Survey and performance evaluation of real-time operating systems (RTOS) for small microcontrollers

Tran Nguyen Bao Anh*†, Su-Lim Tan†

**Renesas Technology Singapore, Singapore Engineering Centre, Singapore 098632*
†School of Computer Engineering, Nanyang Technological University, Singapore 639708

***Abstract*—** RTOS has gained popularity over the years in microcontroller/processor-based embedded system design. In this paper, we will discuss the important differences between RTOS and generic OS, the advantages and disadvantages of using RTOS for small microcontroller system development, and the benchmarking methods used for RTOS. Several RTOSes are studied and compared based upon numerous selection criteria, and four RTOSes are selected for performance benchmarking on the same microcontroller platform. For the purpose of performance benchmarking, a list of benchmarking criteria which is aimed to be simple and representative of typical RTOS usages are examined. The benchmarking results show that there is no clear winner and each RTOS performed well on certain criteria compared to others.

Index Terms- ***kernel; operating system; real-time system; RTOS; RTOS benchmarking***

I. INTRODUCTION

REAL – time embedded systems are typically designed for various purposes such as to control or to process data. Characteristics of real-time system include meeting certain deadlines at the right time. To achieve this purpose, real-time operating systems (RTOS) are often used. An RTOS is a piece of software with a set of APIs for users to develop applications. Using an RTOS does not guarantee that the system will always meet the deadlines, as it also depend on how the overall system is designed and structured.

While RTOS for embedded systems were predominantly employed in high-end microprocessors or microcontrollers with 32-bit central processing unit (CPU), there is a growing trend to provide these features in the mid-range (16-bit and 8-bit) processor systems. In Section *B*, the use of RTOS for this range of devices will be discussed in details.

An operating system (OS) is a piece of software that manages the sharing of resources in a computer system. RTOS is often differentiated from generic OS as it is specifically designed for scheduling to achieve real-time responses.

A. Generic OS versus RTOS

RTOSes are typically differentiated from generic OSes in the following:

- **Preemptive, priority-based scheduling**: Scheduling scheme refers to how the RTOS assigns CPU cycles to tasks for execution. Scheduling scheme is important in any OS as it affects how the various softwares are executed. Most generic OSes are time-sharing systems in which tasks are allocated the same amount of time slices (e.g. round robin scheduling) for execution. In RTOS, tasks are often assigned priorities and higher-priority tasks can preempt lower-priority tasks during execution (preemptive scheduling). There are also RTOSes that adopt cooperative scheduling. Such scheduling technique usually implies that the running task has to explicitly invoke the scheduler to perform a task switch.
- **Predictability in task synchronization**: For generic OS, task synchronization is unpredictable because the OS can directly or indirectly introduce delays into the application software. In RTOS, synchronization among tasks (such as using semaphore, mailbox, message queue, event flag, etc) must be time-predictable. The system services must have known and expected duration of execution time.
- **Deterministic behaviors**: This can be considered as the key difference between generic OS and RTOS. In RTOS, task dispatch time, task switch latency, interrupt latency must be time-predictable and be consistent even when the number of tasks increases. On the other hand, generic OS (mainly due to its time-sharing approach) reduces the overall system responsiveness and does not guarantee service call execution within certain amount of time when the number of tasks increases. Dynamic memory allocation (*malloc()* in C language), though being widely supported in generic OS, is not recommended in RTOS because the behavior is unpredictable [1]. Instead, fixed-sized memory allocation is provided in which only fixed-size block of memory is allocated for every request.

B. RTOS for small scale-embedded systems

There are a number of variants of RTOSes available nowadays; they ranged from commercial, proprietary, to open-source RTOSes. For small-scaled embedded systems designed using small microcontrollers (i.e. microcontrollers with maximum ROM of 128Kbytes and maximum RAM of 4Kbytes [2]), there is a common perception that no RTOS is

Digital Object Indentifier 10.1109/MM.2009.56

needed. However, there are significant advantages to use an RTOS for this range of devices [2, 3], such as:

- **Optimizing software development**: Even in system development using small microcontrollers, improving software productivity is a critical issue due to time-to-market pressure as well as shorten development cycle [4]. Using an RTOS is one of the approaches that has gained increasing popularity. As the code complexity grows, an RTOS is an efficient tool to manage the software, and to distribute the tasks among developers. Using an RTOS will allow the entire software to be partitioned into modular tasks that can be taken care of by individual programmer. Moreover, low-level driver development can be done by other developers.
- **Better and safer synchronization**: In small embedded system development without using any RTOS, global variables are often used for synchronization and communication among modules/functions. However, especially in highly interrupt-driven system, using global variables lead to bugs and software safety issues [5]. These global variables are often shared and accessed among the functions; hence there are high chances of them being corrupted at any time during the program execution. As the code begins to grow, these bugs become hidden deeper and more difficult to be uncovered. Consequently, development time can be lengthened even for such small-scale system development. With an RTOS in place, synchronization is safely managed and tasks can pass messages or synchronize with each other without any corruption problems.
- **Resource management**: Most RTOSes provide APIs for developers to manage the system resources [5]. These include task management, memory pool management, time management, interrupt management, communication and synchronization methods. These features provide the abstraction layer for developers to freely structure the software, to achieve cleaner code and to even quickly port across different hardware platforms with little code modifications. Especially with small system development - cost of hardware is of critical constraint, and development time is usually short.
- **Timing can be easily managed by RTOS**: With time management functions, software designers can achieve task delay, timer handling or time-triggered processing without resorting to understanding the underlying hardware mechanisms. As compared with a small system that does not use any RTOS, achieving timing related features can be tricky as the software designer needs to understand the underlying peripherals (such as timers), how to use it, and how to link it with the top-level application code. Any modification, such as to lengthen the delay time would requires the developer to examine the code and peripheral again to make changes appropriately. When porting the software to another platform employing a different microcontroller with a different set of peripherals, these timing features need to be rewritten again. Unless for special and critical timing issue with unique hardware peripheral, using an RTOS can helps to speed up the development time significantly to tackle these timing issues.

In [3], an example is given to illustrate the importance of RTOS in small system design – a printer system. Without an RTOS, there is one single chunk of code to manage all the activities, from paper feeding, user-input reading, to printing control. By having an RTOS, individual task will manage each of these activities. Except for passing of status information, each task does not need to know much about what other tasks are performing. Hence, having an RTOS in place can help in partitioning the software in time domain (tasks are running concurrently) and in terms of functionalities (each task performs a specific operation).

