WHITE PAPER

Critical Interrupt Prioritization

Using interrupt routing and prioritization to improve responsiveness for critical interrupts

Michael Williams September 2014

Problem statement

This paper addresses the case of using interrupt routing and prioritization to improve the responsiveness for critical interrupts.

Figure 1 shows the multiple Exception levels, EL0 to EL3, and two Security states implemented by an ARMv8-A [1] processor[1]. Software executing at higher Exception levels has more privilege than software executing at lower Exception levels. EL0 is the least privileged Exception level. EL3 is the most privileged Exception level.

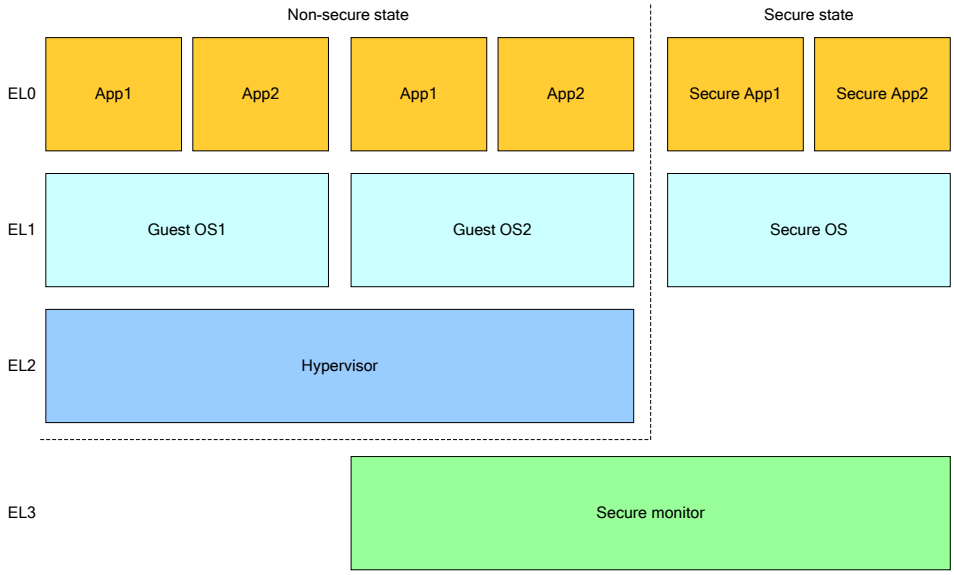

Figure 1: ARMv8-A Exception levels and Security states

The ARMv8-A processor only changes Exception level on taking or returning from an exception:

- An exception can never be taken to a lower Exception level.
- An exception return can never be to a higher Exception level.

[1] Implementations can include fewer Exception levels. Four is the maximum. Only a single Security state is implemented if EL3 is not implemented. The ARMv7-A [6] exception model is similar. In ARMv7, and in ARMv8-A when EL3 is using AArch32, the Secure OS occupies EL3 alongside the Secure monitor, and there is no Secure EL1. Although this paper focuses on ARMv8-A systems, the principles can be applied to ARMv7-A.

ARMv8-A is a RISC architecture that:

- Uses banked syndrome and exception link registers to report exceptions to software.
- Does not manage tasks or virtual machines in hardware using microcode.
- Does not allow the uncontrolled nesting of exception handlers at the same level of privilege when executing in AArch64 state[2].

To control when interrupts can be taken, ARMv8-A provides *process state (PSTATE) interrupt masks*. The PSTATE interrupt masks prevent interrupts from being taken to the current Exception level. The masks are set:

- By hardware on taking any exception to protect the syndrome registers. Software must clear the masks after saving the syndrome register values to memory.
- By software prior to an exception return to protect the syndrome registers.
- By software to prevent reentering *critical regions* of software that might be shared with interrupt handlers, such as software that sets operating system locks.

As a result, the processor does not respond to interrupts targeting a given Exception level when:

- The interrupt is masked because software is executing in a critical region.
- Software is executing at a higher Exception level.

This unresponsiveness is a factor in determining the total interrupt latency at the processor interface.

