

# ARM® Generic Interrupt Controller

Architecture version 2.0

## Architecture Specification



# ARM Generic Interrupt Controller

Copyright © 2008, 2011, 2013 ARM. All rights reserved.

## Release Information

The following changes have been made to this document.

### Change History

Date	Issue	Confidentiality	Change
23 September 2008	A	Non-Confidential	First release for version 1.0
13 June 2011	B	Non-Confidential	First release for version 2.0
26 July 2013	B.b	Non-Confidential	Re-release of issue B with new Proprietary Notice

### Status of Issue B.b of this document

Issue B.b of this document is a re-issue of issue B incorporating the updated Proprietary Notice for the document. Beyond page four of the document the only changes between issue B and issue B.b are:

- Changes to the page footers to show the new version number, copyright dates, and ID code.
- Changed page numbering, because of the longer Proprietary Notice.
- A statement in [Appendix C Revisions](#) that there are no technical changes between issue B and issue B.b.

### Proprietary Notice

#### ARM GENERIC INTERRUPT CONTROLLER (GIC) ARCHITECTURE SPECIFICATION LICENCE

THIS END USER LICENCE AGREEMENT ("LICENCE") IS A LEGAL AGREEMENT BETWEEN YOU (EITHER A SINGLE INDIVIDUAL, OR SINGLE LEGAL ENTITY) AND ARM LIMITED ("ARM") FOR THE USE OF THE RELEVANT GIC ARCHITECTURE SPECIFICATION ACCOMPANYING THIS LICENCE. ARM IS ONLY WILLING TO LICENSE THE RELEVANT GIC ARCHITECTURE SPECIFICATION TO YOU ON CONDITION THAT YOU ACCEPT ALL OF THE TERMS IN THIS LICENCE. BY CLICKING "I AGREE" OR OTHERWISE USING OR COPYING THE RELEVANT GIC ARCHITECTURE SPECIFICATION YOU INDICATE THAT YOU AGREE TO BE BOUND BY ALL THE TERMS OF THIS LICENCE. IF YOU DO NOT AGREE TO THE TERMS OF THIS LICENCE, ARM IS UNWILLING TO LICENSE THE RELEVANT GIC ARCHITECTURE SPECIFICATION TO YOU AND YOU MAY NOT USE OR COPY THE RELEVANT GIC ARCHITECTURE SPECIFICATION AND YOU SHOULD PROMPTLY RETURN THE RELEVANT GIC ARCHITECTURE SPECIFICATION TO ARM.

"LICENSEE" means You and your Subsidiaries.

"Subsidiary" means, if You are a single entity, any company the majority of whose voting shares is now or hereafter owned or controlled, directly or indirectly, by You. A company shall be a Subsidiary only for the period during which such control exists.

1. Subject to the provisions of [Clauses 2, 3 and 4](#), ARM hereby grants to LICENSEE a perpetual, non-exclusive, non-transferable, royalty free, worldwide licence to:
  - a. use and copy the relevant GIC Architecture Specification for the purpose of developing and having developed products that comply with the relevant GIC Architecture Specification;
  - b. manufacture and have manufactured products which either: (i) have been created by or for LICENSEE under the licence granted in [Clause 1a](#); or (ii) incorporate a product(s) which has been created by a third party(s) under a licence granted by ARM in [Clause 1a](#) of such third party's ARM GIC Architecture Specification Licence; and
  - c. offer to sell, sell, supply or otherwise distribute products which have either been (i) created by or for LICENSEE under the licence granted in [Clause 1a](#); or (ii) manufactured by or for LICENSEE under the licence granted in [Clause 1b](#).
2. LICENSEE hereby agrees that the licence granted in [Clause 1](#) is subject to the following restrictions:
  - a. where a product is created under [Clause 1a](#) or manufactured under [Clause 1b](#) it must contain at least one processor core which has either been (i) developed by or for ARM; or (ii) developed under licence from ARM;
  - b. the licences granted in [Clause 1c](#) shall not extend to any portion or function of a product that is not itself compliant with part of the relevant GIC Architecture Specification; and
  - c. no right is granted to LICENSEE to sublicense the rights granted to LICENSEE under this Agreement.

3. Except as specifically licensed in accordance with Clause 1, LICENSEE acquires no right, title or interest in any ARM technology or any intellectual property embodied therein. In no event shall the licences granted in accordance with Clause 1 be construed as granting LICENSEE, expressly or by implication, estoppel or otherwise, a licence to use any ARM technology except the relevant GIC Architecture Specification.
4. THE RELEVANT GIC ARCHITECTURE SPECIFICATION IS PROVIDED "AS IS" WITH NO WARRANTIES EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED TO ANY WARRANTY OF SATISFACTORY QUALITY, MERCHANTABILITY, NONINFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE.
5. No licence, express, implied or otherwise, is granted to LICENSEE, under the provisions of Clause 1, to use the ARM tradename in connection with the relevant GIC Architecture Specification or any products based thereon. Nothing in Clause 1 shall be construed as authority for LICENSEE to make any representations on behalf of ARM in respect of the relevant GIC Architecture Specification.
6. This Licence shall remain in force until terminated by you or by ARM. Without prejudice to any of its other rights if LICENSEE is in breach of any of the terms and conditions of this Licence then ARM may terminate this Licence immediately upon giving written notice to You. You may terminate this Licence at any time. Upon expiry or termination of this Licence by You or by ARM LICENSEE shall stop using the relevant GIC Architecture Specification and destroy all copies of the relevant GIC Architecture Specification in your possession together with all documentation and related materials. Upon expiry or termination of this Licence, the provisions of clauses 6 and 7 shall survive.
7. The validity, construction and performance of this Agreement shall be governed by English Law.

ARM contract references: LES-PRE-20079 ARM GENERIC INTERRUPT CONTROLLER (GIC) ARCHITECTURE Specification Licence.

Where the term ARM is used it means "ARM or any of its subsidiaries as appropriate".

———— **Note** ————

The term ARM can refer to versions of the ARM architecture, for example ARMv6 refers to version 6 of the ARM architecture. The context makes it clear when the term is used in this way.

—————



# Contents

## ARM Generic Interrupt Controller Architecture Specification

	<b>Preface</b>	
	About this specification .....	viii
	Using this specification .....	ix
	Conventions .....	x
	Additional reading .....	xi
	Feedback .....	xii
<b>Chapter 1</b>	<b>Introduction</b>	
	1.1 About the Generic Interrupt Controller architecture .....	1-14
	1.2 Security Extensions support .....	1-16
	1.3 Virtualization support .....	1-17
	1.4 Terminology .....	1-18
<b>Chapter 2</b>	<b>GIC Partitioning</b>	
	2.1 About GIC partitioning .....	2-22
	2.2 The Distributor .....	2-24
	2.3 CPU interfaces .....	2-26
<b>Chapter 3</b>	<b>Interrupt Handling and Prioritization</b>	
	3.1 About interrupt handling and prioritization .....	3-34
	3.2 General handling of interrupts .....	3-37
	3.3 Interrupt prioritization .....	3-44
	3.4 The effect of interrupt grouping on interrupt handling .....	3-48
	3.5 Interrupt grouping and interrupt prioritization .....	3-53
	3.6 Additional features of the GIC Security Extensions .....	3-59
	3.7 Pseudocode details of interrupt handling and prioritization .....	3-61

3.8	The effect of the Virtualization Extensions on interrupt handling .....	3-67
3.9	Example GIC usage models .....	3-68

**Chapter 4**

**Programmers' Model**

4.1	About the programmers' model .....	4-74
4.2	Effect of the GIC Security Extensions on the programmers' model .....	4-80
4.3	Distributor register descriptions .....	4-84
4.4	CPU interface register descriptions .....	4-124
4.5	Preserving and restoring GIC state .....	4-155

**Chapter 5**

**GIC Support for Virtualization**

5.1	About implementing a GIC in a system with processor virtualization .....	5-158
5.2	Managing the GIC virtual CPU interface .....	5-160
5.3	GIC virtual interface control registers .....	5-167
5.4	The virtual CPU interface .....	5-178
5.5	GIC virtual CPU interface registers .....	5-179

**Appendix A**

**Pseudocode Index**

A.1	Index of pseudocode functions .....	A-198
-----	-------------------------------------	-------

**Appendix B**

**Register Names**

B.1	Alternative register names .....	B-202
B.2	Register name aliases .....	B-203
B.3	Index of architectural names .....	B-204

**Appendix C**

**Revisions**

**Glossary**

# Preface

This preface introduces the *ARM® Generic Interrupt Controller Architecture Specification*. It contains the following sections:

- *About this specification* on page viii
- *Using this specification* on page ix
- *Conventions* on page x
- *Additional reading* on page xi
- *Feedback* on page xii.

## About this specification

This specification describes the *ARM Generic Interrupt Controller (GIC)* architecture.

Throughout this document, references to *the GIC* or *a GIC* refer to a device that implements this GIC architecture. Unless the context makes it clear that a reference is to an IMPLEMENTATION DEFINED feature of the device, these references describe the requirements of this specification.

## Intended audience

The specification is written for users that want to design, implement, or program the GIC in a range of ARM-compliant implementations from simple uniprocessor implementations to complex multiprocessor systems.

The specification assumes that users have some experience of ARM products. It does not assume experience of the GIC.



## Using this specification

This specification is organized into the following chapters:

### **Chapter 1** *Introduction*

Read this for an overview of the GIC, and information about the terminology used in this document.

### **Chapter 2** *GIC Partitioning*

Read this for a description of the major interfaces and components of the GIC. The chapter also introduces how they operate, in a simple implementation.

### **Chapter 3** *Interrupt Handling and Prioritization*

Read this for a description of the requirements for interrupt handling, and the interrupt priority scheme for a GIC.

### **Chapter 4** *Programmers' Model*

Read this for a description of the Distributor and CPU interface registers.

### **Chapter 5** *GIC Support for Virtualization*

Read this for a description of how the GIC Virtualization Extensions support the implementation of a GIC in a multiprocessor system that supports processor virtualization. This chapter includes a description of the programmers' model for the virtual interface control and virtual CPU interface registers.

### **Appendix A** *Pseudocode Index*

Read this for an index to the pseudocode functions defined in this specification.

### **Appendix B** *Register Names*

Read this for a description of the differences in the register names in earlier descriptions of the GIC architecture, and for an alphabetic index of the register names.

### **Appendix C** *Revisions*

Read this for a description of the technical changes between released issues of this book.

### *Glossary*

Read this for definitions of some terms used in this book.

## Conventions

The following sections describe conventions that this book can use:

- *General typographic conventions*
- *Signals*
- *Numbers*
- *Pseudocode descriptions.*

### General typographic conventions

The typographical conventions are:

<i>italic</i>	Introduces special terminology, denotes internal cross-references and citations, or highlights an important note.
<b>bold</b>	Denotes signal names, and is used for terms in descriptive lists, where appropriate.
monospace	Used for assembler syntax descriptions, pseudocode, and source code examples. Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.
<b>SMALL CAPITALS</b>	Used for a few terms that have specific technical meanings, and are included in the <a href="#">Glossary</a> .
<b>Colored text</b>	Indicates a link. This can be: <ul style="list-style-type: none"><li>• a URL, for example, <a href="http://infocenter.arm.com">http://infocenter.arm.com</a></li><li>• a cross-reference, that includes the page number of the referenced information if it is not on the current page, for example, <a href="#">Distributor Control Register, GICD_CTLR on page 4-85</a></li><li>• a link, to a chapter or appendix, or to a glossary entry, or to the section of the document that defines the colored term, for example <a href="#">Banked register</a> or <a href="#">GICD_CTLR</a>.</li></ul>

### Signals

In general this specification does not define processor signals, but it does include some signal examples and recommendations. The signal conventions are:

<b>Signal level</b>	The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means: <ul style="list-style-type: none"><li>• HIGH for active-HIGH signals</li><li>• LOW for active-LOW signals.</li></ul>
<b>Lower-case n</b>	At the start or end of a signal name denotes an active-LOW signal.

### Numbers

Numbers are normally written in decimal. Binary numbers are preceded by `0b`, and hexadecimal numbers by `0x`. In both cases, the prefix and the associated value are written in a monospace font, for example `0xFFFF0000`.

### Pseudocode descriptions

This specification uses a form of pseudocode to provide precise descriptions of the specified functionality. This pseudocode is written in a monospace font, and follows the conventions described in the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

## Additional reading

This section lists relevant publications from ARM and third parties.

See the Infocenter, <http://infocenter.arm.com>, for access to ARM documentation.

### ARM publications

- *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* (ARM DDI 0406), issue C or later.

### Other publications

The following books are referred to in this manual, or provide more information:

- JEDEC Solid State Technology Association, *Standard Manufacture's Identification Code*, JEP106.

## Feedback

ARM welcomes feedback on its documentation.

### Feedback on this specification

If you have comments on the content of this specification, send e-mail to [errata@arm.com](mailto:errata@arm.com). Give:

- the title
- the number, ARM IHI 0048B.b
- the page numbers to which your comments apply
- a concise explanation of your comments.

ARM also welcomes general suggestions for additions and improvements.

# Chapter 1

## Introduction

This chapter gives an overview of the GIC and information about the terminology used in this document. It contains the following sections:

- *About the Generic Interrupt Controller architecture on page 1-14*
- *Security Extensions support on page 1-16*
- *Virtualization support on page 1-17*
- *Terminology on page 1-18.*

## 1.1 About the Generic Interrupt Controller architecture

The *Generic Interrupt Controller* (GIC) architecture defines:

- the architectural requirements for handling all interrupt sources for any processor connected to a GIC
- a common interrupt controller programming interface applicable to uniprocessor or multiprocessor systems.

---

### Note

---

The architecture describes a GIC designed for use with one or more processors that comply with the ARM A and R architecture profiles. However the GIC architecture does not place any restrictions on the processors used with an implementation of the GIC.

---

The GIC is a centralized resource for supporting and managing interrupts in a system that includes at least one processor. It provides:

- registers for managing interrupt sources, interrupt behavior, and interrupt routing to one or more processors
- support for:
  - the ARM architecture Security Extensions
  - the ARM architecture Virtualization Extensions
  - enabling, disabling, and generating processor interrupts from hardware (peripheral) interrupt sources
  - Software-generated Interrupts (SGIs)
  - interrupt masking and prioritization
  - uniprocessor and multiprocessor environments
  - wakeup events in power-management environments.

The GIC includes *interrupt grouping* functionality that supports:

- configuring each interrupt as either Group 0 or Group 1
- signaling Group 0 interrupts to the target processor using either the IRQ or the FIQ exception request
- signaling Group 1 interrupts to the target processor using the IRQ exception request only
- a unified scheme for handling the priority of Group 0 and Group 1 interrupts
- optional lockdown of the configuration of some Group 0 interrupts.

---

### Note

---

- Interrupt grouping is present in all GICv2 implementations and in GICv1 implementations that include the GIC Security Extensions, see [Changes in version 2.0 of the Specification on page 1-15](#).
  - In many implementations the IRQ and FIQ interrupt requests correspond to the IRQ and FIQ asynchronous exceptions that are supported by all variants of the ARM architecture except the *Microcontroller profile* (M-profile). For more information about IRQ, FIQ, and asynchronous exceptions, see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.
- 

### 1.1.1 GIC architecture specification version

This specification defines version 2.0 of the GIC architecture (GICv2), and also describes version 1.0 of the architecture (GICv1).

The GIC architecture specification version is independent of the *mpn* version, or major and minor revision description, used for ARM product releases.

## 1.1.2 Changes in version 2.0 of the Specification

Version 2.0 of the Architecture Specification contains the following changes and additions to version 1.0:

1. The addition of the optional GIC Virtualization Extensions, that support the implementation of the GIC in a system that supports processor virtualization. For more information, see [Virtualization support on page 1-17](#).
2. A change to the architectural status of *interrupt grouping*. Interrupt grouping, and the ability to use FIQs to signal Group 0 interrupts, are provided:
  - in all GICv2 implementations
  - only as part of the optional Security Extensions in GICv1 implementations.

———— **Note** —————

In version 1.0 of the Specification, interrupt grouping is presented only as the classification of interrupts as Secure or Non-secure, see item 7 of this list.

3. The addition of wakeup event support in power management environments. For more information, see [Power management, GIC v2 on page 2-31](#).
4. The addition of support for the save and restore of all GIC state, for power-down, or context switching, including virtual machine context switching in a system that supports virtualization. This means that some state that is read-only in GICv1 becomes read/write in GICv2. For more information, see [Preserving and restoring GIC state on page 4-155](#).
5. The addition of an option to split interrupt completion into two stages, *Priority drop* and *interrupt deactivation*. For more information, see [Priority drop and interrupt deactivation on page 3-38](#).
6. The addition of controls to disable the forwarding of legacy interrupt signals to a connected processor when forwarding of interrupts from the GIC to that processor is also disabled. For more information see [Interrupt signal bypass, and GICv2 bypass disable on page 2-27](#).
7. Changes to the terminology used to describe the interrupt grouping features of the GICv1 Security Extensions, to clarify that these features can be used to implement functionality that is unrelated to the scope of the ARM Security Extensions present on an ARM processor.

———— **Note** —————

As indicated in item 2, these features of the GICv1 Security Extensions are included in all GICv2 implementations. That is, in GICv2 they are not part of the optional Security Extensions.

The terminology change includes renaming the Interrupt Security Registers to Interrupt Group Registers. These registers separate interrupts into two groups, Group 0 and Group 1. In specific contexts, typically when a GIC that implements the GIC Security Extensions is connected to an ARM processor that implements the processor Security Extensions, Group 0 interrupts are Secure interrupts and Group 1 interrupts are Non-secure interrupts. For more information, see [Security Extensions support on page 1-16](#).

## 1.2 Security Extensions support

The ARM processor Security Extensions are an optional extension to the ARMv7-A architecture profile. This means it is IMPLEMENTATION DEFINED whether an ARMv7-A implementation includes the Security Extensions. The ARM Security Extensions facilitate the development of secure applications by:

- integrating hardware security features into the architecture
- providing Secure virtual memory space that is accessed by memory accesses in the Secure state
- providing Non-secure virtual memory space that is accessed by memory accesses in the Non-secure state.

See [Processor security state and Secure and Non-secure GIC accesses on page 1-20](#) for more information.

When a GIC that implements the GIC Security Extensions is connected to a processor that implements the ARM Security Extensions:

- Group 0 interrupts are Secure interrupts, and Group 1 interrupts are Non-secure interrupts.
- The behavior of processor accesses to registers in the GIC depends on whether the access is Secure or Non-secure, see [Processor security state and Secure and Non-secure GIC accesses on page 1-20](#).

Except where this document explicitly indicates otherwise, when accessing GIC registers:

- a Non-secure read of a register field holding state information for a Secure interrupt returns zero
- the GIC ignores any Non-secure write to a register field holding state information for a Secure interrupt.

Non-secure accesses can only read or write information corresponding to Non-secure interrupts. Secure accesses can read or write information corresponding to both Non-secure and Secure interrupts.

- Secure system software individually defines each implemented interrupt as either Secure or Non-secure.
- A Non-secure interrupt signals an IRQ interrupt request to a target processor.
- A Secure interrupt can signal either an IRQ or an FIQ interrupt request to a target processor.
- Secure software can manage interrupt sources securely without the possibility of interference from Non-secure software. See [Controlling Secure and Non-secure interrupts independently on page 3-69](#) for more information.

Secure systems are backwards-compatible with software written for systems without the Security Extensions. See [Supporting IRQs and FIQs when not using the processor Security Extensions on page 3-70](#) for more information.



## 1.3 Virtualization support

The ARM processor Virtualization Extensions are optional extensions to the ARMv7-A architecture profile. This means it is IMPLEMENTATION DEFINED whether an ARMv7-A implementation includes the Virtualization Extensions.

The processor Virtualization Extensions provide hardware support for virtualizing the Non-secure state of an VMSAv7 implementation. The extensions support system use of a virtual machine monitor, known as the hypervisor, to switch guest operating systems.

Whether implemented in a uniprocessor or in a multiprocessor system, the processor Virtualization Extensions support running multiple virtual machines on a single processor.

Interrupt handling is a major consideration in a virtualization implementation. The hypervisor can either handle a physical interrupt itself, or generate a corresponding virtual interrupt that is signaled to a virtual machine. It is also possible for the hypervisor to generate virtual interrupts that do not correspond to physical interrupts.

GICv2 extends the GIC architecture to include the GIC Virtualization Extensions. These extensions support the handling of virtual interrupts, in addition to physical interrupts, in a system that supports processor virtualization. An example of such a system is one where a GIC is integrated with processors that implement the ARM processor Virtualization Extensions. The GIC Virtualization Extensions provide mechanisms to minimize the hypervisor overhead of routing interrupts to virtual machines. See [Chapter 5 GIC Support for Virtualization](#) for more information.

———— **Note** —————

- A processor that implements the ARM Virtualization Extensions must also implement the ARM Security Extensions.
  - A GIC that implements the GIC Virtualization Extensions is not required to implement the GIC Security Extensions.
-

## 1.4 Terminology

The following sections define architectural terms used in this specification:

- [Interrupt states](#)
- [Interrupt types](#)
- [Models for handling interrupts on page 1-19](#)
- [Spurious interrupts on page 1-20](#)
- [Processor security state and Secure and Non-secure GIC accesses on page 1-20](#)
- [Banking on page 1-20](#).

See also [GIC register names on page 4-74](#).

### 1.4.1 Interrupt states

The following states apply at each interface between the GIC and a connected processor:

<b>Inactive</b>	An interrupt that is not active or pending.
<b>Pending</b>	An interrupt from a source to the GIC that is recognized as asserted in hardware, or generated by software, and is waiting to be serviced by a target processor.
<b>Active</b>	An interrupt from a source to the GIC that has been acknowledged by a processor, and is being serviced but has not completed.
<b>Active and pending</b>	A processor is servicing the interrupt and the GIC has a pending interrupt from the same source.

### 1.4.2 Interrupt types

A device that implements this GIC architecture can manage the following types of interrupt:

**Peripheral interrupt** This is an interrupt asserted by a signal to the GIC. The GIC architecture defines the following types of peripheral interrupt:

**Private Peripheral Interrupt (PPI)**

This is a peripheral interrupt that is specific to a single processor.

**Shared Peripheral Interrupt (SPI)**

This is a peripheral interrupt that the Distributor can route to any of a specified combination of processors.

Each peripheral interrupt is either:

**Edge-triggered**

This is an interrupt that is asserted on detection of a rising edge of an interrupt signal and then, regardless of the state of the signal, remains asserted until it is cleared by the conditions defined by this specification.

**Level-sensitive**

This is an interrupt that is asserted whenever the interrupt signal level is active, and deasserted whenever the level is not active.

———— **Note** —————

While a level-sensitive interrupt is asserted its state in the GIC is pending, or active and pending. If the peripheral deasserts the interrupt signal for any reason the GIC removes the pending state from the interrupt. For more information see [Interrupt handling state machine on page 3-41](#).

**Software-generated interrupt (SGI)**

This is an interrupt generated by software writing to a [GICD\\_SGIR](#) register in the GIC. The system uses SGIs for interprocessor communication.

An SGI has edge-triggered properties. The software triggering of the interrupt is equivalent to the edge transition of the interrupt request signal.

When an SGI occurs in a multiprocessor implementation, the CPUID field in the Interrupt Acknowledge Register, [GICC\\_IAR](#), or the Aliased Interrupt Acknowledge Register, [GICC\\_AIAR](#), identifies the processor that requested the interrupt.

In an implementation that includes the GIC Virtualization Extensions:

- when an SGI occurs, management registers in the GIC virtualization Extensions enable the requesting processor to be reported to the Guest OS, as required by the GIC specifications
- by writing to the management registers in the GIC Virtualization Extensions, a hypervisor can generate a virtual interrupt that appears to a virtual machine as an SGI.

See [Software-generated interrupts on page 5-165](#) and [List Registers, GICH\\_LRn on page 5-176](#) for more information.

**Virtual interrupt**

In a GIC that implements the GIC Virtualization Extensions, an interrupt that targets a virtual machine running on a processor, and is typically signaled to the processor by the connected virtual CPU interface. For more information, see [About GIC partitioning on page 2-22](#).

**Maintenance interrupt**

In a GIC that implements the GIC Virtualization Extensions, a level-sensitive interrupt that is used to signal key events, such as a particular group of interrupts becoming enabled or disabled. See [Maintenance interrupts on page 5-164](#) for more information.

**1.4.3 Models for handling interrupts****———— Note ————**

When describing the GIC interrupt handling models, the terms 1-N and N-N do not correspond to the mathematical uses of the terms 1:N and N:N.

In a multiprocessor implementation, there are two models for handling interrupts:

**1-N model** Only one processor handles this interrupt. The system must implement a mechanism to determine which processor handles an interrupt that is programmed to target more than one processor.

**———— Note ————**

- The ARM GIC architecture does not guarantee that a 1-N interrupt is presented to:
  - all processors listed in the target processor list
  - an enabled interface, where at least one interface is enabled.
- A 1-N interrupt might be presented to an interface where the processor has masked the interrupt event, see [Implications of the 1-N model on page 3-41](#).

**N-N model** All processors receive the interrupt independently. When a processor acknowledges the interrupt, the interrupt pending state is cleared only for that processor. The interrupt remains pending for the other processors.

See [Handling different interrupt types in a multiprocessor system on page 3-35](#) for more information.

#### 1.4.4 Spurious interrupts

It is possible that an interrupt that the GIC has signaled to a processor is no longer required. If this happens, when the processor acknowledges the interrupt, the GIC returns a special Interrupt ID that identifies the interrupt as a *spurious interrupt*. Example reasons for spurious interrupts are:

- prior to the processor acknowledging an interrupt:
  - software changes the priority of the interrupt
  - software disables the interrupt
  - software changes the processor that the interrupt targets
- for a 1-N interrupt, another target processor has previously acknowledged that interrupt.

#### 1.4.5 Processor security state and Secure and Non-secure GIC accesses

A processor that implements the ARM Security Extensions has a security state, either Secure or Non-secure:

- a processor in Non-secure state can make only Non-secure accesses to a GIC
- a processor in Secure state can make both Secure and Non-secure accesses to a GIC
- software running in Non-secure state is described as Non-secure software
- software running in Secure state is described as Secure software.

For more information about the implementation of the Security Extensions on a processor see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

A multiprocessor system with a GIC that implements the Security Extensions might include one or more processors that do not implement the Security Extensions. Such a processor is implemented so that either:

- it makes only Secure accesses to the GIC, meaning any software running on the processor is Secure software that can only make Secure accesses to the GIC
- it makes only Non-secure accesses to the GIC, meaning any software running on the processor is Non-secure software.

#### 1.4.6 Banking

*Banking* has a special meaning in ARM architectural specifications:

##### Interrupt banking

In a multiprocessor implementation, for PPIs and SGIs, the GIC can have multiple interrupts with the same interrupt ID. Such an interrupt is called a *banked interrupt*, and is identified uniquely by the combination of its interrupt ID and its associated CPU interface. For more information see [Interrupt IDs on page 2-24](#).

##### Register banking

Register banking refers to implementing multiple copies of a register at the same address. This occurs:

- in a multiprocessor implementation, to provide separate copies for each processor of registers corresponding to banked interrupts
- in a GIC that implements the Security Extensions, to provide separate Secure and Non-secure copies of some registers.

For more information see [Register banking on page 4-77](#).

# Chapter 2

## GIC Partitioning

This chapter describes the architectural partitioning of the major GIC interfaces and components, and introduces the functionality of the major GIC components, the *Distributor* and the *CPU interfaces*. It contains the following sections:

- [About GIC partitioning on page 2-22](#)
- [The Distributor on page 2-24](#)
- [CPU interfaces on page 2-26.](#)

## 2.1 About GIC partitioning

The GIC architecture splits logically into a Distributor block and one or more CPU interface blocks. The GIC Virtualization Extensions add one or more virtual CPU interfaces to the GIC. Therefore, as [Figure 2-1 on page 2-23](#) shows, the logical partitioning of the GIC is as follows:

**Distributor** The Distributor block performs interrupt prioritization and distribution to the CPU interface blocks that connect to the processors in the system.

The Distributor block registers are identified by the GICD\_ prefix.

**CPU interfaces** Each CPU interface block performs priority masking and preemption handling for a connected processor in the system.

CPU interface block registers are identified by the GICC\_ prefix.

When describing a GIC that includes the GIC Virtualization Extensions, a CPU interface is sometimes called a *physical CPU interface*, to avoid possible confusion with a virtual CPU interface.

### Virtual CPU interfaces

The GIC Virtualization Extensions add a virtual CPU interface for each processor in the system. Each virtual CPU interface is partitioned into the following blocks:

#### Virtual interface control

The main component of the virtual interface control block is the GIC virtual interface control registers, that include a list of active and pending virtual interrupts for the current virtual machine on the connected processor. Typically, these registers are managed by the hypervisor that is running on that processor.

Virtual interface control block registers are identified by the GICH\_ prefix.

#### Virtual CPU interface

Each virtual CPU interface block provides physical signaling of virtual interrupts to the connected processor. The ARM processor Virtualization Extensions signal these interrupts to the current virtual machine on that processor. The GIC virtual CPU interface registers, accessed by the virtual machine, provide interrupt control and status information for the virtual interrupts. The format of these registers is similar to the format of the physical CPU interface registers.

Virtual CPU interface block registers are identified by the GICV\_ prefix.

#### ————— Note —————

The virtual CPU interface does not support the power management functionality described in [Power management, GIC v2 on page 2-31](#).

A GIC can implement up to eight CPU interfaces, numbered from 0-7. In a GIC that implements the GIC Virtualization Extensions, virtual CPU interface numbering corresponds to the CPU interface numbering, so that CPU interface 0 and virtual CPU interface 0 connect to the same processor.

This model supports implementation of the GIC in uniprocessing or multiprocessing environments, and the GIC Virtualization Extensions extend that support to processors that support virtualization, in which, in Non-secure state:

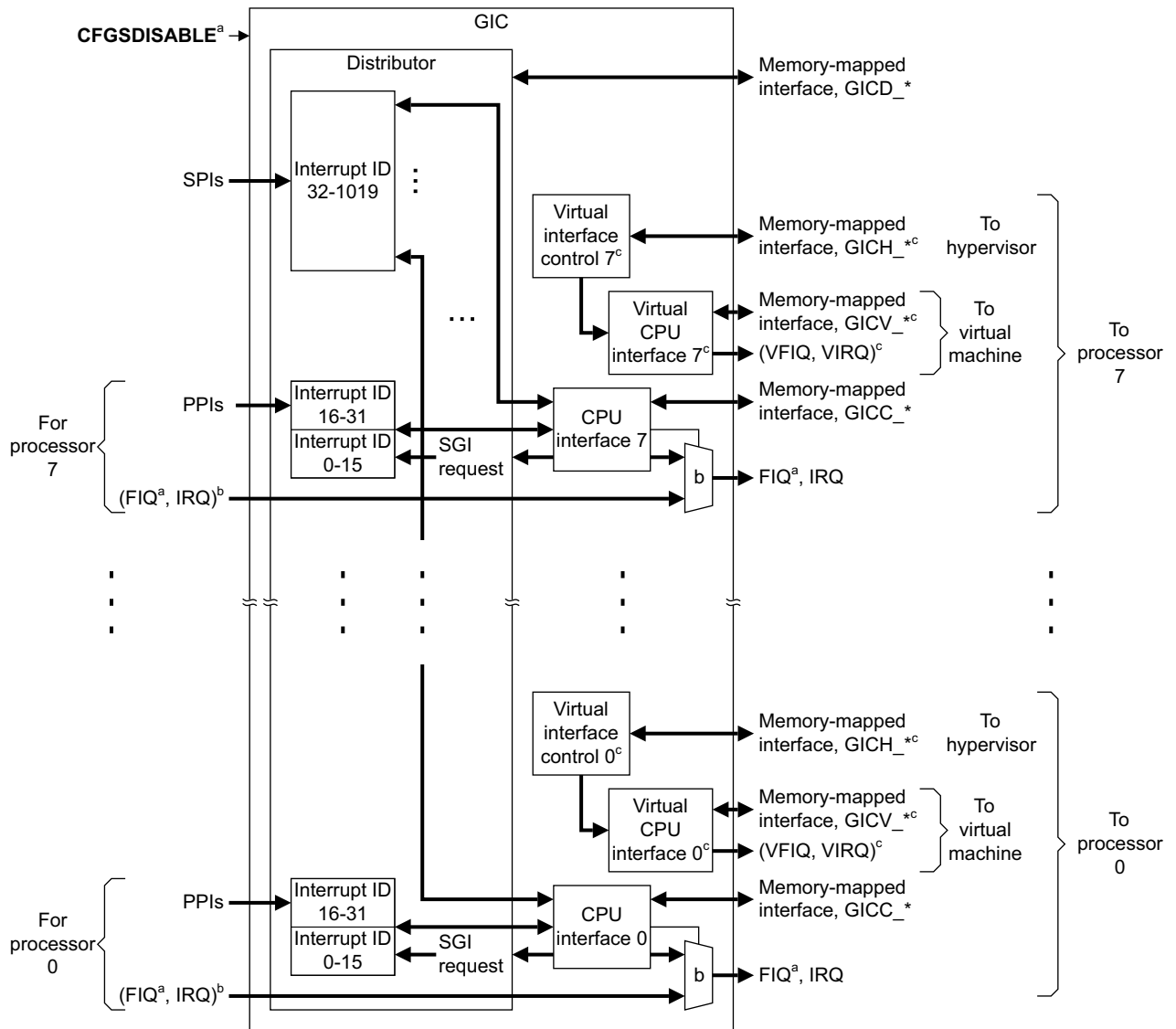
- A Guest OS runs on a virtual machine
- A hypervisor is responsible for switching between virtual machines. This switching includes switching the state held in the GIC virtual interface control registers.

Each block provides part of the GIC programmers' model, and:

- the programmers' model is generally the same for each implemented CPU interface.
- the programmers' model for a virtual CPU interface is generally the same as the programmers' model for a physical CPU interface.

**Note**

- The partitioning of the GIC described in this section is an architectural abstraction. Whether these blocks are implemented separately or combined is IMPLEMENTATION SPECIFIC.
- In a GIC that implements the GIC Security Extensions in a multiprocessor system, a CPU interface can be implemented so that it receives:
  - both Secure and Non-secure accesses
  - only Secure accesses
  - only Non-secure accesses.



<sup>a</sup> In GICv1, applies only if Security Extensions are implemented

<sup>b</sup> Optional input and bypass multiplexer, see text

<sup>c</sup> Applies only to GICv2 with Virtualization Extensions

**Figure 2-1 GIC logical partitioning**

The remainder of this chapter, and [Chapter 3 Interrupt Handling and Prioritization](#) and [Chapter 4 Programmers' Model](#), describe the GIC without the GIC Virtualization Extensions. [Chapter 5 GIC Support for Virtualization](#) describes the features added by the GIC Virtualization Extensions.

## 2.2 The Distributor

The Distributor centralizes all interrupt sources, determines the priority of each interrupt, and for each CPU interface forwards the interrupt with the highest priority to the interface, for priority masking and preemption handling.

The Distributor provides a programming interface for:

- Globally enabling the forwarding of interrupts to the CPU interfaces.
- Enabling or disabling each interrupt.
- Setting the priority level of each interrupt.
- Setting the target processor list of each interrupt.
- Setting each peripheral interrupt to be level-sensitive or edge-triggered.
- Setting each interrupt as either Group 0 or Group 1.

———— **Note** ————

For GICv1, setting interrupts as Group 0 or Group 1 is possible only when the implementation includes the GIC Security Extensions.

- Forwarding an SGI to one or more target processors.

In addition, the Distributor provides:

- visibility of the state of each interrupt
- a mechanism for software to set or clear the pending state of a peripheral interrupt.

### 2.2.1 Interrupt IDs

Interrupts from sources are identified using *ID numbers*. Each CPU interface can see up to 1020 interrupts. The banking of SPIs and PPIs increases the total number of interrupts supported by the Distributor.

The GIC assigns interrupt ID numbers ID0-ID1019 as follows:

- Interrupt numbers ID32-ID1019 are used for SPIs.
- Interrupt numbers ID0-ID31 are used for interrupts that are private to a CPU interface. These interrupts are banked in the Distributor.