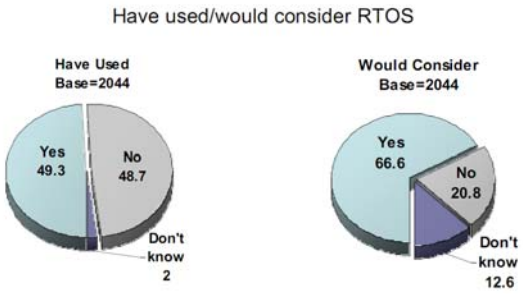

Figure 1: RTOS usage, embedded market survey (CMP, EE Times) 2004 [4]

RTOS usage is gaining popularity in the past few years as clearly indicated in the CMP, EE Times embedded system survey for 2004 - 2006 [4, 5, 6]. Figure 1 shows the embedded market survey conducted by CMP, EE Times in 2004 on the number of developers that have used and would consider using RTOS in the current and next projects. The number of developers who "have used" RTOS takes up more than 49%. This percentage rises to 80.9% in the 2005 survey, and 71% in the 2006 survey. The number of developers who "would consider" to use RTOS in the next project in 2004 is 66.6%, and in 2005 is 86%, which shows a steady trend towards employing RTOS. Definitely, more and more developers and designers are adopting RTOS in long run.

Table 1 indicates there is another trend in RTOS selection - companies are moving towards open-source RTOS - from 16% in current projects to 19% in the next projects in 2006) or commercial distribution of an open-source RTOS - from 12% in current projects to 17% in the next projects in 2006. Commercial OS and in-house OS, though currently being extensively used, are declining - from 51% in current projects to 47% in next projects in 2006 for the case of commercial OS, and from 21% to 17% for the case of in-house OS. In the latest 2007 survey [7], the percentage for commercial OS drops further to 41%. Also according to the 2007 survey, the key influencing factors in RTOS selection for commercial OS are the quality and the availability of technical support. Hence

Digital Object Indentifier 10.1109/MM.2009.56 0272-1732/$26.00 © 2009 IEEE

companies are willing to adopt commercial OS only when the technical support provided is adequate. Otherwise, companies may look for other more cost-effective choices.

Types of operating systems used	Current project	Next project
Commercial OS	51 %	47 %
Internally developed or in-house OS	21 %	17 %
Open-source OS without commercial support	16 %	19 %
Commercial distribution of an open-source OS	12 %	17 %

Table 1: Types of operating system used, embedded market survey (CMP, EE Times) 2006 [**5**]

Having detailed about the advantages and trends of using RTOS in general, there are certain disadvantages and concerns associated with using RTOS for small microcontrollers. Firstly, an RTOS takes up additional memory (ROM and RAM), computational resources and consume extra power [8], hence can the system absorb these overheads? For small microcontrollers, it is important that the RTOS must be compact in ROM and RAM requirements. There are various RTOSes available for this segment of devices, and some are flexible enough to be configured to have only those functions and APIs required by the application [2, 9, 10, 11] so that the code size can be reduced. In the latter sections, a more detailed analysis will be done on the possible ROM and RAM requirements of RTOSes. Besides memory footprint, an RTOS also takes up additional CPU resource. Most RTOSes require a periodic timer (OS 'tick') [12] to execute the scheduler and other relevant system services. RTOS services such as task synchronizations must have known execution time (e.g. how much time does 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 and structure the whole system. Hence it is essential to understand the performance measurements and the benchmarking metrics among the RTOSes.

C. RTOS benchmarking

There are different approaches towards RTOS benchmarking: based on applications or based on the most frequently used system services (fine-grained benchmarking) [13]. As there are various types of applications with each having very different requirements, benchmarking against any generic applications will not be reflective of the RTOS strengths and weaknesses.

There are various research publications related to benchmarking method based on frequently used system services. In [14], the Rhealstone benchmark is proposed with the following measurement: task switch time, preemption time, interrupt latency time, semaphore shuffling time, deadlock breaking time, and datagram throughput time. Rhealstone benchmark is not suitable for several reasons. Firstly, very few RTOSes are capable of breaking deadlock (which we will see later in the RTOSes survey). Datagram throughput time is based on message passing by copying to a memory area managed by the OS. However, not all RTOSes use the same concept for message passing. Some RTOSes pass messages by passing only the memory pointer, and hence there is no need to use the special memory area managed by the OS. This approach is also more suitable for small microcontrollers because there is no extra memory for OS internal use. Interrupt latency time as defined by Rhealstone is purely dependent on the CPU architecture and is not determined by the RTOS. Rhealstone, in general, are "somewhat adhoc", and do not cover other situations commonly found in real-time applications [13].

In [13], some metrics are proposed (based on frequently used system services): response to external event (interrupt), inter-task synchronization and resource sharing, and inter-task data transfer (message passing). Inter-task data transfer, as explained previously, is also based on data copying into a memory area managed by the OS, similar to the "datagram throughput time" in Rhealstone benchmark. In the test for "response to external event (interrupt)", the interrupt handler wakes up another task via a semaphore. Using a semaphore in this case does not seem to be the best approach. Waking up the task directly by using system service call (such as sleep/wakeup service call) instead of going through a semaphore is a better approach to reduce the overhead delay.

In [15], the metrics proposed are (based on frequently used system services): tests for measuring the duration of message transfer and the communication through a pipe, tests for measuring the speed of task synchronization through proxy and signal, and tests for measuring the duration of task switching. These metrics are based only on the QNX distributed RTOS platform, some concepts such as proxy and signal do not exist on most RTOSes (as illustrated in the RTOSes survey later).

In the next section, a list of RTOSes, including open-source, commercial and research, will be discussed based on their features and APIs. Those found to be unsuitable for small microcontrollers will be eliminated. Finally, among those selected, four of the more popular RTOSes will be ported to a MCU (microcontroller) platform for benchmarking (in terms of code size, RAM usage, and execution speed) and evaluated against a list of proposed benchmarking metrics.

II. RTOS FEATURES AND API COMPARISON

A. Criteria for comparison

The objective of this section is to investigate RTOSes available (open-source, commercial, and research) and determine those that are suitable for small microcontrollers only. Information is mainly based on documentations and APIs available on websites. These RTOSes are: µITRON, µTKernel, µC-OS/II, EmbOS, FreeRTOS, Salvo, TinyOS, SharcOS, XMK OS, Echidna, eCOS, Erika, Hartik, KeilOS and PortOS.

As described in [16], criteria used for selecting an RTOS includes the following: language support, tool compatibility, system service APIs, memory footprint (ROM and RAM usage), performance, device drivers, OS-awareness debugging

Digital Object Indentifier 10.1109/MM.2009.56 0272-1732/$26.00 © 2009 IEEE

tools, technical support, source/object code distribution, licensing scheme and company reputation. Similarly, criteria mentioned in [17] are: installation/configuration, RTOS architecture, API richness, documentation and support, and tools support. Embedded market surveys conducted by CMP, EE Times in 2005 to 2007 [5, 6, 7] also concluded that the priority of criteria for RTOS selection (see Figure 2), in which real-time capability has taken the highest weighting.

