walking around, or engaging in other activities. Therefore, among the most salient aspects of wearable sensor devices should be a small form factor and real-time characteristics.

Consequently, to solve the problems of hardware constraints and to support real-time characteristics, wearable sensor devices must use small-sized real-time operating systems (RTOSs). RTOSs are designed for wearable sensor devices with relatively limited hardware resources [5]-[10].

The RTOSs must ensure that all real-time tasks completed by their deadlines, and that no low-priority execution or communication activities are able to block higher-priority tasks for an extended period. The real-time scheduler and task management service should guarantee real-time deadlines. The task scheduler's overhead ($\Delta t$) is the time consumed by the execution of the scheduler code, whenever some task blocks and unblocks. When a running task block, the RTOS must update some data structures, to identify the task as being blocked, and then pick a new task for execution. The overheads associated with these two steps are the blocking overhead $\Delta t_b$, and the selection overhead $\Delta t_s$, respectively. Similarly, when a blocked task unblocks, the RTOS must again update some internal data structures, incurring the unblocking overhead $\Delta t_u$. The RTOS must also pick a task to execute (since the newly-unblocked task may have higher priority than the previously-executing one), so that selection overhead is incurred, as well. The typical implementation for most commercial RTOSs is to have a queue of ready tasks sorted by task priority. Tasks are blocked and unblocked, by changing one variable in the appropriate task control block (TCB). All blocked and unblocked tasks are in a single queue, sorted by priority, with the highest-priority task first. A single pointer *highestP* points to the highest-priority ready task, so $\Delta t_s$ is *O(1)*, because *highestP* is the task that should execute next. To block a task, one variable is updated in the TCB (i.e. $\Delta t_b = O(1)$), and then *highestP* is set to point to the next task in the ready queue (i.e. $\Delta t_s = O(1)$). However, to unblock a task, the scheduler parses down the queue, till it finds the appropriate place for the task in the sorted queue. Algorithms for the sorting queue are a linear search, binary search, selection sort, etc. The simplest way is the linear search. The linear search is why $\Delta t_u$ takes *O(n)* time, where *n* is the number of tasks. This means that the worst-case scheduling overhead, called time complexity, increases, as *n* increases. It has a significantly bad impact on the performance of a real-time kernel, since worst-case overheads should be taken into account, so as to guarantee real-time deadlines.

The μC/OS real-time kernel [11], [12] suggested a deterministic scheduler, using novel data structures. Its task scheduler's overhead, $\Delta t$, is constant, irrespective of the number of tasks created in an application. Since μC/OS was originally targeted for an 8-bit microcontroller, it can handle only up to 64 tasks, each with a unique priority. However, considering the fact that applications are becoming increasingly sophisticated and processing power is increasing, this restriction on the maximum number of tasks (i.e. priority) is becoming rapidly unacceptable, in many applications of wearable sensor devices.

This paper proposes deterministic multi-dimensional task scheduling algorithms, to determine the highest-priority task. The task scheduling time of the 4-bit multi-dimensional task scheduling algorithm for an ARC wearable sensor device is constant. The complexity of the time and space of the proposed algorithms is *O(1)*.

In Section 2 describes the task scheduler of operating systems for wearable sensor devices, the deterministic task scheduler of μC/OS, and a 2-dimensional task scheduling algorithm. Section 3 describes the multi-dimensional task scheduling algorithm, and a 4-bit multi-dimensional task scheduling algorithm for an ARC. The analysis of the proposed task scheduling algorithms is described in Section 4. This paper concludes with Section 5.

## 2. Related Work

### 2.1 Task scheduler of operating systems for wearable sensor devices

Wearable sensor devices are usually built with inexpensive, battery-powered devices that have limited residual energy, computation, memory and communication capacities. Despite the relatively low computing power of the microcontrollers, the controllers should be able to perform complex tasks since applications become more sophisticated. To manage this complexity, it is becoming more and more desirable to use operating systems in wearable sensor devices [13].

There are some RTOSs which can be adopted in wearable sensor devices. TinyOS [14] is a well-known operating system. The component based and event-driven execution model of TinyOS enables fine-grained power management and allows some scheduling flexibility. MANTIS [15] is a thread-based embedded operating system. It enables wearable sensor nodes to natively interleave complex tasks with time-sensitive task. The finely interleaved concurrency of multi-threading is useful in wearable sensor devices to prevent one long-lived task of blocking the execution of a second time-sensitive task. Nano-Qplus [16] is a multi-threaded, lightweight, and low-power sensor network operating system integrated with a general-purpose single-board hardware platform. It uses the notion of task to describe a piece of code that needs to be executed. Usually, there are several task activities at the same time and a task scheduler decides the run order.

Tak and Kim [17] propose a task scheduling scheme which attempts to guarantee the deadlines of time-bounded tasks and provide time-unbounded tasks with faster response time whilst minimizing energy consumption in the sensor node platform. The time-bounded tasks are periodic tasks with real-time deadline, and the time-unbounded tasks are aperiodic tasks that they have either soft deadline or no deadlines [8].

Li et al. [18] try to overcome the limit of the number of tasks and improve the efficiency of task scheduler of μC/OS-II. So they modify the scheduling algorithm by adding the comparison mechanism for the same priority level and showed that scheduling cost is reduced.

Task scheduler for wearable sensor devices should be lightweight and optimize the scheduling cost. That is, the overhead of task scheduler, the time consumed by the execution of the scheduler code, should be reduced. Without proper consideration of the priority in scheduling policy, some tasks with high priority may be delayed unfairly and even miss critical deadlines that cause severe economic loss [19], [20].

