To find the RTOS with the best real-time performance, you've got to do an apples-to-apples comparison.

Measure your RTOS's real-time performance

BY WILLIAM LAMIE AND JOHN CARBONE

Developers are making embedded systems applications more complex, not only because they can, but because they have to, to remain competitive.To manage those complex applications in systems that use modern 32- and 64-bit processors, developers are turning to the real-time operating system (RTOS). The RTOS provides basic thread scheduling, communication, and resource allocation services, insulating the application from these details with a relatively simple application programming interface (API). If using a 32- to 64-bit processor, how do developers of these complex applications determine if an RTOS is the best move for their system, and if so, which RTOS?

Because embedded systems typically have limited resources, the challenge for developers is to assess the amount of overhead introduced by an RTOS to determine if its services are worth the additional performance cost. Concretely determining performance costs can be very difficult. Although developers generally agree that real-time performance is one of the most important criteria to consider when selecting an RTOS for embedded applications, not all agree on what "real-time capability" means and how to measure it.

The *2006 Embedded Systems Design State of Embedded Market Survey* (available at *www.embedded.com/columns/survey*) found that "real-time capability" ranked first among factors considered by developers when selecting an operating system, as shown in Figure 1. This survey was taken by embedded systems development engineers who attended the Embedded System Conferences in 2005 or subscribed to *EE Times* or *Embedded Systems Design*. Although these an-

s

e

nual surveys show that developers consider many other factors important, year after year real-time performance ranks at the top of the list.

Real-time systems must respond rapidly or suffer loss of data or even a fundamental system failure. For example, a jet fighter's flight-control system must respond to a pilot's input in time to initiate an evasive maneuver, or a disk-drive controller 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 packets arriving into a DSL router also must be handled promptly to avoid triggering a retry because one interrupt (and, thus, a packet) was missed.

Processor speed is critical in executing the RTOS instructions required to perform any of its services, but brute force can't succeed because of cost and power. For pure performance, yes; but that performance comes at a price not acceptable in most embedded systems. While a 2-GHz processor might breeze through code in satisfactory time, it might be too costly, draw too much power, or present physical packaging challenges that make it undesirable for some embedded applications.

How far can system designers reduce processor performance before it can no longer handle the system's real-time demands?

How far can system designers reduce processor performance before it can no longer handle the system's real-time demands? A big part of that answer lies in knowing the amount of processor cycles required by the RTOS. How can one determine just how much an RTOS hurts system performance?

A benchmark suite that exercises the RTOS across its many functions would enable developers to make a reasonable assessment of the operating system's real-time behavior. That suite must be vendor-neutral and use consistent code base to make fair comparisons among RTOSes. Vendor-supplied measurements are inadequate for making comparisons, as they're likely to be based on different code and might actually measure different ranges of events while calling them the same thing.

INTERRUPT PROCESSING

Real-time systems are generally reactive in nature, usually responding to external events via interrupts. The hardware services the event by transferring control to an RTOS-supplied or user-supplied interrupt service routine (ISR). An RTOS typically would save the context of the interrupted thread and service the highest-priority thread (perhaps the interrupted thread, but perhaps a new thread made ready by the ISR)

This survey question was asked of embedded systems development engineers.

Which factors most influenced your decision to use a commercial operating system?