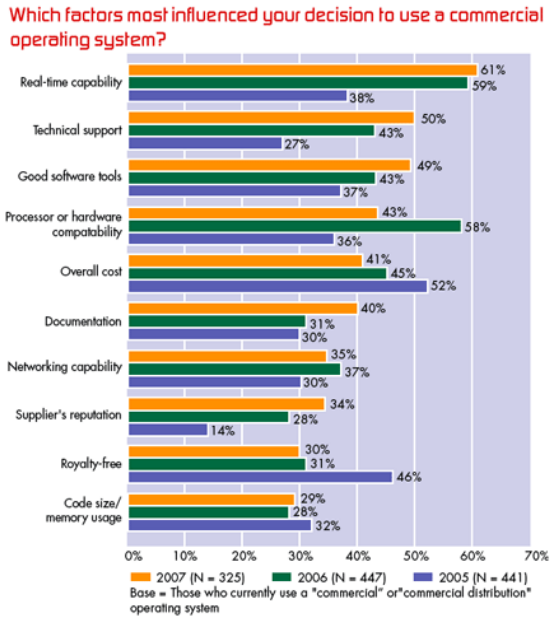

Figure 2: Influential factors in operating system selection, embedded market survey, CMP, EE Times 2005, 2006, and 2007 (N=441 is the total number of people surveyed)

Based on the above, a list of criteria to compare among the RTOSes is established. In the scope of this paper, it is not feasible to take all the above criteria into considerations. Criteria, such as "suppliers' reputation" and "company reputation" are subjective to each and individual company's judgments. "Overall cost" is project and application dependent, and "royalty fee" is normally based on quantity, even though RTOS vendors may use other business models to charge their customers (such as per application, per product model, or per MCU model, etc). "Memory footprint" (ROM and RAM usage) may not always be available and it is highly dependent upon the compiler settings as well as RTOS configurations. For this paper, the following criteria will be used for comparison:

- **Design objective**: The origins of the RTOSes being surveyed are different from one another, as some are open-source, some are personal hobby-based, and some are commercial. It is important to understand the history and the background motivation that led to the creation of each RTOS. A personal hobby-based RTOS would be less likely to be as stable compared to a popular open-source, or to a commercial RTOS.
- **Author**: Similar to design objective, it is essential to understand the author who originated the RTOS – whether it was by a person, an organization, or a company.
- **Scheduling scheme**: RTOS scheduling approach will be investigated to determine whether preemptive scheduling, cooperative scheduling or other scheduling scheme is used.
- **Real-time capability and performance**: Real-time capability is generally considered as a system characteristic to describe whether the system is able to meet the timing deadline. Using an RTOS in the system takes up CPU cycles; however the RTOS must not have indeterminist behaviors. The amount of CPU cycles and time consumed by the RTOS for any service call should be measurable and of low or acceptable values to the system designers. Real-time capability and performance information are not available for some RTOSes. Even if these information are available, they might be based on different hardware platforms. In Section III, a selected list of RTOSes will be benchmarked against one another on the same hardware platform so that more comparative results can be obtained.
- **Memory footprint**: Besides CPU cycles, an RTOS also occupies additional ROM and RAM spaces. This could lead to larger ROM and RAM sizes for the entire system. There is always a tradeoff between memory footprint and the functionalities required from the RTOS. To have more robust and reliable APIs, probably more lines of code are needed. On the other hand, basic and simple APIs will require only minimum amount of code. Hence, it is important for the designers to understand the features offered by the RTOS with the corresponding memory footprint requirement. This criterion will also be compared among the selected RTOSes in Section III.
- **Language support**: Programming language supported by the RTOS.
- **System call/API richness**: This criterion determines how comprehensive the RTOS APIs are as compared to the rest of the RTOSes. The total number of system calls for each RTOS will be counted.
- **OS-awareness debugging support**: This criterion determines if the RTOS is being supported by any of the Integrated Development Environment (IDE). OS-awareness debugging [3] will ease the development work as users can use these RTOS internal information (e.g. task states, system states, semaphores, event flags) provided by the IDE.
- **License type**: This is to investigate how the RTOS is distributed: free or fee-based for different purposes such as educational or commercial.
- **Documentation**: This criterion will focus on what type of documentations are available for the RTOS (detail APIs, simple tutorial, book or specification).

Digital Object Indentifier 10.1109/MM.2009.56

	Design objective	Author	Scheduling	License type	Documentation	System call/API richness	Language supported	OS-awareness support in IDE	References
µITRON	Commercial	Ken Sakamura & Tron association	Priority-based preemptive	Fee-based	Open specification and user manual	93	C	Supported by Renesas IDE	[2, 18]
µTKernel	Commercial/ Educational/ Research	T-Engine forum	Priority-based preemptive	Free for educational and commercial	Open specification and user manual	81	C	Supported by Renesas IDE	[10]
µC-OS/II	Commercial/ Educational/ Research	Jean Labrosse	Priority-based preemptive	Free for educational	Book by author	42	C	Supported by IAR	[9]
EmbOS	Commercial	Segger	Priority-based preemptive	Fee-based	Online document	56	C	Supported by IAR	[19]
Free-RTOS	Hobby	Richard Barry	Priority-based preemptive	Free for educational and commercial	Online document	27	C	No support	[11]
Salvo	Commercial	Pumpkin Inc.	Cooperative	Fee-based	Online document	31	C	No support	[20, 21]
TinyOS	Educational/ Research	UC Berkeley	Cooperative	Free for educational and commercial	Tutorials	-	nesC	No support	[22, 23]
SharcOS	Commercial	JDC Electronics SA	Priority-based preemptive	Fee-based	User manual	-	C	No support	[24]
XMK OS	Educational/ Research	Shift-right Technologies	Priority-based preemptive	Free for educational and commercial	Online document (incomplete)	-	C	No support	[25]
Echidna	Educational/ Research	Maryland University	Priority-based preemptive	Free for educational and commercial	Online document (incomplete)	18	C	No support	[26]
eCOS	Commercial/ Educational/ Research	eCosCentric	Priority-based preemptive	Free for educational and commercial	Book and online document	-	C	No support	[27]
Erika	Educational/ Research	Universita di Siena	Priority-based preemptive	Free for educational and commercial	Online document	19	C	No support	[28]
Hartik	Educational/ Research	RETIS Lab (Italy)	Priority-based preemptive	Free for educational and commercial	Online document	33	C	No support	[29]
KeilOS	Commercial	Keil	Priority-based preemptive	Fee-based	Online document	-	C	Supported by Keil IDE	[30]
PortOS	Research	Software Wireless	Priority-based preemptive	Fee-based	Online document	-	C	No support	[31]

Table 2: Basic features comparison of RTOSes for small microcontrollers

B. Comparison results

Table 2 shows the features comparison among the RTOSes. From the table, it can be seen that:

- Priority-based preemptive scheduling has been adopted by majority of the RTOSes, except for two RTOSes in the list (Salvo and TinyOS) using cooperative scheduling.
- Majority of RTOSes support C language, which is the popular choice for embedded system programming, especially in small system design [32].
- Only a few RTOSes have OS-awareness support in IDE: µC-OS/II and EmbOS have plug-in modules for IAR compiler; KeilOS is supported by Keil compiler; µITRON and µTKernel are supported by Renesas HEW compiler.
- In the case of eCOS [27], a bootloader (known as Redboot) of at least 64K ROM is required. Redboot will boot up first and load programs into the RAM via a user terminal (typically over a serial port). Hence, eCOS requires much more ROM and RAM spaces.
- As far as documentations are concerned, some RTOSes (KeilOS, PortOS and XMK) do not have details of the APIs available. SharcOS is based on µC-OS/II; hence it follows the same APIs of µC-OS/II. For the above reasons, these RTOSes will not be considered in the following APIs comparison.
- In the "system call & API richness" column, those RTOSes that do not have details available will be represented by a "-".

