expresslogic

# Measuring Real-Time Performance of an RTOS

Real-time performance generally is agreed to be one of the most important criteria considered by developers when selecting an RTOS for embedded applications. Embedded market researchers Evans Data Corporation and Venture Development Corporation (VDC) recently found that “real-time performance” ranked #1 among factors considered by developers when selecting a commercial real-time operating system. But, just what is “real-time performance,” and how is it measured?

**Real-time performance**
Real-time performance can be defined as the speed with which an RTOS (or any software for that matter) can complete its functions in response to an event. The “real-time” aspect implies that the software response to one event is needed before some independently occurring second event takes place. For example, in response to an automobile engine’s intake valve opening, the engine control software must calculate the correct air-fuel mixture and have it injected into the cylinder before the valve closes in preparation for the compression stroke. It is critical that the response to the first event is completed in time to meet the needs of the second event. This response may include many things, but paramount among them are *interrupt processing* and *system services*.

**Why is performance so critical?**
The time required for completion of these functions is particularly critical in real-time systems, which must be deterministic, and also must respond rapidly or suffer loss of data or even fundamental malfunction of the system. An example might be a flight-control system that must respond to a pilot input in time to avoid a stall, or a disk-drive controller that must stop the drive’s read head at precisely the point at which data is to be read or written. Rapid-fire interrupts from high-speed data packet arrival into a DSL router also must be handled promptly to avoid triggering a retry because one was missed.

Processor speed is critical in executing all of the RTOS instructions required to perform the desired function, but brute force alone cannot satisfy system demands, nor can it provide the most economical or efficient solution. While a 2GHz processor might breeze through code in satisfactory time, it also might be too costly, or draw too much power, or present physical packaging challenges that make it undesirable for some embedded applications. A more economical processor, running an efficient RTOS, might do just as well in performance, or even better, yet cost far less and not pose the power/heat/packaging problems of a faster processor.

How is real-time performance measured? By focusing on the most significant elements of performance, a reasonable assessment of real-time performance can be measured in a rigorous fashion that lends itself to comparisons between multiple RTOSes on a common hardware platform. Thus, by comparing each RTOS according to how well it performs specific critical functions, developers can quantify real-time performance and make the best decision according to their application needs.

**Interrupt Processing**
Real-time systems are generally designed to be reactive in nature, and the events to which they must react are generally made known to the system as interrupts. The processor, upon recognizing an interrupt, performs certain actions and executes instructions that were designed to react to this event. In most cases, the processor is already performing some instructions immediately prior to recognizing the interrupt. This processing must be "interrupted," and then later resumed when the critical real-time response of the interrupt has been completed. Most RTOSes are designed to provide a means for the developer to handle interrupt processing and also to schedule and manage execution of application software threads. Interrupt processing generally includes the following:

- suspending whatever thread currently is executing,
- saving thread-related data that will be needed when the thread is resumed,
- transferring control to an interrupt service routine (ISR),
- performing some amount of processing in the ISR to determine exactly what action is needed,
- retrieving/saving any critical (incoming) data associated with the interrupt,
- setting any required device-specific (output) values,
- determining which thread should now execute given the change in environment created by the interrupt and its processing,
- clearing the interrupt hardware to allow the next interrupt to be recognized,
- transferring control to the selected thread, including retrieval of any of its environment data that was saved when it was last interrupted.

Interrupt processing is only one aspect of real-time performance. Implementation of these operations in a particular RTOS can make a significant difference in real-time performance.

**System Services**
Real-time operating systems must do more than respond to interrupts. They also must schedule and manage the execution of application software threads. The RTOS handles requests from threads to perform scheduling, message passing, resource allocation, and many other services. In most instances, services must be performed quickly, so the thread can complete its assigned processing before the occurrence of the next interrupt. While not a part of interrupt processing, the system service is a critical real-time response that can make or break a system. System service processing includes the following:

- scheduling a task or thread to run upon the occurrence of some future event,
- passing a message from one thread to another,
- claiming a resource from a common pool,

Even more variable than interrupt processing, the implementation of system services is equally critical in achieving good real-time performance in an RTOS. Together with interrupt processing, system services combine to form the most significant processing that an RTOS is asked to perform. Different RTOS implementations will approach these functions differently, and with different architectures, producing a wide range of performance. System services generally are implemented in one of two ways in an RTOS, by "Trap" or by "Call." Each method has certain characteristics:

