MEASURING RTOS PERFORMANCE: WHAT? WHY? HOW?

FAHEEM SHEIKH, SENIOR TECHNICAL LEAD, MENTOR GRAPHICS
DAN DRISCOLL, SOFTWARE ARCHITECT, MENTOR GRAPHICS

WHITEPAPER

Mentor Graphics®

EMBEDDED SOFTWARE

www.mentor.com

INTRODUCTION

In the world of smart phones and tablet PCs memory might be cheap, but in the more constrained universe of deeply embedded devices, it is still a precious resource. This is one of the many reasons why most 16- and 32-bit embedded designs rely on the services of a scalable real-time operating system (RTOS). An RTOS allows product designers to focus on the added value of their solution while delegating efficient resource (memory, peripheral, etc.) management. In addition to footprint advantages, an RTOS operates with a degree of determinism that is an essential requirement for a variety of embedded applications.

There are many different types of competing RTOS solutions available today: proprietary, commercial, and open source software offerings dominate the landscape. While most of these offerings provide similar feature sets, the question is "How do you determine which RTOS is right for you?" How can an embedded product designer with specific requirements for memory usage, deterministic responsiveness, and overall efficiency of a system decide which particular RTOS is best suited for his or her product? Quite often, instead of asking the right questions, it's the lack of understanding the answers that can lead to the wrong choice of software. Obviously, selecting the wrong RTOS can be detrimental to the entire product development effort.

This white paper takes a look at "typical" reported performance metrics for an RTOS in the embedded industry. It's an attempt to explain what these numbers signify, how they are measured, and why an embedded system designer should give particular attention to these numbers. Further, it's important to understand the potential pitfalls stemming from the improper interpretation of these performance metrics. For demonstrative purposes, this white paper will use the performance metrics of the Mentor Graphics Nucleus RTOS kernel. Throughout the paper, a set of guidelines will be provided to help formulate a well-informed comparison criterion.

"PROPAGATED" METRICS

A quick online survey of RTOS metrics maintained by third party consultants, students, researchers, and official records (of distributor companies) reveals that the following three categories are used to evaluate an RTOS solution:

- **Memory Footprint**
- **Latency**
- **Services Performance**

Among these measurements, footprint provides an estimated usage of memory by an RTOS on an embedded platform. The other two categories measure various types of RTOS overhead or runtime performance. Latency is reported in two different ways: interrupt and scheduling. And services performance is the minimum time taken by the RTOS interface to complete a system call. All of these metrics are explained in greater detail in the following sections.

Although independent benchmarks such as the Embedded Microprocessor Benchmark Consortium (EEMBC) are geared toward embedded devices, they have not been standardized by the RTOS vendors to report OS specific performance measurements. This is probably because benchmark algorithms are more suited to evaluate a particular hardware architecture or processor which means that the OS services used in these benchmarks are limited, and hence, cannot be relied upon for a complete analysis of the OS capabilities. For instance, a benchmark algorithm corresponding to networking market will incorporate packet checking and check sums calculations. Obviously this is not of much help to determine the OS overhead like interrupt or scheduling latency.

MEMORY FOOTPRINT

Memory footprint is an estimate of RAM and ROM requirements of an RTOS on a specific embedded platform. Effective code, read-only data of the kernel, and any runtime library code are all collectively part of the ROM size, which can be put into a Flash or any other read-only memory. This assists in booting directly into the RTOS image without any boot-loader. RAM requirements on the other hand are a sum of data structures and global variables.

HOW IS THE MEMORY FOOTPRINT MEASURED?