Digital Object Indentifier 10.1109/MM.2009.56 0272-1732/$26.00 © 2009 IEEE

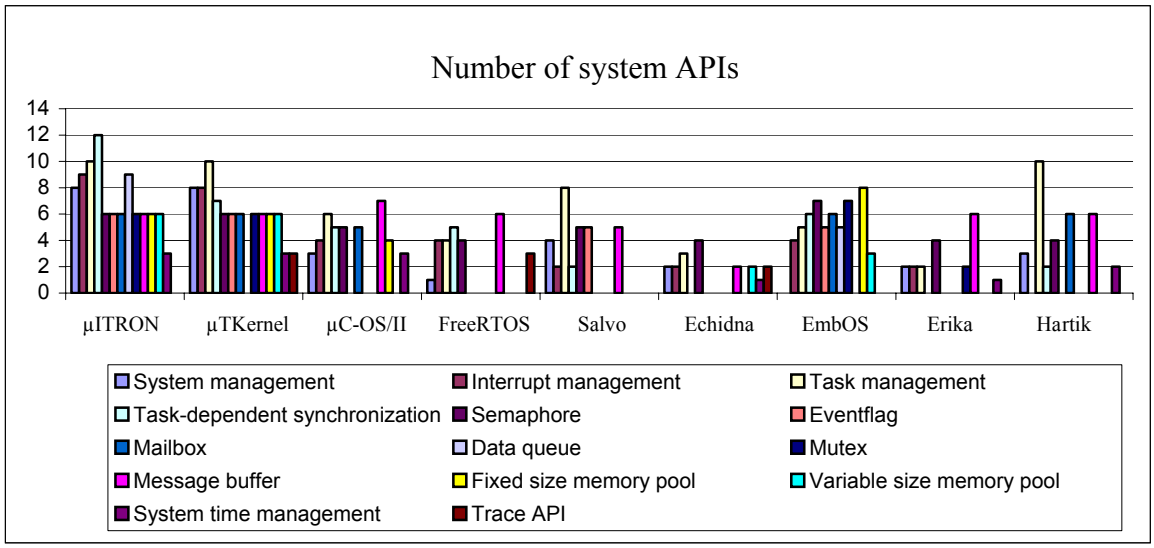

Figure 3: Number of system APIs for various RTOSes

Figure 3 shows a comparison of the number of system APIs available for each RTOS. Based on the RTOSes common definitions, the APIs in Figure 3 are categorized into:

- **System management**: initialize OS, start/shutdown OS, lock/unlock CPU, etc.
- **Interrupt management**: entry/exit function, begin/end critical section, etc.
- **Task management**: create task, delete task, start task, terminate task, etc.
- **Task-dependent synchronization**: sleep task, wakeup task, resume task, etc.
- **Communication and synchronization**: semaphore, data queue, event flag, mailbox, mutex, message buffer.
- **Memory management**: fixed size memory pool, variable-size memory pool.
- **Time management**: get system operating time, OS timer, etc.
- **Trace API**: hook routine into certain RTOS functions such as scheduler.

µITRON, µTKernel, µC-OS/II and EmbOS support comprehensive APIs for all the categories listed above. Most commercial RTOSes 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 RTOSes also exceed other RTOSes because they have been developed and improved, and have been in the market for a relatively long period.

As far as open-source RTOSes (such as FreeRTOS, Echidna, Erika and Hartik) are concerned, most have minimal implementation. These RTOSes are more suitable for small system development. However, µC-OS/II stands out to have more APIs available. µC-OS/II originally was an open-source RTOS for personal and educational purposes. Due to its stability and popularity, it has been commercialized and is being widely used [8]. Among those open-source RTOSes surveyed, µC-OS/II and µTKernel have the most number of APIs available. µTKernel supports all categories, except for data queue. It has almost the same number of APIs as the commercial RTOS µITRON, as it is backed by T-Engine Forum [33] led by Professor Ken Sakamura who is also the designer of µITRON architecture. For the above reasons, these four RTOSes: µITRON, µTKernel, µC-OS/II and EmbOS will be focused for subsequent comparison and benchmarking.

Besides those functions mentioned in Figure 3, the following are also supported by some RTOSes:

- **Timeout**: timeout is supported in some system

Digital Object Indentifier 10.1109/MM.2009.56

APIs. For example, a task can wait for a semaphore for a maximum number of *n* milliseconds. RTOSes such as µITRON and µTKernel have mechanism to allow users to specify the timeout in absolute values (typically in milliseconds). Other RTOSes such as µC-OS/II, FreeRTOS, Salvo and EmbOS can only allow users to specify timeout values in terms of the number of clock ticks.

- **Debug API**: these are the APIs that allow user's application to retrieve information managed by the kernel. Currently, only µTKernel supports these APIs (e.g. get task register, set task register).
- **Cyclic handler, alarm handler**: cyclic handler is a mechanism to indicate to the RTOS to execute a function at a periodic interval while alarm handler allows RTOS to execute a function after a certain amount of time. These APIs are currently available in µITRON and µTKernel.
- **Rendezvous**: µITRON and µTKernel also support rendezvous mechanism (similar to Ada language [34] for real-time) for synchronization and communication between tasks.

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

Parameters	µC-OS/II	EmbOS	µTKernel	µITRON
Name/ information			Extended information	Semaphore name
Attributes			- Tasks can be queued in order of FIFO or in order of priority	- Tasks can be queued in order of FIFO or in order of priority
			- The first task in queue has precedence or task with fewer request has precedence	
Initial Count	Initial semaphore count	Initial semaphore count	Initial semaphore count	Initial semaphore count
Maximum Count				Maximum value of semaphore count

Table 3: Users controllable parameters for semaphore creation in RTOSes

Based on Table 3, tasks in µC-OS/II and EmbOS are queued in a FIFO (first in first out) buffer when waiting for a semaphore, and developers are not allowed to change the order. However, in µITRON and µTKernel, developers can specify whether tasks are queued in FIFO order or in priority order. This flexibility not only applies to semaphore, but is also extended to other APIs (including mailbox, message queue, memory pool and event flag). To achieve such features in µITRON and µTKernel, there are tradeoffs in memory footprint as well as performance.

III. Performance and Memory Footprint Benchmarking

A. Benchmarking methods

In this section, these RTOSes will be benchmarked: µITRON, µTKernel, µC-OS/II and EmbOS. As discussed in the previous section, the main reasons these RTOSes were selected are:

- Comprehensive and mature APIs.
- Memory footprint and design concepts are suitable for small microcontrollers.
- These four RTOSes are made available on the same platform for the purpose of benchmarking: the Renesas M16C/62P starter kit with the HEW IDE together with the NC30 toolchain [35] are used. Details of the RTOSes ports on this platform will be described in the next section.

