Pardon the Interruption:

Two Approaches to RTOS Interrupt Architectures

Author:
William E. Lamie, Co-Founder and CEO, Express Logic, Inc.

Synopsis:
While both Segmented and Unified Interrupt Architecture RTOSs for ARM core-based designs enable deterministic real-time management of an embedded system, there are significant differences between them with regard to the efficiency and simplicity of the resulting system.

This article evaluates the two approaches to RTOSs Interrupt Architectures, and makes a compelling argument for the unified approach.

Like stand-alone systems, embedded applications running on top of real-time operating systems (RTOSs) require Interrupt Service Routines (ISRs) to handle interrupts generated by external events. Many things can cause external events, from an asynchronous character arrival on a UART to a periodic timer interrupt. ISRs have the responsibility of acknowledging 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. Examples include moving data into a buffer for processing, adding an entry to a queue for processing, and setting a value to indicate that an event has occurred. 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. This enables the highest priority application code to be 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 is supporting asynchronous access to internal RTOS data structures by interrupt routines and RTOS services. While modifying a data structure, it cannot be allowed to interrupt a service or ISR to let a different service or ISR make 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 RTOSs 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 address this challenge. The more popular approach, used by the majority of RTOSs, is to briefly lock out 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 critical 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 (ISR).

Another, less popular, approach is not to disable interrupts in system service routines, but instead (by rule or convention), 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 we denote "ISR2," which gets executed along with application threads under scheduler control. This

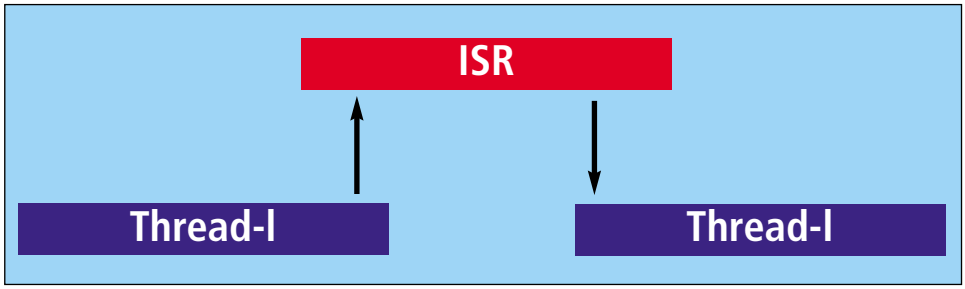

Figure 1: In a Unified Interrupt Architecture, all interrupt-related processing is done in a single ISR, including any required system service calls

approach also reliably prevents interference with the actions of an executing system service call, by not allowing any threads or ISR2 routines, 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. In this paper, we examine the performance implications of each approach on real-time system responsiveness.

Terminology

The following is a list of symbols used to represent the processing performed in each type of RTOS interrupt architecture:

Symbol	Meaning
CC:	Context Create. Create context for ISR2. Typically involves creating a stack frame and alerting the scheduler to schedule it next.
CR:	Context Restore. Restore context of scheduled entity
CS:	Context Save. Saves the context of the running thread.
ID:	Interrupt Dispatcher. Applicable in single interrupt vector architectures like ARM, MIPS, and PowerPC.
ISR:	Interrupt Service Routine. Traditional ISR, allowed to interact with application threads through RTOS services.
ISR1:	Interrupt Service Routine, Part-1. Similar to traditional ISR, but not allowed to interact with RTOS.
ISR2:	Interrupt Service Routine, Part-2. Scheduled entity run in a thread or super-thread context. Allowed to inter act with RTOS services and threads.
ITRA:	Interrupted Thread Running Again.
L:	Lockout. Interrupt lockout time.
S:	Scheduler. Schedule the most important thread or ISR2.
NT:	New Thread running.
1:	Start of ISR processing.
2:	End of ISR processing.
3:	New thread running – preemption case.
4:	Return to interrupted thread.
BOLD	Items that are purely application code.

