GICv3 and GICv4 Software Overview

Non-Confidential

GICv3 and GICv4 Software Overview

Release Information

The following changes have been made to this document.

Change history

Date	Issue	Confidentiality	Change
July 2015	A	Non-Confidential	First release
Feburary 2016	B	Non-Confidential	Added coverage of virtualization

Proprietary notice

ARM Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

LES-PRE-20327

In this document, where the term ARM is used to refer to the company it means "ARM or any of its subsidiaries as appropriate".

Confidentiality status

This document is Non-Confidential. The right to use, copy and disclose this document may be subject to license restrictions in accordance with the terms of the agreement entered into by ARM and the party that ARM delivered this document to.

Product Status

The information in this document is final, that is for a developed product.

Feedback on content

If you have any comments on content, then send an e-mail to errata@arm.com. Give:

- The title.
- The number.
- The page numbers to which your comments apply.
- A concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

Web Address

`http://www.arm.com`

Table of Contents

1. Preface

This document provides an overview of version 3 of the Generic Interrupt Controller Architecture (GICv3). It is primarily intended for software engineers writing bare metal code for ARMv8-A based platforms. A familiarity with ARMv8-A and writing bare metal code is assumed.

1.1 Document status

This is release B of the document.

1.2 References

This document refers to the following documents:

- *ARM® Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and 4.0* (ARM IHI 0069A)
- *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* (ARM DDI 0487A)
- *ARM® CoreLink™ GIC-500 Generic Interrupt Controller Technical Reference Manual* (ARM DDI 0516A)
- *ARM® Cortex®-A57 MPCore™ Processor Technical Reference Manual (*ARM DDI 0488D)

1.3 Terms and Abbreviations

Table 1 Terms and Abbreviations shows the terms and abbreviations that are used in this document.

Term	Description
ARE	Affinity Routing Enable
BPR	Binary Point Register
EL	Exception level (ARMv8-A)
EOIR	End of Interrupt Register
GIC	Generic Interrupt Controller
GICv3	Version 3 of the Generic Interrupt Controller Architecture
GICv4	Version 4 of the Generic Interrupt Controller Architecture
IAR	Interrupt Acknowledge Register
ITS	Interrupt Translation Service
ITT	Interrupt Translation Table
LPI	Locality-specific Peripheral Interrupt
PE	Processing element. The abstract machine defined in the ARM architecture, as documented in an ARM Architecture Reference Manual. See also ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile.
PPI	Private Peripheral Interrupt
RAO/WI	Read-As-One, Writes Ignored
RAZ/WI	Read-As-Zero, Writes Ignored

SGI	Software Generated Interrupt
SPI	Shared Peripheral Interrupt
SRE	System Register Enable
VM	Virtual Machine
vPE	Virtual PE
VPT	Virtual LPI Pending Table

Table 1 Terms and Abbreviations

The *ARM® Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and 4.0* and *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* use the term *Processing Element* or *PE* as a generic term for a machine that implements the ARM architecture. As an example, the ARM® Cortex®-A57 MPCore™ is a multi-core *processor*, with up to four *cores*. For the ARM® Cortex®-A57 MPCore™ , each *core* is what the architecture specifcations refer to as a *PE*.

2. Introduction

This document provides a software focused overview of the features of GICv3, and describes the operation of a GICv3 compliant interrupt controller. It is also a primer on how to configure a GICv3 interrupt controller for use in a bare metal environment.

This document compliments the *ARM® Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and 4.0*. It is not a replacement or alternative. Refer to the *ARM® Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and 4.0* for detailed descriptions of registers and behaviors.

2.1 Scope

GICv3 allows for a number of different configurations and use cases. For simplicity, this document concentrates on a sub-set. It only describes the case where:

- Two Security states are present.
- Affinity routing is enabled for both Security states.
- System register access is enabled at all Exception levels.
- The connected processor, or processors, are ARMv8-A compliant, implement all Exception levels and use AArch64 at all Exception levels.

This document does not cover:

- Legacy operation, other than in the introduction.
- Use from an Exception level that is using AArch32.

2.2 Brief history of the GIC architecture

GICv3 adds several new features. To put these new features in context Table 2 provides a brief overview of the different versions of the GIC architecture, and their key features.

Table 2 GIC version history

Version	Key features	Typically used with
GICv1	Support for up to eight PEs. Support for up to 1020 interrupt IDs. Support for two Security states.	ARM Cortex-A5 MPCore ARM Cortex-A9 MPCore ARM Cortex-R7 MPCore
GICv2	All key features of GICv1 Support for virtualization.	ARM Cortex-A7 MPCore ARM Cortex-A15 MPCore ARM Cortex-A53 MPCore ARM Cortex-A57 MPCore
GICv3	All key features of GICv2 Support for more than eight PEs. Support for message-based interrupts. Support for more than 1020 interrupt IDs. System register access to the CPU Interface registers. An enhanced security model, separating Secure and Non-secure Group 1 interrupts.	ARM Cortex-A53 MPCore ARM Cortex-A57 MPCore ARM Cortex-A72 MPCore
GICv4	All key features of GICv3 and: Direct injection of virtual interrupts	ARM Cortex-A53 MPCore ARM Cortex-A57 MPCore ARM Cortex-A72 MPCore

NOTE: GICv2m is an extension to GICv2 to add support for message based interrupts. For more information contact ARM.

2.3 Implementations of the GICv3 architecture

The ARM® CoreLink™ GIC-500 is an implementation of GICv3. The ARM® Cortex®-A53, ARM® Cortex® -A57 and ARM® Cortex®-A72 MPCore processors implement the required CPU interface.

2.4 Legacy support

GICv3 makes a number of changes to the programmers' model. To support legacy software written for GICv2 systems, GICv3 supports legacy operation.

The programmers' model that is used is controlled by the Affinity Routing Enable (ARE) bits in GICD_CTRL:

- When ARE == 0, affinity routing is disabled (legacy operation).
- When ARE == 1, affinity routing is enabled.

NOTE: For readability, GICD_CTRL.ARE_S and GICD_CTRL.ARE_NS are referred to collectively as ARE in this document where appropriate.

In a system with two Security states, affinity routing can be controlled separately for each Security state. Only specific combinations are permitted, and these are shown in Figure 1.

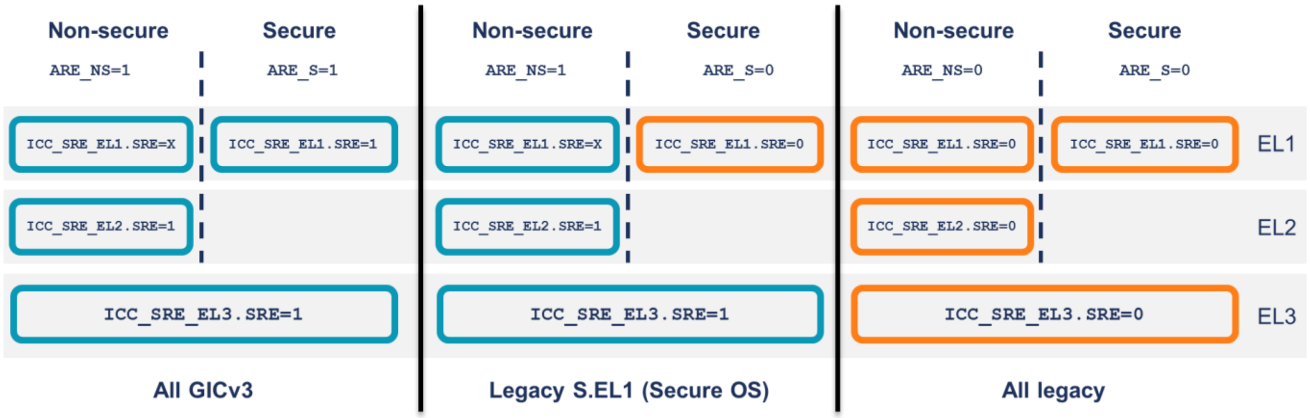


Figure 1 Supported ARE combinations

This documents focusses on the new GICv3 programmers' model, where ARE=1 for both security states. Legacy operation, where ARE==0, is not described.

NOTE: Support for legacy operation is OPTIONAL. When support for legacy operation is implemented, legacy operation is selected out of reset.

3. GICv3 fundamentals

This chapter describes the basic operation of an interrupt controller that is compliant with the GICv3 architecture. It also describes the different programming interfaces.

3.1 Interrupts types

GICv3 defines the following types of interrupt:

SPI (Shared Peripheral Interrupt)

This is a global peripheral interrupt that can be routed to a specified PE, or to one of a group of PEs.

PPI (Private Peripheral Interrupt)

This is peripheral interrupt that targets a single, specific PE.

An example of a PPI is an interrupt from the Generic Timer of a PE.

SGI (Software Generated Interrupt)

SGIs are typically used for inter-processor communication, and are generated by a write to an SGI register in the GIC.

LPI (Locality-specific Peripheral Interrupt)

LPIs are new in GICv3, and they are different to the other types of interrupt in a number of ways. In particular, LPIs are always message-based interrupts, and their configuration is held in tables in memory rather than registers. This is described in more detail in Chapter 6.

NOTE: LPIs are only supported when `GICD_CTLR.ARE_NS==1`.

3.1.2 Interrupt Identifiers

Each interrupt source is identified by an ID number, referred to as an INTID. The available INTIDs are grouped into ranges, and each range is assigned to a particular type of interrupt.

Table 3 Interrupt ID ranges

INTID	Interrupt Type	Notes
0 - 15	SGIs	Banked per PE
16 - 31	PPIs	Banked per PE
32 - 1019	SPIs	-
1020 - 1023	Special interrupt number	Used to signal special cases, see section 5.3
1024 - 8191	Reserved	-
8192 and greater	LPIs	The upper boundary is IMPLEMENTATION DEFINED

3.1.3 How interrupts are signaled to the interrupt controller

Traditionally, interrupts are signaled from a peripheral to the interrupt controller using a dedicated hardware signal.

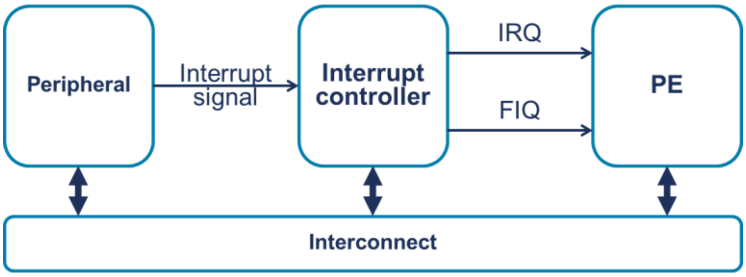


Figure 2 Dedicated interrupt signal

GICv3 supports this model, and additionally supports message-based interrupts. A message-based interrupt is an interrupt that is set and cleared by a write to a register in the interrupt controller.

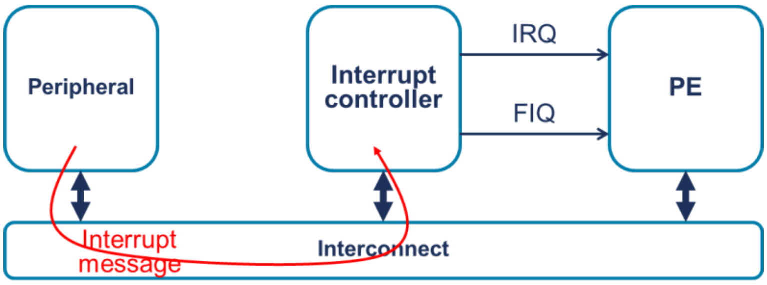


Figure 3 Message-based interrupt transported over the interconnect

Using a message to forward the interrupt from a peripheral to the interrupt controller removes the requirement for a dedicated signal per interrupt source. This can be an advantage for hardware designers of large systems, where potentially hundreds or even thousands of signals might be routed across a SoC and converge on the interrupt controller.

In GICv3, SPIs *can* be message-based interrupts, but LPIs *are always* message-based interrupts. Different registers are used for the different interrupt types, as shown in Table 4.

Table 4 Message-based interrupt registers

Interrupt Type	Registers
SPI	`GICD_SETSPI_NSR` asserts an interrupt `GICD_CLRSPI_NSR` deasserts an interrupt
LPI	`GITS_TRANSLATER`

Impact of message-based interrupts on software

Whether an interrupt is sent as a message or using a dedicated signal has little effect on the way the interrupt handling code handles the interrupt.

Some configuration of the peripherals might be required. For example, it might be necessary to specify the address of the interrupt controller. This is outside of the scope of this document and is not described.

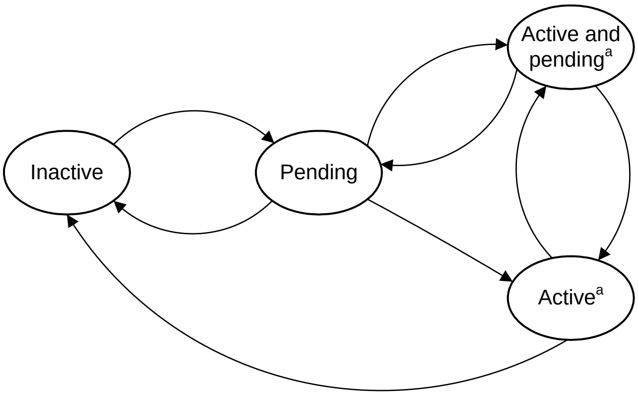
3.2 Interrupt state machine

The interrupt controller maintains a state machine for each SPI, PPI and SGI interrupt source. This state machine consists of four states:

- **Inactive**
  The interrupt source is not currently asserted.
- **Pending**
  The interrupt source has been asserted, but the interrupt has not yet been acknowledged by a PE.
- **Active**
  The interrupt source has been asserted, and the interrupt has been acknowledged by a PE.
- **Active and Pending**
  An instance of the interrupt has been acknowledged, and another instance is now pending.