For execution time measurements, oscilloscope and logic analyzer have been used in combination with IO port toggling to achieve the best accuracy (in terms of micro-seconds).

(1) Ports of the RTOSes on the same M16C/62P platform

This section describes the platform, the IDE and toolchain (compiler, assembler, and linker) used. Also, the differences of these RTOSes in implementation and distribution form will be discussed. Hopefully, this will helps the readers to understand and appreciate the comparison results better in the later sections. The following are information on the M16C/62P microcontroller platform:

- Microcontroller: Renesas M16C/62P 16-bit.
- Operating frequency: 24MHz.
- ROM: 512Kbytes.
- RAM: 31Kbytes (no cache and MMU (memory management unit)).
- Interrupt mask level: 7 levels.

Digital Object Indentifier 10.1109/MM.2009.56 0272-1732/$26.00 © 2009 IEEE

- IDE: Renesas HEW version 4.03.00.001.
- Toolchain: NC30 toolchain version 5.43.00.

The Renesas M16C/62P has a 16-bit CISC (complex instruction set computer) architecture CPU with a total of 91 instructions available. Most instructions take 2 to 3 clock cycles to complete. The MCU is designed with a 4-stage instruction queue buffer [41] which is similar to a simplified pipeline often used in larger 32-bit processor.

To ensure all the RTOSes operate on the same platform with the same timer resolution, the following settings have been used:

- The 'OS tick' resolution for all the RTOSes above is taken from timer A0 [36] of the microcontroller. Timer A0 is configured as a periodic timer at 10ms.
- Default settings of NC30 compiler have been used for compiling the workspaces.
- µC-OS/II: the whole original workspace and full source code of the OS have been taken from Micrium website [37]. Timer A0 was configured for the OS, and the whole workspace was compiled again with NC30 toolchain version 5.43.00.
- µTKernel: the whole original workspace and full source code of the OS have been taken from superh-tkernel.org website [38], Timer A0 was configured at 10ms.
- EmbOS: the whole original workspace and library files (.lib) of the OS have been taken from Segger website [39]. Timer A0 was configured for the OS and the whole workspace was compiled with NC30 toolchain version 5.43.00 – except for the .lib files. The .lib files might have been compiled in older toolchain or with different optimization settings which is not known to the author of this paper. This means that for the case of EmbOS, the toolchain and compiler settings might not be exactly the same as µC-OS/II or µTKernel.
- µITRON: the whole workspace, library files (.lib) of the OS and timer A0 configuration have been generated from the Renesas configuration tool for µITRON [40]. The entire workspace was compiled with NC30 toolchain version 5.43.00 – except for the .lib files. This is similar to the case of EmbOS because the OS is distributed in the form of .lib files. Furthermore, the toolchain and compiler settings that were used when generating these .lib files might not be totally the same as µC-OS/II or µTKernel.
- The amount of stack per task is set to be the same for all the RTOSes.
- If a particular RTOS feature is not used (e.g. semaphore, message queue), that feature is disabled by C preprocessor for µC-OS/II and µTKernel, or disable during linking for EmbOS. However, for µITRON, unused features are still included as the RTOS is provided in library format and system calls are invoked via software interrupts.

Table 4 shows the differences in implementation of system service call and critical section of the RTOSes.

	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	Not disable any interrupt (based on internal variable)	Raise interrupt mask level to level 4

Table 4: System service call implementation and critical section implementation

In µC-OS/II and EmbOS, whenever a system service call is issued, the function is called directly from the user task. The advantage is that the function is executed immediately with minimal overhead time, while the disadvantage is that the service call will use the current task's stack for execution. In µITRON and µTKernel, whenever a system service call is issued, a non-maskable software interrupt is raised (INT instruction in M16C/62P [41]), hence the current execution context is switch to the kernel space (i.e. separate stack) to execute the service call. The advantage of using this approach is that the service call will not use the current task's stack for execution, while the disadvantage is that there will be an overhead time incur for every system service call.

Table 4 also shows the different methods for critical section implementation. µC-OS/II starts the critical section by disabling all the interrupts, so no external interrupt can be accepted. The advantage is that the critical section part can execute safely from start to end without any intervention, but this also implies that highly important real-time interrupt will not be accepted

Digital Object Indentifier 10.1109/MM.2009.56 0272-1732/$26.00 © 2009 IEEE

during this period. In µITRON and µTKernel, critical section is implemented by raising the interrupt mask level (for M16C/62P the interrupt mask level is set to 4, but can be changed) so that highly critical interrupt can still be accepted, as long as the interrupt handler does not interfere with the RTOS internal variables. EmbOS does not disable nor raise the interrupt mask level for critical section. It still allows all interrupts to come in but uses some internal variables to control the critical section. This has the advantage that any interrupt can be accepted and handled during the critical section; however the disadvantage is that it requires additional code to handle the internal variables of the critical section.

(2) Benchmarking criteria

The proposed benchmarking criteria in this section are aimed to be simple and easy to port to different platforms. For each criterion, execution time measurement together with memory footprint (ROM and RAM) will be collected.

(a) Task switch time

Task switch time is the time taken by the RTOS to transfer the current execution context from one task to another task. The measurement method is explained in Figure 4.

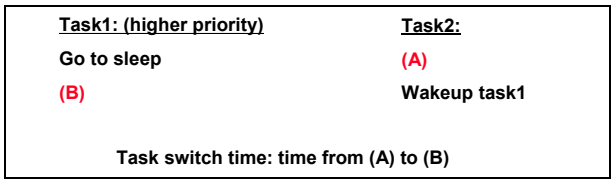

Figure 4: Task switch time measurement

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

Table 5: APIs used for task switch time benchmark

There are two tasks: Task1 and Task2 with Task1 having higher priority. At the beginning, Task1 is first executed, and it will go into *sleep/inactive* state. The execution context is then switched over to Task2. Task2 will *wake up/make active* Task1, and right after waking up, the execution context is switched over to Task1 because it has higher priority. Different RTOSes use different terms to describe *sleep/inactive* and *ready/active* states (such as µC-OS/II and EmbOS use the term *suspend, resume* while µITRON and µTKernel use the term *sleep/wakeup*). Table 5 shows the system calls used in each RTOS.

(b) Get/Release semaphore time

Semaphore is commonly used for synchronization primitive in RTOS [42]. For semaphore benchmarking, the time taken by *get* and *release* semaphore service call will be measured, and the time required to pass the semaphore from one task to another task will also be measured. Figure 5 illustrates the method used to measure the *get* and *release* semaphore time.

