UbiFOS: A Small Real-Time Operating System for Embedded Systems

Hee-Joong Ahn, Moon-Haeng Cho, Myoung-Jo Jung, Yong-Hee Kim, Joo-Man Kim, and Cheol-Hoon Lee

The ubiquitous flexible operating system (UbiFOS) is a real-time operating system designed for cost-conscious, low-power, small to medium-sized embedded systems such as cellular phones, MP3 players, and wearable computers. It offers efficient real-time operating system services like multi-task scheduling, memory management, inter-task communication and synchronization, and timers while keeping the kernel size to just a few to tens of kilobytes. For flexibility, UbiFOS uses various task scheduling policies such as cyclic time-slice (round-robin), priority-based preemption with round-robin, priority-based preemptive, and bitmap. When there are less than 64 tasks, bitmap scheduling is the best policy. The scheduling overhead is under 9 μs on the ARM926EJ processor. UbiFOS also provides the flexibility for user to select from several inter-task communication techniques according to their applications. We ported UbiFOS on the ARM9-based DVD player (20 kB), the Calm16-based MP3 player (under 7 kB), and the ATmega128-based ubiquitous sensor node (under 6 kB). Also, we adopted the dynamic power management (DPM) scheme. Comparative experimental results show that UbiFOS could save energy up to 30% using DPM.

Keywords: Real-time operation systems, embedded systems, wearable computers, power management.

Manuscript received Oct. 09, 2006; revised Feb. 07, 2007.

Hee-Joong Ahn (phone: + 82 42 860 6897, email: hjahn@etri.re.kr) is with Digital Home Research Division, ETRI, Daejeon, Korea.

Moon-Haeng Cho (email: hjhong@etri.re.kr), Myoung-Jo Jung (email: mjjung@cnu.ac.kr), Yong-Hee Kim (email: root4567@cnu.ac.kr), and Cheol-Hoon Lee (phone: +82 42 821 6659, email: clee@cnu.ac.kr) are with Department of Computer Engineering , Chungnam National University, Daejeon, Korea.

Joo-Man Kim (email: joomkim@pusan.ac.kr) is with Department of BioInformation & Electronics, Pusan National University, Miryang, Korea.

I. Introduction

In recent years, there has been a rapid and wide spread proliferation of non-traditional embedded computing platforms such as cellular phones, MP3 players, and wearable computers. As applications become increasingly sophisticated and processing power increases, the application designer has to rely on the services provided by real-time operating systems (RTOSs). These RTOSs must not only provide predictable services but must also be efficient and compact. Moreover, since most embedded systems consist of a battery-operated microprocessor with a limited battery life, the RTOS should also be energy efficient [1], [2].

In this paper, we present the ubiquitous flexible operating system (UbiFOS) real-time operating system designed specifically for small mass-produced, embedded systems, such as cellular phones, MP3 players, and wearable computers. The system consists of a low-speed microprocessor and small memory. This necessitates that any RTOS used in these systems must be predictable, efficient (in performance and energy), and compact. The reason for requiring efficiency is obvious: an RTOS which incurs less overhead needs less powerful (and cheaper) processors to do the same job that a less efficient RTOS will do using more expensive hardware. Our target applications usually have their executable code (including the RTOS) stored in non-volatile memory like ROM. A smaller RTOS means that less ROM is needed to store it and less time to execute it. Therefore, if the RTOS size is a few kilobytes instead of hundreds of kilobytes, it will result in considerable cost savings in ROM chips. Since the system is mass-produced, a savings of even a few dollars per unit translates into millions of dollars of overall savings.

Our main goal in designing UbiFOS was to see which features of embedded systems we could use to reduce size and increase efficiency [3]. An important question related to reducing size was which RTOS services we would have to include in UbiFOS and which we would leave out. For this purpose, UbiFOS provides a system configuration service such as selecting a scheduling policy according to the applications.

In the next section, we present some representative features of UbiFOS. We describe how UbiFOS was optimized according to the characteristics of embedded systems. In section III, we present the features which make it efficient and easy to use. We conducted some experiments to evaluate the performance of UbiFOS, and the results are given in section IV. The experimental results show that UbiFOS is quite efficient in performance and saves energy up to 30% using the dynamic power management (DPM) scheme when running some well-known applications. Finally, we draw some conclusions and mention possible future works.

II. Overview of UbiFOS

UbiFOS is a small real-time operating system designed for embedded systems. It has been developed at the System Software Laboratory of Chungnam National University in cooperation with Aijisystem Inc. [4]. It has been ported on various embedded microprocessors such as ARM9, Calm16, MPC750, and ATmega128 [5]-[7]. As shown in Fig. 1, UbiFOS incorporates major features of a real-time operating system, such as task management, multitask scheduling, memory management, inter-task communication and synchronization, interrupt handling, a timer, and power management. For portability, UbiFOS also has a hardware abstraction layer (HAL). Since UbiFOS is designed for embedded systems which typically run on low processing power low-capacity systems with well-defined applications, it does not have provisions for dynamic modification of the task set [5].

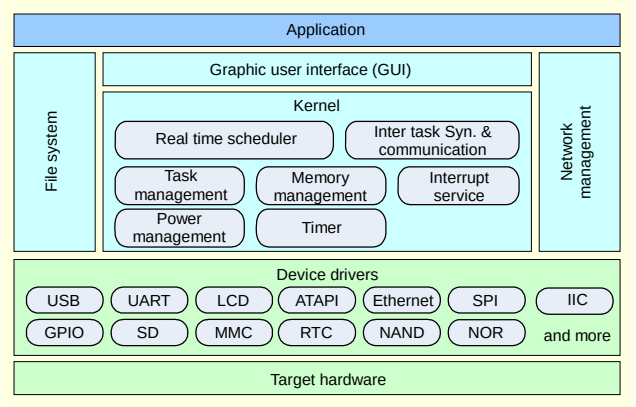


Fig. 1. Overall diagram of UbiFOS.