A banked interrupt is one where the Distributor can have multiple interrupts with the same ID. A banked interrupt is identified uniquely by its ID number and its associated CPU interface number. Of the banked interrupt IDs:

- ID0-ID15 are used for SGIs
- ID16-ID31 are used for PPIs

In a multiprocessor system:

- A PPI is forwarded to a particular CPU interface, and is private to that interface. In prioritizing interrupts for a CPU interface the Distributor does not consider PPIs that relate to other interfaces.
- Each connected processor issues an SGI by writing to the **GICD\_SGIR** in the Distributor. Each write can generate SGIs with the same ID that target multiple processors.

In the Distributor, an SGI is identified uniquely by the combination of its interrupt number, ID0-ID15, the target processor ID, CPUID0-CPUID7, and the *processor source ID*, CPUID0-CPUID7, of the processor that issued the SGI. When the CPU interface communicates the interrupt ID to a targeted processor, it also provides the processor source ID, so that the targeted processor can uniquely identify the SGI.

SGI banking means the GIC can handle multiple SGIs simultaneously, without resource conflicts.

The Distributor ignores any write to the **GICD\_SGIR** that is not from a processor that is connected to one of the CPU interfaces. How the Distributor determines the processor source ID of a processor writing to the **GICD\_SGIR** is IMPLEMENTATION SPECIFIC.

In a uniprocessor system, there is no distinction between shared and private interrupts, because all interrupts are visible to the processor. In this case the processor source ID value is 0.



- Interrupt numbers ID1020-ID1023 are reserved for special purposes, see [Special interrupt numbers on page 3-43](#).

System software sets the priority of each interrupt. This priority is independent of the interrupt ID number.

In any system that implements the ARM Security Extensions, to support a consistent model for message passing between processors, ARM strongly recommends that all processors reserve:

- ID0-ID7 for Non-secure interrupts
- ID8-ID15 for Secure interrupts.

## 2.3 CPU interfaces

Each CPU interface block provides the interface for a processor that is connected to the GIC. Each CPU interface provides a programming interface for:

- enabling the signaling of interrupt requests to the processor
- acknowledging an interrupt
- indicating completion of the processing of an interrupt
- setting an interrupt priority mask for the processor
- defining the preemption policy for the processor
- determining the highest priority pending interrupt for the processor.

When enabled, a CPU interface takes the highest priority pending interrupt for its connected processor and determines whether the interrupt has *sufficient priority* for it to signal the interrupt request to the processor. To determine whether to signal the interrupt request to the processor, the CPU interface considers the interrupt priority mask and the preemption settings for the processor. At any time, the connected processor can read the priority of its highest priority active interrupt from its `GICC_HPPIR`, a CPU interface register.

The mechanism for signaling an interrupt to the processor is IMPLEMENTATION DEFINED.

### ————— Note —————

On ARM processor implementations, the traditional mechanism for signaling an interrupt request is by asserting **nIRQ** or **nFIQ**.

The processor acknowledges the interrupt request by reading the CPU interface Interrupt Acknowledge Register. This read returns one of:

- The ID number of the highest priority pending interrupt, if that interrupt is of sufficient priority for it to be signaled to the processor. This is the normal response to an interrupt acknowledge.
- Exceptionally, an ID number that indicates a *spurious interrupt*.

When the processor acknowledges the interrupt at the CPU interface, the Distributor changes the status of the interrupt from pending to either active, or active and pending. At this point the CPU interface can signal another interrupt to the processor, to preempt interrupts that are active on the processor. If there is no pending interrupt with sufficient priority for signaling to the processor, the interface deasserts the interrupt request signal to the processor.

When the interrupt handler on the processor has completed the processing of an interrupt, it writes to the CPU interface to indicate interrupt completion. There are two stages to interrupt completion:

- priority drop, meaning the priority of the processed interrupt can no longer prevent the signaling of another interrupt to the processor
- interrupt deactivation, meaning the Distributor removes the active state of the interrupt.

In a GICv1 implementation, these two stages always happen together, when the processor writes to the CPU interface End of Interrupt register.

In a GICv2 implementation, the `GICC_CTLR.EOImode` bit determines whether:

- the two stages happen together, when the processor writes to the CPU interface End of Interrupt register
- the two stages are separated, so that:
  - priority drop happens when the processor writes to the CPU interface End of Interrupt register
  - interrupt deactivation happens later, when the processor writes to the CPU interface Deactivate Interrupt register.

For more information, see [Priority drop and interrupt deactivation on page 3-38](#).

### 2.3.1 Interrupt signal bypass, and GICv2 bypass disable

In all GIC implementations, a CPU interface optionally includes interrupt signal bypass, so that, when the signaling of an interrupt by the interface is disabled, a system legacy interrupt signal is passed to the interrupt request input on the processor, bypassing the GIC functionality.

Figure 2-2 shows the implementation of interrupt signal bypass on a GICv1 implementation that does not include the GIC Security Extensions.

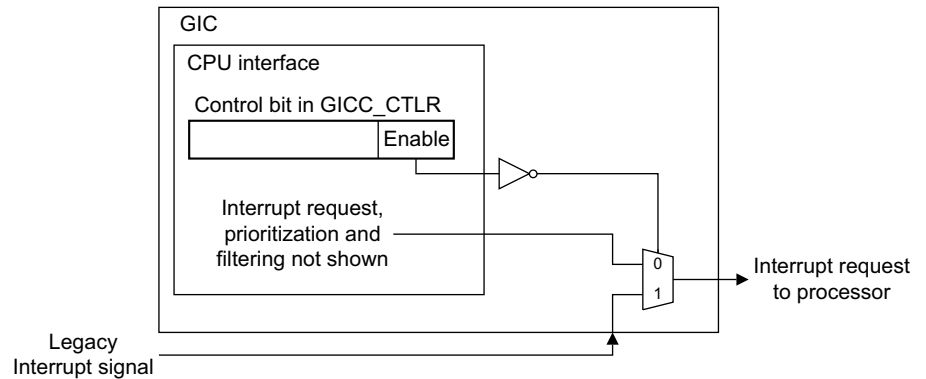


Figure 2-2 Interrupt signal bypass, GICv1 without Security Extensions

Figure 2-2 shows the simplest implementation of interrupt signal bypass. In other GIC implementations, interrupt signal bypass is more complicated:

- A GICv1 implementation that includes the GIC Security Extensions supports interrupt grouping, and the use of FIQ interrupts to signal Group 0 interrupts. *Interrupt bypass, GICv1 with GIC Security Extensions* describes the implementation of interrupt bypass on such an implementation.
- If a GICv2 implementation interrupt supports signal bypass, it uses the same model as a GICv1 implementation that includes the GIC Security Extensions, but must also provide disable bits for the interrupt signal bypass operation. For more information see *GICv2 interrupt bypass, with bypass disable* on page 2-28.

#### Note

Many ARM processors, including processors that implement the ARMv7-A or ARMv7-R architecture profiles, implement two active-LOW interrupt request signals, **nIRQ** and **nFIQ**. However, this GIC architecture specification describes only the logic of the interrupt request signals, not the physical signaling of interrupts to a connected processor. Therefore, it describes two active-HIGH interrupt requests, **IRQ** and **FIQ**.

### Interrupt bypass, GICv1 with GIC Security Extensions

When a GIC implementation supports interrupt grouping, a CPU interface can provide two interrupt exception request outputs, **IRQ** and **FIQ**. It always uses the **IRQ** output to signal Group 1 interrupts, but can use the **FIQ** output to signal Group 0 interrupts. In such an implementation, the CPU interface can include interrupt signal bypass for both interrupt signals. For this case, Table 2-1 on page 2-28 shows how **GICC\_CTLR** controls the GIC interrupt outputs.

Table 2-1 Interrupt signal bypass behavior, GICv1 with Security Extensions

GICC_CTLR register bits			GIC interrupt outputs	
FIQEn	EnableGrp0	EnableGrp1	IRQ request behavior	FIQ request behavior
0	0	0	Bypass	Bypass
		1	Driven by GIC CPU interface	Bypass
	1	0	Driven by GIC CPU interface	Bypass
		1	Driven by GIC CPU interface	Bypass
1	0	0	Bypass	Bypass
		1	Driven by GIC CPU interface	Bypass
	1	0	Bypass	Driven by GIC CPU interface
		1	Driven by GIC CPU interface	Driven by GIC CPU interface

For such an implementation, Figure 2-3 shows the signaling of the Group 0 and Group 1 interrupts.

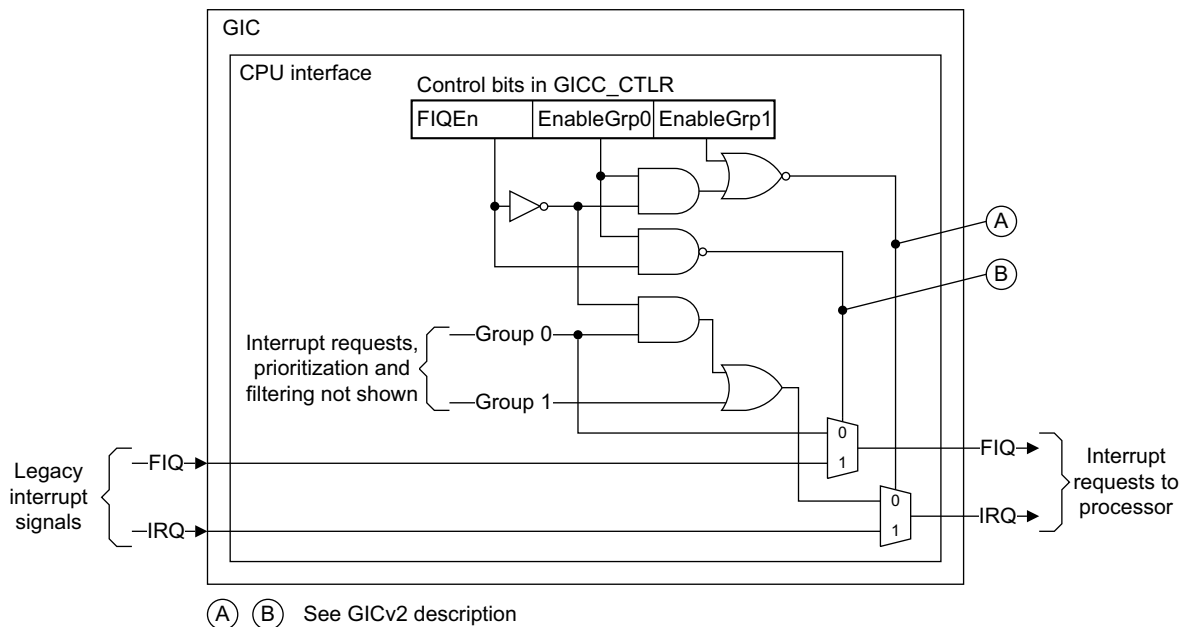


Figure 2-3 GICv1 Group 0 and Group 1 interrupt signaling, with interrupt signal bypass

### GICv2 interrupt bypass, with bypass disable

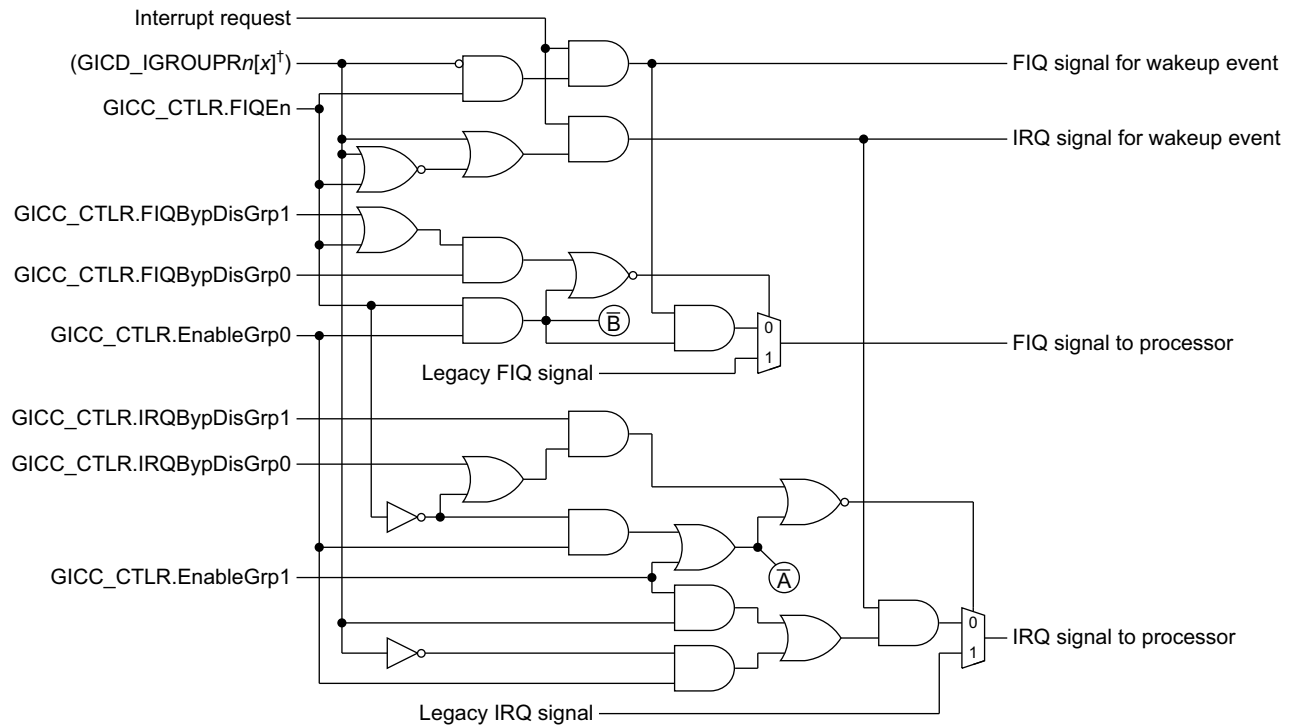
When a CPU interface in a GICv2 implementation includes interrupt signal bypass, it:

- implements the bypass scheme described in [Interrupt bypass, GICv1 with GIC Security Extensions on page 2-27](#)
- in addition, must implement GICC\_CTLR control bits that disable the interrupt signal bypass functionality.

When not being driven by the CPU interface, each interrupt output signal can be deasserted rather than being driven by the legacy interrupt input. This behavior is controlled by the GICC\_CTLR bypass disable bits:

- FIQByDisGrp0
- FIQByDisGrp1
- IRQByDisGrp0
- IRQByDisGrp1.

Figure 2-4 shows the control logic of the signaling of interrupts by a CPU interface. *Power management, GIC v2* on page 2-31 gives more information about the wakeup event signals shown in Figure 2-4.



† Values of  $n$  and  $x$  correspond to the requested interrupt

$\bar{A}$  is the inverse of  $A$  in the GICv1 implementation that supports interrupt grouping

$\bar{B}$  is the inverse of  $B$  in the GICv1 implementation that supports interrupt grouping

**Figure 2-4 GICv2 interrupt bypass logic, with bypass disable**

*Exception generation pseudocode* on page 3-64 also describes this interrupt signaling.

Table 2-2 on page 2-30 shows how, when a CPU interface might signal an IRQ request to a connected processor, bits in *GICC\_CTLR*, and whether the IRQ request is Group 0 or Group 1, determine the IRQ signaling by the interface. In the *IRQ request signaling behavior* column of this table:

<b>Bypass</b>	Indicates that the IRQ signal to the processor is driven by the legacy IRQ signal.
<b>Deasserted</b>	Indicates that the IRQ signal to the processor is deasserted.
<b>Driven by GIC</b>	Indicates that the IRQ signal to the processor is driven by the GIC CPU interface logic.

Table 2-2 IRQ request behavior, GICv2

GICC_CTLR register bits					IRQ for signaling	IRQ request signaling behavior
EnableGrp1	EnableGrp0	FIQEn	IRQBypDisGrp1	IRQBypDisGrp0		
0	0	0	0	x	x	Bypass
0	0	0	1	x	x	Deasserted
0	0	1	0	x	x	Bypass
0	0	1	1	0	x	Bypass
0	0	1	1	1	x	Deasserted
0	1	0	x	x	Group 0	Driven by GIC
0	1	0	x	x	Group 1	Deasserted
0	1	1	0	x	x	Bypass
0	1	1	1	0	x	Bypass
0	1	1	1	1	x	Deasserted
1	0	x	x	x	Group 0	Deasserted
1	0	x	x	x	Group 1	Driven by GIC
1	1	0	x	x	x	Driven by GIC
1	1	1	x	x	Group 0	Deasserted
1	1	1	x	x	Group 1	Driven by GIC

Table 2-3 shows how, when a CPU interface might signal an FIQ request to a connected processor, bits in GICC\_CTLR the FIQ signaling by the interface:

Table 2-3 FIQ request behavior, GICv2

GICC_CTLR register bits				FIQ request signaling behavior
EnableGrp0	FIQEn	FIQBypDisGrp0	FIQBypDisGrp1	
0	0	0	x	Bypass, driven by legacy FIQ signal
0	0	1	0	Bypass, driven by legacy FIQ signal
0	0	1	1	FIQ interrupt output deasserted
0	1	0	x	Bypass, driven by legacy FIQ signal
0	1	1	x	FIQ interrupt output deasserted
1	0	0	x	Bypass, driven by legacy FIQ signal
1	0	1	0	Bypass, driven by legacy FIQ signal
1	0	1	1	FIQ interrupt output deasserted
1	1	x	x	Driven by GIC CPU interface

### 2.3.2 Power management, GIC v2

The GICv2 architecture supports wakeup events in implementations that require power management.

As shown in [Figure 2-4 on page 2-29](#), the GICv2 interrupt bypass logic described in [GICv2 interrupt bypass, with bypass disable on page 2-28](#) includes signals that can be used as wakeup signals to a system power controller. These signals are available even when both interrupt signaling by the GIC, and interrupt bypass, are disabled.

In addition, the `GICC_APRn` registers provide support for preserving and restoring state in power-management applications. However, to ensure that Non-secure accesses do not interfere with Secure operation, Secure and Non-secure copies of these registers are provided.





# Chapter 3

## Interrupt Handling and Prioritization

This chapter describes the requirements for interrupt handling and prioritization in the GIC. It contains the following sections:

- *About interrupt handling and prioritization on page 3-34*
- *General handling of interrupts on page 3-37*
- *Interrupt prioritization on page 3-44*
- *The effect of interrupt grouping on interrupt handling on page 3-48*
- *Interrupt grouping and interrupt prioritization on page 3-53*
- *Additional features of the GIC Security Extensions on page 3-59*
- *Pseudocode details of interrupt handling and prioritization on page 3-61*
- *The effect of the Virtualization Extensions on interrupt handling on page 3-67*
- *Example GIC usage models on page 3-68.*

## 3.1 About interrupt handling and prioritization

The following subsections give more information about the interrupts supported by a GIC, and how a connected processor must determine the range of interrupt IDs supported by the GIC:

- [Handling different interrupt types in a multiprocessor system on page 3-35](#)
- [Identifying the supported interrupts on page 3-35.](#)

The remainder of the chapter describes interrupt handling and prioritization.

Interrupt handling describes:

- how the GIC recognizes interrupts
- how software can program the GIC to configure and control interrupts
- the state machine the GIC maintains for each interrupt on each CPU interface
- how the exception model of a processor interacts with the GIC.

Prioritization describes:

- the configuration and control of interrupt priority
- the order of execution of pending interrupts
- the determination of when interrupts are visible to a target processor, including:
  - interrupt priority masking
  - priority grouping
  - preemption of an active interrupt.

The following sections describe interrupt handling and prioritization:

- [General handling of interrupts on page 3-37](#)
- [Interrupt prioritization on page 3-44.](#)

The GIC architecture supports uniprocessor and multiprocessor systems:

- in a uniprocessor system the GIC has a single processor interface, the *CPU interface*
- in a multiprocessor system the GIC has a CPU interface for each connected processor.

In either a uniprocessor or a multiprocessor system, a GIC implementation can include the GIC Security Extensions. The GIC Security Extensions:

- recognize that a connected processor that implements the ARM Security Extensions makes either Secure accesses or Non-secure accesses to the GIC registers
- implement the GIC registers to take account of Secure and Non-secure accesses, so that:
  - some registers are *banked*, to provide separate Secure and Non-secure copies
  - some registers are *Secure*, meaning they are only accessible using Secure accesses
  - the remaining registers are *Common*, meaning they are accessible by Secure and Non-secure accesses.
- use the GIC interrupt grouping feature to support the handling of Secure and Non-secure interrupts, in which case:
  - Group 0 interrupts are Secure interrupts
  - Group 1 interrupts are Non-secure interrupts.
- in a multiprocessor system, might implement the GIC Security Extensions on only some of its CPU interfaces.

Except for a GICv1 implementation that does not include the GIC Security Extensions, all implementations of the GIC architecture support *interrupt grouping*. With interrupt grouping:

- by default, all interrupts are Group 0 interrupts, and are signaled to a connected processor using the IRQ interrupt request
- each interrupt can be configured as Group 1 interrupt, or as a Group 0 interrupt
- a CPU interface can be configured to signal Group 0 interrupts to a connected processor using the FIQ interrupt request.

Interrupt grouping, and the GIC Security Extensions, make interrupt handling and prioritization more complex. The following sections describe the effect of interrupt grouping and the GIC Security Extensions:

- [The effect of interrupt grouping on interrupt handling on page 3-48](#)
- [Interrupt grouping and interrupt prioritization on page 3-53.](#)

### 3.1.1 Handling different interrupt types in a multiprocessor system

A GIC supports *peripheral interrupts* and *software-generated interrupts*, see [Interrupt types on page 1-18](#).

In a multiprocessor implementation the GIC handles:

- software generated interrupts (SGIs) using the GIC N-N model
- peripheral (hardware) interrupts using the GIC 1-N model.

See [Models for handling interrupts on page 1-19](#) for definitions of the two models.

### 3.1.2 Identifying the supported interrupts

The GIC architecture defines different ID values for the different types of interrupt, see [Interrupt IDs on page 2-24](#). However, there is no requirement for the GIC to implement a continuous block of interrupt IDs for any interrupt type.

#### ————— Note —————

ARM strongly recommends that implemented interrupts are grouped to use the lowest ID numbers and as small a range of interrupt IDs as possible, because this reduces the number of registers that must be implemented, and that discovery routines must check.

To correctly handle interrupts, software must know what interrupt IDs are supported by the GIC. Software can use the [GICD\\_ISENABLERns](#) to discover this information. If the processor implements the ARM Security Extensions, Secure software determines the interrupts that are visible to Non-secure software. The Non-secure software must know which interrupts it can see, and might use this discovery process to find this information.

GICD\_ISENABLER0 provides the Set-enable bits for both:

- SGIs, using interrupt IDs 15-0, corresponding to register bits [15:0]
- PPIs, using interrupt IDs 31-16, corresponding to register bits [31:16].

The remaining [GICD\\_ISENABLERns](#), from GICD\_ISENABLER1, provide the Set-enable bits for the SPIs, starting at interrupt ID 32.

If an interrupt is:

- not supported, the Set-enable bit corresponding to its interrupt ID is RAZ/WI
- supported and permanently enabled, the Set-enable bit corresponding to its interrupt ID is RAO/WI.

Software discovers the interrupts that are supported by:

1. Reading the [GICD\\_TYPER](#). The [GICD\\_TYPER.ITLinesNumber](#) field identifies the number of implemented [GICD\\_ISENABLERns](#), and therefore the maximum number of SPIs that might be supported.
2. Writing to the [GICD\\_CTLR](#) to disable forwarding of interrupts from the distributor to the CPU interfaces. For more information, see [Enabling and disabling the Distributor and CPU interfaces on page 4-77](#).
3. For each implemented [GICD\\_ISENABLERn](#), starting with GICD\_ISENABLER0:
  - Writing 0xFFFFFFFF to the [GICD\\_ISENABLERn](#).
  - Reading the value of the [GICD\\_ISENABLERn](#). Bits that read as 1 correspond to supported interrupt IDs.

Software uses the `GICD_ICENABLERn`s to discover the interrupts that are permanently enabled. For each implemented `GICD_ICENABLERn`, starting with `GICD_ICENABLER0`, software:

1. Writes `0xFFFFFFFF` to the `GICD_ICENABLERn`. This disables all interrupts that can be disabled.
2. Reads the value of the `GICD_ICENABLERn`. Bits that read as 1 correspond to interrupts that are permanently enabled.
3. Writes 1 to any `GICD_ISENABLERn` bits corresponding to interrupts that must be re-enabled.

The GIC implements the same number of `GICD_ISENABLERn`s and `GICD_ICENABLERn`s.

When software has completed its discovery, it typically writes to the `GICD_CTLR` to re-enable forwarding of interrupts from the Distributor to the CPU interfaces.

If the GIC implements the GIC Security Extensions, software can use Secure accesses to discover all the supported interrupt IDs, see *The effect of interrupt grouping on interrupt handling on page 3-48* for more information.

Software using Non-secure accesses can discover and control only the interrupts that are configured as Non-secure.

If Secure software changes the security configuration of any interrupts after Non-secure software has discovered its supported interrupts, it must communicate the effect of those changes to the Non-secure software.

In a GIC that provides interrupt grouping, software can:

- write to the `GICD_IGROUPRn` registers, to configure interrupts as Group 0 or Group 1
- control the forwarding of Group 0 and Group 1 interrupts independently, using the `GICD_CTLR.EnableGrp0` and `GICD_CTLR.EnableGrp1` bits.

## 3.2 General handling of interrupts

The Distributor maintains a state machine for each supported interrupt on each CPU interface. [Interrupt handling state machine on page 3-41](#) describes this state machine and its state transitions. The possible states of an interrupt are:

- inactive
- pending
- active
- active and pending.

---

### Note

- This section gives an overview of the handling of interrupts in a GIC implementation that does not include the GIC Security Extensions. It does not give a full description of handling grouped interrupts. Interrupt grouping, and the GIC Security Extensions, extend the basic model of GIC operation described in this section. For more information see [The effect of interrupt grouping on interrupt handling on page 3-48](#).
  - This basic model of interrupt handling also applies to the handling of virtual interrupts in an implementation that includes the GIC Virtualization Extensions. For more information, see [Chapter 5 GIC Support for Virtualization](#).
- 

When the GIC recognizes an interrupt request, it marks its state as *pending*. Regenerating a pending interrupt does not affect the state of the interrupt.

The GIC interrupt handling sequence is:

1. The GIC determines the interrupts that are enabled.
2. For each pending interrupt, the GIC determines the targeted processor or processors.
3. For each CPU interface, the Distributor forwards the highest priority pending interrupt that targets that interface.
4. Each CPU interface determines whether to signal an interrupt request to its processor, and if required, does so.
5. The processor acknowledges the interrupt, and the GIC returns the interrupt ID and updates the interrupt state.
6. After processing the interrupt, the processor signals *End of Interrupt* (EOI) to the GIC.

In more detail, these steps are as follows:

1. The GIC determines whether each interrupt is enabled. An interrupt that is not enabled has no effect on the GIC.
2. For each enabled interrupt that is pending, the Distributor determines the targeted processor or processors.
3. For each processor, the Distributor determines the highest priority pending interrupt, based on the priority information it holds for each interrupt, and forwards the interrupt to the targeted CPU interfaces.
4. If the distributor is forwarding an interrupt request to a CPU interface, the CPU interface determines whether the interrupt has *Sufficient priority* to be signaled to the processor. If the interrupt has sufficient priority, the GIC signals an interrupt request to the processor.
5. When a processor takes the interrupt exception, it reads the `GICC_IAR` of its CPU interface to acknowledge the interrupt. This read returns an Interrupt ID, and for an SGI, the source processor ID, that the processor uses to select the correct interrupt handler. When it recognizes this read, the GIC changes the state of the interrupt as follows:
  - if the pending state of the interrupt persists when the interrupt becomes active, or if the interrupt is generated again, from pending to active and pending.
  - otherwise, from pending to active

————— **Note** —————

- A level-sensitive peripheral interrupt persists when it is acknowledged by the processor, because the interrupt signal to the GIC remains asserted until the *Interrupt Service Routine (ISR)* running on the processor accesses the peripheral asserting the signal.
- In a multiprocessor implementation, the GIC handles:
  - PPIs and SGIs using the GIC N-N model, where the acknowledgement of an interrupt by one processor has no effect on the state of the interrupt on other CPU interfaces
  - SPIs using the GIC 1-N model, where the acknowledgement of an interrupt by one processor removes the pending status of the interrupt on any other targeted processors, see [Implications of the 1-N model on page 3-41](#).
- In GICv2, when using a software model with the `GICC_CTLR.AckCtl` bit set to 0, separate registers are used to manage Group 0 and Group 1 interrupts, as follows:
  - `GICC_IAR`, `GICC_EOIR`, and `GICC_HPPIR` for Group 0 interrupts
  - `GICC_AIAR`, `GICC_AEOIR`, and `GICC_AHPPIR` for Group 1 interrupts.ARM deprecates the use of `GICC_CTLR.AckCtl`, and strongly recommends using a software model where `GICC_CTLR.AckCtl` is set to 0, see [The effect of interrupt grouping on interrupt acknowledgement on page 3-50](#).

6. When the processor has completed handling the interrupt, it must signal this completion to the GIC. As described in [Priority drop and interrupt deactivation](#), this:
- always requires a valid write to an *end of interrupt register (EOIR)*
  - might also require a subsequent write to the deactivate interrupt register, `GICC_DIR`.

For each CPU interface, the GIC architecture requires the order of the valid writes to an EOIR to be the reverse of the order of the reads from the `GICC_IAR` or `GICC_AIAR`, so that each valid EOIR write refers to the most recent interrupt acknowledge.

If, after the EOIR write, there is no pending interrupt of *Sufficient priority*, the CPU interface deasserts the interrupt exception request to the processor.

A CPU interface never signals to the connected processor any interrupt that is active and pending. It only signals interrupts that are pending and have sufficient priority:

- For PPIs and SGIs, the active status of particular interrupt ID is banked between CPU interfaces. This means that if a particular interrupt ID is active or active and pending on a CPU interface, then no interrupt with that same ID is signaled on that CPU interface.
- For SPIs, the active status of an interrupt is common to all CPU interfaces. This means that if an interrupt is active or active and pending on one CPU interface then it is not signaled on any CPU interface.

For more information about the steps in this process see:

- [Priority drop and interrupt deactivation](#)
- [Interrupt prioritization on page 3-44](#)
- [The effect of interrupt grouping on interrupt handling on page 3-48](#)
- [Interrupt grouping and interrupt prioritization on page 3-53](#).

### 3.2.1 Priority drop and interrupt deactivation

When a processor completes the processing of an interrupt, it must signal this completion to the GIC. Interrupt completion requires the following changes to the GIC state:

**Priority drop** Priority drop is the drop in the *Running priority* that occurs on a valid write to an EOIR, either the `GICC_EOIR` or the `GICC_AEOIR`. A valid write is a write that is not UNPREDICTABLE, is not ignored, and is not writing an interrupt ID value greater than 1019.

On priority drop, the running priority is reduced from the priority of the interrupt referenced by the EOIR write to either:

- the priority of the highest-priority active interrupt for which there has been no EOIR write

- the *Idle priority*, if there is no active interrupt for which there has been no EOIR write.

See [Preemption on page 3-45](#) for more information about running priority.

### Interrupt deactivation

Interrupt deactivation is the change of the state of an interrupt, either:

- from active and pending, to pending
- from active, to idle.

On a GICv1 implementation, and on a GICv2 implementation when `GICC_CTLR.EOImode` is set to 0, a valid EOIR write also deactivates the interrupt it references.

On a GICv2 implementation, setting `GICC_CTLR.EOImode` to 1 separates the priority drop and interrupt deactivation operations, and interrupt handling software must:

1. Perform a valid EOIR write, to cause priority drop on the GIC CPU interface.
2. Subsequently, write to the `GICC_DIR`, to deactivate the interrupt.

The GIC architecture specification requires that valid EOIR writes are ordered, so that:

- a valid `GICC_EOIR` write corresponds to the most recently acknowledged interrupt
- a valid `GICC_AEOIR` write corresponds to the most recently acknowledged Group 1 interrupt.
- whether a `GICC_EOIR` write affects Group 0 or Group 1 interrupts depends on both:
  - the value of the `GICC_CTLR.AckCtl` bit
  - if the GIC implements the GIC Security Extensions, whether the write is Secure or Non-secure.

#### ———— Note —————

In a GICv2 implementation that includes the Security Extensions:

- `GICC_AEOIR` is an alias of the Non-secure copy of `GICC_EOIR`
- `GICC_AIAR` is an alias of the Non-secure copy of `GICC_IAR`
- `GICC_AIAR` and `GICC_AEOIR` are Secure registers, meaning they are accessible only by Secure accesses.

There is no ordering requirement for `GICC_DIR` writes. However, the effect is UNPREDICTABLE if software writes to `GICC_DIR` when:

- `GICC_CTLR.EOImode` is set to 0
- `GICC_CTLR.EOImode` is set to 1 and there has not been a corresponding write to `GICC_EOIR` or `GICC_AEOIR`.

When virtualizing physical interrupts, ARM recommends that, for each CPU interface that corresponds to a processor running virtual machines:

- `GICC_CTLR.EOImode` bit is set to 1
- if the GIC implements the GIC Security Extensions, the `GICC_CTLR.EOImodeNS` bit is set to 1

See [Completion of virtualized physical interrupts on page 5-161](#) for more information.

## 3.2.2 Interrupt controls in the GIC

The following sections describe the interrupt controls in the GIC:

- [Interrupt enables](#)
- [Setting and clearing pending state of an interrupt on page 3-40](#)
- [Finding the active or pending state of an interrupt on page 3-40](#)
- [Generating an SGI on page 3-40.](#)

### Interrupt enables

For peripheral interrupts, a processor:

- enables an interrupt by writing to the appropriate `GICD_ISENBLERn` bit
- disables an interrupt by writing to the appropriate `GICD_ICENBLERn` bit.

Whether SGIs are permanently enabled, or can be enabled and disabled by writes to the [GICD\\_ISENABLER<sub>n</sub>](#) and [GICD\\_ICENABLER<sub>n</sub>](#), is IMPLEMENTATION DEFINED.

Writes to the [GICD\\_ISENABLER<sub>n</sub>](#)s and [GICD\\_ICENABLER<sub>n</sub>](#)s control whether the Distributor forwards specific interrupts to the CPU interfaces. Disabling an interrupt by writing to the appropriate [GICD\\_ICENABLER<sub>n</sub>](#) does not prevent that interrupt from changing state, for example becoming pending.

### Setting and clearing pending state of an interrupt

For peripheral interrupts, a processor can:

- set the pending state by writing to the appropriate [GICD\\_ISPENDR<sub>n</sub>](#) bit
- clear the pending state by writing to the appropriate [GICD\\_ICPENDR<sub>n</sub>](#) bit.

For a level-sensitive interrupt:

- If the hardware signal of an interrupt is asserted when a processor writes to the corresponding [GICD\\_ICPENDR<sub>n</sub>](#) bit then the write to the register has no effect on the pending state of the interrupt.
- If a processor writes a 1 to an [GICD\\_ISPENDR<sub>n</sub>](#) bit then the corresponding interrupt becomes pending regardless of the state of the hardware signal of that interrupt, and remains pending regardless of the assertion or deassertion of the signal.

For more information about the control of the pending state of a level-sensitive interrupt see [Control of the pending status of level-sensitive interrupts on page 4-100](#).

For SGIs, the GIC ignores writes to the corresponding [GICD\\_ISPENDR<sub>n</sub>](#) and [GICD\\_ICPENDR<sub>n</sub>](#) bits. A processor cannot change the state of a software-generated interrupt by writing to these registers. Typically, an SGI is made pending by writing to the [GICD\\_SGIR](#). In GICv2, the pending state of SGIs can also be modified directly using the [GICD\\_SPENDSGIR<sub>n</sub>](#) and [GICD\\_CPENDSGIR<sub>n</sub>](#) bits.

### Finding the active or pending state of an interrupt

A processor can find:

- the pending state of an interrupt by reading the corresponding [GICD\\_ISPENDR<sub>n</sub>](#) or [GICD\\_ICPENDR<sub>n</sub>](#) bit
- the active state of an interrupt by reading the corresponding [GICD\\_ISACTIVER<sub>n</sub>](#) or [GICD\\_ICACTIVER<sub>n</sub>](#) bit.