NOTE: LPIs do not have an active or active and pending state. For more information, see section 6.2.

Figure 4 shows the structure of the state machine, and the possible transitions.

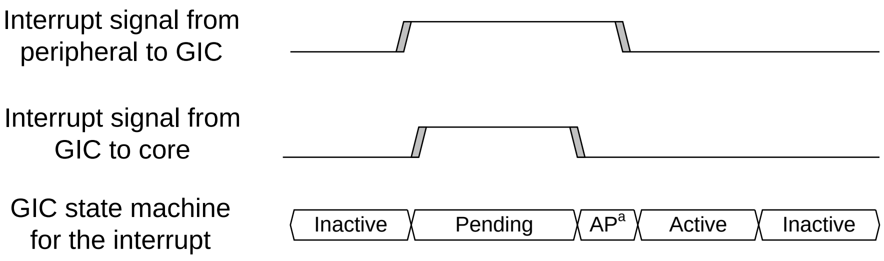


a. Not applicable for LPIs.

Figure 4 Interrupt state machine for PPIs, SGIs and SPIs

The life cycle of an interrupt depends on whether it is configured to be level-sensitive or edge-triggered. Sections 3.2.1 and 3.2.2 provide example sequences.

3.2.1 Level sensitive


Interrupt signal from peripheral to GIC
Interrupt signal from GIC to core
GIC state machine for the interrupt
Inactive
Pending
AP[a]
Active
Inactive

[a]Active and Pending

Figure 5 Interrupt life cycle - level sensitive interrupts

Inactive to Pending

An interrupt transitions from *inactive* to *pending* when the interrupt source is asserted.

At this point the GIC asserts the interrupt signal to the PE (if the interrupt is enabled and is of sufficient priority).

Pending to Active & Pending

The interrupt transitions from *pending* to *active and pending* when a PE acknowledges the interrupt by reading one of the IARs (Interrupt Acknowledge Registers) in the CPU interface. This read is typically part of an interrupt handling routine that executes after an interrupt exception is taken. However, software can also poll the IARs.

At this point the GIC deasserts the interrupt signal to the PE.

Active and Pending to Active

The interrupt transitions from *active and pending* to *active* when the peripheral de-asserts the interrupt signal. This typically happens in response to the interrupt handling software that is executing on the PE writing to a status register in the peripheral.

Active to Inactive

The interrupt goes from *active* to *inactive* when the PE writes to one of the EOIRs (End of Interrupt Registers) in the CPU interface. This indicates that the PE has finished handling the interrupt.

3.2.2 Edge-triggered

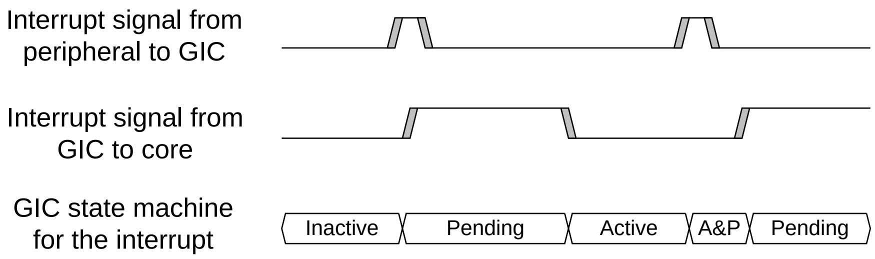

Interrupt signal from peripheral to GIC
Interrupt signal from GIC to core
GIC state machine for the interrupt
Inactive
Pending
Active
A&P
Pending

Figure 6 Interrupt life cycle - edge-triggered interrupts

Inactive to Pending

An interrupt transitions from *inactive* to *pending* when the interrupt source is asserted.

At this point the GIC asserts the interrupt signal to the PE (if the interrupt is enabled and is of sufficient priority).

Pending to Active

The interrupt transitions from *pending* to *active* when a PE acknowledges the interrupt by reading one of the IARs in the CPU interface. This read is typically part of an interrupt handling routine that executes after an interrupt exception is taken. However, software can also poll the IARs.

At this point the GIC de-asserts the interrupt signal to the PE.

Active to Active and Pending

The interrupt goes from *active* to *active and pending* if the peripheral re-asserts the interrupt signal.

Active and Pending to Pending

The interrupt goes from *active and pending* to *pending* when the PE writes to one of the EOIRs in the CPU interface. This indicates that the PE has finished handling the first instance of the interrupt.

At this point the GIC re-asserts the interrupt signal to the PE.

3.3 Affinity routing

GICv3 uses affinity routing to identify connected PEs and to route interrupts to a specific PE or group of PEs. The affinity of a PE is represented as four 8-bit fields:

GICv3使用affinity routing来标识每个连接的PE，并且使用affinity routing将一个中断路由到target PEs。

<affinity level 3>.<affinity level 2>.<affinity level 1>.<affinity level 0>

Figure 7 shows an example of an affinity level hierarchy.

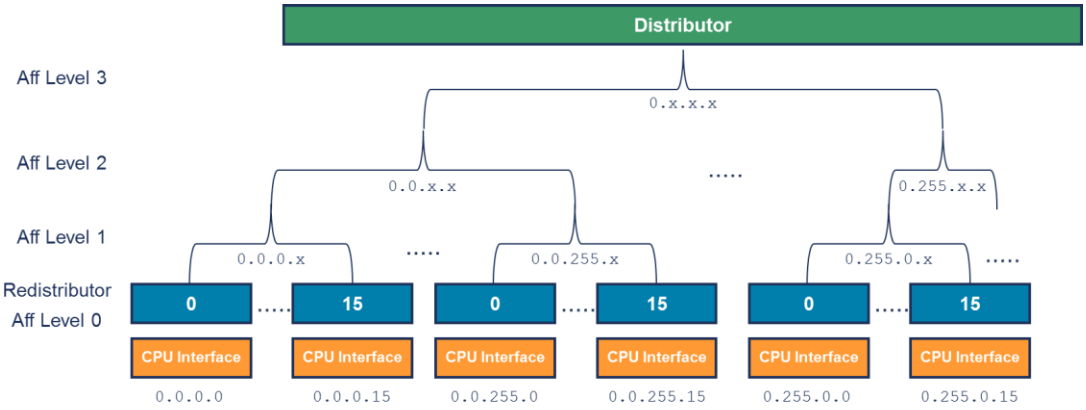


Figure 7 Example of an affinity hierarchy

At affinity level 0 there is a Redistributor. Each Redistributor connects to a single CPU interface. The Redistributors controls SGIs, PPIs and LPIs, see chapter 4.

The affinity scheme matches that used in ARMv8-A, with the affinity of a PE reported in `MPIDR_EL1`. System designers must ensure that the affinity value indicated by `MPIDR_EL1` is identical to that indicated by `GICR_TYPER` for the Redistributor connected to the PE.

The exact meaning of the different levels of affinity is defined by the specific processor and SoC. The following are examples:

<group of groups>. <group of processors>.<processor>.<core>

<group of processors>.<processor>.<core>.<thread>

It is highly unlikely that all the possible nodes exist in a single implementation. For example, a SoC for a mobile device could have a layout similar to this:

0.0.0.[0:3]	Cores 0 to 3 of a Cortex-A53 processor
0.0.1.[0:1]	Cores 0 to 1 of a Cortex-A57 processor

In ARMv8-A, AArch64 state supports four levels of affinity. AArch32 state, and ARMv7, can only support three levels of affinity. This means a design that uses AArch32 state is limited to a single node at affinity level 3 (0.x.y.z). `GICD_TYPER.A3V` indicates whether the interrupt controller can support multiple level 3 nodes.

NOTE: Although each level 1 node can host up to 256 Redistributors at level 0, in practice it is likely to be 16 or fewer. This is because of the way the target PEs for an SGI are encoded, as described in Chapter 7.

3.4 Security model

The GICv3 architecture supports the ARM TrustZone technology. Each INTID must be assigned a group and security setting. GICv3 supports three combinations, as shown in Table 5.

Table 5 Security and groupings

Interrupt Type	Example use
Secure Group 0	Interrupts for EL3 (Secure Firmware)
Secure Group 1	Interrupts for Secure EL1 (Trusted OS)
Non-secure Group 1	Interrupts for the Non-secure state (OS and/or Hypervisor)

Group 0 interrupts are always signaled as FIQs. Group 1 interrupts are signaled as either IRQs or FIQs depending on the current Security state and Exception level of the PE.

Table 6 Mapping between security settings and exception type when EL3 is using AArch64

EL and Security state of PE	Group 0	Group 1	
		Secure	Non-secure
Secure EL0/1	FIQ	IRQ	FIQ
Non-secure EL0/1/2	FIQ	FIQ	IRQ
EL3	FIQ	FIQ	FIQ

These rules are designed to complement the ARMv8-A Security state and Exception level routing controls. Figure 8 shows a simplified software stack, and what happens when different types of interrupt are signaled while executing at EL0:

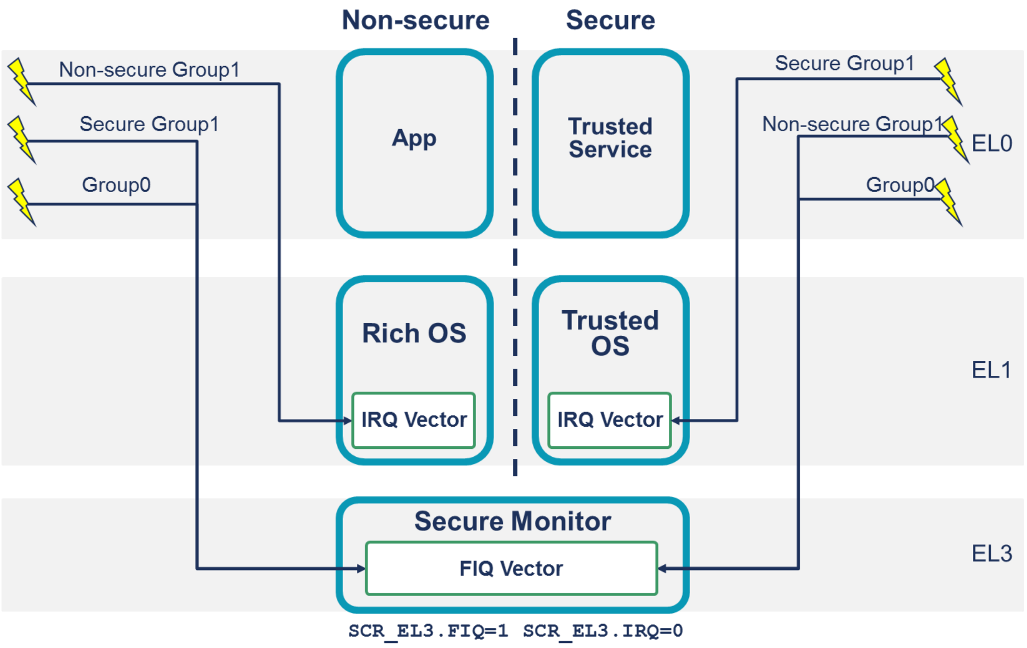


Figure 8 Interrupt routing example

In this example, IRQs are routed to EL1 (`SCR_EL3.IRQ==0`) and FIQs routed to EL3 (`SCR_EL3.FIQ==1`). Given the rules described in Table 6, while executing at EL1 or EL0 a Group 1 interrupt for the *current* Security state is taken as an IRQ.

An interrupt for the *other* Security state triggers an FIQ, and the exception is taken to EL3. This then allows software executing at EL3 to perform the necessary context switch. A more detailed example of this can be found in chapter 5.3.

3.4.1 Impact on software

Software controls the allocation of INTIDs to interrupt groups when configuring the interrupt controller. Only software executing in Secure state can allocate INTIDs to interrupt groups.

Typically only software executing in Secure state must be able to access the settings and state of Secure interrupts (Group 0 and Secure Group 1).

Accesses from Non-secure state to Secure interrupt settings and state can be enabled. This is controlled individually for each INTID, using the `GICD_NSACRn` and `GICR_NSACR` registers.

NOTE: The interrupt group to which an INTID belongs at reset is IMPLEMENTATION DEFINED.

NOTE: LPIs are always treated as Non-secure Group 1 interrupts.

3.4.2 Support for single Security state

Support for two Security states is OPTIONAL in ARMv8-A and GICv3. An implementation can choose to implement only a single Security state or two Security states.

In a GICv3 implementation that supports two Security states, one Security state can be disabled. This is controlled by `GICD_CTLR.DS`.

- `GICD_CTLR.DS == 0`
  Two Security states (Secure and Non-secure) are supported.

- `GICD_CTLR.DS == 1`
  Only a single Security state is supported. On implemenations that only implement a single Security state, this bit is RAO/WI.

When only a single Security state is supported, there are two interrupt groups. These are Group 0 and Group 1.

This document describes the case where two Security states are implemented.

NOTE: If software sets `GICD_CTLR.DS` to 1, it can only be cleared by a reset.

3.5 Programmers’ model

The register interface of a GICv3 interrupt controller is split into three groups:

- Distributor interface.
- Redistributor interface.
- CPU interface.