### 2.2 The deterministic task scheduler of μC/OS

The μC/OS always executes the highest-priority task that is ready to run. Each task is assigned a unique priority level, between 0 and 63. Each task that is ready to run is placed on a ready table, consisting of two variables, *OSRdyGrp* and *OSRdyTbl[8]*. The task priority is grouped (8 tasks per group) in *OSRdyGrp*. Each bit in *OSRdyGrp* is used to indicate whenever any task in a group is ready to run. When a task is ready to run, it also sets its corresponding bit in the ready table, *OSRdyTbl[8]*. To determine which priority (and thus which task) will run next, the scheduler determines the lowest priority number that has its bit set in *OSRdyTbl[8]*.

When a blocked task unblocks, the task is placed in the ready table, by the following code:

```
OSRdyGrp |= OSMapTbl[p>>3];
OSRdyTbl[p>>3] |= OSMapTbl[p&0x07];
```
(1)

where, *p* is the task's priority. This means that the unblock operation can be done with a fixed number of machine instructions. Therefore, $\Delta t_u$ is *O(1)*. As you can see in **Fig. 1**, the lower 3 bits of the task's priority are used to determine the bit position in *OSRdyTbl[8]*, whereas the next three most significant bits are used to determine the index of *OSRdyTbl[8]*. Note that *OSMapTbl[8]* (see **Fig. 2**) is a table used to equate an index from 0 to 7, to a bit mask.

When a running task block, it is removed from the ready table, by reversing the process;

```
if ((OSRdyTbl[p>>3] &= ~OSMapTbl[p&0x07]) == 0)
    OSRdyGrp &= ~OSMapTbl[p>>3];
```
(2)

This code clears the ready bit of the task in *OSRdyTbl[8]*, and clears the bit in *OSRdyGrp*, only if all the tasks in a group are not ready to run, i.e. all bits in *OSRdyTbl[p>>3]* are 0. Therefore, the blocking overhead $\Delta t_b$ is also *O(1)*.

To find the highest-priority task, it can just look up the *OSUnMapTbl[256]* (see **Fig. 2**), rather than scanning through the table *OSRdyTbl[8]*. Eight bits are used to represent when tasks are ready in a group. The least significant bit is the highest-priority. Using this byte to index the table returns the bit position of the highest-priority bit set, a number between 0 and 7.

The highest-priority task is determined by the following code:

```
y = OSUnMapTbl[OSRdyGrp];
x = OSUnMapTbl[OSRdyTbl[y]];
p = (y << 3) + x;
```
(3)

where, *p* is the highest-priority in the ready queue, and getting the pointer to the TCB of the corresponding task (i.e. *highestP*) is trivial, since each priority is mapped to a unique task. Therefore, the highest-priority task can be selected, by executing a fixed number of machine instructions, i.e. $\Delta t_s$ is *O(1)*. Since, $\Delta t_u = \Delta t_b = \Delta t_s = O(1)$, it can conclude that the task scheduling time of μC/OS is constant (i.e. the scheduler is deterministic). For the detail of the source code of the task scheduler, refer to Refs [11], [12].

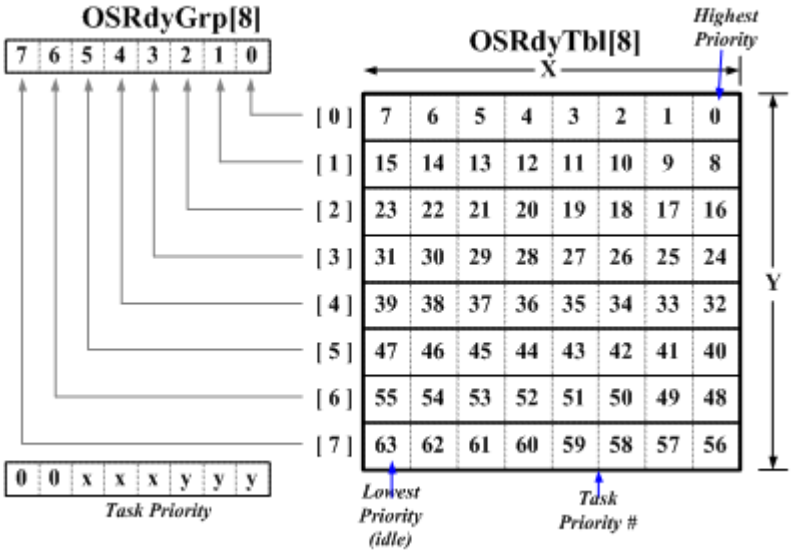


**Fig. 1.** Ready table of μC/OS

```
/* 64 level Mapping Table to Map Bit Position to Bit Mask */
char  const  OSMapTbl[8] = {0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80};

/* Priority Resolution Table */
char const OSUnMapTbl[256] = {
                    0,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    4,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    5,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    4,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    6,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    4,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    5,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    4,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    7,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    4,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    5,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    4,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    6,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    4,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    5,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0,
                    4,  0,  1,  0,  2,  0,  1,  0,  3,  0,  1,  0,  2,  0,  1,  0
};

/* 256 level Mapping Table to Map Bit Position to Bit Mask */
short  const  OSMapTbl[16] = {
                    0x0001, 0x0002, 0x0004, 0x0008, 0x0010, 0x0020, 0x0040, 0x0080,
                    0x0100, 0x0200, 0x0400, 0x0800, 0x1000, 0x2000, 0x4000, 0x8000
};
```

**Fig. 2.** Listing of map and unmap tables

However, the maximum number of tasks is restricted to 64. If it eliminates the restriction of μC/OS, the memory requirement increases rapidly.

As noted early, *OSMapTbl[8]* is used to equate an index from 0 to 7 as a bit mask. To find the highest-priority task, it can just look up the *OSUnMapTbl[256]* (see **Fig. 2**). To be extended to 256 levels of priorities (see **Fig. 4**), task priorities are grouped (16 tasks per group) in *OSRdyGrp*, which is a 16 bit variable. *OSRdyTbl[16]* is 16-bit-width. For a bit mask, it uses *OSMapTbl[16]*, to equate an index from 0 to 15. Since *OSMapTbl[16]* is 16-bit-width (2bytes), the amount of memory required increases from 8Bytes to 32Bytes. And the amount of memory required *OSUnMapTbl[]* is $2^{16}$(65536) bytes by Equation (3), because *OSRdyGrp* is a 16 bit variable.

