Deferred Interrupt Processing

improves system response

by Ralph Moore
smx Architect

Introduction

All RTOS kernels must deal with the problem of protecting kernel data structures from damage caused by asynchronous interrupts. In a recent article, William Lamie[1] has categorized RTOSs by how they handle interrupts:

1. *Unified Interrupt Architecture* wherein system services may be accessed from ISRs.
2. *Segmented Interrupt Architecture* wherein system services may not be accessed from ISRs.

Most commercial kernels (e.g. ThreadX, MQX, and VxWorks) fall into the first category. Because system services can be called from ISRs, these kernels must disable interrupts whenever kernel data structures are being changed. If, for example, a system service is removing an object from a queue, ISRs must be blocked from adding objects to the queue, or else the queue could be damaged. Generally, this requires disabling all interrupts for the entire code section (called a *critical section*) during which a kernel object is being changed.

A few kernels (e.g. *smx* and Velocity) fall into the second category. These kernels provide a mechanism to defer calling kernel services until after the ISR has finished and any interrupted system service has also finished. Consequently, these kernels do not need to disable interrupts in order to protect kernel objects.

Lamie goes on to discuss the advantages of the unified architecture versus the segmented architecture and concludes that the former is preferable for embedded systems.

In another recent article, David Kleidermacher[2] discusses the need to keep ISRs simple and short and to disable interrupts as little as possible. He distinguishes between RTOSs that have a *Simple* architecture and those that have an *Advanced* architecture. These match categories 1 and 2 above, respectively. He presents an Advanced architecture wherein ISRs are split into two parts. The first part does the minimum necessary processing to handle the hardware and to schedule the second part. The second part is performed later, when interrupts are enabled, by a call back mechanism in the scheduler. Unfortunately, not much detail is provided concerning this mechanism.

We fully agree with Kleidermacher concerning the importance of minimizing interrupt latency in the kernel. We start with a discussion of why this is important in real-time

embedded systems, and then contrast the two types of kernels with respect to it. We follow this with a presentation, and code example, of an efficient mechanism for deferring interrupt processing. This uses what we call *LSRs*. LSRs introduce the additional benefit of smoothing peak interrupt overloads, and thus making applications more rugged.

Theory

The Need for a Good Defense against Interrupts

As a rule, it is best to keep ISRs as short as possible and to not enable interrupts during ISRs. Generally, it is not possible to make ISRs reentrant, without unacceptable performance penalties. Therefore, the same ISR must be blocked from running again until it has finished its current run. This is normally done by the hardware, which masks all further interrupts or, on some processors, those at the current and lower levels. Unmasking the interrupt and reenabling the interrupt source are usually the last things an ISR does. If the ISR takes too long to run, the next interrupt may be lost. This problem can occur if the ISR was excessively delayed getting started by kernel and application interrupt latency, higher priority ISRs running, and the next interrupt occurring sooner than normal.

A missed interrupt can be a nightmare to find because it may occur only once in a blue moon. We know that we must be concerned about such rare events because experience (and Murphy's Law) teaches us that they will occur at the worst possible times. Even if this kind of bug occurs frequently, it can be hard to find because normal debugging techniques are not tuned to detecting it. (In fact, debuggers normally cause interrupts to be missed.) The best defense is to design this problem out of the system, from the outset. Keeping ISRs as short as possible helps to do this.

In most processors, the hardware disables all interrupts when an interrupt occurs. This allows the programmer to block higher priority interrupts as well as the same and lower priority interrupts, by not reenabling interrupts until the end of the ISR. If an ISR is very short, this may be acceptable, even though higher priority ISRs are delayed. Doing so eliminates interrupt nesting, which is another cause of problems: (1) Interrupt nesting increases stack usage and can cause unexpected stack overflows for the unwary programmer – another hard problem to track. (2) There can be resource-sharing problems even though each interrupt invokes a different ISR. For example, without realizing it, two ISRs might be using a common global variable disguised by a macro, or even worse, there might be some subtle, undocumented hardware interaction.

In this emerging era of complex hardware hacks, it is hard to trust the hardware, anymore. Therefore, unless you want to be burning late-night oil, defensive interrupt programming is a must. Regarding nested interrupts, this means they should be avoided, if possible.