**Trap**
- Uses unimplemented instruction trap to force interrupt
- ISR examines parameters and transfers to appropriate RTOS service

- Similar to debugger software-breakpoint technique
- Locks out interrupts for a time

**Function Call**

- Uses processor branch instruction, no interrupts
- Low overhead
- Requires linking

RTOS Performance is platform-sensitive, processor-sensitive, clock-speed sensitive, compiler-sensitive, and design-sensitive. Performance also is "context sensitive." What exactly is being measured, and how is it being measured? In order to make fair performance comparisons among various RTOSes, it is important to compare "apples to apples." That means assuring that, to the greatest extent possible, all measurement sensitivities are identical across the RTOSes being measured.

**The Thread-Metric Benchmark Suite**

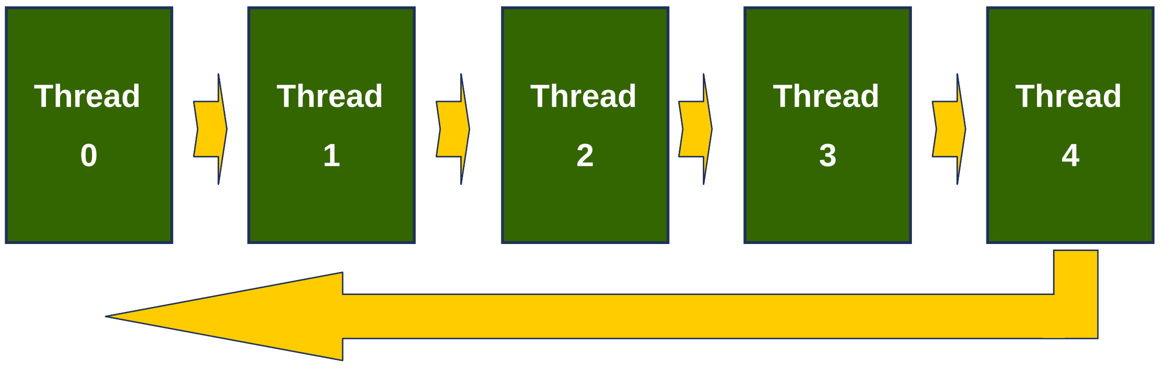
"Thread-Metric," is a free-source benchmark suite for measuring RTOS performance. The Thread-Metric suite is freely available from Express Logic and other web sites. See below for currently available links. Thread-Metric has been designed to be easily adapted to any RTOS. To be applicable to multiple RTOSes, for comparison, a set of common services has been selected. The selected services are commonly found among most RTOSes:

1. **Cooperative Scheduling**
2. **Preemptive Scheduling**
3. **Interrupt Processing**
4. **Interrupt Preemption Processing**
5. **Message Passing**
6. **Synchronization Processing**
7. **RTOS Memory Allocation**

Each service type is described in the following pages, with a diagram representing the flow of the Thread-Metric test program for that service.

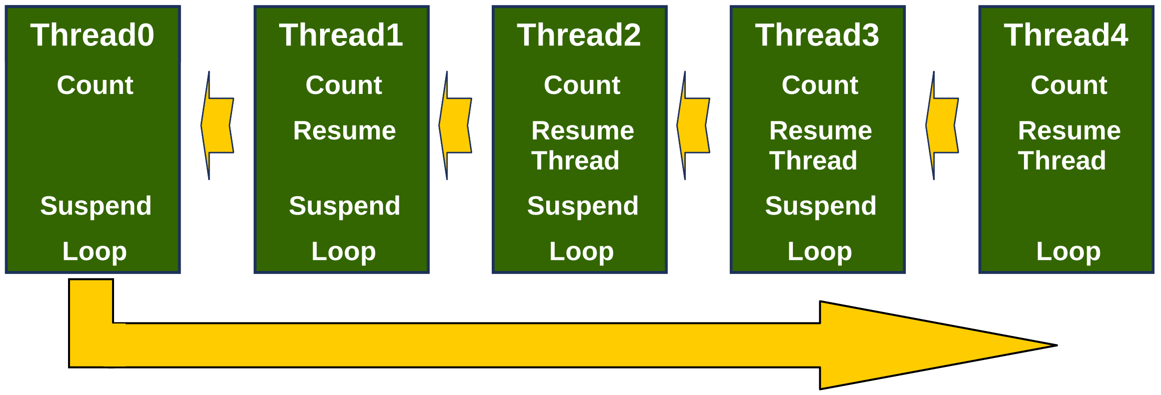
## 1. Cooperative Scheduling Test

This test consists of 5 threads created at the same priority that voluntarily release control to each other in a round-robin fashion. Each thread will increment its run counter and then relinquish to the next thread. At the end of the test the counters will be verified to make sure they are valid (should all be within 1 of the same value). If valid, the numbers will be summed and presented as the result of the cooperative scheduling test.



