By James F. Ready,
Vice-President,
Microtec.

Designing for Worst Case: The Impact of Real-Time OS Performance on Real-World Embedded Design

The engineer who designs and develops real-time embedded software must become adept at a sophisticated juggling act. On one hand, today's embedded systems include more functionality than ever before: elaborate graphical user interfaces, network connectivity, and interprocessor communication. On the other hand, the same time constraints still prevail. Embedded systems must interact predictably with devices in the outside world. They generally operate unattended, and the consequences of any kind of failure range from severe to unthinkable.

In response to these competing pressures, use of off-the-shelf system software in real-time applications has increased dramatically. Vendors such as Microtec offer proven software that can be used in a real-time embedded system, improving the functionality and robustness of the design. This article will explore the relationship between the timing requirements of real-world applications and the performance metrics typically published by real-time operating system (RTOS) suppliers. It will also present some data on the actual response times of leading commercial products, including VRTXsa and VRTX32, high performance general purpose kernels offered by Microtec . VRTXsa comprises the most recent advances in kernel technology and is the successor to the second generation of the VRTX family, VRTX32.

CONTEXT SWITCH TIME: A MYTHICAL INDICATOR

The two metrics which are used most frequently to compare RTOS performance are context switch time and interrupt latency.

Context or task switch time purports to be an indicator of RTOS efficiency. Unfortunately, it is largely a mythical indicator. This metric simply measures the amount of time it takes an RTOS to save the context of one task — that is, put its registers and stack pointer into a control block — and load the context of another task from a second control block. The context switch times provided by RTOS vendors frequently do not even include the time it takes the RTOS to decide which task to schedule.

Context switch time never shows up in isolation in an application. It shows up instead as added overhead on an RTOS call. For example, if a VRTXsa application calls sc_qpost() to post a message to a queue at which no task is waiting, the call will take 5 µsec to execute (25 MHz 68040, caching enabled). If a task is waiting, it must be made ready and the call will take 7 µsec to execute. If the task which was made ready is at a higher software priority than the currently running task, a context switch must take place so that the newly readied task can run. In this case, the same sc_qpost() call will take 16 µsec. This last added overhead includes a context switch time of less than 9 µsec. But the fact that the context switch time is less than 9 µsec tells the software engineer very little about the real overhead VRTXsa imposes.

Since context switch time alone provides very little information about the real overhead of the RTOS, and since it is not reflective of any outwardly visible operation within a real-time system, its worth as an indicator of efficiency is very questionable. Developers should never rely on context switch time to compare RTOSs.

INTERRUPT LATENCY: NOT THE WHOLE STORY

The other widely published RTOS metric is interrupt latency. Interrupt latency is defined as the worst-case time that can pass between the assertion of an interrupt and the execution of the first instruction of the corresponding Interrupt Service Routine (ISR). The interrupt latency of an RTOS depends on three things:

- The interrupt latency of the processor hardware, which is usually negligible.
- The time it takes the RTOS to handle the interrupt and pass control to the appropriate ISR.
- The RTOS' interrupt disable time, which is the worst-case length of time for which the RTOS disables interrupts in order to protect its own critical code sections.

Every RTOS has some interrupt latency associated with it. Interrupt latency is certainly one indication of the responsiveness of an RTOS. However, interrupt latency is not the whole story when it comes to understanding the impact of an RTOS on the behavior of an embedded system. How an RTOS-based system ac-

tually responds in several different real-time scenarios will be examined in this paper.

WORST-CASE INTERRUPT RESPONSE TIME

The need to guarantee a worst-case response time to a certain interrupt is very common in real-time applications.

The worst-case response time to a single interrupt is easy to predict. As long as your application does not disable interrupts or disables them only for very brief periods, the interrupt latency of the RTOS will be the worst-case interrupt response time of your system. In a VRTXsa application on a 25 MHz 68040 with caching enabled, that would give you a guaranteed worst-case interrupt response time of 5 μsec.

However, if the application can disable interrupts longer than the RTOS, the worst-case interrupt latency of your overall system becomes the sum of the latency induced by the processor and the worst-case interrupts-off time of your application. For this reason, it is very important to be careful about the design of any code that might run with interrupts disabled. A single lengthy search loop with interrupts disabled will degrade the worst-case interrupt response time of your entire system.

WORST-CASE RESPONSE TO ADDITIONAL INTERRUPTS

What if you must guarantee a response time to more than one interrupt? Predicting the worst-case response time to the second interrupt is a more involved process than simply looking at the RTOS' published interrupt latency.

