Pardon The Interruption!

Two Approaches to RTOS Interrupt Architectures: Unified and Segmented

Like stand-alone systems, embedded applications running on top of real-time operating systems (RTOSes) generally use pieces of code called Interrupt Service Routines (ISRs) to handle interrupts generated by external events. External events can be caused by just about anything, from an asynchronous character arrival via a UART to the expiration of a timer. ISRs acknowledge the hardware condition and provide the initial handling of data sent or received as required by the interrupt. An ISR often is responsible for providing the RTOS with information necessary to provide services to application threads. Data might be moved into a buffer for subsequent processing, added to an entry to a queue, values set indicating that an event has occurred, and so on. Since application code execution is interrupted (delayed) during the execution of an ISR, most applications minimize the amount of code in the ISR and rely instead on non-ISR code (an application “Thread” or “Task”) to complete the processing at a priority appropriate to the importance of the work. The highest priority application code is executed as quickly as possible and delayed as little as possible, even in situations with intense interrupt activity.

RTOS Interrupt Architectures

A fundamental challenge in RTOS design involves supporting asynchronous access by interrupt routines and RTOS services to internal RTOS data structures. It cannot be allowed that, while modifying a data structure, a service or ISR gets interrupted and a different service or ISR makes unrelated modifications to the same structure, leaving it in a changed state for the original code to (unknowingly) continue modifying. The results can be catastrophic.

All RTOSes must address this challenge and prevent multiple ISRs (or system calls) from modifying the same structure at the same time. There are at least two approaches to this problem, one used by the majority of RTOSes and another used by a few.

The more popular approach is to briefly lockout or disable interrupts while an ISR or system service is modifying critical data structures inside the RTOS. This reliably prevents any other program from jumping in and making uncoordinated changes to the area being used.
by the executing code. This approach is called the “Unified Interrupt Architecture” because all interrupt processing is performed at one time, in a single, “unified” service routine.

Another approach is not to disable interrupts in system service routines, but instead (by rule or convention) to not allow any asynchronous access to critical data structures by ISRs or other service calls. Service call access to critical data structures from an ISR is “deferred” to a secondary routine, denoted here as “ISR2,” which gets executed along with application threads under scheduler control.

This approach reliably prevents interference with the actions of an executing system service call or ISR, by not allowing any threads, which might make system service calls, to execute until processing of critical data structures is completed. This approach is called a “Segmented Interrupt Architecture,” because it breaks up the processing required in response to an interrupt into multiple (usually 2) “segments” executed at different priorities. Let’s examine the performance implications of each approach on real-time system responsiveness.

The *Unified Interrupt Architecture*

RTOSes that employ a unified architecture treat the ISR no differently than an ISR in a stand-alone application. In this architecture, the ISR is allowed to make RTOS calls to modify kernel data structures and to interact with application threads and other resources. An example of such service calls might be to add a message to a queue. To do this and still maintain data security, the RTOS disables interrupts over certain (typically very short) code segments, thereby protecting common resources against asynchronous access from multiple ISRs or system service calls. The period of time for which interrupts are disabled adds to interrupt latency, and hence must be kept as brief as possible.

In this approach, the ISR typically performs minimal data movement and processing, and determines whether any application threads should be run as a result of the interrupt. If an application thread is required, it is marked “ready to run” at its assigned priority, and its execution is controlled by the scheduler along with all other application threads, in priority order. If this new application thread is of higher priority than the thread that was interrupted, then the new thread, not the interrupted thread, will run after the ISR finishes. This is called Thread Preemption.

In the unified Architecture approach, If Thread Preemption is required, then the Scheduler must come into play to run the new thread, and the interrupted thread
will resume when its priority so dictates. In a simple system, that might be right after the new, higher priority thread is finished.

If no Thread Preemption is caused by service calls made by the ISR, then a context save and restore around the ISR are all that’s required, and the interrupted thread is resumed.

The **Segmented Interrupt Architecture**

RTOSes that do not disable interrupts during system services cannot make modifications to RTOS data structures from within an ISR or system service routine, because the ISR or system service might be interrupted by another interrupt, which then might attempt to use the same data structure. To prevent such potentially damaging modifications, segmented architectures do not allow system service calls from ISRs, since such calls might result in manipulation of critical data structures.