Therefore, as shown in **Table 1** and **Fig. 3**, the space complexity could be exponential.

**Table 1.** Memory requirements

| Priorities | Unmap Table | Map Table |
|---|---|---|
| 64 | 256($2^8$) | 8 |
| 256 | 65536($2^{16}$) | 32 |
| 1024 | $2^{32}$ | 128 |
| 4096 | $2^{128}$ | 512 |

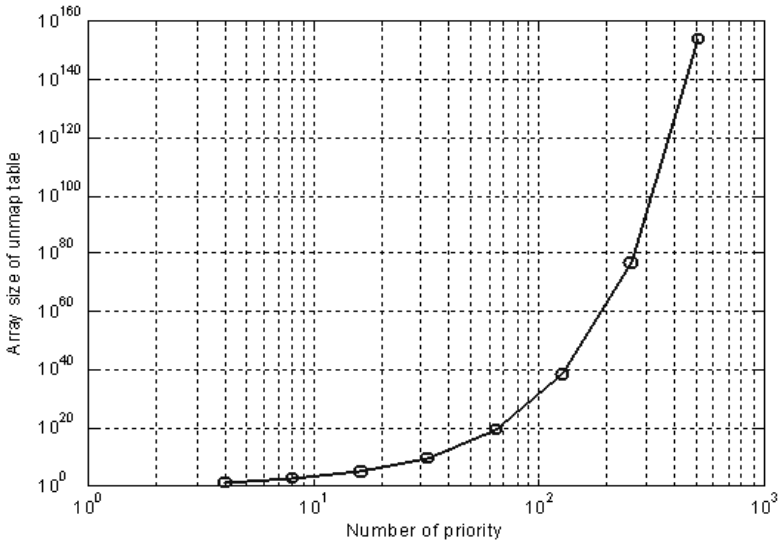


**Fig. 3.** The space complexity

## 2.3 The 2-dimensional task scheduling algorithm

The 2-dimensional task scheduling algorithm (2-D) [21] determines the highest-priority task, using the same data structure *OSUnMapTbl[256]*. There is no limit to the maximum number of task priorities.

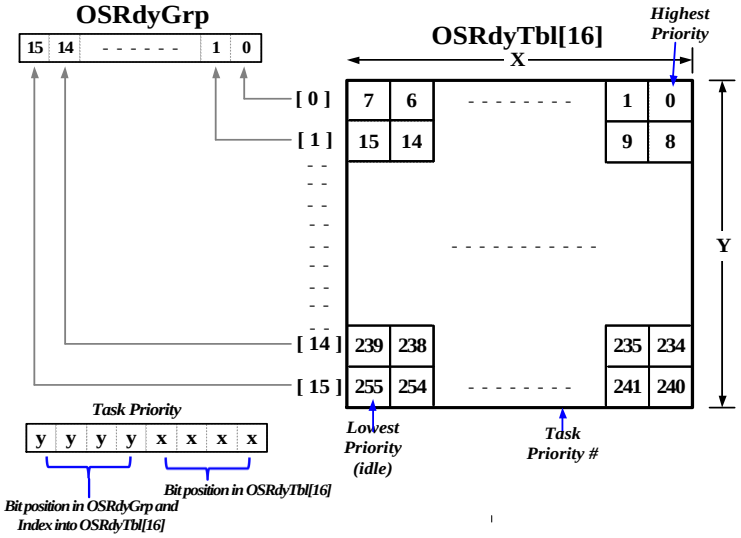


**Fig. 4.** Ready table of 2-dimensional methodology

To be extended to 256 levels of priorities, task priorities are grouped (16 tasks per group) in *OSRdyGrp*, which is a 16 bit variable. *OSRdyTbl[16]* is 16-bit-width. For a bit mask, it uses *OSMapTbl[16]*, to equate an index from 0 to 15. Since *OSMapTbl[16]* is 16-bit-width (2bytes), the amount of memory required increases from 8Bytes to 32Bytes. The *OSRdyGrp* and *OSRdyTbl[16]* are shown in **Fig. 4**.

When a blocked task unblocks, the task is placed in the ready table, by the following code:

```
OSRdyGrp  |=  OSMapTbl[p >> 4];
OSRdyTbl[p >> 4]   |=  OSMapTbl[p & 0x0F];
```
(4)

A task is removed from the ready table, by reversing the process, as follows:

```
if ((OSRdyTbl[p>>4]  &= ~OSMapTbl[p & 0x0F]) == 0)
        OSRdyGrp  &=  ~OSMapTbl[p >> 4];
```
(5)

Determining the highest-priority in the ready table is accomplished with the following section of code:

```
if ((OSRdyGrp & 0x00FF)  ==  0)
        y  =  OSUnMapTbl[(OSRdyGrp & 0xFF00) >> 8] + 8;
Else
        y  =  OSUnMapTbl[OSRdyGrp & 0xFF00];
if ((OSRdyTbl[y] & 0x00FF)  ==  0)
        x  =  OSUnMapTbl[(OSRdyTbl[y] & 0xFF00) >> 8] + 8;
Else
        x  =  OSUnMapTbl[OSRdyTbl[y] & 0xFF00];
p  =  (y << 4) + x;
```
(6)

Therefore, the highest-priority task can be selected, by executing a fixed number of machine instructions, i.e. $\Delta t_s$ is $O(1)$. Since, $\Delta t_u = \Delta t_b = \Delta t = O(1)$, it can be concluded that the task scheduler of μC/OS is extended to 256 levels of priorities.

