International Journal of Electronics Communication and Computer Engineering
Volume 4, Issue (2) REACT-2013, ISSN 2249–071X

National Conference on Research in Electronics Advances & Collaborative Technologies – 2013

Comparison of Open Source RTOSs Using Various Performance Parameters

Sanjay Deshmukh
Department of Electronics and Telecommunication
D.J Sanghvi COE, Mumbai

Dr. N. N. Mhala
Department of Electronics,
BDCOE Wardha

***Abstract* — The performance parameters of RTOSs is the most important factor for consideration to a system developer as they determine the performance of the entire system. In this paper we give a fundamental overview of some of the most important parameters and how they affect the system performance. It is observed that scheduling is the most important parameter. Further, we considered three famous Open-Source RTOSs- RT Linux, Free RTOS and eCos- based on the parameters, kernel architecture and applications. Effort is made to include details which will simplify the task of a developer who wishes to use these RTOSs. Finally, we provide simple and universally implementable benchmarking technique for measurement of parameters for any RTOS. It is our effort to organize the complex and often contradictory information about RTOSs and present it in a manner to make the decision process is simple and effective.**

***Keywords* – Benchmarking, Latency, Open-Source, Preemption, RTOS, RTOS Parameters, Scheduling,Thread Control Block.**

I. INTRODUCTION

A generic real-time system has requirement to produce the results within a specified deadline period. RTOS is like any other OS, a software having a set of Application Programming Interface (APIs), which programmers can use to build systems and solutions. However, what makes these Operating Systems special is that they assist an embedded system to meet deadlines. Selecting any RTOS for a specific application is governed by certain parameters, that determine its performance. Hence, there is always a trade-off required as for example a particular application may require least interrupt latency, but the RTOS providing this may have higher priority inversion probability. The parameters we considered are- Scheduler, Thread Synchronization, Interrupt Handler, Thread Switch Latency, Interrupt Latency, Synchronization and Language Support. Linux, eCos and Free RTOS are three commonly used open source RTOSs and we use these in our consideration. All three use C language as their kernel programming language, and eCos can make use of C++ as well. As the scheduler is the decision making body of the RTOS and determines its entire functioning, thus extra emphasis is laid on it. The mentioned parameters are affixed to these RTOSs. Conventional comparison of these RTOSs is not possible due to the diverse variations in their performance under different conditions. Also as the process of choosing a RTOS for a particular application is not clear. Some other factors that affect a developer's choice of RTOS selection are documentation support, developer reputation, cost, size of code etc. These are sole choices based on an individual's perception and outside the technical scope of this paper. We also outline the algorithms, which the user can use to measure the important parameters and hence decide whether the RTOS is suitable for his/her requirements. These steps are generic and not restricted to any particular RTOS.

II. FUNDAMENTALS

Every RTOS has a core component, Kernel .The core is surrounded by shell layers, providing protection and access authorizations. The RTOS manufacturer provides a set of APIs, that are used to access this kernel to perform the specified tasks. The memory of the RTOS is divided into two parts, Kernel space consisting of the kernel code and the user space, which consists of the user code. The threads in an RTOS are similar to functions, having its own stack and a Thread Control Block (TCB). The kernel has a scheduler to execute these threads in a sequential manner. RTOSs are of three types; Hard real-time, Firm real-time and Soft real-time. In a hard real-time system the tasks and interrupt requests must be completed within their required deadlines. Failure to meet a single deadline can cause a critical catastrophic system failure. Some RTOS like Firm-real time system tolerates the missing of a few deadlines, however, missing more than a few may lead to complete disaster. A soft real-time system is one in which deadlines are mostly met, but this constraint is not very tightly time bound. Every RTOS should have some support for multitasking (threads). RTOS usually supports thread synchronization using semaphores or mutexes. It is preferably to have RTOS with sufficient number of priority levels and with provision to avoid priority inversion.

