

Minimizing Interrupt Response Time

Author:

David Kleidermacher,
Vice President, Engineering,
Green Hills Software, Inc.

Synopsis:

There are many aspects of operating systems, such as context-switch time, service call performance, scheduling algorithms, business model, tools and middleware integrations, that embedded systems developers should be concerned about; however, the most important characteristic – what makes an operating system a real-time operating system – is its ability to service interrupts quickly. A failure to meet a response time requirement in a real-time system can be catastrophic. In addition, operating system vendors usually publish best case, average case, or worst case response times in a particular test environment. But this is merely marketing hype; what really counts is the true, guaranteed worst-case time. This article will describe how a minimal yet guaranteed worst-case interrupt response time can be achieved in any type of electronic product, from simple control applications to the most complex, multi-process, I/O-intensive systems. We focus on minimizing response time for the highest priority interrupt (since lower priority interrupts may be delayed by it).

When an interrupt fires, the microprocessor executes an interrupt service routine (ISR) that has been installed to service the interrupt. The amount of time that elapses between a device interrupt request and the first instruction of the corresponding ISR is known as interrupt latency. The interrupt handling code can optionally execute an operating system API that will cause a thread to be awakened. The amount of time that elapses between the interrupt request and the first instruction of the thread awakened to handle it is known as thread response time. ISRs are usually written at least partially in assembly language and typically have limited resources. Code executing in a thread is written in a high level language such as C, has full access to the operating system API, and is easier to debug, analyze, and profile. This article will address the ability to guarantee both a minimal interrupt latency and thread response time.

In order to compute the system's worst case response time, it is necessary to examine all of the sources of interrupt response delays to ascertain which source causes the longest delay to the servicing of the highest priority interrupt. Possibilities include:

- Longest hardware-induced latency
- Longest software-induced (e.g. by the kernel) interrupt disabling region
- Longest disabling region caused by the ISR of a lower priority interrupt

The theoretical worst-case delay may depend on the choice of CPU, choice of operating system, and how device drivers and other software are written.

Minimum Interrupt Response Time: 5 Simple Rules

Sound programming techniques coupled with proper RTOS interrupt architecture can ensure the minimal response time. The recipe:

- 1. Keep ISRs simple and short.**
- 2. Do not disable interrupts.**
- 3. Avoid instructions that increase latency.**
- 4. Avoid improper use of operating system API calls in ISRs.**
- 5. Properly prioritize interrupts relative to threads.**

The following sections discuss these ingredients.

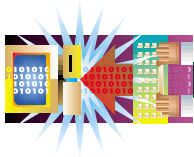
Short ISRs

Developers should keep ISRs short and simple, avoiding loops and other constructs which can increase latency and complexity. When an interrupt fires, the microprocessor typically disables interrupts globally before transferring control to the ISR; then either the ISR must reenables interrupts (when safe to do so) or the ISR return instruction will reenables interrupts automatically. By keeping ISRs short and simple, developers will avoid the common pitfall of leaving interrupts disabled for too long, thereby increasing the latency of higher priority interrupts. In addition, ISRs are notoriously difficult to debug and often must be analyzed by inspection – practical only for simple implementations. Keeping ISRs short will minimize interrupt response time, testing and debugging time, and your frustration level.

Do Not Disable Interrupts

Operating system architecture is often the most significant factor for determining response times in an embedded system. A major contributor to increased interrupt latency is the number and length of regions in which the kernel disables interrupts. By disabling interrupts, the kernel may delay the handling of high priority interrupt requests that arrive in those windows in which interrupts are disabled. Most operating systems employ what we call a Simple architecture: whenever the operating system wants to prevent preemption within a critical section of its code, the operating system simply disables interrupts for the duration of the critical section. Most RTOSes employ this Simple architecture since it is easy to implement and the commonly used and understood mechanism (this is what we learn in university operating systems class).