1. Task Management and Scheduling

Generally, an application is composed of several independent programs. Each of these independent programs is called a task or thread. Each task runs with its own execution functions, stack region, priority, and time slice. UbiFOS is a thread-or task-based operating system, where the system resources and the address space are shared among tasks, and it provides scheduling methods to solve time constraints of a real-time operating system.

A. Multitasking

Multitasking means that several tasks are simultaneously run on a single CPU. UbiFOS supports such multitasking environments. Each task has its own context and the context is saved in the task control block (TCB). The context of a task includes the following:

- program counter of task
- CPU registers
- stack and function call
- delay timer (delta ticks)
- time slice timer
- kernel control structure and task priority
- signal handler

B. Task State Transition

Each task has one of five states shown in Fig. 2: suspended, ready, running, delayed, and pending states. Once a particular event occurs, the state of a task is changed accordingly. When a task is waiting to get a particular resource, it is possible to set the maximum pending time. In this case, the task is in the state of pending + delayed state. Figure 2 shows the state transition of tasks by events. Once a task is created, it goes into the suspend state. When a created task is activated, it goes into the ready state. A task can be deleted from any of the five states.

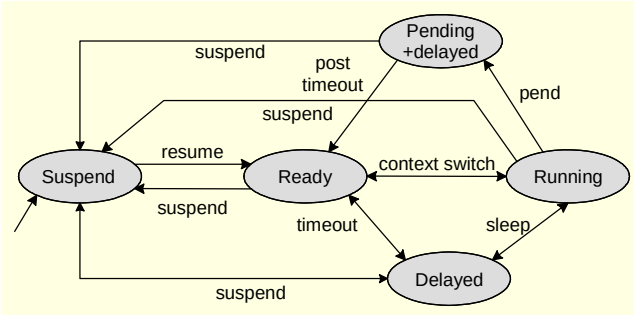


Fig. 2. State transition diagram.

C. Run-Time Overhead

The RTOS's real-time scheduler and task management

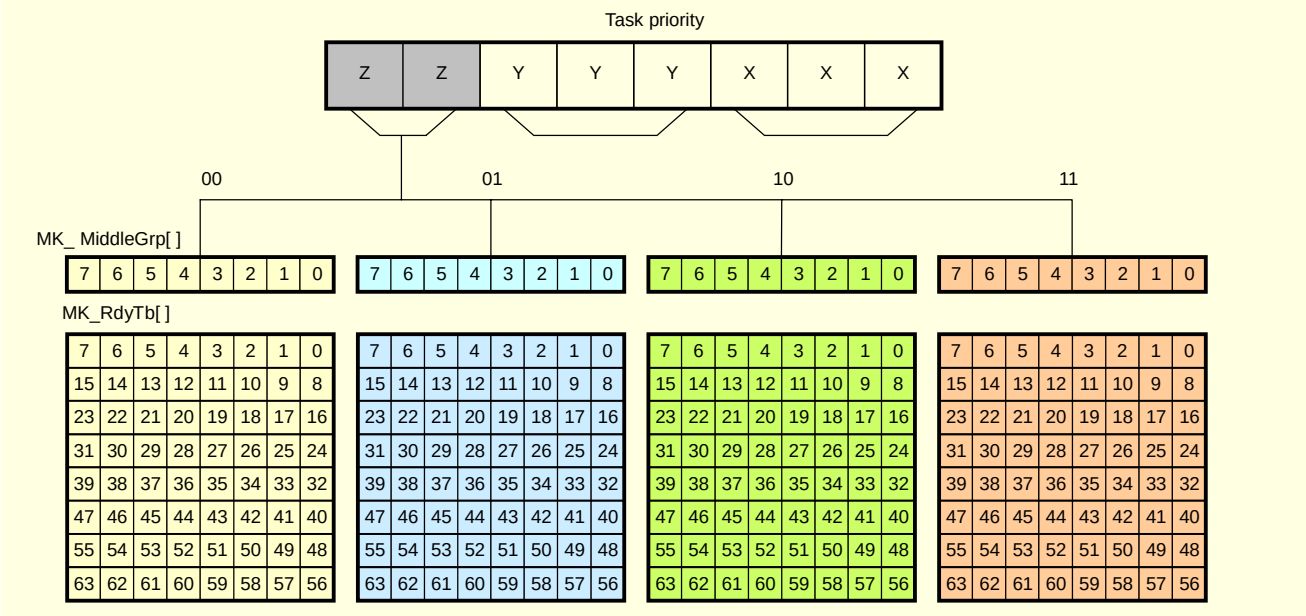


Fig. 3. Kernel structure for 256 priorities.

service should guarantee real-time deadlines. The task scheduler's overhead (Δt) is the time consumed by the execution of the scheduler code. This is related to the management of task lists and selecting the highest-priority task to execute whenever a task is blocked or unblocked. When a running task is blocked, the RTOS must update some data structures to identify the task as being blocked and then pick a new task for execution. We call the overheads associated with these two steps the blocking overhead Δt_b and the selection overhead Δt_s, respectively. Similarly, when a blocked task is unblocked, the RTOS must again update some internal data structures, incurring the unblocking overhead Δt_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 the selection overhead is incurred as well. The typical implementation for the most commercial RTOSs is to have a queue of ready tasks sorted by task priorities.

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, highest-priority task first. A single pointer, highestP, indicates the highest-priority ready task, so Δt_s is O(1) because highestP is the task which should be executed next. To block a task, one variable is updated in the TCB (that is, Δt_b = O(1)), and highestP is set to indicate the next task in the ready queue (that is, Δt_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. This is why Δt_u takes O(n) time, where n is the number of tasks. This means that the worst-case scheduling overhead increases as n increases. It has a significantly bad impact on the performance of a real-time operating system, since worst-case overheads should be taken into account so as to guarantee real-time deadlines [2], [8].

Real-time kernel services should be deterministic by specifying how long each service call will take to execute. Having this information allows the application designers to better plan their application software [9].

The µC/OS real-time kernel [10] includes a deterministic scheduler using novel data structures. Its task scheduler's overhead, Δt, 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 become increasingly sophisticated and processing power increases, this restriction on the maximum number of tasks (namely, priorities) is becoming rapidly unacceptable in many applications. In a prior study [11], we proposed a deterministic scheduling algorithm which can be generalized such that it eliminates the restriction on the maximum number of task priorities imposed on µC/OS without additional memory overhead. We have also presented the complete generalized algorithm to determine the highest priority in the ready list with 2^{2^r} levels of priorities for an arbitrary integer number of r [12].