## 3. The multi-dimensional task scheduling algorithms

In μC/OS, to be extended from 64 levels of priorities to 256 levels of priorities, the amount of memory due to *OSUnMapTbl[]* rapidly increases from $2^8$(256) bytes to $2^{16}$(65536) bytes. However, the 2-dimensional task scheduling algorithm is not limited to a maximum number of task priorities, and memory requirement can be dramatically reduced. Nevertheless, the proposed algorithm has still a memory overhead and a performance overhead. Memory requirement due to *OSMapTbl[]* increases, and it occurs performance degradation.

To be extended to 1024 levels of priorities, it uses *OSMapTbl[32]* to equate an index from 0 to 31. Since *OSMapTbl[32]* is 32-bit-wide (4Bytes), the amount of memory due to *OSMapTbl[]* increases from 32Bytes to 128(32*4)Bytes.

### 3.1 The multi-dimensional task scheduling algorithm

This section describes how the multi-dimensional task scheduling algorithm (MD) is extended to 512 and 4096 levels of priorities.

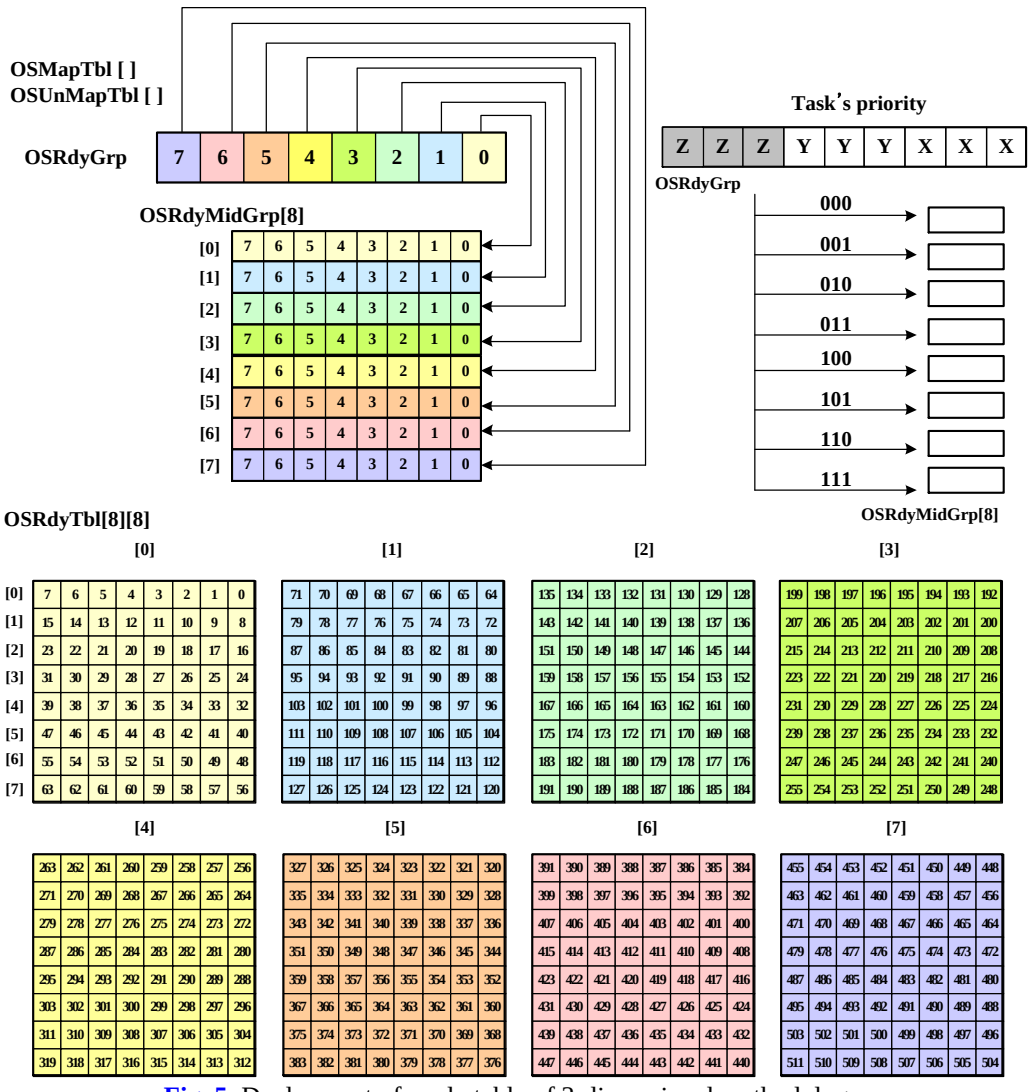


**Fig. 5.** Deployment of ready table of 3-dimensional methodology

To be extended to 512 levels of priorities, each task that is ready to run is placed on a ready table, consisting of three variables in **Fig. 5**, *OSRdyGrp*, *OSRdyMidGrp[8]* and *OSRdyTbl[8][8]*. The task priority is grouped (8 tasks per middle group) in *OSRdyMidGrp*, and the *OSMidRdyGrp* is grouped (8 *OSRdyMidGrp* per group) in *OSRdyGrp*. Each bit in *OSRdyGrp* is used to indicate whenever any task in *OSRdyMidGrp* is ready to run. When a task is ready to run, it also sets its corresponding bit in the ready table, *OSRdyTbl[8][8]*.