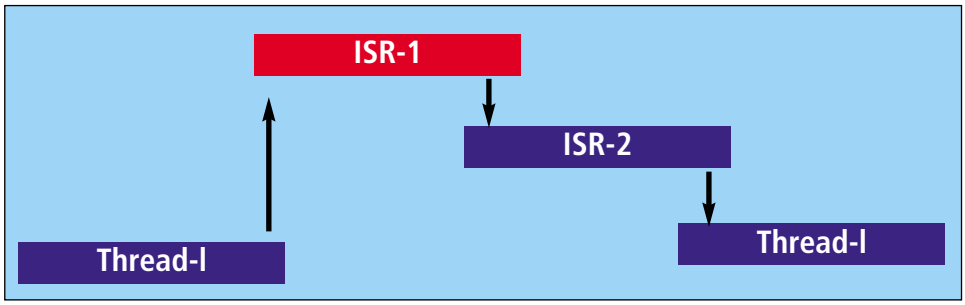

Figure 2: In a Segmented Interrupt Architecture, interrupt processing is split into multiple parts, to avoid system services from within an ISR

The *Unified* Interrupt Architecture

RTOSs 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. To do this, 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 following are the functional components of the unified interrupt architecture:

Case-1: *No Thread Preemption Caused by service calls made by ISR:*

1 2 4
[L][CS][ID][**ISR**][CR][ITRA]

Case-2: *Thread Preemption Caused by service calls made by ISR:*

1 2 3 4
[L][CS][ID][**ISR**][S][CR][NT][CS][S][CR][ITRA]

The *Segmented* Interrupt Architecture

RTOSes that do not disable interrupts during system services cannot make modifications to RTOS data structures from within an ISR, because the ISR might be interrupted by another interrupt, which then might attempt to use the same data structure. To prevent this, segmented architectures basically disable the application scheduler, and 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. Finally, application thread processing can be performed once both ISR1 and ISR2 have been completed. This effectively disables applications while ISR1 and ISR2 are operating, and defers context switching while RTOS data structures are being modified.

The following are the functional components of the segmented interrupt architecture:

Case-1: *No Thread Preemption Caused by service calls made by ISR2:*

1 2
[L][CS][ID][**ISR1**][CC][S][CR][**ISR2**][CS][S]

4
[CR][ITRA]

Case-2: *Thread Preemption Caused by service calls made by ISR2:*

1 2
[L][CS][ID][**ISR1**][CC][S][CR][**ISR2**][CS][S]

3 4
[CR][NT][CS][S][CR][ITRA]

Comparing the Two Approaches

The performance observations in this section are somewhat abstract. The amount of time for RTOS performance objects (e.g. CS, CC, CR, etc.) on most RTOS products will be somewhat similar. However, the 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

No Preemption																
Segmented	L	CS	ID	ISR1	CC	S	CR	ISR2	CS	S	CR	ITRA				
Unified	L	CS	ID	ISR							CR	ITRA				
With Preemption																
Segmented	L	CS	ID	ISR1	CC	S	CR	ISR2	CS	S	CR	NT	CS	S	CR	ITRA
Unified	L	CS	ID	ISR		S	CR					NT	CS	S	CR	ITRA

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

protection or virtual addresses are supported, the time required to process an RTOS block will most definitely be greater. The succeeding comparisons assume that the various RTOSs 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 see Figure 3 (Comparing the Two Approaches).

Total System Overhead Comparison

Ironically, as can be seen in Figure 3, the total system overhead is greater in the RTOS with a segmented interrupt architecture that "never disables interrupts." In both the non-preemption and preemption cases, as shown in Figure 3, the segmented interrupt architecture RTOS introduces an additional (1*CS, 1*CR, 2*S, and 1*CC) of overhead. It appears the most wasteful overhead case is the non-preemptive case, 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 RTOSs must save the full thread context on every interrupt. In the non-preemptive case, the CS and CR performance is much slower in the segmented interrupt RTOS, although this additional overhead has not been factored into this comparison.