III. PARAMETERS

A. Scheduler

A scheduling algorithm determines how RTOS assigns CPU time for task execution. Scheduler is the prime component of the kernel as it affects the way in which the system will execute tasks and machine cycles. The

scheduler executes periodically and during the state transition of thread. The most common scheduling used in RTOSs is Preemptive scheduling, especially in hard real-time RTOSs**.**

Here, the currently being executed kernel code or user code, can be interrupted by a higher priority task. The kernel will branch and service the interrupting task and then resume the execution of original code. Another common scheduling is Earliest Deadline First (EDF) in which the task having the earliest deadline is assigned the highest priority and task having the furthermost deadline the lowest priority.

B. Event Latency and Jitter

An event could be an interrupt request by some hardware or a task being scheduled by the operating system. For a hardware interrupt, the latency is the time elapsed from the interrupt request generation to the execution of the first line of ISR code. For a system event, latency is the time elapsed from the generation of the signal task to the execution of the first instruction of the task. Jitter is any variation of the fixed time period for a task execution, servicing of an interrupt etc.

C. Priority Inversion

Priority inversion is a major concern of most RTOS. In this, the lower priority task pre-empts the higher priority task and this reverses the priority structure. Most RTOS has a provision to avoid priority inversion and the common ones are priority ceiling and priority inheritance. In priority ceiling, every process that shares a resource is assigned a priority equal to the highest priority process which may lock the resource. In priority inheritance, if a high priority task is waiting for a currently being executed lower priority task, then the low priority task is temporarily (till its execution) assigned the priority of the waiting highest priority task. This avoids the pre-emption of low priority task by other medium priority tasks.

D. Memory Management

It is the allocation of memory space available in the system, to any task that requests it. After the task completion using allotted the memory space and does not require it any further, then the memory needs to be de-allocated and made available to other threads. Different kernels use different techniques to ensure this. The two common models are the Flat Memory Model and Segmented Address Model. [10]

E. Mutexes and Semaphores

Certain critical sections of the code access a shared resource. Mutex is a flag, which protects this shared resource from simultaneous accessment of resource by more than one tasks. Semaphore is a non-binary mutex, that is, it can count greater than one. Mutexes and Semaphores provide mechanism for accessment to a shared resource, by aligning the tasks which request access to a shared resource in a queue.

IV. RT LINUX

RT linux is designed to impart hard real time performance in LINUX. The linux kernel is monolithic with modules. The system has two components, real-time and non-real time system. The real-time component runs on the RT kernel, while the non-real time part runs on Linux. The communication between the real-time and non-real time component is via FIFO buffers, called RT-FIFOs. The RT kernel is placed between hardware and the Linux kernel and thus intersects and attends to all interrupts. If an interrupt occurs for a real-time task to run, the RT kernel preempts linux if it is running at the time and allows the real-time task to run.

A. Scheduler

RTLinux support a limited number of architectures, like x86, PowerPC, MIPS, Alpha, unlike eCos. Linux itself can support only 100 prioirty levels altogether. Using system function sched_get_priority_min() and sched_get _priority_max() the scheduling policy can determine the min and max priority. With the RT extension, the kernel can support prioritized FIFO scheduler and extensible scheduler with 1024 priority levels. RTLinux claims to have hard real-time interrupt latency, whereas the other latencies are CPU dependent. [11]

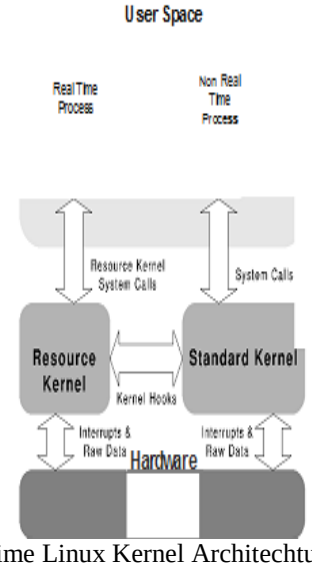

Fig1. Real Time Linux Kernel Architechture. [10], Pg. 36

The normal Linux kernel does not support preemption by a real-time constraint unless only a user space code is being executed. For implementation of full preemption, the CONFIG_PREEMPT patch needs to be installed. This will allow preemption of user code and kernel space code by higher priority real time threads. This reduces , the worst case latency approximately to single figure millisecond. However, the disadvantage of this method is that the kernel code is pre-empted non-voluntarily. This leads to a there is lot of context switching reducing the throughput of the system.