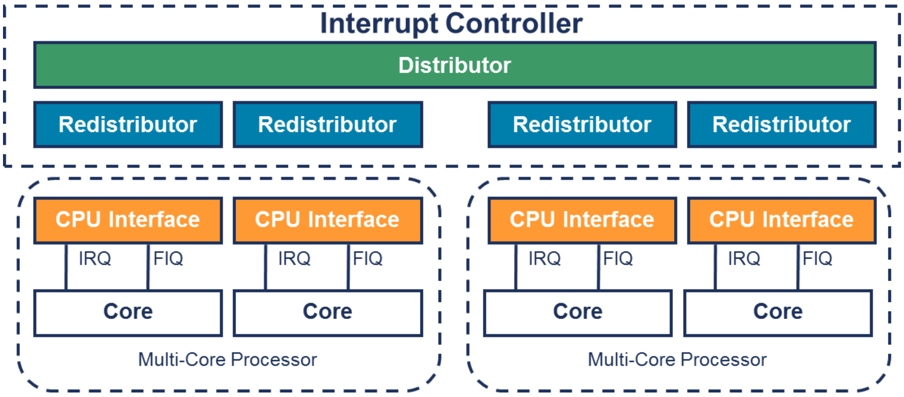


Figure 9 The programming interfaces of a GICv3 interrupt controller

Distributor (`GICD_*`)

The Distributor registers are memory-mapped, and contain global settings that affect all PEs connected to the interrupt controller. The Distributor provides a programming interface for:

- Interrupt prioritization and distribution of SPIs.
- Enabling and disabling SPIs.
- Setting the priority level of each SPI.
- Routing information for each SPI.
- Setting each SPI to be level-sensitive or edge-triggered.
- Generating message-based SPIs.
- Controlling the active and pending state of SPIs.
- Controls to determine the programmers’ model that is used in each Security state (affinity routing or legacy).

显然，GICD中包含的寄存器
主要配置SPI中断相关的信息

Redistributors (`GICR_*`)

For each connected PE there is a Redistributor. The Redistributors provides a programming interface for:

- Enabling and disabling SGIs and PPIs.
- Setting the priority level of SGIs and PPIs.
- Setting each PPI to be level-sensitive or edge-triggered.
- Assigning each SGI and PPI to an interrupt group.
- Controlling the state of SGIs and PPIs.
- Base address control for the data structures in memory that support the associated interrupt properties and pending state for LPIs.
- Power management support for the connected PE.

CPU interfaces (`ICC_*_ELn`)

Each Redistributor is connected to a CPU interface. The CPU interface provides a programming interface for:

- General control and configuration to enable interrupt handling.
- Acknowledging an interrupt.
- Performing a priority drop and deactivation of interrupts.
- Setting an interrupt priority mask for the PE.
- Defining the preemption policy for the PE.
- Determining the highest priority pending interrupt for the PE.

In GICv3 the CPU Interface registers are accessed as System registers (ICC_*_ELn).

Software must enable the System register interface before using these registers. This is controlled by the `SRE` bit in the `ICC_SRE_ELn` registers, where “n” specifies the Exception level (EL1-EL3).

NOTE: In GICv1 and GICv2 the CPU Interface registers were memory mapped (GICC_*).

NOTE: Software can check for GIC System register support by reading `ID_AA64PFR0_EL1` for the PE, see *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile* for details.

4. Configuring the GIC

This chapter describes how to enable and configure a GICv3 compliant interrupt controller in a bare metal environment. For detailed register descriptions see the *ARM® Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and 4*.

The configuration of LPIs is significantly different to the configuration of SPIs, PPIs and SGIs, and they are therefore described separately in Chapter 6.

Most systems that use a GICv3 interrupt controller are multi-core systems, and possibly also multi-processor systems. Some settings are global, that is, they affect all the connected PEs. Other settings are particular to a single PE.

This chapter will first look at the global settings, and then the per-PE settings.

4.1 Global settings

The Distributor control register (`GICD_CTLR`) must be configured to enable the interrupt groups and to set the routing mode.

- **Enable Affinity routing (`ARE` bits)**
  The `ARE` bits in `GICD_CTLR` control whether affinity routing is enabled. If affinity routing is not enabled, GICv3 can be configured for legacy operation . Whether affinity routing is enabled or not can be controlled separately for Secure and Non-secure state.

- **Enables**
  `GICD_CTLR` contains separate enable bits for Group 0, Secure Group 1 and Non-secure Group 1:

  - `GICD_CTLR.EnableGrp1S` enables distribution of Secure Group 1 interrupts.
  - `GICD_CTLR.EnableGrp1NS` enables distribution of Non-secure Group 1 interrupts.
  - `GICD_CTLR.EnableGrp0` enables distribution of Group 0 interrupts.

4.2 Individual PE settings

4.2.1 Redistributor configuration

On reset, a Redistributor treats the PE to which it is connected as sleeping. Wake-up is controlled through `GICR_WAKER`. To mark the connected PE as being awake, software must:

- Clear `GICR_WAKER` .`ProcessorSleep` to 0.
- Poll `GICR_WAKER` .`ChildrenAsleep` until it reads 0.

Enabling and configuring LPIs is described in Chapter 6.

Writing to the CPU interface registers, other than `ICC_SRE_ELn`, when either `GICR_WAKER.ProcessorSleep==1` or `GICR_WAKR.ChildrenAsleep==1` leads to UNPREDICTABLE behaviour.

4.2.2 CPU interface configuration

The CPU interface is responsible for delivering interrupts to the PE to which it is connected. To enable the CPU interface software must configure the following:

- **Enable System register access.**
  Chapter 3.5 describes the CPU interface registers, and how they are accessed as System registers in GICv3. Software must enable access to the CPU interface registers, by setting the SRE bit in the`ICC_SRE_ELn` registers.

- **Set priority mask and binary point registers.**
  The CPU interface contains the Priority Mask register (ICC_PMR_EL1) and the Binary Point registers (ICC_BPRn_EL1). The Priority Mask sets the minimum priority an interrupt must have in order to be forwarded to the PE. The Binary Point register is used for priority grouping and preemption. The use of both of these registers is described in more detail in Chapter 5.

- **Set EOI mode.**
  The EOImode bits in ICC_CTLR_EL1 and ICC_CTLR_EL3 in the CPU interface control how the completion of an interrupt is handled. This is described in more detail in chapter 5.5.

- **Enable signaling of each interrupt group.**
  The signalling of each interrupt group must be enabled before interrupts of that group will be forwarded by the CPU interface to the PE. To enable signaling software must write to ICC_IGRPEN1_EL1 register for Group 1 interrupts and ICC_IGRPEN0_EL1 registers for Group 0 interrupts.

  ICC_IGRPEN1_EL1 is banked by Security state. This means that ICC_GRPEN1_EL1 controls Group 1 for the current Security state. At EL3, software can access both Secure Group 1 interrupt enables and Non-secure Group 1 interrupt enables using ICC_IGRPEN1_EL3.

4.2.3 PE configuration

Some configuration of the PE is also required to allow it to receive and handle interrupts. A detailed description of this is outside of the scope of this document. It is sufficient here to describe the basic steps required for an ARMv8-A compliant PE executing in AArch64 state.

- **Routing controls**
  The routing controls for interrupts are in SCR_EL3 and HCR_EL2 of the PE. The routing control bits determine the Exception level to which an interrupt is taken. The routing bits in these registers have an UNKNOWN value at reset, so they must be initialized by software.

- **Interrupt masks**
  The PE also has exception mask bits in PSTATE. When these bits are set, interrupts are masked. These bits are set at reset.

- **Vector table**
  The location of the vector tables of the PE is set by the VBAR_ELn registers. As with SCR_EL3 and HCR_EL2, VBAR_ELn registers have an UNKNOWN value at reset. Software must set the VBAR_ELn registers to point to the appropriate vector tables in memory.

For more information, see *ARM® Architecture Reference Manual, ARMv8, for ARMv8-A architecture profile*.

4.3 SPI, PPI and SGI configuration

SPIs are configured through the Distributor, using the GICD_* registers. PPIs and SGIs are configured through the individual Redistributors, using the GICR_* registers.

For each INTID, software must configure the following:

- **Priority** (GICD_IPRIORITYn, GICR_IPRIORITYn)
  Each INTID has an associated priority, represented as an 8-bit unsigned value. 0x00 is the highest possible priority, and 0xFF is the lowest possible priority. Chapter 5

describes how the priority value in GICD_IPRIORITYn and GICR_IPRIORITYn masks low priority interrupts, and how it controls preemption.

An interrupt controller is not required to implement all 8 priority bits. A minimum of 5 bits must be implemented if the GIC supports two Security states. A minimum of 4 bits must be implemented if the GIC support only a single Security state.

- **Group** (GICD_IGROUPn, GICD_IGRPMODn, GICR_IGROUP0, GICR_IGRPMOD0)
  As described in section 3.4, an interrupt can be configured to belong to one of the three distinct interrupt groups. These interrupt groups are Group 0, Secure Group 1 and Non-secure Group 1.

- **Edge-triggered/level-sensitive** (GICD_ICFGRn, GICR_ICFGRn)
  If the interrupt is sent as a physical signal, it must be configured to be either edge-triggered or level-sensitive. SGIs are always treated as edge-triggered, and therefore GICR_ICFGR0 behaves as RAO/WI for these interrupts.

- **Enable** (GICD_ISENABLERn, GICD_ICENABLER, GICR_ISENABLER0, GICR_ICENABLER0)
  Each INTID has an enable bit. Set-enable registers and Clear-enable registers remove the requirement to perform read-modify-write routines. ARM recommends that the settings outlined in this section are configured before enabling the INTID.

For a bare metal environment, it is often unnecessary to change settings after initial configuration. However, if an interrupt must be reconfigured, for example to change the Group setting, it is advisable to first disable that particular INTID.

The reset values of most of the configurations registers are IMPLEMENTATION DEFINED. This means that the designer of the interrupt controller decides what the values are, and the values might vary between systems.

4.3.1 Setting the target PE for SPIs

For SPIs, the target of the interrupt must additionally be configured. This is controlled by GICD_IROUTERn. There is a GICD_IROUTERn register per SPI, and the Interrupt_Routing_Mode bit controls the routing policy. The options are:

- GICD_IROUTERn.Interrupt_Routing_Mode == 0
  The SPI is to be delivered to the PE A.B.C.D, the affinity co-ordinates specified in the register.

- GICD_IROUTERn.Interrupt_Routing_Mode == 1
  The SPI can be delivered to any connected PE that is participating in distribution of the interrupt group. The Distributor, rather than software, selects the target PE, and this can vary each time the interrupt is signaled.
  This type of routing is referred to as 1-of-N.

A PE can opt-out of receiving 1-of-N interrupts. This is controlled by the DPG1S, DPG1NS and DPG0 bits in GICR_CTLR.

5. Handling Interrupts

5.1 What happens when an interrupt becomes pending

Section 3.2 describes how an interrupt transitions from the *inactive* to the *pending* state when the source of the interrupt is asserted. This is typically due to a peripheral asserting a dedicated interrupt signal.

When an interrupt becomes pending, the interrupt controller decides whether to send the interrupt to one of the connected PEs. The PE which the interrupt controller selects, if any, depends on the following settings:

- **Group enables**
  Section 3.4 described how INTID is assigned to a Group (Group 0, Secure Group 1 or Non-secure Group 1). For each Group, there is a Group bit in the Distributor and in the CPU Interface. An interrupt that is a member of a disabled Group cannot be signaled to a PE.

- **Interrupt enables**
  Individually disabled interrupts can become pending, but will not be forwarded to a PE.

- **Routing controls**
  Depending on the type of interrupt, the interrupt controller must decide which PEs can receive the interrupt.

  For SPIs, this is controlled by `GICD_IROUTERn`. An SPI can target one specific PE, or any one of the connected PEs.

  For LPIs, the routing information comes from the ITS if an ITS is implemented (see section 6.1).

  PPIs are specific to one PE, and can only be handled by that PE.

  For SGIs, the originating PE defines the list of target PEs. This is described further in chapter 7.

- **Interrupt priority & priority mask**
  Each PE has a Priority Mask register (`ICC_PMR_EL1`) in its CPU Interface. This register sets the minimum priority that is required for an interrupt to be forwarded to that PE. Only interrupts with a higher priority than the value in the register are signaled to the PE.

- **Running priority**
  Section 5.4 covers *running priority*, and how this affects preemption. If the PE is not already handling an interrupt, the running priority is the idle priority (`0xFF`). Only an interrupt with a higher priority than the running priority can preempt the current interrupt.

5.2 Interrupt acknowledge

The CPU interface has two IARs. Reading the IAR returns the INTID, and advances the interrupt state machine. In a typical interrupt handler, one of the first steps when handling an interrupt is to read one of the IARs.

Table 7 Interrupt acknowledge registers

Register	Use
`ICC_IAR0_EL1`	Used to acknowledge Group 0 interrupts.
`ICC_IAR1_EL1`	Used to acknowledge Group 1 interrupts.

5.3 Spurious interrupts

Section 3.1.2 describes how the INTID range 1020 to 1023 is reserved for special purposes. These INTIDs can be returned by reads of the IARs, and indicate special cases in exception handling.

Table 8 Reserved IDs

ID	Meaning	Example scenario
1020	Only returned by reads of `ICC_IAR0_EL1`. Highest pending interrupt is Secure Group 1. Only seen when taking FIQ to EL3	An interrupt for the Trusted OS was signaled while the PE was executing in Non-secure state. This is taken as an FIQ to EL3, so that the Secure Monitor could context switch to the Trusted OS.
1021	Only returned by reads of `ICC_IAR0_EL1`. Highest pending interrupt is Non-secure Group 1. Only seen when taking FIQ to EL3	An interrupt for the rich OS was signaled while the PE was executing in Secure state. This would be taken as a FIQ to EL3, so that the Secure Monitor could context switch to the rich OS.
1022	Used only for legacy operation.	Legacy operation is not addressed in this document.
1023	Spurious interrupt. There are no enabled INTIDs in the pending state, or all INTIDs in that pending are of insufficient priority to be taken.	When polling the IARs, this value indicates that there are no interrupts to available to acknowledge.

Example

In the following example, a mobile system has a modem interrupt which signals an incoming phone call. This interrupt is intended to be handled by the Rich OS in the Non-secure state.