The corresponding register bit is 1 if the interrupt is pending or active. If an interrupt is pending and active the corresponding bit is 1 in both registers.

When preserving or restoring GIC state, a processor must take account of the pending and active state of all interrupts. For more information see [Preserving and restoring GIC state on page 4-155](#).

For an SGI, the corresponding [GICD\\_ISPENDR<sub>n</sub>](#) and [GICD\\_ICPENDR<sub>n</sub>](#) bits RAO if there is a pending interrupt from at least one generating processor that targets the processor reading the [GICD\\_ISPENDR<sub>n</sub>](#) or [GICD\\_ICPENDR<sub>n</sub>](#). In GICv2, the processor that issues the SGI can also be determined by reading the corresponding [GICD\\_SPENDSGIR<sub>n</sub>](#) or [GICD\\_CPENDSGIR<sub>n</sub>](#) bits.

### Generating an SGI

A processor generates an SGI by writing to an [GICD\\_SGIR](#). An SGI can target multiple processors, and the [GICD\\_SGIR](#) write specifies the target processor list. The [GICD\\_SGIR](#) includes optimization for:

- interrupting only the processor that writes to the [GICD\\_SGIR](#)
- interrupting all processors other than the one that writes to the [GICD\\_SGIR](#).

SGIs from different processors use the same interrupt IDs. Therefore, any target processor can receive SGIs with the same interrupt ID from different processors. However, the pending states of any two SGIs are independent if any of the following are different:

- interrupt ID
- source processor



- target processor.

Only one interrupt with a specific interrupt ID can be active on a CPU interface at any time. This means that a CPU interface cannot have two SGIs with the same interrupt ID active at the same time, even if different processors have signaled SGIs with the same interrupt ID to that processor.

On the CPU interface of the target processor, reading the [GICC\\_IAR](#) for an SGI returns both the interrupt ID and the CPU ID of the processor that generated the interrupt, the *source processor* for the interrupt. The combination of interrupt ID and source CPU ID uniquely identifies the interrupt to the target processor.

In a multiprocessor implementation, the interrupt priority of each SGI interrupt ID is defined independently for each target processor, see [Interrupt Priority Registers, GICD\\_IPRIORITYRn](#) on page 4-104. For each CPU interface, all SGIs with a particular interrupt ID that are pending on that interface have the same priority and must be handled serially. The order in which the CPU interface serializes these SGIs is IMPLEMENTATION SPECIFIC.

### 3.2.3 Implications of the 1-N model

In a multiprocessor implementation, the GIC uses the GIC 1-N model, described in [Models for handling interrupts on page 1-19](#), to handle peripheral interrupts that target more than one processor, that is, SPIs. This means that when the GIC recognizes an interrupt acknowledge from one of the target processors it clears the pending state of the interrupt on all the other targeted processors. A GIC implementation must ensure that any interrupt being handled using the 1-N model is only acknowledged by one CPU interface, and that all other interfaces return a spurious interrupt ID.

When multiple target processors attempt to acknowledge the interrupt, the following can occur:

- A processor reads the [GICC\\_IAR](#) and obtains the interrupt ID of the interrupt to be serviced.

———— **Note** —————

In GICv1, more than one target processor might have obtained this interrupt ID, if the processors read their [GICC\\_IAR](#) registers at very similar times. The system might require software on the target processors to ensure that only one processor runs its interrupt service routine. A typical mechanism to achieve this is implementing, in shared memory, a lock on the *interrupt service routine (ISR)*.

- A processor reads the [GICC\\_IAR](#) and obtains the interrupt ID 1023, indicating a spurious interrupt. The processor can return from its interrupt service routine without writing to its [GICC\\_EOIR](#).  
The spurious interrupt ID indicates that the original interrupt is no longer pending, typically because another target processor is handling it.

———— **Note** —————

- A GICv1 implementation might ensure that only one processor can make a 1-N interrupt active, removing the requirement for a lock on the ISR. This is not required by the architecture, and generic GIC code must not rely on this behavior.
- For any processor, if an interrupt is active and pending, the GIC does not signal an interrupt exception request for the interrupt to any processor until the active status is cleared.

### 3.2.4 Interrupt handling state machine

The GIC maintains a state machine for each supported interrupt on each CPU interface. [Figure 3-1 on page 3-42](#) shows an instance of this state machine, and the possible state transitions.

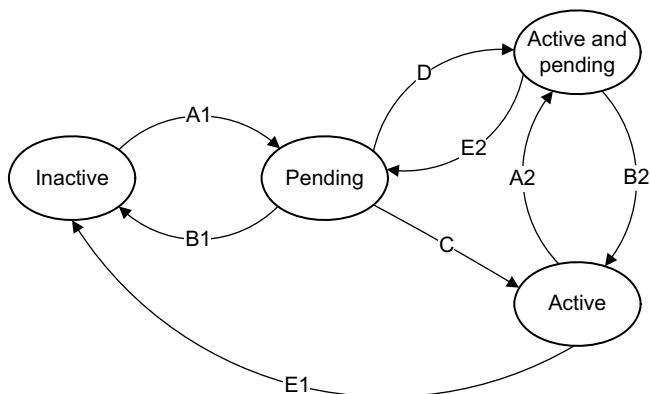


Figure 3-1 Interrupt handling state machine

**Note**

- SGIs are generated only by writes to [GICD\\_SGIR](#) or [GICD\\_SPENDSGIRn](#). Peripheral interrupts are generated by either the assertion of a hardware interrupt request signal to the GIC, or by a write to an [GICD\\_ISPENDRn](#).
- As described in [Priority drop and interrupt deactivation on page 3-38](#):
  - in a GICv1 implementation, priority drop is always associated with interrupt deactivation
  - in a GICv2 implementation, priority drop can be separated from interrupt deactivation.

[Figure 3-1](#) does not show possible separation of priority drop and interrupt deactivation. This happens within the Active state.

When interrupt forwarding by the Distributor and interrupt signaling by the CPU interface are enabled, the conditions that cause each of the state transitions are as follows:

**Transition A1 or A2, add pending state**

For an SGI, occurs if either:

- Software writes to a [GICD\\_SGIR](#) that specifies the processor as a target.
- Software on the target processor writes to the [GICD\\_SPENDSGIRn](#) bit that corresponds to the required source processor and interrupt ID

**Note**

If the GIC implements the GIC Security Extensions and the write to the [GICD\\_SGIR](#) is Secure, the transition occurs only if the security configuration of the specified SGI, for the appropriate CPU interface, corresponds to the [GICD\\_SGIR.NSATT](#) bit value.

For an SPI or PPI, occurs if either:

- a peripheral asserts an interrupt request signal
- software writes to an [GICD\\_ISPENDRn](#).

**Transition B1 or B2, remove pending state**

For an SGI, occurs if software on the target processor writes to the relevant bit of the [GICD\\_CPENDSGIRn](#).

For an SPI or PPI, occurs if either:

- the level-sensitive interrupt is pending only because of the assertion of an input signal, and that signal is deasserted
- the interrupt is pending only because of the assertion of an edge-triggered interrupt signal, or a write to an [GICD\\_ISPENDRn](#), and software writes to the corresponding [GICD\\_ICPENDRn](#).

#### Transition C, pending to active

If the interrupt is enabled and of [Sufficient priority](#) to be signaled to the processor, occurs when software reads from the [GICC\\_IAR](#).

#### Transition D, pending to active and pending

For an SGI, this transition occurs in either of the following circumstances:

- If a write to set the SGI state to pending occurs at approximately the same time as a read of [GICC\\_IAR](#).
- When two or more pending SGIs with the same interrupt ID originate from the same source processor and target the same processor. If one of the SGIs follows transition C, the other SGIs follow transition D

For an SPI or PPI this transition occurs if all the following apply:

- The interrupt is enabled.
- Software reads from the [GICC\\_IAR](#). This read adds the active state to the interrupt.
- In addition, one of the following conditions applies:
  - For a level-sensitive interrupt, the interrupt signal remains asserted. This is usually the case, because the peripheral does not deassert the interrupt until the processor has serviced the interrupt.
  - For an edge-triggered interrupt, whether this transition occurs depends on the timing of the read of the [GICC\\_IAR](#) relative to the detection of the reassertion of the interrupt. Otherwise the read of the [GICC\\_IAR](#) causes transition C, possibly followed by transition A2.

#### Transition E1 or E2, remove active state

Occurs when software deactivates an interrupt by writing to either [GICC\\_EOIR](#) or [GICC\\_DIR](#). For more information see [Priority drop and interrupt deactivation on page 3-38](#). In a GIC implementation that includes the Virtualization Extensions, also occurs if the virtual CPU interface signals that the corresponding physical interrupt has been deactivated.

### 3.2.5 Special interrupt numbers

The GIC architecture reserves interrupt ID numbers 1020-1023 for special purposes. In a GICv1 implementation that does not implement the GIC Security Extensions, the only one of these used is ID 1023. This value is returned to a processor, in response to an interrupt acknowledge, if there is no pending interrupt with sufficient priority for it to be signaled to the processor. It is described as a response to a *spurious interrupt*.

#### ———— Note —————

A race condition can cause a spurious interrupt. For example, a spurious interrupt can occur if a processor writes a 1 to a field in an [GICD\\_ICENABLER<sub>n</sub>](#) that corresponds to a pending interrupt after the CPU interface has signaled the interrupt to the processor and the processor has recognized the interrupt, but before the processor has read from the [GICC\\_IAR](#).

For more information about the special interrupt numbers see [Special interrupt numbers when a GIC supports interrupt grouping on page 3-50](#).

## 3.3 Interrupt prioritization

This section describes interrupt prioritization in the GIC architecture. It includes the following subsections:

- [Preemption on page 3-45](#)
- [Priority masking on page 3-45](#)
- [Priority grouping on page 3-45](#)
- [Interrupt generation on page 3-47](#).

---

### Note

This section describes an implementation that is not using interrupt grouping, and does not include the GIC Security Extensions. Interrupt grouping, and the GIC Security Extensions, extend this basic model of GIC interrupt prioritization. For more information, see [Interrupt grouping and interrupt prioritization on page 3-53](#).

Software configures interrupt prioritization in the GIC by assigning a priority value to each interrupt source. Priority values are 8-bit unsigned binary. A GIC supports a minimum of 16 and a maximum of 256 priority levels. If the GIC implements fewer than 256 priority levels, low-order bits of the priority fields are RAZ/WI. This means that the number of implemented priority field bits is IMPLEMENTATION DEFINED in the range 4-8, as [Table 3-1](#) shows.

**Table 3-1 Effect of not implementing some priority field bits**

Implemented priority bits	Possible priority field values	Number of priority levels
[7:0]	0x00-0xFF (0-255), all values	256
[7:1]	0x00-0xFE, (0-254), even values only	128
[7:2]	0x00-0xFC (0-252), in steps of 4	64
[7:3]	0x00-0xF8 (0-248), in steps of 8	32
[7:4]	0x00-0xF0 (0-240), in steps of 16	16

In the GIC prioritization scheme, lower numbers have higher priority, that is, the lower the assigned priority value the higher the priority of the interrupt. Priority field value 0 always indicates the highest possible interrupt priority, and the lowest priority value depends on the number of implemented priority levels, as [Table 3-1](#) shows.

The [GICD\\_IPRIORITYRn](#) registers hold the priority value for each supported interrupt. An implementation might reserve an interrupt for a particular purpose and assign a fixed priority to that interrupt, meaning the priority value for that interrupt is read-only. For other interrupts, software writes to the [GICD\\_IPRIORITYRn](#) registers to set the interrupt priorities. It is IMPLEMENTATION DEFINED whether a write to [GICD\\_IPRIORITYRn](#) changes the priority of any active interrupt.

To determine the number of priority bits implemented, software can write 0xFF to a writable [GICD\\_IPRIORITYRn](#) priority field, and read back the value stored.

---

### Note

ARM recommends that, before checking the priority range in this way:

- for a peripheral interrupt, software first disables the interrupt
- for an SGI, software first checks that the interrupt is inactive.

If, on a particular CPU interface, multiple pending interrupts have the same priority, and have [Sufficient priority](#) for the interface to signal them to the processor, it is IMPLEMENTATION SPECIFIC how the interface selects which interrupt to signal.

When an interrupt is active on a CPU interface, the GIC might signal a higher-priority interrupt on that CPU interface, see [Preemption on page 3-45](#).

### 3.3.1 Preemption

A CPU interface supports signaling of higher priority pending interrupts to a target processor before an active interrupt completes. A pending interrupt is only signaled if both:

- Its priority is higher than the priority mask for that CPU interface, see [Priority masking](#).
- Its group priority is higher than that of the [Running priority](#) on the CPU interface, see [Priority grouping](#) and [Running Priority Register, GICC\\_RPR](#) on page 4-142.

Preemption occurs at the time when the processor acknowledges the new interrupt, and starts to service it in preference to the previously active interrupt or the currently running process. When this occurs, the initial active interrupt is said to have been *preempted*. Starting to service an interrupt while another interrupt is still active is sometimes described as *interrupt nesting*.

---

**Note**

---

- For a processor that complies with the ARM architecture:
  - The value of the I or F bit in the CPSR determines whether the processor responds to the signaled interrupt by starting the interrupt acknowledge procedure.
  - When processing a preempting interrupt, the processor must save and later restore the context of the previously active ISR.

For more information, see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

- *Priority drop* means the priority of an interrupt no longer affects the [Running priority](#) on the CPU interface, and therefore does not prevent interrupt preemption. In GICv1 implementations, priority drop happens only when an interrupt is deactivated, but in GICv2 implementations, priority drop and interrupt deactivation can be separated. For more information see [Priority drop and interrupt deactivation](#) on page 3-38.
- 

### 3.3.2 Priority masking

The [GICC\\_PMR](#) for a CPU interface defines a priority threshold for the target processor. The GIC only signals pending interrupts with a higher priority than this threshold value to the target processor. A value of zero, the register reset value, masks all interrupts from being signaled to the associated processor. The GIC does not use priority grouping when comparing the priority of a pending interrupt with the priority threshold.

The GIC always masks an interrupt that has the largest supported priority field value. This provides an additional means of preventing an interrupt being signaled to any processor.

---

**Note**

---

Writing 255 to the [GICC\\_PMR](#) always sets it to the largest supported priority field value. [Table 3-1](#) on page 3-44 shows how the largest supported field value varies with the number of implemented priority bits.

---

### 3.3.3 Priority grouping

Priority grouping uses the Binary Point Register, [GICC\\_BPR](#), to split a priority value into two fields, the *group priority* and the *subpriority*. When determining preemption, all interrupts with the same group priority are considered to have equal priority, regardless of the subpriority. This means that there can only be one interrupt active at each group priority. The active group priority is also known as the [Preemption level](#). For more information, see [Active Priorities Registers, GICC\\_APRn](#) on page 4-149.

The GIC uses the group priority field to determine whether a pending interrupt has sufficient priority to preempt an active interrupt, as follows:

- For a pending interrupt to preempt an active interrupt, its group priority must be higher than the group priority of the active interrupt. That is, the value of the group priority field for the new interrupt must be less than the value of the group priority field of the [Running priority](#).
- If there are no active interrupts on the CPU interface, the highest priority pending interrupt can be signaled to a processor, regardless of the group priority.

In each case, the pending interrupt priority is compared with the priority mask, and the interrupt is signaled only if it is not masked. For more information, see [Priority masking on page 3-45](#).

The binary point field in the [GICC\\_BPR](#) controls the split of the priority bits into the two parts. This 3-bit field specifies how many of the least significant bits of the 8-bit interrupt priority field are excluded from the group priority field, as [Table 3-2](#) shows.

**Table 3-2 Priority grouping by binary point**

Binary point value	Interrupt priority field [7:0]		
	Group priority field	Subpriority field	Field with binary point
0	[7:1]	[0]	ggg gggg.s
1	[7:2]	[1:0]	gg gggg.ss
2	[7:3]	[2:0]	g gggg.sss
3	[7:4]	[3:0]	gggg.ssss
4	[7:5]	[4:0]	ggg.sssss
5	[7:6]	[5:0]	gg.ssssss
6	[7]	[6:0]	g.ssssss
7	No preemption	[7:0]	.sssssss

The minimum binary point value supported is IMPLEMENTATION DEFINED in the range 0-3.

GICv1 implementations with the GIC Security Extensions and GICv2 implementations have two binary point registers. The copy of the binary point register used to calculate priority grouping depends on whether the interrupt is a Group 0 interrupt or a Group 1 interrupt, as defined by the [GICD\\_IGROUPRn](#) registers, and also on the value of the [GICC\\_CTLR.CBPR](#) bit.

[Table 3-2](#) shows which binary point register is used for different GIC implementations.

**Table 3-3 Binary point register used to calculate priority grouping**

GIC implementation	Condition	
	(Group 0 interrupt)    CBPR==1 <sup>a</sup>	(Group 1 interrupt) && CBPR==0
GICv1 without Security Extensions <sup>b</sup>	-	-
GICv2 without Security Extensions	<a href="#">GICC_BPR</a>	<a href="#">GICC_ABPR</a>
GIC with Security Extensions	Secure <a href="#">GICC_BPR</a>	Non-secure <a href="#">GICC_BPR</a> <sup>c</sup>

- a. [GICC\\_CTLR.CBPR](#). Not implemented in a GICv1 implementation that does not include the Security Extensions.
- b. A GICv1 implementation without Security Extensions has no interrupt grouping and only one binary point register, [GICC\\_BPR](#), that it always uses to determine the priority grouping.
- c. For a GICv2 with Security Extensions, the [GICC\\_ABPR](#) and Non-secure [GICC\\_BPR](#) are aliases of the same register.

When the [GICC\\_CTLR.CBPR](#) bit is set to 1, software can configure the CPU interface to determine the priority grouping for a Group 1 interrupt using the same binary point register as for a Group 0 interrupt.

Where multiple pending interrupts have the same group priority, the GIC uses the subpriority field to resolve the priority within a group. Where two or more pending interrupts in a group have the same subpriority, how the GIC selects between the interrupts is IMPLEMENTATION SPECIFIC.

### 3.3.4 Interrupt generation

The pseudocode in [Exception generation pseudocode on page 3-64](#) describes the generation of interrupts by the GIC.

## 3.4 The effect of interrupt grouping on interrupt handling

This section describes the effect of interrupt grouping and the GIC Security Extensions on interrupt handling.

A GICv1 implementation that includes the GIC Security Extensions, or any GICv2 implementation, provides two interrupt output signals for IRQ and FIQ exception requests:

- The CPU interface always uses the IRQ exception request for Group 1 interrupts
- Software can configure the CPU interface to use either IRQ or FIQ exception requests for Group 0 interrupts.

At power-on, or after a reset, any GIC implementation is configured to use only a single interrupt output signal, as described in [GIC power on or reset configuration on page 3-51](#).

The remainder of this section describes a GIC that implements interrupt grouping, as follows:

- [GIC interrupt grouping support](#)
- [Special interrupt numbers when a GIC supports interrupt grouping on page 3-50](#)
- [The effect of interrupt grouping on interrupt acknowledgement on page 3-50](#)
- [GIC power on or reset configuration on page 3-51](#).

### 3.4.1 GIC interrupt grouping support

———— **Note** —————

In a GICv1 implementation, interrupt grouping is provided only as part of the GIC Security Extensions.

The [GICD\\_IGROUPRn](#) registers configure each interrupt as Group 0 or Group 1.

In a CPU interface, in a GICv2 implementation, the [GICC\\_\\*](#) alias registers can provide independent control of Group 0 and Group 1 registers, as [Table 3-4](#) shows.

**Table 3-4 CPU interface control of Group 0 and Group 1 interrupts, GICv2**

Function	Register, Group 0	Register, Group 1
Binary point <sup>a</sup>	<a href="#">GICC_BPR</a>	<a href="#">GICC_ABPR</a>
Interrupt acknowledge	<a href="#">GICC_IAR</a>	<a href="#">GICC_AIAR</a>
EOI	<a href="#">GICC_EOIR</a>	<a href="#">GICC_AEOIR</a>
Highest priority pending interrupt	<a href="#">GICC_HPPIR</a>	<a href="#">GICC_AHPPIR</a>

a. See [Table 3-3 on page 3-46](#) for more information.

In an implementation that includes the GIC Security Extensions, the alias registers:

- typically represent aliases of the Non-secure copy of the Group 0 registers, for example [GICC\\_ABPR](#) is an alias of the Non-Secure copy of [GICC\\_BPR](#)
- are accessible only by Secure accesses.



In a GICv1 implementation that includes the GIC Security Extensions:

- The only implemented alias register is [GICC\\_ABPR](#)
- The other controls of the Group 1 interrupts are provided only by the Non-secure copies of the Group 0 control registers, as [Table 3-5](#) shows.

**Table 3-5 CPU interface Non-secure control of Group 1 interrupts**

Function	Non-secure Group 1 control register
Binary point	Non-secure <a href="#">GICC_BPR</a>
Interrupt acknowledge	Non-secure <a href="#">GICC_IAR</a>
EOI	Non-secure <a href="#">GICC_EOIR</a>
Highest priority pending interrupt	Non-secure <a href="#">GICC_HPPIR</a>

In a GIC implementation that includes the GIC Security Extensions, CPU interface Non-secure control of Group 1 interrupts is identical in GICv1 and GICv2. This means that, in a GICv2 implementation, [Table 3-5](#) shows the [GICC\\_\\*](#) registers that provide the Non-secure control of Group 1 interrupts.

In an implementation that supports interrupt grouping, [GICC\\_CTLR](#) contains additional fields, including fields to control the handling of the grouped interrupts:

- Separate enable bits to control the signaling of Group 0 and Group 1 interrupts to the connected processor:
  - bit[0], the Enable bit in a GIC that does not support interrupt grouping, becomes the EnableGrp0 bit, and controls whether Group 0 interrupts are signaled to the processor
  - the EnableGrp1 bit is added, to control whether Group 1 interrupts are signaled to the processor.
- The FIQEn bit, that controls whether the interface signals Group 0 interrupts to the processor using the IRQ or FIQ interrupt request.
- The CBPR bit, that controls whether [GICC\\_BPR](#) or [GICC\\_ABPR](#) is used when determining possible interrupt preemption by Group 1 interrupts, see [Control of preemption by Group 1 interrupts on page 3-57](#).
- The AckCtl bit, that controls whether a read of the [GICC\\_IAR](#), or the Secure [GICC\\_IAR](#) if the GIC implements the Security Extensions, can acknowledge a Group 1 interrupt. For more information see [The effect of interrupt grouping on interrupt acknowledgement on page 3-50](#).

———— **Note** —————

As described in [The effect of interrupt grouping on interrupt acknowledgement on page 3-50](#), ARM deprecates setting the AckCtl bit to 1.

- In a GICv2 implementation:
  - the IRQ and FIQ bypass disable bits, that control whether the bypass IRQ and FIQ signals are forwarded to the processor, see [Interrupt signal bypass, and GICv2 bypass disable on page 2-27](#).
  - The EOImode bit, that controls whether priority drop is separated from interrupt deactivation, see [Priority drop and interrupt deactivation on page 3-38](#). If the GIC implements the Security Extensions, separate EOImodeNS and EOImodeS bits are implemented for Non-secure and Secure accesses. This provides independent control of the End of interrupt mode for Non-secure and Secure interrupt handling.

### 3.4.2 Special interrupt numbers when a GIC supports interrupt grouping

[Special interrupt numbers on page 3-43](#) describes the use of interrupt ID 1023 to indicate a *spurious interrupt*. The full list of the interrupt ID numbers the GIC architecture reserves for special purposes is as follows:

- 1020-1021** Reserved.
- 1022** Used only if the GIC supports interrupt grouping.  
The GIC returns this value to a processor in response to an interrupt acknowledge only when all of the following apply:
- the interrupt acknowledge is a read of [GICC\\_IAR](#)
  - the highest priority pending interrupt is a Group 1 interrupt
  - [GICC\\_CTLR.AckCtl](#) is set to 0
  - the priority of the interrupt is sufficient for it to be signaled to the processor.
- **Note** —————
- Interrupt ID 1022 indicates that there is a Group 1 interrupt of sufficient priority to be signaled to the processor, that must be acknowledged by a read of the [GICC\\_AIAR](#), or in an implementation that includes the GIC Security Extensions, by a read of the Non-secure [GICC\\_IAR](#).
  - When using a GICv1 implementation, in this situation Secure software on a processor might alter its schedule to permit Non-secure software to handle the interrupt, to minimize the interrupt latency.
- 
- 1023** This value is returned to a processor, in response to an interrupt acknowledge, if there is no pending interrupt with sufficient priority for it to be signaled to the processor.

On a processor that supports interrupt grouping, values of 1022 and 1023 are spurious interrupt IDs.

### 3.4.3 The effect of interrupt grouping on interrupt acknowledgement

In a GIC implementation that does not support interrupt grouping, when a processor takes an interrupt, it acknowledges the interrupt by reading the [GICC\\_IAR](#), see [General handling of interrupts on page 3-37](#). This read of the [GICC\\_IAR](#) always acknowledges the highest priority pending interrupt for the processor performing the read.

In a GIC implementation that supports interrupt grouping, ARM strongly recommends setting [GICC\\_CTLR.AckCtl](#) to 0, meaning:

- for a GICv2 implementation:
  - a group 0 interrupt is acknowledged by a read of [GICC\\_IAR](#), or a Secure read of [GICC\\_IAR](#) if the implementation includes the GIC Security Extensions
  - a group 1 interrupt is acknowledged by a read of [GICC\\_AIAR](#), or a Non-secure read of [GICC\\_IAR](#) if the implementation includes the GIC Security Extensions
- for a GICv1 implementation:
  - a group 0 interrupt must be acknowledged by a read of the Secure [GICC\\_IAR](#)
  - a group 1 interrupt must be acknowledged by a read of Non-secure [GICC\\_IAR](#).

In each case, the read must be an acknowledgement of the highest priority pending interrupt on the CPU interface.

For more information about the registers used for interrupt handling, see [GIC interrupt grouping support on page 3-48](#).

If the Interrupt Acknowledge register access does not correspond to the highest-priority pending interrupt on the CPU interface then:

- a read of [GICC\\_IAR](#) when the highest-priority pending interrupt is a Group 1 interrupt returns the spurious interrupt value 1022

- a read of [GICC\\_AIAR](#) when the highest-priority pending interrupt is a Group 0 interrupt returns the spurious interrupt value 1023.

When the [GICC\\_CTLR.AckCtl](#) bit is set to 0, to ensure system correctness, every Group 0 interrupt must have a higher priority than any Group 1 interrupt.

When the [GICC\\_CTLR.AckCtl](#) bit is set to 1, a read of [GICC\\_IAR](#) acknowledges the highest-priority pending interrupt on the CPU interface, regardless of whether it is a Group 0 or a Group 1 interrupt. However, ARM deprecates this use of [GICC\\_CTLR.AckCtl](#), and strongly recommends using a software model where [GICC\\_CTLR.AckCtl](#) is set to 0.

### Interrupt acknowledgement with the GIC Security Extensions

This subsection describes how the requirements for acknowledging grouped interrupts apply to interrupt handling when a processor that implements the ARM processor Security Extensions is connected to a GIC CPU interface that included the GIC Security Extensions. In this configuration:

- Group 0 interrupts are Secure interrupts
- Group 1 interrupts are Non-secure interrupts.

The subsection only describes operation with [GICC\\_CTLR.AckCtl](#) set to 0, the recommended configuration.

If the highest priority pending interrupt is a Secure interrupt, the processor must make a Secure read of the [GICC\\_IAR](#) to acknowledge it.

To acknowledge a Non-secure interrupt, the processor can:

- perform a Non-secure read of the [GICC\\_IAR](#) register
- in a GICv2 implementation, perform a Secure read of the [GICC\\_AIAR](#) register.

This means that, when Non-secure software is handling a Non-secure interrupt, the processor makes a Non-secure read of the [GICC\\_IAR](#) to acknowledge a Non-secure interrupt.

If a read of the [GICC\\_IAR](#) does not match the security of the interrupt, the [GICC\\_IAR](#) read does not acknowledge any interrupt and returns the value:

- 1022 for a Secure read when the highest priority interrupt is Non-secure
- 1023 for a Non-secure read when the highest priority interrupt is Secure.

See [Effect of interrupt grouping on reads of the GICC\\_IAR on page 4-136](#) for more information.

#### 3.4.4 GIC power on or reset configuration

On power-up, or after a reset, a GIC implementation that supports interrupt grouping is configured with:

- all interrupts assigned to Group 0
- the FIQ exception request disabled.

This means that Group 0 interrupts are signaled using the IRQ interrupt request. [Figure 3-2 on page 3-52](#) shows this configuration.

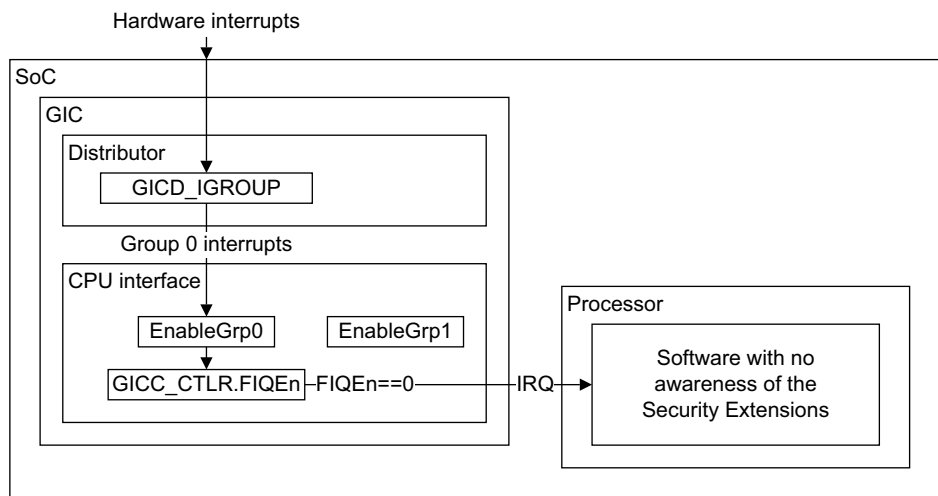


Figure 3-2 Reset configuration of a GIC that includes the FIQ exception request

## 3.5 Interrupt grouping and interrupt prioritization

Many system implementations require that no Group 1 interrupt ever preempt any Group 0 interrupt. For such an implementation, ARM strongly recommends that:

- Group 0 interrupts are always assigned priority values in the lower half of the supported priority value range. These values correspond to the higher-priority interrupts
- Group 1 interrupts are always assigned priority values in the upper half of the supported priority value range. These values correspond to the lower-priority interrupts.

This ensures that every Group 1 interrupt is of lower priority than any Group 0 interrupt.

If the GIC supports the GIC Security Extensions:

- The GIC provides Secure and Non-secure views of the interrupt priority settings, see [Software views of interrupt priority in a GIC that includes the Security Extensions](#).
- The minimum number of priority values supported increases from 16 to 32.
- Non-secure accesses can see only half of the supported priority values. Therefore, if the GIC implements 32 priority values, Non-secure accesses see only 16 priority values.

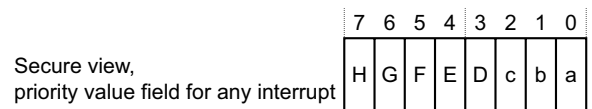
———— **Note** —————

See [Processor security state and Secure and Non-secure GIC accesses on page 1-20](#) for the definitions of Secure software and Secure and Non-secure accesses.

### 3.5.1 Software views of interrupt priority in a GIC that includes the Security Extensions

When a processor reads the priority value of a Group 1 interrupt, the GIC returns either the Secure or the Non-secure view of that value, depending on whether the access is Secure or Non-secure. This section describes the two views of interrupt priority, and the relationship between them.

The GIC implements a minimum of 32 and a maximum of 256 priority levels. This means it implements 5-8 bits of the 8-bit priority value fields in the `GICD_IPRIORITYRn` registers. All of the implemented priority bits can be accessed by a Secure access, and unimplemented low-order bits of the priority fields are RAZ/WI. [Figure 3-3](#) shows the Secure view of a priority value field for an interrupt. The priority value stored in the Distributor is equivalent to the Secure view.

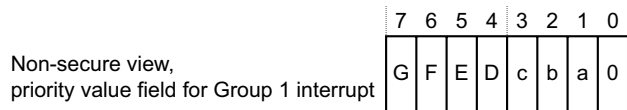


**Figure 3-3 Secure view of the priority field for any interrupt**

In this view:

- bits H-D are the bits that the GIC must implement, corresponding to 32 priority levels
- bits c-a are the bits the GIC might implement, that are RAZ/WI if not implemented.
- the GIC must implement bits H-a to provide the maximum 256 priority levels
- ARM recommends that, for a Group 1 interrupt, bit[7] is set to 1.

A Non-secure access can only see a priority value field that corresponds to the Non-secure view of interrupt priority. For Non-secure accesses, the GIC supports half the priority levels it supports for Secure accesses. [Figure 3-4 on page 3-54](#) shows the Non-secure view of a priority value field for a Group 1 interrupt.



**Figure 3-4 Non-secure view of the priority field for a Group 1 interrupt**

In this view:

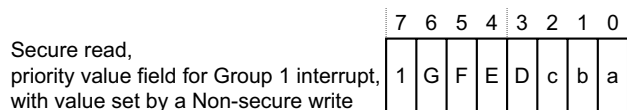
- bits G-D are the bits that the GIC must implement, corresponding to 16 priority levels
- bits c-a are the bits the GIC might implement, that are RAZ/WI if not implemented
- the GIC must implement bits G-a to provide the maximum 128 priority levels
- bit [0] is RAZ/WI.

The Non-secure view of a priority value does not show how the value is stored in the Distributor. Taking the value from a Non-secure write to a priority field, before storing the value:

- the value is right-shifted by one bit
- bit [7] of the value is set to 1.

This translation means the priority value for the Group 1 interrupt is in the top half of the possible value range, meaning the interrupt priority is in the bottom half of the priority range.

A Secure read of the priority value for an interrupt returns the value stored in the Distributor. [Figure 3-5](#) shows this Secure view of the priority value field for a Group 1 interrupt that has had its priority value field set by a Non-secure access, or has had a priority value with bit [7] == 1 set by a Secure access:



**Figure 3-5 Secure read of the priority field for a Group 1 interrupt**

A Secure write to the priority value field for a Group 1 interrupt can set bit [7] to 0, but see [Recommendations for managing priority values on page 3-56](#). If a Secure write sets bit [7] to 0:

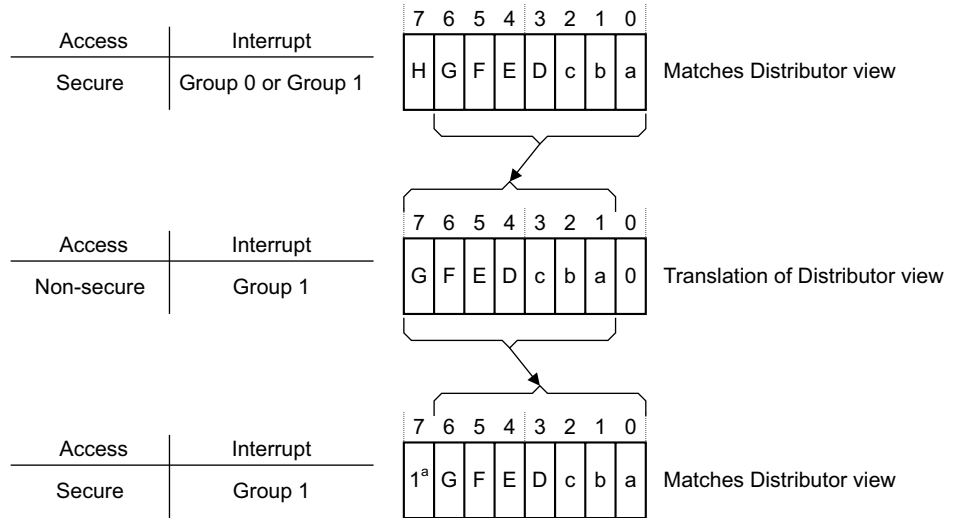
- A Non-secure read returns the value 0bGFEDcba0.
- A Non-secure write can change the value of the field, but only to a value that has bit [7] set to 1 in the Distributor view of the field.