By disabling interrupts, the Simple RTOS is in effect sacrificing the latency of the highest priority interrupt to avoid problems caused by handling of lower priority interrupts. A better solution, implemented in



the Advanced architecture, is to never disable interrupts in kernel service calls. By never disabling interrupts, not only does the Advanced RTOS guarantee the minimum possible latency for the highest priority interrupt, but also the absolute worst case latency can be trivially proven (no need to examine all of those disabling sequences in the kernel).

All other things equal, an RTOS that does not disable interrupts in service calls will achieve better response times than an RTOS that does disable interrupts. This is common sense: if a high priority interrupt arrives while executing within a Simple RTOS's critical section, the latency for the high priority interrupt will be increased by the amount of time it takes to execute the remainder of the critical section.

Avoid High-Latency Instructions

Certain CPU instructions, such as integer divide and string manipulations, can take many cycles to execute; interrupts are inhibited until execution is complete. Although it is not always practical to avoid these instructions in application code, the RTOS kernel itself should avoid them. An RTOS that avoids these instructions will, all other things equal, achieve better interrupt latency than an RTOS that ignores this restriction.

Avoid Improper API Use in ISRs

ISRs commonly do not require any kernel API access at all. The ISR may record some information, check status, or perform some other basic operation before acknowledging the interrupt and returning. In some cases, a more complex ISR is required in order to wake up a thread for higher level processing. This wake up may be accomplished by releasing a semaphore or writing to a message queue. However, many RTOS vendors permit the use of a plethora of service calls from ISRs. Although this may seem convenient, it can be deadly when misused. The RTOS vendor should carefully control and limit ISRs to a small set of service calls that are absolutely necessary and absolutely deterministic.

Non-deterministic API calls

As an example of the peril regarding API usage in ISRs, consider how an RTOS handles the queue of threads waiting for a resource (e.g. a semaphore or message). Many Simple RTOSes use an ordinary linked list to hold the queue of threads

waiting on a resource. When the resource becomes available, the first thread, regardless of its importance relative to other waiting threads on the list, is provided the resource and allowed to run (in contrast, the Advanced RTOS provides a mechanism to automatically prioritize waiting threads).

Some of the Simple RTOSes have bolted on a new service call that pulls the highest priority thread out of the linked list and jams it onto the front of the list. This allows the highest priority thread to be awakened when the resource becomes available. One obvious problem with this approach is its poor usability: the developer must remember to insert these prioritization calls and determine where in the code flow they belong.

You may have guessed the other problem. The prioritization request call is not deterministic. The RTOS must search linearly through the unordered list to find the highest priority blocked thread. Unbounded time service calls are anathema in RTOS design since they can cause prohibitively long interrupt latency and make it difficult if not impossible for system designers to apply standard real-time scheduling techniques such as Rate Monotonic Analysis (RMA). Furthermore, use of the nondeterministic API may not cause failures in the lab or in simplistic applications. However, complex applications or applications that behave differently in the field may encounter a missed deadline that results in a system failure. In innocently following the vendor's guidance that permits, within an ISR, the use of an API to prioritize the wait list, the developer has unwittingly increased interrupt latency by an unpredictable, unbounded amount of time.

ISR Service Calls in the Advanced Architecture

Some claim that a drawback of the Advanced interrupt architecture is that service calls cannot be executed directly from an ISR (doing so could cause corruption since the kernel does not disable interrupts in critical sections). When a complex ISR is required, the Advanced architecture provides the option of using a second-level handler, sometimes termed a callback, to perform higher level interrupt handling services, such as releasing a semaphore or placing a message into a queue. The callback is executed by the kernel once it has

returned to a consistent state. If the developer must write code in the ISR to spawn the callback and then write more code in the callback to do the service call, this makes programming more difficult.

