Real-Time Operating Systems for Small Microcontrollers

Real-time operating systems have gained popularity in microcontroller- and processor-based embedded system design. This article discusses differences between RTOSs and generic operating systems, the advantages and disadvantages of using RTOSs for small microcontroller system development, and the benchmarking methods used for RTOSs. Benchmarking results for four RTOSs show no clear winner, with each RTOS performing better on some criteria than others.

Tran Nguyen Bao Anh
Singapore Engineering Center

Su-Lim Tan
Nanyang Technological University

Real-time embedded systems serve various purposes, such as to control or process data. A real-time operating system is a piece of software with a set of APIs that developers can use to build applications. RTOSs support the need of some embedded systems to meet deadlines. However, using an RTOS doesn't guarantee that a system will always meet the deadlines, because these systems also depend on the overall system's design.

Although RTOSs for embedded systems are predominantly used in high-end microprocessors or microcontrollers with 32-bit CPUs, there is a growing trend to provide these features in mid-range (16-bit and 8-bit) processor systems.

Generic operating systems versus RTOSs

Operating systems manage resource sharing in computer systems. Unlike generic operating systems, an RTOS is specifically designed to achieve real-time responses. RTOS differ from generic operating systems in several other ways as well.

First, RTOSs offer preemptive, priority-based scheduling. A *scheduling scheme* refers to how the RTOS assigns CPU cycles to tasks for execution. Thus, scheduling schemes affect how the operating system will execute the various software programs. Most generic operating systems are time-sharing systems, which allocate tasks the same number of time slices for execution (for example, by round-robin scheduling). RTOSs often assign tasks priorities, and higher-priority tasks can preempt lower-priority tasks during execution (*preemptive* scheduling). Other RTOSs adopt *cooperative* scheduling, which usually implies that the running task must explicitly invoke the scheduler to switch between tasks.

In addition, RTOSs allow predictable task synchronization. In generic operating systems, task synchronization is unpredictable because the operating system can directly or indirectly introduce delays into the application software. In an RTOS, task synchronization must be time-predictable. The system services must have a known and expected duration of execution time.

The key difference between generic operating systems and RTOSs is that an RTOS supports deterministic behaviors. In an RTOS, task dispatch time, task switch latency, and interrupt latency must be time-predictable and consistent even when the number of tasks increases. In contrast, generic operating systems (mainly owing to their time-sharing approach) reduce a system's overall responsiveness and don't guarantee service call execution within a certain amount of time when the number of tasks increases. Dynamic memory allocation (*malloc()* in C language), although widely supported in generic operating systems, isn't recommended in RTOSs because it generates unpredictable behavior.[1] Instead, RTOSs provide fixed-size memory allocation in which they allocate only a fixed-size memory block for every request.

RTOSs for small-scale embedded systems

Available RTOSs include commercial, proprietary, and open source systems. Many system designers believe that small-scale embedded systems designed using small microcontrollers (that is, microcontrollers with a maximum ROM of 128 Kbytes and maximum RAM of 4 Kbytes[2]) don't need an RTOS. However, RTOSs offer significant advantages for this range of devices.[2,3]

For example, developers can use an RTOS to optimize software development. In system development using small microcontrollers, software productivity is a critical issue because of time-to-market pressures as well as a shortened development cycle (see the *Embedded System Design* 2004 survey at http://www.embedded.com/columns/survey). For projects involving complex code, an RTOS is an efficient tool to manage the software and to distribute tasks among developers. Using an RTOS lets project leaders partition the entire software into modular tasks that individual programmers can handle. Moreover, other developers can develop the low-level drivers.

An RTOS also provides better and safer synchronization. In small embedded system development without an RTOS, developers often use global variables for synchronization and communication among modules and functions. However, using global variables can lead to bugs and software safety issues, especially in highly interrupt-driven systems (see the *Embedded System Design* 2006 survey at http://www.embedded.com/columns/survey). Because these global variables are often shared and accessed among functions, they're highly vulnerable to corruption during program execution. As the code begins to grow, these bugs are more deeply hidden and thus more difficult to uncover. Consequently, development time can lengthen even for such small-scale systems. With an RTOS in place, tasks can safely pass messages or synchronize with each other without corruption problems.

Most RTOSs provide APIs that let developers manage system resources to establish functions including task management, memory pool management, time management, interrupt management, communication, and synchronization. RTOSs provide the abstraction layer for developers to freely structure the software, to achieve cleaner code, and even to quickly port across different hardware platforms with few code modifications. In small system development in particular, hardware cost is a critical constraint and development time is usually short.

Time management functions let software designers achieve task delay, timer handling, or time-triggered processing without having to understand the underlying hardware mechanisms. Achieving timing-related features in a small system with no RTOS can be tricky because the software designer must understand the underlying peripherals (such as timers), how to use them, and how to link them with the top-level application code. Any modification, such as a longer delay time, would require the developer to re-examine the code and peripherals to make changes appropriately. To port the software to another platform using a different microcontroller with a different set of peripherals, the developer must rewrite these timing features. Unless the project involves a critical timing issue with a unique hardware peripheral, using an RTOS can help significantly speed up the development time required to tackle these timing issues.

Ganssle illustrates the importance of RTOSs in small system design using a printer system example.[3] Without an RTOS, the

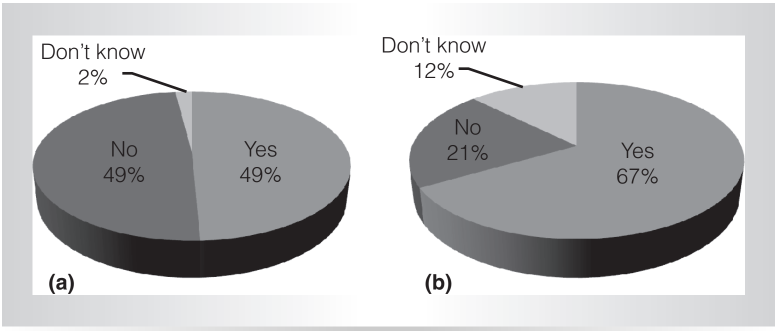

Figure 1. RTOS usage as reported in the 2004 *Embedded Systems Design* embedded market survey (http://www.embedded.com/columns/survey): have used (a) and would consider using (b). (Copyright 2009 TechInsights. Used with permission.)

Table 1. Results from the 2006 *Embedded System Design* survey on the types of operating systems used (http://www.embedded.com/columns/survey).

Type of operating system	Current project (%)	Upcoming project (%)
Commercial operating system	51	47
Internally developed or in-house operating system	21	17
Open source operating system without commercial support	16	19
Commercial distribution of an open source operating system	12	17

(Copyright 2009 TechInsights. Used with permission.)

system uses a single chunk of code to manage all printer activities—paper feeding, user-input reading, and printing controls. An RTOS lets individual tasks manage each of these activities. Except for passing status information, each task doesn't need to know much about what other tasks are performing. Hence, having an RTOS in place can help in partitioning the software in the time domain (for tasks to run concurrently) and in terms of functionalities (for each task to perform a specific operation).