B. Interrupt Handling and Event Jitter

The interrupt mechanism is divided into two parts. The bottom part accepts the interrupt request from the peripheral device and stores the request in the memory buffer. The top part reads this buffer and transfers it to the kernel accessible buffer. RT linux kernel does not accept any interrupt enable/disable request that is issued by the Linux kernel to the hardware ,instead it is emulated . For example if Linux disables a hardware interrupt, it will actually remain enabled and the RT kernel will queue this request and hence delay the delivery. However, real-time interrupts are not affected at all and handled in the normal manner by the RT kernel. All RT Linux variants are enhanced to reduce interrupt latency and jitter. The 2.6 kernel version patched with preemption code and running with 750MHz Pentium processor has a jitter of less than 1 millisecond for a periodic task of 5.8 milliseconds reading 768 bytes of data [4]. The faster Ghz processor and latest kernels report even less jitter and hence the Linux performance is constantly improving. [2].

C. Thread synchronization and priority inversion

RT linux uses Mutexes, joins and condition variables to achieve thread synchronization. Joins make a thread wait till the other thread is terminated. In the condition variable mechanism the execution of a thread is terminated and it gets the processor time until some condition is true. A condition variable is always used with a mutex to avoid dead-lock and race conditions.RT linux uses the priority inheritance and priority ceiling techniques to avoid priority inversion.

V. FREE RTOS

FreeRTOS is a powerful operating system and supports a host of architectures like Intel, ARM, Atmel, PIC, Xilinx etc.It uses preemptive as well as cooperative scheduling and can be configured to meet requirements.

A. Scheduling

The scheduler in FREERTOS has two operating modes Preemptive and Cooperative; this is decided by configuration switch. Cooperative scheduling avoids the reentrance problems faced by preemptive scheduling. This is because tasks that are being executed can only be interrupted at positions permitted by developer and not arbitrarily. It is important that although real-time performance is affected at task levels, interrupts continue to enjoy real-time responses by using semaphores. Tasks with highest priority are executed first, and for more than one high priority tasks, round-robin mechanism is used.

B. Latency

In FreeRTOS, when any task is created, the kernel does two memory allocations. The time required for memory allocation in the Task Control Block is fixed. The time required for the initialization of task stack is proportional to the complexity of the task, i.e. the stack size required. Also, latency of the first task creation is maximum, after which latency decreases and is equal for tasks requiring same stack size.[8] The task creation latency for the first call for a stack size of 100 units, is 0.502ms, for the second call is 0.3984ms and third call is 0.3957ms.

C. Memory Allocation

FreeRTOS has three models for allocation of memory spaces. The simplest models allocate a fixed memory to each task but do not deallocate it. Thus memory cannot be reused and leads to the wastage of memory space. The second model allows allocation and de allocation and uses a best-fit algorithm to find free space in memory. The most complex model uses custom algorithms for specific requirements of task..

D. Priority Inversion

The greatest disadvantage of freeRTOS is that it doesn't implement advanced mechanism for shared resources like priority ceilings to avoid priority inversion. This is not suitable for its use in critical applications.

VI. EMBEDDED CONFIGURABLE OPERATING SYSTEM (ECOS)

The eCos has a modular and layered Real-time Kernel based architecture. Its prime feature is the configuration system, which allows only those components, which are required for the specific applications are built into the kernel. The special Hardware Abstraction Layer is incorporated to have this configurality. This will provide a isolation between architecture and platform details and give an independent access to both, making the rest of the system completely independent of the machine. Further, this enables easy porting of the system to diverse architecture. eCos provides a GUI called eCos configuration tool for the selection of kernel components. Thus it simplifies the configuring task and attaining fine adaptation of the kernel. Its APIs are compatible with POSIX standard.