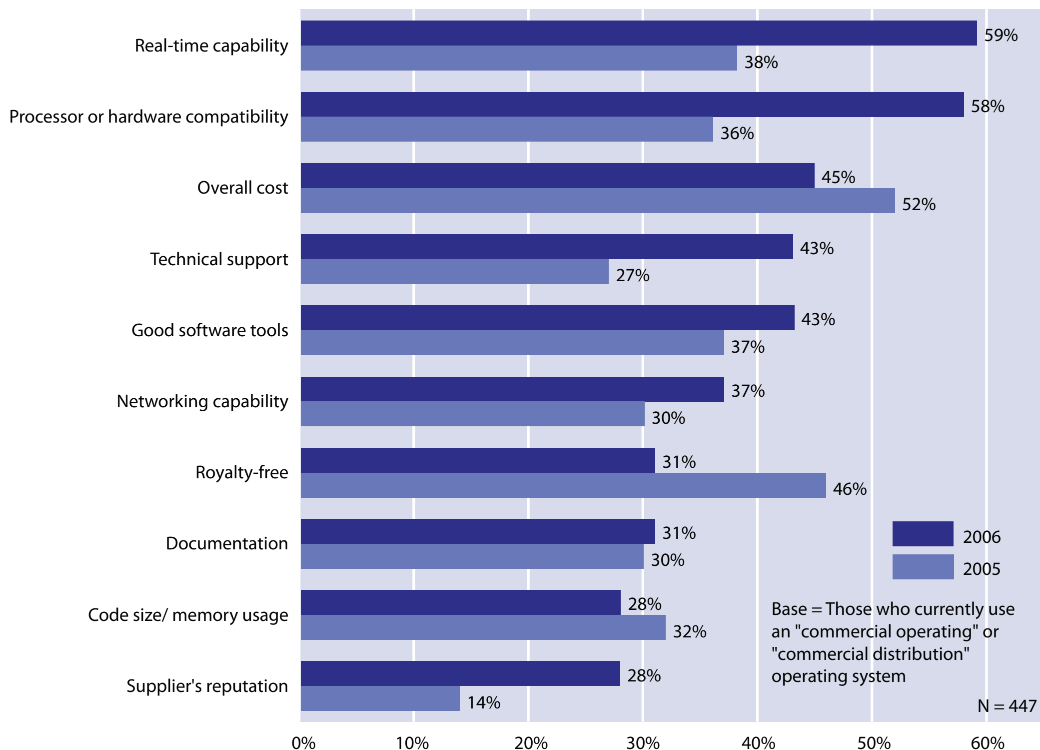

Figure 1

upon completion of the ISR.

Interrupt processing generally:

- Suspends the active thread.
- Saves thread-related data that will be needed when the thread is resumed.
- Transfers control to an ISR.
- Performs some processing in the ISR to determine what action is needed.
- Retrieves and saves any critical (incoming) data associated with the interrupt.
- Sets any required device-specific (output) values.
- Determines which thread should execute given the change in environment created by the interrupt and its processing.
- Clears the interrupt hardware to allow the next interrupt to be recognized.
- Transfers control to the selected thread, including retrieval of its environmental data that was saved when it was last interrupted.

It's no wonder that implementing these operations in a particular RTOS can make a significant difference in real-time performance. What's needed is a consistent, if not ideal, set of measurements that can be made in conjunction with various RTOSes under consideration.

SYSTEM SERVICES

RTOSes must also schedule and manage the execution of application software tasks or threads. The RTOS handles requests from threads to perform scheduling, message passing, resource allocation, and other services. Services must be performed quickly so the thread can resume. System service processing includes:

- 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.

RTOS services are even more variable than interrupt processing implementations. At least ISRs are within the control of the user; developers, however, can't control RTOS services as well, because they're supplied by the RTOS vendor. Several services might be called the same thing by different RTOS vendors but actually perform completely different operations.

The Thread-Metric Benchmark Suite is an open-source, vendor-neutral, free benchmark suite that measures RTOS performance.

Nevertheless, the implementation of system services is equally critical in achieving good real-time performance in an RTOS. Interrupt processing and system services are together the most significant processing that an RTOS must perform. Different RTOS implementations will approach these functions differently and with different hardware architectures, producing a wide range of performance. Once again, vendor measurements aren't entirely suitable for comparison, given the high likelihood of variations in the actual service being measured in each case.

OTHER MEASUREMENTS

Additional time will be taken if control is transferred from the application program (making the request) to the system service that performs the work. There are two common control transfer methods employed by RTOSes—Trap and Call. Each offers advantages and disadvantages.