Figure 1 shows results from the 2004 *Embedded Systems Design* embedded market survey on the number of developers who have used and would consider using an RTOS in their current and upcoming projects. (Results from the 2004–2006 surveys are available at http://www.embedded.com/columns/survey.) In 2004, more than 49 percent of developers said they had used an RTOS. This percentage rose to 80.9 percent in the 2005 survey, and was 71 percent in the 2006 survey. The percentage of developers who would consider using an RTOS in an upcoming project was 66.6 in 2004 and 86 in 2005, indicating a steady trend toward using RTOSs.

Table 1, which shows results from the 2006 *Embedded System Design* embedded market survey, suggests another trend in RTOS selection. Companies are moving toward open source RTOSs—from 16 percent in current projects to 19 percent in upcoming projects; and toward commercial distributions of open source RTOSs—from 12 percent in current projects to 17 percent in upcoming projects. Use of commercial and in-house systems, although currently extensive, is declining—from 51 percent in current projects to 47 percent in upcoming projects for commercial operating systems, and from 21 to 17 percent for in-house operating systems. In the 2007 survey, commercial operating system use drops further, to 41 percent.[4] The 2007 survey also shows that the key influencing factors in RTOS selection for commercial operating systems are the quality and availability of technical support. If adequate technical support is unavailable, companies might look for other, more cost-effective choices.

RTOSs also have some disadvantages when used for small microcontrollers. Because an RTOS consumes additional memory (both ROM and RAM), computational resources, and power,[5] system designers must determine whether the system can absorb these overheads. For small microcontrollers, the RTOS must have compact ROM and RAM requirements. Various RTOSs are available for these devices, and some are flexible enough that designers can configure them to have only those functions and APIs that the application requires,[2,6-8] thus allowing a smaller code size. In addition, most RTOSs require a periodic timer (an operating system "tick")[9] to execute the scheduler and other relevant system services. RTOS services such as task synchronization must have a known execution time (that is, the amount of time it takes for a task switch to occur). Depending on these timing factors, and by making use of the relevant RTOS services, the system designer can decide on the API suite and structure the whole system. Hence, designers must understand the performance measurements and benchmarking metrics among RTOSs.

RTOS benchmarking

Most RTOS benchmarking approaches are based on applications or on the most frequently used system services (fine-grained benchmarking).[10] Because applications have different requirements, benchmarking against a generic application won't reflect an RTOS's strengths and weaknesses. Benchmarking methods based on frequently used system services include the Rhealstone benchmark, which measures task switch time, preemption time, interrupt latency time, semaphore shuffling time, deadlock breaking time, and datagram throughput time.[11] However, Rhealstone is unsuitable for RTOS benchmarking for several reasons. First, few RTOSs can break deadlock (as we discuss later). Datagram throughput time is based on message passing, in which the RTOS copies messages to a memory area managed by the operating system. However, not all RTOSs use the same message-passing approach. Some RTOSs only pass the memory pointer, so there's no need to use the special memory area managed by the operating system. This approach is more suitable for small microcontrollers, which have no extra memory for operating system internal use. Interrupt latency time in Rhealstone isn't determined by the RTOS; rather, it depends purely on the CPU architecture. Rhealstone is generally somewhat ad hoc and doesn't cover other common situations in real-time applications.[10]

Garcia-Martinez et al. propose several metrics based on frequently used system services: response to external events (interrupt), intertask synchronization and resource sharing, and intertask data transfer (message passing).[10] Intertask data transfer is similar to Rhealstone's datagram throughput time. In the "response to external event (interrupt)" test, the interrupt handler wakes another task via a semaphore. In this case, using a semaphore doesn't seem to be the best approach. Waking up the task directly using a system service call (such as a sleep/wakeup service call) is a better approach to reduce the overhead delay.

Other researchers have proposed tests for measuring message transfer duration and communication through a pipe, the speed of task synchronization through proxy and signal, and task-switching duration.[12] These metrics are based only on the RTOS platform distributed by QNX; some concepts, such as proxy and signal, don't exist on most RTOSs.

RTOS features and API comparison

We based our RTOS comparisons on online documentation and APIs. We examined the following systems: μITRON,[2,13] μTKernel,[7] μC-OS/II,[6] EmbOS,[14] FreeRTOS,[8] Salvo (http://www.pumpkininc.com), TinyOS (http://www.tinyos.net), SharcOS (http://www.sharcotech.com), eXtreme Minimal Kernel (XMK, http://www.shift-right.com/xmk/index.html), Echidna (http://www.ece.umd.edu/serts/research/echidna/index.shtml), eCOS,[15] Erika,[16] Hartik,[17] KeilOS (http://www.keil.com/rtos), and PortOS.[18]

Criteria for comparing RTOSs

Developers base their choice of an RTOS on criteria such as language support, tool compatibility, system service APIs, memory footprint (ROM and RAM usage), performance, device drivers, operating system awareness, debugging tools, technical support, source and object code distribution, licensing scheme, and vendor reputation.[19] They might also consider installation and configuration, RTOS architecture, API richness, documentation and support, and tool support.[20] As Figure 2 shows, the *Embedded System Design* 2005–2007 embedded market surveys found that real-time capability is the top criteria for RTOS selection.[4]

We used these documents to establish a list of criteria for comparing the RTOSs, which we describe in the following sections. This article's scope limits the number and type of criteria we could consider. Criteria