## 2. Preemptive Scheduling Test

This test consists of 5 threads that each have a unique priority. Thread0 is the highest, and Thread4 the lowest. In this test, all threads except the lowest priority thread (Thread4) are left in a suspended state. The lowest priority thread will resume the next highest priority thread. That thread will resume the next highest priority thread and so on until the highest priority thread (Thread0) executes. Each thread will increment its run count and then call thread suspend. Eventually the processing will return to the lowest priority thread, which is still in the middle of the thread resume call. Once processing returns to the lowest priority thread, it will increment its run counter and once again resume the next highest priority thread - starting the whole process over once again.



**Interrupt Processing**
There are two components to Interrupt Processing: Latency and Task/Thread Activation. Each must be considered to fully understand how responsive the RTOS is. Interrupt Latency ("Time to ISR") is the time required for the RTOS to respond to an interrupt. In this case, the time to get to the first instruction of an ISR. Task Activation Overhead ("Time to Task/Thread") is the time required to get to a thread/task that is activated or scheduled in response to the interrupt. Quite often, the processing required in response to an interrupt is more extensive than would be desirable to include in an ISR. As a result, the application designer might keep the ISR minimal, and plan to perform the remaining processing in an application task/thread. That task/thread is normally "scheduled" within the ISR, and its activation is managed by the RTOS scheduler, based on its priority.

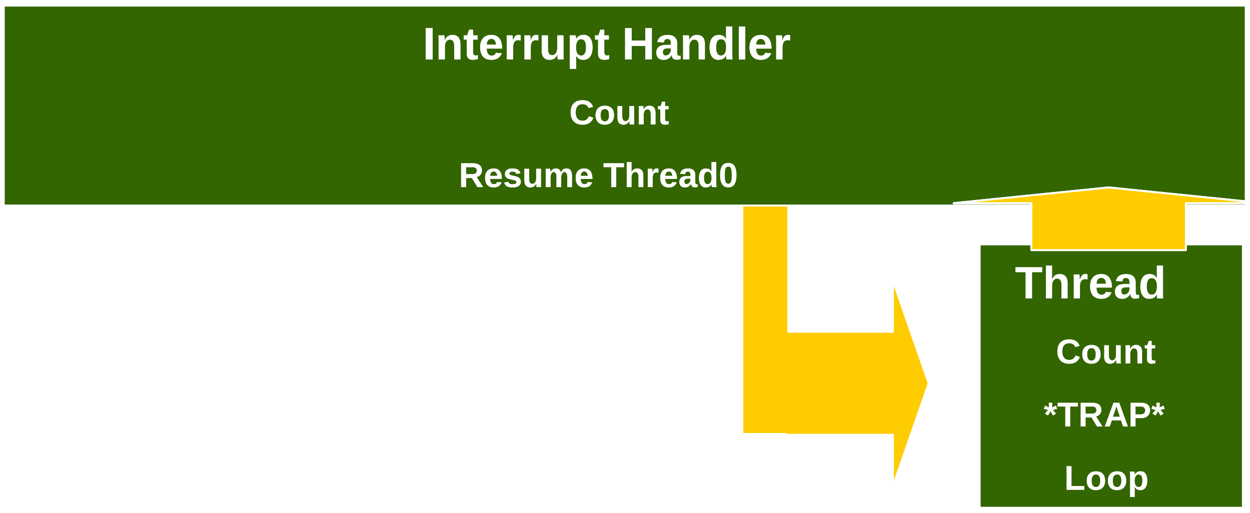
It's interesting to note that an RTOS might optimize response to interrupts by never disabling them. Thus, getting to the ISR will be very responsive. But, as a consequence, such RTOSes generally suspend task/thread scheduling, to protect critical sections and prevent other interference. This can lead to a "Hurry Up and Wait" situation. Some questions to consider are:

- Can OS services be called from ISR? If not, then they must be called from a task/thread, which must be activated before any such services can be used by the application;

- Is there "Split-Level" Interrupt Handling? In such cases, a Low level ISR responds to hardware interrupts, while a High Level ISR/Task calls OS services, and a Task performs processing. This introduces additional overhead in interrupt processing.