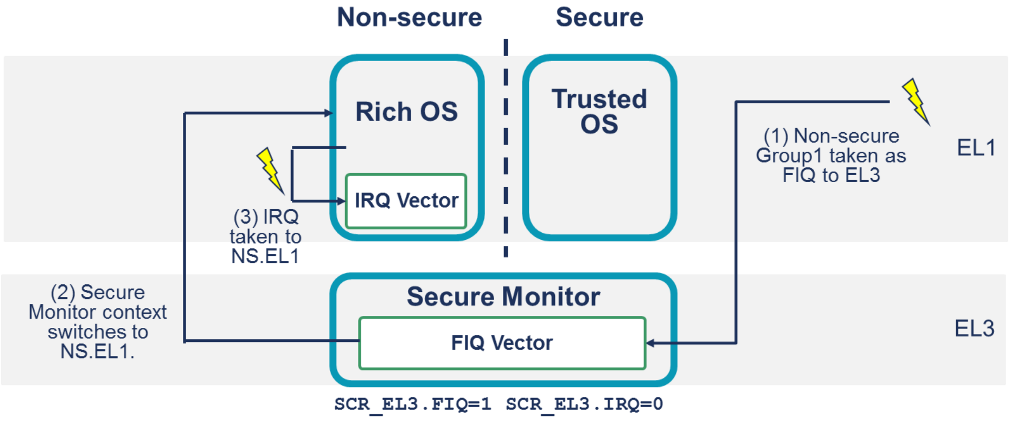

Non-secure
Secure
Rich OS
IRQ Vector
Trusted OS
(1) Non-secure Group1 taken as FIQ to EL3
EL1
(3) IRQ taken to NS.EL1
(2) Secure Monitor context switches to NS.EL1.
Secure Monitor
FIQ Vector
EL3
SCR_EL3.FIQ=1 SCR_EL3.IRQ=0

Figure 10 Example of using reserved INTID 1021

1. The modem interrupt becomes pending while the PE is executing the Trusted OS at Secure EL1. As the modem interrupt is configured as Non-secure Group 1, it will be signaled as an FIQ. With `SCR_EL3.FIQ==1`, the exception is taken to EL3.

2. Secure Monitor software executing at EL3 reads the IAR, which returns 1021. This values indicates that the interrupt is expected to be handled in Non-secure state. The Secure Monitor then performs the necessary context switching operations.

3. Now that the PE is in Non-secure state, the interrupt is signaled as an IRQ and taken to Non-secure EL1 to be handled by the Rich OS.

In the example shown in Figure 10 the Non-secure Group 1 interrupt caused an immediate exit from the Secure OS. This might not always be required or wanted. Figure 11 shows an alternative model, where the interrupt is initially taken to Secure EL1.

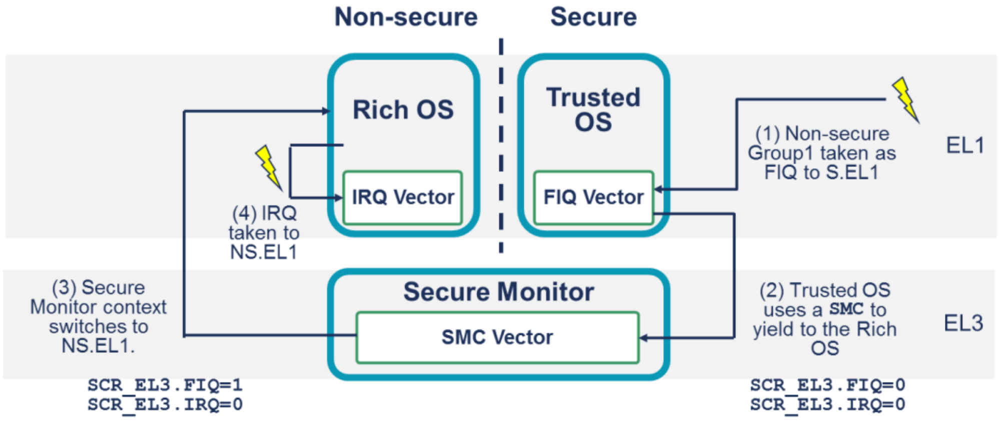


Figure 11 Alternative routing model

1. The modem interrupt becomes pending while the PE is executing the Trusted OS at Secure EL1. As the modem interrupt is configured as Non-secure Group 1, it will be signaled as an FIQ. With `SCR_EL3.FIQ==0`, the exception is taken to Secure EL1.

2. The Trusted OS performs actions to tidy up its internal state. When it is ready, the Trusted OS uses an `SMC` instruction to yield to Non-secure state.

4. The SMC exception is taken to EL3. The Secure Monitor software executing at EL3 performs the necessary context switching operations.

5. Now that the PE is in Non-secure state, the interrupt is signaled as an IRQ and taken to Non-secure EL1 to be handled by the Rich OS.

5.4 Running priority & preemption

The PMR sets the minimum priority that an interrupt must have to be forwarded to a particular PE. The GICv3 architecture has the concept of a running priority. When a PE acknowledges an interrupt, its running priority becomes that of the interrupt. The running priority returns to its former value when the PE writes to one of the EOI registers.

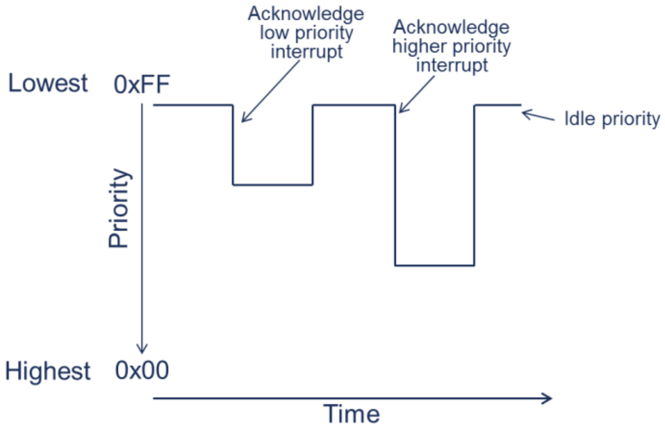


Figure 12 Running priority value over time

The current running priority is reported in the Running Priority register in the CPU interface (`ICC_RPR_EL1`).

The concept of running priority is important when considering preemption. Preemption occurs when a high priority interrupt is signaled to a PE that is already handling a lower priority interrupt. Preemption introduces some additional complexity for software, but it can prevent a low priority interrupt blocking the handling of a higher priority interrupt.

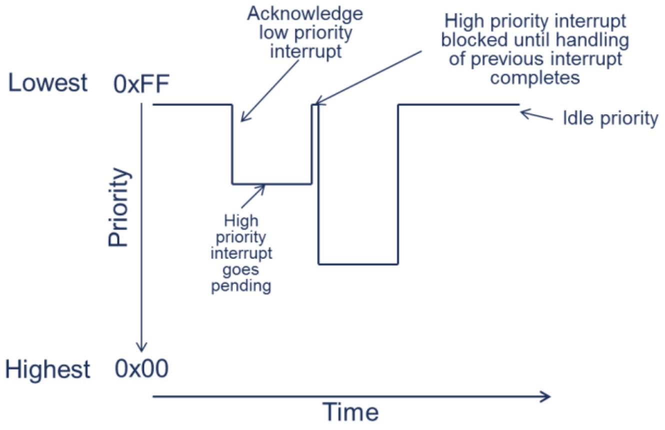


Figure 13 Without preemption

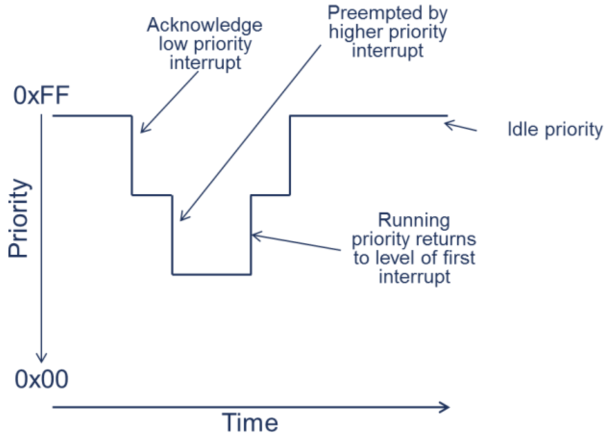


Figure 14 With preemption

Figure 14 shows one level of preemption. However, it is possible to have multiple levels of preemption.

In some situations preemption might not be required or wanted. The GICv3 architecture allows the difference in priority required for preemption to be controlled through the Binary Point registers (`ICC_BPRn_EL1`).

The Binary Point registers split the priority into two fields, group priority and subpriority:

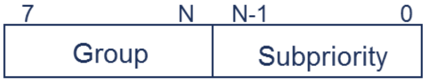


Figure 15 Eight bit priority value split between group priority and subpriority fields

For preemption, only the group priority bits are considered. The subpriority bits are ignored.

For example, consider the following three interrupts:

INTID A has priority `0x10`

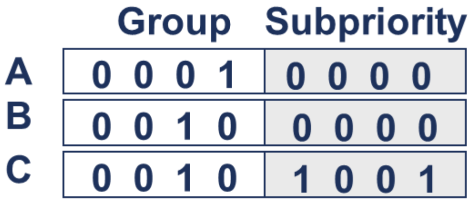
INTID B has priority `0x20`

INTID C has priority `0x21`

In this scenario it is decided that:

- A can preempt B or C.
- B cannot preempt C, because B and C have similar priorities.

To achieve this the the split between Group and Subpriority could be set at N=4:



	Group	Subpriority
A	0 0 0 1	0 0 0 0
B	0 0 1 0	0 0 0 0
C	0 0 1 0	1 0 0 1

Figure 16 Group priority/Subpriority example

With this split, B and C are now considered to have the same priority for the purpose of preemption. However, A still has a higher priority so it can preempt either B or C.

NOTE: Preemption requires that the interrupt handler, or handlers, are written to support nesting. Details of this are outside of the scope of this document, and are not described here.

5.5 End of interrupt

When the interrupt has been handled, software must inform the interrupt controller that the interrupt has been handled so that the state machine can transition to the next state. The GICv3 architecture treats this as two tasks:

- **Priority drop**
  This means dropping the running priority back to the value that it had before the interrupt was taken.
- **Deactivation**
  This means updating the state machine of the interrupt that is currently being handled. Typically this will be a transition from the *Active* state to the *Inactive* state.

In the GICv3 architecture priority drop and deactivation can happen together or separately. This is determined by the settings of `ICC_CTLR_ELn.EOImode`.

- **`EOImode = 0`**
  A write to `ICC_EOIR0_EL1` for Group 0 interrupts, or `ICC_EOIR1_EL1` for Group 1 interrupts, performs both the priority drop and deactivation. This is the model typically used for a simple bare metal environment.

- **`EOImode = 1`**
  A write to `ICC_EOIR_EL10` for Group 0 interrupts, or `ICC_EOIR1_EL1` for Group 1 interrupts results in a priority drop. A separate write to `ICC_DIR_EL1` is required for deactivation. This mode is often used for virtualization purposes.

When writing these registers, software must write the INTID.

5.6 Checking the current state of the system

5.6.1 Highest priority pending interrupt and running priority

As the names suggests, the Highest Priority Pending Interrupt registers (`ICC_HPPIR0_EL1` & `ICC_HPPIR1_EL1`) report the INTID of the highest priority pending interrupt for this PE. This might be different on different PEs, for example because of different routing settings for SPIs.

Running priority was introduced in section 5.4, and is reported by the Running Priority register (`ICC_RPR_EL1`).

5.6.2 State of individual INTIDs

The Distributor provides registers that indicate the current state of each SPI. Similarly the Redistributors provide registers that indicate the state of PPIs and SGIs for their connected PEs.

These registers can also move an interrupt to a specific state. This can be useful, for example, for testing that the configuration is correct without requiring the peripheral to assert the interrupt.

There are separate registers to report the active state and the pending state. Table 9 lists the active state registers. The pending state registers have the same format.

Table 9 Active State registers

Register	Description
GICD_ISACTIVERn	Sets the active state for SPIs. One bit per INTID. Reads of a bit return the current state of the INTID: • 1 – the INTID is active • 0 – the INTID is not active Writing 1 to a bit activates the corresponding INTID. Writing 0 to a bit has not effect.
GICD_ICACTIVERn	Clears the active state for SPIs. One bit per INTID. Reads of a bit return the current state of the interrupt: • 1 – the INTID is active • 0 – the INTID is not active Writing 1 to a bit deactivates the corresponding INTID. Writing 0 to a bit has not effect.
GICR_ISACTIVER0	Sets the active state for SGIs and PPIs. One bit INTID. (Covers INTIDs 0 to 31, which are private to each PE) Reads of a bit return the current state of the interrupt: • 1 – the INTID is active • 0 – the INTID is not active Writing 1 to a bit activates the corresponding INTID. Writing 0 to a bit has not effect.
GICR_ICACTIVER0	Clears the active state for SGIs and PPIs. One bit INTID. (Covers INTIDs 0 to 31, which are private to each PE) Reads of a bit return the current state of the interrupt • 1 – the INTID is active • 0 – the INTID is not active Writing 1 to a bit deactivates the corresponding INTID. Writing 0 to a bit has not effect.

NOTE: GICD_ISACTIVER0 and GICD_ICACTIVER0 are treated as RES0 when affinity routing is enabled. This is because GICD_ISACTIVER0 and GICD_ICACTIVER0 correspond to INTIDs 0 to 31, which are banked per-PE and reported through the Redistributor of each PE.

NOTE: Software executing in Non- secure state cannot see the state of Group 0 or Secure Group 1 interrupts, unless access is permitted by GICD_NASCRn or GICR_NASCRn.

6. Configuring LPIs

LPIs are only supported when affinity routing is enabled, and they are configured differently compared to the other interrupt types.

Configuring LPIs involves setting up the:

- Redistributors.
- The optional ITSs (Interrupt Translation Service).

LPI都是基于MSI

LPIs are always message–based interrupts, and they can be supported by an ITS. An ITS is responsible for receiving interrupts from peripherals and forwarding them to the appropriate Redistributor as LPIs. A system might include more than one ITS, in which case each ITS must be configured individually.