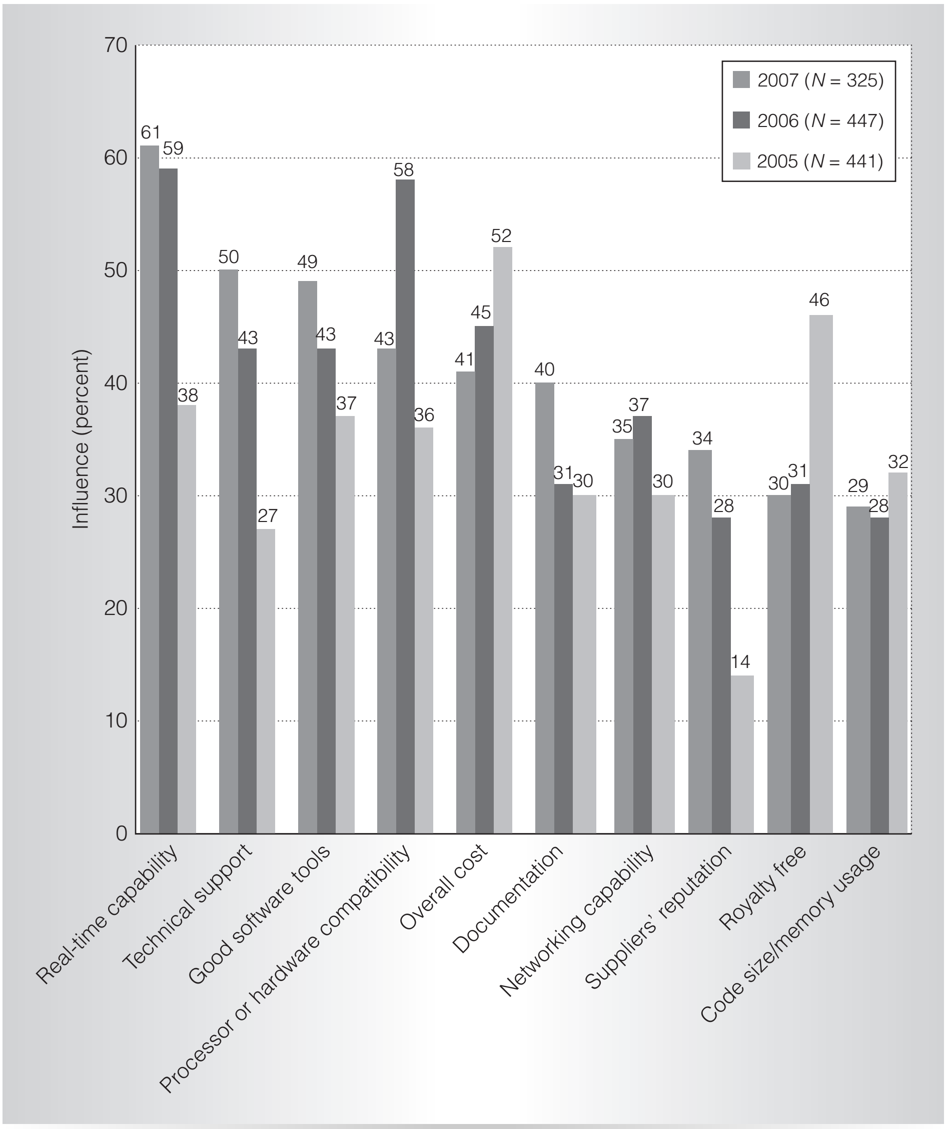

Figure 2. Influential factors in operating system selection according to *Embedded Systems Design* embedded market surveys in 2005, 2006, and 2007 (http://www.embedded.com/columns/survey). The total number of people surveyed is $N = 441$. (Copyright 2009 TechInsights. Used with permission.)

such as suppliers' reputation and company reputation are subjective; overall cost depends on the project and application; royalty fee is normally based on quantity, even though RTOS vendors might use other business models to charge their customers (such as per application, product model, or microcontroller unit [MCU] model); and memory footprint might be unavailable and depends on the compiler settings and RTOS configurations.

Design objective. RTOSs can be open source, personal hobby based, or commercial. A designer should understand an RTOS's history and the motivation that led to its creation. A personal hobby-based RTOS will likely be less stable than a popular open source or commercial RTOS.

Author. A designer should also know who originated the RTOS—whether it was a person or an organization.

Scheduling scheme. It's also important to investigate the RTOS's scheduling approach to determine whether it uses preemptive scheduling, cooperative scheduling, or some other scheduling scheme.

Real-time capability and performance. Real-time capability generally describes whether a system can meet the timing deadline. Using an RTOS takes up CPU cycles; however, the RTOS must not have nondeterministic behaviors. The amount of CPU cycles and time consumed by the RTOS for any service call should be measurable, and of low or acceptable value to the system designers. Real-time capability and performance information aren't available for some RTOSs. Even if this information is available, it might not be based on the hardware platform a designer is using.

Memory footprint. In addition to CPU cycles, an RTOS consumes ROM and RAM space, which could increase the ROM and RAM sizes for the entire system. A trade-off always exists between the memory footprint and the functionalities required from the RTOS. More robust and reliable APIs typically require more lines of code. However, simple APIs require only a minimum amount of code. Hence, designers should understand the features offered by the RTOS and the corresponding memory footprint requirement.

Language support. Another criterion to consider is the programming language supported by the RTOS.

System call/API richness. This criterion determines how comprehensive an RTOS's APIs are compared to the other RTOSs. We count the total number of system calls for each RTOS.

Operating-system-awareness debugging support. This criterion determines whether the RTOS is supported by an integrated development environment (IDE). Operating-system-awareness debugging will ease the development work because users can employ the RTOS internal information (for example, task states, system states, semaphores, and event flags) provided by the IDE.[3]

License type. This criterion determines how the RTOS is distributed: free or fee-based and for purposes such as educational or commercial.

Documentation. This criterion focuses on the type of documentation available for the RTOS (detailed APIs, simple tutorial, book, or specification).

Comparison results

Table 2 compares the RTOSs. We identified several significant similarities and idiosyncrasies among them:

- Most RTOSs use priority-based preemptive scheduling. Only two—Salvo and TinyOS—use cooperative scheduling.
- Most RTOSs support C language, which is the popular choice for embedded system programming, especially in small system design.[21]
- Only a few RTOSs have operating-system-awareness support in an IDE: μC-OS/II and EmbOS have plug-in modules for the IAR compiler; KeilOS is supported by the Keil compiler; and μITRON and μTKernel are supported by the Renesas High-Performance Embedded Workshop (HEW) compiler.
- The eCOS RTOS requires a bootloader (known as Redboot) of at least 64-Kbytes ROM.[15] Redboot boots and loads programs into the RAM via a user terminal (typically over a serial port). Hence, eCOS requires much more ROM and RAM space.
- Some RTOSs (KeilOS, PortOS, and XMK) don't make available details of their APIs (marked "N/A" in the "System call/API richness" column). SharcOS is based on μC-OS/II, so uses the same APIs. We therefore don't consider these RTOSs when comparing APIs.

Figure 3 compares the number of system APIs available for each RTOS. We categorize them (with examples) according to the following functions:

- *System management:* initialize operating system, start/shut down operating system, lock/unlock CPU

Table 2. Basic features comparison of RTOSs for small microcontrollers.

RTOS	Design objective	Scheduling scheme	License type	Documentation	System call/API richness	Language supported	Operating-system-awareness support in IDE
μITRON	Commercial	Priority-based preemptive	Fee-based	Open specification and user manual	93	C	Renesas IDE
μTKernel	Commercial/ educational/ research	Priority-based preemptive	Free for educational and commercial	Open specification and user manual	81	C	Renesas IDE
μC-OS/II	Commercial/ educational/ research	Priority-based preemptive	Free for educational	Book	42	C	IAR
EmbOS	Commercial	Priority-based preemptive	Fee-based	Online document	56	C	IAR
Free-RTOS	Hobby	Priority-based preemptive	Free for educational and commercial	Online document	27	C	None
Salvo	Commercial	Cooperative	Fee-based	Online document	31	C	None
TinyOS	Educational/ research	Cooperative	Free for educational and commercial	Tutorials	N/A	nesC	None
SharcOS	Commercial	Priority-based preemptive	Fee-based	User manual	N/A	C	None
eXtreme Minimal Kernel (XMK)	Educational/ research	Priority-based preemptive	Free for educational and commercial	Online document (incomplete)	N/A	C	None
Echidna	Educational/ research	Priority-based preemptive	Free for educational and commercial	Online document (incomplete)	18	C	None
eCOS	Commercial/ educational/ research	Priority-based preemptive	Free for educational and commercial	Book and online document	N/A	C	None
Erika	Educational/ research	Priority-based preemptive	Free for educational and commercial	Online document	19	C	None
Hartik	Educational/ research	Priority-based preemptive	Free for educational and commercial	Online document	33	C	None
KeilOS	Commercial	Priority-based preemptive	Fee-based	Online document	N/A	C	Keil IDE
PortOS	Research	Priority-based preemptive	Fee-based	Online document	N/A	C	None

- *Interrupt management:* entry/exit function, begin/end critical section
- *Task management:* create task, delete task, start task, and terminate task
- *Task-dependent synchronization:* sleep task, wake up task, and resume task
- *Communication and synchronization:* semaphore, data queue, event flag, mailbox, mutex, and message buffer
- *Memory management:* fixed-size and variable-size memory pool

