Nested Interrupts

Are Nested Interrupts Needed?

Most NuttX architectures do not support nested interrupts: Interrupts are disabled when the interrupt is entered and restored when the interrupt returns. Being able to handle nested interrupt is critical in simple architectures where a lot of interrupt level processing is performed: In this case, you can prioritize interrupts and assure that the highest priority interrupt processing is not delayed by lower level interrupt processing.

In an RTOS model, however, all interrupt processing should be as brief as possible; any extended processing should be deferred to a user task and not performed in the interrupt handler. However, you may find a need to have nested interrupt handling in NuttX too. The lack of support of nested interrupts is not inherently an issue with NuttX and need not be the case; it should be a simple matter to modify the interrupt handling so that interrupts are nested.

Layered Interrupt Handling Architecture

Interrupt handling occurs in several files. In most implementations, there are several layers of interrupt handling logic:

1. Some low-level logic, usually in assembly language, that catches the interrupt and determines the IRQ number. Consider `arch/arm/src/armv7-m/up_exception.S` as an example for the Cortex-M family.
2. That low-level logic than calls some MCU-specific, intermediate level function usually called `up_doirq()`. An example is `arch/arm/src/armv7-m/up_doirq.c`.
3. That MCU-specific function then calls the NuttX common interrupt dispatching logic `irq_dispatch()` that can be found at sched/irq_dispatch.c

How to Implement Nested Interrupts in the Layered Interrupt Handling Architectgure

The logic in these first two levels that would have to change to support nested interrupt handling. Here is one technical approach to do that:

1. Add an global variable, say `g_nestlevel`, that counts the interrupt nesting level. It would have an initial value of zero; it would be incremented on each interrupt entry and decremented on interrupt exit (making sure that interrupts are disabled in each case because incrementing and decrementing are not usually atomic operations).
2. At the lowest level, there is usually some assembly language logic that will switch from the user's stack to a special interrupt level stack. This behavior is controlled `CONFIG_ARCH_INTERRUPTSTACK`. The logic here would have to change in the following way: If `g_nestlevel` is zero then behave as normal, switching from the user to the interrupt stack; if `g_nestlevel` is greater than zero, then do not switch stacks. In this latter case, we are already using the interrupt stack.
3. In the middle-level, MCU-specific is where the `g_nestlevel` would be increment. And here some additional decision must be made based on the state of `g_nestlevel`. If `g_nestlevel` is zero, then we have interrupted user code and we need to handle the context information specially and handle interrupt level context switches. If `g_nestlevel` is greater than zero, then the interrupt handler was interrupt by an interrupt. In this case, the interrupt handling must always return to the interrupt handler. No context switch can occur here. No context switch can occur until the outermost, nested interrupt handler returns to the user task.
4. You would also need to support some kind of *critical section* within interrupt handlers to prevent nested interrupts. For example, within the logic of functions like `up_block_task()`. Such logic must be atomic in any case.

NOTE 1: The ARMv7-M could also be configured to use separate MSP and PSP stacks with the interupt processing using the MSP stack and the tasks all using the PSP stacks. This is not compatible with certain parts of the existing design and would be more effort, but could result in a better solution.

NOTE 2: SMP has this same issue as 2 but it is addressed differently: With SMP there is an array of stacks indexed by the CPU number so that all CPUs get to have an interrupt stack. See for example, LC823450 [https://bitbucket.org/nuttx/nuttx/src/ca4ef377fb789ddc3e70979b28acb6730ff6a98c/arch/arm/src/lc823450/chip.h#lines-92] or i.MX6 [https://bitbucket.org/nuttx/nuttx/src/ca4ef377fb789ddc3e70979b28acb6730ff6a98c/arch/arm/src/imx6/chip.h#lines-102] SMP logic.

A generic up_doirq() might look like the following. It can be very simple because interrupts are disabled:

```
uint32_t *up_doirq(int irq, uint32_t *regs)
{
  /* Current regs non-zero indicates that we are processing an interrupt;
   * current_regs is also used to manage interrupt level context switches.
   */

  current_regs = regs;

  /* Deliver the IRQ */

  irq_dispatch(irq, regs);

  /* If a context switch occurred while processing the interrupt then
   * current_regs may have change value.  If we return any value different
   * from the input regs, then the lower level will know that a context
   * switch occurred during interrupt processing.
   */

  regs = (uint32_t*)current_regs;
  current_regs = NULL;
  return regs;
}
```

What has to change to support nested interrupts is:

1. If we are nested, then we must retain the original value of current_regs. This will be need when the outermost interrupt handler returns in order to handle interrupt level context switches.
2. If we are nested, then we need to always return the same value of regs that was received.