When a blocked task unblocks, the task is placed in the ready table by the following code:

```
OSRdyGrp |= OSMapTbl[p >> 6];
OSRdyMidGrp[p>>6] |= OSMapTbl[(p&0x38)>>3];                    (7)
OSRdyTbl[p >> 6][ (p&0x38) >> 3] |= OSMapTbl[p&0x07];
```

When a task is removed from the ready table, it is removed from the ready table, by reversing the process.

```
if((OSRdyTbl[p >> 6][ (p&0x38) >> 3] &= ~OSMapTbl[p&0x07]) == 0)
{
    if((OSRdyMidGrp[p >> 6] &= ~OSMapTbl[(p&0x38) >> 3]) == 0)
    {                                                          (8)
        OSRdyGrp &= ~OSMapTbl[p >> 6];
    }
}
```

The highest-priority task is determined by the following code:

```
z = OSUnMapTbl[OSRdyGrp];
y = OSUnMapTable[OSRdyMidGrp[z]];                              (9)
x = OSUnMapTable[OSRdyTbl[z][y]];
p = (z << 6) + (y << 3) + x;
```

Also, it is easily extended to 4096($2^{12}$) levels of priorities. Each task that is ready to run is placed in a ready table consisting of four variables, *OSRdyGrp*, *OSRdyMidGrp1[8]*, *OSRdyMidGrp2[8][8]* and *OSRdyTbl[8][8][8]*.

When a blocked task unblocks, the task is placed in the ready table, by the following code.

```
OSRdyGrp |= OSMapTbl[p>>9];
OSRdyMidGrp1[p>>9] |= OSMapTbl[(p&0x1C0)>>6];
OSRdyMidGrp2[p>>9] [(p&0x1C0)>>6] |= OSMapTbl[(p&0x38)>>3];    (10)
OSRdyTbl[p>>9] [(p&0x1C0)>>6] [(p&0x38)>>3] |= OSMapTbl[p&0x07];
```

When a task is removed from the ready table, it is removed from the ready table, by reversing the process.

```
if((OSRdyTbl[p >> 9] [(p&0x1C0)>>6] [ (p&0x38) >> 3]
                                    &= ~OSMapTbl[p&0x07]) == 0)
{
    if((OSRdyMidGrp2[p >> 9][(p&0x1C0) >> 6]
                                    &= ~OSMapTbl[(p&0x38)>>3]) == 0)
   {
       if( (OSRdyMidGrp1[p >> 9] &= ~OSMapTable[(p&0x1C0) >> 6]) ==0)
       {
            OSRdyGrp &= ~OSMapTbl[p >> 9];
       }
   }
}
```
(11)

The highest-priority task is determined by the following code:

```
z = OSUnMapTbl[OSRdyGrp];
y = OSUnMapTbl[OSRdyMidGrp1[z]];
x = OSUnMapTable[OSRdyMidGrp2[z][y]];
w = OSUnMapTable[OSRdyTbl[z][y][x]];
p = (z << 9) + (y << 6) + (x << 3) + w;
```
(12)

Therefore, the highest-priority task can be selected by executing a fixed number of machine instructions, i.e. $\Delta t_s$ is $O(1)$. Since, $\Delta t_u = \Delta t_b = \Delta t_s = O(1)$, the task scheduling time of the multi-dimensional task scheduling algorithm is constant, irrespective of the number of tasks created in an application of wearable sensor devices (i.e. the scheduler is deterministic).

### 3.2 The 4bit multi-dimensional task scheduling algorithm for ARC wearable sensor device

The 4-bit multi-dimensional task scheduling algorithm (4bit-MD) is implemented for an ARC wearable sensor device. It uses *OSUnMapTbl[16]* and *OSMapTbl[4]* (see **Fig. 6**), and the levels of priorities can be easily extended.

```
/* Mapping Table to Map Bit Position to Bit Mask */
char  const  OSMapTbl[4] = { 0x01, 0x02, 0x04, 0x08 };

/* Priority Resolution Table */
char const OSUnMapTbl[16] = { 0, 0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0 };
```

**Fig. 6.** Map and unmap tables in ARC

To be extended to 256 levels of priorities, each task that is ready to run is placed on a ready table, consisting of the four variables in **Fig. 7**, *OSRdyGrp*, *OSRdyMidGrp1[4]*, *OSRdyMidGrp2[4][4]* and *OSRdyTbl[4][4][4]*.

When a blocked task unblocks, the task is placed in the ready table, by the following code:

```
OSRdyGrp  |=  OSMapTbl[p >> 6];
OSRdyMidGrp1[p>>6]  |= OSMapTbl[(p&0x30)>>4];
OSRdyMidGrp2[p>>6] [(p&0x30)>>4]  |= OSMapTbl[(p&0x0C)>>2];
OSRdyTbl[p>>6] [(p&0x30)>>4] [(p&0x0C)>>2]  |= OSMapTbl[p&0x03];
```
(13)

When a task is removed from the ready table, it is removed from the ready table, by reversing the process.

```
if((OSRdyTbl[p>>6] [(p&0x30)>>4] [ (p&0xC)>>2]
                                &= ~OSMapTbl[p&0x03]) == 0)
{
    if((OSRdyMidGrp2[p>>6][ (p&0x30)>>4]
                                &= ~OSMapTbl[(p&0x0C)>>2]) == 0)
    {                                                                    (14)
        if( (OSRdyMidGrp1[p>>6] &= ~OSMapTbl[(p&0x30)>>4]) ==0)
        {
            OSRdyGrp &= ~OSMapTbl[p >> 6];
        }
    }
}
```

The highest-priority task is determined by the following code:

```
z = OSUnMapTbl[OSRdyGrp];
y = OSUnMapTbl[OSRdyMidGrp1[z]];
x = OSUnMapTable[OSRdyMidGrp2[z][y]];                                    (15)
w = OSUnMapTable[OSRdyTbl[z][y][x]];
p = (z << 6) + (y << 4) + (x << 2) + w;
```