Unified Interrupt Kernels

Because of the above problems, many seasoned embedded programmers shudder at the thought of introducing a Unified Interrupt Kernel into their designs.

A simple example illustrates the reason for this: In most kernels, the *ready queue* consists of a doubly linked list of Task Control Blocks (TCBs). The TCBs are linked in descending priority order. TCBs of the same priority are linked by descending waiting-time. Any kernel operation, which causes a task to become ready, must search the ready queue, from its start to the correct position at which to link the task's TCB. Interrupts must be disabled, all during that time, in order to protect the ready queue from damage. The ready queue may contain only a few tasks or it may contain all tasks of the system. Sometimes, the new task will be placed at the start, sometimes in the middle, and sometimes at the very end. The devil is in this uncertainty. From time to time, interrupts may be disabled for too long and an interrupt will be missed.

Making this problem even worse, is that it is not possible to enable higher priority interrupts that do not use the particular resource being protected. Such selectivity is not possible. Kernel services must disable all interrupts, when in critical sections, because there is no practical way to know which ISRs make which kernel calls. Thus, the interrupt latency introduced by kernel services directly impacts the most important, highest priority interrupts – even if they make no kernel calls!

Service Routine Definitions

- **Interrupt Service Routines (ISRs)** are driven by external interrupts that are asynchronous to everything else. If not blocked, an ISR can run between any two *machine* instructions. (This can catch a C programmer unawares because it is easy to fall into the trap of thinking that ISRs can run only between C statements.) ISRs are often written in assembly language to optimize their performance. They take no parameters and have no return values.

- **Link Service Routines (LSRs)** are invoked by ISRs to do *deferred interrupt processing*. They are written as ordinary C functions with a single parameter and no return value. LSRs run with interrupts enabled, unless disabled by the programmer. LSRs are permitted to call all kernel services, but cannot wait for results. Link Service Routines are so named because they link ISRs to tasks.

- **System Service Routines (SSRs)** are the routines that perform system services. They are written as ordinary C functions that take parameters and return results.

Segmented Interrupt Kernels

Segmented Interrupt Kernels avoid the above problem because they do not permit calling kernel services (i.e. SSRs) from ISRs. Instead, *Link Service Routines* (*LSRs)* are *invoked* to perform deferred interrupt processing and to call kernel services. Kernel services cause tasks to be scheduled to run, by sending messages to them, signaling semaphores where they are waiting, resuming them directly, etc. Hence, LSRs are so named because they *link* foreground activity (interrupts and ISRs) to background activity (tasks). Without such linkage, nothing much would happen!

LSRs run only after all ISRs have completed *and* after any interrupted SSR has also finished. Because of this, there are no *critical sections* within SSRs that need to be protected from interrupts. Therefore, SSRs run with interrupts fully enabled and introduce no *interrupt latency*. LSRs, too, run with interrupts fully enabled, unless the application programmer chooses otherwise. (In which case, he will be aware of the interrupt latency he has introduced and it is up to him to keep it under control.)

Like ISRs, LSRs run in the context of the current task. This means that they use its stack and they save only the registers that they use. Thus, compared to task switching, LSR switching has much lower overhead. This makes LSRs ideal for deferred interrupt processing. The following code illustrates this:

Code Example

```
_rx_isr_shell:   ISR_ENTER
                 jsr        _rx_isr
                 ISR_EXIT

void rx_isr (void)
{
  // Clear RXF interrupt event handled by this isr.
  frp->eir = EIR_RXF;

  while(!(rbd[crx].flags & RxBD_E))   // Process all received frames.
  {
    if (!((rbd[crx].flags & RxBD_L) && (rbd[crx].flags & RxBD_ERROR)) ||
        (rbd[crx].flags & RxBD_TR))
    {
       // Frame ok -- accept packet
    }
    else
    {
      INVOKE(rx_error_lsr, rbd[crx].flags);  // Frame error -- reject packet.
    }
    if(++crx == NUM_RX_BD)
      crx = 0;
  }
}

static void rx_error_lsr(u32 flags)
{
```

```
  if ...

  if (flags & RxBD_RO1)
  {
    ERROR_MSG1("Out of receive buffers\n");
    error_count++;
  }
}
```

This example is drawn from an Ethernet driver for a ColdFire® processor. The receive interrupt causes control to vector to _rx_isr_shell, an interrupt shell, which is written in assembly language. It brackets the core of the ISR, written in C, with enter and exit ISR macros. (More on these later.) A portion of the receive ISR is shown. Note that it resets the interrupt source, then tests for any errors that the Ethernet controller has detected and recorded in the receive buffer descriptor (rbd) flags field. If an error is found, the rx_error_lsr() LSR is invoked to deal with it at a later time. Therefore, this and other interrupts are not delayed while the error is processed and its error message is displayed.