Note: The total interrupt latency also includes the time taken from the interrupt being asserted by hardware, through the interrupt controller recognizing, prioritizing, and delivering the interrupt to the processor, to the processor terminating the current thread of execution and taking the interrupt. On complex systems with many interrupts and out-of-order processors with many instructions in flight, this is finite but might be substantial.

Figure 2 shows how interrupts are masked.

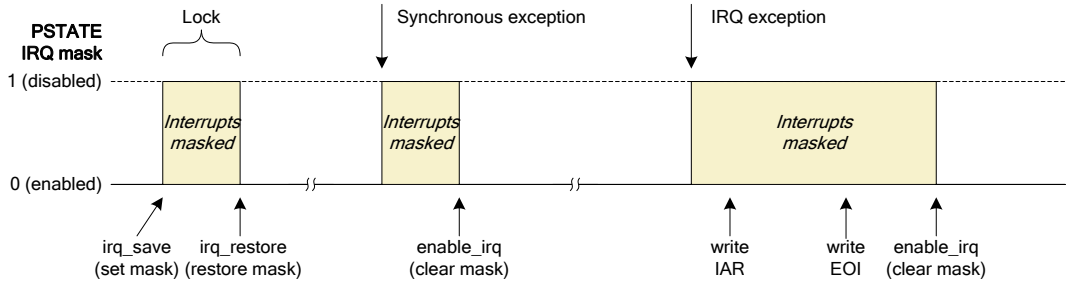

Figure 2: Interrupt masking

How this unresponsiveness manifests itself

There are cases when software requires an interrupt that can be taken even when software is executing in a critical region. For example:

Code profiling:

A time-based or event-based interrupt handler collects samples to statistically measure performance. Maskable interrupts create *blind spots* in the profile: regions of code that cannot be profiled in this way. Furthermore, any sample in a blind spot will be wrongly attributed to the code that unmasks the interrupt, creating *false hot spots*.

[2] When an exception is taken to an Exception level using AArch32, limited nesting of exceptions is permitted, but otherwise ARMv7-A and ARMv8-A using AArch32 have the same constraints.

Kernel debug:

An interrupt breaks into code to execute a debugger. Maskable interrupts create *blind spots* in the code: regions of code that cannot be debugged by the kernel debugger.

Watchdog:

A timer interrupt breaks into code to reset a watchdog timer. Maskable interrupts might delay the interrupt being taken. Usually this is not an issue, because the purpose of the watchdog is to detect faults when both the hardware and the software are responsive. If interrupts are stuck in the masked state, this is a faulty state.

Error interrupts:

Error interrupts can be time critical, and delaying the interrupt causes errors to unnecessarily propagate.

Note: This is only a valid concern if there is some guaranteed low-latency delivery of the interrupt. This is not usually true for interrupts, but might be the case for certain classes of error interrupt.

In these cases, the main issue is masking the interrupt at the current Exception level. The interrupt is not required to be truly non-maskable, but it must not be masked *often*. These interrupts are typically handled outside of the main operating system kernel interrupt handling code. This means that, for example, the interrupt handlers would not access critical operating system locks. Therefore these interrupts can be permitted to interrupt critical regions in the kernel.

First and second class interrupts

It is desirable to add a second source of interrupts for those tasks that must be handled even when the processor is executing in critical software regions. There are a number of ways to achieve this in the ARM architecture.

Some of these approaches are discussed in the following sections:

- *Using a second physical interrupt*.
- *Routing interrupts to a higher Exception level*.
- *Using interrupt priorities*.

The section *Failure modes* describes mechanisms to detect and recover from the rare case when the system remains unresponsive even when these approaches are employed.

Using a second physical interrupt

The ARM architecture defines three physical interrupt exceptions:

- SError interrupt.
  - SError is an ARMv8 concept. ARMv7 has the similar concept of *asynchronous external aborts*.
- FIQ interrupt.
- IRQ interrupt.