The memory foot print calculation is heavily dependent on:

- The **architecture for which the RTOS** is being compiled, since this hugely affects the number of instructions in the kernel image.
- The **software configuration** including which kernel components and runtime libraries are being utilized.
- Also of great importance are the **compiler optimizations** used to reduce the code size as much as possible (sometimes at the expense of performance). Normally the highest order of complier size optimization is used to gather these results.
- Across the compilers, there are different methods of obtaining the footprint information. These methods include:
  - Memory MAP files that list the sizes of the built image
  - Command-line tools (*objdump*) that will show footprint information for a selected executable image

WHY MEASURING THE MEMORY FOOTPRINT IS IMPORTANT

Footprint metrics are often an important decision factor when considering an RTOS solution, especially in situations where devices have limited on-chip memory and no possibility of interfacing with external memory. For large embedded designs with very complex applications, the footprint information for the kernel may be negligible compared to the overall memory requirements, but limited cache sizes or available on-chip memory, may still make the RTOS footprint an important decision point to meet performance requirements (the kernel can reside in on-chip memory or be locked into cache).

Another consideration is whether a boot-loader will be employed in the final design. When using a boot-loader, the kernel image will be contained in some type of non-volatile memory (sometimes compressed), so the ROM size is still important. Typically, the kernel image is copied to RAM during the boot process. This means that all of the kernel image (i.e. code and data) will occupy RAM during runtime.

COMMON PITFALLS WHEN ANALYZING THE DATA

- Ignoring how a minimum kernel configuration is defined. For some vendors, this minimum configuration only includes a very tiny subset of the services that would not constitute a “useful” set for the application.
- Most often runtime library functions are not included in memory calculations.
- In general, RAM-ROM sizes should always be reported in a min/max range, since RAM size depends on the application and ROM size as explained above is dependent on kernel configuration. This usually means having a minimum “useable” size and a maximum measure that shows the size of all services.

- Having the smallest size does not automatically guarantee a scalable RTOS solution with respect to memory. Sometimes, entirely different versions of an RTOS are marketed for low-end devices. This is a serious problem if configurability of solution is a key product requirement.

NUCLEUS KERNEL REFERENCE

The Nucleus RTOS kernel is a completely scalable solution with respect to memory. When compiled for an ARM Cortex A8 processor in ARM mode using the Mentor Sourcery CodeBench toolchain and full optimization for size, it results in a ROM size of 12 to 30 Kbytes and RAM requirements of a mere 500 bytes. The minimum footprint is calculated with all essential kernel services to include dynamic memory, threads, semaphores, events, and queues while excluding the runtime library. Maximum footprint, on the other hand, includes all kernel services. In general, compilation with Thumb-2 mode results in 35 percent reduction in ROM size, which means the Nucleus kernel can be contained in just 7.8KB of memory on any Cortex-M based controller.

INTERRUPT LATENCY

Interrupt latency is the measurement of system's response-time to an interrupt. Other, more explicit definitions exist for interrupt latency, but then each has a different perspective. Figure 1 explains how the same quantity can be defined and reported differently. From a system perspective, it is the time between interrupt assertion and the instant an observable response happens, shown as "Interrupt Latency" in Figure 1. From the OS perspective however, interrupt latency is measured as the duration of when the CPU was interrupted until the start of the corresponding interrupt service routine (ISR). Clearly this measure is actually the OS overhead to Interrupt Latency but in many RTOS data sheets you will find this overhead reported as the actual latency. This also prompts some vendors to claim a zero interrupt latency arguing that OS adds no additional overhead to interrupt processing.