D. Task Scheduling

In the multitasking environment, the real-time operating system assigns the CPU control to the ready task with the highest priority. Most of the operating system kernel is preemptive. This means that the running task is preempted

when a higher-priority task gets ready and the CPU control goes to that task. UbiFOS has a deterministic task scheduler using the same novel data structures as μC/OS. Its task scheduler's overhead (Δt) is constant irrespective of the number of tasks created in an application.

UbiFOS has 256 ($r = 8$) priorities from 0 to 255 for tasks and the ready list is managed on the basis of each priority. The scheduler finds the highest priority in the ready list and then transfers the CPU control to the highest priority task. Figure 3 shows the data structure to support 256 priorities. As shown in Fig. 3, the highest two bits point to one of four 64-priority groups. UbiFOS provides a priority-based preemptive scheduling policy, and for tasks with the same priority, it also provides round-robin and first-in-first-out (FIFO) scheduling policies.

2. Memory Management

UbiFOS provides a two-step scheme for dynamic memory management (see Fig. 4). The lower step is the heap storage manager which allocates and deallocates variable-size memory, and the upper step is the memory pool which allocates and deallocates fixed-size memory. The heap storage manager alone may cause a problem in two respects. First, it takes a long time to allocate memory; therefore it undermines the time determinism. In order to find a memory block with a suitable size, the entire memory block free list should be searched, and the search time varies case by case. Second, the external fragmentation of the heap can make it difficult to allocate a memory block. One way to solve this problem is to assume the necessary memory size in advance and then allocate memory from the fixed-size memory pool.

3. Semaphores

Semaphores are important means to satisfy the requirements of mutual exclusion and inter-task synchronization. A semaphore manages shared resources through mutual exclusion so it is more effective than interrupt or scheduling locks. There are three kinds of semaphores: binary semaphores, mutual exclusion semaphores, and counting semaphores.

A binary semaphore can take on one of only two values (usually values 0 and 1). The binary representation is all that is required to provide mutual exclusion protection for a critical section (see Fig. 5). Binary semaphores are therefore the most commonly used of the semaphore forms.

A mutual exclusion semaphore is a special case of binary semaphore, which serves as a key to access a shared resource (see Fig. 6).

A counting semaphore can take on any value between 0 and an upper integer value. Counting semaphores are normally used for synchronization purposes when counting out resources in use.

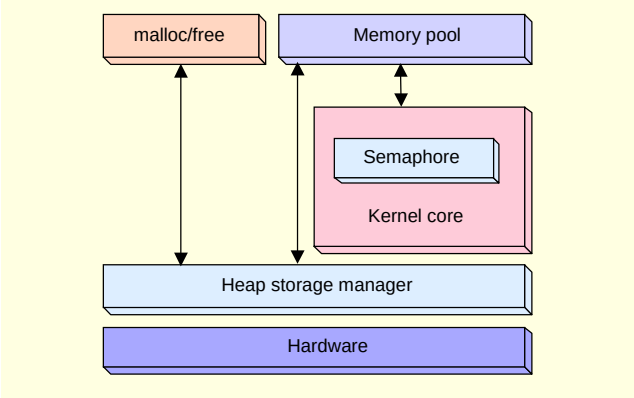


Fig. 4. Memory management.

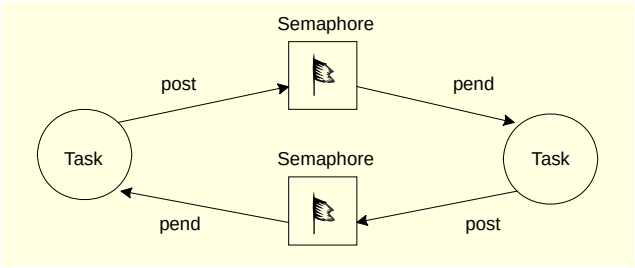


Fig. 5. Synchronizing two tasks.

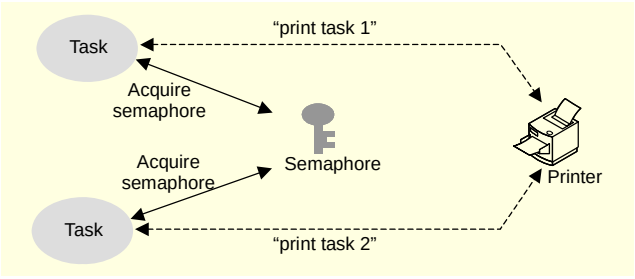


Fig. 6. Mutual exclusion between two tasks.

4. Interrupts

Interrupts are used to make a processor perform a predefined action when an asynchronous event occurs in the middle of processing. That is, it is a mechanism which can immediately respond to an internal or external event. Once an interrupt occurs, the running task stops and the control is immediately handed over to the interrupt service routine (ISR). However, in the event that an ISR needs to call a kernel service, it is necessary to protect shared data against the service call that the ISR has access to simultaneously. The simplest way to protect the shared data is to disable interrupts. By disabling interrupts, a task can work in order without being disturbed by other tasks

or interrupts. In order to effectively reduce the interrupt response time, UbiFOS divides each ISR into low-level ISRs (LISRs) and high-level ISRs (HISRs). While an ISR is working, it is impossible to create a task. It should also be impossible to call the kernel service which changes the running state of a task into another state, since if the state of a task is changed, it can cause the system to crash.

The LISR is a general interrupt handling routine which requires a fast response time. It uses the same stack as the running task. By calling MK_IRQInstall(), we can register the interrupt service routine with the relevant interrupt vector. To perform the relevant ISR according to an interrupt, MK_ServiceIRQ() is used.

The HISR is used for interrupt handling and does not require a fast response time compared with the LISR. Unlike the LISR which uses the same stack as the running task, the HISR uses its own stack and is given its own scheduling priority. The priority of an HISR is differentiated from those of tasks. Each HISR is scheduled with a higher priority than any task. Therefore, until all the activated HISRs complete, no task is executed. Like tasks, each HISR has its own control block.

