

Interrupts in a preemptive RTOS environment are necessary code that must run when the interrupt occurs, usually preempting any lower priority process such as a task or kernel service.

To make it easier for the developer to build an effective ISR, RTOS developers usually provide all the code necessary to enter and exit from an ISR in a very efficient manner. These code sections of the ISR are often referred to as the prologue and epilogue, respectively. They are usually associated with a unified ISR model but also can be found in segmented models. The prologue takes care of handling the entry into the ISR by saving all or part of the machine context before setting up the user-defined code that services the source of the interrupt. When the interrupt processing is complete, the epilogue is entered to restore the context of the process that is to receive control of the CPU.

In between the prologue and the epilogue a lot of things can be handled in different ways depending on whether the RTOS model is *segmented* or *unified*. There are very successful operating systems available for embedded applications that use one model or the other. Why? Because different RTOS designers have different design goals or policies they try to achieve.

Some RTOS vendors have claimed that they do not turn off interrupts in kernel services (or anytime) by virtue of a segmented ISR approach. Can it be done? Certainly. Is it an effective approach? It depends.

Not disabling interrupts during kernel services is a nice feature to tout but merely hides the fact that there must be another mechanism to protect internal system structures and data from interrupts during critical regions. One advantage of that choice is that it allows an RTOS to have a single set of APIs for kernel services that can be called from any processing level – tasks or threads as well as ISRs. Having only one set of APIs to learn becomes a simplifying element in the use of the RTOS, a very legitimate goal of an RTOS designer. Does it increase system overhead? Perhaps, but not necessarily by an amount that would mitigate against its attributes, including the ease of use.

The unified ISR model is a simple one and that in itself is an advantage. The simplicity derives from the necessity to disable interrupts during critical regions in kernel service to prevent overlapping use by an ISR of an internal system structure that could be in use by the kernel service at the time of the interrupt. The critical region is a period of time within which interrupts will not occur, protecting the internal system tables. However,

it reduces responsiveness to interrupts. It is also not uncommon to see a restricted set of services that is available to the ISR for making connections with the internal structures of the RTOS. Depending on the designer's choices, these ISR-only services may or may not allow the desired degrees of freedom one would prefer in developing an ISR. If not, then the ISR developer has to work around the restrictions, causing undesired cycles in the ISR processing.

The segmented approach, on the other hand, defers the processing that requires connections with the RTOS objects to the second part of the ISR, (called ISR2 by some). Thus, ISR2 logically resides in between the ISR1, the non-deferred part, and the task or thread. In that position, it may have a more complete set of services available to it, but again, that is a designer's choice. Some operating system designs such as Linux, Nucleus Plus make use of segmented ISR models and they have a fairly rich set of services that are available in the ISR2 section.

Different designers will use different ways of doing things in an ISR because they know that efficient handling is important, but it has to be in agreement with overall design goals of the operating system model.

Some RTOS products, notably the RTXC Quadros RTOS, utilize a unified ISR model but augment it by adding a defined layer that sits between the ISR and any task. The code bodies in that layer are called threads (Not to be confused with objects of the same name in other OS models). Those threads can actually be scheduled not only by an ISR but also by tasks and even other threads. When scheduled by an ISR, they effectively become something like an ISR2 but with fewer restrictions. They observe strict rules of priority that put ISR at the highest level, then threads and tasks at the lowest priority. The result is an RTOS design that extends the unified ISR model with its excellent responsiveness by giving it the flexibility of the segmented model but with the ease of use of an API set common to the tasks.

In summary, there is no one way to handle interrupts. Get into an ISR and get out as soon as possible to a level where other interrupts can occur and you can do further processing of data from the interrupting device with a flexible set of services that allow access to the kernel objects you want to use. Do that by the rules the RTOS affords you and you'll be a much happier developer.