A peripheral can also send the LPI directly to a Redistributor, bypassing the ITS. However, the ITS provides a number of features to allow efficient handling of large numbers of interrupt sources.

NOTE: Support for LPIs is optional, and is indicated by GICD_TYPER.LPIS. If at least one ITS is present, it is IMPLEMENTATION DEFINED whether a peripheral can send LPIs directly to a Redistributor, bypassing the ITSs.

LPI可以绕过ITS直接由外设发送给GICR，但如果平台提供了ITS，则由硬件实现决定是否可以这么做。

6.1 ITS

6.1.1 Operation of an ITS

A peripheral generates an LPI by writing to GITS_TRANSLATER in the ITS. The write provides the ITS with the following information:

- **EventID**
  This is the value written to GITS_TRANSLATER. The EventID idenitifies which interrupt the peripheral is sending. The EventID might be the same as the INTID, or it might be translated by the ITS into the INTID.

- **DeviceID**
  The DeviceID identifies the peripheral. The manner in which a DeviceID is generated is IMPLEMENTATION DEFINED. For example, the AXI User signals could be used.

The ITS translates the EventID that is written to GITS_TRANSLATER by the peripheral to an INTID. How the EventID translates into an INTID is specific to each peripheral, which is why a DeviceID is required.

LPI INTIDs are grouped together in *collections*. All INTIDs in a collection are routed to the same Redistributor. Software allocates LPI INTIDs to Collections, allowing it to efficiently move interrupts from one PE to another.

An ITS uses three types of table to handle the translation and routing of LPIs. These are:

- **Device Tables**
  These map DeviceIDs to Interrupt Translation Tables.

- **Interrupt Translation Tables**
  These contain the DeviceID specific mappings between EventID and INTID. They also contain the Collection of which the INTID is a member.

- **Collection Tables**
  These map collections to Redistributors.

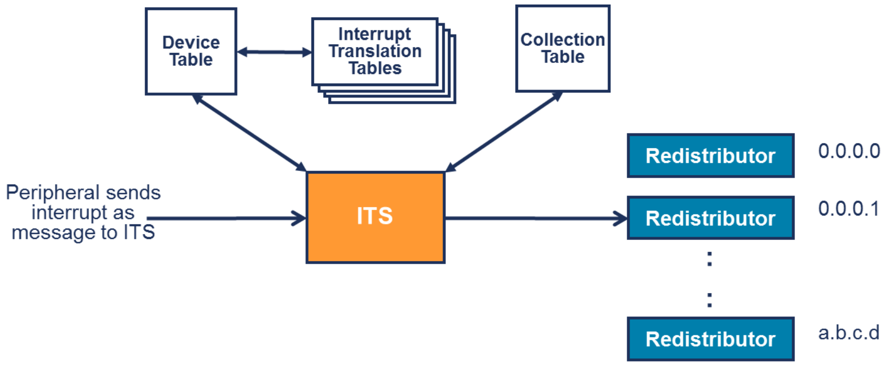


Figure 17 An ITS forwarding an LPI to a Redistributor

When a peripheral writes to `GITS_TRANSLATER`, the ITS:

1. Uses the DeviceID to select the appropriate entry from the Device Table. This entry identifies which Interrupt Translation Table to use.
2. Uses the EventID to select the appropriate entry from the Interrupt Translation Table. This entry provides the INTID, and the Collection ID.
3. Uses the Collection ID to select the required entry in the Collection Table, which returns the routing information.
4. Forwards the interrupt to the target Redistributor.

NOTE: An ITS can optionally support a number of hardware collections. Hardware collections are where the ITS holds the configuration internally, rather than storing it in memory. `GITS_TYPER.HCC` reports the number of hardware collections that are supported.

6.1.2 The command queue

An ITS is controlled using a command queue in memory. The command queue is a circular buffer and it is defined by three registers.

- **`GITS_CBASER`**
  This register specifies the base address and size of the command queue. The command queue must be 64KB aligned, and the size must be a multiple of 4KB. Each entry in the command queue is 32 bytes. `GITS_CBASER` also specifies the cacheability and shareability settings that the ITS uses when accessing the command queue.
- **`GITS_CREADR`**
  This register points to the next command that the ITS will process.
- **`GITS_CWRITER`**
  This register points to the entry in the queue where the next new command should be written.

Figure 18 shows a simplified representation of a command queue.

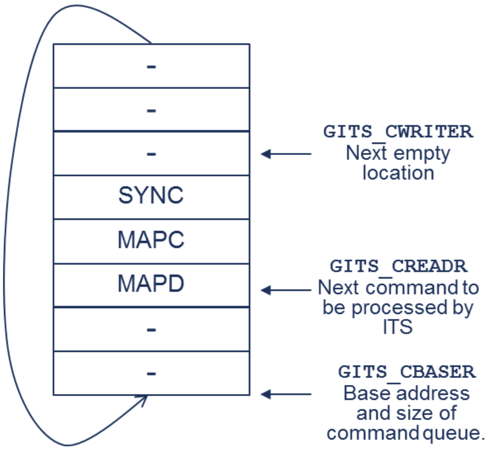

-
-
-
SYNC
MAPC
MAPD
-
-
GITS_CWRITER
Next empty location
GITS_CREADR
Next command to be processed by ITS
GITS_CBASER
Base address and size of command queue.

Figure 18 ITS circular command queue

The *ARM® Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and 4.0* provides details of all the commands supported by an ITS, and how these are encoded.

6.1.3 Initial configuration of an ITS

To configure an ITS at system start up, software must:

1. **Allocate memory for the Device and Collection tables.**
   The `GITS_BASER[0..7]` registers specify the base address and size of the ITS Device and Collection tables. Software uses these registers to discover the number and type of tables that the ITS supports. Software must then allocate the required memory, and set the `GITS_BASERn` registers to point to this allocated memory.

2. **Allocate memory for the command queue.**
   Software must allocate the memory for the command queue and set `GITS_CBASER` and `GITS_CWRITER` to point to the start of this allocated memory.

3. **Enable the ITS**.
   When the tables and command queue have been allocated, the ITS can be enabled. This is done by setting the `GITS_CTLR.Enable` bit to 1.
   Once `GITS_CTLR.Enable` has been set, the `GITS_BASERn` and `GITS_CBASER` registers become read-only.

6.1.4 The sizes and layout of Collection and Device tables

The location and size of the Device and Collection tables is configured using the `GITS_BASERn` registers. Software must allocate sufficient memory for these tables, and configure the `GITS_BASERn` registers, before enabling the ITS.

Software can allocate a flat (single level) table, or two-level tables. This is specified by `GITS_BASERn.Indirect`.

NOTE: Support for two-level tables is OPTIONAL. If the ITS only supports flat tables, `GITS_BASERn.Indirect` is RAZ/WI.

Flat level tables

With a flat table, a single contiguous block of memory is allocated to the ITS to record mappings. Software is required to fill the memory with 0s before enabling the ITS. Thereafter the table is populated by the ITS as it processes commands from the command queue.

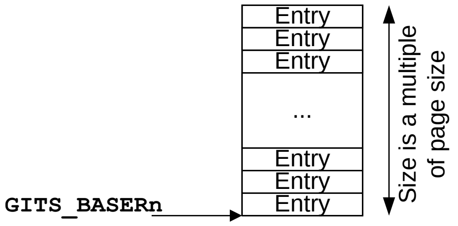


Figure 19 A flat Device or Collection table

The size of the table scales with the width of DeviceID or Collection ID, as appropriate. The required size can be calculated as follows:

Size in bytes = 2^{ID_width} * entry_size

Where *entry_size* is the number of bytes per table entry, and is reported by `GITS_BASERn.Entry_Size`.

When configuring the `GITS_BASERn` registers, the size of the table is specified as a number of pages. The size of a page is controlled by `GITS_BASERn.Page_Size`, and can be 4KB, 16KB or 64KB. Therefore, the result of the formula given above must be *rounded up* to the next whole page size.

For example, if a system implements an 8-bit DeviceID, the bytes per table entry is 8 and a 4K page size is used:

2^8 * 8 = 2048 bytes => which rounded up to the next full page is 4K

Two-level tables

With two-level tables, software allocates a single first level table, and a number of second level tables.

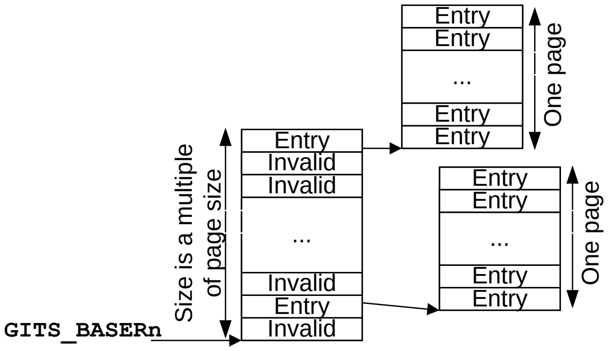


Figure 20 A two-level Device or Collection table

The first level table is populated by software, with each entry either pointing at a second level table or marked as invalid. The second level tables must be filled with 0s before they are allocated to the ITS, and are populated by the ITS as it processes commands from the command queue.

While the ITS is enabled (`GITS_CTLR.Enabled==1`) software might allocate additional second level tables, and update the corresponding first level table entry to point at these additional tables. Software must not remove allocations, or change existing allocations, while the ITS is enabled.

The size of each second level table is one page. As with the flat tables, the page size is configured by `GITS_BASERn.Page_Size`. It therefore contains (page_size / entry_size) entries.

Each first level table entry represents (page_size / entry_size) IDs, and can either point to a second level table or be marked as *invalid*. Any ITS command that uses an ID which corresponds to an invalid entry will be discarded.

The required size of the first level table can be calculated by:

$$\text{Size in bytes} = (2^{\text{ID\_width}} \;/\; (\text{page\_size} / \text{entry\_size})) \;*\; 8$$

As with the single level tables, the size of the first level table is specified as a number of pages. Therefore the result of the formula must be rounded up to the next whole page size.

6.1.5 Adding a new command to the command queue

To add a new command to the command queue, software must:

1. **Write the new command to the queue.**
   `GITS_CWRITER` points to the next entry that does not contain a valid command in the command queue. Software must write the command to this entry, and it must ensure global visibility.

2. **Update `GITS_CWRITER`**
   Software must update `GITS_CWRITER` to the next entry that does not contain a new command. Updating `GITS_CWRITER` informs the ITS that a new command has been added.

   Software can add multiple commands to the queue at the same time, provided there are enough empty spaces in the command queue and that `GITS_CWRITER` is updated accordingly.

3. **Wait for the command to be read by the ITS**
   Software can check whether the command has been read by the ITS by polling `GITS_CREADR`. All commands have been read by the ITS when `GITS_CWRITER.Offset == GITS_CREADR.Offset`.

   Alternatively, an `INT` command can be added to generate an interrupt to signal that a group of commands has been read by the ITS.

   The ITS reads the commands from the command queue in order. However, the effects that these commands have on the Redistributors might be visible out-of-order. A `SYNC` command can ensure that the effects of previously issued commands are visible.

NOTE: The command queue is full when `GITS_CWRITER` points at the location before `GITS_CREADR`. Software must check that there is sufficient space in the queue before attempting to add new commands.

6.1.6 Mapping an interrupt to a Redistributor

Mapping a DeviceID to a translation table.

Every peripheral that can send interrupts to an ITS has its own DeviceID. Each DeviceID requires its own Interrupt Translation Table (ITT) to hold its EventID to INTID mappings. Software must allocate memory for the ITT, and then use the `MAPD` command to map the DeviceID to the ITT.

deviceid -> eventid

```
MAPD <DeviceID>, <ITT_Address>, <Size>
```

Mapping INTIDs to a collection, and collections to a Redistributor

When the DeviceID of a peripheral has been mapped to an ITT, the different EventIDs it can send must be mapped to INTIDs, and these INTIDs must be mapped to collections. Each collection is mapped to a target Redistributor.

INTIDs can be mapped to a collection using the `MAPTI` and `MAPI` commands. The `MAPI` command is used when the EventID and INTID are the same.

(deviceid, eventid) -> (intid, colid)

```
MAPI <DeviceID>, <EventID>, <Collection ID>
```

The MAPTI command is used when the EventID and INTID are different.

```
MAPTI <DeviceID>, <EventID>, <INTID>, <Collection ID>
```

colid -> redist

Collections are mapped to a Redistributor using the MAPC command:

```
MAPC  <Collection ID>, <Target Redistributor>
```

Idenitification of the target Redistributor depends on GITS_TYPER.PTA:

- GITS_TYPER.PTA==0
  The Redistributor is specified by ID, which can be read from GICR_TYPER.Processor_Number.
- GITS_TYPER.PTA==1
  The Redistributor is specified by physical address.

Example

A timer has DeviceID 5 and uses a two bit EventID. We want EventID 0 to be mapped to INTID 8725. The ITT allocated for the timer is at address 0x84500000.

We decide to use collection number 3 and deliver the interrupt to the Redistributor at physical address 0x78400000.

The command sequence for this is:

```
MAPD 5, 0x84500000, 2    Map DeviceID 5 to an ITT
MAPTI 5, 0, 8725, 3      Map EventID 0 to INTID 8725 and collection 3
MAPC  3, 0x78400000      Map collection 3 to Redistributor at address 0x78400000
SYNC 0x78400000
```

NOTE: The example assumes that none of the mappings have previously been set up, and that GITS_TYPER.PTA==1.

6.1.7 Migrating interrupts between Redistributors

There is more than one way to move an interrupt from one Redistributor to another.

- **Remapping a collection**
  Software can move all interrupts from one Redistributor to a different Redistributor by rermapping the entire collection. This is typically done when the PE attached to the Redistributor is powering down, and the interrupts must be moved to another Redistributor. This can be done using the following command sequence:

  ```
  MAPC <Collection ID>, <RDADDR2>   Remap collection to new Redistributor
  SYNC <RDADDR2>                    Ensure visibility of the mapping
  MOVALL <RDADDR1>, <RDADDR2>       Move pending state to new Redistributor
  SYNC <RDADDR1>                    Ensure visibility of move
  ```

  In this command sequence RDADDR1 is the previously targeted Redistributor, and RDADDR2 is the new target Redistributor.