5. Inter-Task Communication

UbiFOS provides various inter-task communication methods, such as shared memory which uses global variables; semaphores which provide synchronization and mutual exclusion against access to shared resources; message queues, message ports, and task ports which deliver messages among tasks; and signals which are used asynchronously for exception handling.

A. Shared Data

Providing direct accesses to shared data is the clearest method in inter-task communication. As shown in Fig. 7, the UbiFOS kernel resides in a consecutive address space so it can simply deliver shared data among tasks. Shared data may include global variables, linear buffers, linked lists, and pointers.

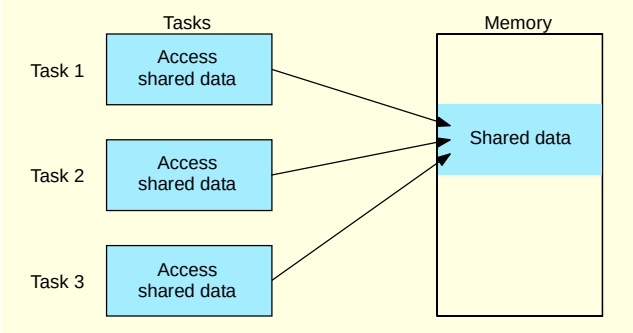


Fig. 7. Access to shared data.

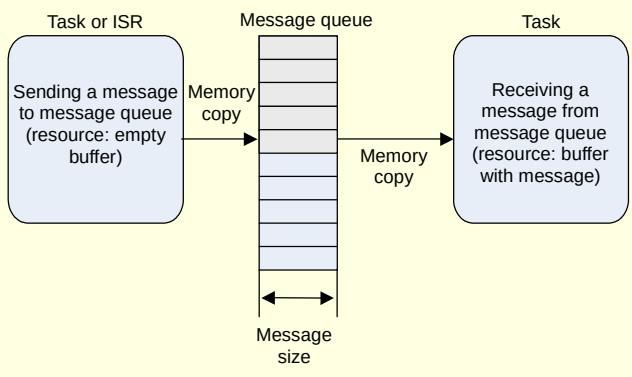


Fig. 8. Message queues.

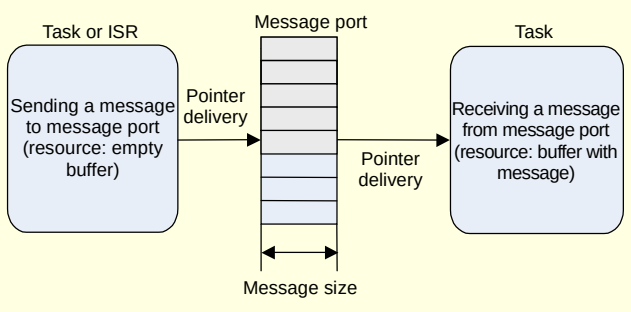


Fig. 9. Message ports.

B. Message Queues

As seen in Fig. 8, a message queue is used when a task or an ISR delivers several messages to another task. UbiFOS provides variable-size and fixed-size message queues, depending on the variability of the message size which is delivered through message queues. Note that message delivery is done through copy, not through pointers.

C. Message Ports

Unlike message queues, when a message is sent to a message queue or a message is received from a message queue, the message port delivers a message pointer instead of copying the message (see Fig. 9). In other words, in contrast to message queues, there is no copy overhead in message ports.

D. Task Ports

Task ports provide one-to-one communication among tasks (see Fig. 10). They are similar to signals, but while signals provide asynchronous communication, task ports provide synchronous communication. Because of this difference, a task port is implemented using a field of the task control block (TCB) rather than a separate data structure, and it is used for sending short messages among tasks.

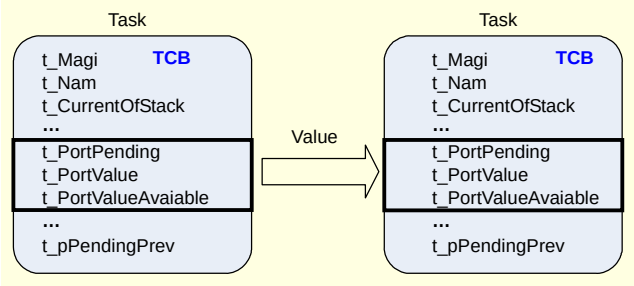


Fig. 10. Task ports.

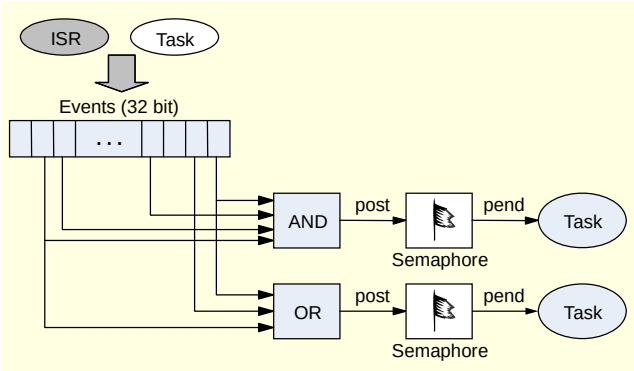


Fig. 11. Event flags.

E. Event Flags

An event flag is used when a task needs synchronization for several events. It can enter into a waiting state and then stay blocked until one of the specified events occurs; therefore, events can be used to synchronize multiple tasks or as a very crude method of inter-task communication. As seen in Fig. 11, there are two types of synchronization: disjunctive synchronization (logical OR), which synchronizes if any desirable event occurs and conjunctive synchronization (logical AND), which synchronizes if all the desirable events occur.

F. Signals

Signals are used to inform tasks of asynchronous events. In terms of asynchronous events, signals are similar to interrupts. However, while an interrupt informs only the running task of an asynchronous event, signals inform an asynchronous event not only to ready tasks, but also to delay pending tasks which are waiting for the event. Therefore, in order to handle signals, there should be a context separate from that of the running task. Signals can also be used for error or exception handling, rather than just for general inter-task communication.