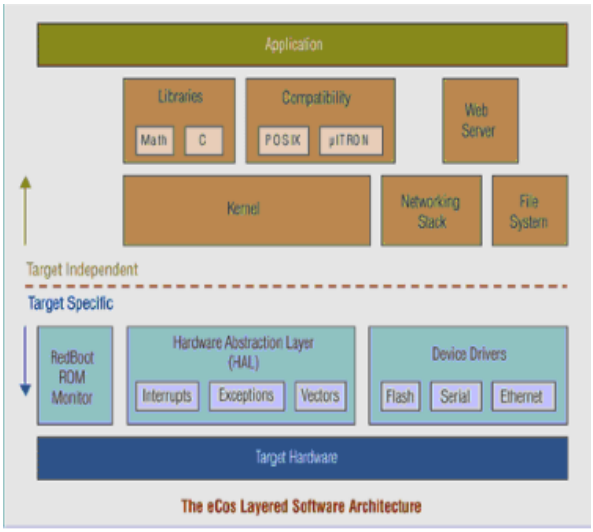

Fig.2. eCos Architecture [8],pg. 7.

A. Scheduler

eCos can have two common schedulers: Bitmap scheduler and multi-level queue scheduler. Both of them support preemption. However, at present only a single scheduler can be implemented in the system. Scheduler access can be protected by disabling interrupts during critical regions. But this increases the maximum dispatch latency. Bitmap Scheduler consists of a 32 level priority queue, with 0 being the highest priority and 31 being the lowest. Each thread is assigned a specific priority and hence at any time instant there can be only 32 executable threads. The scheduler schedules the thread with the highest current priority for execution. Preemption can be disabled. If it is enabled then a higher priority thread by the scheduler can preempt the current thread being executed. Multi Level Queue scheduler has set of queues. Each queue contains a number of threads, which have the same priority. By default, every queue uses a time-slicing algorithm to share the CPU time equally amongst all the threads of the queue. These threads assign the priority based on some individual characteristics of the thread. All of these queues are scheduled according to a normal priorityqueue where a higher priority queue gets scheduled.

B. Interrupts and Latencies

To ensure minimum latency, eCos uses a split interrupt handling mechanism. The first part is Interrupt Service Routine or ISR. The other part is the Deferred Service Routine or DSR. This ensures that least amount of time is spent in the ISR and hence greatly reduces interrupt latencies. Generally no DSR is needed if ISR is small. Further, this technique allows execution of both DSRs and higher priority ISRs by enabling interrupt enable flag. To allow DSR to be executed while keeping interrupts enabled, the ISR must ensure that the same interrupt does not recur until the DSR is finished execution. The problem of this is that the programmer does not know how much stack space to allocate to each thread because eCos does not mention the size of thread stacks. Hence, during DSR execution, if another interrupt occurs, during its service, the DSR or ISR may run out of stack space.

C. Thread Synchronisation

The eCos kernel uses the mutexes and semaphores along with other communication mechanism like flags and queues to obtain thread synchronization.

D. Priority Inversion

eCos uses both priority ceiling and priority inheritance to avoid priority inversion. eCos implements the moderate level algorithm in its priority inversion protection protocols package. This algorithm is fast and deterministic and reduces code sizes. However, it can only be used in the MLQ scheduling.

VII. BENCHMARKING TECHNIQUES

As the parameters of RTOS are highly relative and subject to the hardware and processor used, standardization of parameters like latency for a particular RTOS is almost impossible. This is because of not only will the figure vary from system to system, but also even on the same system for two different tests, the figure may be different. This is due to different background tasks at different times, different load conditions etc. Hence, very often, the mean of the results is calculated and taken as reference. There are many commercial tool kits and softwares to measure parameters such as Thread Metric Benchmark Suite. It is not in the scope of this paper to show the working of these commercial products. A study of literature on this topic show a multitude of independent researchers as well as corporations publishing their own benchmarking results, but as expected they always vary and no standardization is possible. Hence, instead of adding our own benchmarking which would be specific to only our system, we write about the fundamental processes and algorithm that can be implemented for measurement of parameters of any RTOS on any system, which would be more beneficial to a system developer.