If there were multiple Collections targeting RDADDR1, then we would need multiple MAPC commands, one for each collection. This sequence assumes that all the collections are being remapped to the same new target Redistributor.

- **Mapping an interrupt to a different collection**
  Individual interrupts can be remapped to a different collection. This can be done using the following command sequence:

  ```
  MOVI <DeviceID>, <EventID>, <ID of new Collection>
  SYNC <RDADDR1>
  ```

  In this command sequence RDADDR1 is the Redistributor that is targeted by the collection to which the interrupt was originally assigned, before the interrupt was remapped.

6.1.8 Removing interrupts mappings

To remap or remove the mapping of an interrupts, software must:

1. Disable the physical INTID to which interrupt is currently mapped. This is done in the LPI configuration tables, see section 6.2.2.
2. Issue a DISCARD command. This removes the mapping of the interrupt and clears the pending state of the mapped INTID.
3. Issue a SYNC command, and wait until the command has completed.

After the command has completed, no more interrupts are delivered to the Redistributor to which the interrupts were previously mapped.

6.1.9 Remapping or removing the mapping of devices

To change or remove the mapping for devices software must:

1. Follow the steps in 6.1.8 for each EventID of that peripheral that is currently mapped.
2. Issue a MAPD command to remap the device. Alternatively, a MAPD command with the valid bit cleared to 0 removes the mapping.
3. Issue a SYNC command and wait until the command has completed.

6.2 Redistributors

The Redistributors hold the control, prioritization, and pending information for all physical LPIs, using tables held in memory.

Configuration information for LPIs is stored in a table in memory. This is the LPI Configuration tables, which is pointed to by GICR_PROPBASER. LPI configuration is global, that is, all Redistributors must see the same configuration. Typically a system has a single LPI Configuration table that is shared by all Redistributors.

Similarly, state information for LPIs is also stored in tables in memory. These are the LPI Pending tables, which are pointed to by GICR_PENDBASER. Each Redistributor has its own LPI Pending table, and these tables are not shared between Redistributors.

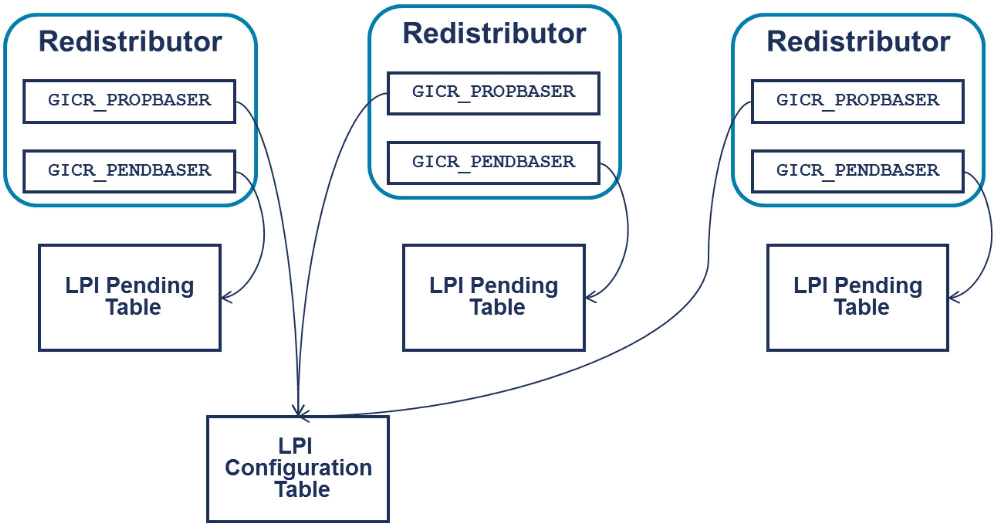


Figure 21 LPI Configuration and LPI Pending tables

6.2.1 Initial configuration of a Redistributor

The steps to initialize the Redistributors in a system are:

1. Allocate memory for the LPI Configuration table, and initialize the table with the appropriate configurations for each LPI.

2. Set `GICR_PROPBASER` in each Redistributor to point at the LPI Configuration table.

3. Allocate memory for the LPI Pending table of each Redistributor, and initialize the content of each table. At system start-up, this typically means zeroing the memory, meaning that all LPI INTIDs are in the inactive state.

4. Set `GICR_PENDBASER` in each Redistributor to point to the particular LPI Pending table associated with the Redistributor.

5. Set `GICR_CTLR.EnableLPIs` to 1 in each Redistributor to enable LPIs.
   When `GICR_CTLR.EnableLPIs` has been set to 1, the `GICR_PENDBASER` and `GICR_PROPBASER` registers become read-only

LPI Configuration table

The LPI Configuration table has one byte for each LPI INTID. Figure 22 shows the format of these entries.

7	2	1	0
Priority		RES0	Enable

Figure 22 Format of an entry in the LPI Configuration table

Although priority values are 8 bits for SPIs, PPIs and SGIs, there are only 6 bits in the table to record the priority of an LPI. The lower two bits of the priority of an LPI are always treated as `0b00`.

There is no field for recording the security configuration. LPIs are always treated as Non-secure Group 1 interrupts.

The size of the LPI Configuration table and the amount of memory that must be allocated depend on the number of LPIs. The maximum number of INTIDs (SPIs, PPIs, SGIs and LPIs) that are supported by the GIC is indicated by `GICD_TYPER.IDbits`. The LPI Configuration table handles LPIs, which use INTIDs that are greater than 8191. Therefore to support all the possible LPIs the LPI Configuration table size is calculated as follows:

$$\text{Size in bytes} = 2^{\texttt{GICD\_TYPER.IDbits+1}} - 8192$$

However, it is possible to support a smaller range of INTIDs. `GICR_PROPBASER` also includes an `IDbits` fields, that indicates the number of INTIDs that are supported by the LPI Configuration table. This number must be equal to or smaller than the value in `GICD_TYPER`. Software must allocate enough memory for this number of entries. In this case the required size of the LPI Configuration table becomes:

$$\text{Size in bytes} = 2^{\texttt{GICR\_PROPBASER.IDbits+1}} - 8192$$

The interrupt controller must be able to read the memory allocated for the LPI Configuration table. However, it never writes to this memory.

LPI Pending tables

The states information for LPIs is stored in memory. LPIs have two states, *inactive* and *pending*.

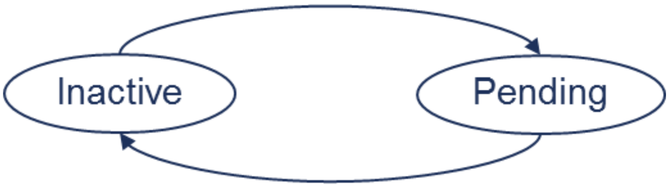


Figure 23 State machine for LPIs

Interrupts transition from pending to inactive when they have been acknowledged.

Because there are only two states, there is only 1 bit for each LPI in the LPI Pending tables. Therefore, to support all possible INTIDs in an implementation, the tables must be:

$$\text{Size in bytes} = (2^{\texttt{GICD\_TYPER.IDbits+1}}) / 8$$

Unlike the LPI Configuration table, the size of the LPI Pending tables is not adjusted to take account of LPIs starting at INTID 8192. The first 1KB of the table (corresponding to the entries for INTIDs 0 to 8291) stores IMPLEMENTATION DEFINED state.

As described in this section, it is possible to use a smaller range of INTIDs than is supported by hardware. `GICR_PROPBASER.IDBits` controls the size of the INTID range. Therefore, it affects both the size of the LPI Configuration tables and the size of the LPI Pending table. To support the configured INTID range, the required LPI Pending table size is:

$$\text{Size in bytes} = (2^{\texttt{GICR\_PROPBASER.IDbits+1}})/8$$

The interrupt controller must be able to read and write the memory allocated for the LPI Pending table. Typically, a Redistributor will cache the highest priority pending interrupts internally, and write out state information to the LPI Pending table when there are too many pending interrupts to cache or when entering a low power state.

6.2.2 Reconfiguring LPIs

LPI configuration information is stored in a table in memory, not in registers. Redistributors are allowed to cache the LPI configuration information. This means that to reconfigure an LPI, software must:

1. Update the entry in the LPI Configuration table.
2. Ensure global visibility of the update or updates.
3. Invalidate any caching of the configuration in the Redistributors.

The invalidation of the cache in the Redistributor is performed by issuing the ITS `INV` or `INVALL` commands. The `INV` command invalidates the entry for a specific interrupt, so this command is typically used when reconfiguring a small number of LPIs. The `INVALL` command invalidates entries for all interrupts in a specified collection. For more information about ITS commands, see section 6.1.5.

If an ITS is not implemented, software must write to `GICR_INVLPIR` or `GICR_INVALLR` in any of Redistributors instead.

7. Sending and receiving SGIs

Software Generated Interrupts, SGIs, are interrupts that software can trigger by writing to a register in the interrupt controller.

7.1 Generating SGIs

An SGI is generated by writing to one of the SGI registers in the CPU interface.

Table 10 SGI registers that are used when System register access is enabled

System register interface	Description
`ICC_SGI0R_EL1`	Generates a Secure Group 0 interrupt
`ICC_SGI1R_EL1`	Generates a Group 1 interrupt, for the current Security state of the PE
`ICC_ASGI1R_EL1`	Generates a Group 1 interrupt, for the other Security state of the PE

The basic format of the SGI registers is shown in Figure 24.

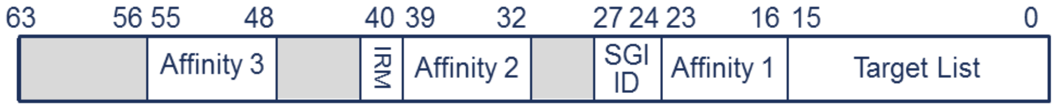


Figure 24 Format of the SGI registers, when `SRE`=1

Controlling the SGI ID

The SGI ID field controls which INTID is generated. As described in section 3.1.2, INTIDs 0-15 are used for SGIs.

Controlling the target

The IRM (Interrupt Routing Mode) field in the SGI registers controls which PE or PEs an SGI is sent to. There are two options:

- `IRM = 0`
  The interrupt is sent to <aff3>.<aff2>.<aff1>.<Target List>, where <target list> is encoded as 1 bit for each affinity 0 node under <aff1>. This means that the interrupt can be sent to a maximum of 16 PEs, which might include the originating PE.
- `IRM = 1`
  The interrupt is sent to all connected PEs, except the originating PE (self).

As described in section 3.3, the exact meaning of the hierarchal affinity fields depends on the particular design. Typically, affinity level 1 identifies a multi-core processor and affinity level 0 a PE within that processor.

Controlling the Security state and grouping

The Security state and grouping of SGIs is controlled by:

- The SGI register (ICC_SGI0R_EL1, ICC_SGI1R_EL1, or ICC_ASGIR_EL1) that is written by software on the originating PE.
- The GICR_IGROUPR0 and GICR_IGRPMODR0 registers of the target PE or PEs.

Software executing in Secure state can send both Secure and Non-secure SGIs. Whether software executing in Non-secure state can generate Secure SGIs is controlled by GICR_NSACR. This register can only be accessed by software executing in Secure state. Table 11 shows how the Security state of the originating PE, the interrupt handling configuration of the PE which the interrupt is targetting, and the SGI register, affect whether an interrupt is forwarded or not.

Table 11 SGI security/group controls, when GICD_CTLR.DS=0

Security state of sending PE	SGI register written	Configuration on receiving PE	Forwarded?
Secure EL3/EL1	ICC_SGI0R_EL1	Secure Group 0	Yes
		Secure Group 1	No
		Non-secure Group 1	No
	ICC_SGI1R_EL1	Secure Group 0	No (*)
		Secure Group 1	Yes
		Non-Secure Group 1	No
	ICC_ASGI1R_EL1	Secure Group 0	No
		Secure Group 1	No
		Non-secure Group 1	Yes
Non-secure EL2/EL1	ICC_SGI0R_EL1	Secure Group 0	Configurable by GICR_NSACR (*)
		Secure Group 1	No
		Non-secure Group 1	No
	ICC_SGI1R_EL1	Secure Group 0	Configurable by GICR_NSACR (*)
		Secure Group 1	Configurable by GICR_NSACR
		Non-secure Group 1	Yes
	ICC_ASGI1R_EL1	Secure Group 0	Configurable by GICR_NSACR (*)
		Secure Group 1	Configurable by GICR_NSACR
		Non-secure Group 1	No

NOTE: Table 11 assumes that `GICD_CTLR.DS==0`. When `GICD_CTLR.DS==1`, the SGIs marked with (*) are also forwarded.

7.2 GICv3 vs GICv2

In GICv2, SGI INTIDs are banked by the originating PE and the target PE. This means that a given PE could have the same SGI INTID pending a maximum of eight times, once from each PE in the system.

In GICv3, SGIs are only banked by the target PE. This means that a given PE can only have one instance of an SGI INTID pending.

This difference is best illustrated with an example. PEs A and B simultaneously send SGI INTID 5 to PE C.

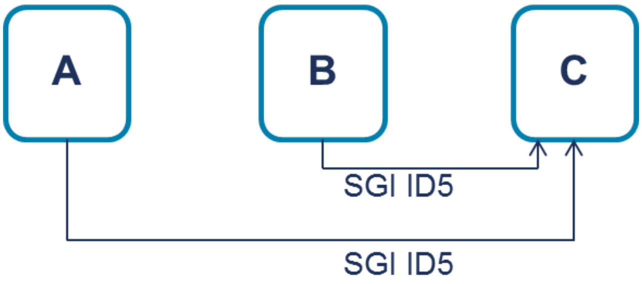


Figure 25 Multiple senders of the same ID example

How many interrupts will C see?