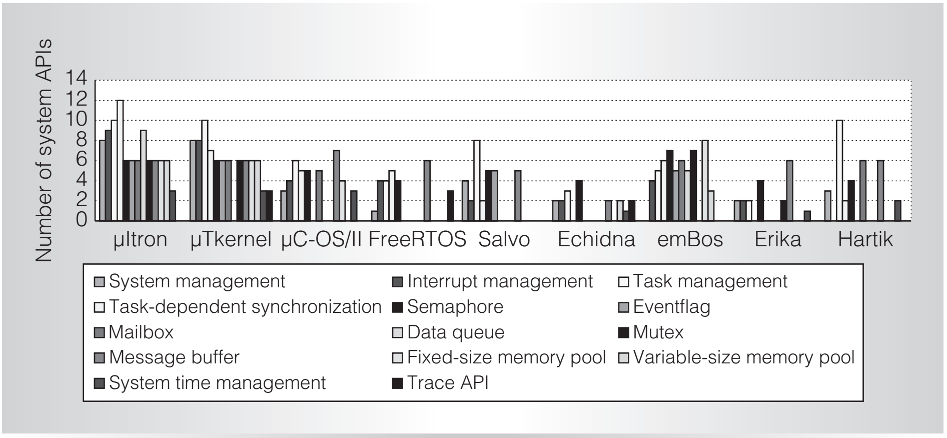

Figure 3. Number of system APIs for various RTOSs.

- *Time management:* get system operating time, operating system timer
- *Trace API:* hook the routine into certain RTOS functions such as a scheduler

Several of the RTOSs compared—μITRON, μTKernel, μC-OS/II, and EmbOS—support comprehensive APIs for these categories. Most commercial RTOSs are well-implemented, with μITRON supporting all the categories except for trace functions. EmbOS also supports most of the categories, except for trace, system time management, system management, and message buffer. For each category, the number of APIs for these four RTOSs also exceeds those for other RTOSs because they've been developed and improved and have been on the market longer.

Most open source RTOSs (such as FreeRTOS, Echidna, Erika, and Hartik) have minimal implementation. These RTOSs are more suitable for small system development. However, μC-OS/II stands out as having more available APIs. This system was originally an open source RTOS for personal and educational purposes. Its stability and popularity led to its commercialization and wide use.[4] Next to μC-OS/II, μTKernel has the most available APIs. This system supports all the function categories but data queue. It has almost the same number of APIs as the commercial RTOS μITRON, because it's backed by the T-Engine Forum (http://www.t-engine.org), led by Ken Sakamura, the μITRON architecture designer.

For these reasons, we focus on μITRON, μTKernel, μC-OS/II, and EmbOS for our subsequent comparison and benchmarking.

In addition to the functions mentioned in Figure 3, some RTOSs support the following:

- Some system APIs support *timeout*. For example, a task can wait for a semaphore for a maximum number of *n* milliseconds. RTOSs such as μITRON and μTKernel have mechanisms that let users specify the timeout in absolute values. Other RTOSs, such as μC-OS/II, FreeRTOS, Salvo, and EmbOS, only let users specify timeout values in terms of clock ticks.
- *Debug APIs* let a user's application retrieve information managed by the kernel. Currently, only μTKernel supports these APIs (for example, get task register and set task register).
- A *cyclic handler* instructs the RTOS to execute a function at a periodic interval, and an *alarm handler* lets the RTOS execute a function after a certain amount of time. These APIs are currently available in μITRON and μTKernel.
- A *rendezvous* mechanism allows synchronization and communication between tasks (similar to Ada language[22] for real-time). Both μITRON and μTKernel support this mechanism.

For μITRON and μTKernel, several APIs provide better controllability and flexibility.

Table 3. User-controllable parameters for RTOS semaphore creation.

Parameters	μC-OS/II	EmbOS	μTKernel	μITRON
Name/information	Not supported	Not supported	Extended information	Semaphore name
Attributes	Not supported	Not supported	Tasks can be queued in first-in, first-out (FIFO) or priority order Either the first task in the queue or the task with fewer requests has precedence	Tasks can be queued in FIFO or priority order
Initial count	Initial semaphore count	Initial semaphore count	Initial semaphore count	Initial semaphore count
Maximum count	Not supported	Not supported	Not supported	Maximum value of semaphore count

As Table 3 shows, tasks in μC-OS/II and EmbOS are queued in a first-in, first-out (FIFO) buffer when waiting for a semaphore, and developers may not change the order. However, in μITRON and μTKernel, developers can specify whether tasks are queued in FIFO or priority order. This flexibility not only applies to semaphores, but also to other APIs (including mailbox, message queue, memory pool, and event flag). To achieve such features in μITRON and μTKernel, developers must make trade-offs in both memory footprint and performance.

Performance and memory footprint benchmarking

We ran benchmarks for the four RTOSs identified earlier: μITRON, μTKernel, μC-OS/II and EmbOS. For execution time measurements, we used oscilloscopes and logic analyzers in combination with I/O port toggling to achieve the best accuracy (in terms of microseconds).

Ports of the RTOSs on the same M16C/62P platform

For our evaluations, we used the Renesas M16C/62P microcontroller platform, which has the following characteristics:

- Operating frequency of 24 MHz
- 512 Kbytes of ROM
- 31 Kbytes of RAM (no cache or memory management unit)
- Seven-level interrupt mask
- Renesas HEW version 4.03.00.001 IDE
- NC30 toolchain version 5.43.00

The M16C/62P has a 16-bit complex instruction set computer (CISC) architecture CPU with a total of 91 instructions available. Most instructions complete within two or three clock cycles. The MCU has a four-stage instruction queue buffer, which is similar to the simplified pipeline common in larger 32-bit processors.

To ensure that all the RTOSs operate on the same platform with the same timer resolution, we used the following settings. First, we took the OS tick resolution for all the RTOSs from timer A0[23] of the microcontroller. We used the NC30 compiler's default settings for compiling the workspaces.

We took the μC-OS/II original workspace and full source code from the Micrium website (http://www.micrium.com/renesas/index.html). We configured timer A0 for the operating system, and compiled the entire workspace again using NC30 toolchain version 5.43.00.

We took the μTKernel original workspace and full source code from the Renesas website.[24] We configured timer A0 at 10 ms.

We took the EmbOS original workspace and library files (.lib) from the Segger website (http://www.segger.com). We configured timer A0 for the operating system and configured the entire workspace (except the .lib files) with NC30 toolchain version 5.43.00. We're unaware of whether the .lib files were compiled in an older toolchain or with different optimization settings. This means that the toolchain and compiler settings for EmbOS might differ from those for μC-OS/II or μTKernel.

Table 4. System service call implementation and critical section implementation.

Call	μC-OS/II	μTKernel	EmbOS	μITRON
Service call	Direct call	Using software interrupt	Direct call	Using software interrupt
Critical section	Disable all interrupts	Raise interrupt mask level to level 4	Don't disable any interrupt (based on an internal variable)	Raise interrupt mask level to level 4

We generated the μITRON workspace, library files (.lib), and timer A0 configuration from the Renesas configuration tool for μITRON. We compiled the entire workspace (except for the .lib files) using NC30 toolchain version 5.43.00. This is similar to EmbOS because the operating system is distributed in the form of .lib files. Furthermore, the toolchain and compiler settings used when generating the .lib files might differ from those for μC-OS/II or μTKernel.