6. Timers

Embedded systems typically require some time-based task activations. A timer is tied to the system clock (or to any counter), and is triggered when the count reaches the timer value. When triggered, a timer either activates a task or signals the task with an event. Timers are defined statically at system generation, but the time instant at which they are triggered is set dynamically to either relative or absolute values. Timers may also be set to be triggered cyclically to activate tasks. Each timer can be activated or disabled dynamically.

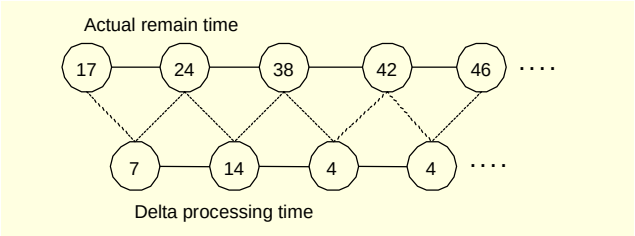


Fig. 12. Delta processing diagram.

Delta processing is a method to calculate the expiration time of each activated timer by maintaining only the difference between every two consecutive timers. Using delta processing, we can reduce the expiration times of all activated timers by simply reducing the expiration time of the header in the timer list as shown in Fig. 12.

7. Power Management

The current embedded processors are so power-efficient that the processor may no longer be the major energy-consumer in systems which include high-performance memories and large color displays. We are committed to enabling aggressive power management strategies which encompass the entire system. For example, scaling bus frequencies can drive significant reductions in system-wide energy consumption. The real-time operating system and device drivers may also manage the power consumption of peripheral devices, for example, by spinning down disks during periods of inactivity. Highly integrated processors often include software-controlled clock management capabilities to reduce power consumed by inactive peripherals and peripheral controllers. The IBM and MontaVista Software [13] developed a general, flexible, DPM architecture for embedded systems. UbiFOS adapts the DPM scheme for energy savings. Depending on the embedded microprocessor, DPM has the following policy architecture features. First, the *operating policies* specify the components and device-state transitions which can ensure reliable operation in line with the power management strategy.

Second, the *operating points* are described by such parameters as the core voltage, CPU, and bus frequencies and the states of peripheral devices. Third, regarding operating states, some rules and mechanisms are required to move the

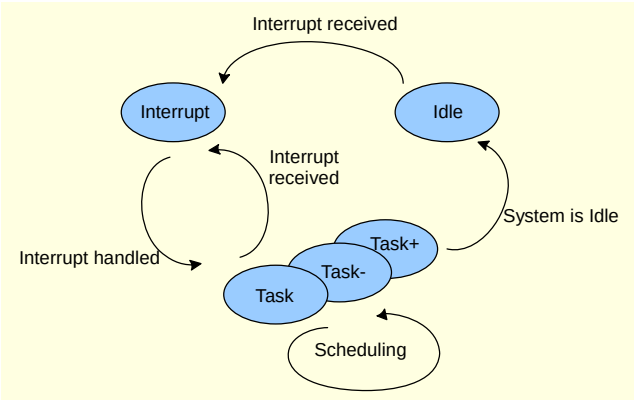


Fig. 13. Operating state transition diagram.

system from one operating point to another. Current dynamic control mechanisms may set operating points in response to changes in activity or in response to the requests of key programs (see Fig. 13).

8. System Configuration

In embedded applications such as wearable personal computing applications, most of the parameters are statically defined at system generation. UbiFOS uses a system description file which specifies all of the parameters of the application. Each task is described by the name, the scheduling priority, and resources it uses. Also, the amount of stack space needed by the task to store its local variables as well as the processor state when the task is traded is specified. All the resources, counters, timers, and interrupt handlers are also declared. This description file is processed by a system generation script which produces a C language which is compiled with the application code and linked with the kernel code to produce the runtime object file to be executed on the embedded processor. The generated files perform the initialization of the operating system.

III. Kernel Downsizing and Improvements

In designing UbiFOS, we have incorporated several optimizations which reduce the memory requirements of the kernel, since the available RAM size is often the most restrictive constraint in embedded systems such as wearable computers. Also, we have incorporated several improvement techniques and efficient power management techniques.

1. Kernel Optimization

Some applications allow only one task per priority level. In the event that every task has a unique priority level in the system, UbiFOS uses the priority-based preemptive scheduler or the bitmap scheduler. In the event that there is more than one task per priority level, we can use the priority-based preemptive scheduler along with the round-robin scheduling policy for tasks with the same priority level.

UbiFOS uses map (8 bytes) and unmap (512 bytes) tables for priority-based preemptive scheduling. Also, it uses an array of lists for the round-robin scheduler, where the index into the array is the priority level and each item in the array is linked-listed. The scheduler should locate the first task in the ready list. To do this, the priority-based preemptive scheduler uses map and unmap tables. On the other hand, the bitmap scheduler uses just a bit field without using any arrays or tables. Even though it takes time to scan the bit field, when the number of tasks in the system is not so high (that is, less than 64), the bitmap scheduler is most efficient in terms of both performance and size constraint. Depending on the application, we can select the best-suited scheduler. For example, if the application is composed of few tasks (as is the case with wearable computers), we can use the bitmap scheduler.

2. Kernel Improvements

Time is of central importance in real-time systems. Applications often need to take actions based on the current time. One way UbiFOS allows this is through the timer mechanism. Applications can set a timer to expire after a user-specified number of clock ticks. UbiFOS also incorporates counters which are incremented every time a certain event occurs (such as a hardware clock timer interrupt). Considering the importance of time in real-time systems and the need to access the current time value, UbiFOS offers a system call to read the system clock counter.

Often it is necessary for an application to disable some or all interrupts within certain critical portions of code. To do this, UbiFOS has MaskInterrupt and UnmaskInterrupt system calls. Using these system calls, we can enable or disable one or some interrupts in the system using a bit field as a parameter. Even though these system calls are generally used to control interrupts, scanning a bit field takes time; therefore, it can cause significant performance degradation in applications which frequently disable and enable interrupts. Very common use of interrupt control is to disable all interrupts, perform some critical functions, and then restore the interrupt function to its prior settings. For this reason, we provide two additional system calls: InterruptDisable and InterruptEnable which will be used at the beginning and end of each critical section. Since many microprocessors provide a mechanism for disabling all interrupt processing through a control register (or status register) without changing

individual interrupts, these system calls can be implemented very efficiently.

