High Performance, Zero Latency Interrupts

Generic Interrupt Handling

NuttX includes a generic interrupt handling subsystem that makes it convenient to deal with interrupts using only IRQ numbers. In order to integrate with this generic interrupt handling system, the platform specific code is expected to collect all thread state into an container, `struct xcptcontext`. This container represents the full state of the thread and can be saved, restored, and exchanged as a *unit of thread*.

While this state saving has many useful benefits, it does require processing time. It was reported to me that this state saving required about two microseconds on an STM32F4Discovery board. That added interrupt latency might be an issue in some circumstance.

Terminology: The concepts discussed in this Wiki are not unique to NuttX Other RTOS have similar concepts but will use different terminology. The Nucleus [https://www.embedded.com/design/operating-systems/4461604/Interrupts-in-the-Nucleus-SE-RTOS] RTOS, for example use the terms *Native* and *Managed* interrupts.

Bypassing the Generic Interrupt Handling

Most modern MCUs (such as the ARM Cortex-M family) receive and dispatch interrupts through a *vector table*. The vector table is a table in memory. Each entry in the table holds the address of an interrupt handler corresponding to different interrupts. When the interrupt occurs, the hardware fetches the corresponding interrupt handler address and gives control to the interrupt handler.

In the implementation of the generic interrupt handler, these vectored interrupts are not used as intended by the hardware designer. Rather, they are used to obtain an IRQ number and then to transfer control to the common, generic interrupt handling logic.

One way to achieve higher performance interrupts and still retain the benefits of the generic interrupt handling logic is to simply replace an interrupt handler address in the vector table with a different interrupt handler; one that does not vector to the generic interrupt handling logic logic, but rather to your custom code.

Often, the vector table is in ROM. So you can hard-code a special interrupt vector by modifying the ROM vector table so that the specific entry points to your custom interrupt handler. Or, if the architecture permits, you can use a vector table in RAM. Then you can freely attach and detach custom vector handlers by writing directly to the vector table. The ARM Cortex-M port provides interfaces to support this mode when the CONFIG_ARCH_RAMVECTORS option is enabled.

So what is the downside? There are two:

- Your custom interrupt handler will not have collected its state into the `struct xcptcontext` container. Therefore, it cannot communicate with operating system. Your custom interrupt handler has been taken “out of the game” and can no longer work with the system.
- If your custom interrupt is truly going to be *high performance* then you will also have to support nested interrupts! The custom interrupt must have a high priority and must be able interrupt the generic interrupt handling logic. Otherwise, it will be occasionally delayed when there is a collision between your custom interrupt and other, lower priority interrupts.

Getting Back into the Game

As mentioned, the custom interrupt handler can not use most of the service of the OS since it has not created a `struct xcptcontext` container. So it needs a mechanism to “get back into the game” when it needs to interact with the operating system to, for example, post a semaphore, signal a thread, or send a message.

The ARM Cortex-M family supports a special way to do this using the *PendSV* interrupt:

- The custom logic would connect with the *PendSV* interrupt using the standard `irq_attach()` interface.
- In the custom interrupt handler, it would schedule the *PendSV* interrupt when it needs to communicate with the OS.
- The *PendSV* interrupt is dispatched through generic interrupt system so when the attached *PendSV* interrupt is handled, it will be in a context where it can perform any necessary OS interactions.

With the ARMv7_M architecture, the *PendSV* interrupt can be generated with:

```
up_trigger_irq(NVIC_IRQ_PENDSV);
```

On other architectures, it may be possible to do something like a software interrupt from the custom interrupt handler to accomplish the same thing.

The custom logic would be needed to communicate the events of interest between the high priority interrupt handler and *PendSV* interrupt handler. A detailed discussion of that custom logic is beyond the scope of this Wiki page.

Nested Interrupt Handling

Some general notes about nested interrupt handling are provided in another Wiki page. In this case, handling the nested custom interrupt is simpler because the generic interrupt handler is not re-entered. Rather, the generic interrupt handler must simply be made to co-exist with the custom interrupt interrupt handler.