Consider the scenario presented in Figure 1. Two interrupts are asserted almost simultaneously. The worst-case response time to the second interrupt will depend on three factors:

- The RTOSs interrupt latency.
- The worst-case execution time of the first ISR.
- The worst-case overhead imposed by the RTOS at interrupt exit.

The first factor was analyzed in the previous section. As for the second factor, a fundamental principle of design for RTOS-based applications has to do with the interaction between ISRs and tasks. In order to prevent unnecessary masking of additional interrupts, ISRs generally perform only minimal activities needed to clear the interrupt, handle the device, and collect any data. Any time-consuming activities that must occur in response to the interrupt — e.g., archiving or displaying, checking or processing the data — are deferred if at all possible to a task that executes at the appropriate software priority with interrupts enabled. For this reason, almost any ISR which executes in an RTOS environment will execute at least one kernel call to communicate or synchronize with a task.

The worst-case execution time of the ISR will depend on the worst-case execution time of this RTOS call as well as on the execution time of any user-supplied code. It follows that only system calls with published, guaranteed execution times should be used from ISRs. These published execution times must include the time required to ready or schedule the task.

In most cases, ISRs in real-time systems must exit through the kernel rather than using the processor's interrupt return instruction directly. This allows the kernel to determine whether a context switch to a new task should be performed. This interrupt exit operation takes time and will have an impact on the worst-case response time to a second interrupt.

WORST-CASE INTERRUPT TASK RESPONSE TIME

Both of the scenarios discussed previously assumed the response to the interrupt occurs within its ISR. In a real-world application, this is frequently not the case.

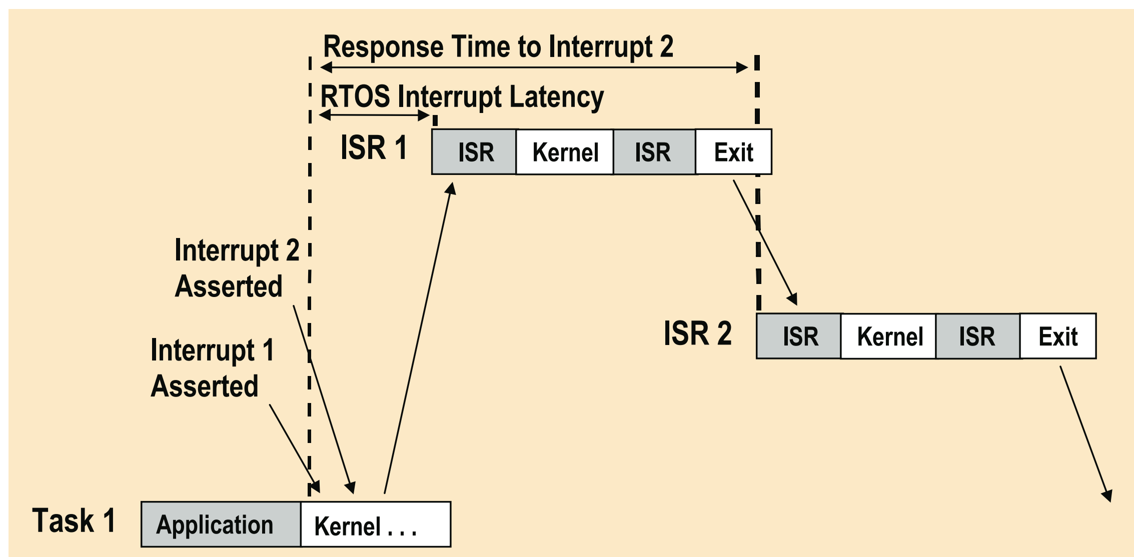

Figure 1: Response to Additional Interrupts
Published RTOS interrupt latency is only one component of the worst-case response time to a second interrupt. Execution times for kernel system calls are also important.