Trap is just a software interrupt that invokes an ISR that requests the appropriate RTOS service; the ISR examines parameters and transfers to the appropriate RTOS service; it's similar to debugger software-breakpoint technique; and it may lock out interrupts for a time. *Call* uses processor branch instructions and no interrupts; it has a low overhead and requires linking.

RTOS performance is also sensitive to platform, processor, clock-speed, compiler, and design. To gain an apples-to-apples comparison, each of these factors must be held constant between tests of two different RTOSes.

A FREE BENCHMARK SUITE

The Thread-Metric Benchmark Suite is an open-source, vendor-neutral, free benchmark suite that measures RTOS performance. The tool can be applied to systems running an RTOS on single-core, multicore, or multithreaded architectures, using RTOS facilities to manage the hardware, while presenting a consistent API to the application. By using Thread-Metric, developers can compare RTOSes on any given architecture and get an apples-to-apples comparison. Developers can compare different RTOSes by measuring each RTOS's overhead on a given processor or can compare processor performance by running the same RTOS on different architectures.

Thread-Metric includes benchmark programs that exercise basic RTOS interrupt processing and system services (context switching, message passing, memory allocation, and synchronization). Although not exhaustive, this set of services is generally found in all RTOSes. Any attempt to make an exhaustive test would run the danger of introducing services that not all RTOSes might provide. The chosen set of services is likely to be found in all RTOSes, making the tool a widely applicable benchmark suite for evaluating an RTOS.

Thread-Metric consists of the following benchmarks, each of which measures a particular aspect of RTOS performance:

- Cooperative context switching
- Preemptive context switching
- Interrupt processing
- Interrupt processing with preemption
- Message passing

Cooperative context switching test measures the time the RTOS scheduler takes to change the execution context from one thread to another at the same priority.

- Semaphore processing
- Memory allocation and deallocation

Here's what each test consists of:

- **Cooperative context switching test** (see Figure 2) measures the time the RTOS scheduler takes to change the execution context from one thread to another at the same priority.
  1. Five threads are created at the same priority.
  2. Each of the five threads sits in an endless loop, calling the relinquish service.

- **Preemptive context switching test** (see Figure 3) measures the time the RTOS takes to change the execution context from one thread to a higher-priority thread,which preempts the executing thread.
  1. Five threads are created that each have a unique priority.
  2. The threads run until preempted by a higher-priority thread.
  3. All threads except the lowest-priority thread are in a suspended state.
  4. The lowest-priority thread resumes the next highest-priority thread, and so on until the highest-priority thread executes.
  5. Each thread increments its run count and then suspends.
  6. Once processing returns to the lowest-priority thread, it incre-

Cooperative context switching: each thread does not run to completion but sits in an endless loop calling the relinquish service.

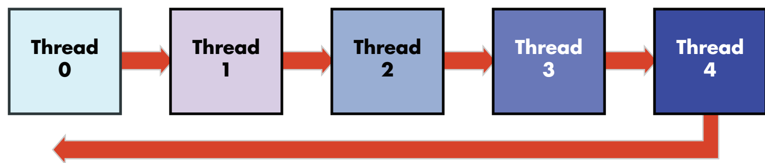

Figure 2

Preemptive context switching: each thread resumes a higher-priority thread, causing that higher-priority thread to be run, preempting the lower-priority thread and causing a context switch.

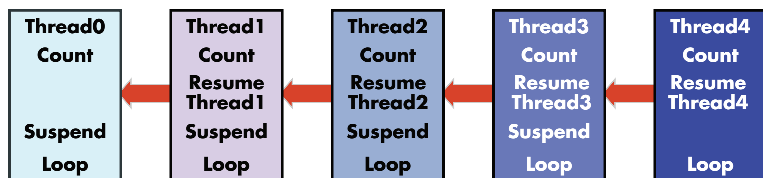

Figure 3

ments its run counter and again resumes the next highest-priority thread, starting the process over again.