**Note**

This behavior of Non-secure accesses applies only to the Priority value fields in the [GICD\\_IPRIORITYRn](#):

- if the Priority field in the [GICC\\_PMR](#) holds a value with bit [7] == 0, then the field is RAZ/WI to Non-secure accesses
- if the Priority field in the [GICC\\_RPR](#) holds a value with bit [7] == 0, then the field is RAZ to Non-secure reads.

[Figure 3-6 on page 3-55](#) shows the relationship between the views of the Priority value fields.

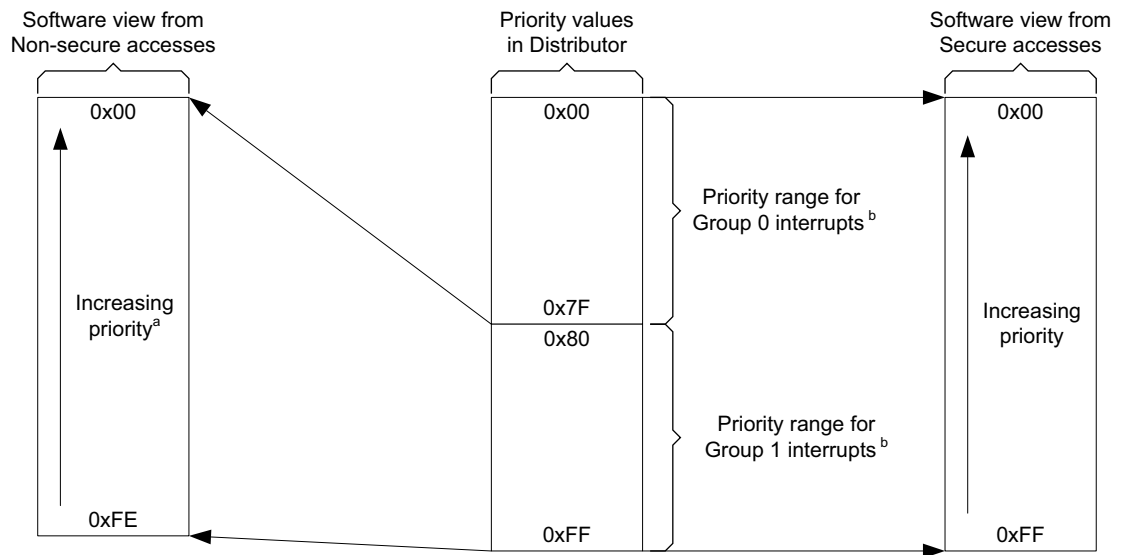


a If the priority value was set by a Non-secure write, bit [7] is set to 1 in the Distributor, and a Secure read sees this value. A Secure write to the field can set this bit to 0, see text for how this affects Non-secure accesses to the field.

The priority field for a Group 0 interrupt is RAZ/WI to Non-secure accesses.

**Figure 3-6 Relationship between Secure and Non-secure views of interrupt priority fields**

Figure 3-7 shows how the software views of the interrupt priorities, from Secure and Non-secure accesses, relate to the priority values held in the Distributor, and the interrupt value that are visible to Secure and Non-secure accesses. This is for a GIC that implements the maximum range of priority values.



a All priority values are even (bit [0] == 0) in the view from Non-secure accesses.

b Ranges recommended by ARM. See text for more information, including about cases where these ranges might not be appropriate.

**Figure 3-7 Software views of the priorities of Group 1 and Group 0 interrupts**

Table 3-6 shows how the number of priority value bits implemented by the GIC affects the Secure and Non-secure views of the priority of a Group 1 interrupt.

———— **Note** —————

Non-secure software has no visibility of the priority settings of Group 0 interrupts.

**Table 3-6 Effect of not implementing some priority field bits, with GIC Security Extensions**

Implemented priority bits, as seen in Secure view	Possible priority field values, for a Group 1 interrupt	
	Secure view	Non-secure view
[7:0]	0xFF-0x00 (255-0), all values	0xFE-0x00 (254-0), even values only
[7:1]	0xFE-0x00 (254-0), even values only	0xFC-0x00 (252-0), in steps of 4
[7:2]	0xFC-0x00 (252-0), in steps of 4	0xF8-0x00 (248-0), in steps of 8
[7:3]	0xF8-0x00 (248-0), in steps of 8	0xF0-0x00 (240-0), in steps of 16

This model for the presentation of priority values ensures software written to operate with an implementation of this GIC architecture functions as intended regardless of whether the GIC implements the GIC Security Extensions. However, programmers must ensure that software assigns appropriate priority levels to the Group 0 and Group 1 interrupts. See *Priority management and the GIC Security Extensions on page 3-60* for more information.

For more information about priority-related register access restrictions associated with the GIC Security Extensions, see the pseudocode in *Interrupt generation when the GIC supports interrupt grouping on page 3-58*.

### Recommendations for managing priority values

ARM strongly recommends that:

- for a Group 0 interrupt, software sets bit [7] of the priority value field to 0
- if using a Secure write to set the priority of a Group 1 interrupt, software sets bit [7] of the priority value field to 1.

This ensures that all Group 0 interrupts have lower priority values, and therefore higher priorities, than all Group 1 interrupts. However, a system might have requirements that cannot be met with this scheme, see *Priority management and the GIC Security Extensions on page 3-60*.

———— **Note** —————

- When both the GIC and the connected processor include the Security Extensions, Group 0 interrupts are Secure interrupts, and Group 1 interrupts are Non-secure interrupts.
- Software might not have any awareness of the GIC Security Extensions, and therefore might not know whether it is making Secure or Non-secure accesses to GIC registers. However, for any implemented interrupt, software can write 0xFF to the corresponding `GICD_IPRIORITYRn` priority value field, and then read back the value stored in the field to determine the supported interrupt priority range. ARM recommends that, before checking the priority range in this way:
  - for a peripheral interrupt, software first disables the interrupt
  - for an SGI, software first checks that the interrupt is inactive.



### 3.5.2 Control of preemption by Group 1 interrupts

See [Preemption on page 3-45](#) and [Priority grouping on page 3-45](#) for more information about preemption.

When a GIC implementation supports interrupt grouping, the `GICC_BPR` is always used to determine whether a Group 0 interrupt is signaled to the processor, for possible preemption. By default, the `GICC_ABPR` is used to determine whether a Group 1 interrupt is signaled for possible preemption. However, when `GICC_CTLR.CBPR` is set to 1, `GICC_BPR` is used for determining possible preemption, for both Group 0 and Group 1 interrupts.

#### Effect of the GIC Security Extensions on control of preemption by Group 1 interrupts

If the GIC implementation includes the Security Extensions:

- the CBPR bit is implemented only in the Secure copy of `GICC_CTLR`.
- it is the Secure copy of `GICC_BPR` that is:
  - always used to determine whether Group 0 interrupts are signaled to the processor
  - when `GICC_CTLR.CBPR` is set to 1, also used to determine whether Group 0 interrupts are signaled
- `GICC_ABPR` is an alias of the Non-secure copy of `GICC_CTLR`
- `GICC_ABPR` is a Secure register, accessible only by Secure software accesses.

### 3.5.3 The effect of interrupt grouping on priority grouping

When an interrupt is using the `GICC_ABPR`, the effective binary point value is one less than that stored in the register, as [Table 3-7](#) shows. This means that software with no awareness of the effects of interrupt grouping and the GIC Security Extensions sees the same priority grouping mechanism regardless of whether it is running on a processor that is in Secure or Non-secure state.

#### ———— Note ————

- In GICv2, the effective binary point value adjustment also occurs in GIC implementations that do not include the Security Extensions.
- Priority grouping always operates on the priority value held in the Distributor, not the value visible to a Non-secure read of the priority value corresponding to a Non-secure interrupt. See [Figure 3-6 on page 3-55](#) and [Figure 3-7 on page 3-55](#).

The minimum binary point value supported for the `GICC_ABPR` register is:

- IMPLEMENTATION DEFINED
- in the range 1-4
- one greater than the minimum value supported for the Secure copy of the `GICC_BPR` register.

[Table 3-7](#) shows the resultant priority grouping for Group 1 interrupts when `GICC_CTLR.CBPR==0`.

**Table 3-7 Priority grouping for Group 1 interrupts when `GICC_CTLR.CBPR==0`**

GICC_ABPR value	Interrupt priority field [7:0]		
	Group priority field	Subpriority field	Field with binary point <sup>a</sup>
0 <sup>b</sup>	-	-	-.
1	[7:1] <sup>c</sup>	[0]	HGFEDcb.s
2	[7:2] <sup>c</sup>	[1:0]	HGFEDc.ss
3	[7:3] <sup>c</sup>	[2:0]	HGFED.sss
4	[7:4] <sup>c</sup>	[3:0]	HGFE.ssss
5	[7:5] <sup>c</sup>	[4:0]	HGF.sssss

**Table 3-7 Priority grouping for Group 1 interrupts when GICC\_CTLR.CBPR==0 (continued)**

GICC_ABPR value	Interrupt priority field [7:0]		
	Group priority field	Subpriority field	Field with binary point <sup>a</sup>
6	[7:6] <sup>c</sup>	[5:0]	HG.ssssss
7	[7] <sup>c</sup>	[6:0]	H.ssssss

- a. Group labelling aligns with that shown in [Figure 3-6 on page 3-55](#).
- b. Not supported.
- c. If a Non-secure write sets the priority value field for a Non-secure interrupt then bit [7] is 1.

For Group 0 interrupts, the priority grouping behavior is as described in [Priority grouping on page 3-45](#).

In a GIC implementation that includes the Security Extensions, when `GICC_CTLR.CBPR == 1`:

- A Non-secure read of the `GICC_BPR` returns the value of the Secure `GICC_BPR`, incremented by 1, and saturated to `0b111`.
- Non-secure writes to `GICC_BPR` are ignored
- the `GICC_ABPR` register is redundant.

### 3.5.4 Interrupt generation when the GIC supports interrupt grouping

The pseudocode in [Exception generation pseudocode, with interrupt grouping on page 3-64](#) describes the generation of interrupts by the GIC when the GIC supports interrupt grouping.

## 3.6 Additional features of the GIC Security Extensions

*The effect of interrupt grouping on interrupt handling on page 3-48 and [Interrupt grouping and interrupt prioritization on page 3-53](#) describe many features of the GIC Security Extensions, especially for a GICv1 implementation, where interrupt grouping is supported only as part of the GIC Security Extensions. This section describes the other features of the GIC Security Extensions.*

Software can detect support for the GIC Security Extensions by reading the `GICD_TYPER.SecurityExtn` bit, see [Interrupt Controller Type Register, `GICD\_TYPER` on page 4-88](#).

### ———— Note —————

In the context of a GIC that implements the GIC Security Extensions connected to a processor that implements the ARM Security Extensions, Group 0 interrupts are Secure interrupts, and Group 1 interrupts are Non-secure interrupts. See [Security Extensions support on page 1-16](#) for more information.

In addition:

- The banking of registers provides independent control of Secure and Non-secure interrupts, see [Effect of the GIC Security Extensions on the programmers' model on page 4-80](#).
- The Non-secure copy of the `GICC_BPR` is aliased as the `GICC_ABPR`. This is a Secure register, meaning it is only accessible by Secure accesses.

### 3.6.1 Access from processors not implementing the ARM Security Extensions

When connecting a processor that does not support the ARM Security Extensions to a GIC that implements the GIC Security Extensions, typically all processor accesses to the GIC are assigned as either Secure or Non-secure:

- For a processor making Secure accesses:
  - The processor can control all aspects of the GIC, and therefore can make configuration changes that might affect Secure software running on other processors.
  - In a GICv2 implementation, the processor uses Secure accesses to aliased registers, such as the `GICC_AIAR`, to process Group 1 interrupts.
  - Because GICv1 implementations do not include the aliased registers, if the implementation uses interrupt grouping the processor might have to use the deprecated `GICC_CTLR.AckCtl` bit to enable Group 1 interrupts to be processed using the standard CPU interface registers.
- For a processor making Non-secure accesses:
  - The processor cannot control Group 0 interrupts. For the GIC to be programmed, the system implementation must include at least one processor that can make Secure accesses. A system might use a Secure processor to perform Secure accesses on behalf of a Non-secure processor. This usage model is possible if the GIC or the system provides a method for the Secure processor to access processor-banked copies of registers that belong to the Non-secure processor.
  - To permit a Non-secure processor to control its own Group 0 interrupts, a GICv2 implementation can implement the `GICD_NSACRn` registers. An implementation of these registers might permit a Secure processor to permit the use of Non-secure accesses from a particular processor to control some aspects of the operation of some Group 0 SGIs and SPIs.
  - A GIC implementation can configure the `GICD_IGROUPRn` reset value so that interrupts are Group 1 on reset. see [GICD\\_IGROUPR0 reset value on page 4-92](#) for more information.

### 3.6.2 The effect of the GIC Security Extensions on priority masking

This section describes how the GIC Security Extensions change the information given in [Priority masking on page 3-45](#).

If the GIC implements the GIC Security Extensions, the `GICC_PMR` is RAZ/WI to Non-secure accesses if it holds a value with bit [7] == 0. In normal operation, Non-secure software does not access the `GICC_PMR` when it is programmed with such a value. For more information see [Non-secure access to register fields for Group 0 interrupt priorities on page 4-81](#).

### 3.6.3 Priority management and the GIC Security Extensions

A system that implements the GIC Security Extensions can use the following schemes for managing interrupt priority:

- |                        |  |
|------------------------|--|
| <b>Non-cooperative</b> | All Secure interrupts have higher priority than any Non-secure interrupt, and can always preempt any Non-secure interrupt.     |
| <b>Co-operative</b>    | Secure and Non-secure software interact to program some Secure interrupts with lower priority than some Non-secure interrupts. |

Secure software is software executing on a processor that implements the ARM Security Extensions, that can make Secure accesses to the GIC, and might be able to make Non-secure accesses. Non-secure software can make only Non-secure accesses.

Where Secure software manipulates the priority level of a Non-secure interrupt, normally it ensures bit [7] of the priority value field is set to 1, so that the priority of the interrupt is in the lower half of the implemented range. However, it might have to program the priority level of a Non-secure interrupt to a value in the upper half of the implemented priority range, for example to manage an SGI from Non-secure software that targets a processor that executes only Secure software.

Secure software can also set the priority of a Secure interrupt to a value in the lower half of the implemented priority range, so that it has lower priority than some Non-secure interrupts.

———— **Note** —————

- Setting the priority of a Secure interrupt in the lower half of the priority range provides an opportunity for security attacks, such as denial of service. Secure software must consider the possibility of attacks of this kind before setting a Secure interrupt priority to a value in the priority range visible to Non-secure software.
- The GIC architecture does not require all processors in the system to use the same scheme for managing interrupt priority.

## 3.7 Pseudocode details of interrupt handling and prioritization

The following sections provide pseudocode descriptions of interrupt handling and prioritization, with and without the GIC Security Extensions, and describe the accesses to the registers that control prioritization in a system that implements the GIC Security Extensions:

- [General helper functions and definitions](#)
- [Exception generation pseudocode on page 3-64](#)
- [The effect of the GIC Security Extensions on accesses to prioritization registers on page 3-66.](#)

### 3.7.1 General helper functions and definitions

The following pseudocode provides helper functions and definitions used elsewhere in the GIC pseudocode:

```
// Helper functions
// =====

SignalFIQ(boolean next_fiq, integer cpu_id) // Signals an interrupt on the FIQ input to the
                                           // processor, according to the value of next_fiq.

SignalIRQ(boolean next_irq, integer cpu_id) // Signals an interrupt on the IRQ input to the
                                           // processor, according to the value of next_irq.

boolean IsGrp0Int(integer InterruptID, cpu_id)
                                           // Returns TRUE if the field in the GICD_IGROUPRn
                                           // register associated with the argument InterruptID
                                           // is set to 0, indicating that the interrupt is
                                           // configured as a Group 0 interrupt.

boolean IsEnabled(integer InterruptID, cpu_id)
                                           // Returns TRUE if the interrupt specified by the
                                           // argument InterruptID is enabled in the associated
                                           // GICD_ISENBLERn or GICD_ICENBLERn register.

bits(3) SGI_CpuID(integer InterruptID, cpu_id)
                                           // Returns the ID of a source CPU for a pending interrupt
                                           // with the given interruptID targeting the current
                                           // CPU. If there are multiple source CPUs, the one
                                           // chosen is IMPLEMENTATION SPECIFIC.

bits(8) ReadGICD_ITARGETSR(integer InterruptID, cpu_id)
                                           // Returns an 8-bit field specifying which CPUs should
                                           // receive the interrupt specified by argument InterruptID

boolean AnyActiveInterrupts(integer cpu_id) // Returns TRUE if any interrupts are active on this
                                           // processor.

bits(8) ReadGICD_IPRIORITYR(integer InterruptID, cpu_id)
                                           // Returns the 8-bit priority field from the
                                           // GICD_IPRIORITYR associated with the argument InterruptID.

WriteGICD_IPRIORITYR(integer InterruptID, cpu_id, bits(8) Value)
                                           // Updates the priority field in the GICD_IPRIORITYR
                                           // associated with the argument InterruptID with the 8-bit
                                           // Value.

IgnoreWriteRequest() // Ignore the register write request (no operation).

AcknowledgeInterrupt(integer InterruptID, cpu_id)
                                           // Set the active state and attempt to clear the pending
                                           // state for the interrupt associated with the argument
                                           // InterruptID

// Global variables
// =====

integer cpu_id // An identifier for a specific CPU Interface. The value of this
```

### 3 Interrupt Handling and Prioritization

#### 3.7 Pseudocode details of interrupt handling and prioritization

```
// variable has implicit effects on which CPU interface register,
// CPU interface signal or banked version of a Distributor
// register is accessed.

boolean NS_access // current GIC access state:
// TRUE: Non-secure
// FALSE: Secure.

// NOTE: Architected registers are considered global variables identified
// by their architecture mnemonic, and as such are not declared here.

// global constants
// =====

integer MINIMUM_BINARY_POINT // A minimum binary point value of 0,1,2 or 3,
// this is an IMPLEMENTATION DEFINED value.
// NOTE: min. value is the SECURE value where supported

boolean IGNORE_GROUP_ENABLE // IMPLEMENTATION DEFINED boolean that determines whether the
// highest priority pending interrupt is masked by the distributor
// enable BEFORE or AFTER prioritisation:
//
// BEFORE prioritisation Value = FALSE
// AFTER prioritisation Value = TRUE

boolean GICC_MASK_HPPAIR // IMPLEMENTATION DEFINED boolean that determines whether a read
// of GICC_HPPAIR returns a spurious interrupt for pending
// interrupts disabled by GICC_CTLR.EnableGrp{0,1} == '0'

bits(8) P_MASK // IMPLEMENTATION DEFINED mask of valid priority bits:
// Consists of an 8-bit field where the top N bits are set to 1,
// where N is the number of priority bits implemented.
// For systems without the Security Extensions, supported
// values are 0xF0, 0xF8, 0xFC, 0xFE and 0xFF.
// For systems with the Security Extensions, supported
// values are 0xF8, 0xFC, 0xFE and 0xFF.

// PriorityIsHigher()
// =====

boolean PriorityIsHigher(bits(8) pr1, bits(8) pr2)
return UInt(pr1) < UInt(pr2); // Lower number represents higher priority.

// GIC_PriorityMask()
// =====

// NOTE: where the Security Extensions are not supported, NS_mask = '0'

bits(8) GIC_PriorityMask(integer n, bit NS_mask) // Calculate the Binary Point (group) mask.
assert n >= 0 && n <= 7; // Range check for the priority mask.

if NS_mask == '1' then // Mask generation for a secure GIC access.
n = n - 1;

// CHECK:
if n < MINIMUM_BINARY_POINT then // Saturate n on the minimum value supported; range 0-3
n = MINIMUM_BINARY_POINT; // NOTE: min. value is the SECURE value where supported

mask = '11111110000000' <14-n:7-n>; // Generate the 8-bit group priority mask.
return mask;

// boolean IsPending()
// =====
//
// Returns TRUE if the interrupt specified by argument interruptID
// is pending for the CPU specified by argument cpuID
//
```

```

boolean IsPending(integer interruptID, integer cpuID)
    pending = FALSE;

    target_cpus = ReadGICD_ITARGETSR(interruptID);

    if PEND && !ACTIVE(interruptID) && target_cpus<cpuID> == '1' then
        pending = TRUE;

    return pending;

// HighestPriorityPendingInterrupt()
// =====
//
// Returns the ID of the highest priority interrupt that is pending and enabled.
// Otherwise, returns 1023 (i.e. a spurious interrupt)
//
// In implementations where interrupts are masked by the distributor group enable bits AFTER
// prioritisation (i.e. IGNORE_GROUP_ENABLE is TRUE), this function may return the ID of a pending
// interrupt in a disabled group even though there is a (lower priority) pending interrupt that is
// fully enabled (i.e. enabled in GICD_IENABLER and the appropriate group enable bit is '1' in
// GICD_CTLR). This is a helper function only and does not explain the full effect of GICC_HPPIR.
// The value returned by a read of GICC_HPPIR is explained in the pseudocode provided with the
// register description.

bits(10) HighestPriorityPendingInterrupt(integer cpu_id)

    num_interrupts = 32 * (UInt(GICD_TYPER<4:0>) + 1); // Work out how many interrupts are supported

    hppi= 1023; // Set initial ID to be no interrupt pending

    for intID = 0 to num_interrupts - 1
        group_enabled = ( IsGrp0Int(intID) && (GICD_CTLR.EnableGrp0 == '1')) ||
            (!IsGrp0Int(intID) && (GICD_CTLR.EnableGrp1 == '1'));

        if IsPending(intID, cpu_id) && IsEnabled(intID) then
            if group_enabled || IGNORE_GROUP_ENABLE then
                if PriorityIsHigher(ReadGICD_IPRIORITYR(intID), ReadGICD_IPRIORITYR(hppi)) then
                    hppi = intID;

    return(hppi);

```

### 3.7.2 Exception generation pseudocode

Interrupt grouping, and the GIC Security Extensions, make the exception generation model significantly more complicated:

- [Exception generation pseudocode, with interrupt grouping](#) describes exception generation by a GIC implementation that supports interrupt grouping, and might include the Security Extensions
- [Exception generation pseudocode, when interrupt grouping is not supported on page 3-65](#) describes the simplified exception generation model for a GIC implementation that does not support interrupt grouping.

#### Exception generation pseudocode, with interrupt grouping

The following pseudocode describes how exceptions are generated by a CPU interfaces that implement the GIC Security Extensions:

```
// GenerateExceptions()
// =====
//

GIC_GenerateExceptions(
    boolean systemFIQ,
    boolean systemIRQ)

while TRUE do // Loop continuously.
    cpu_count = UInt(GICD_TYPER<7:5>) + 1; // Determine the number of CPU interfaces.

    for cpu_id = 0 to cpu_count - 1 // Loop though CPU interfaces. The iterations of
        // this loop are permitted to occur in parallel.

        (next_int, next_grp0) = UpdateExceptionState(cpu_id); // Returns pending interrupts, masked
        // by distributor enables but not cpu i/f enables

        irq_wake = FALSE; // IRQ wake up signal to power management, if required
        fiq_wake = FALSE; // FIQ wake up signal to power management, if required

        cpu_irq = FALSE; // IRQ signal to CPU
        cpu_fiq = FALSE; // FIQ signal to CPU

        if (next_int) then
            if (next_grp0 && GICC_CTLR[cpu_id].FIQEn == '1') then
                fiq_wake = TRUE;
                if (GICC_CTLR[cpu_id].EnableGrp0 == '1') then
                    cpu_fiq = TRUE;
            else
                irq_wake = TRUE;
                if ( next_grp0 && GICC_CTLR[cpu_id].EnableGrp0 == '1' ||
                    !next_grp0 && GICC_CTLR[cpu_id].EnableGrp1 == '1')
                then
                    cpu_irq = TRUE;

// Optional bypass logic
//
if GICC_CTLR[cpu_id].EnableGrp0 == '0' || GICC_CTLR[cpu_id].FIQEn == '0'
then
    if GICC_CTLR[cpu_id].FIQBypDisGrp0 == '0' ||
        (GICC_CTLR[cpu_id].FIQBypDisGrp1 == '0' && GICC_CTLR[cpu_id].FIQEn == '0')
    then
        cpu_fiq = systemFIQ; // Set FIQ to bypass

if GICC_CTLR[cpu_id].EnableGrp1 == '0' &&
    (GICC_CTLR[cpu_id].EnableGrp0 == '0' || GICC_CTLR[cpu_id].FIQEn == '1')
then
    if GICC_CTLR[cpu_id].IRQBypDisGrp1 == '0' ||
        (GICC_CTLR[cpu_id].IRQBypDisGrp0 == '0' && GICC_CTLR[cpu_id].FIQEn == '1')
    then
        cpu_irq = systemIRQ; // Set IRQ to bypass
```



```
//
// End, optional bypass logic

        SignalFIQ(cpu_fiq, cpu_id);           // Update driven status of FIQ.
        SignalIRQ(cpu_irq, cpu_id);         // Update driven status of IRQ.

// UpdateExceptionState()
// =====
//
(boolean, boolean) UpdateExceptionState(integer cpu_id)

    sbp = UInt(GICC_BPR[cpu_id]<2:0>);           // Secure version of this register.
    nsbp = UInt(GICC_ABPR[cpu_id]<2:0>);

    next_int = FALSE;
    next_grp0 = FALSE;

    intID = HighestPriorityPendingInterrupt(cpu_id); // Establish the ID of the highest pending
                                                    // interrupt on the this CPU interface.

    if PriorityIsHigher(ReadGICD_IPRIORITYR(intID), GICC_PMR[cpu_id]<7:0>) &&
        IsPending(intID, cpu_id)
    then
        smask = GIC_PriorityMask(sbp, '0');
        if GICC_CTLR[cpu_id].CBPR == '1' then
            nsmask = smask;
        else
            nsmask = GIC_PriorityMask(nsbp, '1');

        if IsGrp0Int(intID) && // Highest pending interrupt is secure
            (GICD_CTLR.EnableGrp0 == '1') // and secure interrupts are enabled
        then
            if !AnyActiveInterrupts() ||
                PriorityIsHigher(ReadGICD_IPRIORITYR(intID), GICC_RPR[cpu_id]<7:0> AND smask)
            then
                next_int = TRUE;
                next_grp0 = TRUE;
        else
            if (!IsGrp0Int(intID)) && // Highest pending interrupt is non-secure
                (GICD_CTLR.EnableGrp1 == '1') // and non-secure interrupts are enabled
            then
                if !AnyActiveInterrupts() ||
                    PriorityIsHigher(ReadGICD_IPRIORITYR(intID), GICC_RPR[cpu_id]<7:0> AND nsmask)
                then
                    next_int = TRUE;
                    next_grp0 = FALSE;

    return(next_int, next_grp0);
```

### Exception generation pseudocode, when interrupt grouping is not supported

The following pseudocode describes how exceptions are generated by a GIC that does not support interrupt grouping. This means it applies only to a GICv1 implementation that does not include the Security Extensions.

```
// GenerateExceptions()
// =====
//
GIC_GenerateExceptions()
    while TRUE do // Loop continuously.
        cpu_count = UInt(GICD_TYPER<7:5>) + 1; // Determine the number of CPU interfaces.

        for cpu_id = 0 to cpu_count - 1 // Loop though CPU interfaces. The iterations of
                                        // this loop are permitted to occur in parallel.
            next_irq = UpdateExceptionState(cpu_id);
```

```

        SignalIRQ(next_irq, cpu_id);           // Update driven status of IRQ.

// UpdateExceptionState()
// =====
//

boolean UpdateExceptionState(integer cpu_id)

    next_irq = FALSE;

    intID = HighestPriorityPendingInterrupt(cpu_id); // Establish the ID of the highest pending
                                                    // interrupt on the this CPU interface.

    if PriorityIsHigher(ReadGICD_IPRIORITYR(intID), GICC_PMR[cpu_id]<7:0>) &&
        IsPending(intID, cpu_id)
    then
        if GICD_CTLR.Enable == '1' && GICC_CTLR.Enable == '1' then
            mask = GIC_PriorityMask(GICC_BPR[cpu_id]<2:0>, '0');

            if !AnyActiveInterrupts() ||
                PriorityIsHigher(ReadGICD_IPRIORITYR(intID), GICC_RPR[cpu_id]<7:0> AND mask)
            then
                next_irq = TRUE;

    return(next_irq);

```

### 3.7.3 The effect of the GIC Security Extensions on accesses to prioritization registers

The GIC Security Extensions change some of the behavior of accesses to the prioritization registers. See the pseudocode functions in:

- [Interrupt Priority Registers, GICD\\_IPRIORITYRn](#) on page 4-104
- [Interrupt Priority Mask Register, GICC\\_PMR](#) on page 4-131
- [Binary Point Register, GICC\\_BPR](#) on page 4-133
- [Interrupt Acknowledge Register, GICC\\_IAR](#) on page 4-135
- [Running Priority Register, GICC\\_RPR](#) on page 4-142
- [Highest Priority Pending Interrupt Register, GICC\\_HPPIR](#) on page 4-143.

See [Non-secure access to register fields for Group 0 interrupt priorities](#) on page 4-81 for more information.

## 3.8 The effect of the Virtualization Extensions on interrupt handling

In general, the Virtualization Extensions have no effect on how the GIC handles and prioritizes physical interrupts. See [Chapter 5 GIC Support for Virtualization](#) for information about how the GIC Virtualization Extensions support virtual interrupt handling.

### 3.9 Example GIC usage models

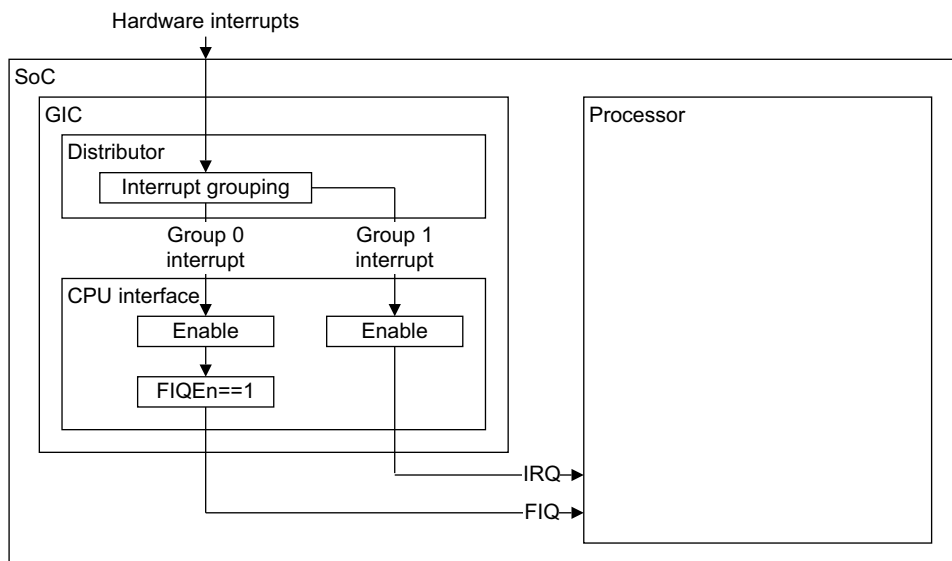
An ARM processor that implements the ARMv7-A or ARMv7-R architecture profile supports two interrupt request signals, **nIRQ** and **nFIQ**, each with an associated exception and processor mode:

- Asserting an IRQ request generates an IRQ exception. By default, this is taken in IRQ mode, and taking the exception masks subsequent IRQ exceptions.
- Asserting an FIQ request generates an FIQ exception. By default, this is taken in FIQ mode, and taking the exception masks both FIQ and IRQ exceptions.

The following sections describe different GIC usage models, that meet specific system requirements:

- [Using IRQs and FIQs to provide Non-secure and Secure interrupts](#)
- [Supporting IRQs and FIQs when not using the processor Security Extensions on page 3-70.](#)
- [Supporting IRQs and FIQs in a virtualized processor environment on page 3-71.](#)

All of these usage model examples use the hardware implementation shown in [Figure 3-8](#), with a GIC that supports Group 0 and Group 1 interrupts.



**Figure 3-8 Generic GIC usage model**

In each usage model, software uses the `GICD_IGROUPRn` registers to assign interrupts to the two groups, signaled to the processor using the IRQ and FIQ interrupt requests.

———— **Note** ————

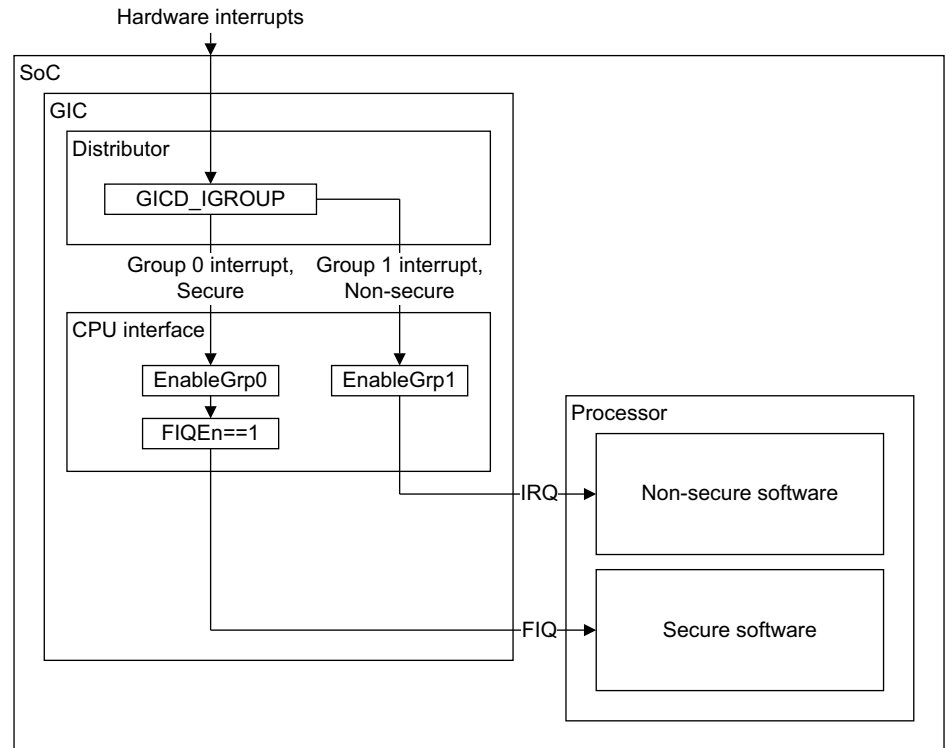
The usage model described in [Supporting IRQs and FIQs in a virtualized processor environment on page 3-71](#) also requires the GIC to implement the GIC Virtualization Extensions.

#### 3.9.1 Using IRQs and FIQs to provide Non-secure and Secure interrupts

[Figure 3-9 on page 3-69](#) shows a system that implements the GIC Security Extensions, connected to a processor that implements the ARM processor Security Extensions. This implementation:

- uses Group 0 interrupts as Secure interrupts, signaled as FIQs
- uses Group 1 interrupts as Non-secure interrupts, signaled as IRQs.

This means that, on the processor, FIQ interrupts are never routed to Non-secure software, and IRQ interrupts are never routed to Secure software.



**Figure 3-9 Using the GIC to route Secure and Non-secure interrupts**

**Note**

The use of Group 0 and Group 1 interrupts to signal Secure interrupts as FIQs, and Non-secure interrupts as IRQs, requires the processor to:

- route FIQs to be taken in Secure Monitor mode
- prevent Non-secure software from masking FIQs
- ensure that IRQs are masked whenever it is operating in Secure state.

**Controlling Secure and Non-secure interrupts independently**

The system shown in Figure 3-9 fulfils the general security requirement that Non-secure operation must not interfere with Secure operation. Secure software takes full control of FIQs by routing them to the Secure software and not permitting the Non-secure software to mask them.