Modifications may be required to the generic interrupt handling logic to accomplish. A few points need to be made here:

- The MCU should support interrupt prioritization so that the custom interrupt can be scheduled with a higher priority.
- The generic interrupt handlers currently disable interrupts during interrupts. Instead, they must be able to keep the custom interrupt enabled throughout interrupt process but still prevent re-entrancy by other standard interrupts (This can be done by setting an interrupt base priority level in the Cortex-M family).
- The custom interrupt handler can now interrupt the generic interrupt handler at any place. Is the logic safe in all cases to be interrupted? Sometimes interrupt handlers place the MCU in momentarily perverse states while registers are being manipulated. Make sure that it is safe to take interrupts at any time (or else keep the interrupts disabled in the critical times).
- Will the custom interrupt handler have all of the resources it needs in place when it occurs? Will it have a valid stack pointer? (In the Cortex-M implementation, for example, the MSP may not be valid when the custom interrupt handler is entered).

Some of these issues are complex and so you should expect some complexity in getting the nested interrupt handler to work.

Cortex-M3/4 Implementation

Such high priority, nested interrupt handler has been implemented for the Cortex-M3/4 families. The following paragraphs will summarize that implementation.

Configuration Options

CONFIG_ARCH_HIPRI_INTERRUPT

If *CONFIG_ARMV7M_USEBASEPRI* is selected, then interrupts will be disabled by setting the BASEPRI register to *NVIC_SYSH_DISABLE_PRIORITY* so that most interrupts will not have execution priority.

SVCall must have execution priority in all cases.

In the normal cases, interrupts are not nest-able and all interrupts run at an execution priority between *NVIC_SYSH_PRIORITY_MIN* and *NVIC_SYSH_PRIORITY_MAX* (with *NVIC_SYSH_PRIORITY_MAX* reserved for SVCall).

If, in addition, *CONFIG_ARCH_HIPRI_INTERRUPT* is defined, then special high priority interrupts are supported. These are not "nested" in the normal sense of the word. These high priority interrupts can interrupt normal processing but execute outside of OS (although they can "get back into the game" via a PendSV interrupt).

Disabling the High Priority Interrupt

In the normal course of things, interrupts must occasionally be disabled using the `up_irq_save()` inline function to prevent contention in use of resources that may be shared between interrupt level and non-interrupt level logic. Now the question arises, if we are using the `BASEPRI` to disable interrupts and have high priority interrupts enabled (`CONFIG_ARCH_HIPRI_INTERRUPT=y`), do we disable all interrupts except SVCall (we cannot disable SVCall interrupts)? Or do we only disable the "normal" interrupts?

If we are using the BASEPRI register to disable interrupts, then the answer is that we must disable *ONLY* the *normal* interrupts. That is because we cannot disable `SVCALL` interrupts and we cannot permit `SVCAll` interrupts running at a higher priority than the high priority interrupts. Otherwise, they will introduce jitter in the high priority interrupt response time.

Hence, if you need to disable the high priority interrupt, you will have to disable the interrupt either at the peripheral that generates the interrupt or at the interrupt controller, the `NVIC`. Disabling global interrupts via the `BASEPRI` register cannot affect high priority interrupts.

Dependencies

- *CONFIG_ARCH_HAVE_IRQPRIO*. Support for prioritized interrupt support must be enabled.
- Floating Point Registers. If used with a Cortex-M4 that supports hardware floating point, you cannot use hardware floating point in the high priority interrupt handler UNLESS you use the common vector logic that supports saving of floating point registers on all interrupts.

Configuring High Priority Interrupts

How do you specify a high priority interrupt? You need to do two things:

First, You need to change the address in the vector table so that the high priority interrupt vectors to your special C interrupt handler. There are two ways to do this:

- If you select *CONFIG_ARCH_RAMVECTORS*, then vectors will be kept in RAM and the system will support the interface: *int up_ramvec_attach(int irq, up_vector_t vector)*. That interface can be used to attach your C interrupt handler to the vector at run time.
- Alternatively, you could keep your vectors in FLASH but in order to this, you would have to develop your own custom vector table.

Second, you need to set the priority of your interrupt to NVIC to *NVIC_SYSH_HIGH_PRIORITY* using the standard interface: *int up_prioritize_irq(int irq, int priority);*

Example Code

You can find an example that tests the high priority, nested interrupts in the NuttX source:

- *nuttx/boards/arm/stm32/viewtool-stm32f107/README.txt*. Description of the configuration
- *nuttx/boards/arm/stm32/viewtool-stm32f107/highpri*. Test configuration
- *nuttx/boards/arm/stm32/viewtool-stm32f107/src/stm32_highpri*. Test driver.