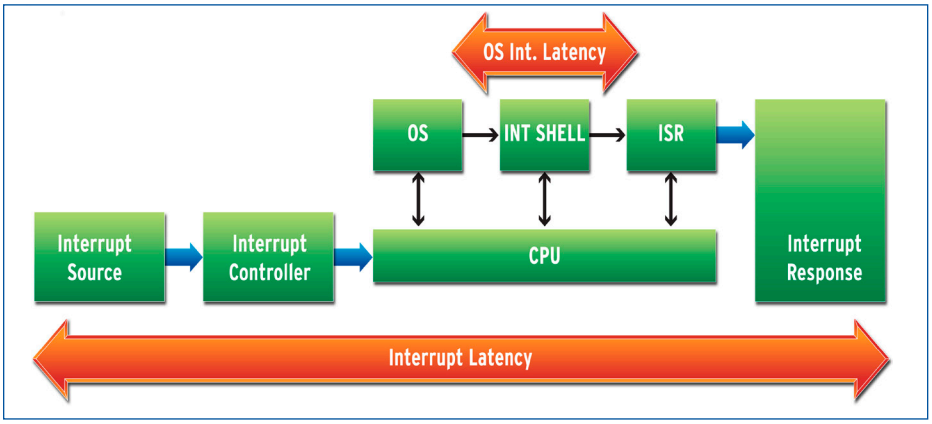

Figure 1: Interrupt latency is measured as the duration of when the CPU was interrupted until the start of the corresponding interrupt service routine (ISR).

HOW IS INTERRUPT LATENCY MEASURED?

From Figure 1, one can express the interrupt response time as a sum of two distinct times:

$$\tau_{IL} = \tau_H + \tau_{OS}$$

Where τ_H is the hardware dependent time contributing to interrupt latency which depends on the interrupt controller on the board as well as the type of the interrupt.

τ_{OS} is the OS induced overhead in processing the interrupt. This quantity has a best case and a worst case scenario worth considering. The best case situation is depicted in Figure 1 where only minimal overhead is added before the ISR starts. The worst case scenario occurs when the kernel has disabled interrupts to protect critical sections. In this case τ_{OS} is the sum of best case and the longest interrupt lockout time in the kernel.

WHY IS MEASURING INTERRUPT LATENCY IMPORTANT?

Generally all time-critical and fault-tolerant designs will rely on this metric. In addition, if your system relies on high I/O bandwidth (i.e., how many packets processed per second) you might be interested in first measuring the latency of that particular interrupt. Most other systems can live with interrupt latency in tens of microseconds.

COMMON PITFALLS WHEN ANALYZING THE DATA

One can see that there are multiple factors at play which impact the interrupt latency measurement. In general, the following guidelines might be helpful when you are interested in this metric:

Hardware Configuration:

- Which platform (architecture + vendor) was the measurement taken on?
- Speed of the CPU
- Cache configuration
- Timer frequency
- Interrupt controller type and its location on platform

OS Configuration:

- Where is the code running from, Flash, SRAM, SDRAM?
- Which interrupt has been used for measurement a GPIO, an on-chip timer?
- Has the code been optimally compiled for time?
- Does the metric correspond to the best or average case? (The worst case time is hardly reported for obvious reasons)

By far the best approach to verify interrupt latency is to record the time between interrupt source and its response on an oscilloscope. For instance, a GPIO pin can be used to generate an interrupt while another GPIO pin can be toggled at the start of an ISR. In this way a signal generator and an oscilloscope can be used to get very accurate interrupt latency information without excessive software or hardware set-up.

NUCLEUS KERNEL REFERENCE

Mentor's Nucleus RTOS kernel has an average interrupt latency of less than 0.5 microseconds when running from SDRAM on a 600MHZ ARM Cortex A8 processor.

SCHEDULING LATENCY

Scheduling latency is usually a measure of performance for the RTOS thread scheduler. Compared to interrupt latency there is even *more variation* in how scheduling latency is measured as well as the interpretation of the data. Generally speaking there are two distinct, but related quantities affecting scheduling latency, namely "context switch time" and "scheduling overhead."

Figure 2 describes one possibility of measuring context switch time. It shows a system in state 'A' with Task A running. Let us assume as a result of an event, interrupt, or an API call, the OS has to save the context of the current task and load another Task B context registers referred here as state 'B'. Another variation in scheduling latency measurement is the current state of the system. In Figure 2 this was state 'A' but it could be an idle system state to start with. Naturally, the overall metric is dependent on the initial state.