ARMv7-A, and ARMv8-A using AArch32, allows nesting of physical interrupt handlers at an Exception level. That is, FIQ and IRQ interrupts are handled in separate modes with banked registers and the processor does not set the PSTATE FIQ interrupt mask on most exception entries.

However, AArch64 supports only a single exception handling mode at each Exception level. There is no register banking within an Exception level, and all PSTATE interrupt masks are set on taking an exception, meaning interrupts targeting the current Exception level are masked.

In addition, an interrupt controller, such as GIC-400 or GIC-500, can multiplex many different interrupt sources to the FIQ and IRQ interrupts. Each interrupt source is assigned to a group:

- GIC-400, which implements GICv2 [2], has two interrupt groups:
  - Group 0 interrupts, which are always Secure.

  - Group 1 interrupts, which are always Non-secure.
- GIC-500, which implements GICv3 [3], has three interrupt groups:
  - Group 0 interrupts, which are always secure.
  - Secure Group 1 interrupts.
  - Non-secure Group 1 interrupts.

Each group is mapped to either FIQ or IRQ interrupts by the GIC. SError interrupts are not handled by the GIC.

In an ARMv8-A system using GICv3:

- Group 1 interrupts for the current Security state are mapped to the IRQ interrupt. The IRQ interrupt is:
  - Taken to the Secure OS at EL1, if the current Security state is Secure state and the processor is not executing at EL3.
  - Taken to either the Non-secure OS at EL1 or the hypervisor at EL2 if the current Security state is Non-secure. The hypervisor decides which of these Exception levels physical IRQ interrupts are taken to. If the interrupts are taken to EL2, the hypervisor schedules *virtual* interrupts for its guest operating systems. In this case, the physical interrupts are never masked inside the guest OS, but the virtual interrupts can be.
- Secure Group 0 interrupts, and Group 1 interrupts for the other Security state, are mapped to the FIQ interrupt. The FIQ interrupt is taken to the Secure monitor at EL3. If the interrupt is a Group 1 interrupts for the other Security state then the Secure monitor must switch context to the other Security state.

There is effectively only a single physical interrupt source for each layer of software:

- Group 0 interrupts for the Secure monitor.
- Secure Group 1 interrupts for the Secure OS.
- Non-secure Group 1 interrupts for the Non-secure hypervisor and all its guest operating systems.

Therefore a second physical interrupt source is not available.

Routing interrupts to a higher Exception level

One mechanism to allow critical regions to be interrupted is to route the interrupt to a higher Exception level. This is recommended for handling system critical interrupts such as system watchdogs and for error handling.

However, for less critical interrupts, such as debugging, profiling and OS watchdogs, it is preferable to keep the interrupt handling at the same Exception level, because:

- The software at different Exception levels is usually supplied by different vendors.
- The Exception levels might not share a common virtual address space.

However, an exception handler at a higher Exception level can triage the interrupt and, if necessary, delegate the interrupt to the lower Exception level for handling. This is an asynchronous entry to the lower Exception level and software might have masked interrupt at the lower Exception for good reason. Software can emulate an exception-like entry using a *software delegated exception* model.

Software delegated exception model

A *software delegated exception* (SDE) is a software contract between two Exception levels (the *delegator* and the *surrogate*) to delegate certain types of exception from the delegator to the surrogate, and as such has no explicit support in the architecture.

To implement an SDE:

- The surrogate, using an `HVC` or `SMC` instruction, must request that the delegator delegates exceptions. The surrogate can also revoke this request using a second `HVC` or `SMC` call. See [4] for the recommended `SMC` calling convention.
- The surrogate and the delegator must agree a surrogate exception entry vector address. This is a virtual address in the translation regime of the surrogate. Preferably, this is an offset from the *vector base address register* (VBAR) for the surrogate:
  - Delegated IRQ and FIQ interrupts are taken to the FIQ interrupt vector offset.

  - Delegated SError interrupt are taken through the SError interrupt vector offset.