On a GIC reset, all interrupts are assigned to Group 0, making them Secure interrupts. Secure software on the processor:

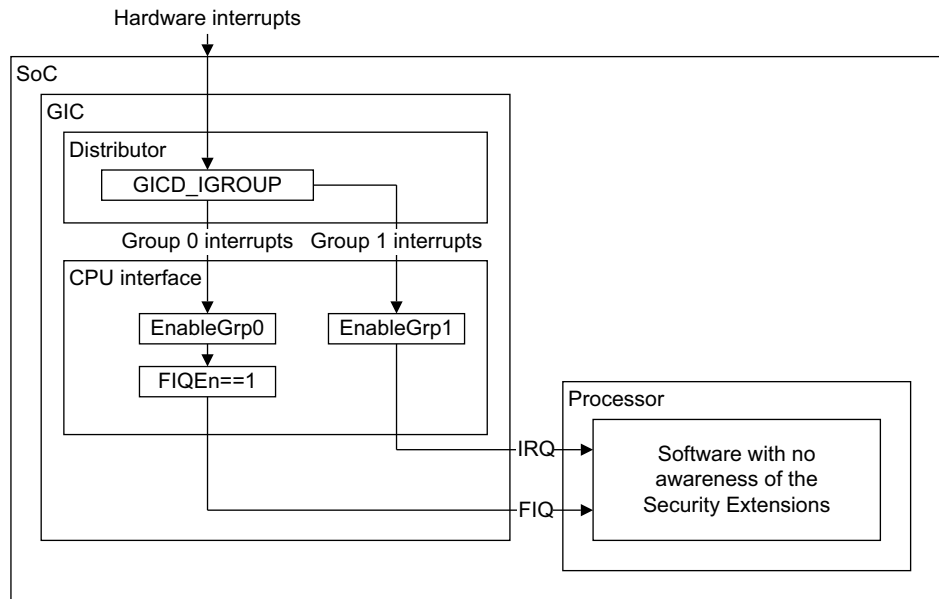
- programs the **GICD\_IGROUPRn** registers to indicate which interrupts are Group 1, Non-secure
- sets the Secure **GICC\_CTLR.FIQEn** bit to 1 to configure the CPU interface to use FIQ for Group 0 interrupts.
- must enable Group 0 interrupts and Group 1 interrupts, independently, in the Distributor:
  - **GICD\_CTLR.EnableGrp0** enables Group 0 interrupts
  - **GICD\_CTLR.EnableGrp1** enables Group 1 interrupts.
- must enable Group 0 interrupts and Group 1 interrupts, independently, in the CPU interface:
  - **GICC\_CTLR.EnableGrp0** enables Group 0 interrupts
  - **GICC\_CTLR.EnableGrp1** enables Group 1 interrupts.

### 3.9.2 Supporting IRQs and FIQs when not using the processor Security Extensions

Figure 3-10 shows a system in which the processor does not implement, or is not using, the Processor Security Extensions. This system can use the interrupt grouping provided by the `GICD_IGROUPn` registers to control both IRQs and FIQs, based on:

- assigning FIQs to interrupt Group 0
- assigning IRQs to interrupt Group 1.

This section applies to any GICv1 implementation that includes the GIC Security Extensions, or any GICv2 implementation.



**Figure 3-10 Using interrupt grouping to route IRQs and FIQs**

On a GIC reset, for a GIC implementation that supports interrupt grouping, all interrupts are assigned to Group 0. Therefore, to use this configuration, software executing on the processor must:

- Program the `GICD_IGROUPn` registers to assign IRQ interrupts to Group 1.

———— **Note** ————

For GICv2 implementations that do not include the Security Extensions, the `GICD_IGROUPn` reset values are IMPLEMENTATION DEFINED, see *Interrupt Group Registers, GICD\_IGROUPn* on page 4-91.

- Set `GICC_CTLR.FIQEn` to 1, to assign Group 0 interrupts to FIQ.
- Set `GICC_CTLR.AckCtl` to 0, so that both FIQ and IRQ interrupts are acknowledged from the single address space, using:
  - the `GICC_IAR` to acknowledge a Group 0 interrupt
  - the `GICC_AIAR` to acknowledge a Group 1 interrupt
  - the `GICC_EOIR` to indicate completion of a Group 0 interrupt
  - the `GICC_AEOIR` to indicate completion of a Group 1 interrupt.

However, `GICC_AIAR` and `GICC_AEOIR` are implemented only in a GICv2 implementation. A processor operating with a GICv1 implementation might have to use the deprecated mode of operation with `GICC_CTLR.AckCtl` set to 1.

- Configure the required binary point support model, by either:
  - setting `GICC_CTLR.CBPR` to 0, so that Group 0 uses `GICC_BPR`, and Group 1 uses `GICC_ABPR`

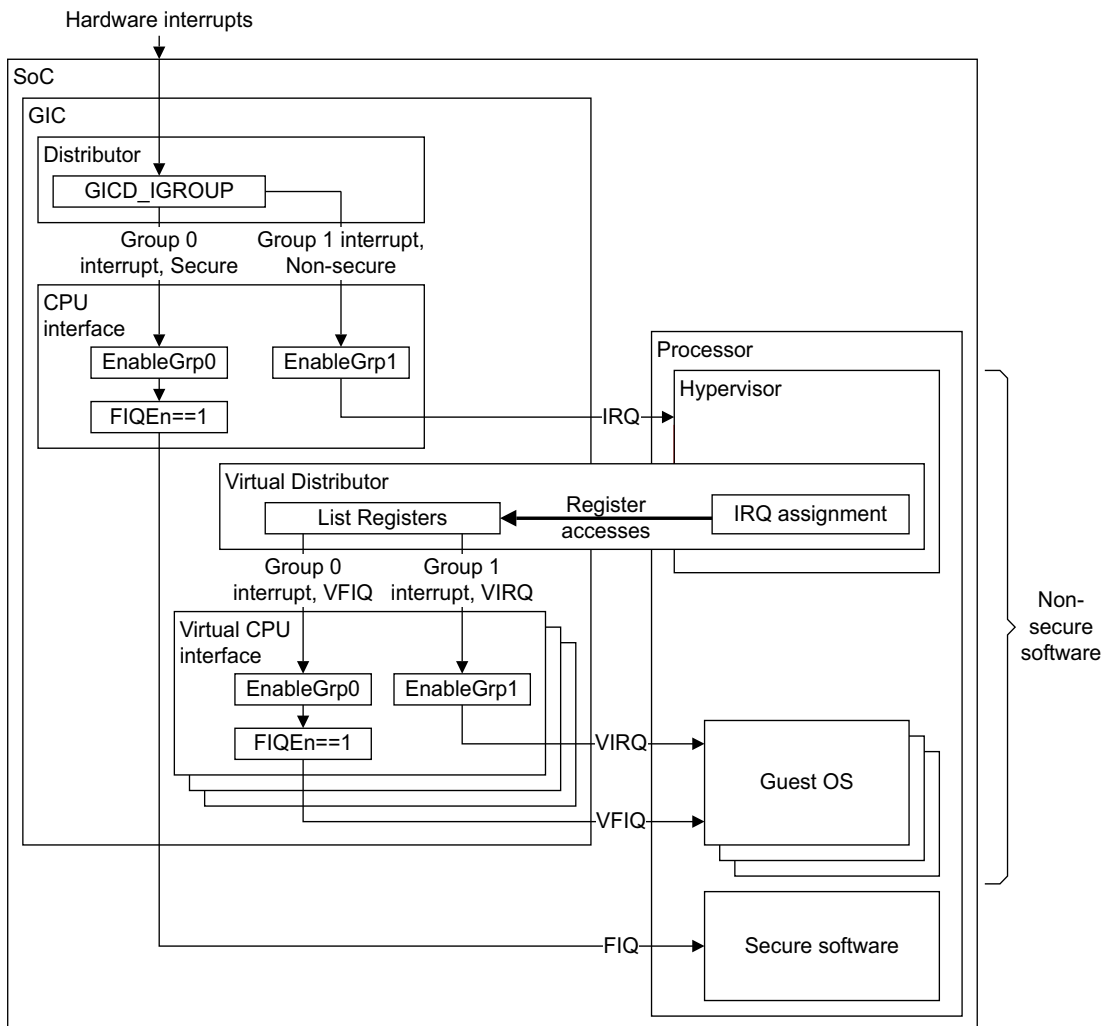
- setting `GICC_CTLR.CBPR` to 1, so that Group 0 and Group 1 use a common binary point register, `GICC_BPR`.

### 3.9.3 Supporting IRQs and FIQs in a virtualized processor environment

Figure 3-11 on page 3-72 shows a system that supports processor virtualization, with the execution of legacy software on virtual machines. The basis of the processor usage model is:

- Secure software assigns:
  - Secure interrupts to Group 0, signaled to the processor as FIQs
  - Non-secure interrupts to Group 1, signaled to the processor as IRQs.This is the usage model described in *Using IRQs and FIQs to provide Non-secure and Secure interrupts* on page 3-68.
- A hypervisor:
  - Implements a *virtual distributor*, using features of the Virtualization Extension on the GIC. This virtual distributor can virtualize IRQ interrupts from the GIC as Virtual IRQ and Virtual FIQ interrupts, that it routes to an appropriate virtual machine.
  - Routes physical IRQs to Hyp mode, so they can be serviced by the virtual distributor.
- A Guest OS running on a virtual machine assigns interrupts to Group 0 or Group 1, to assign them as FIQs or IRQs, using the model described in *Supporting IRQs and FIQs when not using the processor Security Extensions* on page 3-70. The accesses to the GIC Distributor registers are trapped to the hypervisor, and therefore access the virtual distributor.

The virtual CPU interface signals these interrupts as virtual FIQs or virtual IRQs. This virtualization is under the control of the hypervisor and is invisible to the Guest OS.



VFIQ is Virtual FIQ, VIRQ is Virtual IRQ

**Figure 3-11 Using the GIC in a virtualized system**

When the GIC signals an IRQ to the processor, the interrupt is routed to Hyp mode. The hypervisor determines whether the interrupt is for itself, or for a Guest OS. If it is for a Guest OS it determines:

- which Guest OS must handle the interrupt
- whether that Guest OS has configured the interrupt as an FIQ or as an IRQ
- the interrupt priority, based on the priority configuration by the target Guest OS.

If the interrupt targets the current Guest OS, the hypervisor updates the List registers, to add the interrupt to the list of pending interrupts for the current virtual machine.

**Note**

- On receiving an IRQ that cannot be handled by the current Guest OS, the hypervisor can either:
  - transfer control to a Guest OS that can handle the interrupt
  - mark the interrupt as pending, as part of the saved context of the appropriate Guest OS.
- A system can have some interrupts that can be handled by more than one Guest OS, and other interrupts that must be routed to a specific Guest OS.

A Guest OS handles a virtual interrupt exactly as it would handle the corresponding physical interrupt. The Guest OS cannot detect that it is handling a virtual interrupt rather than a physical interrupt.



# Chapter 4

## Programmers' Model

This chapter describes the Distributor and CPU interface registers. It contains the following sections:

- *About the programmers' model* on page 4-74
- *Effect of the GIC Security Extensions on the programmers' model* on page 4-80
- *Distributor register descriptions* on page 4-84
- *CPU interface register descriptions* on page 4-124
- *Preserving and restoring GIC state* on page 4-155.

## 4.1 About the programmers' model

The programmers' model provides the software interface to the GIC. This chapter describes the programmers' model for the GIC Distributor and CPU interfaces, that operates using a memory-mapped register interface.

The following sections describe the programmers' model:

- [GIC register names](#)
- [Distributor register map](#)
- [CPU interface register map on page 4-76](#)
- [GIC register access on page 4-77](#)
- [Enabling and disabling the Distributor and CPU interfaces on page 4-77](#)
- [Effect of the GIC Security Extensions on the programmers' model on page 4-80.](#)

[Table 4-1 on page 4-75](#) and [Table 4-2 on page 4-76](#) describe the register access type as follows:

<b>RW</b>	Read and write.
<b>RO</b>	Read only. Writes are ignored.
<b>WO</b>	Write only. Reads return an UNKNOWN value.

---

### Note

This section does not describe the programmers' model for the GIC virtual interface control registers and the virtual CPU interfaces, that the GIC Virtualization Extensions add to a GIC implementation. See [Chapter 5 GIC Support for Virtualization](#) for the description of the additions to the programmers' model in a GIC that implements the GIC Virtualization Extensions.

---

### 4.1.1 GIC register names

All of the GIC registers have names that provide a short mnemonic for the function of the register. In these names:

- the first three letters are GIC, indicating a GIC register
- the fourth letter is one of:
  - D, indicating a Distributor register
  - C, indicating a CPU interface register
  - H, indicating a virtual interface control register, typically accessed by a hypervisor
  - V, indicating a virtual CPU interface register.
- the remaining letters are a mnemonic for the register, for example the GIC Distributor Control Register is called [GICD\\_CTLR](#).

---

### Note

[Chapter 5 GIC Support for Virtualization](#) describes the [GICH\\_\\*](#) and [GICV\\_\\*](#) registers.

---

### 4.1.2 Distributor register map

[Table 4-1 on page 4-75](#) shows the Distributor register map. Address offsets are relative to the *Distributor base address* defined by the system memory map. All GIC registers are 32-bits wide. Reserved register addresses are RAZ/WI.

---

### Note

For more information about legacy register names, see [Appendix B Register Names](#).

---

Table 4-1 Distributor register map

Offset	Name	Type	Reset <sup>a</sup>	Description
0x000	<a href="#">GICD_CTLR</a>	RW	0x00000000	Distributor Control Register
0x004	<a href="#">GICD_TYPER</a>	RO	IMPLEMENTATION DEFINED	Interrupt Controller Type Register
0x008	<a href="#">GICD_IIDR</a>	RO	IMPLEMENTATION DEFINED	Distributor Implementer Identification Register
0x00C-0x01C	-	-	-	Reserved
0x020-0x03C	-	-	-	IMPLEMENTATION DEFINED registers
0x040-0x07C	-	-	-	Reserved
0x080	<a href="#">GICD_IGROUPR<sub>n</sub></a> <sup>b</sup>	RW	IMPLEMENTATION DEFINED <sup>c</sup>	Interrupt Group Registers
0x084-0x0FC			0x00000000	
0x100-0x17C	<a href="#">GICD_ISENBALER<sub>n</sub></a>	RW	IMPLEMENTATION DEFINED	Interrupt Set-Enable Registers
0x180-0x1FC	<a href="#">GICD_ICENABLER<sub>n</sub></a>	RW	IMPLEMENTATION DEFINED	Interrupt Clear-Enable Registers
0x200-0x27C	<a href="#">GICD_ISPENDR<sub>n</sub></a>	RW	0x00000000	Interrupt Set-Pending Registers
0x280-0x2FC	<a href="#">GICD_ICPENDR<sub>n</sub></a>	RW	0x00000000	Interrupt Clear-Pending Registers
0x300-0x37C	<a href="#">GICD_ISACTIVER<sub>n</sub></a> <sup>d</sup>	RW	0x00000000	GICv2 Interrupt Set-Active Registers
0x380-0x3FC	<a href="#">GICD_ICACTIVER<sub>n</sub></a> <sup>e</sup>	RW	0x00000000	Interrupt Clear-Active Registers
0x400-0x7F8	<a href="#">GICD_IPRIORITYR<sub>n</sub></a>	RW	0x00000000	Interrupt Priority Registers
0x7FC	-	-	-	Reserved
0x800-0x81C	<a href="#">GICD_ITARGETSR<sub>n</sub></a>	RO <sup>f</sup>	IMPLEMENTATION DEFINED	Interrupt Processor Targets Registers
0x820-0xBF8		RW <sup>f</sup>	0x00000000	
0xBFC	-	-	-	Reserved
0xC00-0xFC	<a href="#">GICD_ICFGR<sub>n</sub></a>	RW	IMPLEMENTATION DEFINED	Interrupt Configuration Registers
0xD00-0xDFC	-	-	-	IMPLEMENTATION DEFINED registers
0xE00-0xEFC	<a href="#">GICD_NSACR<sub>n</sub></a> <sup>e</sup>	RW	0x00000000	Non-secure Access Control Registers, optional
0xF00	<a href="#">GICD_SGIR</a>	WO	-	Software Generated Interrupt Register
0xF04-0xF0C	-	-	-	Reserved
0xF10-0xF1C	<a href="#">GICD_CPENDSGIR<sub>n</sub></a> <sup>e</sup>	RW	0x00000000	SGI Clear-Pending Registers
0xF20-0xF2C	<a href="#">GICD_SPENDSGIR<sub>n</sub></a> <sup>e</sup>	RW	0x00000000	SGI Set-Pending Registers
0xF30-0xFCC	-	-	-	Reserved
0xFD0-0xFFC	-	RO	IMPLEMENTATION DEFINED	<i>Identification registers on page 4-119</i>

- For details of any restrictions that apply to the reset values of IMPLEMENTATION DEFINED cases see the appropriate register description.
- In a GICv1 implementation, present only if the GIC implements the GIC Security Extensions, otherwise RAZ/WI.
- For more information see [GICD\\_IGROUPR0 reset value on page 4-92](#).
- In GICv1, these are the Active Bit Registers, ICDABR<sub>n</sub>. These registers are RO.

- e. GICv2 only.
- f. In a uniprocessor implementation, these registers are RAZ/WI.

### 4.1.3 CPU interface register map

Table 4-2 shows the CPU interface register map. Address offsets are relative to the *CPU interface base address* defined by the system memory map. All GIC registers are 32-bits wide. Reserved register addresses are RAZ/WI.

For a multiprocessor implementation, the GIC implements a set of CPU interface registers for each CPU interface. ARM strongly recommends that each processor has the same CPU interface base address for the CPU interface that connects it to the GIC. This is the private CPU interface base address for that processor. It is IMPLEMENTATION DEFINED whether a processor can access the CPU interface registers of other processors in the system.

#### Note

For more information about:

- the registers added by the GIC Virtualization Extensions, see [Chapter 5 GIC Support for Virtualization](#)
- legacy register names, see [Appendix B Register Names](#).

Table 4-2 CPU interface register map

Offset	Name	Type	Reset	Description
0x0000	<a href="#">GICC_CTLR</a>	RW	0x00000000	CPU Interface Control Register
0x0004	<a href="#">GICC_PMR</a>	RW	0x00000000	Interrupt Priority Mask Register
0x0008	<a href="#">GICC_BPR</a>	RW	0x0000000x <sup>a</sup>	Binary Point Register
0x000C	<a href="#">GICC_IAR</a>	RO	0x000003FF	Interrupt Acknowledge Register
0x0010	<a href="#">GICC_EOIR</a>	WO	-	End of Interrupt Register
0x0014	<a href="#">GICC_RPR</a>	RO	0x000000FF	Running Priority Register
0x0018	<a href="#">GICC_HPPIR</a>	RO	0x000003FF	Highest Priority Pending Interrupt Register
0x001C	<a href="#">GICC_ABPR</a> <sup>b</sup>	RW	0x0000000x <sup>a</sup>	Aliased Binary Point Register
0x0020	<a href="#">GICC_AIAR</a> <sup>c</sup>	RO	0x000003FF	Aliased Interrupt Acknowledge Register
0x0024	<a href="#">GICC_AEOIR</a> <sup>c</sup>	WO	-	Aliased End of Interrupt Register
0x0028	<a href="#">GICC_AHPPIR</a> <sup>c</sup>	RO	0x000003FF	Aliased Highest Priority Pending Interrupt Register
0x002C-0x003C	-	-	-	Reserved
0x0040-0x00CF	-	-	-	IMPLEMENTATION DEFINED registers
0x00D0-0x00DC	<a href="#">GICC_APRn</a> <sup>c</sup>	RW	0x00000000	Active Priorities Registers
0x00E0-0x00EC	<a href="#">GICC_NSAPRn</a> <sup>c</sup>	RW	0x00000000	Non-secure Active Priorities Registers
0x00ED-0x00F8	-	-	-	Reserved
0x00FC	<a href="#">GICC_IIDR</a>	RO	IMPLEMENTATION DEFINED	CPU Interface Identification Register
0x1000	<a href="#">GICC_DIR</a> <sup>c</sup>	WO	-	Deactivate Interrupt Register

a. See the register description for more information.

b. Present in GICv1 if the GIC implements the GIC Security Extensions. Always present in GICv2.

c. GICv2 only.

#### 4.1.4 GIC register access

All registers support 32-bit word accesses with the access type defined in [Table 4-1 on page 4-75](#) and [Table 4-2 on page 4-76](#).

In addition, the `GICD_IPRIORITYRn`, `GICD_ITARGETSRn`, `GICD_CPENDSGIRn`, and `GICD_SPENDSGIRn` registers support byte accesses.

Whether any halfword register accesses are permitted is IMPLEMENTATION DEFINED.

———— **Note** —————

In the GIC architecture, all registers that are halfword-accessible or byte-accessible use a little endian memory order model.

If the GIC implements the GIC Security Extensions these affect register accesses as follows:

- some registers are banked, see [Register banking](#)
- some registers are accessible only using Secure accesses
- optionally, the GIC supports lockdown of the values of some registers.

For more information see [Effect of the GIC Security Extensions on the programmers' model on page 4-80](#).

#### Register banking

Register banking refers to providing multiple copies of a register at the same address. The properties of a register access determine which copy of the register is addressed. The GIC banks registers in the following cases:

- If the GIC implements the Security Extensions, some registers are banked to provide separate Secure and Non-secure copies of the registers. The Secure and Non-secure register bit assignments can differ. A Secure access to the register address accesses the Secure copy of the register, and a Non-secure access accesses the Non-secure copy. See [Effect of the GIC Security Extensions on the programmers' model on page 4-80](#) for more information.
- If the GIC is implemented as part of a multiprocessor system:
  - Some registers are banked to provide a separate copy for each connected processor. These include the registers associated with PPIs and SGIs, and the `GICD_NSACRn`, when implemented.
  - The GIC implements the CPU interface registers independently for each CPU interface, and each connected processor accesses these registers for the interface it connects to.

#### 4.1.5 Enabling and disabling the Distributor and CPU interfaces

This section describes how to enable and disable the Distributor and CPU interfaces, and the differences in behavior in an implementation that supports interrupt grouping. It describes:

- [Implementations that support interrupt grouping](#)
- [Implementations that do not support interrupt grouping on page 4-79](#).

#### Implementations that support interrupt grouping

Interrupt grouping is present in all GICv2 implementations and in GICv1 implementations that include the GIC Security Extensions,

In a GIC that supports interrupt grouping:

- the `GICD_CTLR.EnableGrp0` bit controls the forwarding of Group 0 interrupts from the Distributor to the CPU interfaces
- the `GICD_CTLR.EnableGrp1` bit controls the forwarding of Group 1 interrupts from the Distributor to the CPU interfaces
- the `GICC_CTLR.EnableGrp0` bit controls the signaling of Group 0 interrupts by the CPU interface to the processor

- the `GICC_CTLR.EnableGrp1` bit controls the signaling of Group 1 interrupts by the CPU interface to the processor.

For the Distributor:

- If the `GICD_CTLR.EnableGrp0` and `GICD_CTLR.EnableGrp1` bits are both 0:
  - the Distributor does not forward pending interrupts to the CPU interfaces
  - it is IMPLEMENTATION DEFINED whether an edge-triggered interrupt signal sets the interrupt to the pending state.
  - reads of `GICC_IAR`, `GICC_AIAR`, `GICC_HPPIR`, or `GICC_AHPPIR` return a spurious interrupt ID
  - software can read or write the Distributor registers
  - it is IMPLEMENTATION DEFINED whether SGIs can be set pending using `GICD_SGIR`
- If either, but not both, of the `GICD_CTLR.EnableGrp0` and `GICD_CTLR.EnableGrp1` bits is set to 1, and the highest priority pending interrupt is in the disabled group, the Distributor does not forward any pending interrupts to the CPU interfaces. Although this is IMPLEMENTATION DEFINED, this applies in the following cases:
  - `GICD_CTLR.EnableGrp0` set to 0 and `GICD_CTLR.EnableGrp1` set to 1, and the highest priority pending interrupt is in group 0
  - `GICD_CTLR.EnableGrp0` set to 1 and `GICD_CTLR.EnableGrp1` set to 0, and the highest priority pending interrupt is in group 1.

In an implementation that includes the GIC Security Extensions, this means that, in cases where there are Group 1 interrupts with a higher priority than some Group 0 interrupts, it is possible for Non-secure software to deny service to Secure software, by clearing the `GICD_CTLR.EnableGrp1` bit. To prevent this, ARM strongly recommends that all Group 0 interrupts are assigned a higher priority than all Group 1 interrupts.

In addition, to prevent Secure software from denying service to Non-secure software, Secure software must ensure that when `GICD_CTLR.EnableGrp1` is set to 1, either `GICD_CTLR.EnableGrp0` is also set to 1, or that there are no pending Group 0 interrupts.

See [Recommendations for managing priority values on page 3-56](#) for more information.

For a CPU interface, when `GICC_CTLR.AckCtl == 0`:

- When `GICC_CTLR.EnableGrp0 == 0`
  - Group 0 interrupts forwarded from the Distributor are not signaled to the processor
  - any read of `GICC_IAR` returns a spurious interrupt ID
- When `GICC_CTLR.EnableGrp0 == 1`, Group 0 interrupts forwarded from the Distributor are signaled to the processor.
- When `GICC_CTLR.EnableGrp1 == 0`
  - Group 1 interrupts forwarded from the Distributor are not signaled to the processor
  - any read of `GICC_AIAR` returns a spurious interrupt ID
- When `GICC_CTLR.EnableGrp1 == 1`, Group 1 interrupts forwarded from the Distributor are signaled to the processor
- if either `GICC_CTLR.EnableGrp0` or `GICC_CTLR.EnableGrp1` is set to 0, and there is a pending interrupt of sufficient priority in the disabled group, it is IMPLEMENTATION DEFINED whether a read of `GICC_HPPIR` returns the ID of that interrupt, or a spurious interrupt ID.

For a CPU interface, when `GICC_CTLR.AckCtl == 1`:

- When `GICC_CTLR.EnableGrp1 == 0`, any Non-secure read of `GICC_IAR` returns a spurious interrupt ID
- When `GICC_CTLR.EnableGrp0 == 0`:
  - if `GICC_CTLR.EnableGrp1 == 0`, any Secure read of `GICC_AIAR` returns a spurious interrupt ID

- if `GICC_CTLR.EnableGrp1 == 1`, Group 0 interrupts are ignored and `GICC_IAR` behaves as `GICC_AIAR`
- When `GICC_CTLR.EnableGrp1 == 0`, a Secure read of `GICC_AIAR` always returns a spurious interrupt ID
- if either `GICC_CTLR.EnableGrp0` or `GICC_CTLR.EnableGrp1` is set to 0, and there is a pending interrupt of sufficient priority in the disabled group, it is IMPLEMENTATION DEFINED whether a read of `GICC_HPPIR` returns the ID of that interrupt, or a spurious interrupt ID.

———— **Note** —————

ARM deprecates use of `GICC_CTLR.AckCtl`, and strongly recommends using a software model where `GICC_CTLR.AckCtl` is set to 0.

---

## Implementations that do not support interrupt grouping

———— **Note** —————

The only implementations that do not support interrupt grouping are GICv1 implementations that do not include the GIC Security Extensions.

---

In a GIC that does not support interrupt grouping:

- the `GICD_CTLR.Enable` bit controls the forwarding of interrupts from the Distributor to the CPU interfaces
- the `GICC_CTLR.Enable` bit controls the signaling of interrupts by the CPU interface to the connected processor.

For the Distributor:

- When `GICD_CTLR.Enable` is set to 1, the Distributor forwards the highest priority pending interrupt for each CPU interface, subject to the prioritization rules.
- When `GICD_CTLR.Enable` is set to 0:
  - the Distributor does not forward pending interrupts to the CPU interfaces
  - it is IMPLEMENTATION DEFINED whether an edge-triggered interrupt signal sets the interrupt to the pending state.
  - reads of `GICC_IAR`, `GICC_AIAR`, `GICC_HPPIR`, or `GICC_AHPPIR` return a spurious interrupt ID
  - software can read or write the Distributor registers
  - it is IMPLEMENTATION DEFINED whether SGIs can be set pending using `GICD_SGIR`.

For a CPU interface:

- When `GICC_CTLR.Enable` is set to 1, the highest priority pending interrupt forwarded from the Distributor to the CPU interface is signaled to the connected processor
- When `GICC_CTLR.Enable` is set to 0:
  - any pending interrupts forwarded from the Distributor are not signaled to the processor
  - software can read or write the CPU interface registers
  - any read of the `GICC_IAR` returns a spurious interrupt ID
  - if the Distributor is forwarding an interrupt to the CPU interface, that the interface cannot signal because `GICC_CTLR.Enable` is set to 0, it is IMPLEMENTATION DEFINED whether a read of `GICC_HPPIR` returns the ID of that interrupt, or a spurious interrupt ID.

———— **Note** —————

The `EnableGrp1` bit in the Non-secure copies of the `GICD_CTLR` and `GICC_CTLR` registers are cleared to 0 on reset. This means that software can program the Distributor and CPU interface registers before enabling the GIC.

---

See *Distributor Control Register, GICD\_CTLR* on page 4-85 and *CPU Interface Control Register, GICC\_CTLR* on page 4-125 for more information.

## 4.2 Effect of the GIC Security Extensions on the programmers' model

---

### Note

For an overview of the GIC Security Extensions, see [Security Extensions support on page 1-16](#).

If the GIC implements the Security Extensions, the `GICD_TYPER.SecurityExtn` bit is RAO.

The GIC Security Extensions provide the following features:

- The GIC must support interrupt grouping, and:
  - the GIC might implement some interrupts as always Group 0, or as always Group 1
  - otherwise, software configures each interrupt as Group 0 or Group 1
  - some aspects of interrupt handling depend on whether interrupts are Group 0 or Group 1.
- Register implementations that are consistent with those on a processor that implements the ARM Security Extensions, with banked, Common, and Secure registers, as described in this section. The GIC Security Extensions recognise that register accesses are either Secure or Non-secure, see [Processor security state and Secure and Non-secure GIC accesses on page 1-20](#), and that the security level of the access can determine the required response.

---

### Note

- In a GICv1 implementation, interrupt grouping is a feature of the GIC Security Extensions. All GICv2 implementations include support for interrupt grouping, regardless of whether they include the GIC Security Extensions.
- When a processor that implements the ARM Security Extensions is connected to the GIC, Secure software executing on the processor usually accesses the GIC using only Secure accesses.

---

The *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* defines the following ARM Security Extensions register types:

**Banked** The device implements Secure and Non-secure copies of the register. The register bit assignments can differ in the Secure and Non-secure copies of a register. A Secure access always accesses the Secure copy of the register, and a Non-secure access always accesses the Non-secure copy.

---

### Note

The GIC can also bank registers when implemented as part of a multiprocessor system, where registers associated with PPIs or SGIs are banked to provide a separate copy for each connected processor.

**Secure** The register is accessible only from a Secure access. The address of a Secure register is RAZ/WI to any Non-secure access.

**Common** The register is accessible from both Secure and Non-secure accesses. The access permissions of some or all fields in the register might depend on whether the access is Secure or Non-secure.

In addition, in a GIC that implements the GIC Security Extensions, the priority range available for Group 1 interrupts is half the range available for Group 0 interrupts, see [Interrupt grouping and interrupt prioritization on page 3-53](#).

[Table 4-3](#) shows the registers that are implemented differently as part of the GIC Security Extensions. All registers not listed in [Table 4-3](#) are Common registers.

**Table 4-3 Registers implemented differently when the GIC includes the GIC Security Extensions**

Register	Type	Description	Effect
<a href="#">GICD_CTLR</a>	Banked	Distributor Control Register	Register is banked <sup>a</sup>
<a href="#">GICD_TYPER</a>	Common	Interrupt Controller Type Register	Adds the LSPI field
<a href="#">GICD_IGROUPRn</a>	Secure	Interrupt Group Registers	Register is Secure



**Table 4-3 Registers implemented differently when the GIC includes the GIC Security Extensions (continued)**

Register	Type	Description	Effect
<a href="#">GICD_SGIR</a>	Common	Software Generated Interrupt Register	Adds the NSATT bit
<a href="#">GICC_CTLR</a>	Banked	CPU Interface Control Register	Register is banked <sup>a</sup>
<a href="#">GICC_BPR</a>	Banked	Binary Point Register	Register is banked <sup>a</sup>
<a href="#">GICC_ABPR</a>	Secure	Aliased Binary Point Register	Register is Secure
<a href="#">GICC_AIAR</a>	Secure	Aliased Interrupt Acknowledge Register	Register is Secure
<a href="#">GICC_AEOIR</a>	Secure	Aliased End of Interrupt Register	Register is Secure
<a href="#">GICC_AHPPIR</a>	Secure	Aliased Highest Priority Pending Interrupt Register	Register is Secure
<a href="#">GICC_NSAPRn</a>	Secure	Non-secure Active Priorities Registers	Register is Secure

a. For more information, see [Register banking](#) on page 4-77.

The following sections give more information about the effect of the GIC Security Extensions on the GIC programmers' model:

- [Non-secure access to register fields for Group 0 interrupt priorities](#)
- [Configuration lockdown](#) on page 4-82.

#### 4.2.1 Non-secure access to register fields for Group 0 interrupt priorities

[Processor security state and Secure and Non-secure GIC accesses](#) on page 1-20 provides definitions of Secure software and Secure and Non-secure accesses.

The GIC Security Extensions support the use of Group 0 interrupts as Secure interrupts, and Group 1 interrupts as Non-secure interrupts. This means that the register fields associated with Group 0 interrupts are RAZ/WI to Non-secure accesses, and in addition:

##### Non-secure access to a priority field in the [GICD\\_IPRIORITYRn](#)

If the priority field corresponds to a Group 1 interrupt, the access operates as defined by the Non-secure view of interrupt priority, see [Software views of interrupt priority in a GIC that includes the Security Extensions](#) on page 3-53.

##### Non-secure access to the [GICC\\_PMR](#) and [GICC\\_RPR](#)

- If the current priority mask value is in the range 0x00-0x7F:
  - a read access returns the value 0x00
  - the GIC ignores a write access to the [GICC\\_PMR](#).
- If the current priority mask value is in the range 0x80-0xFF:
  - A read access returns the Non-secure view of the current value.
  - A write access to the [GICC\\_PMR](#) succeeds, based on the Non-secure view of the priority mask value written to the register. This means a Non-secure write cannot set a priority mask value in the range 0x00-0x7F.

The pseudocode in [The effect of the GIC Security Extensions on accesses to prioritization registers](#) on page 3-66 describes accesses to the [GICD\\_IPRIORITYRn](#), [GICC\\_PMR](#), and [GICC\\_RPR](#) when the GIC implements the Security Extensions.

## 4.2.2 Configuration lockdown

A GIC implementation that includes the GIC Security Extensions can implement configuration lockdown. This provides a control signal that the system can assert to prevent write access to:

- the register fields controlling a configured range of SPIs, when those SPIs are configured as Group 0 interrupts
- some configuration registers.

When the control signal is asserted, the affected register fields and registers are described as being *locked down*.

Lockdown is controlled by an active HIGH disable signal, **CFGSDISABLE**. That is, the system asserts **CFGSDISABLE** HIGH to disable write access to the register fields and registers.

The SPIs that can be locked down are called *lockable SPIs* (LSPIs). The number of LSPIs is IMPLEMENTATION DEFINED, between 0 and 31:

- If the GIC supports any LSPIs then the first possible LSPI has Interrupt ID 32
- The **GICD\_TYPER**.LSPI field defines the maximum number of LSPIs. If **GICD\_TYPER**.LSPI is greater than 0 then the possible LSPIs have interrupt IDs 32 to (31+(**GICD\_TYPER**.LSPI)).

————— **Note** —————

**GICD\_TYPER**.LSPI only defines the range of possible LSPIs. The GIC might not support all the interrupts in this range.

If **GICD\_TYPER**.LSPI is 0 lockdown is not supported. This means software cannot lockdown any control registers if the GIC does not implement any LSPIs.

When the SPI control fields and configuration registers are locked down, the GIC prevents write accesses to:

- The EnableGrp0 bit of the Secure copy of **GICD\_CTLR**.
- The following bits in the Secure copy of **GICC\_CTLR**:
  - EOImodeS
  - IRQBypDisGrp0
  - FIQBypDisGrp0
  - CBPR
  - FIQEn
  - AckCtl
  - EnableGrp0

See *CPU Interface Control Register, GICC\_CTLR* on page 4-125.