In addition, we used the same amount of stack memory per task for all RTOSs. Finally, if a particular RTOS feature isn't used (for example, a semaphore or message queue), it's disabled by the C preprocessor for μC-OS/II and μTKernel, or during linking for EmbOS. However, for μITRON, we included the unused features because the RTOS is provided in library format and system calls are invoked via software interrupts.

Table 4 shows the differences in the implementation of the RTOSs' system service calls and critical sections.

Whenever μC-OS/II and EmbOS issue a system service call, they call the function directly from the user task. The advantage is that the function is executed immediately with minimal overhead time; the disadvantage is that the service call will use the current task's stack for execution. When μITRON and μTKernel issue a system service call, it raises a nonmaskable software interrupt (an INT instruction in M16C/62P). Hence, the current execution context is to switch to the kernel space (that is, a separate stack) to execute the service call. The advantage of using this approach is that the service call won't use the current task's stack for execution; the disadvantage is that every system service call will incur a time overhead.

Table 4 also shows different methods for implementing a critical section. To start a critical section, μC-OS/II disables all interrupts, so no external interrupt can be accepted. This lets the critical section execute safely from start to finish without intervention, but it also implies that the system won't accept a highly important real-time interrupt during this period. The μITRON and μTKernel RTOSs implement a critical section by raising the interrupt mask level (for M16C/62P the interrupt mask level is set to 4, but can be changed) so that a critical interrupt can be accepted as long as the interrupt handler doesn't interfere with the RTOS internal variables. EmbOS doesn't disable or raise the interrupt mask level for a critical section. It allows all interrupts to come in but uses internal variables to control the critical section. Thus, any interrupt can be accepted and handled during the critical section; however, EmbOS requires additional code to handle the critical section's internal variables.

Benchmarking criteria

We aimed to make our benchmarking criteria easy to port to different platforms. For each criterion, we collected execution time measurement and memory footprint (ROM and RAM).

Task switch time. Task switch time is the time it takes for the RTOS to transfer the current execution context from one task to another. Figure 4a shows the measurement method.

It includes two tasks, task 1 and task 2, with task 1 having higher priority. Each RTOS must first execute task 1, which then goes into a sleep or inactive state. The execution context then switches to task 2. Task 2 wakes up or activates task 1. Right after waking up, the execution context switches back to task 1 because it has higher priority. Different RTOSs use different

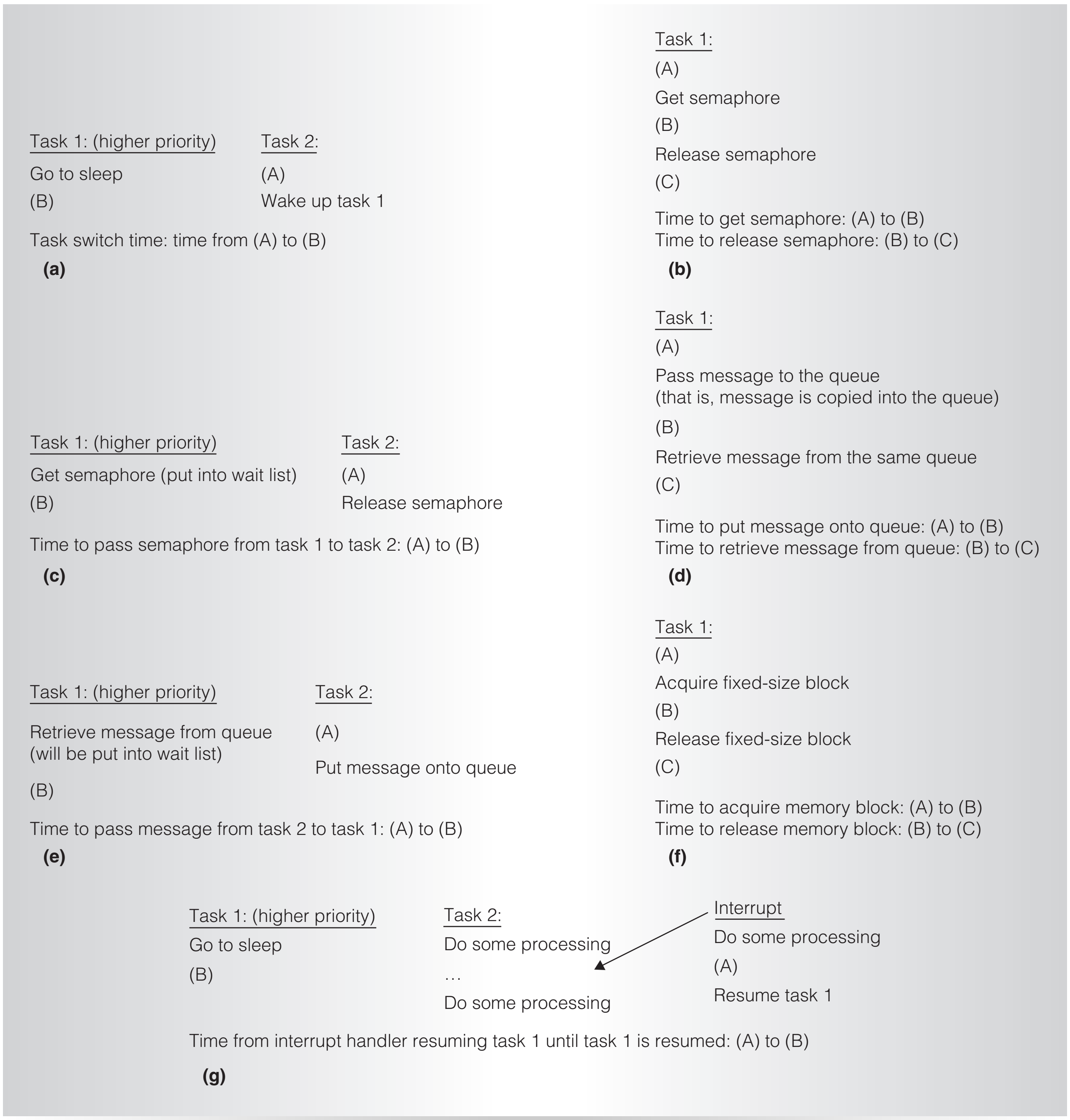
Task 1: (higher priority)
Task 2:
Go to sleep
(A)
(B)
Wake up task 1
Task switch time: time from (A) to (B)
(a)
Task 1:
(A)
Get semaphore
(B)
Release semaphore
(C)
Time to get semaphore: (A) to (B)
Time to release semaphore: (B) to (C)
(b)
Task 1: (higher priority)
Task 2:
Get semaphore (put into wait list)
(A)
(B)
Release semaphore
Time to pass semaphore from task 1 to task 2: (A) to (B)
(c)
Task 1:
(A)
Pass message to the queue
(that is, message is copied into the queue)
(B)
Retrieve message from the same queue
(C)
Time to put message onto queue: (A) to (B)
Time to retrieve message from queue: (B) to (C)
(d)
Task 1: (higher priority)
Task 2:
Retrieve message from queue
(will be put into wait list)
(A)
Put message onto queue
(B)
Time to pass message from task 2 to task 1: (A) to (B)
(e)
Task 1:
(A)
Acquire fixed-size block
(B)
Release fixed-size block
(C)
Time to acquire memory block: (A) to (B)
Time to release memory block: (B) to (C)
(f)
Task 1: (higher priority)
Task 2:
Interrupt
Go to sleep
Do some processing
Do some processing
(B)
…
(A)
Do some processing
Resume task 1
Time from interrupt handler resuming task 1 until task 1 is resumed: (A) to (B)
(g)

Figure 4. Benchmarking criteria measurements: task switch time (a), get/release semaphore time (b), semaphore-passing time (c), pass/retrieve message time (d), message-passing time (e), acquire/release fixed-size memory time (f), and task activation from interrupt handler time (g).

terms to describe sleep/inactive and ready/active states (for example, μC-OS/II and EmbOS use suspend/resume, and μITRON and μTKernel use sleep/wakeup).