A. Latency Measurement

Latency is calculated by measuring the execution time for any event. For this a timer based code needs to be designed. It should consist of a measurement timer, which will capture the timings with time stamps. This technique is known as time stamping. The algorithm for measurement is the timer of the software will give a time stamp at the beginning and end of function execution. Their difference is the latency. For accuracy, this should be repeated for n iterations and total time difference divided by n gives the latency, which is the average latency. It is to be noted that the latency often gets affected by the delay introduced by external peripherals such as taking the control of data buses, which could be transferring data over the bus and hence preempting CPU bus access.

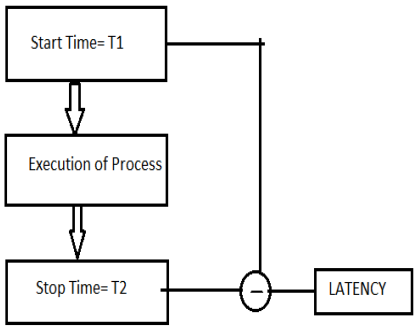

Fig.3. Algorithm for Latency Measurement.

B. Periodic Jitter Measurement

The jitter is most commonly concern to the RTOS. It can be measured using the internal clock counter register.

A code is written to increment the clock register at every clock tick and the values are stored in memory. Any time variation from the expected time periods is the measurement of jitter. The main advantage of this method is that the measurement is not affected by external peripheral delay since the clock register is internal to the processor and thus all steps occur within the CPU itself.

C. Thread Context Switching Measurement.

The objective of this measurement is to measure the time required for a lower priority thread to get control to a higher priority thread. For this, a code is written containing say seven threads, with thread 7 having lowest and thread 1 having highest priority. Each thread has assigned the specific priority. Each thread runs till it is pre-empted by a higher priority thread. Thread 7 is preempted by Thread 6 and so on. While the thread is running, its counter is incrementing. At the beginning of execution, the lowest priority thread is activated and all others are suspended. On initialization, the lowest priority thread will run for a certain time, indicated by its run count, and will be pre-empted by the next higher priority thread. The thread run-counter starts incrementing till it gets pre-empted by the next priority thread. This process continues for all n threads, in our example 7 and the code should be written to loop this entire process n times. The counter of every thread provides the thread switching time.

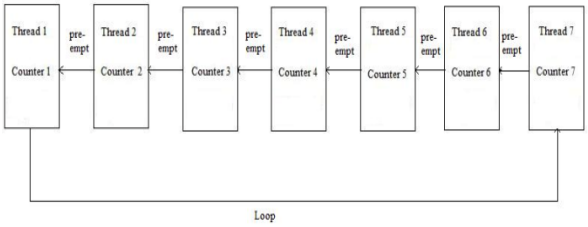

Fig.4. Algorithm for measurement of latency Thread Switching

VIII. COMPARISON

The following table gives the comparison of RT Linux, FreeRTOS and eCos based on the above mentioned parameters as well as other popular primitives. It is not possible to explicitly mention which RTOS would be best for a particular application, but by studying the following table a clear idea can be outlined as to which RTOS can be used for the specific requirements.

From the table it is evident that RT Linux is preferred in many applications. It supports multiple architectures, has a monolithic architecture, easy kernel manipulation, robust.