- **Interrupt processing** is the combined time to get to the ISR (time interrupts are disabled), plus the time to perform the ISR, and the time to go through the scheduler and determine which thread now should run and then to enable that thread to run, restoring its context if necessary. In these tests, the threads use software interrupts to trigger preemption, and the reporting thread "prints" results every 30 seconds.
  - **Interrupt processing without preemption test.** In this first case (see Figure 4), it's non-preemptive, meaning the interrupted thread is resumed. The total time measured is the total of interrupt disabled time, ISR time, and scheduler time. Context save/restore may or may not be performed depending on the RTOS. Many RTOSes advertise "interrupt latency" but fail to account for "thread latency." Both are critical, and the total is what really matters.
    - Considers two components, interrupt latency (time to ISR) and task activation overhead (time to task).
    - Measures how long interrupts are disabled.
    - Measures how quickly the highest-priority thread (in this case, still the interrupted thread) can be activated.
      1 An interrupt is created as Thread5 executes a trap instruction [machine dependent].
      2. The interrupt handler executes and Thread5 is resumed, with no change of context.
  - **Interrupt processing with preemption test** (see Figure 5) measures the time it takes in situations where a different thread is to run after the interrupt, rather than simply a resumption of the interrupted thread. This case occurs when the ISR results in making a higher-priority thread ready to run. The time to perform a context save of the interrupted thread and a restore of the context for the new thread are included in the test score.
    1. Thread1 suspends, al-

Interrupt processing: Thread1 suspends itself allowing Thread5 (a lower-priority thread) to run. An interrupt is created as Thread5 executes a trap instruction [machine dependent]. The interrupt handler runs a different thread rather than a resume the interrupted thread. That involves a context save of the interrupted thread and a restore of the context for the new thread.

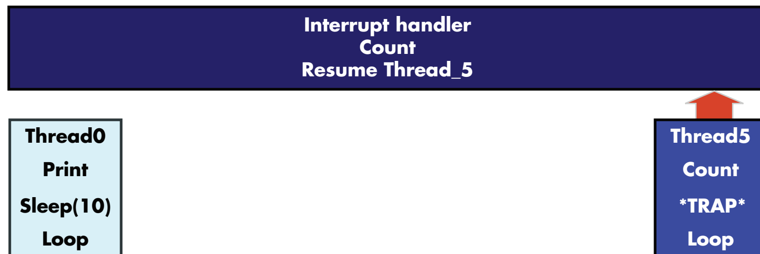

Figure 4

Interrupt processing: Thread1 suspends itself allowing Thread5 (a lower-priority thread) to run. An interrupt is created as Thread5 executes a trap instruction (machine dependent). The interrupt handler resumes Thread1, and the scheduler switches context to the higher-priority Thread1.

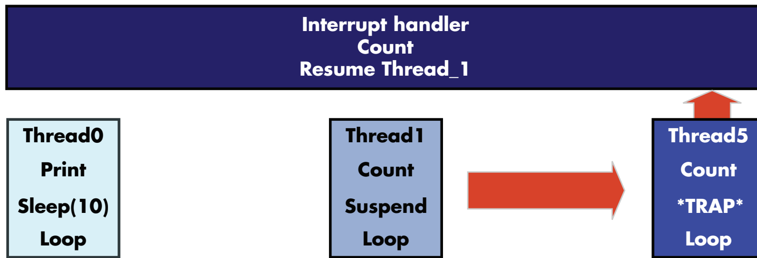

Figure 5

Message passing: a message is sent to a queue, then retrieved from the queue in an endless loop.

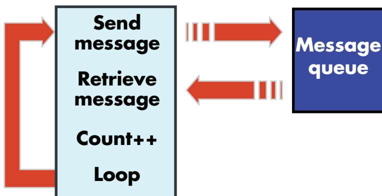

Figure 6

Semaphore processing: a thread gets a semaphore and then releases it in an endless loop.

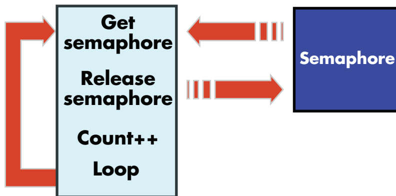

Figure 7

Memory allocation: memory is allocated and deallocated in an endless loop.