  The SDE might be used for synchronous external aborts, which are also used by hardware to signal errors. These are also delegated through the SError interrupt vector offset.
- The delegator configures the processor to route the delegated exception to itself. When the exception is taken, the processor enters the delegator.
- The delegator triages the exception and decides whether to delegate it to the surrogate. To delegate the exception, the delegator:
  - Writes any necessary syndrome information for the exception in the syndrome registers of the surrogate, that is, ELR_EL*x*, SPSR_EL*x*, and, if applicable, ESR_EL*x*.
  - Executes an exception return instruction that changes to the surrogate Exception level and branches to the appropriate exception entry vector.
- To signal the end of the exception handler to the delegator, the surrogate executes an `SMC` or `HVC` instruction.
- The delegator is then responsible for returning to the point from which the exception was taken, if applicable.

In addition, software must consider:

- What to do if a new exception is taken to the delegator while the delegator and surrogate are already processing a delegated exception. For example, this might be considered a fatal double fault event.
- Whether a hypervisor supports multiple guest OS contexts, where each OS implements delegated exception handling. For example, interrupts taken to a Secure monitor might be first delegated to the hypervisor which in turn delegates them to the correct guest OS.

For a hypervisor, this provides two mechanisms for delegating interrupts:

- The HCR_EL2.{VSEI, VF, VI} mechanisms to delegate *maskable* interrupts. The guest OS can mask these interrupts using the PSTATE interrupt masks.
- The SDE model to delegate *unmaskable* interrupts.

Note that interrupts routed to a higher Exception level are not masked by the PSTATE interrupt masks whilst executing in the surrogate. This is an advantage over "non-maskable" interrupt schemes.

Using interrupt priorities

In addition to assigning interrupts to a group, a GIC can assign a priority value to each interrupt source. Although a maximum of 256 priority levels are supported, implementations might have fewer levels. The minimum number of levels available for Non-secure interrupts is 16.

In the GIC prioritization scheme, smaller numbers have higher priority. This means that the smaller the assigned priority value, the higher the priority of the interrupt. Priority value 0 always indicates the highest possible interrupt priority, and the lowest priority depends on the number of implemented priority levels.

The priorities work as follows:

- On activating an interrupt, the *running priority* of the CPU interface is set to the group priority of the interrupt. The ICC_RPR_EL1 register is used to discover the running priority.
- Software can set a *GIC priority mask* to mask higher priority interrupts. The ICC_PMR_EL1 register is used to set the GIC priority mask.
- The GIC will signal a pending interrupt only if both:
  - Its priority is higher than the GIC priority mask for that CPU interface.
  - Its group priority is higher than that of the running priority on the CPU interface.

The priority numbering is shared by interrupts across all groups. Only software executing in Secure state can assign a physical priority value that is less than 128 to an interrupt[3].

[3] Non-secure software can *write* any value from 0 to 255, but hardware compresses this to the range 128 to 255. For more information, see [2, 3].

Software is not required to use the priority scheme. However, it can make use of priorities to create a second source of interrupts by:

- Using a low priority level, such as 240, for all OS interrupts.
- Using a higher priority level, such as 226, for all other interrupts.

When entering critical regions, rather than setting the PSTATE interrupt mask, or leaving the mask set, software uses the GIC priority mask to mask only the OS interrupts. This is shown in Figure 3.