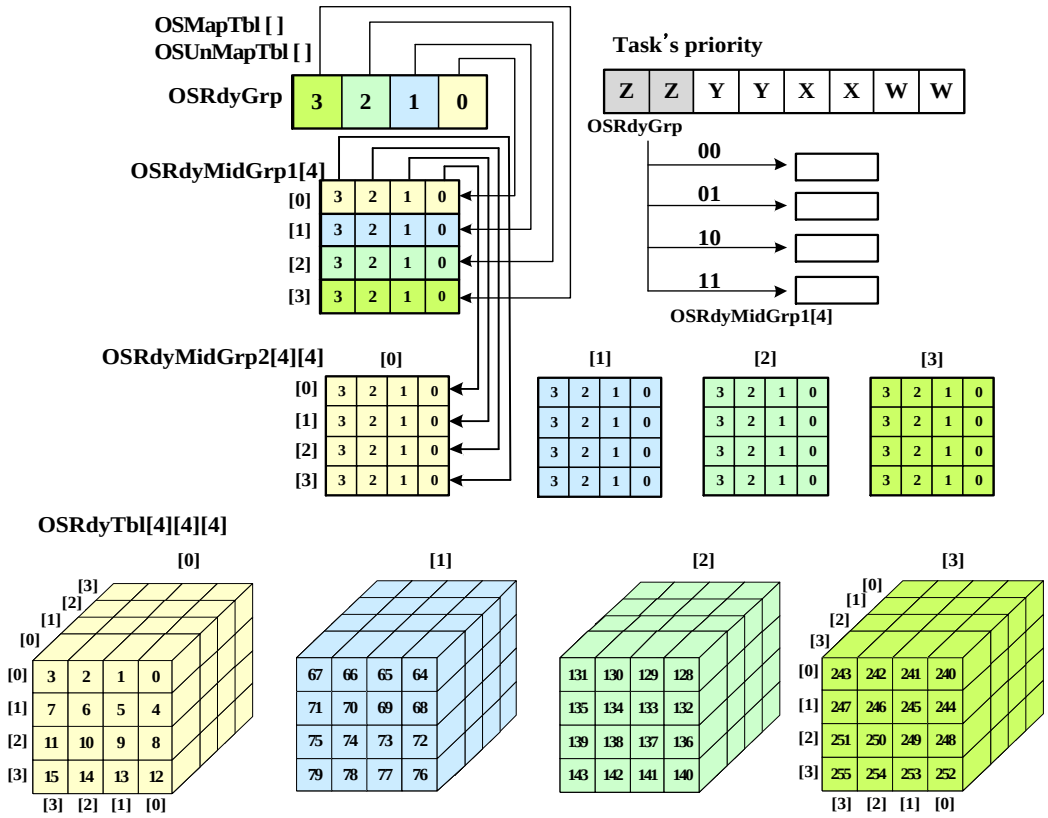


**Fig. 7.** Deployment of ready table of 4-dimensional methodology for ARC

The complexity of the time and space of the proposed algorithms is *O(1)*. In this way, the proposed algorithms can extend the task scheduling algorithm to any number of priority levels, by appropriately modifying the code for placing a task in the ready table, and the code for selecting the highest-priority task in the ready table.

Consequently, the proposed algorithms could easily be implemented on wearable sensor devices, with relatively limited hardware resources.

## 4. Evaluation and measurement

To evaluate the performance, the proposed algorithms had been implemented in a low-power real-time operating system [22], for an ARC wearable sensor device. The ARC is a wearable wristwatch-type remote controller; it also can serve as a universal remote control for various consumer electronic devices. The ARC can be worn on the wrist, and is based on a 3-axis accelerometer sensor, in order to recognize forearm gestures. However, the ARC platform is composed of small-sized memory (1MByte SDRAM), limited battery lifetime, and embedded microprocessors with low-processing power.

**Table 2.** Complexity of the space to extend the levels of priorities

| Priorities | μC/OS | 2-D | MD | 4bit-MD |
|---|---|---|---|---|
| 64 | $256(2^8)+8$ | $256(2^8)+8$ | $256(2^8)+8$ | $16(2^4)+4$ |
| 256 | $65536(2^{16})+32$ | $256(2^8)+32$ | $256(2^8)+8$ | $16(2^4)+4$ |
| 1024 | $2^{32}+128$ | $256(2^8)+128$ | $256(2^8)+8$ | $16(2^4)+4$ |
| 4096 | $2^{128}+512$ | $256(2^8)+512$ | $256(2^8)+8$ | $16(2^4)+4$ |

**Table 2** shows the memory requirements of the deterministic task scheduler of μC/OS, and the proposed multi-dimensional task scheduling algorithms. The sum of *OSMapTbl[]* and *OSUnMapTbl[]* is the space complexity. The complexity of the space of μC/OS could be exponential. Therefore, the deterministic task scheduling algorithm of μC/OS can't be implemented on wearable sensor devices with small-sized memory resources, but the proposed algorithms can be implemented. In the 2-dimensional task scheduling algorithm, the memory requirement of *OSUnMapTbl[]* does not increase, but the memory requirement of *OSMapTbl[]* does increase. However, regardless of the number of tasks, the complexity of space of the proposed multi-dimensional task scheduling algorithm and 4-bit multi-dimensional task scheduling algorithm is *O(1)*.

When a running task block, the task scheduler must update some data structures, to identify the task as being blocked, and then pick a new task for execution. The overheads associated with these two steps are the blocking overhead, and the selection overhead. Similarly, when a blocked task unblocks, the task scheduler must again update some internal data structures, incurring the unblocking overhead.

The performance overhead to extend the levels of priorities is shown in **Fig. 8**. The performance overhead of blocking (see **Fig. 8 (a)**) and unblocking (see **Fig. 8 (b)**) operations are slightly increased, according to the extended levels of priorities. As you can see **Fig. 8(c)**, when it is extended to 256 levels of priorities, the selection overhead ($\Delta t_s$) of the 2-D is 3.730μs and $\Delta t_s$ of the 8-bit MD is 3.814μs, $\Delta t_s$ of the 4-bit MD is 3.832μs. Also, when it is

extended to 4096 levels of priorities, $\Delta t_s$ of the 2-D is 4.779μs and $\Delta t_s$ of the 8-bit MD is 3.961μs, $\Delta t_s$ of the 4-bit MD is 4.098μs. While selection overhead of the 2-dimensional task scheduling algorithm is rapidly increasing, the multi-dimensional algorithm and 4-bit multi-dimensional task scheduling algorithm are slightly increased. Consequently, the multi-dimensional task scheduling algorithms are better than the 2-dimensional task scheduling algorithm.