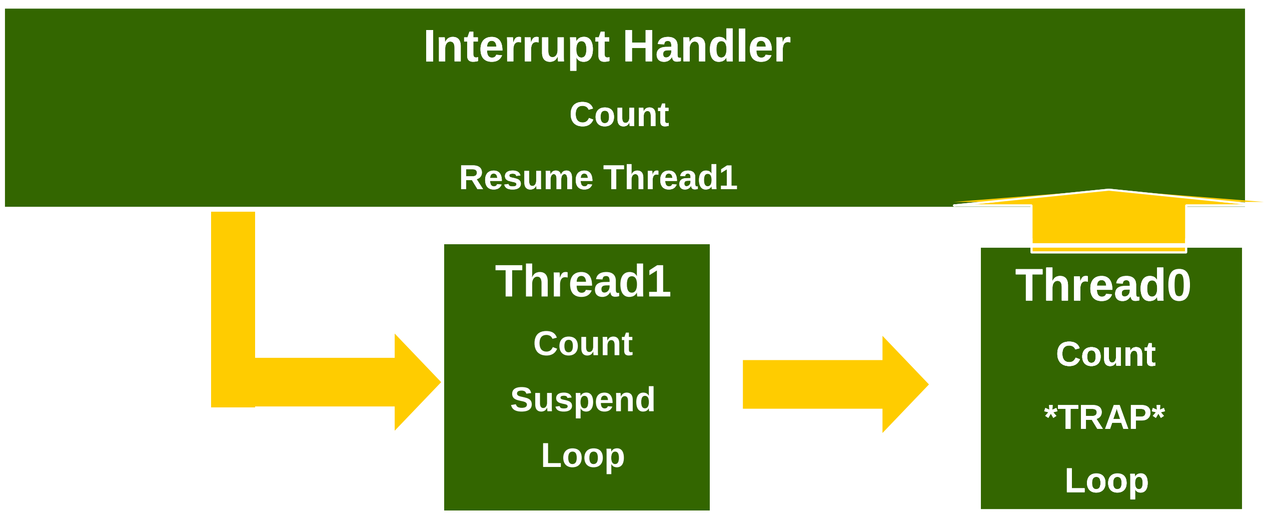
3. **Interrupt Processing Test**

This test consists of a single thread. The thread will cause an interrupt (typically implemented as a trap), which will result in a call to the interrupt handler. The interrupt handler will increment a counter and then post to a semaphore. After the interrupt handler completes, processing returns to the test thread that initiated the interrupt. The thread then retrieves the semaphore set by the interrupt handler, increments a counter and then generates another interrupt.



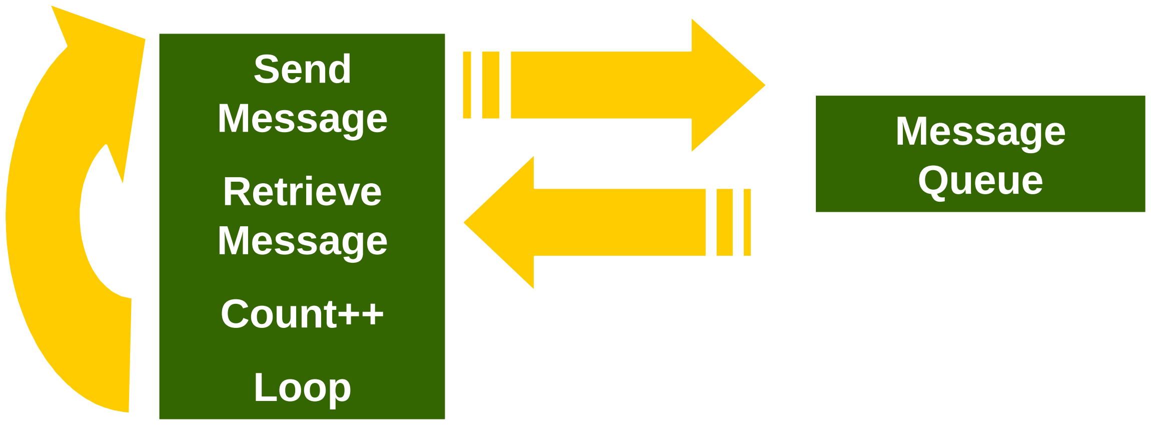
## 4. Interrupt Preemption Processing Test

This test is similar to the previous interrupt test. The big difference is the interrupt handler in this test resumes a higher priority thread, Thread1, which causes thread preemption. Thread1 then suspends, resuming Thread0.