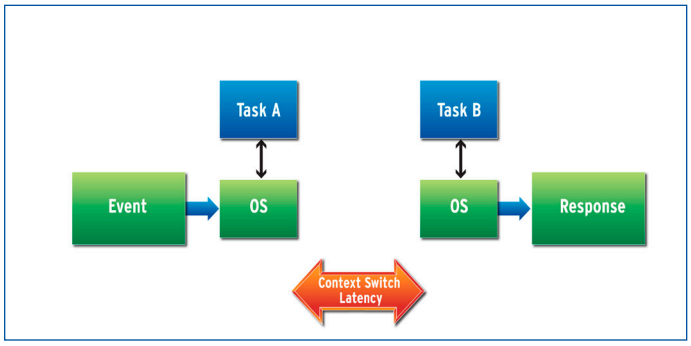

Figure 2: One possibility of measuring context switch latency.

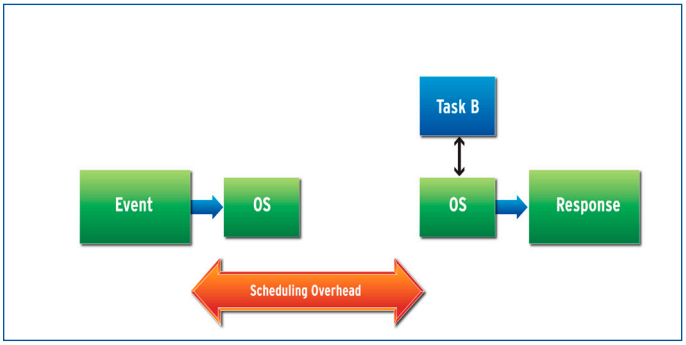

Figure 3: The expected scheduling overhead when a thread has to be scheduled or resumed as an interrupt response.

Figure 3 on the other hand, illustrates the expected scheduling overhead when a thread has to be scheduled/ resumed as an interrupt response. This can include the time it takes for the OS to decide which task to run and placing the current thread on the waiting queue. One can notice an overlap with context switch time, but this is a more specific way of measuring how quickly scheduler can respond to an interrupt.

HOW IS SCHEDULING LATENCY MEASURED?

In the worst case scenario, scheduling latency can be considered the maximum of the latencies describe above. That is:

$$\tau_{SL} = MAX\,(\tau_{SO}\,,\,\tau_{CS})$$

Where τ_{SO} is the scheduling overhead of the kernel measured from the end of an ISR to the start of task being scheduled...

...And τ_{CS} is the time taken by the kernel to save and restore a thread context.

WHY IS MEASURING SCHEDULING LATENCY IMPORTANT?

In general all systems with hard interrupt latency requirements will also be interested in this metric. Please refer to the *"Why is Measuring Interrupt Latency Important?"* section for examples.

COMMON PITFALLS WHEN ANALYZING THE DATA

- Ignoring the initial state of the system. Referring to Figure 2 again, if state A of the system is idle, like it is in Figure 3, there is no requirement for saving a context, just load a new task in which case the scheduling latency is low.
- Context switch latency is tightly dependent of the underlying architectures. Different processor versions can have varying context switch times because of the difference in register sets.
- It can also be impacted by kernel configuration and the compile time optimizations

NUCLEUS KERNEL REFERENCE

As a reference the Nucleus RTOS kernel, while running from SDRAM on a 600MHz ARM Cortex A8, reports scheduling latency of 1.3 microseconds.

TIMING KERNEL SERVICES

Another useful performance metric is the time it takes for the kernel to complete some of its essential services. Although an RTOS can potentially expose a long list of APIs to the application/system developer, it's the timing of most frequently used APIs that is generally of interest. Some of these services have been categorized below.

THREADING SERVICES

This is the RTOS supplied interface to create, start, resume, and stop a thread. If your embedded system needs to respond to numerous external events in real-time, chances are your multi-threaded application will rely on these services frequently during runtime and you should be interested in these numbers.