The good news, however, is that the Advanced RTOS can hide the details of the two-level handling by providing the required set of service calls (e.g. releasing a semaphore) to ISRs. The service call knows that it is being called from an ISR and will place the callback on behalf of the programmer. Thus, the code is identical to that used with a Simple RTOS: the ISR calls an API to release a semaphore or post a message. From the developer's perspective, there is no additional setup or complexity. There are some important advantages to the Advanced architecture's optional two-level handler. Service calls can take a significant amount of time (especially in the case of the Simple RTOS that provides unbounded time service calls). By pushing this work into the callback (where interrupts are enabled), the Advanced RTOS reduces the temporal footprint of the ISR which in turn reduces the latency for higher priority interrupts. Again, this is just common sense: in the Simple RTOS, the entire execution of the service call is within the ISR itself; in the Advanced RTOS, the ISR is much shorter because it does not contain that service call footprint. Note that the overhead of separating service call processing into a callback is negligible: a callback entails merely placing an item containing a function address on a list for the scheduler to call; no extra stack creation or any other heavyweight processing is required. Figure 1 depicts the performance difference in interrupt latency between the Simple and Advanced architectures in the midst of a high priority interrupt preempting a lower priority interrupt.

In addition to reducing ISR footprint, callbacks enable the bulk of complex ISR processing time to be allocated to specific threads (for example, time spent handling Ethernet interrupts can be attributed to the network TCP/IP thread using the Ethernet device). These per-thread callbacks provide finer grained control over time spent in the system and reduce priority inversion (when a low priority interrupt can delay a higher priority thread handling a higher priority interrupt source). This level of control of processing time may not be important in simple embedded systems,

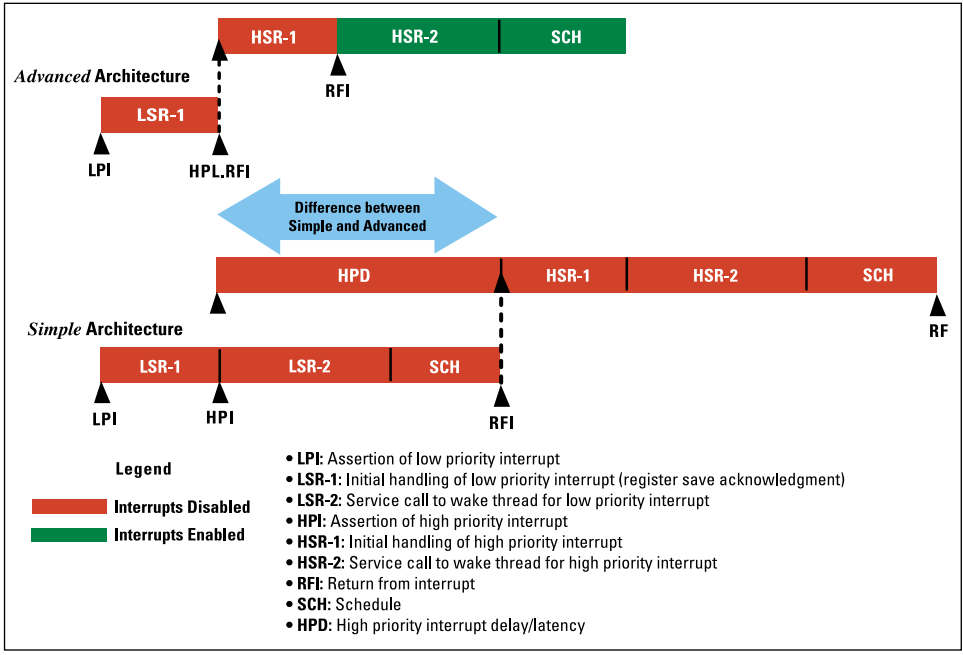
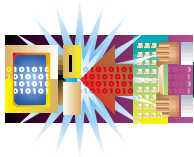


Figure 1: Typical sequence for Simple architecture results in a longer interrupt latency (time to reach HSR-1) than the Advanced architecture sequence,

but as systems grow in complexity with many tasks and interrupt sources, this Advanced approach can mean the difference between meeting time budgets and going over them.

Because ISRs in the Advanced RTOS reenables interrupts earlier and reduce overall interrupt latency in the system, the Advanced RTOS is typically capable of

handling systems with higher frequency interrupts (e.g. from multiple interrupt sources). The Simple RTOS may not be able to keep up, dropping important data and missing deadlines in the face of high interrupt loads.