Note that the flags field is passed to rx_error_lsr() as a parameter. Thus, when it later runs, rx_error_lsr() has all it needs to do its job – namely: determine what error occurred, output an appropriate error message, and increment the error counter. This is a good example of using LSRs effectively – an error in one packet does not cause other packets or other interrupts to be missed. For this particular driver, the data in and data out paths are also handled by LSRs.

In this example, before returning to the ISR shell, the receive ISR checks if there is another received packet in the descriptor ring and processes it, if so. If this packet also has errors, rx_error_lsr() will be invoked again with the error flags for it. When the second invocation of rx_error_lsr() runs, it will report the errors for the second packet. This is a good example of the value of being able to invoke the same LSR multiple times, before it runs. Finally, rx_isr() returns to the ISR shell and the exit ISR macro runs.

How LSRs Work

The ISR_ENTER and ISR_EXIT macros determine when LSRs run. A nesting counter, *srnest*, is incremented by ISR_ENTER. It keeps track of ISR nesting. If srnest is greater than one, ISR_EXIT returns to the interrupted ISR. There is very little overhead, in this case. If srnest is one (meaning no more ISRs to run) and there are LSRs to run, interrupts are enabled and the LSR scheduler is called. It runs all waiting LSRs, then the task scheduler runs to determine if a higher priority task is ready to run. If so, the current task is suspended and the higher priority task is dispatched. Since ISRs cannot call kernel services, a new task can become ready to run only if an LSR has called a kernel service, which resumed or started it.

The INVOKE() macro stores the LSR address and a 32-bit parameter in the LSR queue. This queue is a cyclic queue of a size specified by the programmer. It can store as many LSRs as necessary to handle peak loads. An important feature of LSRs is that, unlike

tasks, the same LSR may be scheduled multiple times, before running. Each time it can be scheduled with a different parameter. The parameter could be a pointer to a block that contains captured information needed by the LSR. This makes deferred interrupt processing feasible because each instantiation of the same LSR is given its own data to process. These could be contents of device registers as well as global variables -- i.e. a freeze frame.

An additional benefit of the LSR mechanism is that it has the ability to smooth out interrupt overloads. When hit with a burst of interrupts, ISRs run in rapid succession, performing minimal processing and invoking LSRs to complete needed processing later. When the burst of interrupts is over and all ISRs have run, then LSRs run in the order invoked. Thus they maintain *temporal integrity*. The LSRs process freeze frames of what happened in real time. While running late, the external manifestation is some sluggishness, but not loss of control nor system breakdown. Thus, LSRs impart a certain ruggedness to systems using them.

Summary

We have discussed the importance of minimizing interrupt latency and of doing defensive interrupt programming in real-time embedded applications. If tight control cannot be maintained here, the integrity of the whole system is undermined. In addition, tracking bugs in interrupt handling code is especially difficult. Therefore bugs must be designed out, if at all possible. An important principle to follow is to minimize the size of ISRs – non-essential processing is best deferred in order to avoid missing new interrupts. It is particularly important to defer kernel service calls because they can be quite lengthy. Also, if kernel calls are permitted from ISRs they can significantly increase interrupt latency for the system. The Link Service Routine (LSR*)* offers a good mechanism for doing deferred interrupt processing. It has been presented in detail, with an example. LSRs have been used, with good results, in hundreds of embedded applications over the past 15 years. For more details on LSRs, see reference 3.

References:

1. Lamie, William, “Pardon the Interruption”, p58, Information Quarterly Vol. 3, Number 4, 2004.
2. Kliedermacher, David, “Minimizing Interrupt Response Time”, p52, Information Quarterly Vol. 4, 2005.
3. Moore, Ralph, “Link Service Routines” http://www.smxrtos.com/articles/lsr_art/lsr_art.htm