Table 5 shows the system calls used in each RTOS.

Get/release semaphore time. RTOSs often use semaphores for synchronizing primitives.[25] For semaphore benchmarking, we measure the time taken by the get and release semaphore service calls as well as the time required to pass the semaphore from one

Table 5. APIs used for the task switch time benchmark.

API	μC-OS/II	μTKernel	EmbOS	μITRON
Pass message	`OSTaskSuspend()`	`tk_slp_tsk()`	`OS_Suspend()`	`slp_tsk()`
Retrieve message	`OSTaskResume()`	`tk_wup_tsk()`	`OS_Resume()`	`wup_tsk()`

Table 6. APIs used for the get/release semaphore benchmark.

API	μC-OS/II	μTKernel	EmbOS	μITRON
Get semaphore	`OSSemPend()`	`tk_wai_sem()`	`OS_WaitCSema()`	`wai_sem()`
Release semaphore	`OSSemPost()`	`tk_sig_sem()`	`OS_SignalCSema()`	`signal_sem()`

Table 7. APIs used for the message-passing benchmark.

API	μC-OS/II	μTKernel	EmbOS	μITRON
Pass message API	`OSQPost()`	`tk_snd_mbx()`	`OS_Q_Put()`	`snd_mbx()`
Retrieve message API	`OSQPend()`	`tk_rcv_mbx()`	`OS_Q_GetPtr()`	`rcv_mbx()`

task to another. Figure 4b shows how we measure the get and release semaphore time.

Here there is only one task (task 1) and one binary semaphore (initialized to 1). Task 1 gets and then releases the semaphore. Different RTOSs use different terms to describe this process. Table 6 shows the APIs used by each RTOS.

Semaphore-passing time. To measure the performance of semaphore passing, we use the following method, as Figure 4c illustrates.

This measurement involves two tasks, task 1 and task 2, with task 1 having higher priority, and a binary semaphore (initialized to 0). The system executes task 1, which tries to get the semaphore. Because the semaphore's value is 0, task 1 enters a sleep/inactive state, waiting for the semaphore's release. The current execution context then switches to task 2, which releases the semaphore. Once released, the semaphore wakes task 1 and the execution context switches to task 1.

Pass/receive message time. In addition to semaphores, message passing has become increasingly popular for synchronization.[26] This measurement uses a message-passing mechanism based on memory pointer passing. It doesn't copy the message into an internal RTOS area because not all RTOSs support this approach. Figure 4d shows how we perform this measurement.

The measurement involves only one task (task 1). The task first passes the message pointer (usually to an internal message queue), and then retrieves the same message pointer. Table 7 shows the APIs used in each RTOS.

Intertask message-passing time. Figure 4e shows how we measure the message-passing time between tasks.

This measurement involves two tasks, task 1 and task 2, with task 1 having higher priority. Task 1 is executed first, and it tries to retrieve a message pointer from the queue. Because no message is yet available, task 1 enters a sleep/inactive state and waits for a new message. The current execution context switches to task 2, which puts a new message into the queue. The new message wakes task 1, and the execution context switches to task 1. The difference between this measurement and the pass/receive message time benchmark is that this method includes the RTOS's overhead time to process the message queue and wake the receiving task.

Fixed-size memory acquire/release time. In an RTOS, only fixed-size dynamic memory

Table 8. APIs used for the fixed-size memory benchmark.

API	μC-OS/II	μTKernel	EmbOS	μITRON
Acquire fixed-size block API	`OSMemGet()`	`tk_get_mpf()`	`OS_MEMF_ Alloc()`	`get_mpf()`
Release fixed-size block API	`OSMemPut()`	`tk_rel_mpf()`	`OS_MEMF_ Release()`	`rel_mpf()`

Table 9. APIs for the task activation from within interrupt handler benchmark.

API	μC-OS/II	μTKernel	EmbOS	μITRON
Go to sleep API	`OSTaskSuspend()`	`tk_slp_tsk()`	`OS_Suspend()`	`slp_tsk()`
Resume from interrupt API	`OSTaskResume()`	`tk_wup_tsk()`	`OS_Resume()`	`iwup_tsk()`

allocation should be used. The allocation and deallocation time must be deterministic. Figure 4f illustrates how we measure the time to acquire and to release a fixed-size memory block.

This measurement involves only one task. Task 1 acquires a fixed-size memory block (128 bytes) and then releases it. Table 8 shows the APIs used in this process.

Task activation from within interrupt handler time. An RTOS must deal with external interrupts that might be asserted at any time. It will typically keep an interrupt handler's execution as short as possible to avoid affecting the system response. If the interrupt requires long processing, the handler can activate another task to do the necessary processing. The time from when the interrupt handler resumes the task until the time when the task is executed is crucial to the system's design.

Figure 4g shows the measurement's setup, which involves two tasks, task 1 and task 2, with task 1 having higher priority, and an external interrupt with a proper handler. Task 1 executes first. It goes to the sleep/inactive state, and the execution context switches to task 2. Task 2 runs continuously. When an external interrupt occurs, the interrupt handler executes and resumes task 1. The execution context switches over to task 1. Table 9 shows the APIs used for each RTOS.

Benchmarking results

For each criterion, we compiled the benchmarking code and obtained the ROM and RAM usage from the toolchain report. By averaging the ROM information across all test criteria, we got the average ROM size. Figure 5a shows the code sizes for the four RTOSs when running the seven benchmarks.

As the figure shows, μTKernel has a larger code size. This is due to its APIs' flexibility and comprehensive support. The commercial RTOSs, μITRON and EmbOS, offer a relatively compact code size. Nevertheless, all four RTOSs fit well into microcontrollers of limited ROM sizes.

Figure 5b shows the RAM information for the four RTOSs. Two of the systems—μTKernel and μITRON—have relatively lower RAM usage, while μC-OS/II and EmbOS have slightly higher usage. According to each benchmark's requirement, we set the number of tasks, stack size, and number of RTOS objects (for example, semaphore and event flags) to be the same for all RTOSs. The amount of RAM differences among the RTOSs range from 7 to 10 bytes, which might be due to internal implementations or to the API design approach. In summary, the ROM and RAM usage of all these RTOSs are well suited for small microcontrollers. However, μITRON has the optimal usage of both ROM and RAM.