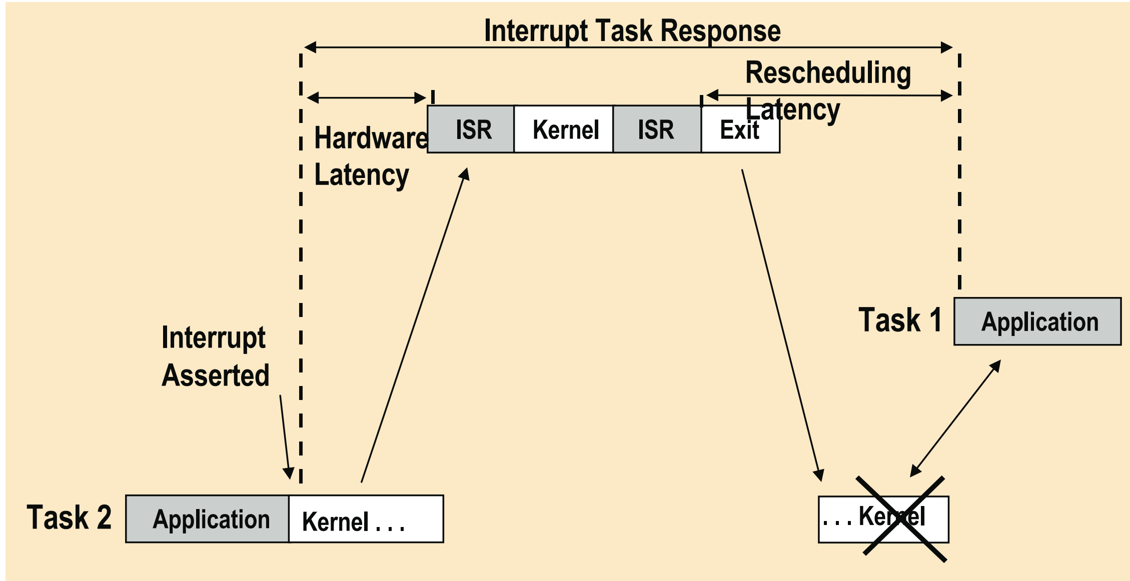

Figure 2: Interrupt Task Response
For a non-preemptable kernel, worst-case rescheduling latency includes the time required to complete an outstanding kernel service. A preemptable kernel such as VRTXsa avoids this overhead.

Frequently the real response to an event occurs within some task with which the ISR communicates. For example, an ISR may receive characters into a buffer and pass this buffer to a task. It is the task which processes the data and decides on a response. In this case, the real end-to-end event response time (interrupt task response) would include the worst-case time required to reschedule the task.

A particularly interesting situation arises if an interrupt arrives while the kernel is executing a system call. (This is illustrated in Figure 2.)

Here a distinction must be made between preemptable and non-preemptable kernels. Most commercial RTOS kernels support preemptive task scheduling, which means that application tasks can preempt each other essentially at will. Most commercial RTOS kernels are also interruptible, which means that when an interrupt occurs while the kernel is executing, the ISR will be invoked almost immediately (within the constraints of the published interrupt latency). However, many commercial RTOS kernels are not preemptable, that is, kernel services cannot be preempted. This means that when an ISR exits through the kernel, any kernel service which happened to be executing when the interrupt occurred must be completed before the kernel's task dispatcher can run. In other words, if the kernel was executing a non-time-critical operation on behalf of a low-priority task, this operation must complete before a high-priority task the ISR made ready can run. This adds to the rescheduling latency between the time an ISR exits and the time a task can be dispatched in response.

How long is this rescheduling latency? In a non-preemptable kernel such as BSD UNIX or pSOS+, it could be the length of the slowest system call. In the case of BSD UNIX, if a low-priority task is in the middle of a non-deterministic heap insertion operation, this rescheduling latency is in fact unbounded.

A preemptable kernel, on the other hand, can put the low-priority operation on hold and dispatch the high-priority task at once. This technique minimizes rescheduling latency and improves the responsiveness of the entire system.

THE PRIORITY INVERSION PROBLEM

Given the obvious advantages of a preemptable architecture in minimizing rescheduling latency, one is certainly led to wonder why many popular kernels feature non-preemptable implementations. One reason is to avoid a potential problem with unbounded priority inversion.

To understand the problem, consider the following scenario:

1. Low-priority task L gets semaphore s
2. High-priority task H preempts L
3. H pends on s
4. L resumes execution
5. Any number of medium-priority tasks Mx can now preempt L for an indefinite period of time while H remains blocked

This scenario results in "priority inversion" because the tasks H and L behave as if their priorities were inverted. Further, the existence of one or more medium-priority tasks Mx means that the problem can persist for an unbounded period of time. This can result in unsafe and highly unpredictable behavior.