- GICv2: Two.
  It will see both the interrupts from A and B. The order of the two interrupts is dependent on the individual design and the precise timing. The two interrupts can be distinguished by the fact that the ID of the originating PE is prefixed to the INTID that is returned by `GICC_IAR`.

- GICv3: One.
  Because the originating PE does not bank the SGI, the same interrupt cannot be pending on two PEs. Therefore, C only sees one interrupt, with ID 5 (no prefix).

The example assumes that the two interrupts are sent simultaneously or almost simultaneously. If C were able to acknowledge the first SGI before the second arrived, then C would see two interrupts in GICv3 as well.

NOTE: During legacy operation, that is when `GICD_CTLR`.`ARE=0`, the behavior of SGIs is the same as in GICv2.

8. Virtualization

ARMv8-A includes optional support for virtualization. To complement this, GICv3 also supports virtualization. Support for virtualization support in GICv3 adds:

- Hardware virtualization of the CPU interface registers.
- Virtual interrupts.
- Maintenance interrupts.

NOTE: The GIC architecture does not provide features for virtualizaing the Distributor, Redistributors or ITSs. Virtualization of these interfaces must be handled by software. This is outside the scope of this document and is not described here.

8.1 Terminology

Hypervisors create, control and schedule virtual machines (VM). A virtual machine is functionally equivalent to a physical system, and contains one or more virtual processors. Each of those virtual processors contain one or more virtual PEs (vPEs).

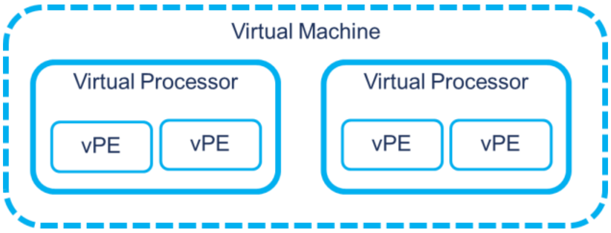


Figure 26 Virtual machine, virtual processor and virtual PE

Most of the controls that are discussed in this chapter work at the level of vPE.

8.2 Interfaces

The CPU Interface registers are split into three groups:

- Physical CPU interface registers.
- Virtualization control registers.
- Virtual CPU interface registers.

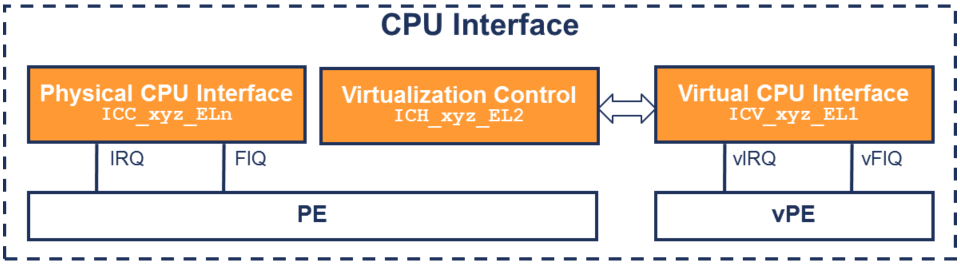


Figure 27 CPU interface registers with virtualization

Physical CPU interface (`ICC_*_ELn`)

The hypervisor software executing at EL2 uses the regular `ICC_*_ELn` registers to handle physical interrupts.

Virtualization Control (`ICH_*_EL2`)

The hypervisor has access to additional registers to control the virtualization features provided by the architecture. These features are:

- Enabling/disabling the virtual CPU interface.
- Accessing virtual register state to enable context switching.
- Configuring maintenance interrupts.
- Controlling virtual interrupts.

These registers control the virtualization features of the physical PE from which they are accessed. It is not possible to access the state of another PE. That is, software on PE X cannot access state for PE Y.

Virtual CPU interface (ICV_*_ELn)

Software executing in a virtualizaed environment uses the ICV_*_EL1 registers to handle interrupts. These registers have the same format and function as the ICC_*_EL1 registers.

The ICV and ICC registers have the same instruction encodings. At EL2, EL3 and Secure EL1, the ICC registers are always accessed. At Non-secure EL1, whether the ICC or the ICV registers are accessed is determined by the routing bits in HCR_EL2.

同样的指令，根据CPU当前EL和HCR_EL2来判断访问ICV或ICC寄存器，所以无需修改虚拟化环境中无需修改指令。

The ICV registers are split into three groups:

- **Group 0**
  Registers used for handling Group 0 interrupts, for example ICC_IAR0_EL1/ICC_IAR0_EL1. When HCR_EL2.FMO==1, ICV registers, instead of ICC registers, are accessed at Non-secure EL1.

- **Group 1**
  Registers used for handling Group 1 interrupts, for example ICC_IAR1_EL1/ICC_IAR1_EL1. When HCR_EL2.IMO==1, ICV registers, instead of ICC registers, are accessed at Non-secure EL1.

- **Common**
  Registers used for handling both Group 0 and 1 interrupts, for example ICC_DIR_EL1/ICV_DIR_EL1 and ICC_PMR_EL1/ICV_PMR_EL1. When either HCR_EL2.IMO==1 or HCR_EL2.FMO==1, ICV registers, instead of ICC registers, are accessed at Non-secure EL1.

Figure 28 shows an example of how the same instruction can access either an ICC or ICV register based on the HCR_EL2 routing controls.

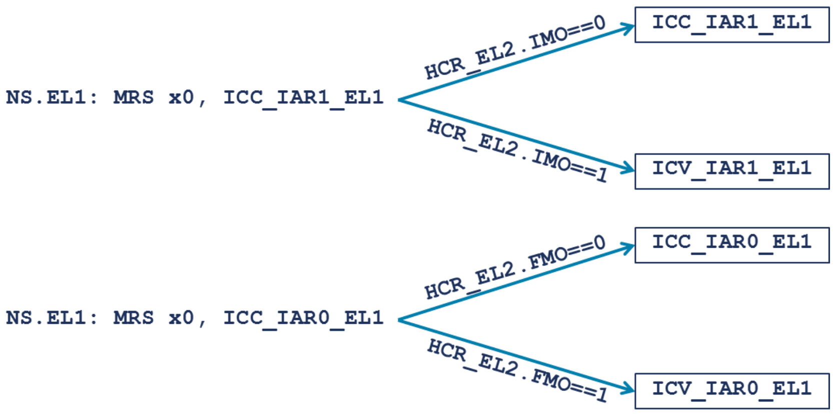


Figure 28 Example of ICC/ICV register selection

VMM可以通过写LRs来产生一个虚拟中断，即向VM中注入一个中断。

8.3 Managing virtual Interrupts

A hypervisor executing at EL2 can generate virtual interrupts using the List registers, ICH_LRn_EL2. Each register represents one virtual interrupt, and records:

- **vINTID** (virtual INTID)
  This is the INTID reported in the virtual environment.

- **State**
  The state (Pending, Active, Active and Pending or Inactive) of the virtual interrupt. The state machine is automatically updated as software in the virtual environment interacts with the GIC. For example, the hypervisor might create a new virtual interrupt, initially setting the state as pending. When software on the vPE reads `ICV_IARn_EL1`, the state is updated to Active.

- **Group**
  The virtual environment always behaves as if `GICD_CTLR.DS==1`. Therefore virtual interrupts can be Group 0 or Group 1. Group 0 interrupts are delivered as vFIQs. Group 1 interrupts are delivered as vIRQs.

- **pINTID** (physical INTID)
  A virtual interrupt can be optionally tagged with the INTID of a physical interrupt. When the state machine of the vINTID is updated, so is that of the pINTID.

8.3.1 Example of a physical interrupt being forwarded to a vPE

Figure 29 shows an example sequence of a physical interrupt that is forwarded to a vPE.

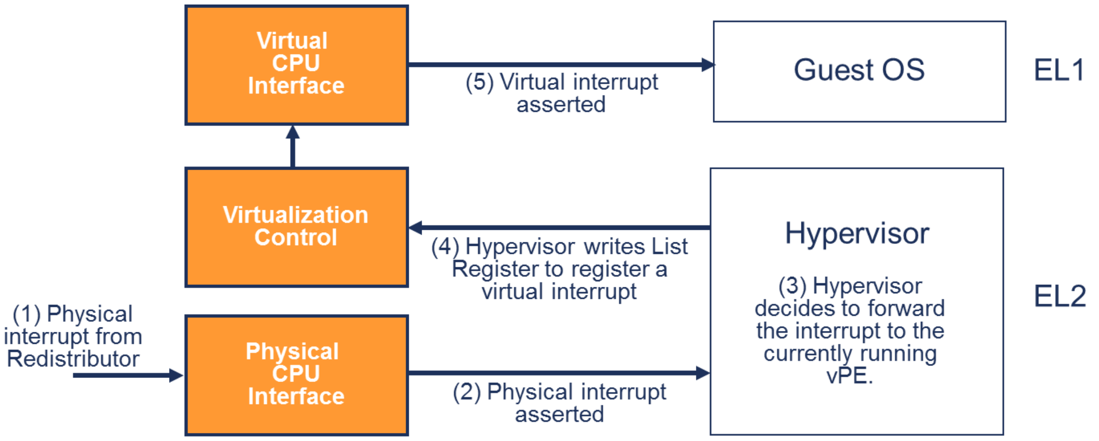

Virtual CPU Interface
(5) Virtual interrupt asserted
Guest OS
EL1
Virtualization Control
(4) Hypervisor writes List Register to register a virtual interrupt
Hypervisor
(3) Hypervisor decides to forward the interrupt to the currently running vPE.
EL2
(1) Physical interrupt from Redistributor
Physical CPU Interface
(2) Physical interrupt asserted

Figure 29 Example of forwarding a physical interrupt to a vPE

1. A physical Non-secure Group 1 interrupt is forwarded to the physical CPU interface from the Redistributor.

2. The physical CPU interface checks whether the physical interrupt can be forwarded to the PE. This process is described in section 5.1. In this instance, the checks pass and a physical exception is asserted.

3. The interrupt is taken to EL2. The hypervisor reads the IAR, which returns the pINTID. The pINTID is now in the Active state. The hypervisor determines that the interrupt is to be forwarded to the currently running vPE. The hypervisor writes the pINTID to `ICC_EOIR1_EL1`. With `ICC_CTLR_EL1.EOImode==1`, this only performs priority drop without deactiving the physical interrupt.

4. The hypervisor writes one of the List register, in order to register a virtual interrupt as pending. The List register entry specifies the vINTID that is to be sent and the original pINTID. The hypervisor then performs an exception return, returning execution to the vPE.

5. The virtual CPU interface checks whether the virtual interrupt can be forwarded to the vPE. These checks are the same as for physical interrupts, other than that they use the ICV registers. In this instance, the checks pass and a virtual exception is asserted.

6. The virtual exception is taken to Non-secure EL1. When software reads the IAR, the vINTID will be returned and the virtual interrupt is now in the Active state.

7. The Guest OS handles the interrupt. When it has finished handling the interrupt, it writes the EOIR to perform a priority drop and deactivation. As the List register recorded the pINTID, this deactivates both the vINTID and pINTID.

8.4 Maintenance interrupts

The CPU interface can be configured to generate physical interrupts if certain conditions are true in the virtual CPU interface.

These interrupts are reported as a PPI, with INTID 25. This interrupt is typically configured as Non-secure Group 1, and handled by the hypervisor software at EL2.
The generation of maintenance interrupts is controlled by ICH_HCR_EL2, and the interrupts that are currently asserted are reported in ICH_MISR_EL2.

Example

A maintenance interrupt can be generated if the vPE clears one of the Group enable bits in the virtual CPU interface. On seeing this, a hypervisor could remove any List register entries for pending virtual interrupts belonging to the disabled group.

8.5 Legacy virtual machines

A hypervisor using the GICv3 system registers (ICC_SRE_EL2.SRE==1) can host VMs that use legacy operation (ICC_SRE_EL1(NS)==0). In this case software running in the virtual environment uses the memory mapped GICV registers, as in GICv2.

8.6 Context switching

When context switching between vPEs, the hypervisor software saves off the state of one vPE and loads the context of another. The state of the Virtual CPU interface forms part of the context of a vPE. The Virtual CPU interface state consists of:

- The state of the ICV registers.
- The active virtual priorities.
- Any pending, active or active and pending virtual interrupts.

The state of the ICV registers can be accessed from EL2 using the ICH registers. As an example, Figure 30 shows how the fields in ICH_VMCR_EL2 map on to the ICV register state.

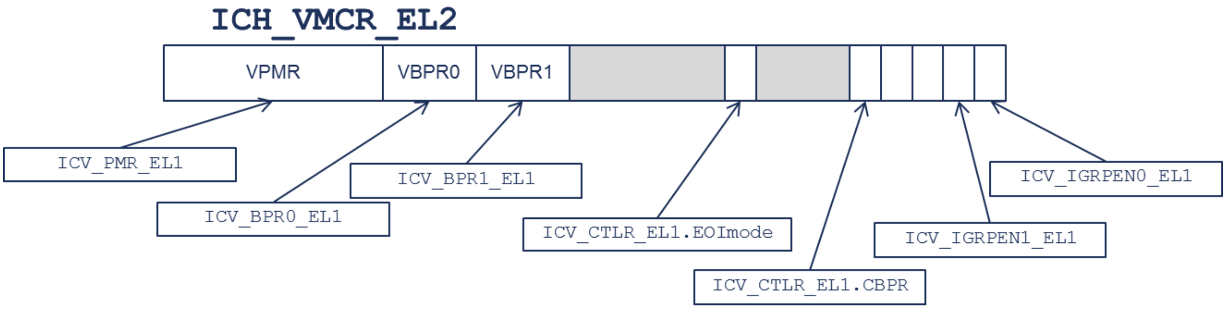


Figure 30 Accessing ICV state from EL2

The active virtual priorties must be saved and restored when switching vPEs. The active priorities for the current vPE can be accessed via the ICV_APnR_EL2 registers.