They also disable the application scheduler for a period of time, since application threads also can make system service calls that might result in access to critical data structures. To implement these two restrictions, they divide the interrupt processing into two (or more) pieces. The first piece (ISR1) behaves like a traditional ISR but performs no interaction with RTOS data structures. This enables it to run with interrupts enabled. The second piece (ISR2) is a “scheduled entity” that makes all necessary RTOS data structure updates at the application level, and is invoked through the processing in ISR1, but it is allowed to run only when the scheduler transfers to it. Application thread processing can be performed once both ISR1 and ISR2 have been completed. This effectively delays applications while ISR1 and ISR2 are operating, and prevents asynchronous system services while RTOS data structures are being modified.

In the segmented architecture approach, if no Thread Preemption is caused by service calls made by ISR2, then, in addition to the processing performed in the unified approach, the context switches and processing of ISR2 also are required.

If Thread Preemption is caused by service calls made by ISR2, then, in addition to the processing just described, as additional context switch for the new thread also is required.

The performance observations in this section are somewhat abstract. Segmented architecture products are typically larger operating systems so their actual processing times for each RTOS performance object would most likely be greater than a simpler unified interrupt architecture RTOS. In addition, if memory protection or virtual addresses are supported, the time required to process an RTOS block will most definitely be greater. However, the following comparisons assume that the various RTOSes take the same amount of time to perform the same RTOS operation. It is further assumed that the time to perform ISR
processing in the unified approach is equal to the sum of the times to perform ISR1 and ISR2 in the segmented approach.

<table>
<thead>
<tr>
<th>Preemption</th>
<th>ISR Processing</th>
<th>ISR2 Related Processing</th>
<th>Return To Interrupted Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Preemption</td>
<td>ISR Processing</td>
<td>Return To Interrupted Thread</td>
<td></td>
</tr>
<tr>
<td>Segmented</td>
<td>ISR Processing</td>
<td>ISR2 Related Processing</td>
<td>New Thread + Return To Interrupted Thread</td>
</tr>
<tr>
<td>Unified</td>
<td>ISR Processing</td>
<td>New Thread + Return To Interrupted Thread</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Preemption</th>
<th>ISR Processing</th>
<th>ISR2 Related Processing</th>
<th>Return To Interrupted Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Preemption</td>
<td>ISR Processing</td>
<td>ISR2 Related Processing</td>
<td>New Thread + Return To Interrupted Thread</td>
</tr>
<tr>
<td>Segmented</td>
<td>ISR Processing</td>
<td>New Thread + Return To Interrupted Thread</td>
<td></td>
</tr>
<tr>
<td>Unified</td>
<td>ISR Processing</td>
<td>New Thread + Return To Interrupted Thread</td>
<td></td>
</tr>
</tbody>
</table>

Figure-3
In both preemption and non-preemption cases, the segmented approach takes longer.

**Total System Overhead Comparison**

Ironically, as can be seen in Figure-3, the total system processing time is *greater* in the RTOS with a segmented interrupt architecture. In both the non-preemption and preemption cases, as shown in Figure 3, the segmented interrupt architecture RTOS introduces additional overhead to schedule ISR2. It appears the most wasteful overhead case is the nonpreemptive since the unified interrupt RTOS simply returns to the point of interrupt if a higher-priority thread was not made ready by the ISR processing.

Another performance benefit of the unified RTOS approach is that only the interrupted thread’s scratch registers need to be saved/-restored in this case. This is not possible with a segmented interrupt RTOS since it does not know what the ISR2 portion of the ISR will do during the actual interrupt processing. Hence, segmented interrupt RTOSes must save the full thread context on every interrupt. In the non-preemptive case, the context save and restore performance is much slower in the segmented interrupt RTOS, although this additional overhead has not been factored into this comparison.

The segmented RTOS architecture intuitively appears to have an advantage in its ability to respond to interrupts. After all, it makes sense that if interrupts are never disabled, response should be faster. Although the idea behind not disabling interrupts is to make interrupt response faster, there are several practical problems introduced by this approach.

Although the segmented interrupt RTOS doesn’t disable interrupts, the hardware itself does when processing other interrupts. So the worst case lockout in the segmented interrupt RTOS approach is actually the time interrupts are locked out during the processing of another interrupt.

In addition, interrupts could also be locked out frequently in an application if the segmented interrupt RTOS uses a trap or software interrupt to process RTOS
service requests. In such cases, the hardware again will lock out interrupts while processing the trap.