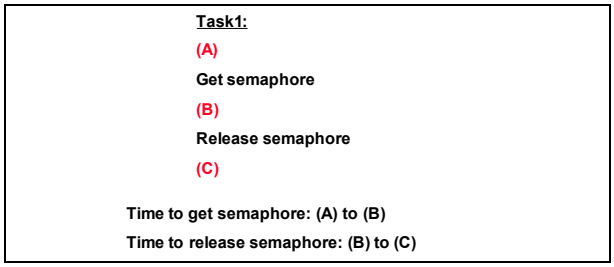

Figure 5: Get/Release semaphore time measurement

There is only one task (Task1) and one binary semaphore (initialized to 1). Task1 will get the semaphore and then it will release it. Different RTOSes use different terms to describe the get/release of semaphore. Table 6 shows the APIs used by each RTOS.

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

Table 6: APIs used for semaphore benchmark

(c)Semaphore passing time

To measure the performance of semaphore passing, the following measurement method as shown in Figure 6 is used.

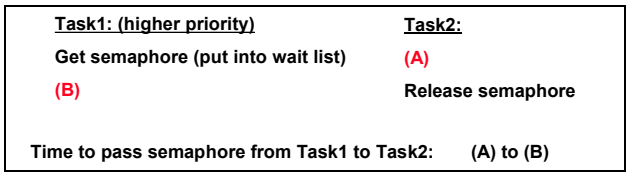

Figure 6: Semaphore passing time measurement

There are two tasks: Task1 and Task2 with Task1

Digital Object Indentifier 10.1109/MM.2009.56 0272-1732/$26.00 © 2009 IEEE

having higher priority, and a binary semaphore (initialized to 0). At the beginning, Task1 is first executed and it tries to get the semaphore. Since the semaphore value is 0, Task1 will be put into a *sleep/inactive* state, waiting for the semaphore to be released. The current execution context will then be switched to Task2 which will release the semaphore. The semaphore, once released, will wake up Task1, and the execution context will be switched over to Task1.

(d) Pass/Receive message time

Besides semaphore, message passing has become more and more popular for synchronization purposes [43]. In this measurement, message passing mechanism based on memory pointer passing is used, i.e. not the copying of message into an internal RTOS area because not all RTOS support this approach. The measurement method is explained in Figure 7.

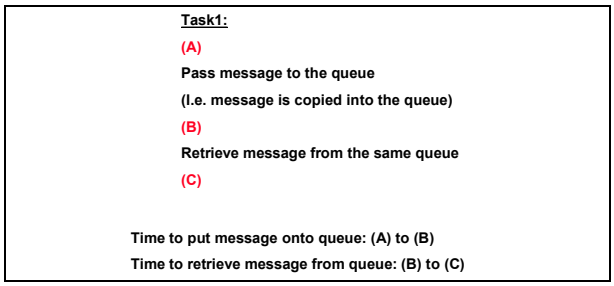

Figure 7: Pass/Retrieve message time measurement

There is only one task: Task1. Firstly, the task will pass the message pointer (normally to an internal message queue), and after that the task will retrieve the same message pointer. Table 7 shows the APIs used in each RTOS.

	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()

Table 7: APIs used for message passing benchmark

(e)Inter-task message passing time

Figure 8 illustrates the method used to measure the message passing time between tasks. There are two tasks: Task1 and Task2 with Task1 having higher priority. Task1 will be first executed, and it will try to retrieve a message pointer from the queue. As there is no message available yet, Task1 will be put into *sleep/inactive* state, waiting for a new message. The current execution context will be switched to Task2, which will then put a new message onto the queue. The new message will wake up Task1, and the execution context will be switched over to Task1. The difference between this measurement and the benchmark in the previous section is that this method includes the overhead time by RTOS to process the message queue and to wake up the receiving task.

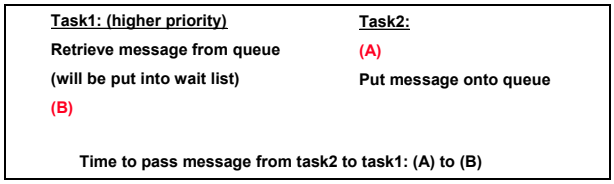

Figure 8: Message passing time measurement

(f) Fixed-size memory acquire/release time

In RTOS, only fixed-size dynamic memory allocation should be used. The allocation and de-allocation time must be deterministic. Figure 9 illustrates the method used to measure the time to acquire and the time to release a fixed-size memory block.

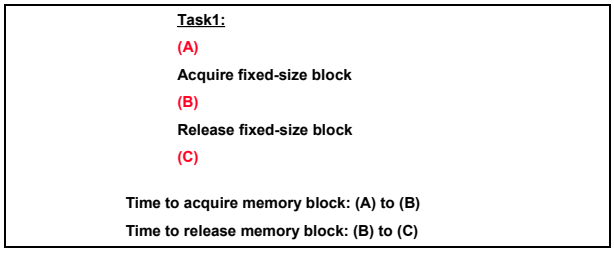

Figure 9: Acquire/Release fixed-size memory time measurement

There is only one task: Task1. Task1 first acquires a fixed-size memory block (128 bytes), and then releases the block. Table 8 shows the APIs used to acquire/release memory block in each RTOS.

	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 8: APIs used for fixed-size memory benchmark

(g) Task activation from within interrupt handler time

An RTOS has to deal with external interrupts that may be asserted at any time. Execution of interrupt handler is normally kept as short as possible to avoid

Digital Object Indentifier 10.1109/MM.2009.56

affecting the system response. In the case where long processing is required, the handler can activate another task that will do the necessary processing. The time from when the interrupt handler resumes the task till the time when the task is executed is crucial to the system design. The measurement setup is explained in Figure 10.

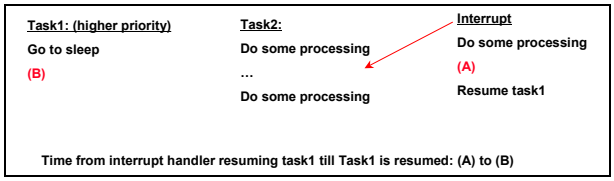

Figure 10: Task activation from interrupt handler time measurement

There are two tasks: Task1 and Task2 with Task1 having higher priority. Besides, an external interrupt with proper handler is also set up. At the beginning, Task1 is first executed. Task1 goes to *sleep/inactive* state, and the execution context switches over to Task2. Task2 simply does some processing continuously. When an externally interrupt occurs, the interrupt handler will be executed and it will resume Task1. Execution context will be switched over to Task1. Table 9 shows the APIs used for each RTOS.

	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()

Table 9: APIs for task activation from within interrupt handler benchmark

B. Benchmarking results

(1) Memory footprint

For each criterion, the benchmarking code is compiled, and the ROM and RAM usage can be obtained from the toolchain report. By averaging the ROM information across all the test criteria, the average ROM size can be obtained. Figure 11 shows the code sizes for the 4 RTOSes when running the 7 benchmarks.

µTKernel can be seen to have a larger code size. This is due to the flexibility and comprehensiveness support in the APIs, as explained previously in Table 3. µITRON and EmbOS, which are commercial RTOSes, offer relatively compact code size. Nevertheless, all these RTOSes can fit well into small microcontrollers of limited ROM sizes.