Unbounded priority inversion can be a problem in any application in which tasks of different priorities use traditional mutual exclusion mechanisms (e.g., semaphores) to share resources. It can also occur within an operating system if operating system services do not operate automatically. For example, if an RTOSs message queue implementation uses underlying semaphores, the RTOS itself can be subject to unbounded priority inversion if a high-priority task and a low-priority task attempt concurrent access to a

queue.

Traditional non-preemptable kernel designs prevent this problem, at least at the kernel level, by eliminating concurrency within the kernel. However, they do so at the expense of increased rescheduling latency.

AN ALTERNATIVE DESIGN WHICH MINIMIZES RESCHEDULING LATENCY

Recently an alternative kernel architecture has emerged which minimizes rescheduling latency while avoiding unbounded priority inversion problems. This design is typified by real-time UNIX implementations such as Mach and Chorus and has also been employed in Microtec's VRTXsa. It involves implementing higher-level, preemptable services based on lower-level services. The lower-level services are not preemptable and therefore are not subject to priority inversion problems. Since the non-preemptable services are simple and execute rapidly, they have minimal impact on rescheduling latency.

The more sophisticated higher-level services are preemptable to minimize rescheduling latency. How can priority inversion be bounded in such an implementation? A technique called "priority inheritance" provides the answer. Consider the same interaction between tasks H, L, and Mx described previously, but with the introduction of a new mutual exclusion mechanism called a priority inheritance mutex in place of the traditional semaphore:

1. Low-priority task L locks priority inheritance mutex m
2. High-priority task H preempts L
3. H attempts to lock m
4. L resumes execution, but temporarily "inherits" H's priority
5. Medium-priority tasks Mx cannot preempt
6. L runs without interruption until it relinquishes m to H and returns to its original priority
7. H resumes execution

The use of priority inheritance eliminates the risk of unbounded priority inversion within a preemptable kernel. This means that kernels can now be designed to be largely preemptable and to impose minimal rescheduling latency. If mechanisms that implement priority inheritance are exported to the application as well, applications can be designed easily with bounded priority inversion.

AN APPLES-TO-APPLES COMPARISON

As can be seen, assessing the appropriateness of an RTOS product to a specific application requires a fair amount of performance data and architectural information. While most RTOS vendors publish performance metrics of some sort, each vendor approaches the problem differently. Some published metrics reflect RTOS overhead as seen from an assembly-language program; some as seen from C. Benchmarks and other timing data published by different vendors have often been measured on different hardware with varying processors, clock rates, and memory access speeds. All of these factors make meaningful comparison difficult.

These considerations led a group of engineers at the Superconducting Supercollider Laboratory (SSC) to undertake an industry survey. They created a suite of tests which were designed to measure RTOS responsiveness and throughput under conditions approximating an actual application environment. They ran their benchmarks on a consistent hardware platform against four leading RTOSs and published the results in a paper presented at the IEEE 1991 Particle Accelerator Conference.

Since that date, some of the vendors in the original SSC survey, including Microtec, have introduced new products with improved performance. The data presented in this paper includes the latest known results for all vendors and products included in the original study. While SSC and others made some of the measurements by vendors, all were executed on the same hardware platform and with reasonably consistent methodology. Differences in measurement technique which might be material enough to adversely affect the "apples-to-apples" nature of the data are noted.

TEST CONDITIONS

All measurements were run on a Motorola MVME147S-1 board, which features a 25MHz 68030 processor. Measurements were taken with cache and MMU disabled and were repeated 100 times. Minimum, maximum, and average values are reported in order to give an indication of determinism. In the original study, SSC used the MVME147's on-board timer to make the measurements, limiting the resolution which could be achieved on the response measurements to 6.25 µsec. In subsequent benchmarking work, both Wind River Systems and Microtec have found that degree of resolution inadequate. Both vendors have made use of a technique in which a second board with a more precise on-board timer was employed as a time-keeper, permitting measurements to be made with a resolution of plus or minus 1 µsec. Note that a 2 µsec spread between minimum and maximum measurements is within the resolution of this test setup.

INTERRUPT SERVICE AND INTERRUPT TASK RESPONSE

SSC's Interrupt Service Response and Interrupt Task Response measurements are designed to give an indication of the responsiveness of various RTOSs to external events. The Interrupt Service Response metric measures the system's interrupt latency, that is, the amount of time that elapses between the assertion of an interrupt and the execution of the first instruction of an associated ISR written in C. Results are presented in Figure 3.