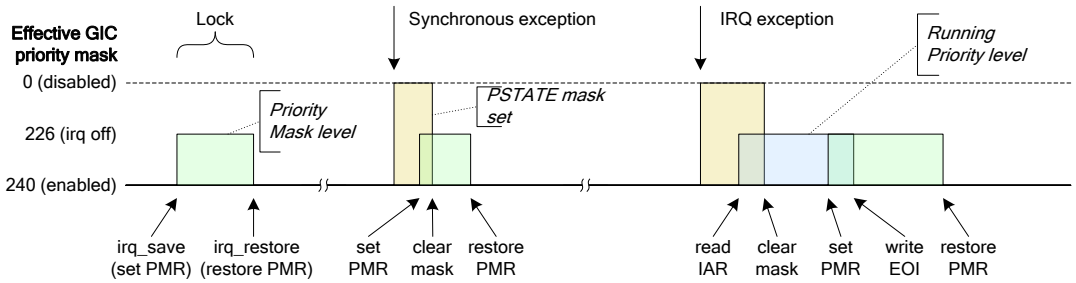

Figure 3: Using interrupt priority masks

Note that the PSTATE interrupt masks are still set on taking an exception, and must be set by software before an exception return. ARM recommends that software sets the GIC priority mask and clears the PSTATE interrupt mask as soon as possible on taking an exception, and sets the PSTATE interrupt mask and clears the GIC priority mask as late as possible before an exception return. This not only reduces the period when interrupts are masked, but also reduces the possibility of leaving interrupts permanently masked as a result of a software fault.

With GICv3, software can alter the GIC priority mask by writing to the ICC_PMR_EL1 system register, as shown in Example 1 below.

```
MRS    X0,ICC_PMR_EL1      ;; Read current priority mask
MOV    X1,#224
MSR    ICC_PMR_EL1,X1      ;; Set new priority mask
...                        ;; Critical region code
MSR    ICC_PMR_EL1,X0      ;; Restore priority mask
```

Example 1: Setting and restoring the interrupt priority mask around a critical region, GICv3

The write to ICC_PMR_EL1 is *self-synchronizing*. This is a useful as it means that no additional instruction synchronization barrier is required to ensure the priority mask is set. Instructions for changing the PSTATE interrupt mask also have this property

Setting the GIC priority mask can be conditional on the current GIC priority mask level. Whether this is a worthwhile optimization is heavily dependent on the processor microarchitecture.

Additional considerations for Secure monitors and hypervisors

Only software executing in Secure state can:

- Assign a physical priority value that is less than 128 to an interrupt[3].
- Set the GIC priority mask to a priority value that is less than 128.

This means software executing in Secure state controls which are the highest priority interrupts. For example:

- Use a medium priority level, such as 112, for all Secure monitor interrupts.
- Use a higher priority level, such as 96, for a second source of Secure monitor interrupts.

In Non-secure state, when physical interrupts are routed to a hypervisor which triages and delegates them to a guest OS using virtual interrupts, the processor uses the *virtual priority mask register* (GICV_PMR) as the GIC priority mask for virtual interrupts. GICV_PMR does not affect physical interrupts and writes to ICC_PMR_EL1 by a guest OS update GICV_PMR.

Additional considerations for idle and power-down code

A pending interrupt masked by a PSTATE interrupt mask is a wake-up event for a `WFI` instruction. This allows software to enter a low-power idle state with interrupts masked, and on assertion of an interrupt, restore some state before processing the interrupt.

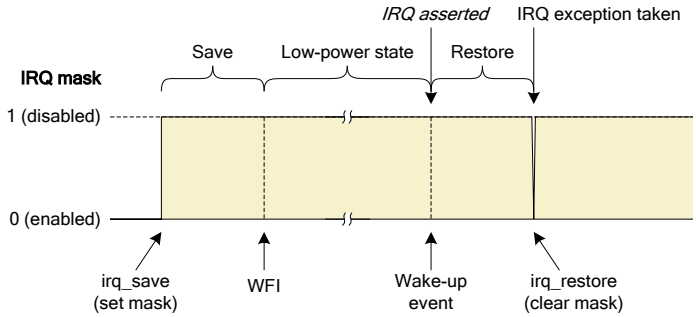

Figure 4: Interrupt masking during low-power state

However, an interrupt with a lower priority than the current GIC priority mask is not a wake-up event for a `WFI` instruction. Care must be taken not to mask any potential wake-up events by entering the low-power state with a raised priority mask.

Software must consider why it is masking interrupts when deciding whether to the GIC priority mask or the PSTATE interrupt mask, as shown in Table 1.

Use case	Reason for masking interrupts	Action
Critical region.	Avoid taking interrupts in non-reentrant code.	Use GIC priority mask.
Low power state.	Recover from low-power state before servicing interrupt.	Use PSTATE interrupt mask.