Interrupt Response Time

The segmented RTOS architecture is claimed to have an advantage in its ability to respond to interrupts. After all, if interrupts are disabled, response should be negatively impacted, right? The whole idea behind "never disabling interrupts" is to make interrupt response faster. However, while it sounds good, 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 L in the segmented interrupt RTOS 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 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 make "L" a non-zero value and largely defeat the purpose of designing an RTOS with the claim of L approaching zero. 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 CS+ID+ISR* time or the time the application itself disables 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

But while interrupt response is important, a more relevant issue involves the completion of 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 it is another. The total processing required for an interrupt in the segmented RTOS approach is on the order of (2*CS)+CC+S+(2*CR)+ ISR1+ISR2***. This is true for both the preemption and non-preemption cases.

Conversely, the unified interrupt RTOS has a minimal amount of overhead in the non-preemptive case: L+CS**+ISR+CR**. The preemption case for the unified ISR is also low in overhead, requiring only L+CS+ISR+S. In both cases, the total processing required for an interrupt is smaller using the unified interrupt architecture.

*ISR time includes processing up to the point where nested interrupts are enabled.

**CS and CR may be optimized to only save/restore compiler scratch registers if the ISR is written in C or conforms to C register usage.

***If the ISR2 is running with memory protection or in a virtual address space the processing required for CS, CC, and CR will be much greater.

Thread Delays

In segmented interrupt RTOSs 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 one problem, at a 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 RTOSs 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 RTOSs 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 RTOSs in fact "disable application threads" instead, impacting system performance.

Resources

Another issue related to interrupt architecture is the need for system resources. In the unified interrupt RTOS, a separate system stack is used to process all interrupts and nested interrupts. The advantage of this is that the worst case stack usage is taken into account in one place – it doesn't have to be addressed in each thread's stack. Doing so would certainly waste an excessive amount of memory.

The segmented interrupt RTOS could have stack resource problems. Some segmented RTOSs implement the ISR2 as a completely separate entity with its own dedicated stack. Other such RTOSs 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.

Determinism

Assuming L is similar between both types of RTOS interrupt architectures, both are equally deterministic – between 0 and L. The one major difference in determinism occurs in segmented interrupt RTOSs 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. ISR1 executes followed by execution of ISR2. Now suppose further that ISR2 attempts to access the same system object B. Some segmented RTOSs 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, it 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 [L][CS][ID][**ISR1**][CC][S][CR][**ISR2**][CS][S][CR], 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 time [L][CS][ID][**ISR**][S][CR]. Hence, a unified interrupt RTOS can inherently support higher frequency interrupts, by a margin of [CC][ISR-ISR1+ISR2][CS][S][CR].

Complexity

This is a very straightforward area. Unified interrupt RTOSs enable ISRs that more or less behave like ISRs in a standalone environment. As the name implies, segmented interrupt RTOSs divide the application's ISR into two pieces, ISR1 and ISR2. Here is a list of complications or additional work this causes for the developer:

1. Setting up the registration of resources for ISR2

2. Figuring out what part of the ISR goes into ISR1 and ISR2

3. Communication between ISR1 and ISR2. In many cases, ISRs need to pass data received to underlying threads for processing. In a unified interrupt RTOS, this is easily done by calling a queue service with the address of the data packet. It is not nearly as easy in a segmented interrupt RTOS. In such an RTOS, the user must devise his/her own buffering schemes to get data from ISR1 to ISR2 where it can be delivered to the application threads. Ironically, many applications temporarily lockout out interrupts to handle the asynchronous contention between the processing in ISR2 and another of the same interrupt.

Summary

While both Segmented and Unified Interrupt Architecture RTOSs enable deterministic real-time management of an embedded system, there are significant differences between them with regard to the efficiency and simplicity of the resulting system. Segmented RTOSs can accurately claim that they "never disable interrupts" within system services. However, in order to never disable interrupts, they instead 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.