- Fields in the **GICD\_ISENBLEn**, **GICD\_ICENABLEn**, **GICD\_ISPENDRn**, **GICD\_ICPENDRn**, **GICD\_ISACTIVERn**, **GICD\_ICACTIVERn**, **GICD\_IPRIORITYn**, **GICD\_ITARGETSRn**, and **GICD\_ICFGRn** registers that correspond to Lockable SPIs that are configured as Group 0:
- Fields in the **GICD\_IGROUPRn** registers that correspond to lockable SPIs that are configured as Group 0. If a lockable SPI is reconfigured from Group 1 to Group 0 while **CFGSDISABLE** remains HIGH, the GIC prevents any more writes to **GICD\_IGROUPRn** fields that correspond to that SPI, and the SPI becomes locked.

The GIC ignores any write to a locked down register or register field.

————— **Note** —————

- ARM recommends that, during the system boot process, the system reads the **GICD\_TYPER**.LSPI field to find the number of lockable SPIs, programs the registers and register fields that can be locked down, and then asserts **CFGSDISABLE** HIGH. Normally, this means that the Secure boot sequence that follows a full system reset must run appropriate Secure configuration code.

- ARM strongly recommends that when **CFGSDISABLE** is first asserted HIGH during the system boot process, the system ensures **CFGSDISABLE** cannot be deasserted except during a processor power-down or reset sequence.
- 

### 4.2.3 Effect of the Virtualization Extensions on the programmers' model

The GIC Virtualization Extensions add the GIC virtual interface control registers and the virtual CPU interface registers to the programmers' model. See [Chapter 5 GIC Support for Virtualization](#) for more information.

## 4.3 Distributor register descriptions

The following sections describe the Distributor registers:

- *Distributor Control Register, GICD\_CTLR* on page 4-85
- *Interrupt Controller Type Register, GICD\_TYPER* on page 4-88
- *Distributor Implementer Identification Register, GICD\_IIDR* on page 4-90
- *Interrupt Group Registers, GICD\_IGROUPRn* on page 4-91
- *Interrupt Set-Enable Registers, GICD\_ISENABLERn* on page 4-93
- *Interrupt Clear-Enable Registers, GICD\_ICENABLERn* on page 4-95
- *Interrupt Set-Pending Registers, GICD\_ISPENDRn* on page 4-97
- *Interrupt Clear-Pending Registers, GICD\_ICPENDRn* on page 4-99
- *Interrupt Set-Active Registers, GICD\_ISACTIVERn* on page 4-102
- *Interrupt Clear-Active Registers, GICD\_ICACTIVERn* on page 4-103
- *Interrupt Priority Registers, GICD\_IPRIORITYRn* on page 4-104
- *Interrupt Processor Targets Registers, GICD\_ITARGETSRn* on page 4-106
- *Interrupt Configuration Registers, GICD\_ICFGRn* on page 4-109
- *Non-secure Access Control Registers, GICD\_NSACRn* on page 4-111
- *Software Generated Interrupt Register, GICD\_SGIR* on page 4-113
- *SGI Clear-Pending Registers, GICD\_CPENDSGIRn* on page 4-115
- *SGI Set-Pending Registers, GICD\_SPENDSGIRn* on page 4-117
- *Identification registers* on page 4-119.

See *Distributor register map* on page 4-74 for address offset and reset information for these registers.

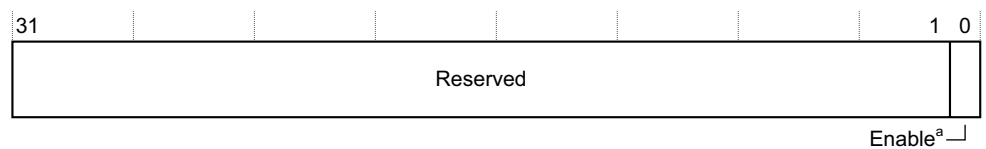
### 4.3.1 Distributor Control Register, GICD\_CTLR

The GICD\_CTLR characteristics are:

- Purpose** Enables the forwarding of pending interrupts from the Distributor to the CPU interfaces.
- Usage constraints** If the GIC implements the Security Extensions with configuration lockdown, the system can lock down the Secure GICD\_CTLR, see [Configuration lockdown on page 4-82](#).
- Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions, this register is banked, see [Register banking on page 4-77](#).
- Attributes** See the register summary in [Table 4-1 on page 4-75](#).

Figure 4-1 and Table 4-4 show the GICD\_CTLR bit assignments for:

- a GICv1 implementation that does not include the GIC Security Extensions
- the Non-secure copy of the register in an implementation that includes the GIC Security Extensions.



<sup>a</sup> Bit name is IMPLEMENTATION DEFINED in an implementation that includes the Security Extensions

**Figure 4-1 GICD\_CTLR bit assignments, GICv1 without Security Extensions or Non-secure**

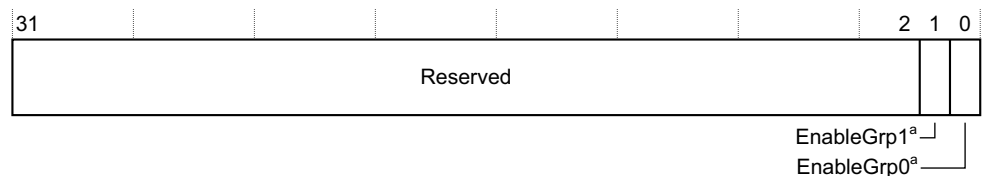
**Table 4-4 GICD\_CTLR bit assignments, GICv1 without Security Extensions or Non-secure**

Bits	Name	Function
[31:1]	-	Reserved.
[0]	Enable <sup>a</sup>	Global enable for forwarding pending interrupts from the Distributor to the CPU interfaces. In the Non-secure copy of this register in an implementation that includes the Security Extensions, this bit controls only the forwarding of Group 1 interrupts: <b>0</b> interrupts not forwarded. <b>1</b> interrupts forwarded, subject to the priority rules. See <a href="#">Enabling and disabling the Distributor and CPU interfaces on page 4-77</a> for more information about this bit.

a. Bit name is IMPLEMENTATION DEFINED in a GICv1 implementation that includes the Security Extensions.

Figure 4-2 and Table 4-5 on page 4-86 shows the GICD\_CTLR bit assignments for:

- Any GICv2 implementation. If the implementation includes the Security Extensions then these assignments apply only to the Secure copy of the register.
- The Secure copy of the register in a GICv1 implementation that includes the Security Extensions.



<sup>a</sup> In a GICv1 implementation that includes the Security Extensions:  
 - Bit[0] is named Enable  
 - Bit[1] is IMPLEMENTATION DEFINED.

**Figure 4-2 GICD\_CTLR bit assignments, GICv2, and GICv1 Secure copy**

**Table 4-5 GICD\_CTLR bit assignments, GICv2, and GICv1 Secure copy**

Bits	Name	Function
[31:2]	-	Reserved.
[1]	EnableGrp1	Global enable for forwarding pending Group 1 interrupts from the Distributor to the CPU interfaces: <b>0</b> Group 1 interrupts not forwarded. <b>1</b> Group 1 interrupts forwarded, subject to the priority rules.
<hr/> <b>Note</b> <hr/>		
In a GICv1 implementation that includes the Security Extensions: <ul style="list-style-type: none"> <li>• Whether this bit is implemented, and the bit name if implemented, is IMPLEMENTATION DEFINED. If not implemented the bit is reserved.</li> <li>• When the bit is implemented, it is an alias of bit[0] of the Non-secure copy of the register.</li> </ul>		
[0]	EnableGrp0	Global enable for forwarding pending Group 0 interrupts from the Distributor to the CPU interfaces: <b>0</b> Group 0 interrupts not forwarded. <b>1</b> Group 0 interrupts forwarded, subject to the priority rules.

When any of the Distributor global enable bits are set to 0, disabling the Distributor functions, other GIC register read and writes still operate normally. This means software can change the state of PPIs and SPIs before re-enabling the Distributor. For example, software can:

- Make an interrupt pending by writing to the corresponding [GICD\\_ISPENDRn](#).
- Remove the active state from an interrupt by writing to the corresponding [GICC\\_EOIR](#) or [GICC\\_AEOIR](#).

---

**Note**

---

Setting a Distributor global enable bit to 0 disables forwarding of interrupts to the CPU interfaces. In addition:

- When forwarding of pending interrupts is disabled for Group 0 or Group 1 interrupts, it is IMPLEMENTATION DEFINED whether an edge-triggered interrupt signal sets an edge-triggered interrupt in a disabled group to the pending state.
- In GICv2, software can manage SGI pending state using the Interrupt Set-Pending Register, [GICD\\_ISPENDRn](#) and Interrupt Clear-Pending Register, [GICD\\_ICPENDRn](#). However, in GICv1, the GIC clears the pending state of an SGI only when the SGI becomes active, and therefore software cannot clear the pending state of an SGI.
- In GICv2, software can manage the active state using the Interrupt Set-Active Registers, [GICD\\_ISACTIVERn](#) and the Interrupt Clear-Active Registers, [GICD\\_ICACTIVERn](#).

If the forwarding of only one group of interrupts is disabled, and the highest priority pending interrupt is in the disabled group:

- In GICv1, it is IMPLEMENTATION DEFINED whether the Distributor forwards any pending interrupts of [Sufficient priority](#) from the other group, to the CPU interfaces.
- In GICv2, the Distributor does not forward any interrupts, from either group, to the CPU interfaces.

When the GICD\_CTLR.{EnableGrp1, EnableGrp0} settings mean the Distributor does not forward any pending interrupts to the CPU interfaces, a read of a [GICC\\_IAR](#) or [GICC\\_AIAR](#) register returns a spurious interrupt ID.

———— **Note** —————

Interrupts are, by definition, asynchronous events and register values take a small but finite time to update. Software must consider this state change associated with the reporting of pending or spurious interrupts on a CPU interface during this transition.

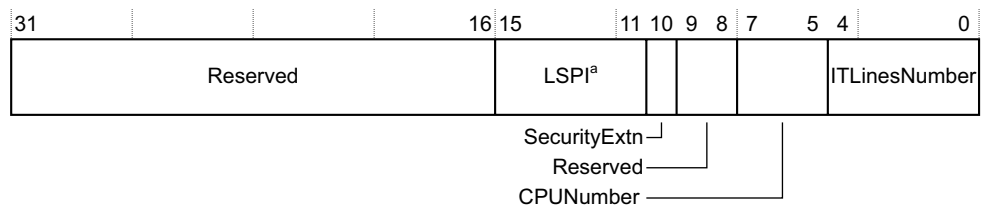
---

### 4.3.2 Interrupt Controller Type Register, GICD\_TYPER

The GICD\_TYPER characteristics are:

- Purpose** Provides information about the configuration of the GIC. It indicates:
  - whether the GIC implements the Security Extensions
  - the maximum number of interrupt IDs that the GIC supports
  - the number of CPU interfaces implemented
  - if the GIC implements the Security Extensions, the maximum number of implemented *Lockable Shared Peripheral Interrupts (LSPIs)*.
- Usage constraints** No usage constraints.
- Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.
- Attributes** See the register summary in [Table 4-1 on page 4-75](#).

[Figure 4-3](#) shows the GICD\_TYPER bit assignments.



a Implemented only if the GIC implements the Security Extensions, Reserved otherwise

**Figure 4-3 GICD\_TYPER bit assignments**

[Table 4-6](#) shows the GICD\_TYPER bit assignments.

**Table 4-6 GICD\_TYPER bit assignments**

Bits	Name	Function
[31:16]	-	Reserved.
[15:11]	LSPI	If the GIC implements the Security Extensions, the value of this field is the maximum number of implemented lockable SPIs, from 0 (0b00000) to 31 (0b11111), see <a href="#">Configuration lockdown on page 4-82</a> . If this field is 0b00000 then the GIC does not implement configuration lockdown. If the GIC does not implement the Security Extensions, this field is reserved.
[10]	SecurityExtn	Indicates whether the GIC implements the Security Extensions. <b>0</b> Security Extensions not implemented. <b>1</b> Security Extensions implemented.



**Table 4-6 GIC\_TYPER bit assignments (continued)**

Bits	Name	Function
[9:8]	-	Reserved.
[7:5]	CPUNumber	Indicates the number of implemented CPU interfaces. The number of implemented CPU interfaces is one more than the value of this field, for example if this field is 0b011, there are four CPU interfaces. If the GIC implements the Virtualization Extensions, this is also the number of virtual CPU interfaces.
[4:0]	ITLinesNumber	Indicates the maximum number of interrupts that the GIC supports. If ITLinesNumber=N, the maximum number of interrupts is 32(N+1). The interrupt ID range is from 0 to (number of IDs – 1). For example: 0b0011      Up to 128 interrupt lines, interrupt IDs 0-127. The maximum number of interrupts is 1020 (0b11111). See the text in this section for more information. Regardless of the range of interrupt IDs defined by this field, interrupt IDs 1020-1023 are reserved for special purposes, see <a href="#">Special interrupt numbers on page 3-43</a> and <a href="#">Interrupt IDs on page 2-24</a> .

The ITLinesNumber field only indicates the maximum number of SPIs that the GIC might support. This value determines the number of implemented interrupt registers, that is, the number of instances of the following registers:

- [GICD\\_IGROUPR<sub>n</sub>](#)
- [GICD\\_ISENABLER<sub>n</sub>](#)
- [GICD\\_ICENABLER<sub>n</sub>](#)
- [GICD\\_ISPENDR<sub>n</sub>](#)
- [GICD\\_ICPENDR<sub>n</sub>](#)
- [GICD\\_ISACTIVER<sub>n</sub>](#)
- [GICD\\_IPRIORITYR<sub>n</sub>](#)
- [GICD\\_ITARGETSR<sub>n</sub>](#)
- [GICD\\_ICFGR<sub>n</sub>](#).

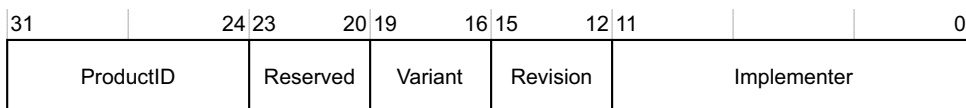
The GIC architecture does not require a GIC to support a continuous range of SPI interrupt IDs, and the supported SPI interrupt ID range is likely to be non-continuous. Software must check which SPI interrupt IDs are supported, up to the maximum value indicated by the ITLinesNumber field, see [Identifying the supported interrupts on page 3-35](#).

### 4.3.3 Distributor Implementer Identification Register, GICD\_IIDR

The GICD\_IIDR characteristics are:

- Purpose** Provides information about the implementer and revision of the Distributor.
- Usage constraints** No usage constraints.
- Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.
- Attributes** See the register summary in [Table 4-1 on page 4-75](#).

[Figure 4-4](#) shows the GICD\_IIDR bit assignments.



**Figure 4-4 GICD\_IIDR bit assignments**

[Table 4-7](#) shows the GICD\_IIDR bit assignments.

**Table 4-7 GICD\_IIDR bit assignments**

Bits	Name	Function
[31:24]	ProductID	An IMPLEMENTATION DEFINED product identifier.
[23:20]	-	Reserved.
[19:16]	Variant	An IMPLEMENTATION DEFINED variant number. Typically, this field is used to distinguish product variants, or major revisions of a product. <sup>a</sup>
[15:12]	Revision	An IMPLEMENTATION DEFINED revision number. Typically, this field is used to distinguish minor revisions of a product. <sup>a</sup>
[11:0]	Implementer	Contains the JEP106 code of the company that implemented the GIC Distributor: <b>Bits [11:8]</b> The JEP106 continuation code of the implementer. For an ARM implementation, this field is 0x4. <b>Bits [7]</b> Always 0. <b>Bits [6:0]</b> The JEP106 identity code of the implementer. For an ARM implementation, bits[7:0] are 0x3B.

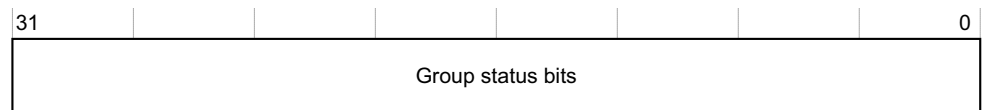
a. This field is not used to distinguish between GICv1 and GICv2 implementations.

### 4.3.4 Interrupt Group Registers, GICD\_IGROUPRn

The GICD\_IGROUPR characteristics are:

<b>Purpose</b>	The GICD_IGROUPR registers provide a status bit for each interrupt supported by the GIC. Each bit controls whether the corresponding interrupt is in Group 0 or Group 1.
<b>Usage constraints</b>	In implementations that include the GIC Security Extensions, accessible by Secure accesses only. The register addresses are RAZ/WI to Non-secure accesses.  A register bit corresponding to an unimplemented interrupt is RAZ/WI.  If the GIC implements configuration lockdown, the system can lockdown the group status bits for lockable SPIs that are configured as Group 0, see <a href="#">Configuration lockdown on page 4-82</a> .
<b>Configurations</b>	In GICv1, only implemented if the GIC implements the Security Extensions. If a GICv1 implementation does not include the Security Extensions the GICD_IGROUPR addresses are RAZ/WI.  ———— <b>Note</b> —————  Typically, when used with a processor that implements the ARM Security Extensions, Group 0 interrupts are Secure interrupts, and Group 1 interrupts are Non-secure interrupts, see <a href="#">Security Extensions support on page 1-16</a> for more information.  —————  In GICv2, these registers are always implemented.  The number of implemented GICD_IGROUPR registers is (GICD_TYPER.ITLinesNumber + 1). The implemented GICD_IGROUPR registers number upwards from GICD_IGROUPR0. If the GIC implements the Security Extensions, these are Secure registers.  In a multiprocessor implementation, GICD_IGROUPR0 is banked for each connected processor. This register holds the group status bits for interrupts 0-31.
<b>Attributes</b>	See the register summary in <a href="#">Table 4-1 on page 4-75</a> , and <a href="#">GICD_IGROUPR0 reset value on page 4-92</a> .

[Figure 4-5](#) shows the GICD\_IGROUPR bit assignments.



**Figure 4-5 GICD\_IGROUPR bit assignments**

[Table 4-8](#) shows the GICD\_IGROUPR bit assignments.

**Table 4-8 GICD\_IGROUPR bit assignments**

Bits	Name	Function
[31:0]	Group status bits	For each bit: <b>0</b> The corresponding interrupt is Group 0. <b>1</b> The corresponding interrupt is Group 1.

———— **Note** —————

On start-up or reset, each interrupt with ID32 or higher resets as Group 0 and therefore all SPIs are Group 0 unless the system reprograms the appropriate GICD\_IGROUPR bit. See *GICD\_IGROUPR0 reset value* for information about the reset configuration of interrupts with IDs 0-31.

For interrupt ID  $m$ , when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD\_IGROUPR $n$  number,  $n$ , is given by  $n = m \text{ DIV } 32$
- the offset of the required GICD\_IGROUPR is  $(0x080 + (4*n))$
- the bit number of the required group status bit in this register is  $m \text{ MOD } 32$ .

### **GICD\_IGROUPR0 reset value**

Typically, the reset value of all GICD\_IGROUPR registers is zero, so that all interrupts are Group 0 unless reprogrammed as Group 1 by Secure accesses to the appropriate GICD\_IGROUPR registers.

For GICv2 implementations that do not include the Security Extensions, the GICD\_IGROUPR $n$  reset values are IMPLEMENTATION DEFINED.

A multiprocessor implementation that supports the Security Extensions might include one or more *Non-secure processors*, meaning processors that cannot make Secure accesses to the GIC. In this situation only, a GIC can implement a Secure IMPLEMENTATION DEFINED mechanism that resets to 1 the GICD\_IGROUPR0 bits for the peripheral interrupts and SGIs of any Non-secure processor. This mechanism must apply only to:

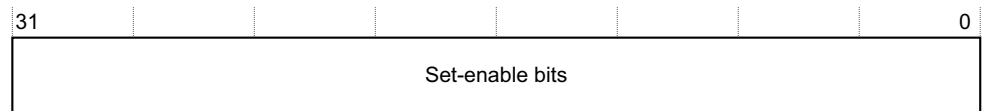
- a banked GICD\_IGROUPR0 that corresponds to a Non-secure processor
- bits in that banked GICD\_IGROUPR0 that correspond to implemented interrupts.

### 4.3.5 Interrupt Set-Enable Registers, GICD\_ISENABLERn

The GICD\_ISENABLER characteristics are:

- Purpose** The GICD\_ISENABLERs provide a Set-enable bit for each interrupt supported by the GIC. Writing 1 to a Set-enable bit enables forwarding of the corresponding interrupt from the Distributor to the CPU interfaces. Reading a bit identifies whether the interrupt is enabled.
- Usage constraints** A register bit corresponding to an unimplemented interrupt is RAZ/WI. If the GIC implements the Security Extensions:
- a register bit that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses
  - if the GIC implements configuration lockdown, the system can lock down the Set-enable bits for the lockable SPIs that are configured as Group 0, see [Configuration lockdown on page 4-82](#).
- Whether implemented SGIs are permanently enabled, or can be enabled and disabled by writes to GICD\_ISENABLER0 and GICD\_ICENABLER0, is IMPLEMENTATION DEFINED.
- Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common. The number of implemented GICD\_ISENABLERs is (GICD\_TYPER.ITLinesNumber+1). The implemented GICD\_ISENABLERs number upwards from GICD\_ISENABLER0. In a multiprocessor implementation, GICD\_ISENABLER0 is banked for each connected processor. This register holds the Set-enable bits for interrupts 0-31.
- Attributes** See the register summary in [Table 4-1 on page 4-75](#).

[Figure 4-6](#) shows the GICD\_ISENABLER bit assignments.



**Figure 4-6 GICD\_ISENABLER bit assignments**

[Table 4-9](#) shows the GICD\_ISENABLER bit assignments.

**Table 4-9 GICD\_ISENABLER bit assignments**

Bits	Name	Function
[31:0]	Set-enable bits	<p>For SPIs and PPIs, each bit controls the forwarding of the corresponding interrupt from the Distributor to the CPU interfaces:</p> <p><b>Reads</b></p> <p>    <b>0</b> Forwarding of the corresponding interrupt is disabled.</p> <p>    <b>1</b> Forwarding of the corresponding interrupt is enabled.</p> <p><b>Writes</b></p> <p>    <b>0</b> Has no effect.</p> <p>    <b>1</b> Enables the forwarding of the corresponding interrupt.</p> <p>After a write of 1 to a bit, a subsequent read of the bit returns the value 1.</p> <p>For SGIs the behavior of the bit on reads and writes is IMPLEMENTATION DEFINED.</p>

For interrupt ID  $m$ , when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD\_ISENABLER number,  $n$ , is given by  $n = m \text{ DIV } 32$
- the offset of the required GICD\_ISENABLER is  $(0x100 + (4*n))$
- the bit number of the required Set-enable bit in this register is  $m \text{ MOD } 32$ .

At start-up, and after a reset, a processor can use this register to discover which peripheral interrupt IDs the GIC supports. If the processor and the GIC both implement the Security Extensions it must do this for the Secure view of the available interrupts, and Non-secure software running on the processor must do this discovery after the Secure software has configured interrupts as Group 0 (Secure) and Group 1 (Non-secure). For more information see [Identifying the supported interrupts on page 3-35](#).

---

**Note**

Disabling an interrupt only disables the forwarding of the interrupt from the Distributor to any CPU interface. It does not prevent the interrupt from changing state, for example becoming pending, or active and pending if it is already active.

---

### 4.3.6 Interrupt Clear-Enable Registers, GICD\_ICENABLERn

The GICD\_ICENABLER characteristics are:

**Purpose** The GICD\_ICENABLERs provide a Clear-enable bit for each interrupt supported by the GIC. Writing 1 to a Clear-enable bit disables forwarding of the corresponding interrupt from the Distributor to the CPU interfaces. Reading a bit identifies whether the interrupt is enabled.

**Usage constraints** A register bit corresponding to an unimplemented interrupt is RAZ/WI. If the GIC implements the Security Extensions:

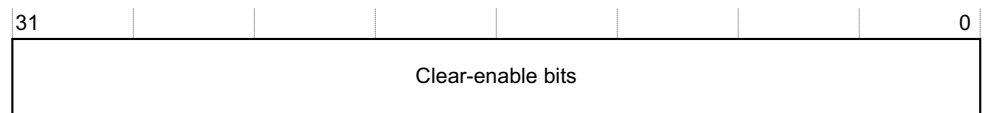
- a register bit that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses
- if the GIC implements configuration lockdown, the system can lock down the Clear-enable bits for the lockable SPIs that are configured as Group 0, see [Configuration lockdown on page 4-82](#).

Whether implemented SGIs are permanently enabled, or can be enabled and disabled by writes to GICD\_ISENABLER0 and GICD\_ICENABLER0, is IMPLEMENTATION DEFINED.

**Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common. The number of implemented GICD\_ICENABLERs is (GICD\_TYPER.ITLinesNumber+1). The implemented GICD\_ICENABLERs number upwards from GICD\_ICENABLER0. In a multiprocessor implementation, GICD\_ICENABLER0 is banked for each connected processor. This register holds the Clear-enable bits for interrupts 0-31.

**Attributes** See the register summary in [Table 4-1 on page 4-75](#).

[Figure 4-7](#) shows the GICD\_ICENABLER bit assignments.



**Figure 4-7 GICD\_ICENABLER bit assignments**

[Table 4-10](#) shows the GICD\_ICENABLER bit assignments.

**Table 4-10 GICD\_ICENABLER bit assignments**

Bits	Name	Function
[31:0]	Clear-enable bits	For SPIs and PPIs, each bit controls the forwarding of the corresponding interrupt from the Distributor to the CPU interfaces: <b>Reads</b> <b>0</b> Forwarding of the corresponding interrupt is disabled. <b>1</b> Forwarding of the corresponding interrupt is enabled. <b>Writes</b> <b>0</b> Has no effect. <b>1</b> Disables the forwarding of the corresponding interrupt. After a write of 1 to a bit, a subsequent read of the bit returns the value 0. For SGIs the behavior of the bit on reads and writes is IMPLEMENTATION DEFINED.

For interrupt ID  $m$ , when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD\_ICENABLERn number,  $n$ , is given by  $m = n \text{ DIV } 32$
- the offset of the required GICD\_ICENABLERn is  $(0 \times 180 + (4 * n))$
- the bit number of the required Clear-enable bit in this register is  $m \text{ MOD } 32$ .

———— **Note** —————

Writing a 1 to an GICD\_ICENABLERn bit only disables the forwarding of the corresponding interrupt from the Distributor to any CPU interface. It does not prevent the interrupt from changing state, for example becoming pending, or active and pending if it is already active.

---

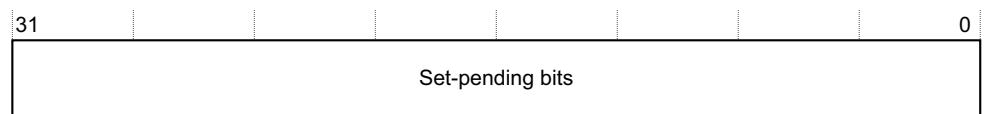


### 4.3.7 Interrupt Set-Pending Registers, GICD\_ISPENDRn

The GICD\_ISPENDR characteristics are:

<b>Purpose</b>	The GICD_ISPENDRs provide a Set-pending bit for each interrupt supported by the GIC. Writing 1 to a Set-pending bit sets the status of the corresponding peripheral interrupt to pending. Reading a bit identifies whether the interrupt is pending.
<b>Usage constraints</b>	<p>A register bit corresponding to an unimplemented interrupt is RAZ/WI.</p> <p>If the GIC implements the Security Extensions:</p> <ul style="list-style-type: none"> <li>• a register bit that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses</li> <li>• if the GIC implements configuration lockdown, the system can lock down the Set-pending bits for the lockable SPIs that are configured as Group 0, see <a href="#">Configuration lockdown on page 4-82</a>.</li> </ul> <p>Set-pending bits for SGIs are read-only and ignore writes.</p>
<b>Configurations</b>	<p>These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.</p> <p>The number of implemented GICD_ISPENDRs is (GICD_TYPER.ITLinesNumber+1). The implemented GICD_ISPENDRs number upwards from GICD_ISPENDR0.</p> <p>In a multiprocessor implementation, GICD_ISPENDR0 is banked for each connected processor. This register holds the Set-pending bits for interrupts 0-31.</p>
<b>Attributes</b>	See the register summary in <a href="#">Table 4-1 on page 4-75</a> .

[Figure 4-8](#) shows the GICD\_ISPENDR bit assignments.



**Figure 4-8 GICD\_ISPENDR bit assignments**

[Table 4-11 on page 4-98](#) shows the GICD\_ISPENDR bit assignments.

**Table 4-11 GICD\_ISPENDR bit assignments**

Bits	Name	Function
[31:0]	Set-pending bits	<p>For each bit:</p> <p><b>Reads</b></p> <p><b>0</b> The corresponding interrupt is not pending on any processor.</p> <p><b>1</b></p> <ul style="list-style-type: none"> <li>For PPIs and SGIs, the corresponding interrupt is pending<sup>a</sup> on this processor.</li> <li>For SPIs, the corresponding interrupt is pending<sup>a</sup> on at least one processor.</li> </ul> <p><b>Writes</b></p> <p>For SPIs and PPIs:</p> <p><b>0</b> Has no effect.</p> <p><b>1</b> The effect depends on whether the interrupt is edge-triggered or level-sensitive:</p> <p><b>Edge-triggered</b></p> <p>Changes the status of the corresponding interrupt to:</p> <ul style="list-style-type: none"> <li>pending if it was previously inactive</li> <li>active and pending if it was previously active.</li> </ul> <p>Has no effect if the interrupt is already pending<sup>a</sup>.</p> <p><b>Level sensitive</b></p> <p>If the corresponding interrupt is not pending<sup>a</sup>, changes the status of the corresponding interrupt to:</p> <ul style="list-style-type: none"> <li>pending if it was previously inactive</li> <li>active and pending if it was previously active.</li> </ul> <p>If the interrupt is already pending<sup>a</sup>:</p> <ul style="list-style-type: none"> <li>because of a write to the GICD_ISPENDR, the write has no effect</li> <li>because the corresponding interrupt signal is asserted, the write has no effect on the status of the interrupt, but the interrupt remains pending<sup>a</sup> if the interrupt signal is deasserted.</li> </ul> <p>For more information see <i>Control of the pending status of level-sensitive interrupts</i> on page 4-100.</p> <p>For SGIs, the write is ignored. SGIs have their own Set-Pending registers, see <i>SGI Set-Pending Registers, GICD_SPENDSGIRn</i> on page 4-117.</p>

a. Pending interrupts include interrupts that are active and pending.

For interrupt ID  $m$ , when DIV and MOD are the integer division and modulo operations:

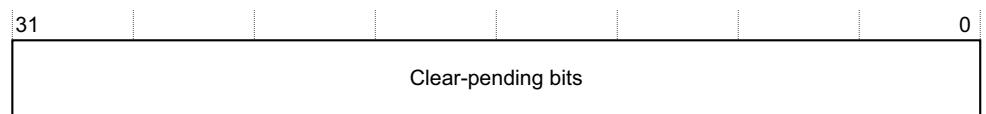
- the corresponding GICD\_ISPENDR number,  $n$ , is given by  $n = m \text{ DIV } 32$
- the offset of the required GICD\_ISPENDR is  $(0 \times 200 + (4 * n))$
- the bit number of the required Set-pending bit in this register is  $m \text{ MOD } 32$ .

### 4.3.8 Interrupt Clear-Pending Registers, GICD\_ICPENDRn

The GICD\_ICPENDR characteristics are:

<b>Purpose</b>	The GICD_ICPENDRs provide a Clear-pending bit for each interrupt supported by the GIC. Writing 1 to a Clear-pending bit clears the pending state of the corresponding peripheral interrupt. Reading a bit identifies whether the interrupt is pending.
<b>Usage constraints</b>	<p>A register bit corresponding to an unimplemented interrupt is RAZ/WI.</p> <p>If the GIC implements the Security Extensions:</p> <ul style="list-style-type: none"> <li>• a register bit that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses</li> <li>• if the GIC implements configuration lockdown, the system can lock down the Clear-pending bits for the lockable SPIs that are configured as Group 0, see <a href="#">Configuration lockdown on page 4-82</a>.</li> </ul> <p>Clear-pending bits for SGIs are read-only and ignore writes.</p>
<b>Configurations</b>	<p>These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.</p> <p>The number of implemented GICD_ICPENDRs is (GICD_TYPER.ITLinesNumber+1). The implemented GICD_ICPENDRs number upwards from GICD_ICPENDR0.</p> <p>In a multiprocessor implementation, GICD_ICPENDR0 is banked for each connected processor. This register holds the Clear-pending bits for interrupts 0-31.</p>
<b>Attributes</b>	See the register summary in <a href="#">Table 4-1 on page 4-75</a> .

[Figure 4-9](#) shows the GICD\_ICPENDR bit assignments.



**Figure 4-9 GICD\_ICPENDR bit assignments**

[Table 4-12 on page 4-100](#) shows the GICD\_ICPENDR bit assignments.

**Table 4-12 GICD\_ICPENDR bit assignments**

Bits	Name	Function
[31:0]	Clear-pending bits	<p>For each bit:</p> <p><b>Reads</b></p> <p><b>0</b> The corresponding interrupt is not pending on any processor.</p> <p><b>1</b></p> <ul style="list-style-type: none"> <li>For SGIs and PPIs, the corresponding interrupt is pending<sup>a</sup> on this processor.</li> <li>For SPIs, the corresponding interrupt is pending<sup>a</sup> on at least one processor.</li> </ul> <p><b>Writes</b></p> <p>For SPIs and PPIs:</p> <p><b>0</b> Has no effect.</p> <p><b>1</b> The effect depends on whether the interrupt is edge-triggered or level-sensitive:</p> <p><b>Edge-triggered</b></p> <p>Changes the status of the corresponding interrupt to:</p> <ul style="list-style-type: none"> <li>inactive if it was previously pending</li> <li>active if it was previously active and pending.</li> </ul> <p>Has no effect if the interrupt is not pending.</p> <p><b>Level-sensitive</b></p> <p>If the corresponding interrupt is pending<sup>a</sup> only because of a write to <a href="#">GICD_ISPENDRn</a>, the write changes the status of the interrupt to:</p> <ul style="list-style-type: none"> <li>inactive if it was previously pending</li> <li>active if it was previously active and pending.</li> </ul> <p>Otherwise the interrupt remains pending if the interrupt signal remains asserted, see <a href="#">Control of the pending status of level-sensitive interrupts</a></p> <p>For SGIs, the write is ignored. SGIs have their own Clear-Pending registers, see <a href="#">SGI Clear-Pending Registers, GICD_CPENDSGIRn</a> on page 4-115.</p>

a. Pending interrupts include interrupts that are active and pending.

For interrupt ID  $m$ , when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD\_ICPENDR number,  $n$ , is given by  $n = m \text{ DIV } 32$
- the offset of the required GICD\_ICPENDR is  $(0 \times 280 + (4 * n))$
- the bit number of the required Set-pending bit in this register is  $m \text{ MOD } 32$ .

### Control of the pending status of level-sensitive interrupts

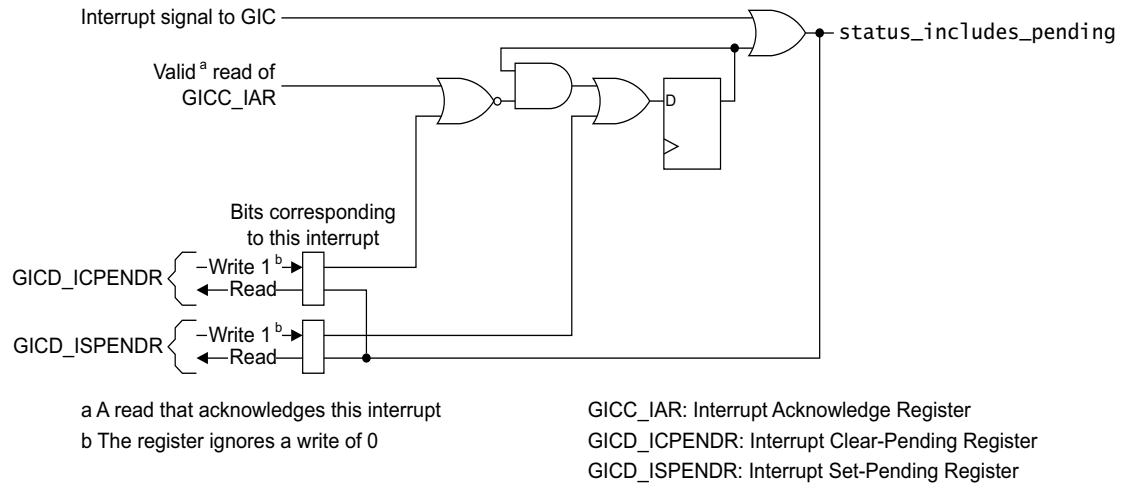
This subsection describes the status of an interrupt as *includes pending* if the interrupt status is one of:

- pending
- active and pending.