Finally, the application itself might have interrupt lockout to enable it to manipulate data structures shared among multiple threads.

All of these issues introduce interrupt lockout, and largely defeat the interrupt-response benefits claimed for an RTOS that doesn’t disable interrupts. Certainly, a worst-case lockout time becomes an application issue, no longer within the control of the RTOS.

Providing that the unified interrupt RTOS doesn’t lockout interrupts any longer than the time the application itself disables interrupts and any time for which the RTOS trap mechanism locks out interrupts, there appears to be no performance advantage in interrupt response with a segmented interrupt architecture. This is really unfortunate since the goal of faster interrupt response is what motivated the segmented approach.

**Interrupt Completion Time**

While interrupt response time is important, a more relevant issue involves the completion of the processing triggered by the interrupt. “The interrupt isn’t done until the interrupt is done!” Being able to respond to the interrupt is one thing; being able to finish the processing required by it is another. Since the segmented approach requires the deferral of ISR2 and related application thread processing, and requires more processing time to perform the required RTOS operations to manage ISR1 and ISR2, in both the preemption and non-preemption cases, the total processing required for an interrupt is smaller using the unified interrupt architecture.

In segmented interrupt RTOSes that never disable interrupts, it is necessary that they avoid allowing application threads to run for periods of time while RTOS data structures are manipulated. Prohibiting ISRs from making system calls prevents the performance penalty as shown above, but it also is necessary to prevent applications from making system calls that could cause the same problem. Since interrupts can occur at any time, it is necessary that segmented RTOSes avoid the possibility that an interrupt occurring during an ISR could result in the scheduling and execution of an application that might request an RTOS service that involves manipulation of an RTOS data structure while the interrupted ISR is in the middle of modifying that same RTOS data structure.

To solve this problem, segmented RTOSes do not allow the scheduler to transfer control to any application while the RTOS is manipulating critical data structures. Instead, the scheduler is “delayed” or “disabled” until the RTOS is finished. Thus, while avoiding interrupt latency by never disabling interrupts, segmented RTOSes in fact “disable application threads,” impacting system performance.
Another issue related to interrupt architecture is the need for system resources. In the unified interrupt RTOS, a separate system stack processes all interrupts and nested interrupts. The advantage of this is that the worst case stack usage is taken into account in one place rather than in each thread’s stack which would certainly waste an excessive amount of memory.

The segmented interrupt RTOS could have stack resource problems. Some segmented RTOSes implement the ISR2 as a completely separate entity with its own dedicated stack. Other such RTOSes execute the ISR2 on top of a predetermined thread’s context. In either case, the ISR2 stack processing needs to be accounted for in multiple places.

Both types of RTOS interrupt architectures are equally deterministic – between 0 and some maximum lockout period. The one major difference in determinism occurs in segmented interrupt RTOSes that discard programming segments that have contention. For example, suppose thread A is running and accessing an RTOS system object B, and an interrupt occurs. In the segmented approach, ISR1 executes followed by execution of ISR2. If that ISR2 attempts to access the same system object B, what happens? Some segmented RTOSes temporarily suspend ISR2, while others simply discard the work already done by thread A and make it restart its access of object B after ISR2 completes. The first approach is deterministic. However, the approach that involves throwing away work already done is not. Under certain conditions – very bizarre for sure – a thread could have a substantial amount of processing discarded. Even worse, whether its work has been discarded would be solely dependant on the timing of random external events.

Another problem occurs when high frequency interrupts are introduced. If the interrupt rate (time between interrupts) is less than the time required to process ISR1, ISR2 and related overhead, then it is possible that the RTOS and application will find themselves in an endless loop trying to service an interrupt. In a unified interrupt RTOS, this problem doesn’t occur until the rate is less than the time required to process ISR, which we’ve seen (above) to be smaller. Hence, a unified interrupt RTOS can inherently support higher frequency interrupts.

Summary
While both segmented and unified interrupt architecture RTOSes enable deterministic real-time management of an embedded system, there exhibit significant differences with respect to the efficiency and simplicity of their system. Segmented RTOSes can claim that they never disable interrupts within system services. However, in order to leave interrupts enabled, they must delay application threads, introducing a possibly worse alternative. The segmented approach also adds measurable overhead to the context switch process, and complicates application development. In the end, a unified interrupt architecture
RTOS demonstrates clear advantages for use in real-time embedded systems development.