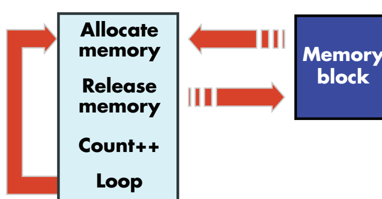

Figure 8

lowing Thread5 to run.

2. Thread5 forces an interrupt.
3. The interrupt handler resumes Thread1.
4. The scheduler changes context from Thread5 to Thread1, and Thread1 runs.

- **Message passing test** (see Figure 6) measures the time it takes a typical 16-byte message to be passed by value, in other words, copied from the source into the queue and then back out to the receiver. Messages larger than 16 bytes would typically be placed in allocated memory and then passed by reference (in other words, its pointer would be sent to the queue, not the entire message).
  1. A thread sends a 16-byte message to a queue and retrieves the same 16-byte message from the queue.
  2. After the send/receive sequence completes, the thread increments its run counter.

- **Semaphore processing test** (see Figure 7) measures the time it takes the RTOS to "get" or "put" a semaphore.
  1. A thread gets a semaphore, then immediately releases it.
  2. After the get/release cycle completes, the thread increments its run counter.

- **Memory allocation and deallocation test** (see Figure 8) measures the time it takes the RTOS to allocate a fixed-size block of memory for a thread. This test is designed to test memory allocation of 128-byte blocks. If an RTOS doesn't have fixed-block memory management, then a malloc-type service would have to be used and would likely be slower.
  1. A thread allocates a 128-byte block of memory and releases the same block.
  3. After the block is released, the thread increments its run counter.

Listing 1 An example of code from the message passing test.

```
while(1)
     {

         /* Send a message to the queue.  */
         tm_queue_send(0, tm_message_sent);

         /* Receive a message from the queue.  */
         tm_queue_receive(0, tm_message_received);

         /* Check for invalid message.  */
         if (tm_message_received[3] != tm_message_sent[3])
              break;

         /* Increment the last word of the 16-byte message.  */
         tm_message_sent[3]++;

         /* Increment the number of messages sent and received.  */
         tm_message_processing_counter++;
     }
```

Example results when an RTOS is tested using Thread-Metric benchmark tests. Each test was run for 30 seconds.

Test name	Iterations	Thread-Metric ratio
Calibration test	8,850	-
Cooperative context switch	1,237,882	140
Preemptive context switch	487,470	55
Message processing	830,196	94
Semaphore processing	1,566,675	177
Memory allocation	1,404,046	159
Interrupt handling	745,664	84
Interrupt preemption	316,092	36

Table 1

Table 1 shows some real-life results for a RTOS we tested using Thread-Metric.

RTOS ADAPTATION LAYER

To facilitate running on different RTOSes, Thread-Metric is structured to use a simple API, which must be adapted to the API of any RTOS to be measured. This adaptation is done using an RTOS adaptation layer that requires customization to a particular RTOS. The tool includes stub routines for each service to be measured. The transport layer (`tm_porting_layer.c`) contains shells of the generic RTOS services used by each of the actual tests. The shells provide the mapping between the tests and the underlying RTOS, and must be adapted for a specific RTOS. Listing 1 shows an example of a message passing service.

Thread-Metric is designed to be run over a sufficiently long period of time so that precise measurement isn't necessary. Simple wall-time measurement provides sufficient resolution, given the large number of service iterations. The tool first runs a calibration measurement to determine processor speed. This calibration value can be used to normalize results from different systems, if desired.

The complete Thread-Metric suite can be downloaded from *www.embedded.com/code/2007code.htm* and requires no license. ■

William Lamie is a co-founder and CEO of Express Logic Inc., and is the author of the ThreadX and Nucleus RTOSes. Lamie can be reached at blamie@expresslogic.com.

John Carbone, vice president of marketing for Express Logic, has 35 years experience in real-time computer systems and software. Carbone has a BS degree in mathematics from Boston College, and can be reached at jcarbone@expresslogic.com.