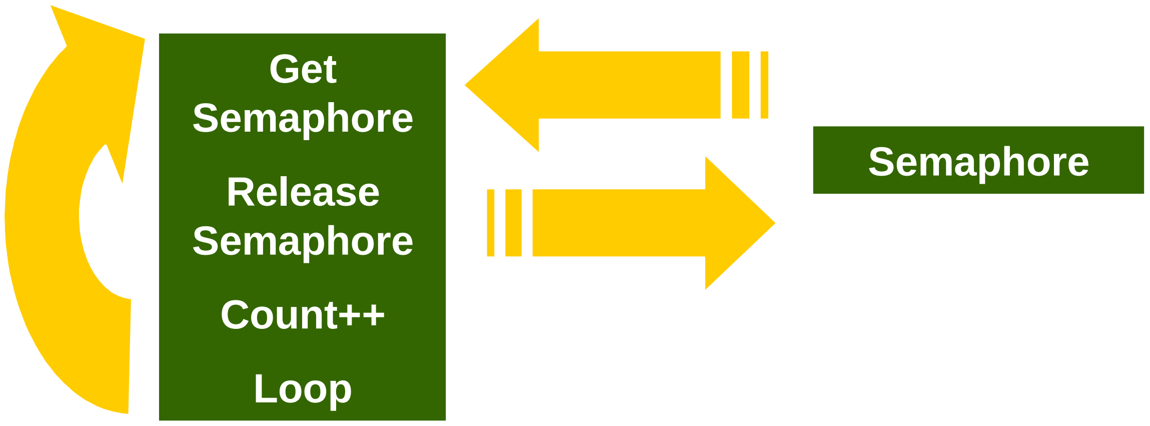
## 5. Message Passing Test

This test consists of a thread sending a 16 byte message to a queue and retrieving the same 16 byte message from the queue. After the send/receive sequence is complete, the thread will increment its run counter.



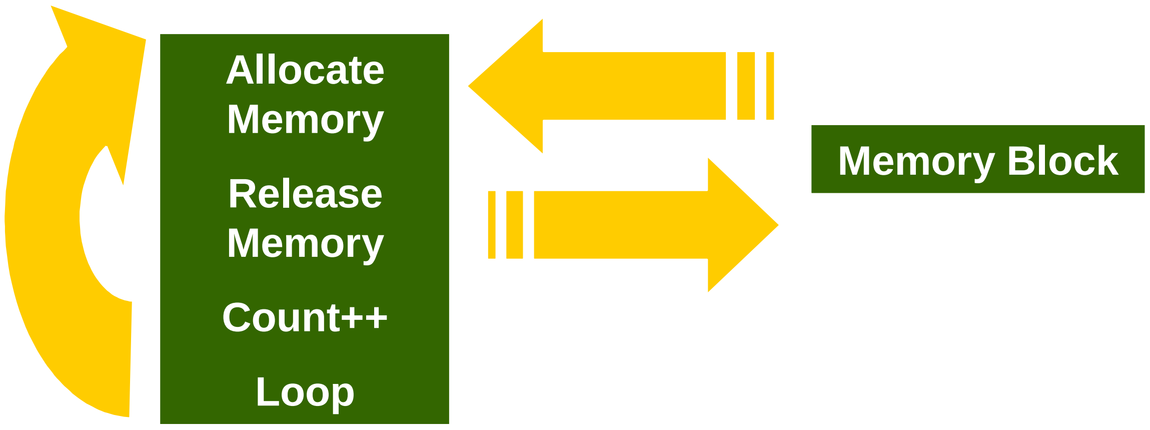
**6. Synchronization Processing Test**

This test consists of a thread getting a semaphore and then immediately releasing it. After the get/put cycle completes, the thread will increment its run counter.



**7. Memory Allocation/Deallocation**

This test consists of a thread allocating a 128-byte block and releasing the same block. After the block is released, the thread will increment its run counter.