3. Efficient Power Management Strategies

In this subsection, we describe some low-power strategies which we have implemented and tested on the experimental platform based on the s3c2440 (using the ARM920T core) processor produced by Samsung [14]. This processor provides three different frequency settings: 399 MHz, 266 MHz, and 96 MHz. Table 1 shows the DPM policy for each of these frequency settings on the platform. In a single policy strategy (idle scaling), the system runs at full speed when processing the workload, but it runs at reduced frequencies only when the system is idle. Multi-policy strategies under DPM would include the well-known load scaling (LS). In Table 1, default, LS399, LS266, and LS96 are examples of such policies. These strategies attempt to balance the system operating point with current or predicted processing demands in order to run the system at the most efficient operating point with minimum idle time. All of our LS policies also have idle scaling.

Table 1. DPM policies (MHz).

Policy	Task+	Task	Task-	Idle
Default	399	399	399	399
Idle scaling	399	399	399	96
LS399	399	399	266	96
LS266	399	266	96	96
LS96	399	96	96	96

IV. Evaluation and Measurements

For performance evaluation, we implemented UbiFOS on the i.MX21 (with ARM926EJ core) processor [15] developed by Freescale and measured some of the primitive real-time operating system performance characteristics. Also, to evaluate power efficiency, we ported UbiFOS on the MBA2440 (with s3c2440 processor) evaluation platform developed by Aijisystem Inc. [4] and measured the power consumed while running two well-known application programs: the PushPush game and a WAV sound programs. The PushPush game program written in Java ran on the KVM (K Virtual Machine) ported on UbiFOS [16]-[18].

1. Performance Evaluation

One of the significant overheads introduced by a real-time operating system is the scheduling overhead. In this subsection, for performance evaluation, we show the scheduling overhead along with the interrupt latency and the overhead of each inter-task communication mechanism provided by UbiFOS on the 266 MHz i.MX21 processor.

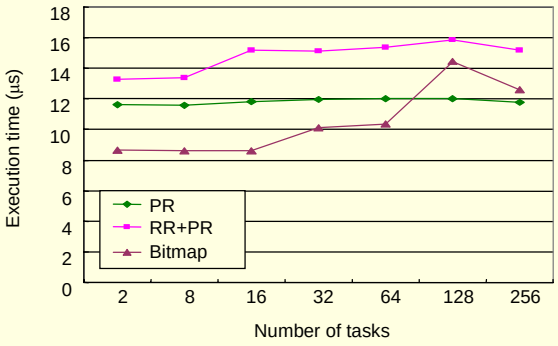


Fig. 14. Scheduling overhead.

A. Scheduling Overhead

Figure 14 shows the scheduling overhead of the three schedulers with various numbers of tasks in the system: the priority-based preemptive scheduler, the priority-based preemptive scheduler with round-robin, and the bitmap scheduler. Here, the scheduling overhead is the sum of the context switch time, the time to insert a task into the ready queue, and the time to select the highest-priority task. As shown in Fig. 14, when the number of tasks is 16, the scheduling overheads are 11.8 μs, 15.1 μs, and 8.6 μs for the priority-based preemptive scheduler, the priority-based preemptive scheduler with round-robin, and the bitmap scheduler, respectively. As expected, the bitmap scheduler outperforms others when the number of tasks in the system is less than or equal to 64.

B. Interrupt Execution Time and Latency

As previously mentioned, UbiFOS provides two efficient system calls to enable and disable all the interrupts in the system: InterruptEnable and InterruptDisable. Each of these system calls was measured to take under 1.15 μs. The (worst-case) interrupt latency of the UbiFOS kernel was measured to be less than 9.3 μs.

C. Inter-task Communication Overhead

UbiFOS provides various mechanisms for inter-task communication, such as message queues, message ports, and task ports. Figure 15 shows the inter-task communication overhead for each mechanism with messages of various sizes. As expected, the overhead of the message queue increases as

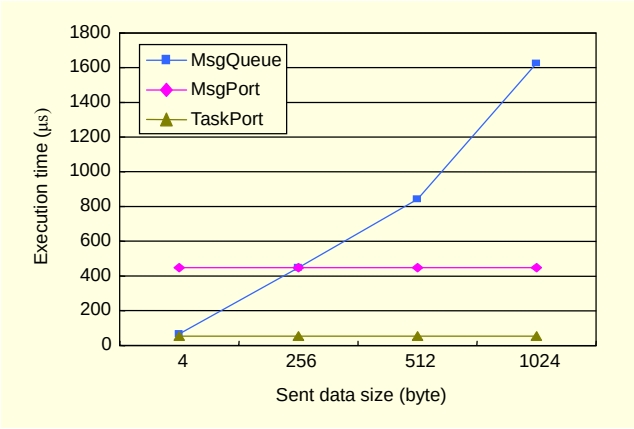


Fig. 15. Inter-task communication overhead.

the message size increases because it must copy each message. On the other hand, the other two mechanisms just transfer the pointer of the message; therefore their performance does not vary with message size. The overhead of the task port is the lowest since it makes use of a field in the TCB data structure. However, it should be noted that the task port can transfer only one message for each communication round.

2. Memory Requirements

UbiFOS is designed to be quite memory-efficient. Usually, the memory requirement of a real-time operating system is variable depending on applications. In a prior study, we ported UbiFOS on the ARM9-based DVD player (20 kB) [6], the Calm16-based MP3 player (under 7 kB) [7], and the ATmega128-based ubiquitous sensor node (under 6 kB).