Table 1: Comparison of RT Linux, Free RTOS and ECOS

Parameter			
	RT LINUX	***FreeRTOS***	***eCos***
Scheduler	Preemptive	Support Preemptive and Cooperative	Bitmap scheduling Multilevel Queue
POSIX Compilant	YES	NO	YES
Interrupt and latencies	RT kernel support interrupt, preemption patch reduces latencies.	Regular interrupt handling, latency reduces as Task number increases	Latency reduction done by split interrupt handling by ISR and DSR
Priority Inversion	Support Priority Ceiling and Lock-free Structures.	No provision for advanced avoidance	Priority Ceiling and Priority Inheritance.
Thread Synchronization	Mutexes, Joins and Condition Variables.	Binary Semaphores	Mutexes and semaphores, Flags and queues
Memory Allocation	Uses regular Linux memory management provisions. No Real Time allocation.	Allocation and deallocation done using default or custom ised algorithms.	Memory pool based Dynamic Memory Allocation
Kernel Type	Monolithic with Modules	RTOS	RTOS
OS Family	Unix like	RTOS	RTOS
Architectures Compatibility	x32/64, x86-64, ARM, XScale, OpenRISC, Alpha, PowerPC, Xen, MIPS, SPARC32.	x32, ARM, OpenRISC, MIPS.	x32, PowerPC, ARM, XScale, OpenRISCSPARC32

IX. CONCLUSION

In this paper, we have attempted to include only the most relevant information and eliminate out the unnecessary. The scheduler is the most important parameter, which manages the working of the entire system, including hardware. Hard real- time systems require extremely low interrupt and event latencies. For this a preemptive system with efficient interrupt handling mechanism is needed. The next generation systems would need even lower latencies and faster thread switching to attain harder real-time systems. RT Linux is the most versatile out of the three mentioned RTOSs as it provides extensive functionalities without compromising on the essentials of a RTOS. Also, it possesses a superior presence in the world market and thus the technical support for RT Linux is better than others. The exclusive configurability provided by eCos makes it suitable for tailor made and specific applications. FreeRTOS provides robust performance characteristics.However its failure to implement effective measures to avoid priority inversion and not being fully POSIX compliant are its greatest disadvantage.

REFERENCES

[1] Z. He, A. Mok, and C. Peng, Timed RTOS Modeling for Embedded System Design, Proceedings of the 11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS), 2005

[2] Real-time operating systems for small microcontrollers, Tran Nguyen Bao Anh, Su-Lim Tan, Published by the IEEE Computer Society, ISBN 0272-1732/09.

[3] A. Garcia-Martinez, J. F. Conde, and A. Vina, ''A Comprehensive Approach in Perfor- mance Evaluation for Modern Real-Time Operating Systems,'' Proc. 22nd EuroMicro Conf., IEEE CS Press, 1996, p. 61.

[4] Using a Low-Cost SoC Computer and a CommercialRTOS in an Embedded Systems Design Course, James O. Hamblen, IEEE transactions on education, vol. 51, no. 3, august 2008

[5] Embedded Software Development With Ecos by Anthony J. Massa, 2003 Pearson Education

[6] K.M. Sacha, ''Measuring the Real-Time Operating System Performance,'' Proc. 7th EuroMicro Workshop Real-Time Systems, IEEE CS Press, 1995, pp. 34-40

[7] G. Buttazzo, Hard Real-Time Computing Systems (The International Series in Engineering and Computer Science), Kluwer Academic Publishers, ISBN 0792399943, 1997

[8] Embedded OS Benchmarking,Technical Document Amir Hossein Payberah, retrieved from www.sics.se/~amir/files /download/document/os_benchmarking.pdf

[9] Performance benchmarking of FreeRTOS and its Hardware Abstraction, Tao Xu, Technische Universiteit Eindhoven, November 2008.

[10] Introduction to Linux for Real Time Control, National Institute of Standards and Technology, prepared by Aeolean Inc. retrieved from, http://tornasol.datsi.fi.upm.es/ciclope-old/doc/rtos/cache/ doc/ realtimelinuxreport.pdf

[11] Performance Comparison of RTOS, Shamil Merchant and Kalpen Dedhia, Columbia University, unpublished.

[12] Impact of RTOS Parameters on End-to-End Timing Performance, Antino Kim and Kang G. Shin, University of Michigan, Ann Arbor, unpublished.

[13] Introduction to Linux for Real Time Control, National Institute of Standards and Technology, prepared by Aeolean Inc.