The task scheduler's overhead (see **Fig. 8 (d)**) is the sum of the context switch time, the time to insert a task in the ready queue, and the time to select the highest-priority task. If there is no limit to the amount of memory, the deterministic task scheduler of μC/OS will work best. However, the deterministic task scheduler of μC/OS can't be implemented on wearable sensor devices with small-sized memory resources.

In summary, it can be concluded that the multi-dimensional task scheduling algorithms do not require additional memory resources to extend levels of priorities, and are deterministic, irrespective of the number of tasks created in an application of wearable sensor devices.

The complexity of the time and space of the proposed algorithms is *O(1)*. Also, the proposed algorithms guarantee real-time deadlines for the applications of wearable sensor devices. Further, the 4-bit multi-dimensional task scheduling algorithm for the ARC wearable sensor device is developed. Although the 4-bit multi-dimensional task scheduling algorithm has slightly more performance overhead than the multi-dimensional deterministic task scheduling algorithm, it can easily be implemented on wearable sensor devices with small-sized memory resources.

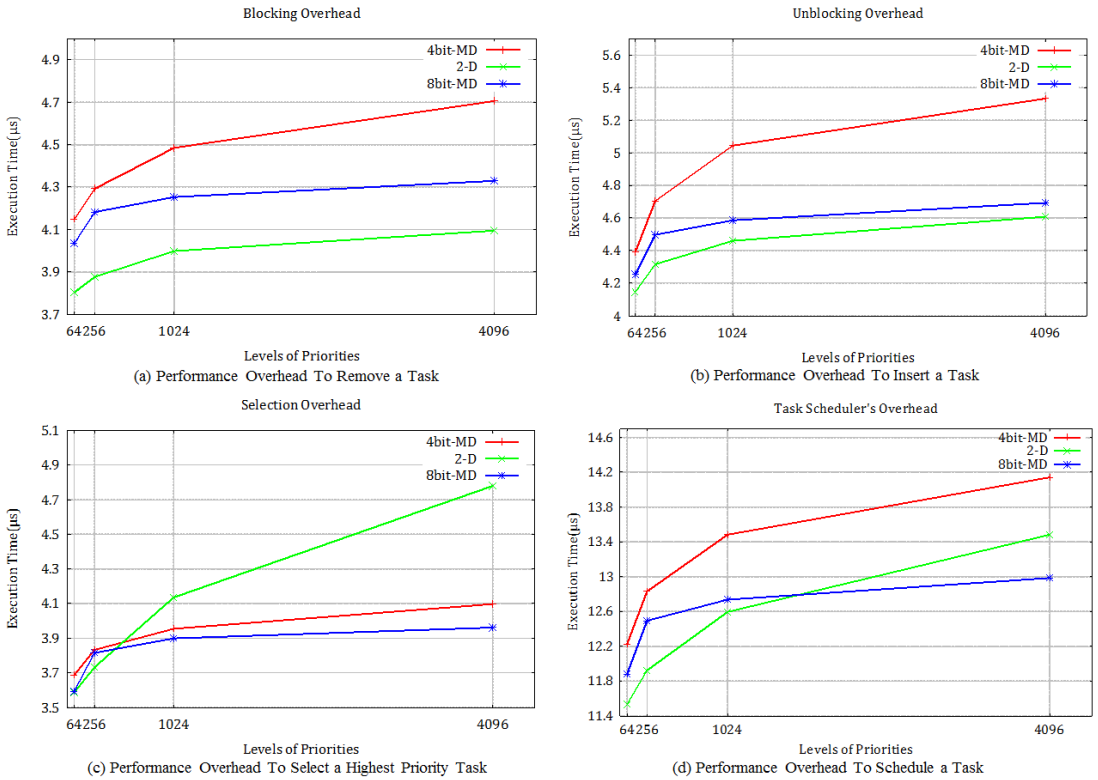


**Fig. 8.** Performance overhead to be extended levels of priorities

## 5. Conclusion

While wearable sensor devices have hardware constraints such as small-sized memory (RAM and ROM), limited battery life time and low processing power, the applications of wearable sensor-based systems become more complex and sophisticated. Therefore, kernel services that make up the applications of wearable sensor-based systems should be deterministic by specifying how long each service call will take to execute.

In this paper, deterministic task scheduling algorithms for a wearable sensor device with relatively limited hardware resources were successfully proposed, and developed to guarantee real-time deadlines for the applications of wearable sensor device. The method to extend the levels of priorities is explained in detail, and the performance overhead of the proposed algorithms is measured and discussed. The 4-bit multi-dimensional deterministic task scheduling algorithm is especially suitable for wearable sensor devices, such as the ARC considered here. In future work, the proposed algorithms need to be improved, for another task set typically found on wearable sensor devices will be done.

## References

[1] Edward S. Sazonov, George Fulk, James Hill, Yves Schutz and Raymond Browning, "Monitoring of posture allocations and activities by a shoe-based wearable sensor," *IEEE Trans. Biomedical engineering,* vol. 58, no. 4, pp. 983-990, April, 2011. Article (CrossRef Link).

[2] Alexandros Pantelopoulos and Nikolaos G. Bourbakis, "A survey on wearable sensor-based systems for health monitoring and prognosis," *IEEE Trans. Systems, Man, and Cybernetics*, vol. 40, no. 1, pp. 1-12, January, 2010. Article (CrossRef Link).

[3] Uwe Maurer, Anthony Rowe, Asim Smailagic and Daniel P. Siewiorek, "eWatch: A werable sensor and notification platform," in *Proc. of International Workshop on Wearable and Implantable Body Sensor Networks,* pp. 145-148, April 3-5, 2006. Article (CrossRef Link).