Note that the results for VRTXsa, VRTX32, and VxWorks 5.1 have been obtained using the auxiliary timer method and so are limited to a resolution of 1

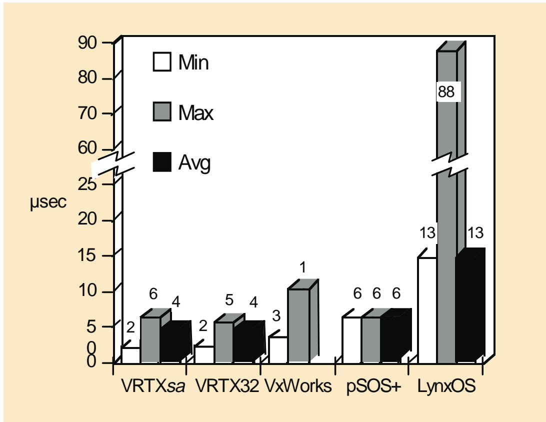

Figure 3. SSC Interrupt Service Response Benchmark
In an apples-to-apples comparison, VRTXsa offers faster response to interrupts than other leading RTOSs

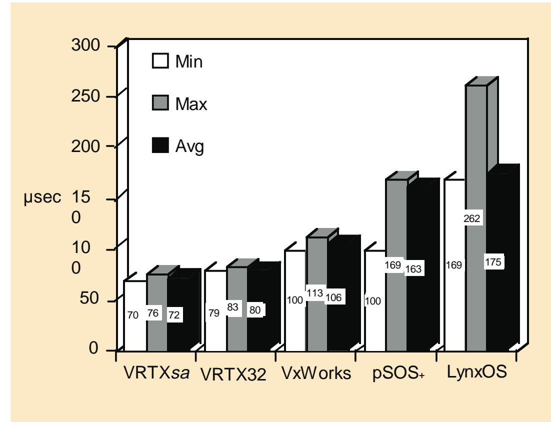

Figure 4: SSC Interrupt Task Response Benchmark
VRTXsa's preemptable architecture minimizes rescheduling latency, contributing to fast, deterministic interrupt task response.

µsec resolution. The results for pSOS+ and LynxOS were taken from the original SSC study and are limited to a resolution of 6.25 µsec. So the consistent result given for pSOS+ on the Interrupt Service Response test (minimum equals maximum equals average equals 6) in reality means only that the time taken was between 6.25 µsec and 12.5 µsec on each iteration.

The Interrupt Task Response benchmark measures the amount of time which elapses between an external interrupt and execution of the first instruction of a high-priority task, written in C, which is dispatched to handle the event. Results of this test are presented in Figure 4. As one would expect, VRTXsa's preemptable architecture contributes to an excellent result.

CONCLUSION

The approach most RTOS vendors take to quantifying the performance of their products is to publish a few metrics such as interrupt latencies, context switch times, and execution times for a few selected system calls. This approach is woefully inadequate for true worst-case design. Attempting to determine the worst-case performance of an application based on this kind of information is like attempting to understand the performance of a processor based on the execution speed of the multiply instruction.

A meaningful analysis of the appropriateness of a kernel to a given time-critical application must consider:

- Interrupt latency time.
- Interrupt exit time.
- In the case of a non-preemptable kernel, worst-case execution times for all system calls that will be used anywhere in the system.
- In the case of a preemptable kernel, worst-case execution times for system calls which will be used in time-critical paths.

Preemptable kernels such as VRTXsa offer significant advantages for worst-case design by minimizing rescheduling latency and improving responsiveness and safety of RTOS-based systems.

In an environment where the right answer late is wrong, predictable performance over the lifetime of a system can only be achieved if the software is designed with worst-case operating conditions in mind. Complete information about RTOS architecture and performance is absolutely essential to attaining this goal.

James F. Ready serves as Vice President and Chief Technical Officer at Microtec. Previously, Mr. Ready, a founder of Ready Systems, served as its Director and its Vice President from 1981 to the merger with Microtec Research in 1993. (Microtec Research changed its name to Microtec in May 1996). Previous to Ready Systems, Mr. Ready held engineering and marketing positions at ROLM Corporation. Mr. Ready holds a B.A. from the University of Illinois and a M.A. from the University of California, Berkeley.

Mr. Ready is best known as the father of the VRTX real-time operating system and is widely credited with starting the off-the-shelf real-time operating system industry. He has published numerous technical articles and papers and has lectured worldwide on the principles of real-time software design. Mr. Ready has also served as a keynote speaker at several conferences and industry trade shows.

Read also our Evaluation of VRTX at page 6-12