Link Service Routines

for better interrupt handling

by Ralph Moore
smx Architect

Introduction

When I set out to design *smx*, I wanted to have very low interrupt latency. Therefore, I created a construct that I called a *Link Service Routine*, or *LSR*, for short. It turns out that some other kernels and RTOSs also have LSRs. However, no two vendors use the same name for these, nor do they work the same.

The *link* part of the LSR name comes from my view of an embedded system as naturally dividing into *foreground* and *background*. Foreground is another problematic word. For a large segment of the IS world it means the operator interface, whereas *background* could mean serial communications, and other potentially high-speed activities. To me, this is a useless definition for embedded systems. The *foreground* is what is most important and that is the interrupt-driven, device-serving part of the system. The human interface and other slow functions are properly put into the *background*. Figure 1 illustrates this concept as well as where LSRs fit into the picture.

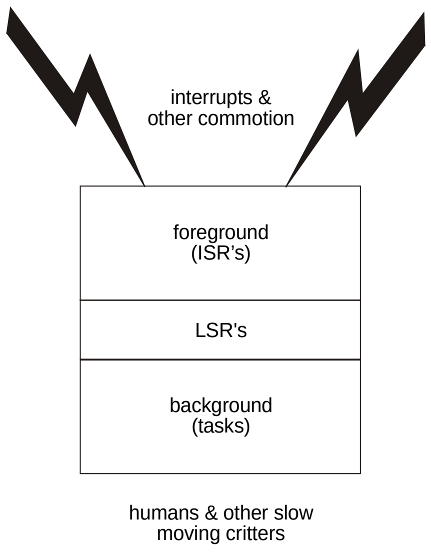


Figure 1: LSR Position

As depicted in Figure 1, LSRs are a layer that links foreground to background. Although initially created to minimize interrupt latency (more about that next), LSRs have proven to have much more importance to embedded system designs. In fact, many *smx* users have stated that LSRs were the most important feature of *smx* for their designs. Hence this article.

Theory

Minimizing Interrupt Latency

Interrupt latency is another term that has multiple meanings. For some people, the interrupt latency of a kernel is the time it takes before a task responds to an interrupt. I prefer the definition that it is the time required before an *ISR responds to an interrupt*. The latter is the sum of three latencies:

interrupt latency = hardware latency + kernel latency + application latency

Here we will focus just on kernel latency, although obviously all three components are important. Kernel latency occurs because the kernel disables interrupts when it enters a *critical section* of code. This is done to assure that another task cannot also enter the critical section due to being started by an interrupt. (A critical section is usually where a kernel resource — e.g. a task control block — is changed or accessed.)

LSRs provide a different mechanism to protect critical sections. It is as follows: ISRs are not permitted to make kernel calls (i.e. request kernel services), except to invoke LSRs. Only LSRs (and tasks) may make kernel calls. Once invoked, an LSR will not run until:

(1) The ISR has completed.
(2) Any *SSR*[1] *(System Service Routine)* that was interrupted has completed.
(3) Any LSR that was interrupted has completed.
(4) The scheduler, if interrupted, has completed.

These rules ensure that there cannot be a conflict for kernel resources. Moreover, it is not necessary to disable interrupts to achieve this!

Although the above rules may seem complex, they are actually implemented with a single nesting-level counter called *srnest*. srnest is incremented when any service routine starts and it is decremented when the service routine stops. Also, srnest is maintained at 1 if the scheduler is running. A new LSR is run only if srnest is 0. Otherwise, the return from a service routine is to the point of interrupt or call.

To complete the implementation picture: LSRs, when invoked, are enqueued in a cyclic buffer called the *LSR queue, lq*. They are run in FIFO order by the scheduler. When all LSRs have run, tasks resume.

[1] An *SSR* is what implements a kernel call.

Flybacks

In a simple world, a kernel using LSRs would have 0 kernel latency because it would never disable interrupts. Unfortunately, things are more complex in the real world, because another parameter, *Interrupt to Task Response* (*ITR*) time, is also important for embedded systems. As it turns out, LSRs are directly in the ITR path:

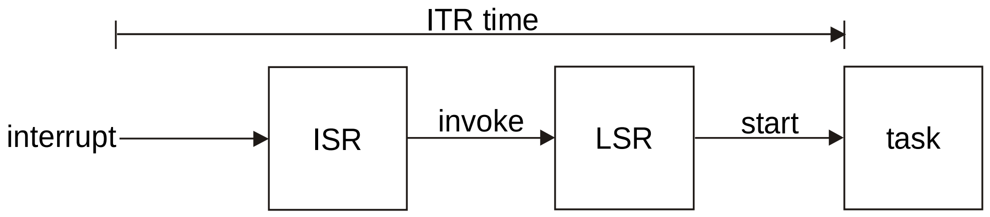


Figure 2: ITR Process

Hence, we cannot allow waiting LSRs to languish for lack of attention. This leads to the concept of *flybacks* in the scheduler. A flyback occurs prior to continuing the current task or dispatching a new one. It consists of the following:

(1) Interrupts are disabled.
(2) The lq counter is tested.
(3) If not zero, interrupts are enabled and the LSR scheduler is entered.
(4) If zero, return is made to the current or new task and interrupts are enabled after the return.