[4] G. Kortuem and T. Trarner, "The challenges of wearable computing: Part 2," *Micro, IEEE,* vol. 21, no. 4, pp. 54-67, Jul. 2001. Article (CrossRef Link).

[5] K. M. Zuberi and K. G. Shin, "EMERALDS: A small-memory real-time microkernel," *IEEE Trans. Software Engineering,* vol. 27, no. 10, pp. 909-928, Oct. 2001. Article (CrossRef Link).

[6] S. Chodrow, F. Jahanian and M. Donner, "Run-time monitoring of real-time systems," in *Proc. of Real-Time Systems Symposium,* pp. 74-83, Dec. 1991. Article (CrossRef Link).

[7] J. Y. Chung, J. W. Liu and K. J. Lin, "Scheduling periodic jobs that allow imprecise results," *IEEE Trans. Computer*, vol. 39, no. 9, pp. 1156-1174, Sep. 1990. Article (CrossRef Link).

[8] K. G. Shin and P. Ramanathan, "Real-time computing: a new discipline of computer science and engineering," in *Proc. of the IEEE,* vol. 82, no. 1, pp. 6-24, Jan. 1994. Article (CrossRef Link).

[9] D. Haban and K. G. Shin, "Application of real-time monitoring for scheduling tasks with random execution times," *IEEE Trans. Software Engineering,* vol. 16, no. 12, pp. 1374-1389, Dec. 1990. Article (CrossRef Link).

[10] H. Kopetz and R. Zainlinger, "The design of real-time systems: from specification to implementation and verification," *Software Engineering Journal,* vol. 6, no. 3, pp. 72-82, May, 1991. Article (CrossRef Link).

[11] J. J. Labrosse, *µC/OS: The Real-Time Kernel*, R&D Publications, Lawrence, 1993.

[12] J. J. Labrosse, *µC/OS II: The Real-Time Kernel 2nd Edition*, R&D Publications, Lawrence, 2002.

[13] Patel S., Park H., Bonato P., Chan L. and Rodgers M., "A review of wearable sensors and systems with application in rehabilitation," *Journal of neuroengineering and rehabilitation*, 9(1), 21., April, 2012. Article (CrossRef Link).

[14] Levis, P., et al., "The emergence of networking abstractions and techniques in tinyos," in *USENIX/ACM Symposium on Networked Systems Design and Implementation*, vol. 4, pp.1-14, 2004. Article (CrossRef Link).

[15] H. Abrach, et al., "MANTIS: System support for multimodal networks of In-situ Sensors," in *Proc. of the 2nd ACM international conference on Wireless sensor networks and applications*. pp. 50-59, 2003. Article (CrossRef Link).
[16] Seungmin Park, et al., "Embedded sensor networked operating system," in *Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing*, pp. 24-26, April, 2006. Article (CrossRef Link).
[17] S. Tak, H. Kim, and T. Kim, "A study on real-time scheduling for low-power sensor node platforms," in *IEEE 12th International Conference on Computer and Information Technology*, pp. 169-176, October 27-29, 2012. Article (CrossRef Link).
[18] Li Y. H., et al., "An improvement of task scheduling algorithms and hardware scheduler of real-time operating system," in *International Journal of Hybrid Information Technology*, vol. 7, no. 3, pp. 337-344, 2014. Article (CrossRef Link).
[19] Buttazzo G. C., Bertogna M. and Yao G., "Limited preemptive scheduling for real-time systems. a survey," *IEEE Trans. Industrial Informatics*, vol. 9, no. 1, pp. 3-15. Feb. 2013. Article (CrossRef Link).
[20] Chen S., Zhang Y., Hu Z. and Yu H., "An application-level priority scheduling for many-task computing in multi-user heterogeneous environment," in *International Conference on High Performance Computing and Simulation*, pp. 558-565, July 1-5, 2013. Article (CrossRef Link).
[21] S. J. Oh, et al., "Deterministic task scheduling for embedded real-time operating systems," *IEICE Trans. Inf. & Syst.,* vol. E87-D, no. 2, pp. 472-474, Feb. 2004. Article (CrossRef Link).
[22] M. H. Cho and C. H. Lee, "A low-power real-time operating system for ARC (Actual Remote Control) wearable device," *IEEE Trans. Consumer Electron*ics, vol. 56, no. 3, pp. 1602-1609, August, 2010. Article (CrossRef Link).

**Jong-Jin Won** received the MS degree in the Dept. of Computer Engineering from SungKyunKwan University, Suwon, S. Korea in 2000. Since 2000, he is working at Attached Institute of Electronics & Telecommunications Research Institute (ETRI). His research interests include wearable computing and embedded system.

**Cheol-Oh Kang** received the MS degree in computer engineering from Inha University, Incheon, S. Korea in 1995. And Ph.D. degree in computer engineering from Chungnam National University, Daejeon, S. Korea in 2014. Since 1995, he is working at Attached Institute of Electronics & Telecommunications Research Institute (ETRI). His research interests include wearable computing, network and cloud security.

**Moon-Hyun Kim** received the BS degree in the Dept. of Electronics Engineering from Seoul National University, S. Korea, in 1978, MS and Ph.D. degrees in Dept. of Electronics Engineering, University of Southern California, USA, in 1985 and 1988. He has served as a professor in the Dept. of Computer Engineering in SungKyunKwan University since 1988. His main interest area is intelligent information system.

**Moon-Haeng Cho** received his MS and Ph.D. degrees in computer engineering from Chungnam National University, Daejeon, S. Korea in 2006 and 2010. Since 2011, he is working at Attached Institute of Electronics & Telecommunications Research Institute (ETRI). His research interests include wearable computing, real-time and ubiquitous computing, light weight and low-power real-time operating systems.