For an edge-triggered interrupt, the includes pending status is latched on either a write to the [GICD\\_ISPENDRn](#) or the assertion of the interrupt signal to the GIC. However, for a level-sensitive interrupt, the includes pending status either:

- is latched on a write to the [GICD\\_ISPENDRn](#)
- follows the state of the interrupt signal to the GIC, without any latching.

This means that the operation of the Set-pending and Clear-pending registers is more complicated for level-sensitive interrupts. [Figure 4-10 on page 4-101](#) shows the logic of the pending status of a level-sensitive interrupt. The logical output `status_includes_pending` is TRUE when the interrupt status includes pending, and FALSE otherwise.



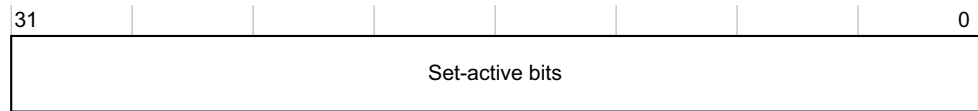
**Figure 4-10 Logic of the pending status of a level-sensitive interrupt**

### 4.3.9 Interrupt Set-Active Registers, GICD\_ISACTIVERn

The GICD\_ISACTIVER characteristics are:

- Purpose** The GICD\_ISACTIVERs provide a Set-active bit for each interrupt that the GIC supports. Writing to a Set-active bit **Activates** the corresponding interrupt. These registers are used when preserving and restoring GIC state.  
In GICv1, the GICD\_ISACTIVERn registers are the RO Active Bit Registers, ICDABRn.
- Usage constraints** A register bit corresponding to an unimplemented interrupt is RAZ/WI.  
If the GIC implements the Security Extensions a register bit that corresponds to a Group 0 interrupt is RAZ to Non-secure accesses.  
The bit reads as one if the status of the interrupt is active or active and pending. Read the **GICD\_ISPENDRn** or **GICD\_ICPENDRn** to find the pending status of the interrupt.
- Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.  
The number of implemented GICD\_ISACTIVERs is (**GICD\_TYPER.ITLinesNumber**+1). The implemented GICD\_ISACTIVERs number upwards from GICD\_ISACTIVER0.  
In a multiprocessor implementation, GICD\_ISACTIVER0 is banked for each connected processor. This register holds the Set-active bits for interrupts 0-31.  
These registers are RO in GICv1 and RW in GICv2.
- Attributes** See the register summary in [Table 4-1 on page 4-75](#).

[Figure 4-11](#) shows the GICD\_ISACTIVER bit assignments.



**Figure 4-11 GICD\_ISACTIVER bit assignments**

[Table 4-13](#) shows the GICD\_ISACTIVER bit assignments.

**Table 4-13 GICD\_ISACTIVER bit assignments**

Bits	Name	Function												
[31:0]	Set-active bits	For each bit: <table border="0" style="margin-left: 20px;"> <tr> <td><b>Reads</b></td> <td><b>0</b></td> <td>The corresponding interrupt is not active<sup>a</sup>.</td> </tr> <tr> <td></td> <td><b>1</b></td> <td>The corresponding interrupt is active<sup>a</sup>.</td> </tr> <tr> <td><b>Writes</b></td> <td><b>0</b></td> <td>Has no effect.</td> </tr> <tr> <td></td> <td><b>1</b></td> <td><b>Activates</b> the corresponding interrupt, if it is not already active. If the interrupt is already active, the write has no effect.</td> </tr> </table> <p style="margin-left: 20px;">After a write of 1 to this bit, a subsequent read of the bit returns the value 1.</p>	<b>Reads</b>	<b>0</b>	The corresponding interrupt is not active <sup>a</sup> .		<b>1</b>	The corresponding interrupt is active <sup>a</sup> .	<b>Writes</b>	<b>0</b>	Has no effect.		<b>1</b>	<b>Activates</b> the corresponding interrupt, if it is not already active. If the interrupt is already active, the write has no effect.
<b>Reads</b>	<b>0</b>	The corresponding interrupt is not active <sup>a</sup> .												
	<b>1</b>	The corresponding interrupt is active <sup>a</sup> .												
<b>Writes</b>	<b>0</b>	Has no effect.												
	<b>1</b>	<b>Activates</b> the corresponding interrupt, if it is not already active. If the interrupt is already active, the write has no effect.												

a. Active interrupts include interrupts that are active and pending.

For interrupt ID  $m$ , when DIV and MOD are the integer division and modulo operations:

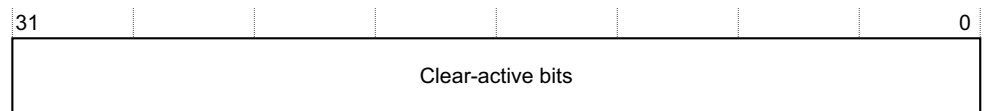
- the corresponding GICD\_ISACTIVERn number,  $n$ , is given by  $n = m \text{ DIV } 32$
- the offset of the required GICD\_ISACTIVERn is  $(0x300 + (4*n))$
- the bit number of the required Set-active bit in this register is  $m \text{ MOD } 32$ .

### 4.3.10 Interrupt Clear-Active Registers, GICD\_ICACTIVERn

The GICD\_ICACTIVER characteristics are:

<b>Purpose</b>	The GICD_ICACTIVERs provide a Clear-active bit for each interrupt that the GIC supports. Writing to a Clear-active bit <b>Deactivates</b> the corresponding interrupt. These registers are used when preserving and restoring GIC state.
<b>Usage constraints</b>	A register bit corresponding to an unimplemented interrupt is RAZ/WI. If the GIC implements the Security Extensions, a register bit that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses.
<b>Configurations</b>	These registers are present only in GICv2. The register locations are reserved in GICv1. The number of implemented GICD_ICACTIVERs is (GICD_TYPER.ITLinesNumber+1). The implemented GICD_ICACTIVERs number upwards from GICD_ICACTIVER0. In a multiprocessor implementation, GICD_ICACTIVER0 is banked for each connected processor. This register holds the Clear-active bits for interrupts 0-31.
<b>Attributes</b>	See the register summary in <a href="#">Table 4-1 on page 4-75</a> .

[Figure 4-12](#) shows the GICD\_ICACTIVER bit assignments.



**Figure 4-12 GICD\_ICACTIVER bit assignments**

[Table 4-14](#) shows the GICD\_ICACTIVER bit assignments.

**Table 4-14 GICD\_ICACTIVER bit assignments**

Bits	Name	Function												
[31:0]	Clear-active bits	For each bit: <table border="0" style="margin-left: 20px;"> <tr> <td style="vertical-align: top;"><b>Reads</b></td> <td style="vertical-align: top;"><b>0</b></td> <td>The corresponding interrupt is not active<sup>a</sup>.</td> </tr> <tr> <td></td> <td style="vertical-align: top;"><b>1</b></td> <td>The corresponding interrupt is active<sup>a</sup>.</td> </tr> <tr> <td style="vertical-align: top;"><b>Writes</b></td> <td style="vertical-align: top;"><b>0</b></td> <td>Has no effect.</td> </tr> <tr> <td></td> <td style="vertical-align: top;"><b>1</b></td> <td><b>Deactivates</b> the corresponding interrupt, if the interrupt is active. If the interrupt is already deactivated, the write has no effect.</td> </tr> </table> <p style="margin-left: 20px;">After a write of 1 to this bit, a subsequent read of the bit returns the value 0.</p>	<b>Reads</b>	<b>0</b>	The corresponding interrupt is not active <sup>a</sup> .		<b>1</b>	The corresponding interrupt is active <sup>a</sup> .	<b>Writes</b>	<b>0</b>	Has no effect.		<b>1</b>	<b>Deactivates</b> the corresponding interrupt, if the interrupt is active. If the interrupt is already deactivated, the write has no effect.
<b>Reads</b>	<b>0</b>	The corresponding interrupt is not active <sup>a</sup> .												
	<b>1</b>	The corresponding interrupt is active <sup>a</sup> .												
<b>Writes</b>	<b>0</b>	Has no effect.												
	<b>1</b>	<b>Deactivates</b> the corresponding interrupt, if the interrupt is active. If the interrupt is already deactivated, the write has no effect.												

a. Active interrupts include interrupts that are active and pending.

For interrupt ID  $m$ , when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD\_ICACTIVERn number,  $n$ , is given by  $n = m \text{ DIV } 32$
- the offset of the required GICD\_ICACTIVERn is  $(0x380 + (4*n))$
- the bit number of the required Clear-active bit in this register is  $m \text{ MOD } 32$ .

### 4.3.11 Interrupt Priority Registers, GICD\_IPRIORITYRn

The GICD\_IPRIORITYR characteristics are:

**Purpose** The GICD\_IPRIORITYRs provide an 8-bit priority field for each interrupt supported by the GIC. This field stores the priority of the corresponding interrupt.

**Usage constraints** These registers are byte-accessible.  
A register field corresponding to an unimplemented interrupt is RAZ/WI.  
A GIC might implement fewer than eight priority bits, but must implement at least bits [7:4] of each field. In each field, unimplemented bits are RAZ/WI.

If the GIC implements the Security Extensions:

- a register field that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses
- a Non-secure access to a field that corresponds to a Group 1 interrupt behaves as described in *Software views of interrupt priority in a GIC that includes the Security Extensions* on page 3-53
- if the GIC implements configuration lockdown, the system can lock down the Priority fields for the lockable SPIs that are configured as Group 0, see *Configuration lockdown* on page 4-82

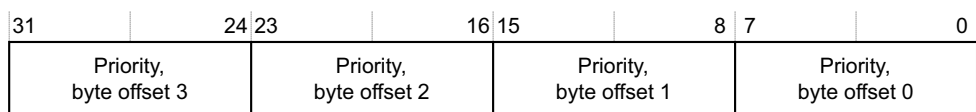
It is IMPLEMENTATION DEFINED whether changing the value of a priority field changes the priority of an active interrupt.

**Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.  
The number of implemented GICD\_IPRIORITYRs is  $(8 * (\text{GICD\_TYPER.ITLinesNumber} + 1))$ . The implemented GICD\_IPRIORITYRs number upwards from GICD\_IPRIORITYR0.

In a multiprocessor implementation, GICD\_IPRIORITYR0 to GICD\_IPRIORITYR7 are banked for each connected processor. These registers hold the Priority fields for interrupts 0-31.

**Attributes** See the register summary in [Table 4-1 on page 4-75](#).

[Figure 4-13](#) shows the GICD\_IPRIORITYR bit assignments.



**Figure 4-13 GICD\_IPRIORITYR bit assignments**

[Table 4-15](#) shows the GICD\_IPRIORITYR bit assignments.

**Table 4-15 GICD\_IPRIORITYR bit assignments**

Bits	Name <sup>a</sup>	Function
[31:24]	Priority, byte offset 3	Each priority field holds a priority value, from an IMPLEMENTATION DEFINED range. The lower the value, the greater the priority of the corresponding interrupt. For more information see <i>Interrupt prioritization</i> on page 3-44 and, if appropriate, <i>Interrupt grouping and interrupt prioritization</i> on page 3-53.
[23:16]	Priority, byte offset 2	
[15:8]	Priority, byte offset 1	
[7:0]	Priority, byte offset 0	

a. Each field holds the priority value for a single interrupt. This section describes how the interrupt ID value determines the GICD\_IPRIORITYR register number and the byte offset of the priority field in that register.



For interrupt ID  $m$ , when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD\_IPRIORITYR $n$  number,  $n$ , is given by  $n = m \text{ DIV } 4$
- the offset of the required GICD\_IPRIORITYR $n$  is  $(0 \times 400 + (4 * n))$
- the byte offset of the required Priority field in this register is  $m \text{ MOD } 4$ , where:
  - byte offset 0 refers to register bits [7:0]
  - byte offset 1 refers to register bits [15:8]
  - byte offset 2 refers to register bits [23:16]
  - byte offset 3 refers to register bits [31:24].

The following pseudocode shows the effects of the GIC Security Extensions on accesses to this register.

```
// PriorityRegRead()
// =====
//

// P_MASK used here to emphasize that the number of valid bits is IMPLEMENTATION DEFINED

bits(8) PriorityRegRead(integer InterruptID)

    read_value = ReadGICD_IPRIORITYR(InterruptID);
    if NS_access then // A non-secure GIC access.
        read_value<7:0> = LSL((read_value AND P_MASK), 1);
        if IsGrp0Int(InterruptID) then
            read_value = '00000000'; // Can't read a Group 0 priority value
        return(read_value);

// PriorityRegWrite()
// =====
//

PriorityRegWrite(integer InterruptID, bits(8) value)

    if NS_access then // A non-secure GIC access.
        if !IsGrp0Int(InterruptID) then
            mod_write_val = ('10000000' OR LSR(value,1)) AND P_MASK;
            WriteGICD_IPRIORITYR(InterruptID, mod_write_val);
        else
            IgnoreWriteRequest();
    else // A secure GIC access.
        mod_write_val = value AND P_MASK;
        WriteGICD_IPRIORITYR(InterruptID, mod_write_val);
```

### 4.3.12 Interrupt Processor Targets Registers, GICD\_ITARGETSRn

The GICD\_ITARGETSR characteristics are:

**Purpose** The GICD\_ITARGETSRs provide an 8-bit CPU targets field for each interrupt supported by the GIC. This field stores the list of target processors for the interrupt. That is, it holds the list of CPU interfaces to which the Distributor forwards the interrupt if it is asserted and has sufficient priority.

**Usage constraints** For a multiprocessor implementation:

- These registers are byte-accessible.
- A register field corresponding to an unimplemented interrupt is RAZ/WI.
- GICD\_ITARGETSR0 to GICD\_ITARGETSR7 are read-only, and each field returns a value that corresponds only to the processor reading the register.
- It is IMPLEMENTATION DEFINED which, if any, SPIs are statically configured in hardware. The CPU targets field for such an SPI is read-only, and returns a value that indicates the CPU targets for the interrupt.
- if the GIC implements the Security Extensions:
  - a register field that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses
  - if the GIC implements configuration lockdown, the system can lock down the CPU targets fields for the lockable SPIs that are configured as Group 0, see [Configuration lockdown on page 4-82](#).

See also [The effect of changes to an GICD\\_ITARGETSR on page 4-108](#).

———— **Note** —————

In a uniprocessor implementation, all interrupts target the one processor, and the GICD\_ITARGETSRs are RAZ/WI.

**Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.

The number of implemented GICD\_ITARGETSRs is  $(8 * (\text{GICD\_TYPER.ITLinesNumber} + 1))$ . The implemented GICD\_ITARGETSRs number upwards from GICD\_ITARGETSR0.

In a multiprocessor implementation, GICD\_ITARGETSR0 to GICD\_ITARGETSR7 are banked for each connected processor. These registers hold the CPU targets fields for interrupts 0-31.

**Attributes** See the register summary in [Table 4-1 on page 4-75](#).

[Figure 4-14](#) shows the GICD\_ITARGETSR bit assignments, for a multiprocessor implementation.

31	24 23	16 15	8 7	0
CPU targets, byte offset 3	CPU targets, byte offset 2	CPU targets, byte offset 1	CPU targets, byte offset 0	

**Figure 4-14 GICD\_ITARGETSR bit assignments**

[Table 4-16 on page 4-107](#) shows the GICD\_ITARGETSR bit assignments, for a multiprocessor implementation.

**Table 4-16 GICD\_ITARGETSR bit assignments**

Bits	Name <sup>a</sup>	Function
[31:24]	CPU targets, byte offset 3	Processors in the system number from 0, and each bit in a CPU targets field refers to the corresponding processor, see <a href="#">Table 4-17</a> . For example, a value of 0x3 means that the Pending interrupt is sent to processors 0 and 1.
[23:16]	CPU targets, byte offset 2	
[15:8]	CPU targets, byte offset 1	For GICD_ITARGETSR0 to GICD_ITARGETSR7, a read of any CPU targets field returns the number of the processor performing the read.
[7:0]	CPU targets, byte offset 0	

- a. Each field holds the CPU targets list for a single interrupt. This section describes how the interrupt ID value determines the GICD\_ITARGETSR register number and the byte offset of the CPU targets field in that register.

[Table 4-17](#) shows how each bit of a CPU targets field targets the interrupt at one of the CPU interfaces.

**Table 4-17 Meaning of CPU targets field bit values**

CPU targets field value	Interrupt targets
0bxxxxxx1	CPU interface 0
0bxxxxxx1x	CPU interface 1
0bxxxxxx1xx	CPU interface 2
0bxxxxxx1xxx	CPU interface 3
0bxxx1xxxx	CPU interface 4
0bxx1xxxxx	CPU interface 5
0bx1xxxxxx	CPU interface 6
0b1xxxxxxx	CPU interface 7

A CPU targets field bit that corresponds to an unimplemented CPU interface is RAZ/WI.

For interrupt ID  $m$ , when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD\_ITARGETSR $n$  number,  $n$ , is given by  $n = m \text{ DIV } 4$
- the offset of the required GICD\_ITARGETSR is  $(0x800 + (4*n))$
- the byte offset of the required Priority field in this register is  $m \text{ MOD } 4$ , where:
  - byte offset 0 refers to register bits [7:0]
  - byte offset 1 refers to register bits [15:8]
  - byte offset 2 refers to register bits [23:16]
  - byte offset 3 refers to register bits [31:24].

### The effect of changes to an GICD\_ITARGETSR

Software can write to an GICD\_ITARGETSR at any time. Any change to a CPU targets field value:

- Has no effect on any active interrupt. This means that removing a CPU interface from a targets list does not cancel an active state for that interrupt on that CPU interface.
- Has an effect on any pending interrupts. This means:
  - adding a CPU interface to the target list of a pending interrupt makes that interrupt pending on that CPU interface
  - removing a CPU interface from the target list of a pending interrupt removes the pending state of that interrupt on that CPU interface.

———— **Note** —————

There is a small but finite time required for any change to take effect.

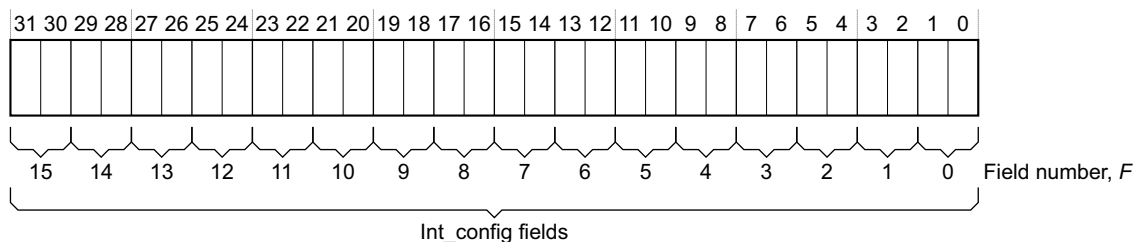
- If it applies to an interrupt that is active and pending, does not change the interrupt targets until the active status is cleared.

### 4.3.13 Interrupt Configuration Registers, GICD\_ICFGRn

The GICD\_ICFGR characteristics are:

<b>Purpose</b>	The GICD_ICFGRs provide a 2-bit Int_config field for each interrupt supported by the GIC. This field identifies whether the corresponding interrupt is edge-triggered or level-sensitive, see <a href="#">Interrupt types on page 1-18</a> .
<b>Usage constraints</b>	<p>For each supported PPI, it is IMPLEMENTATION DEFINED whether software can program the corresponding Int_config field.</p> <p>For SGIs, Int_config fields are read-only, meaning that GICD_ICFGR0 is read-only. For PPIs, it is IMPLEMENTATION DEFINED whether the most significant bit of the Int_config field is programmable. See <a href="#">Table 4-18 on page 4-110</a> for more information.</p> <p>A register field corresponding to an unimplemented interrupt is RAZ/WI.</p> <p>If the GIC implements the Security Extensions:</p> <ul style="list-style-type: none"> <li>• a register field that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses</li> <li>• if the GIC implements configuration lockdown, the system can lock down the Int_config fields for the lockable SPIs that are configured as Group 0, see <a href="#">Configuration lockdown on page 4-82</a>.</li> </ul> <p>Before changing the value of a programmable Int_config field, software must disable the corresponding interrupt, otherwise GIC behavior is UNPREDICTABLE.</p>
<b>Configurations</b>	<p>These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.</p> <p>In a multiprocessor implementation, if bit[1] of the Int_config field for any PPI is programmable then GICD_ICFGR1 is banked for each connected processor. This register holds the Int_config fields for the PPIs, interrupts 16-31.</p> <p>The number of implemented GICD_ICFGRs is <math>(2 * (\text{GICD\_TYPER.ITLinesNumber} + 1))</math>. The implemented GICD_ICFGRs number upwards from GICD_ICFGR0.</p>
<b>Attributes</b>	See the register summary in <a href="#">Table 4-1 on page 4-75</a> .

[Figure 4-15](#) shows the GICD\_ICFGR bit assignments.



See the bit assignment table for more information about the properties of each Int\_config[1:0] field.

**Figure 4-15 GICD\_ICFGR bit assignments**

[Table 4-18 on page 4-110](#) shows the GICD\_ICFGR bit assignments.

**Table 4-18 GICD\_ICFGR bit assignments**

Bits	Name	Function
[2F+1:2F]	Int_config, field F	<p>For Int_config[1], the most significant bit, bit [2F+1], the encoding is:</p> <p><b>0</b> Corresponding interrupt is level-sensitive.</p> <p><b>1</b> Corresponding interrupt is edge-triggered.</p> <p>Int_config[0], the least significant bit, bit [2F], is reserved, but see <a href="#">Table 4-19</a> for the encoding of this bit on some early implementations of this GIC architecture.</p> <p>For SGIs:</p> <p><b>Int_config[1]</b> Not programmable, RAO/WI.</p> <p>For PPIs and SPIs:</p> <p><b>Int_config[1]</b> For SPIs, this bit is programmable.<sup>a</sup> For PPIs it is IMPLEMENTATION DEFINED whether this bit is programmable. A read of this bit always returns the correct value to indicate whether the corresponding interrupt is level-sensitive or edge-triggered.</p>

- a. If the GIC implements the Security Extensions and the bit corresponds to a Group 0 interrupt, it is RAZ/WI to Non-secure accesses. This is the usual behavior of bits that correspond to Group 0 interrupts.

In some implementations of this GIC architecture before the publication of the GICv1 Architecture Specification, the model for handling each peripheral interrupt can be configured using bit [0] of the corresponding Int\_config field. [Table 4-19](#) shows the encoding of Int\_config[0] on these implementations.

**Table 4-19 GICD\_ICFGR Int\_config[0] encoding in some early GIC implementations**

Bits	Name	Function
[2F]	Int_config[0], field F	<p>On a GIC where the handling mode of peripheral interrupts is configurable, the encoding of Int_config[0] for PPIs and SPIs, is:</p> <p><b>0</b> Corresponding interrupt is handled using the N-N model.</p> <p><b>1</b> Corresponding interrupt is handled using the 1-N model.</p>

For interrupt ID  $m$ , when DIV and MOD are the integer division and modulo operations:

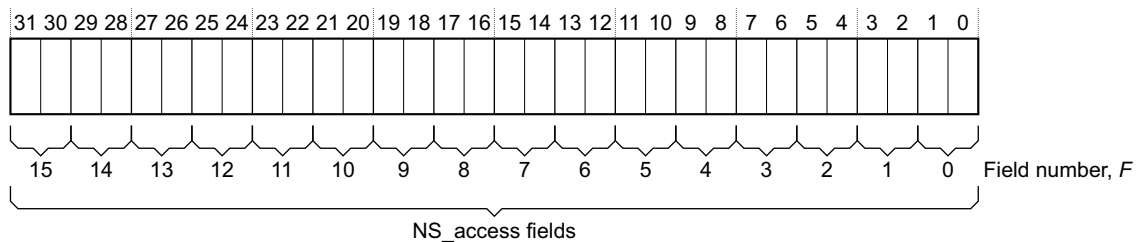
- the corresponding GICD\_ICFGR number,  $n$ , is given by  $n = m \text{ DIV } 16$
- the offset of the required GICD\_ICFGR $n$  is  $(0xC00 + (4*n))$
- the required Priority field in this register,  $F$ , is given by  $F = m \text{ MOD } 16$ , where field 0 refers to register bits [1:0], field 1 refers to bits [3:2], up to field 15 that refers to bits [31:30], see [Figure 4-15 on page 4-109](#).

### 4.3.14 Non-secure Access Control Registers, GICD\_NSACRn

The GICD\_NSACR characteristics are:

<b>Purpose</b>	The GICD_NSACRs enable Secure software to permit Non-secure software on a particular processor to create and manage Group 0 interrupts. They provide an access control for each implemented interrupt.
<b>Usage constraints</b>	These registers can be implemented only if the GIC implements the Security Extensions. These registers are optional Secure registers. If not implemented, the corresponding address space is reserved.
<b>Configurations</b>	These registers are present, optionally, in GICv2. The corresponding address space is reserved in GICv1.  The concept of selective enabling of Non-secure access to Group 0 interrupts applies to SGIs and SPIs.  GICD_NSACR0 is a banked register, with a copy for every processor that has a CPU interface and supports this feature.
<b>Attributes</b>	See the register summary in <a href="#">Table 4-1 on page 4-75</a> .

Figure 4-16 shows the GICD\_NSACR bit assignments:



See the bit assignment table for more information about the properties of each NS\_access[1:0] field.

**Figure 4-16 GICD\_NSACR bit assignments**

Table 4-20 shows the GICD\_NSACR bit assignments:

**Table 4-20 GICD\_NSACR bit assignments**

Bits	Name	Function
[2F+1:2F]	NS_access, Field F	<p>If the corresponding interrupt does not support configurable Non-secure access, the field is RAZ/WI. Otherwise, the field is RW and configures the level of Non-secure access permitted when the interrupt is in Group 0. If the interrupt is in Group 1, this field is ignored. The possible values of the field are:</p> <p>0b00 No Non-secure access is permitted to fields associated with the corresponding interrupt.</p> <p>0b01 Non-secure write access is permitted to fields associated with the corresponding interrupt in the <a href="#">GICD_ISPENDRn</a> registers. A Non-secure write access to <a href="#">GICD_SGIR</a> is permitted to generate a Group 0 SGI for the corresponding interrupt.</p> <p>0b10 Adds Non-secure write access permission to fields associated with the corresponding interrupt in the <a href="#">GICD_ICPENDRn</a> registers. Also adds Non-secure read access permission to fields associated with the corresponding interrupt in the <a href="#">GICD_ISACTIVERn</a> and <a href="#">GICD_ICACTIVERn</a> registers.</p> <p>0b11 Adds Non-secure read and write access permission to fields associated with the corresponding interrupt in the <a href="#">GICD_ITARGETSRn</a> registers.</p>

The GICD\_NSACRn registers do not support PPI accesses, meaning that GICD\_NSACR0 bits [31:16] are RAZ/WI.

For interrupt ID  $m$ , when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD\_NSACR number,  $n$ , is given by  $n = m \text{ DIV } 16$
- the offset of the required GICD\_NSACRn is  $(0xE00 + (4*n))$ .

———— **Note** —————

The address scheme used for a [Remote access](#) is system-defined.

---

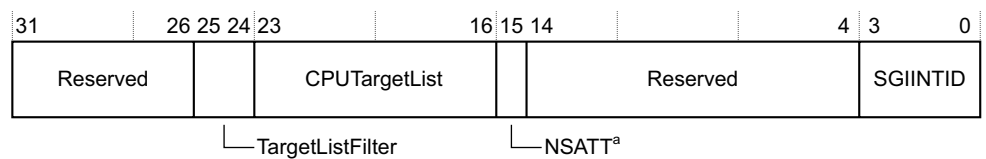


### 4.3.15 Software Generated Interrupt Register, GICD\_SGIR

The GICD\_SGIR characteristics are:

<b>Purpose</b>	Controls the generation of SGIs.
<b>Usage constraints</b>	It is IMPLEMENTATION DEFINED whether the GICD_SGIR has any effect when the forwarding of interrupts by Distributor is disabled by the <a href="#">GICD_CTLR</a> settings.
<b>Configurations</b>	This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.  The NSATT field, bit [15], is implemented only if the GIC implements the Security Extensions.
<b>Attributes</b>	See the register summary in <a href="#">Table 4-1 on page 4-75</a> .

[Figure 4-17](#) shows the GICD\_SGIR bit assignments.



a Implemented only if the GIC implements the Security Extensions, reserved otherwise

**Figure 4-17 GICD\_SGIR bit assignments**

[Table 4-21](#) shows the GICD\_SGIR bit assignments.

**Table 4-21 GICD\_SGIR bit assignments**

Bits	Name	Function
[31:26]	-	Reserved.
[25:24]	TargetListFilter	Determines how the distributor must process the requested SGI: 0b00 Forward the interrupt to the CPU interfaces specified in the CPUTargetList field <sup>a</sup> . 0b01 Forward the interrupt to all CPU interfaces except that of the processor that requested the interrupt. 0b10 Forward the interrupt only to the CPU interface of the processor that requested the interrupt. 0b11 Reserved.
[23:16]	CPUTargetList	When TargetList Filter = 0b00, defines the CPU interfaces to which the Distributor must forward the interrupt. Each bit of CPUTargetList[7:0] refers to the corresponding CPU interface, for example CPUTargetList[0] corresponds to CPU interface 0. Setting a bit to 1 indicates that the interrupt must be forwarded to the corresponding interface. If this field is 0x00 when TargetListFilter is 0b00, the Distributor does not forward the interrupt to any CPU interface.

**Table 4-21 GICD\_SGIR bit assignments (continued)**

Bits	Name	Function
[15]	NSATT	<p>Implemented only if the GIC includes the Security Extensions. Specifies the required security value of the SGI:</p> <p><b>0</b> Forward the SGI specified in the SGIINTID field to a specified CPU interface only if the SGI is configured as Group 0 on that interface.</p> <p><b>1</b> Forward the SGI specified in the SGIINTID field to a specified CPU interfaces only if the SGI is configured as Group 1 on that interface.</p> <p>This field is writable only by a Secure access. Any Non-secure write to the GICD_SGIR generates an SGI only if the specified SGI is programmed as Group 1, regardless of the value of bit[15] of the write. See <i>SGI generation when the GIC implements the Security Extensions</i> for more information.</p> <p>———— <b>Note</b> —————</p> <p>If GIC does not implement the Security Extensions, this field is reserved.</p>
[14:4]	-	Reserved, SBZ.
[3:0]	SGIINTID	The Interrupt ID of the SGI to forward to the specified CPU interfaces. The value of this field is the Interrupt ID, in the range 0-15, for example a value of 0b0011 specifies Interrupt ID 3.

a. When TargetListFilter is 0b00, if the CPUTargetList field is 0x00 the Distributor does not forward the interrupt to any CPU interface.

### SGI generation when the GIC implements the Security Extensions

If the GIC implements the Security Extensions, whether an SGI is forwarded to a processor specified in the write to the GICD\_SGIR depends on:

- whether the write to the GICD\_SGIR is Group 0 (Secure) or Group 1 (Non-secure)
- for a Secure write to the GICD\_SGIR, the value of the GICD\_SGIR.NSATT bit
- whether the specified SGI is configured as Group 0 (Secure) or Group 1 (Non-secure) on the targeted processor.

GICD\_IGROUPR0 holds the security states of the SGIs, see the [GICD\\_IGROUPRn](#) description. In a multiprocessor system, GICD\_IGROUPR0 is banked for each connected processor, so the system configures the security of each SGI independently for each processor. A single write to the GICD\_SGIR can target more than one processor. For each targeted processor, the Distributor determines whether to forward the SGI to the processor.

[Table 4-22](#) shows the truth table for whether the Distributor forwards an SGI to a specified target CPU interface.

**Table 4-22 Truth table for sending an SGI to a target processor**

Status of GICD_SGIR write	NSATT value	SGI configuration for target processor	Forward SGI?
Secure	0	Group 0	Yes
		Group 1	No
	1	Group 0	No
		Group 1	Yes
Non-secure	x	Group 0	No
		Group 1	Yes

### 4.3.16 SGI Clear-Pending Registers, GICD\_CPENDSGIRn

The GICD\_CPENDSGIR characteristics are:

**Purpose** The GICD\_CPENDSGIRs provide a clear-pending bit for each supported SGI and source processor combination. When a processor writes a 1 to a clear-pending bit, the pending state of the corresponding SGI for the corresponding source processor is removed, and no longer targets the processor performing the write. Writing a 0 has no effect. Reading a bit identifies whether the SGI is pending, from the corresponding source processor, on the reading processor.

These registers are used when preserving and restoring GIC state.

———— **Note** ————

In these registers, and in the [GICD\\_SPENDSGIRn](#) registers, an SGI is identified by the combination of SGI number and source processor.

**Usage constraints** A register bit corresponding to an unimplemented SGI is RAZ/WI.

These registers are byte-accessible.

If the GIC implements the Security Extensions:

- a register bit that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses
- if the GIC supports fewer than eight processors, register bits corresponding to the non-implemented processors are RAZ/WI.

———— **Note** ————

- In a multiprocessor implementation, the processor accessing the register can change the SGI pending status only on the corresponding interface. Changing the pending status of an SGI for one target processor does not affect the status of that SGI on any other processor.

- PPIs and SPIs both use the Interrupt Clear-Pending registers, [GICD\\_ICPENDRn](#).

**Configurations** These registers are present only in GICv2. The register locations are reserved in GICv1.

Four SGI Clear-Pending registers are implemented. The registers contain a bit for each of eight possible source processors, for each of the 16 possible SGIs. That is, each register contains eight clear-pending bits for each of four SGIs.

In a multiprocessor implementation, the GICD\_CPENDSGIRn registers are banked for each connected processor.

**Attributes** See the register summary in [Table 4-1 on page 4-75](#).

[Figure 4-18](#) shows the GICD\_CPENDSGIRn bit assignments.

31	24 23	16 15	8 7	0
SGI m+3 clear-pending	SGI m+2 clear-pending	SGI m+1 clear-pending	SGI m clear-pending	

**Figure 4-18 GICD\_CPENDSGIR bit assignments**

Table 4-23 shows the GICD\_CPENDSGIR bit assignments.

**Table 4-23 GICD\_CPENDSGIRn bit assignments**

Bits	Name	Function
[8y+7:8y], for y=0 to 3	SIGI <i>x</i> Clear-pending bits	<p>For each bit:</p> <p><b>Reads</b>     <b>0</b>     SIGI <i>x</i> from the corresponding processor is not pending<sup>a</sup>.</p> <p>                  <b>1</b>     SIGI <i>x</i> from the corresponding processor is pending<sup>a</sup>.</p> <p><b>Writes</b>     <b>0</b>     Has no effect.</p> <p>                  <b>1</b>     Removes the pending state of SIGI <i>x</i> for the corresponding processor.</p> <p>See text for the relation between the SIGI number, <i>x</i>, the GICD_CPENDSGIRn register number, <i>n</i>, and the field number, <i>y</i>.</p> <hr/> <p style="text-align: center;"><b>Note</b></p> <hr/> <p>All accesses relate only to SIGIs that target the processor making the access.</p>

a. Pending interrupts include interrupts that are active and pending.

For SIGI ID *x*, generated by CPU *C* writing to its GICD\_SGIR, when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD\_CPENDSGIR register number, *n*, is given by  $n = x \text{ DIV } 4$
- the offset of the required GICD\_CPENDSGIR is  $(0xF10 + (4*n))$ ;
- the SIGI Clear-pending field offset, *y*, is given by  $y = x \text{ MOD } 4$
- the required bit in the SIGI *x* Clear-pending field is bit *C*.