Execution time

Figure 6 shows the execution time measurement for the RTOSs for the different benchmark criteria. Because timer interrupt is the only variation in the system (for operating system tick), we executed each benchmark at least twice to ensure consistent results. Nevertheless, for all benchmarks, running once is enough to yield the correct measurement. For the task activation from within interrupt handler benchmark, the external interrupt is an additional variation. When measuring this benchmark, the external interrupt might or might not be asserted during the operating system's critical section (during which the operating system disables interrupts). If it's asserted during the critical section, the operating system's response time will be slightly longer. Hence, this measurement might not include the worst-case scenario.

As Figure 6 shows, μTKernel has the lowest task-switching time, followed by μITRON, μC-OS/II, and EmbOS. It also shows that the μC-OS/II semaphore acquire and release times are the fastest. The fastest intertask semaphore passing is achieved by μITRON, whereas μC-OS/II and μTKernel have better message passing and retrieval times than μITRON and EmbOS. As far as fixed-size memory is concerned, μC-OS/II has the best execution time, followed by EmbOS, μITRON, and μTKernel. Finally, μTKernel has the best performance time for task activation from interrupt handler, followed by μC-OS/II, μITRON, and EmbOS.

As these benchmarking results show, each RTOS has its own strengths and weaknesses. In the open source category, μC-OS/II is useful as a small and compact ROM-size RTOS. However, for more comprehensive API support, we recommend μTKernel (at the expense of a slightly higher ROM footprint). On the other hand, if the developer prefers a commercial RTOS, we recommend either μITRON or EmbOS, with μITRON having a slighter lower RAM footprint.

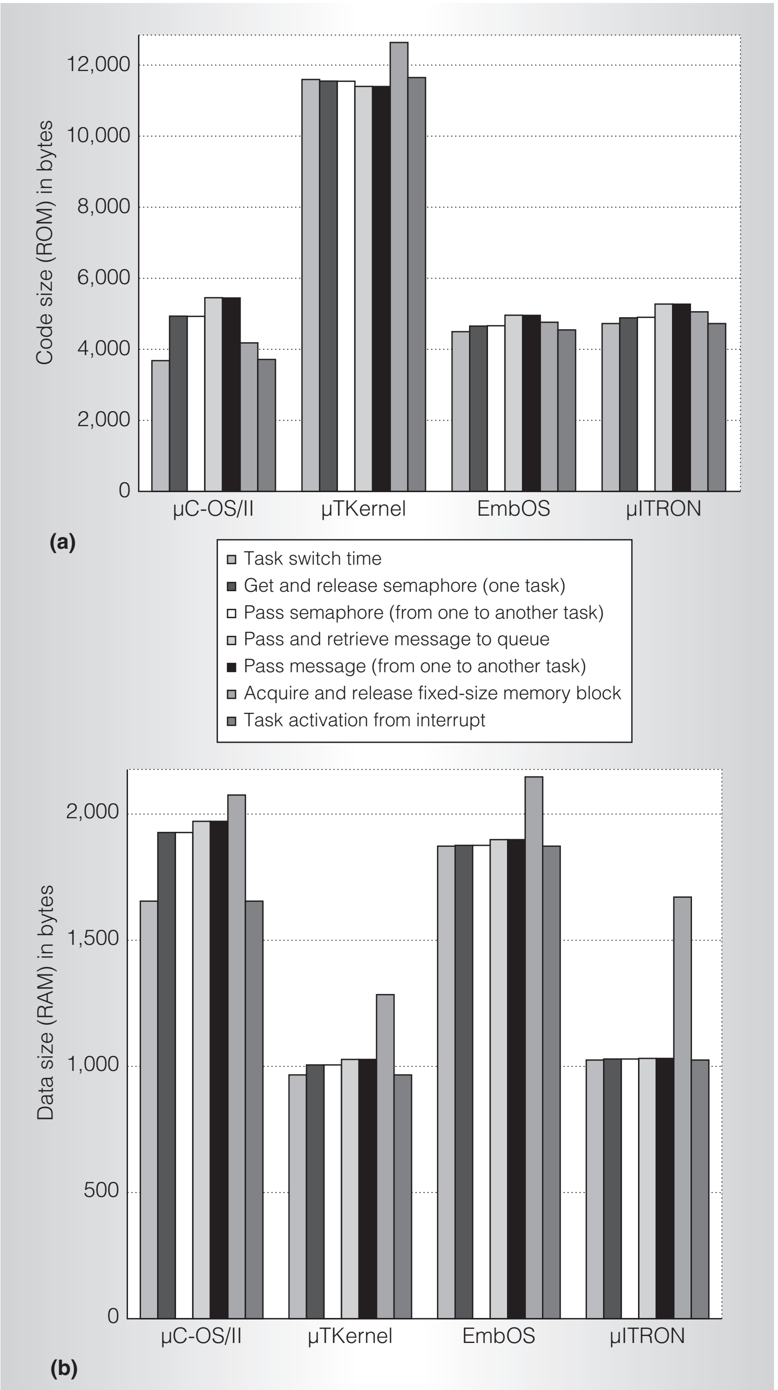

Figure 5. Benchmarking results for the four RTOSs: comparisons of code size (a) and data size (b).

Our benchmarking criteria are simple, easy to port to different platforms, and representative of typical RTOS uses. They show that each RTOS has different strengths and weaknesses, but there's no clear winner. With these detailed performance benchmarks, potential adopters of these RTOSs can simplify their selection by examining their specific application requirements. One of our future plans is to investigate the effectiveness of using RTOS for applications development. It would also be interesting to evaluate the power utilization between different RTOSs and develop