So the modified version of up_doirq() would be as follows. Here we assume that interrupts are enabled.

```
uint32_t *up_doirq(int irq, uint32_t *regs)
{
  irqstate_t flags;

  /* Current regs non-zero indicates that we are processing an interrupt;
   * regs holds the state of the interrupted logic; current_regs holds the
   * state of the interrupted user task.  current_regs should, therefor,
   * only be modified for outermost interrupt handler (when g_nestlevel == 0)
   */

  flags = irqsave();
  if (g_nestlevel == 0)
    {
      current_regs = regs;
    }
  g_nestlevel++
  irqrestore(flags);

  /* Deliver the IRQ */

  irq_dispatch(irq, regs);

  /* Context switches are indicated by the returned value of this function.
   * If a context switch occurred while processing the interrupt then
   * current_regs may have change value.  If we return any value different
   * from the input regs, then the lower level will know that a context
   * switch occurred during interrupt processing.  Context switching should
   * only be performed when the outermost interrupt handler returns.
   */

  flags = irqsave();
  g_nestlevel--;
  if (g_nestlevel == 0)
    {
      regs = (uint32_t*)current_regs;
      current_regs = NULL;
    }

  /* Note that interrupts are left disabled.  This needed if context switch
   * will be performed.  But, any case, the correct interrupt state should
   * be restored when returning from the interrupt.
   */

  return regs;
}
```

NOTE: An alternative, cleaner design might also be possible. If one were to defer all context switching to a PendSV handler, then the interrupts could vector to the *do_irq()* logic and then all interrupts would be naturally nestable.

SVCall vs PendSV

An issue that may be related to nested interrupt handling is the use of the SVCall exceptions in NuttX. The SVCall exception is used as a classic software interrupt in NuttX for performing context switches, user- to kernel-mode changes (and vice versa), and also for system calls when NuttX is built as a kernel.

SVCall exceptions are never performed from interrupt level, handler mode processing; only from thread mode logic. The SVCall exception is used as follows to perform the system call:

- All interrupts are disabled: There are a few steps the must be performed in a critical section. Those setups and the SVCall must work as a single, uninterrupted *atomic* action.
- A special register setup is put in place: Parameters are passed to the SVCall in registers just as with a normal function call.
- The Cortex SVC instruction is executed. This causes the SVCall exception which is dispatched to the SVCall exception handler. This exception *must* occur while the input register setup is in place; it cannot be deferred and perform at some later time. The SVCall exception handler decodes the registers and performs the requested operation. If no context switch occurs, the SVCall will return to the caller immediately.
- Upon return interrupts will be re-enabled.

So what does this have to do with nested interrupt handling? Since interrupts are disabled throughout the SCVall sequence, nothing really. However, there are some concerns because if the BASEPRI is used to disable interrupts then the SVCall exception must have the highest priority: The BASEPRI register is set to disable all interrupt except for the SVCall.

The motivation for supporting nested interrupts is, presumably, to make sure that certain high priority interrupts are not delayed by lower processing interrupt handling. Since the SVCall exception has highest priority, it will delay all other interrupts (but, of course, disabling interrupt also delays all other interrupts).

The PendSV exception is another mechanism offered by the Cortex architecture. It has been suggested that some of these issues with the SVCall exception could be avoided by using the PendSV interrupt. The architecture that would use the PendSV exception instead of the SVCall interrupt is not clear in my mind. But I will keep this note here for future reference if this were to become as issue.

What Could Go Wrong?

Whenever you deal with logic at software hardware interface, lots of things can go wrong. But, aside from that general risk, the only specific NuttX risk issue is that you may uncover some subtle interrupt level logic that assumes that interrupts are already disabled. In those cases, additional critical sections may be needed inside of the interrupt level processing. The likelihood of such a thing is probably pretty low, but cannot be fully discounted.