Many safety critical standards (such as DO-178B for flight safety) require an RTOS vendor to publish worst case execution

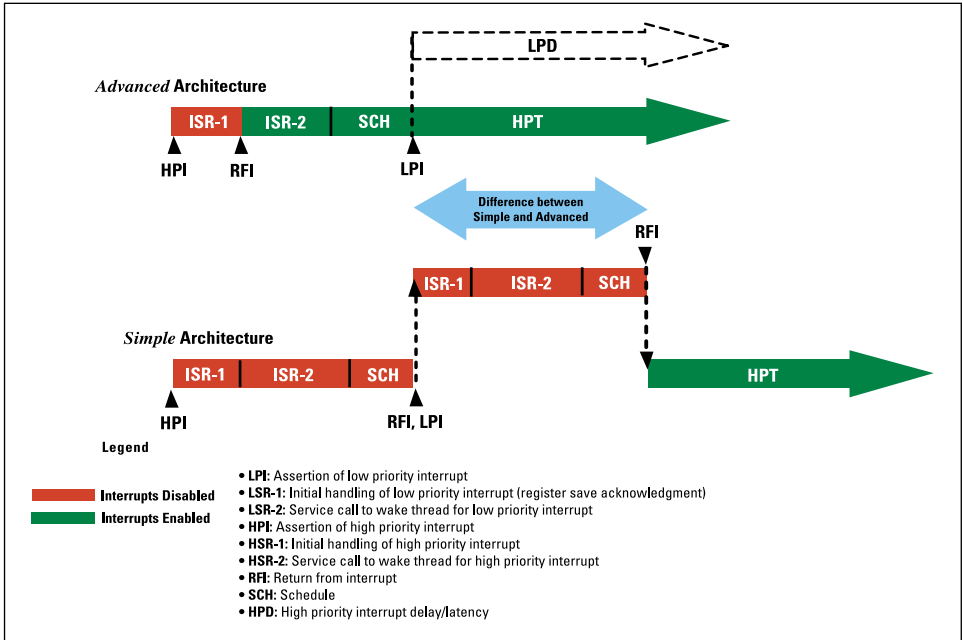


Figure 2: Typical sequence for Simple architecture results in a longer interrupt latency (time to reach HSR-1) than the Advanced architecture sequence.

times for all service calls. Thus, developers who use an RTOS that both employs the Advanced architecture and has been certified to stringent safety standards can have maximum confidence in the response time and robustness of their systems.

Pardon the Interruption: Prioritizing Interrupts Below Threads

When a thread is awakened for higher level processing of the most important real-time interrupt, this thread becomes the most important thread in the system. The thread must complete its processing within a fixed period of time.

In the Simple RTOS, the thread runs with all interrupts enabled. Any low priority interrupt in the system can fire, delaying the most important thread and causing deadlines to be missed. In fact, since interrupts may nest, multiple interrupts, and all of their associated ISR processing, could delay this high priority processing by an unpredictable, unbounded amount of time.

The Advanced RTOS allows interrupts to be prioritized relative to threads and provides services built-in to the scheduler to automatically enforce this prioritization. Thus, when a high priority thread is awakened during interrupt processing, the kernel will automatically inhibit lower priority interrupts prior to switching to the high priority thread. When the high priority thread completes its work and is de-scheduled, the kernel automatically reenables lower priority interrupts. This architecture guarantees the minimum thread response time for the highest priority real-time events. Figure 2 depicts the performance difference in thread response time between the Simple and Advanced architectures in the midst of a low priority interrupt preempting a high priority thread.

Conclusion

Complex embedded systems, with multiple concurrent tasks and interrupt sources, pose a challenge for RTOS interrupt handling architecture. Legacy RTOSes, designed with a Simple architecture sufficient for yesterday's basic embedded systems, fall short of this challenge. Embedded systems developers will achieve the best possible interrupt response time by following a few simple rules and employing an RTOS with an Advanced interrupt architecture.