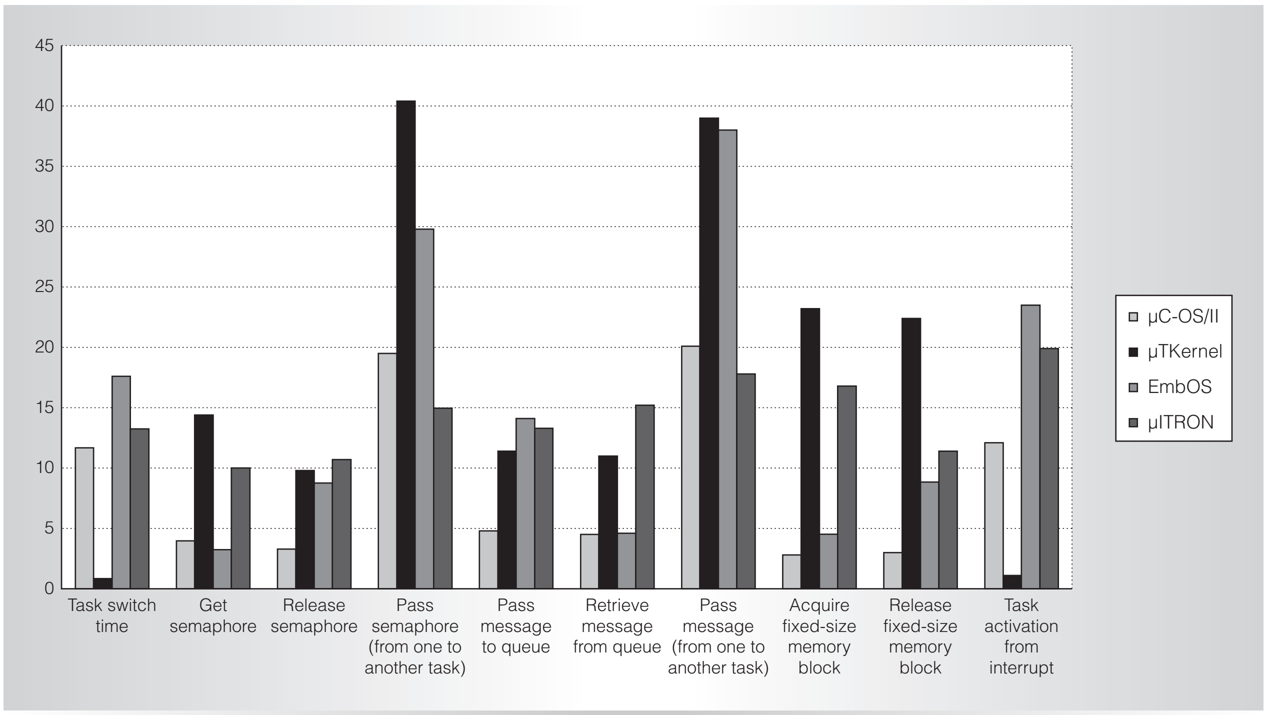

Figure 6. Execution time benchmark for four RTOSs.

a common advanced configuration and power interface (ACPI) framework to ensure efficient power usage. MICRO

References

1. D. Kalinsky, ''Basic Concepts of Real-Time Operating Systems,'' *Linux Devices,* Nov. 2003; http://www.jmargolin.com/uavs/jm_rpv2_npl_16.pdf.
2. K. Sakamura and H. Takada, ''μITRON for Small-Scale Embedded Systems,'' *IEEE Micro,* vol. 15, no. 6, Nov./Dec. 1995, pp. 46-54.
3. J. Ganssle, ''The Challenges of Real-Time Programming,'' *Embedded System Programming,* vol. 11, July 1997, pp. 20-26.
4. R. Nass, ''Annual Study Uncovers the Embedded Market,'' *Embedded Systems Design,* 2 Sept. 2007; http://www.embedded.com/design/opensource/201803499;jsessionid=NGMOMOIGE5ZNNQE1GHOSKHWATMY32JVN?printable=true.
5. K. Baynes et al., ''The Performance and Energy Consumption of Embedded Real-time Operating Systems,'' *IEEE Trans. Computers,* vol. 52, no. 11, 2003, pp. 1454-1469.
6. J.J. Labrosse, *MicroC/OS-II: The Real-Time Kernel,* R&D Books, 1999.
7. T-Engine Forum, ''μTKernel specification, 1.00.00,'' Mar. 2007; http://www.t-engine.org.
8. R. Barry, ''A Portable, Open Source Mini Real-Time Kernel,'' Oct. 2007; http://www.freertos.org.
9. K. Curtis, ''Doing Embedded Multitasking with Small Microcontrollers, Part 2,'' *Embedded System Design,* Dec. 2006; http://www.embedded.com/columns/technicalinsights/196701565?_requestid=242226.
10. A. Garcia-Martinez, J. F. Conde, and A. Vina, ''A Comprehensive Approach in Performance Evaluation for Modern Real-Time Operating Systems,'' *Proc. 22nd EuroMicro Conf.,* IEEE CS Press, 1996, p. 61.
11. R.P. Kar and K. Porter, ''Rhealstone: A Real-Time Benchmarking Proposal,'' *Dr. Dobb's J. of Software Tools,* vol. 14, no. 2, Feb. 1989, pp. 14-22.
12. K.M. Sacha, ''Measuring the Real-Time Operating System Performance,'' *Proc. 7th EuroMicro Workshop Real-Time Systems,* IEEE CS Press, 1995, pp. 34-40.
13. K. Sakamura and H. Takada, *μITRON 4.0 Specifications,* TRON Assoc., 2002; http://www.ertl.jp/ITRON/SPEC/FILE/mitron-400e.pdf.
14. *EmbOS Real-Time Operating System, User & Reference Guide,* Segger Microcontroller,

2008; http://www.segger.com/cms/admin/uploads/productDocs/embOS_Generic.pdf.

15. A.J. Massa, *Embedded Software Development with eCos,* Prentice Hall, 2002.
16. P. Gai et al., *E.R.I.K.A.: Embedded Real-tIme Kernel Architecture; ERIKA Educational User Manual,* Realtime System (RETIS) Lab, Scuola Superiore Sant'Anna, Italy, 2004; http://erika.sssup.it/download.shtml#Doc.
17. G.C. Buttazzo, ''Hartik: A Hard Real-Time Kernel for Programming Robot Tasks with Explicit Time Constraints and Guaranteed Execution,'' *Proc. IEEE Int'l Conf. Robotics and Automation,* IEEE Press, 1993, pp. 404-409.
18. R. Chrabieh, ''Operating System with Priority Functions and Priority Objects,'' *TechOnline,* Feb. 2005; http://www.techonline.com/learning/techpaper/193101942.
19. G. Hawley, ''Selecting a Real-Time Operating System,'' *Embedded System Design,* vol. 12, no. 3, 1999, http://www.embedded.com/1999/9903.
20. M. Timmerman and L. Perneel, ''Understanding RTOS Technology and Markets,'' Dedicated Systems RTOS Evaluation project report, 2005; http://www.dedicated-systems.com/vpr/layout/display/pr.asp?PRID=8972.
21. R. Bannatyne and G. Viot, ''Introduction to Microcontrollers, Part 2,'' *Northcon Conf. Proc.,* IEEE Press, 1998, pp. 250-254.
22. B. Millard, D. Miller, and C. Wu, ''Support for ADA Intertask Communication in a Message-Based Distributed Operating System,'' *Computers and Comm. Conf. Proc.,* IEEE Press, 1991, pp. 219-225.
23. *M16c/62P Group Hardware Manual,* Renesas Technology, 2006; http://documentation.renesas.com/eng/products/mpumcu/rej09b0185_16c62pthm.pdf.
24. Renesas Technology, ''μTKernel for M16C Source Code and Documentation,'' 2007; http://www.superh-tkernel.org/eng/download/misc/software/M30626FJPGP_micro_tkernel/software/index.html.
25. I. Ripoll et al., ''RTOS State of the Art Analysis,'' tech. report, Open Components for Embedded Real-time Applications (OCERA) project, 2002.
26. D. Kalinsky, ''Asynchronous Direct Message Passing Rapidly Gains Popularity,'' *Embedded Control Europe,* Nov. 2004, p. 32.

Tran Nguyen Bao Anh is a senior embedded engineer at STS Wireless Sound Solutions Singapore. His research interests include embedded and real-time system development. Anh has an MSc in embedded systems from Nanyang Technological University.

Su-Lim Tan is an assistant professor in the School of Computer Engineering at Nanyang Technological University, Singapore. His research interests include embedded network sensing and smart sensors. Tan has a PhD in engineering from the University of Warwick.

Direct questions and comments about this article to Su-Lim Tan at the School of Computer Engineering, Nanyang Technological University, Blk N4-02a-32, Nanyang Avenue, Singapore 639798; assltan@ntu.edu.sg.