The flybacks in the scheduler ensure that LSRs will not be left waiting while control passes to a task. (Once the task is running, only another interrupt or system call could cause recognition that an LSR is ready to run.) However, as is apparent from (1) above, flybacks do introduce kernel latency. Fortunately, this latency is on the order of only 10 to 30 machine instructions. The worst-case kernel interrupt latency is typically less than that of the processor itself.

Important Questions

At this point, the attentive reader probably has a few questions:

(1) Why is interrupt latency more important than ITR time?
(2) What are the other benefits of LSRs?
(3) Why don't LSRs have priorities?

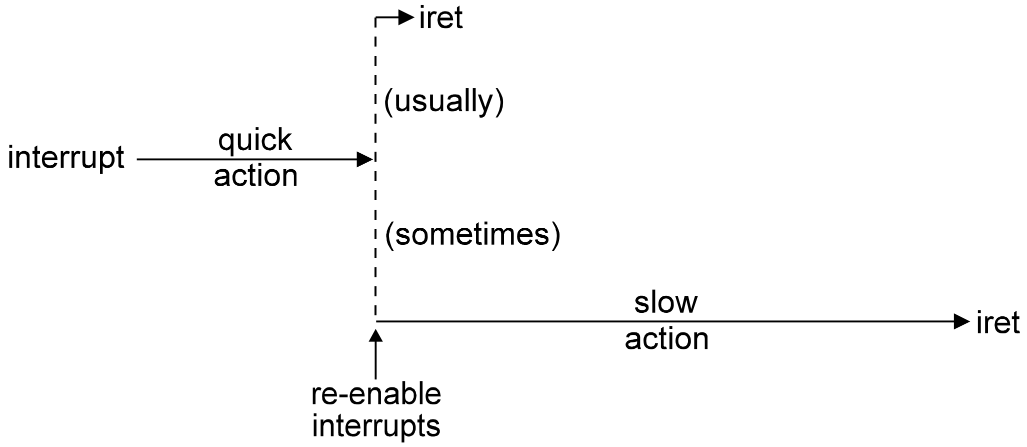
Lets address these questions in order:

(1) Interrupt latency is more important than ITR time because interrupt latency slows down all ISRs — not just those associated with tasks. This can cause serious design limitations. Suppose, for example, we need an ISR to do nothing but count random pulses. Too much interrupt latency could cause pulses to be missed. On the other side of the coin, LSRs obviously add to ITR time (see figure 2). But, where this is critical, an LSR can do the processing instead of a task. This leads to the answer to question (2).

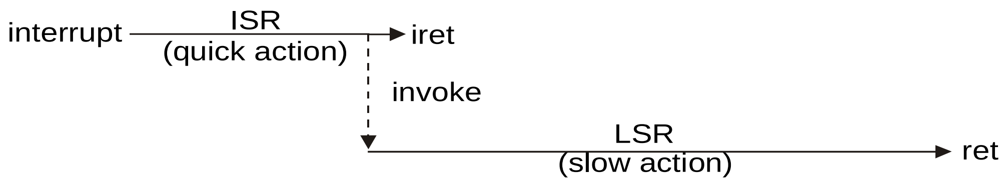
(2) LSRs can serve as *foreground tasks*. This is appealing because:

(1) They run ahead of (i.e. have higher priority than) all background tasks.
(2) They are stripped-down, minimal overhead constructs.

Furthermore, using LSRs encourages creating minimal ISRs. Consider the structure of a typical ISR without LSRs:



Interrupts are re-enabled as soon as possible to minimize application latency. Most of the time, a quick action is performed (e.g. stuff a character into a buffer) and the ISR returns. Sometimes an additional slow action occurs (e.g. send the buffer to a task). Using LSRs results in the following structure:



The ISR performs the quick action and sometimes invokes an LSR, which, at a later time, performs the slow action.

(3) LSRs do not have priorities for two reasons:

1. (1) To minimize overhead.
2. (2) From the perspective of tasks, all LSRs have the same priority, anyway (i.e. higher).

From a system implementation viewpoint, designers have three priorities to work with:

(1) ISRs	–	highest
(2) LSRs	–	middle
(3) Tasks	–	lowest

This permits a lot of flexibility. However, it is still true that many average LSRs could be enqueued ahead of a truly important LSR thus causing it to miss a deadline. On the other hand, a FIFO LSR queue maintains true event sequencing. Thus, bursts of activity can be correctly sorted out later as the processor catches up. We plan to add LSR priorities in the next major smx release.