**Thread-Metric RTOS Adaptation Layer (tm_porting_layer.c)**
As for the porting layer defined in tm_porting_layer.c, this file contain shell services of the generic RTOS services used by the actual tests. The shell services provide the mapping between the tests and the underlying RTOS. The following generic API's are required to map any RTOS to the performance measurement tests:

```
void  tm_initialize(void (*test_initialization_function)(void));
```
This function is typically called by the application from its main() function. It is responsible for providing all the RTOS initialization, calling the test initialization function as specified, and then starting the RTOS.

```
int  tm_thread_create(int thread_id, int priority, void
(*entry_function)(void));
```
This function creates a thread of the specified priority where 1 is the highest and 16 is the lowest. If successful, TM_SUCCESS returned. If an error occurs, TM_ERROR is returned. The created thread is not started.

```
int  tm_thread_resume(int thread_id);
```
This function resumes the previously created thread specified by thread_id. If successful, a TM_SUCCESS is returned.

```
int  tm_thread_suspend(int thread_id);
```
This function suspend the previously created thread specified by thread_id. If successful, a TM_SUCCESS is returned.

```
void  tm_thread_relinquish(void);
```
This function lets all other threads of same priority execute before the calling thread runs again.

```
void  tm_thread_sleep(int seconds);
```
This function suspends the calling thread for the specified number of seconds.

```
int  tm_queue_create(int queue_id);
```
This function creates a queue with a capacity to hold at least one 16-byte message. If successful, a TM_SUCCESS is returned.

```
int  tm_queue_send(int queue_id, unsigned long *message_ptr);
```
This function sends a message to the previously created queue. If successful, a TM_SUCCESS is returned.

```
int  tm_queue_receive(int queue_id, unsigned long *message_ptr);
```
This function receives a message from the previously created queue. If successful, a TM_SUCCESS is returned.

```
int  tm_semaphore_create(int semaphore_id);
```
This function creates a binary semaphore. If successful, a TM_SUCCESS is returned.

```
int  tm_semaphore_get(int semaphore_id);
```

This function gets the previously created binary semaphore. If successful, a TM_SUCCESS is returned.

```
int  tm_semaphore_put(int semaphore_id);
```
This function puts the previously created binary semaphore. If successful, a TM_SUCCESS is returned.

```
int  tm_memory_pool_create(int pool_id);
```
This function creates a memory pool able to satisfy at least one 128-byte block of memory. If successful, a TM_SUCCESS is returned.

```
int  tm_memory_pool_allocate(int pool_id, unsigned char **memory_ptr);
```
This function allocates a 128-byte block of memory from the previously created memory pool. If successful, a TM_SUCCESS is returned along with the pointer to the allocated memory in the "memory_ptr" variable.

```
int  tm_memory_pool_deallocate(int pool_id, unsigned char
*memory_ptr);
```
This function releases the previously allocated 128-byte block of memory. If successful, a TM_SUCCESS is returned.

**Porting Requirements**
The following requirements are made in order to ensure fair benchmarks are achieved on each RTOS performing the test:

1. The time period should be 30 seconds. This will ensure the printf processing in the reporting thread is insignificant.

2. The porting layer services are implemented inside of tm_porting_layer.c and NOT as macros.

3. The tm_thread_sleep service is implemented by a 10ms RTOS periodic interrupt source.

4. Locking regions of the tests and/or the RTOS in cache is not allowed.

**How Is the benchmark run?**
The Thread-Metric benchmark must be run in accordance with the instructions, so that results may be compared across different RTOSes, and different underlying hardware resources. The way this is achieved is through an initial Calibration Test, to measure processing speed of the hardware and compiler. This establishes a baseline for each benchmark platform, and Thread-Metric iterations can be compared, normalized to a common platform.

Each Thread-Metric benchmark test is run repeatedly, completing many iterations, for a 30-second interval. Iteration counters are maintained, and “printed” (through the debugger to the host) after 30-seconds. With this approach:

- No special hardware is required, as the calibration accounts for normalization of basic processing speed.
- Thread-Metric is easily ported to new environments, and multiple RTOSes
  - Thread-Metric is coded in C, and has been tested with various compilers, including IAR Embedded Workbench for ARM (recommended)
  - An RTOS Adaptation Layer is provided, with functions identified for adaptation to a specific RTOS API.

**Summary**
Thread-Metric is a useful means of measuring and comparing RTOS performance over a set of common RTOS services. The source code for Thread-Metric, including READ_ME notes, may be downloaded from:

- http://rtos.com/DownloadCenter/Thread-MetricForm.php

Or, contact Express Logic at: info@expresslogic.com