3. Power Measurement

To evaluate power efficiency, we ported UbiFOS on the MBA2440 (with s3c2440 processor) evaluation platform with the DPM policies shown in Table 1 and measured the power consumed while running the two previously described applications. The MBA2440 platform is composed of a 399-MHz s3c2440 processor, a 64-MB SDRAM, a TFT display, a 32-MB NAND flash memory, a 16-MB NOR flash memory, USB, and other peripherals. Figure 16 shows the current level profiles for the corresponding policies and applications with the supplying voltage level fixed at 5 V. The average current levels are summarized in Table 2. Since the WAV program plays with the LCD display off, it runs at lower current levels than the PushPush game program which should play with the LCD display on all the time.

Since the supplying power at each instant is proportional to the current level at that instant, the total energy consumed during a given time interval is proportional to the average current level during the interval. Therefore, Table 2 demonstrates that, using the low-power policy (such as LS96), we could save energy consumption by about 19% and 30% for the WAV and PushPush applications, respectively.

Table 2. Average current values.

	Default	LS399	LS266	LS96
WAV	0.239	0.198	0.198	0.194
PushPush	0.348	-	-	0.244

The DPM scheme was also implemented under PowerPC Linux 2.4 in [19]. The size of the Linux 2.4 kernel on PowerPC is about 1.1 MB including both code and data [20]. Experimental results with MP3 and MPEG4 applications showed energy savings in a range from 26% to 58%. In the experiments, however, the supply voltage was also scaled along with the operating frequency. The energy dissipated per cycle scale quadratically to the supply voltage ($E \propto V^2$). Therefore, assuming that the supply voltage scales linearly with the operating frequency, the energy savings would be in an approximate range between 10% and 25% if we fix the supply voltage at some (or maximum) level as in our experiment.

V. Conclusion

In this paper, we introduced major features of the UbiFOS real-time operating system. Performance evaluation demonstrated that UbiFOS with a footprint size of 26 kB is quite efficient, especially for small to medium-sized embedded systems such as cellular phones, MP3 players, and wearable computers. For energy savings, we incorporated the DPM scheme. Experimental results with some well-known applications show that UbiFOS could achieve energy savings up to 30% using the DPM scheme.

In the future, we would like to further adjust UbiFOS for task sets typically found on specific application domains such as wearable computers. We would also like to look into creating a better system configuration method to enable applications to efficiently use the optimizations embedded into the UbiFOS kernel. In this work, we only scaled the operating frequency. However, if we scale down both the operating frequency and the supply voltage, we can save much more energy, since the energy consumption is quadratically proportional to the supply voltage. It may also useful in future work to incorporate dynamic voltage and frequency scaling algorithms into UbiFOS.

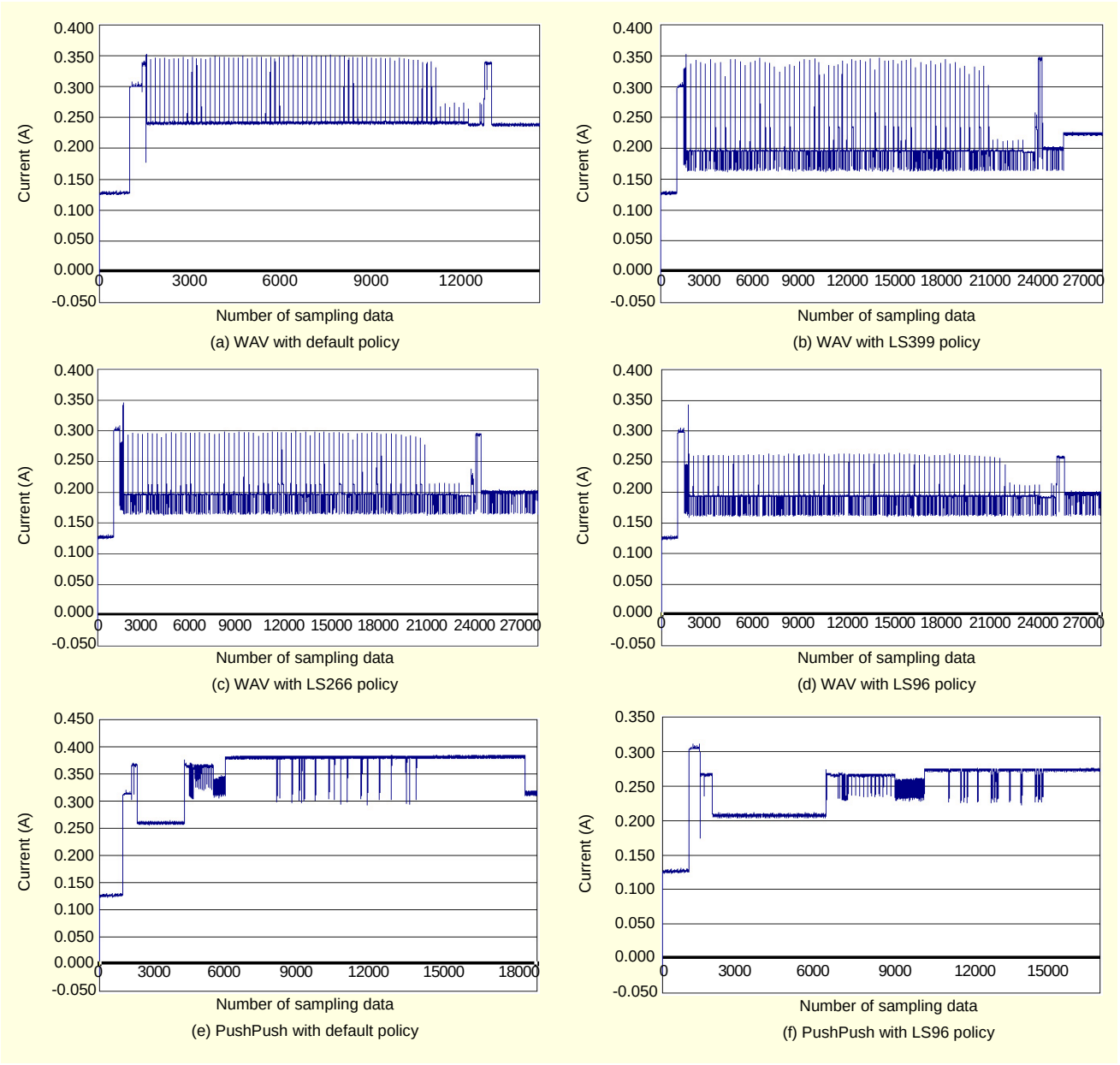


Fig. 16. Current level profiles.