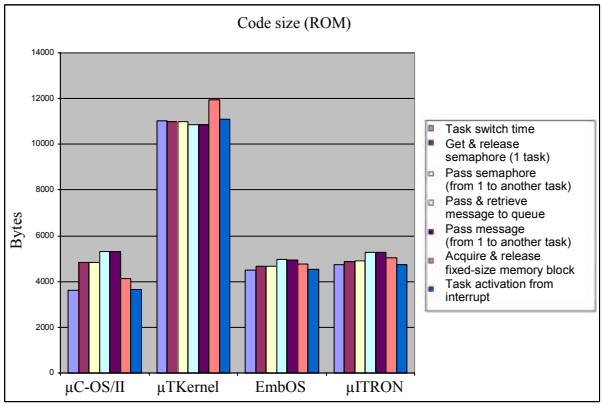

Figure 11: Code size comparison for 4 RTOSes

Similar to the code size comparison, Figure 12 shows the RAM information for the 4 RTOSes. µTKernel and µITRON can be seen to have relatively lower RAM usage, while µC-OS/II and EmbOS are slightly higher. Depending on the requirement of each benchmark, we set the number of tasks, stack size and number of RTOS objects (e.g. semaphore, event flags) to be the same for all RTOSes. The amount of RAM differences among the RTOSes range between 7-10 bytes, which might be due to internal implementations or due to method of designing the APIs. In summary, the ROM and RAM usage of all these RTOSes are well suited for small microcontrollers. Based on Figure 11 and Figure 12, µITRON has the most optimal usage in terms of both ROM and RAM sizes.

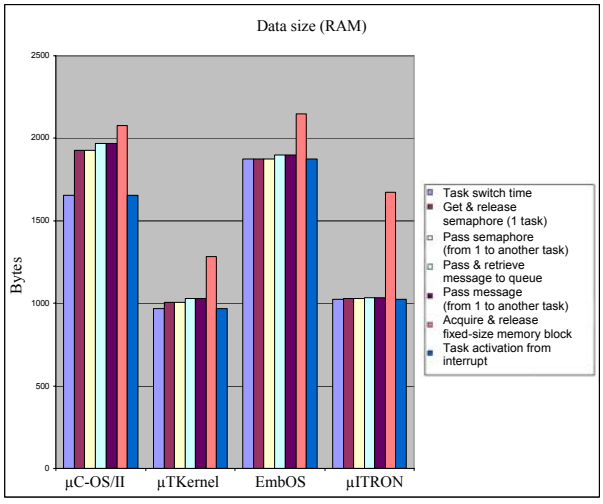

Figure 12: Data size comparison for 4 RTOSes

Digital Object Indentifier 10.1109/MM.2009.56

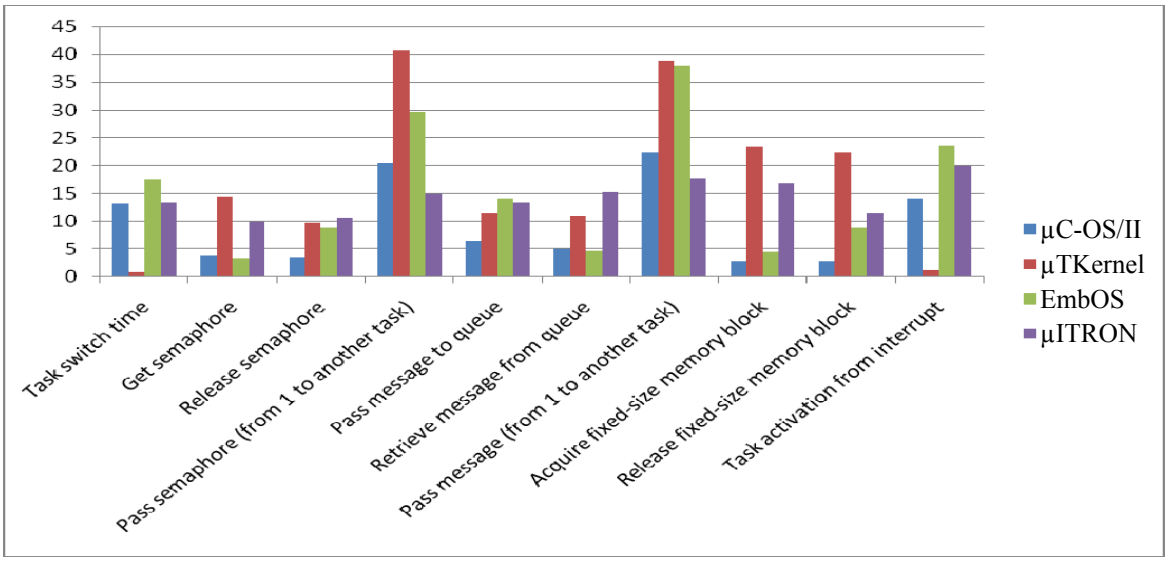

Figure 13: Execution time benchmark for four RTOSes

(2) Execution time

Figure 13 shows the measurement of execution time among the RTOSes for different benchmark criteria. As the only variation in the system is timer interrupt (for OS tick), each benchmark was executed 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, there will be another variation which is an external interrupt (besides the timer interrupt for OS tick). When performing this benchmark, the external interrupt may or may not be asserted during the OS critical section (the duration which OS disable interrupts). If it is asserted during the critical section, the response time of the OS will be slightly longer. Hence this measurement may not include the worst case scenario.

µTKernel is shown to have the lowest task switching time, followed by µITRON, µC-OS/II and EmbOS. On the other hand, µC-OS/II semaphore *acquire* and *release* time are the fastest. The fastest inter-task semaphore passing is achieved by µITRON, while µC-OS/II and µTKernel have better message passing and message retrieval time as compared to 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.

With the benchmarking results shown above, each RTOS stands out to have its own strengths and weaknesses. As far as open-source RTOS is concerned, for a very small and compact ROM size RTOS, µC-OS/II can be used. However, to have a more comprehensive APIs support, µTKernel is recommended (at the expense of a slightly higher ROM footprint). On the other hand, to select a commercial RTOS, either µITRON or EmbOS is recommended with µITRON having slighter lower RAM footprint. Based on all these benchmarked results, different performance criteria can be compared to ensure that they meet the time constraint requirements of the system when selecting for a RTOS.

IV. Conclusion

RTOS is increasingly popular for deployment with microcontroller-based embedded systems design. It can helps to improve the development cycles, code reusability, easy of coding as well as maintenance, better resource and timing management even for small microcontrollers. RTOS can be differentiated from generic OS in terms of scheduling (priority-based), predictability in inter-task synchronization, and deterministic behaviors. In this paper, a list of RTOSes

Digital Object Indentifier 10.1109/MM.2009.56

has been studied based on various selection criteria targeting small microcontrollers with less than 128Kbytes of ROM and 4Kbytes of RAM. Subsequently, four RTOSes were selected and benchmarked by porting them onto the same microcontroller platform. A list of benchmarking criteria which is aimed to be simple, easy to be ported to different platforms and representative of typical RTOS usages were used. The benchmarking results show that each RTOS has different strengths and weaknesses without any clear emerging winner. With these detail performance benchmarks, potential adopters of these RTOSes can simplify their selection by examining their specific application requirements.