Table 1: Reasons for masking interrupts

Power states are usually controlled at the highest privilege level, either through use of a *power-state co-ordination interface* [5], or by trapping use of the `WFE` and `WFI` instructions by software at lower Exception levels. ARM recommends use of a power-state co-ordination interface.

Failure modes

If the processor does not respond to the highest priority interrupt, for example a watchdog, the system might:

- Escalate the interrupt directly to a *baseboard management controller* (BMC) or *system control processor* (SCP) with guaranteed interrupt response.
- Force a processor to halt execution by asserting **SPIDEN** and **EDBGRQ** and put the processor into a special Debug state where it can be examined by the BMC or SCP.

  Note: This requires a mechanism for the BMC or SCP to control the debug interface of the processor.

Summary

The ARMv8 hierarchy of Exception levels avoids the need to mask the most critical types of interrupt, such as system errors in all software apart from the deepest, most trusted levels

A *software delegated exception* model is proposed that allows these exceptions to be returned to less privileged software for handling.

Software can use the GIC priority mechanisms to provide a "not-often-masked" interrupt source for operations such as debugging and profiling, and for error handling at the most trusted software levels. The priority masks allow these interrupts to be taken when general OS interrupts are masked.

Compared to a non-maskable interrupt, there are still pieces of code where the interrupts are masked:

- On taking an exception
- Before return from an exception.

Architectures that support *non-maskable interrupts* might use microcode to manage the interrupt, including saving live register state to a stack in memory. ARMv8—a RISC architecture—requires that software performs this task. This software can be as validated and certified to a similar degree as microcode, giving similar outcomes for reliability.

Bibliography

[1] ARM Limited, ARM® Architecture Reference Manual; ARMv8, for ARMv8-A architecture profile, Issue A.c ed., 2013, 2014.

[2] ARM Limited, ARM® Generic Interrupt Controller; Architecture version 2.0, Issue B ed., 2008, 2011.

[3] ARM Limited, GIC Architecture Specification (version 3), 2014.

[4] ARM Limited, SMC Calling Convention, issue A ed., 2013.

[5] ARM Limited, Power State Coordination Interface (PSCI), 2013.

[6] ARM Limited, ARM® Architecture Reference Manual; ARMv7-A and ARMv7-R edition, Issue C.c ed., 1996-1998, 2000, 2004-2012, 2014.

Appendix: Example GICv3 sequences for interrupt prioritization

Entry to a critical region ("irq_save")

```
MRS    X0,ICC_PMR_EL1      ;; Read current priority mask
MOV    X1,#224
MSR    ICC_PMR_EL1,X1      ;; Set new priority mask
```

Exit from a critical region ("irq_restore")

```
MSR    ICC_PMR_EL1,X0      ;; Restore priority mask
```

Synchronous exception entry

```
SUB    SP,SP,#48           ;; Make a frame on the stack
STP    X0,X1,[SP,#32]
MRS    X0,ELR_EL1          ;; Exception Link Register
MRS    X1,SPSR_EL1         ;; Saved Program Status Register
STP    X0,X1,[SP,#16]
MRS    X0,ESR_EL1          ;; Exception Syndrome Register
MRS    X1,FAR_EL1          ;; Fault Address Register
STP    X0,X1,[SP,#0]
MOV    X0,#224
MSR    ICC_PMR_EL1,X0      ;; Priority Mask Register
MSR    DAIFClr,#0xF        ;; Unmask interrupts
```

Asynchronous exception entry

```
SUB    SP,SP,#32           ;; Make a frame on the stack
STP    X0,X1,[SP,#16]
MRS    X0,ELR_EL1          ;; Exception Link Register
MRS    X1,SPSR_EL1         ;; Saved Program Status Register
STP    X0,X1,[SP,#0]
MRS    X0,ICC_IAR1_EL1     ;; Interrupt Acknowledge Register
ISB
MRS    X1,ICC_RPR_EL1      ;; Running Priority Register
CMP    X1,#240
BLT    HighPriorityIRQVector
MSR    DAIFClr,#0xF        ;; Unmask interrupts
```