SYNCHRONIZATION SERVICES

This is the interface provided by the RTOS to synchronize between multiple programming contexts like an interrupt service routine and a thread. Another objective is to protect critical regions of the program from concurrent accesses. An example is a semaphore that may be used by Ethernet ISR to write packet data into a shared memory buffer that is also used by a task. If an embedded system is being designed with many shared resources or peripherals are going to be multiplexed for several uses these numbers from RTOS should be taken into consideration.

INTER-PROCESS COMMUNICATION SERVICES

These are services to share data between multiple threads. Examples include FIFOs, Queues, Mailboxes, and Events. Generally if an embedded system needs close coordination between different tasks to achieve a goal then the completion time of these services is relevant.

MEMORY SERVICES

Many embedded systems rely on the RTOS to efficiently manage memory with the evolution of runtime requirements. For these applications dynamic memory allocation/de-allocation timings will be important since they can significantly impact performance.

COMMON PITFALLS WHEN ANALYZING THE DATA

- The hardware architecture, processor type, maximum CPU frequency, and cache configuration plays a key role in shaping these numbers.
- Also important is which memory the code was running from SDRAM, ROM, SRAM?
- Most of the time measurements are taken with compiler optimizations set for maximum performance and often the RTOS is configured to have reduced error checking or similar functionality.

NUCLEUS KERNEL REFERENCE

The table below lists service timings for the Nucleus RTOS kernel. As with interrupt and scheduling latency, measurements are taken on an ARM Cortex A8 running on 600MHz with L1 cache of 16KB.

Nucleus RTOS Kernel Service	Time in microseconds
Task resumption	0.3
Task suspension	0.3
Obtaining a semaphore	0.5
Set an event	0.4
Send message to queue	0.9
Allocate memory	0.2
De-allocate memory	0.7
Allocate partition	0.4

CONCLUSION

While RTOS performance data provided by the software vendor can be useful, coincidently, it can be misleading if not viewed through a proper lens. A thorough understanding of the measurement techniques, terminology, and trade-offs involved is critical to conducting a fair comparison between different metrics. The recommendation for a system or application developer is to rely on a holistic measurement technique that makes sense with his platform, application, and environment.

About the Authors:

Faheem Sheikh joined the Embedded Systems Division of Mentor Graphics in 2007, where he is working as a senior technical lead. His current focus is software research and development for symmetric multiprocessor architectures. Faheem has a Masters (2005) and PhD (2009) in computer engineering from Lahore University of Management Sciences, Pakistan. He has more than ten technical publications in leading international conferences and journals.

Dan Driscoll is a software architect for the Nucleus RTOS and middleware products. He has worked in the Embedded Systems Division of Mentor Graphics for nearly ten years in a variety of roles. His background has been focused heavily in kernel and BSP development across numerous embedded architectures. Dan holds a BS degree in Computer Science from the United States Military Academy at West Point.

Corporate Headquarters
Mentor Graphics Corporation
8005 SW Boeckman Road
Wilsonville, OR 97070-7777
Phone: 503.685.7000
Fax: 503.685.1204

Sales and Product Information
Phone: 800.547.3000
sales_info@mentor.com

Silicon Valley
Mentor Graphics Corporation
46871 Bayside Parkway
Fremont, CA 94538 USA
Phone: 510.354.7400
Fax: 510.354.7467

North American Support Center
Phone: 800.547.4303

Europe
Mentor Graphics Deutschland GmbH
Arnulfstrasse 201
80634 Munich
Germany
Phone: +49.89.57096.0
Fax: +49.89.57096.400

Pacific Rim
Mentor Graphics (Taiwan)
Room 1001, 10F
International Trade Building
No. 333, Section 1, Keelung Road
Taipei, Taiwan, ROC
Phone: 886.2.87252000
Fax: 886.2.27576027

Japan
Mentor Graphics Japan Co., Ltd.
Gotenyama Garden
7-35, Kita-Shinagawa 4-chome
Shinagawa-Ku, Tokyo 140-0001
Japan
Phone: +81.3.5488.3033
Fax: +81.3.5488.3004

MGC 07-11 TECH10080-w