Minimal Overhead

The minimal overhead of LSRs is due not only to their lack of priorities and hence FIFO dispatching, but it also is due to the fact that LSRs run in the *context* of the current task — i.e. they share its stack and registers. Hence, stack switching is not necessary. Nor is it necessary to save the full task state. These, together, save quite a bit of time. There is a downside to this sharing, in that LSRs cannot wait for events. If an LSR were to "suspend" itself on a semaphore, for example, what actually would happen is that the LSR and the current task would both be suspended. That would be a surprise for the current task (which could be any task doing anything) and is clearly not acceptable. However, the no-wait limitation turns out not to be a serious limitation — it is always possible to shift the blame to a task that can wait.

An Example

I will conclude with an example. We'll draw from serial communications since most people are familiar with it. The example is illustrated in Figure 3.

The input channel of the UART interrupts each time a character is received. The ISR responding to the interrupt and puts the character into a pipe (the PPUT_CHAR() macro is one of the few kernel services an ISR can use). When the end of the message is detected, the ISR invokes an LSR. The LSR, which runs later, removes characters from the pipe with the PGET_CHAR() macro and performs an error check. It also assembles an internal message (e.g. with protocol fields removed). If the error check passes, the message is sent to an input exchange where a processing task waits. Also, the LSR sends an acknowledgment character, ACK, to the UART output channel. If the error check fails, the pipe is emptied, but no message is sent to the input exchange, and a negative acknowledgment character, NAK, is sent to the UART output channel.

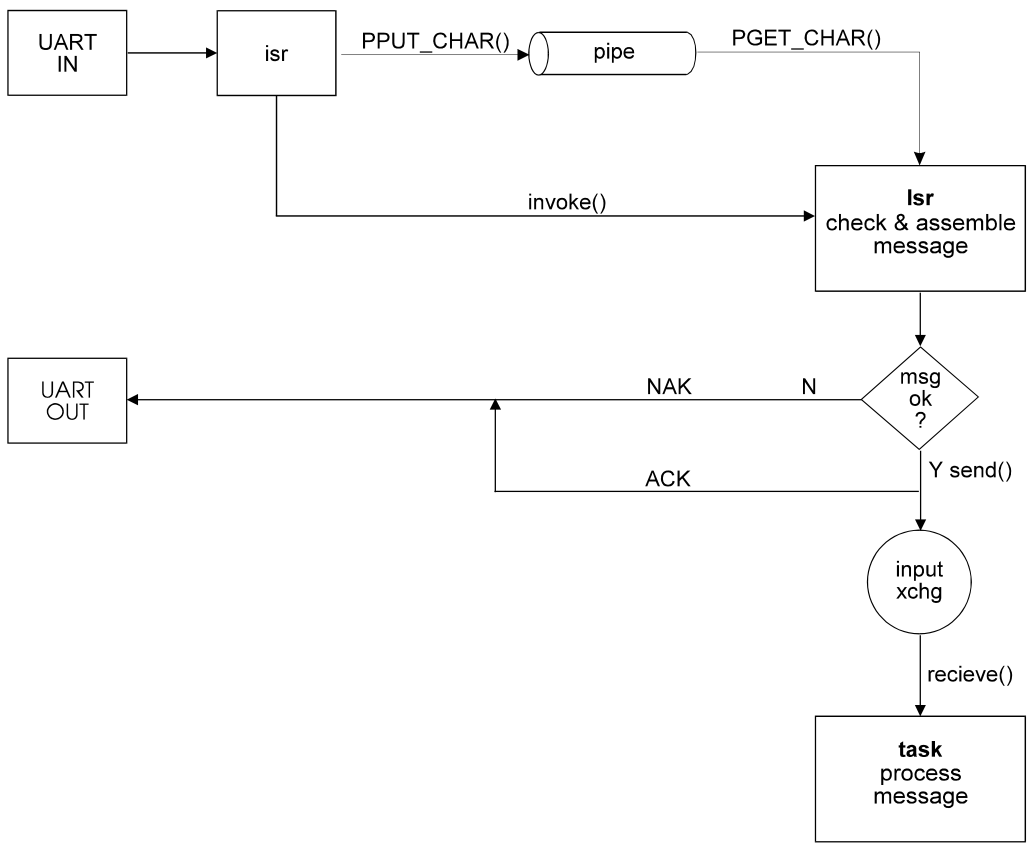


Figure 3: Example

A nice advantage of an LSR implementation is its ability to handle peak loads. Suppose that several messages arrive in rapid succession on this and other channels. Consequently, the processor becomes overloaded, and the LSR is invoked many times before it can run. No problem — as long as the pipe and the LSR queue are long enough. The LSR will simply run multiple times and each time process a message from the pipe (of course, in this case there would have to be a numbered acknowledgment scheme corresponding to numbered messages).

Conclusion

LSRs were initially introduced into smx to achieve very low interrupt latency for smx, itself. However, they soon proved equally beneficial as a mechanism for deferred interrupt processing. (See the white paper on this for more discussion.) As a side benefit, they also have a property to smooth out peak interrupt loads. LSRs have been successfully used in hundreds of embedded systems over the past 15 years. In the future, we will be introducing LSR priorities and making LSRs independent of smx.