As described in section 8.3, virtual interrupts are managed via the List registers. The state of these registers are specific to the current vPE. Therefore these registers must be saved and restored on context switches.

9. GICv4: Direct Injection of Virtual LPIs

GICv4 adds support for the direct injection of virtual LPIs (vLPIs). This feature allows software to describe to the ITS how physical events (a combination of an EventID and a DeviceID) map to virtual interrupts. If the vPE targeted by interrupt is running, the virtual interrupt can be forwarded without the need to first enter the hypervisor. This can reduce the overhead associated with virtualized interrupts.

9.1 Redistributors, vLPI state and configuration

To support the direct injection of vLPIs, the Redistributors have two additional registers:

- **`GICR_VPROPBASER`**
  This register sets the address of the virtual LPI Configuration table. As with the physical LPI Configuration table, the virtual LPI Configuration table records the configuration of vLPIs. The configuration of vLPIs is global to all vPEs in the same VM. ARM expects that all vPEs in a VM will use the same copy of the virtual Configuration Table.

- **`GICR_VPENDBASER`**
  This register sets the address of the virtual LPI Pending table (VPT). As with the physical LPI Pending table, the VPT records the pending state of the vLPIs. Each vPE has its own private VPT.

9.1.1 Scheduled virtual PE

Multiple vPEs might be hosted by a single physical PE, with the hypervisor context switching between them. The currently running vPE is referred to to as being **scheduled**. A vPE is defined as being scheduled when `GICR_VPENDBASER` is set to point at its VPT.

Virtual interrupts for the scheduled vPE can be directly injected. If the target vPE is not scheduled, the virtual interrupt is recorded as being pending in the appropriate VPT.

When performing a context swith between vPEs, a hypervisor must update the Redistributor registers. This means that the hypervisor must:

- **Clear `GICR_VPENDBASER.Valid`**
  Clearing the Valid bit informs the Redistributor that a context switch is taking place. The Redistributor will retrieve any pending virtual interrupts from the virtual CPU interface, and ensure that the VPT in memory is correct.

- **Poll `GICR_VPENDBASER.Dirty` until it reads 0**
  The Dirty bit reports that the Redistributor has finished updating the VPT. The new vPE cannot be scheduled until this bit reads 0.

- **Update `GICR_VPROPBASER`**
  If switching between different vPEs of the same VM, this might not be necessary.

- **Update `GICR_VPENDBASER`, setting Valid==1 in the process**
  Setting the Valid bit to 1 informs the Redistributor that the new vPE is now valid, and that virtual interrupts for that vPE can be forwarded to the virtual CPU interface.

The first 1KB of a VPT is IMPLEMENTATION DEFINED. ARM expects an implementation will use this space to record information that makes parsing the VPT quicker on context switches. When a vPE is scheduled, the Redistributor must be informed whether this region contains valid data. Software indicates whether the space is valid or not using `GICR_VPENDBASER.IDAI`:

- **`GICR_VPENDBASER.IDAI==1`** *(invalid)*
  The reserved region is not valid, and the Redistributor must parse the entire VPT. The IDAI bit must be set when:
  - A vPE is moved to a Redistributor that is connected to a different GIC implementation.

- A vPE is made schuled for the first time since the VPT was allocated, and at the time of allocation the entire table was not filled with zeros.

- **`GICR_VPENDBASER.IDAI==0`** *(valid)*
  The reserved region is valid, and the Redistributor can rely on the values stored there. ARM expects this to be the most common case. The IDAI bit can be cleared when:
  - A vPE is scheduled on the same Redistributor on which it was last scheduled.
  - A vPE is scheduled on a different Redistributor, but connected to the same GIC.
  - A vPE is scheduled for the first time since the VPT was allocated, and at the time of allocation the entire VPT was filled with zeros (meaning there are no pending interrupts).
    - NOTE: The restriction is that the VPT contained all zeros at the time of allocation, not that it contains all zeros when it is first scheduled. If ITS mappings for the vPE exist, virtual interrupts might be set pending between creation and first residency.

9.2 Operation of an ITS in GICv4

GICv4 adds a number of new commands, and an additional table type, to the ITS. This allows software to:

- Map an EventID-DeviceID combination to a vINTID for a specific vPE.
  - Optionally a door-bell interrupt can be specified. This is a pINTID that is generated if the vPE is not scheduled when the interrupt is generated.
- Map a vPE to a physical Redistributor.

Figure 31 shows the process that an ITS follows when forwarding a virtual interrupt.

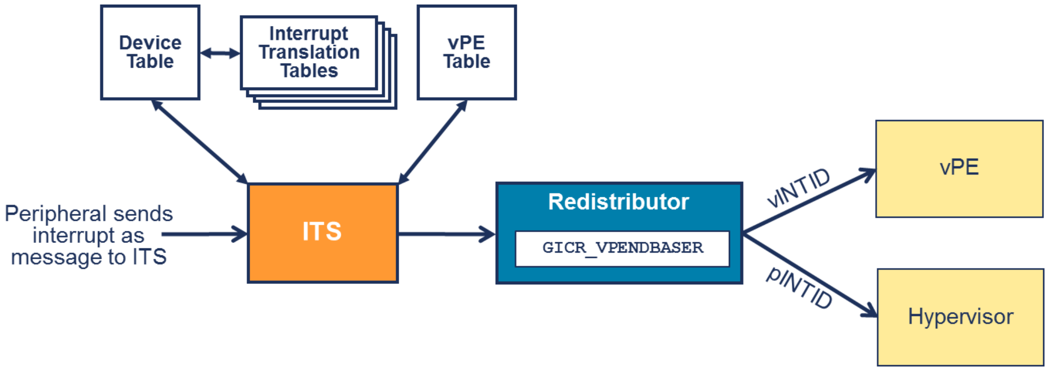


Figure 31 Using an ITS to directly inject virtual interrupts

When a peripheral writes to `GITS_TRANSLATER`:

1. The ITS uses the DeviceID to select the appropriate entry from the Device table. This entry identifies the Interrupt translation table to use.

2. The ITS uses the EventID to select the appropriate entry from the Interrupt translation table. This will return either:

    a. A pINTID and Collection ID, as described in section 6.1.1.

    b. A vINTID and vPE ID, and optionally a pINTID as a door-bell interrupt.

3. The ITS uses the vPE ID to select the required entry in the vPE table and the vPE table returns the target Redistributor and the address of the VPT of the vPE.

4. The ITS forwards the vINTID, door-bell interrupt and VPT address to the target Redistributor.

5. The Redistributor compares the VPT address from the ITS against the current `GICR_VPENDBASER`:

    a. If the VPT address and current `GICR_VPENDBASER` match, the vPE is scheduled, and the vINTID is forwarded to the virtual CPU interface.

    b. If the VPT address and current `GICR_VPENDBASER` do not match, the vPE is not scheduled. The vINTID is set as pending in the VPT. If a door-bell interrupt was provided, the pINTID is forwarded to the physical CPU interface.

9.3 Mapping a vPE and vINTID

EventID DeviceID combinatons are mapped to a vINTID and vPE. The `VMAPI` command is used when the EventID and vINTID are the same.

```
VMAPI <DeviceID>, <EventID>, <Doorbell pINTID>, <vPE ID>
```

The `VMAPTI` command is used when the EventID and vINTID are different.

```
VMAPTI <DeviceID>, <EventID>, <vINTID>, <pINTID>, <vPE ID>
```

In these commands:

- <DeviceID> and <EventID> together identify the interrupt that is being remapped.
- <vPE ID> is the ID of the vPE. For systems that contain multiple ITSs, the same vPE ID must be assigned to a given vPE on all ITSs.
- <pINTID> is the doorbell interrupt that must be generated if the vPE is not scheduled. Specifying 1023 means that there is no door-bell interrupt.
- <vINTID> is the INTID of the virtual LPI. For `VMAPI`, EventID and vINTID have the same value.

The ITS must be aware of which physical PE a vPE will be scheduled on when it is running. The VMAPP command maps a vPE to a physical Redistributor:

```
VMAPP  <vPE ID>, <RDADDR>, <VPT>, <VPT size>
```

In this command:

- <vPE ID> is the the ID of the vPE.
- <RDADDR> is the target Redistributor.
- <VPT> and <VPT size> identify the virtual LPI Pending table of the vPE. As described in section 9.1.1, a vPE is scheduled when GICR_VPENDBASER points at its VPT. When forwarding a virtual interrupt to a Redistributor, the ITS includes the VPT address. This allows the Redistributor to check whether the vPE is scheduled on the PE, and if it is not scheduled, to update the VPT so that the interrupt is not lost.

Example

A timer has DeviceID 5. It generates two EventIDs, 0 and 1. Both EventIDs are mapped to vINTIDs that belong to the vPE with vPE ID 6:

- EventID 0 – vINTID 8725, door-bell pINTID 8192
- EventID 1 – vINTID 9000, no door-bell interrupt

vPE 6 is mapped to the Redistributor at address 0x78400000, and its VPT is at address 0x97500000.

The command sequence for this is:

```
VMAPTI 5, 0, 8725, 8192, 6
VMAPTI 5, 1, 9000, 1023, 6
VMAPP  6, 0x78400000, 0x97500000, 12
VSYNC  6
```

NOTE: The example assumes that GITS_TYPER.PTA==1, and that a MAPD command has previously been issued to map the ITT.

9.4 Mapping a vPE to a different Redistributor

If a hypervisor maps a vPE to a different physical PE, the ITS mappings must be updated so that virtual interrupts are sent to the correct physical PE. The ITS mappings are updated using the VMOVP command, followed by VSYNC to synchronize the context.

A system can include multiple ITSs. Where more than one ITS has mappings for a vPE, any change must be applied to all ITSs that contain the original mappings. GICv4 supports two models for doing this, and GITS_TYPER.VMOVP indicates which model is used.

GITS_TYPER.VMOVP==0

The VMOVP command must be issued on **all** ITSs with a mapping for the vPE.

```
VMOVP <vPE ID>, <RDADDR>, <ITS List>, <Sequence Number>
```

In this command:

- <vPE ID> is the ID of the vPE.
- <RDADDR> is the Redistributor that the vPE is being remapped to.
- <ITS List> is a list of all the ITSs with mappings for the vPE. This field is encoded as one per-bit ITS, where bit 0 maps to ITS 0. The number of an ITS is reported by GITS_CTLR.ITS_Number.

- <Sequence Number> is the synchronization point. Software must use the same value when issuing the `VMOVP` to the different ITSs, and must not re-use the same value until the commands have completed on all ITSs.

GITS_TYPER.VMOVP==1

The `VMOVP` command must be issued on **only one** ITS, regardless of how many ITSs have mappings for the vPE. The hardware is required to propagate the change and handle synchronization. This means that the ITS List and SequenceNumber fields are not required.

```
VMOVP <vPE ID>, <RDADDR>
```

9.5 Remapping or removing the mapping of vPEs/vINTIDs

`VMOVI` remaps an EventID DeviceID combination to a different vINTID or vPE.

```
VMOVI <DeviceID>, <EventID>, <vPE ID>, <Doorbell pINTID>
```

In this command:

- <DeviceID> and <EventID> together identify the interrupt that is being remapped.
- <vPE ID> is the ID of the vPE that the interrupt is being moved to.
- <Doorbell pINTID> is the Redistributor that the vPE is being remapped to.

9.6 Changing vLPI configuration

As with physical LPIs, a Redistributor is permitted to cache the configuration of vLPIs. If the configuration of a vLPI is changed, the cached copy must be invalidated. There are two ITS commands available to do this.

The `INV` command is typically used when changing the configuration of a single, or small number, of vLPIs. A separate `INV` is required for each vLPI that is modified.

The `VINVALL` command invalidates the configuration of all vLPIs that belong to a specified vPE. This command is typically used when modifying a large number of vLPIs.

9.7 Mixing GICv3 and GICv4

A mixture of GICv3 and GICv4 capable CPU interfaces might be connected to a single GIC. Figure 32 shows an example of such a system.

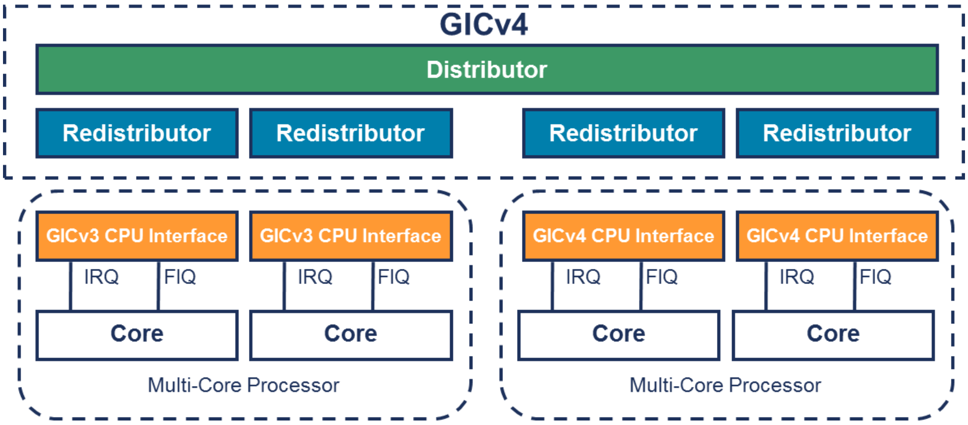


Figure 32 GICv4 interrupt controller, with GICv3 CPU interfaces attached

Only CPU interfaces that implement GICv4 are capable of receiving directly injected virtual LPIs. Scheduling a vPE, that is setting `GICR_VPENDBASER.Valid==1`, on a GICv3 CPU interface is UNPREDICTABLE. Software can determine whether directly injected virtual interrupts are supported by reading `ICH_VTR_EL2.nV4`.