Level Triggered Interrupt handling and nested interrupts

[Updated question as GIC v2 has 3 registers ACK, EOIR, DIR]

This is the most basic question which I need someone else to clarify and state that the sequence below is correct.

In the following arch,

```
[Core] ----- [ Interrupt Controller ] --Level Triggered -- [Device]
```

- a. Device Raises the Level and informs the Interrupt Controller
- b. Interrupt controller triggers core of an interrupt. (Assuming core's interrupts enabled)
- c. Assuming the Interrupt controller is GIC (used with ARM) and it has 3 registers
    1. - Interrupt Deactivate (GICC_DIR)
    2. - Interrupt ACK (which returns the IRQ number), (GICC_IAR)
    3. - End Of Interrupt Register (GICC_EOIR)

Note: On a GICv2 implementation, setting GICC_CTLR.EOImode to 1 separates the priority drop and interrupt deactivation operations.
Ref: 3 Interrupt Handling and Prioritization (ARM IHI 0048B.b ID072613)

Now the points which need confirmation,

- d. Till the Core doesnt ACK the interrupt to get the IRQ, the interrupt remains pending and device interrupt line to Interrupt controller level high
- e. Core disables its interrupts. Core Does ACK get IRQ, no change in interrupt line from device.

Now there can be two cases here.

- A. Core masks the particular interrupt on GIC, but does nothing on the device which will clear the interrupt on device. Core enables its interrupts
- B. Core sets GICC_EOImode =1, and writes interrupt id to EOIR. Core enables its interrupts

Based on (A) or (B)

> Q1. Would the interrupt be raised again from Interrupt Controller to Core?

Now how would interrupt nesting work in this case?

linux arm interrupt

edited Jan 4 '14 at 22:24
artless noise
8,085 3 21 50

asked Oct 30 '13 at 4:22
mSO
668 2 10 27

See my **Edit2**. I think the the `GICC_DIR` is not what you think. It doesn't disable interrupts, it just *ack*-knowledges the interrupt. The GIC *IAR* or ACK is more like a traditional *PEND* and the *EOIR* is a priority drop (with EOImode=1). The *DIR* is a traditional *ACK*; Ie, the CPU acknowledge the interrupt is finished. With EOImode=0, a write to *EOIR* discards the priority of the active interrupt (so a lower priority my punch through) and acknowledges in one go. You need to write to `ICENABLERx` or some distributor register to disable it and stop the level IRQ from reoccuring. – artless noise Jan 15 '14 at 21:51

GIC_DIR is deactivate interrupt register – mSO May 8 '14 at 4:07

1 Answer

> **Q1.** Would the interrupt be raised again from Interrupt Controller to Core?

Of course, it will be re-raised. This is a property of the level triggered interrupt. There is no *state* in the interrupt controller. It would have difficulty to tell if the interrupt has been re-raised or if it persisted. Especially, the interrupt may have been serviced for a very small time and the GIC would not see the *high-low-high* transition to tell the difference between a new and an existing interrupt source.

> **Q2.** If after (e) core directly does (g), will the interrupt be raised again from Interrupt Controller to core

This seems identical to the above question. There maybe a level triggered device where *servicing* the device leaves the interrupt line high. For example, an interrupt might be **FIFO not empty**. If the **FIFO** has two entries, the first read may not clear the interrupt.

See level triggered interrupts at Wikipedia. *After servicing this device...*. You must always service the device with *level triggered* interrupts. The interrupt controller (GIC) has no idea how a peripheral works. Putting assumptions in the controller will limit its use.

> Now how would interrupt nesting work in this case.

It is not clear what is nesting. For instance with the **FIFO** example above, you can read the device for the number of entries or read and check the *interrupt status* after each read. When a *read* clears the interrupt, it is fine to re-anble the interrupt source.

Nesting of separate IRQ sources is standard. At step **f**, the IRQ service routine must service the device until the level is not driven. The *irqActive* bits at 0x300-0x304 can be read to determine if the IRQ service is finished. Then the level triggered ISR returns. If it is pre-empted at any point, the controller will detect an new level source, or the ISR will continue servicing the peripheral.

- Device raises line informs GIC.
- GIC signals ARM core and jumps to vector.
- Vector reads GIC interrupt ACK and jumps to ISR.
- Level routine disables the *level IRQ* and re-enables interrupts.
- Level routine services device until *irqActive* low. (may pre-empt to other ISRs here).
- Mask interrupts, re-enable *level* source, and return to caller.

If in the last step (or just before) an additional service item happens, there will be a *back-to-back* level interrupt. This will be in-frequent as multiple interrupt sources will have to happen during the same time period. This is typical of *interrupt nesting*. The overall system will be busier, but latency will be better.

Section *3.2.1 Priority drop and interrupt deactivation* has the following steps to disable the level interrupt,

1. read IAR - initial read of active interrupt.
2. write EOIR - drop it from the *priority*; allow nesting of lower priority.
3. write DIR - say it has *ended* (or serviced).

When the actual device has been determined to have been serviced, the interrupt is re-enabled. If you wish to allow only higher priority interrupts, then the write to `EOIR` would be delayed until the end of the ISR; the higher priority interrupts will naturally pre-empt the level interrupt.

Edit:

> Now there can be two cases here.
>
> A. Core masks the particular interrupt on GIC, but does nothing on the device which will clear the interrupt on device. Core enables its interrupts

If the interrupt is masked, it will not re-assert.

> B. Core sets GICC_EOImode =1, and writes interrupt id to EOIR. Core enables its interrupts

Writing the *EOIR* will transition from *active+pending* to just *active* and the interrupt will refire (if all you are doing is 'B').

In interrupt nesting, Linux naturally does the first part of the picture. When there are two active ISR (right side), this is an optional configuration; during 'IRQ-k', interrupts must be re-enabled. It takes more stack to do this and you will have to modify the stock Linux afaik.

Edit2: The `GICC_CTRL.EOImode =1` is confusing. This separates the *interrupt serviced* from the *priority drop* portion. If you have an interrupt with a critical portion and a non-critical, you can separate the phases. Write to *EOIR* after the critical portion to drop the priority. Then the *DIR* register says that the interrupt service is finished. I would always leave `GICC_CTRL.EOImode=0` as I don't think this is needed. The manual documentation is written from the perspective of the interrupt controller and not a CPU using it (and hence a programmer's mental model); deactivated means the current *IRQ* line and not the interrupt in general.

edited Jan 15 '14 at 21:44

answered Jan 3 '14 at 18:35
artless noise
8,085 3 21 50

thanks for the answer artless noise, I have updated the question for GICv2 – mSO Jan 4 '14 at 4:34

will wait for your response. if you like we can keep our mail ids and exchange knowledge – mSO Jan 5 '14 at 4:55