REFERENCES

[1] D. Kalinsky, "Basic concepts of real-time operating systems", LinuxDevices magazine, Nov. 2003.
[2] K. Sakamura, H. Takada, "µITRON for small scale embedded systems", IEEE Micro, vol. 15, pp. 46–54, Dec. 1995.
[3] J. Ganssle, "The challenges of real-time programming", Embedded System Programming magazine, vol. 11, pp. 20–26, Jul. 1997.
[4] CMP Media, "State of embedded market survey", Embedded System Design Magazine, 2004.
[5] CMP Media, "State of embedded market survey", Embedded System Design Magazine, 2006.
[6] CMP Media, "State of embedded market survey", Embedded System Design Magazine, 2005.
[7] CMP Media, "State of embedded market survey", Embedded System Design Magazine, 2007.
[8] K. Baynes, C. Collins, E. Filterman, B. Ganesh, P. Kohout, C. Smit, T. Zhang, B. Jacob, "The performance and energy consumption of embedded real-time operating systems", IEEE Transactions on Computers, vol. 52, pp. 1454–1469, 2003.
[9] J. J. Labrosse, "µC-OS/II the real-time kernel", R & D Books, (Miller Freeman, Inc.), Lawrence KS, 1999.
[10] T.-E. Forum, "µTKernel specification, 1.00.00", T-Engine Forum, Mar. 2007.
[11] R. Barry, "A portable, opensource mini real-time kernel", http://www.freertos.org, Oct. 2007.
[12] K. Curtis, "Doing embedded multitasking with small microcontrollers, part 2", Embedded System Design Magazine, Dec. 2006.
[13] A. Martinez, J. F. Conde, A. Vina, "A comprehensive approach in performance evaluation for modern real-time operating systems", Proceedings of the 22nd EUROMICRO Conference, pp. 61, 1996.
[14] R. P. Kar, K. Porter, "Rhealstone: A real-time benchmarking proposal", Dr. Dobb's Journal, Feb. 1989.
[15] K. M. Sacha, "Measuring the real-time operating system performance", Seventh Euromicro workshop on Real-time systems proceedings, Odense, Denmark, pp. 34–40, Jun. 1995.
[16] G. Hawley, "Selecting a real-time operating system", Embedded System Design Magazine, 1999.
[17] M. Timmerman, L. Perneel, "Understanding RTOS technology and markets", Dedicated Systems RTOS Evaluation project report, Sep. 2005.
[18] K. Sakamura, H. Takada, "µITRON 4.0 specifications", TRON Association, Japan, 2002.
[19] Segger, "EmbOS real-time operating system user & reference guide", Segger Microcontroller System GmbH, 2006.
[20] Pumpkin Inc, "Salvo user manual", Pumpkin Inc., 2003.
[21] Pumpkin Inc, "Salvo, the RTOS that runs in tiny places", http://www.pumpkininc.com/, Oct. 2007.
[22] P. Levis, "TinyOS programming, revision 1.3", TinyOS Community Forum, 27 Oct 2006.
[23] U. Berkeley, "TinyOS, an open-source OS for the networked sensor regime", http://www.tinyos.net/, 2006.
[24] J. Electronic, "SharcOS user guide", The SharcOS project, 2002.
[25] S. R. Technologies, "eXtreme Minimal Kernel, a free real-time operating system for microcontrollers", http://www.shift-right.com/xmk/index.html, 2004.
[26] D. Stewart, "Echidna real-time operating system", http://www.glue.umd.edu/dstewart/serts/research/echidna/index.shtml.
[27] A. J. Massa, "Embedded software development with ECOS?" Prentice Hall, Nov. 2002.
[28] P. Gai, D. Cantini, M. Cirinei, A. Macina, and A. Colantonio, "Erika, embedded real-time kernel architecture - education user manual", Realtime System (RETIS) Lab, Scuola Superiore Sant'Anna, Italy, 2004.
[29] G. C. Buttazzo, "Hartik: A hard real-time kernel for programming robot tasks with explicit time constraints and guaranteed execution", Proceedings of the 1993 IEEE International Conference on Robotics and Automation, Atlanta, Georgia, USA, pp. 404–409, May 1993.
[30] Keil, "Keil real-time kernel and operating system", http://www.keil.com/rtos/, Mar. 2007.
[31] R. Chrabieh, "Operating system with priority functions and priority objects", TechOnline technical paper, Feb. 2005.
[32] R. Bannatyne, G. Viot, "Introduction to microcontrollers, part 2", Northcon98 Conference Proceedings, pp. 250–254, Oct. 1998.
[33] T-Engine, "T-Engine forum", http://www.t-engine.org, Nov. 2007.
[34] B. Millard, D. Miller, C. Wu, "Support for ADA intertask communication in a message-based distributed operating system", Computers and Communications Conference Proceedings, pp. 219–225, Mar. 1991.
[35] Renesas Technology Corp., "Renesas high-performance embedded workshop (HEW)", http://www.renesas.com/fmwk.jsp?cnt=ide_hew_tools_product_landing.jsp&fp=/products/tools/ide/ide_hew/, 2007.
[36] Renesas Technology Corp., "M16c/62P group hardware manual", http://documentation.renesas.com/eng/products/mpumcu/rej09b0185_16c62pthm.pdf, Jan. 2006.
[37] Micrium, "µC-OS/II ports for Renesas microcontrollers," http://www.micrium.com/renesas/index.html, Nov. 2007.
[38] Renesas Technology Corp., "µTKernel for M16C source code and documentation", http://www.superh-tkernel.org/eng/download/misc/index.html, Nov. 2007.
[39] Segger, "EmbOS trial version for M16C6X (HEW 4.0 with NC30 version 4.00)", http://www.segger.com/downloadform_embos_m16c_nc30_v540.html, Nov. 2007.
[40] Renesas Technology Corp., "M3T-MR30/4 RTOS for M16c and R8c families conforming to µITRON 4.0-specification", http://www.renesas.com/fmwk.jsp?cnt=m3t-mr30_kernel.htm&fp=/products/tools/os_middleware/u_itron/m3t

Digital Object Indentifier 10.1109/MM.2009.56

_mr30/child_folder/&title=M3T-MR30%20Kernel, Nov. 2007.

[41] Renesas Technology Corp., "M16C/60, M16C/20, M16C/Tiny series software manual", http://documentation.renesas.com/eng/products/mpumcu/rej09b0137_m16csm.pdf, Jan. 2004.

[42] I. Ripoll, P. Pisa, L. Abeni, P. Gai, A. Lanusse, S. Saez, "RTOS state of the art analysis", Technical report, Open Components for Embedded Real-time Applications (OCERA) project, 2002.

[43] D. Kalinsky, "Asynchronous direct message passing rapidly gains popularity", Embedded Control Europe Magazine, Nov. 2004.

Digital Object Indentifier 10.1109/MM.2009.56 0272-1732/$26.00 © 2009 IEEE