References

[1] M.T. Schmitz, M. Bashir, and M. Al-Hashimi, *System-Level Design Techniques for Energy-Efficient Embedded Systems*, Kluwer Academic Publishers, Boston, 2004.

[2] C.M. Krishna and K.G. Shin, *Real-Time Systems*, McGraw-Hill, 1997.

[3] O. Li and C. Yao, *Real-time Concepts for Embedded Systems*, CMP Books, 2003.

[4] Aijisystems, "UbiFOS: Ubiquitous Flexible Real-Time Operating Systems," http://www.aijisystem.com.

[5] H.S. Park et al., "Design of Open Architecture Real-Time OS Kernel," *KISS Autumn Conf.*, vol. 2, 2002, pp. 418-420.

[6] H.J. Ahn et al., "Design and Implementation Real-Time Operating Systems for DVD Player," *KISS Autumn Conf.*, vol. 2, 2003, pp. 340-342.

[7] M.H. Cho et al., "Design and Implementation of Light-Weight Real-Time Operating System for Audio Player," *KISS Autumn Conf.*, vol. 2, 2006, pp. 328-330.

[8] K. Ramamritham and J.A. Stankovic, "Scheduling Algorithms and Operating System Support for Real-Time Systems," *Proc. of IEEE*, 1996, vol. 82, no. 1, pp. 55-67.

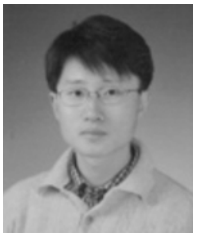
[9] J.W.S. Liu, *Real-Time Systems*, Prentice Hall, New Jersey, 2000.

[10] J.J. Labrosse, *μC/OS, The Real-time Kernel*, R&D Publications,

1993.
[11] S.J. Oh et al., "Deterministic Task Scheduling for Embedded Real-Time Operating Systems," *IEICE Trans. Inf. & Syst*., vol. E87-D, no. 2, Feb. 2004, pp. 123-126.
[12] M.J. Jung et al., "Generalized Deterministic Task Scheduling Algorithm for Embedded Real-Time Operating Systems," *Proc. ESA'06*, June 2006, pp. 79-82.
[13] IBM and MontaVista Software, "Dynamic Power Management for Embedded Systems," http://www.research.ibm.com/arl/projects/dpm.html, Nov. 2002.
[14] *S3C2440A 32-bit Microprocessor User's Manual*, 0.12, Samsung Electronics, 2004.
[15] *i.MX21 Application Processor Reference Manual*, 2, Freescale, 2005.
[16] Kim Topley, *J2ME in a Nutshell*, O'Reilly, 1st ed., 2002.
[17] Sun Microsystems, JSR-139, *Connected Limited Device Configuration (CLDC) Specification*, Version 1.1, 2003.
[18] Sun Microsystems, JSR-118, *Mobile Information Device Profile Specificaton*, Version 2.0, 2002.
[19] B. Brock and K. Rajamani, "Dynamic Power Management for Embedded Systems," *Proc. of IEEE Int'l SoC Conf.* (*SoCC 2003*), Sep. 2003, pp. 416-419.
[20] Embedded PowerPC Linux Boot Project, http://ppcboot.sourceforge.net/.

Hee-Joong Ahn received a master's degree in computer engineering from Chungnam National University, Daejeon, S. Korea, in 2004. Since 2004, he has been a member of the Engineering Staff in Electronics and Telecommunications Research Institute (ETRI), S. Korea. He has been engaged in the development of a wearable computing system. His research interests include small and low power consumption real-time operating systems, and wearable computing systems.

Moon-Haeng Cho received the BS and MS degrees in computer engineering from Chungnam National University, Daejeon, S. Korea, in 2004 and 2006, respectively. Since 2006, he has been pursuing the PhD degree at Chungnam National University, Daejeon, S. Korea. His research interests include embedded system, real-time and ubiquitous computing, and small and low-power real-time operating systems.



Myoung-Jo Jung received the MS degree in computer engineering from Chungnam National University, Daejeon, S. Korea, in 2003. Since 2003, he has been pursuing the PhD degree in System Software Laboratory in Chungnam National University. His research interests include Java virtual machine, real-time scheduling, and real-time operating systems.

Yong-Hee Kim received the MS degree in computer engineering from Chungnam National University, Daejeon, S. Korea, in 2003. Since 2003, she has been pursuing the PhD degree in System Software Laboratory in Chungnam National University. Her research interests include real-time systems and fault-tolerant systems.

Joo-Man Kim received the BS degree in computer science from Soongsil University, Seoul, Korea in 1984 and the MS and PhD degrees in computer engineering from Chungnam National University, Daejeon, Korea, in 1998 and 2003, respectively. From 1985 to 2000, he worked for ETRI in Daejeon, Korea as a principal member of research staff at the OS research team. From 2000 to 2005, he was an assistant professor in the Dept. of Information and Communications Engineering, Miryang National University. He is currently an associate professor in the Dept. of BIO Information and Electronics Engineering, Pusan National University, Miryang, Korea. His research and teaching interests are in embedded systems, real-time and ubiquitous computing.

Cheol-Hoon Lee received the BS degree in electronics engineering from Seoul National University, Seoul, Korea in 1983, and the MS and PhD degrees in electrical engineering from Korea Advanced Institute of Science and Technology, Seoul, Korea in 1988 and 1992, respectively. From 1983 to 1994, he worked for Samsung Electronics in Seoul, Korea as a researcher. From 1994 to 1995 and from 2004 to 2005, he was with the University of Michigan, Ann Arbor, as a research scientist at the Real-Time Computing Laboratory. Since 1995, he has been a professor in the Department of Computer Engineering, Chungnam National University, Daejeon, Korea. His research interests include parallel processing, operating systems, real-time and fault-tolerant computing, and microprocessor design.