### 4.3.17 SGI Set-Pending Registers, GICD\_SPENDSGIRn

The GICD\_SPENDSGIR characteristics are:

**Purpose** The GICD\_SPENDSGIRn registers provide a set-pending bit for each supported SGI and source processor combination. When a processor writes a 1 to a set-pending bit, the pending state is applied to the corresponding SGI for the corresponding source processor. Writing a 0 has no effect. Reading a bit identifies whether the SGI is pending, from the corresponding source processor, on the reading processor.

These registers are used when preserving and restoring GIC state.

———— **Note** —————

In these registers, and in the [GICD\\_CPENDSGIRn](#) registers, an SGI is identified by the combination of SGI number and source processor.

**Usage constraints** A register bit corresponding to an unimplemented SGI is RAZ/WI.

These registers are byte-accessible.

If the GIC implements the Security Extensions:

- a register bit that corresponds to a Group 0 interrupt is RAZ/WI to Non-secure accesses
- if the GIC supports fewer than eight processors, register bits corresponding to the non-implemented processors are RAZ/WI.

———— **Note** —————

- In a multiprocessor implementation, the processor accessing the register can change the SGI pending status only on the corresponding interface. Changing the pending status of an SGI for one target processor does not affect the status of that SGI on any other processor.

- PPIs and SPIs both use the Interrupt Set-Pending registers, [GICD\\_ISPENDRn](#).

**Configurations** These registers are present only in GICv2. The register locations are reserved in GICv1.

Four SGI Set-Pending registers are implemented. The registers contain a bit for each of eight possible source processors, for each of the 16 possible SGIs. That is, each register contains eight set-pending bits for each of four SGIs.

In a multiprocessor implementation, the GICD\_SPENDSGIRn registers are banked for each connected processor.

**Attributes** See the register summary in [Table 4-1 on page 4-75](#).

[Figure 4-19](#) shows the GICD\_SPENDSGIRn bit assignments.

31	24 23	16 15	8 7	0
SGI m+3 set-pending	SGI m+2 set-pending	SGI m+1 set-pending	SGI m set-pending	

**Figure 4-19 GICD\_SPENDSGIR bit assignments**

Table 4-24 shows the GICD\_SPENDSGIR bit assignments.

**Table 4-24 GICD\_SPENDSGIRn bit assignments**

Bits	Name	Function
[8y+7:8y], for y=0 to 3	SIGI <i>x</i> Set-pending bits	<p>For each bit:</p> <p><b>Reads</b></p> <p><b>0</b> SIGI <i>x</i> for the corresponding processor is not pending<sup>a</sup>.</p> <p><b>1</b> SIGI <i>x</i> for the corresponding processor is pending<sup>a</sup>.</p> <p><b>Writes</b></p> <p><b>0</b> Has no effect.</p> <p><b>1</b> Adds the pending state of SIGI <i>x</i> for the corresponding processor, if it is not already pending. If SIGI <i>x</i> is already pending for the corresponding processor then the write has no effect.</p> <p>See text for the relation between the SIGI number, <i>x</i>, the GICD_SPENDSGIRn register number, <i>n</i>, and the field number, <i>y</i>.</p> <hr/> <p style="text-align: center;"><b>Note</b></p> <hr/> <p>All accesses relate only to SIGIs that target the processor making the access.</p>

a. Pending interrupts include interrupts that are active and pending.

For SIGI ID *x*, generated by CPU *C* writing to its GICD\_SGIR, when DIV and MOD are the integer division and modulo operations:

- the corresponding GICD\_SPENDSGIR register number, *n*, is given by  $n = x \text{ DIV } 4$
- the offset of the required GICD\_SPENDSGIR is  $(0xF20 + (4*n))$
- the SIGI Set-pending field offset, *y*, is given by  $y = x \text{ MOD } 4$
- the required bit in the SIGI *x* Set-pending field is bit *C*.

### 4.3.18 Identification registers

This architecture specification defines offsets 0xFD0-0xFFC in the Distributor register map as a read-only identification register space. Table 4-25 shows the architecturally-required implementation of the identification register space.

**Table 4-25 The GIC identification register space**

Offset	Name	Type	Reset <sup>a</sup>	Description
0xFD0-0xFE4	-	RO	-	IMPLEMENTATION DEFINED registers
0xFE8	ICPIDR2	RO	- <sup>b</sup>	Peripheral ID2 Register
0xFEC-0xFFC	-	RO	-	IMPLEMENTATION DEFINED registers

- a. The reset value of an IMPLEMENTATION DEFINED register is IMPLEMENTATION DEFINED.
- b. See the register description for information about the architecturally defined bits in this register.

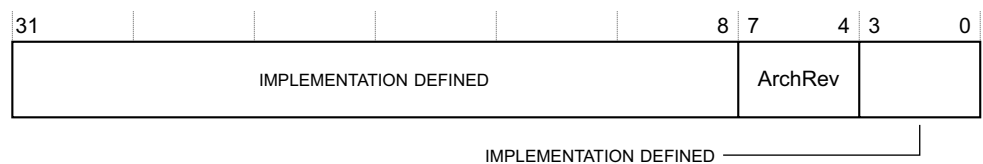
The assignment of this register space, and naming of registers in this space, is consistent with the ARM identification scheme for CoreLink and CoreSight components. ARM implementations of this GIC architecture implement that identification scheme, and ARM strongly recommends that other implementers also use this scheme, to provide a consistent software discovery model, see *The ARM implementation of the GIC Identification Registers* on page 4-120.

#### Peripheral ID2 Register, ICPIDR2

The ICPIDR2 characteristics are:

- Purpose** Provides a four-bit architecturally-defined architecture revision field. The remaining bits of the register are IMPLEMENTATION DEFINED.
- Usage constraints** There are no usage constraints. However, ARM strongly recommends that bits[31:8] of the register are reserved, RAZ.
- Configurations** This register is available in all configurations of the GIC.
- Attributes** See the register summary in Table 4-25.

Figure 4-20 shows the ICPIDR2 bit assignments.



**Figure 4-20 ICPIDR2 bit assignments**

Table 4-26 shows the ICPIDR2 bit assignments.

**Table 4-26 ICPIDR2 bit assignments**

Bits	Name	Function
[31:8]	-	IMPLEMENTATION DEFINED. The CoreLink and CoreSight Peripheral ID Registers scheme requires these bits to be reserved, RAZ, and ARM strongly recommends that implementations follow this scheme.
[7:4]	ArchRev	Revision field for the GIC architecture. The value of this field depends on the GIC architecture version: <ul style="list-style-type: none"> <li>• 0x1 for GICv1</li> <li>• 0x2 for GICv2.</li> </ul>
[3:0]	-	IMPLEMENTATION DEFINED.

### The ARM implementation of the GIC Identification Registers

————— **Note** —————

- The ARM implementation of these registers is consistent with the identification scheme for CoreLink and CoreSight components. This implementation identifies the device as a GIC that implements this architecture. It does not identify the designer or manufacturer of the GIC implementation. For information about the designer and manufacturer of a GIC implementation see the [GICD\\_IIDR](#) and [GICC\\_IIDR](#) descriptions.
- In other contexts, this identification scheme identifies a component in a system. The GIC use of the scheme is different. It identifies only that the device is an implementation of a version of the GIC architecture defined by this specification. Software must read the [GICD\\_IIDR](#) and [GICC\\_IIDR](#) to discover, for example, the implementer and version of the GIC hardware.

Table 4-27 shows the Identification Registers for an ARM implementation of the version of the GIC architecture defined by this specification. ARM recommends other implementers to include the registers described here.

**Table 4-27 Identification Registers for a GIC, with ARM implementation values**

Register <sup>a</sup>	Offset	Bits	ARM implementation	
			Value	Description
Component ID0, ICCIDR0	0xFF0	[7:0]	0x0D	ARM-defined fixed values for the preamble for component discovery.
Component ID1, ICCIDR1	0xFF4	[7:0]	0xF0	
Component ID2, ICCIDR2	0xFF8	[7:0]	0x05	
Component ID3, ICCIDR3	0xFFC	[7:0]	0xB1	
Peripheral ID0, ICPIDR0	0xFE0	[7:0]	0x90	Bits [7:0] of the ARM-defined DevID field.
Peripheral ID1, ICPIDR1	0xFE4	[7:4]	0xB	Bits [3:0] of the ARM-defined ArchID field.
		[3:0]	Example values: <ul style="list-style-type: none"> <li>• 0x3 for ARM GICv1 implementations</li> <li>• 0x4 for ARM GICv2 implementations.</li> </ul>	Bits [11:8] of the ARM-defined DevID field:



**Table 4-27 Identification Registers for a GIC, with ARM implementation values (continued)**

Register <sup>a</sup>	Offset	Bits	ARM implementation	
			Value	Description
Peripheral ID2, ICPIDR2	0xFE8	[7:4]	Architecturally-defined: • 0x1 for GICv1 • 0x2 for GICv2.	ArchRev field.
		[3]	1	ARM-defined UsesJEPcode field.
		[2:0]	0b011	Bits [6:4] of the ARM-defined ArchID field.
Peripheral ID3, ICPIDR3	0xFEC	[3:0]	0x0	Reserved by ARM.
		[7:4]	0x0	ARM-defined Revision field.
Peripheral ID4, ICPIDR4	0xFD0	[3:0]	0x4	ARM-defined ContinuationCode field.
		[7:4]	0x0	Reserved by ARM.
Peripheral ID5, ICPIDR5	0xFD4	[7:0]	0x00	Reserved by ARM.

**Table 4-27 Identification Registers for a GIC, with ARM implementation values (continued)**

Register <sup>a</sup>	Offset	Bits	ARM implementation	
			Value	Description
Peripheral ID6, ICPIDR6	0xFD8	[7:0]	0x00	Reserved by ARM.
Peripheral ID7, ICPIDR7	0xFDC	[7:0]	0x00	Reserved by ARM.

- a. In the ARM implementation, bits [31:8] of each register are reserved. Bits [7:0] of the four Component ID registers together define a conceptual 32-bit Component ID, and bits [7:0] of the eight Peripheral ID registers together define a conceptual 64-bit Peripheral ID. In the GIC implementation, despite their names, Component ID and Peripheral ID refer only to the architecture of the implementation, see the Note at the start of this section for more information.

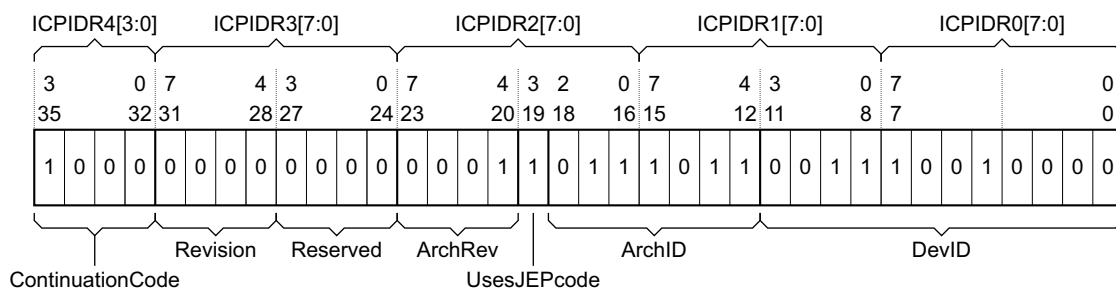
**Note**

Some previous ARM implementations of the GIC did not implement Peripheral ID registers 4-7. Software can use the value of bit [3] of the ICPIDR2 to identify these implementations:

- 0** Legacy format.
- 1** ARM GICv1 or later format.

**The ARM peripheral ID for a GIC**

Together, the Peripheral ID registers ICPIDR0 to ICPIDR7 define an 64-bit peripheral ID. In current ARM implementations, bits [63:36] of that ID are reserved, RAZ. Figure 4-21 shows bits [35:0] of the Peripheral ID for a GIC, and Table 4-28 shows all the fields in the 64-bit Peripheral ID.



**Figure 4-21 ARM Peripheral ID fields for a GIC**

**Table 4-28 Fields in the GIC Peripheral ID, for an ARM implementation**

Name	Bits	Source	Function, ARM-defined fields
-	[39:36]	ICPIDR4[7:4]	Reserved
ContinuationCode	[35:32]	ICPIDR4[3:0]	JEP106 continuation code for ARM
Revision	[31:28]	ICPIDR3[7:4]	Revision field
-	[27:24]	ICPIDR3[3:0]	Reserved
ArchRev	[23:20]	ICPIDR2[7:4]	Architecturally-defined revision number for the ARM GIC architecture, see <a href="#">Peripheral ID2 Register, ICPIDR2 on page 4-119</a>

**Table 4-28 Fields in the GIC Peripheral ID, for an ARM implementation (continued)**

<b>Name</b>	<b>Bits</b>	<b>Source</b>	<b>Function, ARM-defined fields</b>
UsesJEPcode	[19]	ICPIDR2[3]	Indicate that the identifier string uses JEP106 codes to identify ARM as the designer of the architecture
ArchID	[18:12]	ICPIDR2[2:0], ICPIDR1[7:4]	Identifies ARM as the designer of the GIC architecture
DevID	[11:0]	ICPIDR1[3:0], ICPIDR0[7:0]	Identifies the device as a particular GIC implementation

## 4.4 CPU interface register descriptions

The following sections describe the CPU interface registers:

- *CPU Interface Control Register, GICC\_CTLR* on page 4-125
- *Interrupt Priority Mask Register, GICC\_PMR* on page 4-131
- *Binary Point Register, GICC\_BPR* on page 4-133
- *Interrupt Acknowledge Register, GICC\_IAR* on page 4-135
- *End of Interrupt Register, GICC\_EOIR* on page 4-138
- *Running Priority Register, GICC\_RPR* on page 4-142
- *Highest Priority Pending Interrupt Register, GICC\_HPPIR* on page 4-143
- *Aliased Binary Point Register, GICC\_ABPR* on page 4-145
- *Aliased Interrupt Acknowledge Register, GICC\_AIAR* on page 4-146
- *Aliased End of Interrupt Register, GICC\_AEOIR* on page 4-147
- *Aliased Highest Priority Pending Interrupt Register, GICC\_AHPPIR* on page 4-148
- *Active Priorities Registers, GICC\_APRn* on page 4-149
- *Non-secure Active Priorities Registers, GICC\_NSAPRn* on page 4-151
- *CPU Interface Identification Register, GICC\_IIDR* on page 4-152
- *Deactivate Interrupt Register, GICC\_DIR* on page 4-153.

See *CPU interface register map* on page 4-76 for address offset and reset information for these registers.

#### 4.4.1 CPU Interface Control Register, GICC\_CTLR

The GICC\_CTLR characteristics are:

**Purpose** Enables the signaling of interrupts by the CPU interface to the connected processor, and provides additional top-level control of the CPU interface. In a GICv2 implementation, this includes control of the *end of interrupt* (EOI) behavior.

———— **Note** ————

In a GICv2 implementation that includes the GIC Security Extensions, independent EOI controls are provided for:

- Accesses from Secure state. This control applies to the handling of both Group 0 and Group 1 interrupts.
- Accesses from Non-secure state. This control only applies to the handling of Group 1 interrupts.

—————  
The EOI controls affect the behavior of accesses to [GICC\\_EOIR](#) and [GICC\\_DIR](#). See the register descriptions for more information.

**Usage constraints** If the GIC implements the Security Extensions with support for configuration lockdown, the system can prevent write access to certain register fields in the Secure GICC\_CTLR, see [Configuration lockdown on page 4-82](#).

**Configurations** If the implementation supports interrupt grouping, this register provides independent control of Group 0 and Group 1 interrupts.

If the GIC implements the Security Extensions:

- this register is banked to provide Secure and Non-secure copies, see [Register banking on page 4-77](#)
- the register bit assignments are different in the Secure and Non-secure copies of the register, and:
  - the Secure copy of the register can control both Group 0 and Group 1 interrupts
  - the Non-secure copy of the register can control only Group 1 interrupts.

**Attributes** See the register summary in [Table 4-2 on page 4-76](#).

[Figure 4-22 on page 4-126](#) and [Table 4-29 on page 4-126](#) shows the GICC\_CTLR bit assignments for a GICv1 implementation, for

- an implementation that does not include the Security Extensions
- the Non-secure copy of the register, in an implementation that includes the Security Extensions.

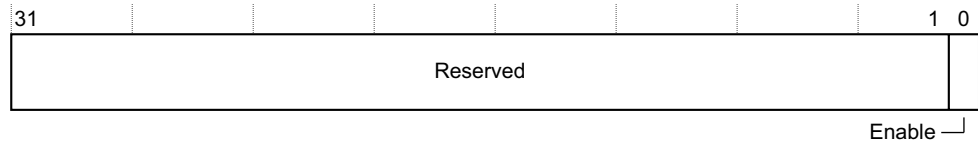


Figure 4-22 GICC\_CTLR bit assignments, GICv1 without Security Extensions or Non-secure

Table 4-29 GICC\_CTLR bit assignments, GIC1 without Security Extensions or Non-secure

Bits	Name	Function
[31:1]	-	Reserved
[0]	Enable	Enable for the signaling of Group 1 interrupts by the CPU interface to the connected processor. <b>0</b> Disable signaling of interrupts <b>1</b> Enable signaling of interrupts.

**Note**

- When this bit is cleared to 0, the CPU interface ignores any pending interrupt forwarded to it. When this bit is set to 1, the CPU interface starts to process pending interrupts that are forwarded to it. There is a small but finite time required for a change to take effect.
- On a GICv1 implementation that does not include the Security Extensions, this bit controls the signaling of all interrupts by the CPU interface to the connected processor.

See [Enabling and disabling the Distributor and CPU interfaces on page 4-77](#) for more information about this bit.

Figure 4-23 and Table 4-30 show the GICC\_CTLR bit assignments for the Non-secure copy of the register in a GIC v2 implementation that includes the Security Extensions,

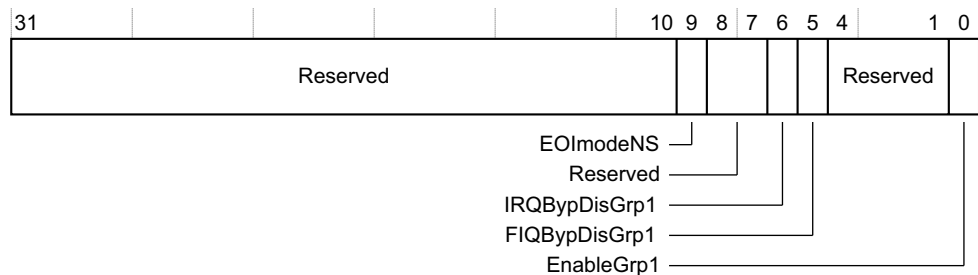


Figure 4-23 GICC\_CTLR bit assignments, GICv2 with Security Extensions, Non-secure copy

Table 4-30 GICC\_CTLR bit assignments, GIC2 with Security Extensions, Non-secure copy

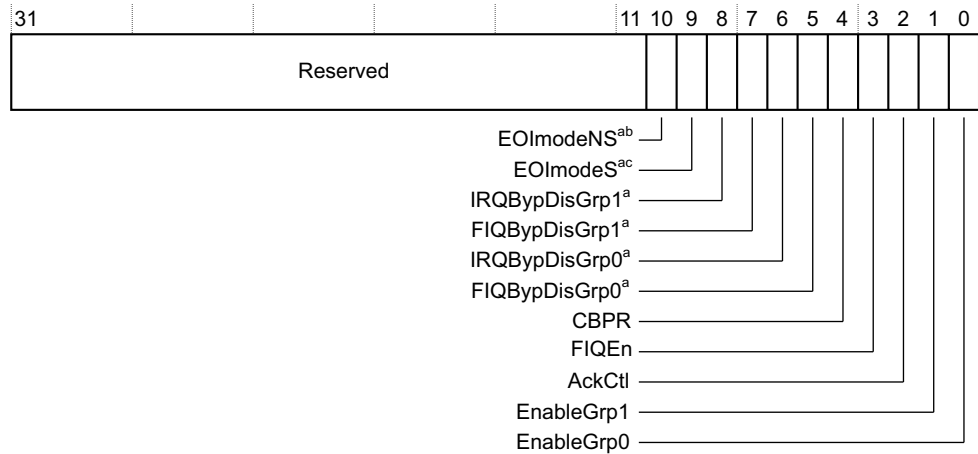
Bits	Name	Function
[31:10]	-	Reserved
[9]	EOImodeNS	Controls the behavior of Non-secure accesses to the <a href="#">GICC_EOIR</a> and <a href="#">GICC_DIR</a> registers: <b>0</b> <a href="#">GICC_EOIR</a> has both priority drop and deactivate interrupt functionality. Accesses to the <a href="#">GICC_DIR</a> are UNPREDICTABLE. <b>1</b> <a href="#">GICC_EOIR</a> has priority drop functionality only. The <a href="#">GICC_DIR</a> register has deactivate interrupt functionality. See <a href="#">Behavior of writes to GICC_EOIR, GICv2 on page 4-140</a> for more information.
[8:7]	-	Reserved.

**Table 4-30 GICC\_CTLR bit assignments, GIC2 with Security Extensions, Non-secure copy (continued)**

Bits	Name	Function
[6]	IRQByDisGrp1	<p>When the signaling of IRQs by the CPU interface is disabled, this bit partly controls whether the bypass IRQ signal is signaled to the processor:</p> <p><b>0</b> Bypass IRQ signal is signaled to the processor</p> <p><b>1</b> Bypass IRQ signal is not signaled to the processor.</p> <p>See <a href="#">Interrupt signal bypass, and GICv2 bypass disable on page 2-27</a> for more information.</p>
[5]	FIQByDisGrp1	<p>When the signaling of FIQs by the CPU interface is disabled, this bit partly controls whether the bypass FIQ signal is signaled to the processor:</p> <p><b>0</b> Bypass FIQ signal is signaled to the processor</p> <p><b>1</b> Bypass FIQ signal is not signaled to the processor.</p> <p>See <a href="#">Interrupt signal bypass, and GICv2 bypass disable on page 2-27</a> for more information.</p>
[4:1]	-	Reserved
[0]	EnableGrp1	<p>Enable for the signaling of Group 1 interrupts by the CPU interface to the connected processor.</p> <p><b>0</b> Disable signaling of interrupts</p> <p><b>1</b> Enable signaling of interrupts.</p> <p style="text-align: center;"><b>———— Note ————</b></p> <p>When this bit is set to 0, the CPU interface ignores any pending Group 1 interrupt forwarded to it. When this bit is set to 1, the CPU interface starts to process pending Group 1 interrupts that are forwarded to it. There is a small but finite time required for a change to take effect.</p> <p>See <a href="#">Enabling and disabling the Distributor and CPU interfaces on page 4-77</a> for more information about this bit.</p>

Figure 4-24 on page 4-128 and Table 4-31 on page 4-128 show the GICC\_CTLR bit assignments for:

- a GICv2 implementation, for:
  - an implementation that does not include the Security Extensions
  - the Secure copy of the register, in an implementation that includes the Security Extensions
- a GICv1 implementation that includes the Security Extensions, for the Secure copy of the register.



- a GICv2 only
- b When the GIC implementation includes the Security Extensions
- c EOImode in a GIC implementation that does not include the Security Extensions

**Figure 4-24 GICC\_CTLR bit assignments, GICv2 without Security Extensions or Secure**

**Table 4-31 GICC\_CTLR bit assignments, GICv2 without Security Extensions or Secure**

Bit	Name	Function
[31:11]	-	Reserved.
[10]	EOImodeNS <sup>ab</sup>	Alias of EOImodeNS from the Non-secure copy of this register, see <a href="#">Table 4-30 on page 4-126</a> . In a GICv2 implementation that does not include the Security Extensions, and in a GICv1 implementation, this bit is reserved.
[9]	EOImodeS <sup>a</sup>	Controls the behavior of accesses to <a href="#">GICC_EOIR</a> and <a href="#">GICC_DIR</a> registers. In a GIC implementation that includes the Security Extensions, this control applies only to Secure accesses, and the EOImodeNS bit controls the behavior of Non-secure accesses to these registers: <b>0</b> <a href="#">GICC_EOIR</a> has both priority drop and deactivate interrupt functionality. Accesses to the <a href="#">GICC_DIR</a> are UNPREDICTABLE. <b>1</b> <a href="#">GICC_EOIR</a> has priority drop functionality only. <a href="#">GICC_DIR</a> has deactivate interrupt functionality. See <a href="#">Behavior of writes to GICC_EOIR, GICv2 on page 4-140</a> for more information.  ———— <b>Note</b> ———— This bit is called EOImode in a GIC implementation that does not include the Security Extensions.  In a GICv1 implementation, this bit is reserved.
[8]	IRQBypDisGrp1 <sup>a</sup>	Alias of IRQBypDisGrp1 from the Non-secure copy of this register, see <a href="#">Table 4-30 on page 4-126</a> . In a GICv1 implementation, this bit is reserved.
[7]	FIQBypDisGrp1 <sup>a</sup>	Alias of FIQBypDisGrp1 from the Non-secure copy of this register, see <a href="#">Table 4-30 on page 4-126</a> . In a GICv1 implementation, this bit is reserved.



**Table 4-31 GICC\_CTLR bit assignments, GICv2 without Security Extensions or Secure (continued)**

Bit	Name	Function
[6]	IRQByDisGrp0 <sup>a</sup>	<p>When the signaling of IRQs by the CPU interface is disabled, this bit partly controls whether the bypass IRQ signal is signaled to the processor:</p> <p><b>0</b> Bypass IRQ signal is signaled to the processor</p> <p><b>1</b> Bypass IRQ signal is not signaled to the processor.</p> <p>See <a href="#">Interrupt signal bypass, and GICv2 bypass disable on page 2-27</a> and <a href="#">Power management, GIC v2 on page 2-31</a> for more information.</p> <p>In a GICv1 implementation, this bit is reserved.</p>
[5]	FIQByDisGrp0 <sup>a</sup>	<p>When the signaling of FIQs by the CPU interface is disabled, this bit partly controls whether the bypass FIQ signal is signaled to the processor:</p> <p><b>0</b> Bypass FIQ signal is signaled to the processor</p> <p><b>1</b> Bypass FIQ signal is not signaled to the processor.</p> <p>See <a href="#">Interrupt signal bypass, and GICv2 bypass disable on page 2-27</a> and <a href="#">Power management, GIC v2 on page 2-31</a> for more information.</p> <p>In a GICv1 implementation, this bit is reserved.</p>
[4]	CBPR <sup>c</sup>	<p>Controls whether the <b>GICC_BPR</b> provides common control to Group 0 and Group 1 interrupts.</p> <p><b>0</b> To determine any preemption, use:</p> <ul style="list-style-type: none"> <li>• the <b>GICC_BPR</b> for Group 0 interrupts</li> <li>• the <b>GICC_ABPR</b> for Group 1 interrupts.</li> </ul> <p><b>1</b> To determine any preemption use the <b>GICC_BPR</b> for both Group 0 and Group 1 interrupts.</p> <p>See <a href="#">The effect of interrupt grouping on priority grouping on page 3-57</a> for more information about how GICC_CTLR.CBPR affects accesses to <b>GICC_BPR</b> and <b>GICC_ABPR</b>.</p>
[3]	FIQEn	<p>Controls whether the CPU interface signals Group 0 interrupts to a target processor using the FIQ or the IRQ signal.</p> <p><b>0</b> Signal Group 0 interrupts using the IRQ signal.</p> <p><b>1</b> Signal Group 0 interrupts using the FIQ signal.</p> <p>The GIC always signals Group 1 interrupts using the IRQ signal.</p> <hr/> <p style="text-align: center;"><b>Note</b></p> <hr/> <p>If using software written for a system that includes an implementation of GICv1 without the Security Extensions, all interrupts are signaled by the IRQ signal. In such systems, ensure that this bit is 0. This is the default value. See <a href="#">Example GIC usage models on page 3-68</a> for more information.</p>

**Table 4-31 GICC\_CTLR bit assignments, GICv2 without Security Extensions or Secure (continued)**

Bit	Name	Function
[2]	AckCtl	<p>When the highest priority pending interrupt is a Group 1 interrupt, determines both:</p> <ul style="list-style-type: none"> <li>• whether a read of <a href="#">GICC_IAR</a> acknowledges the interrupt, or returns a spurious interrupt ID</li> <li>• whether a read of <a href="#">GICC_HPPIR</a> returns the ID of the highest priority pending interrupt, or returns a spurious interrupt ID.</li> </ul> <p><b>0</b> If the highest priority pending interrupt is a Group 1 interrupt, a read of the <a href="#">GICC_IAR</a> or the <a href="#">GICC_HPPIR</a> returns an Interrupt ID of 1022. A read of the <a href="#">GICC_IAR</a> does not acknowledge the interrupt, and has no effect on the pending status of the interrupt.</p> <p><b>1</b> If the highest priority pending interrupt is a Group 1 interrupt, a read of the <a href="#">GICC_IAR</a> or the <a href="#">GICC_HPPIR</a> returns the Interrupt ID of the Group 1 interrupt. A read of <a href="#">GICC_IAR</a> acknowledges and <a href="#">Activates</a> the interrupt.</p> <p>In a GIC implementation that includes the Security Extensions, this control affects only the behavior of Secure register accesses.</p> <p>For more information, see:</p> <ul style="list-style-type: none"> <li>• <a href="#">The effect of interrupt grouping on interrupt acknowledgement on page 3-50</a></li> <li>• <a href="#">Interrupt grouping and interrupt prioritization on page 3-53</a></li> <li>• <a href="#">Behavior of writes to GICC_EOIR, GICv1 with Security Extensions on page 4-139</a></li> <li>• <a href="#">Effect of interrupt grouping and the Security Extensions on reads of the GICC_HPPIR on page 4-143.</a></li> </ul> <p style="text-align: center;"><b>————— Note —————</b></p> <p>ARM deprecates use of GICC_CTLR.AckCtl, and strongly recommends using a software model where GICC_CTLR.AckCtl is set to 0. See <a href="#">Enabling and disabling the Distributor and CPU interfaces on page 4-77</a> for more information about the effects of setting this bit.</p>
[1]	EnableGrp1 <sup>d, e</sup>	<p>Enable for the signaling of Group 1 interrupts by the CPU interface to the connected processor:</p> <p><b>0</b> Disable signaling of Group 1 interrupts.</p> <p><b>1</b> Enable signaling of Group 1 interrupts.</p>
[0]	EnableGrp0 <sup>e, f</sup>	<p>Enable for the signaling of Group 0 interrupts by the CPU interface to the connected processor:</p> <p><b>0</b> Disable signaling of Group 0 interrupts.</p> <p><b>1</b> Enable signaling of Group 0 interrupts.</p>

- a. GICv2 only, see [Power management, GIC v2 on page 2-31](#) for more information.
- b. When the GIC implements the Security Extensions.
- c. SBPR in GICv1.
- d. EnableNS in GICv1.
- e. There is a small but finite time required for a change in the value of this register to take effect.
- f. EnableS in GICv1.

For more information about the optional support for interrupt signal bypass, including the GICv2 disable controls of this functionality, see [Interrupt signal bypass, and GICv2 bypass disable on page 2-27](#).

## 4.4.2 Interrupt Priority Mask Register, GICC\_PMR

The GICC\_PMR characteristics are:

**Purpose** Provides an interrupt priority filter. Only interrupts with higher priority than the value in this register are signaled to the processor.

———— **Note** —————

Higher priority corresponds to a lower Priority field value.

**Usage constraints** If the GIC implements the Security Extensions then:

- a Non-secure access to this register can only read or write a value that corresponds to the lower half of the priority range, see *Interrupt grouping and interrupt prioritization* on page 3-53.
- if a Secure write has programmed the GICC\_PMR with a value that corresponds to a value in the upper half of the priority range then:
  - any Non-secure read of the GICC\_PMR returns 0x00, regardless of the value held in the register
  - any Non-secure write to the GICC\_PMR is ignored.

For more information see *Non-secure access to register fields for Group 0 interrupt priorities* on page 4-81.

When determining interrupt preemption, the priority value can be split into two parts, using the Binary Point register, GICC\_BPR.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions, this register is Common.

**Attributes** See the register summary in [Table 4-2 on page 4-76](#).

[Figure 4-25](#) shows the GICC\_PMR bit assignments.



**Figure 4-25 GICC\_PMR bit assignments**

[Table 4-32](#) shows the GICC\_PMR Register bit assignments.

**Table 4-32 GICC\_PMR Register bit assignments**

Bits	Name	Function
[31:8]	-	Reserved.
[7:0]	Priority	<p>The priority mask level for the CPU interface. If the priority of an interrupt is higher than the value indicated by this field, the interface signals the interrupt to the processor.</p> <p>If the GIC supports fewer than 256 priority levels then some bits are RAZ/WI, as follows:</p> <p><b>128 supported levels</b> Bit [0] = 0.</p> <p><b>64 supported levels</b> Bit [1:0] = 0b00.</p> <p><b>32 supported levels</b> Bit [2:0] = 0b000.</p> <p><b>16 supported levels</b> Bit [3:0] = 0b0000.</p> <p>For more information see <a href="#">Interrupt prioritization</a> on page 3-44.</p>

The following pseudocode shows the effects of the GIC Security Extensions on accesses to this register:

```
// MaskRegRead()
// =====

bits(8) MaskRegRead()

    read_value = GICC_PMR<7:0>;
    if NS_access then // A non-secure GIC access.
        if read_value<7> == '0' then
            read_value = '00000000'; // A secure priority value, RAZ
        else
            read_value = LSL((read_value AND P_MASK), 1);
    return(read_value);

// MaskRegWrite()
// =====

MaskRegWrite(bits(8) value)

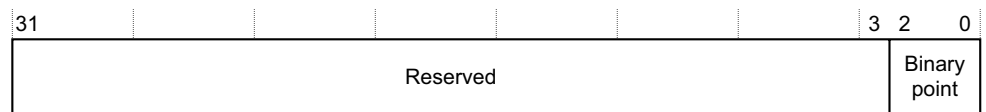
    if NS_access then // A non-secure GIC access.
        mod_write_val = ('10000000' OR LSR(value,1)) AND P_MASK;
        if GICC_PMR<7> == '1' then // Non-secure execution can only update the
            GICC_PMR[cpu_id]<7:0> = mod_write_val; // Priority Mask Register if the current
            // value is in the range 0x80 to 0xFF
        else
            IgnoreWriteRequest();
    else // A secure GIC access
        GICC_PMR<7:0> = value AND P_MASK;
```

### 4.4.3 Binary Point Register, GICC\_BPR

The GICC\_BPR characteristics are:

- Purpose** The register defines the point at which the priority value fields split into two parts, the *group priority* field and the *subpriority* field. The group priority field is used to determine interrupt preemption. For more information see [Preemption on page 3-45](#) and [Priority grouping on page 3-45](#).
- Usage constraints** The minimum binary point value is IMPLEMENTATION DEFINED in the range:
- 0-3 if the implementation does not include the GIC Security Extensions, and for the Secure copy of the register if the implementation includes the Security Extensions
  - 1-4 for the Non-secure copy of the register.
- An attempt to program the binary point field to a value less than the minimum value sets the field to the minimum value. On a reset, the binary point field is set to the minimum supported value.
- Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions:
- this register is banked to provide Secure and Non-secure copies, see [Register banking on page 4-77](#)
  - the GICC\_ABPR is an alias of the Non-secure copy of GICC\_BPR
  - the GICC\_CTLR.CBPR bit affects the view of the Non-secure GICC\_BPR.
- In any GICv2 implementation, or in a GICv1 implementation that includes the Security Extensions, GICC\_CTLR.CBPR controls whether the Secure copy of the GICC\_BPR, or the GICC\_ABPR, is used for the preemption of Group 1 interrupts.
- See [Priority grouping on page 3-45](#) and [The effect of interrupt grouping on priority grouping on page 3-57](#) for more information.
- Attributes** See the register summary in [Table 4-2 on page 4-76](#).

[Figure 4-26](#) shows the GICC\_BPR bit assignments.



**Figure 4-26 GICC\_BPR bit assignments**

[Table 4-33](#) shows the GICC\_BPR bit assignments.

**Table 4-33 GICC\_BPR bit assignments**

Bits	Name	Function
[31:3]	-	Reserved.
[2:0]	Binary point	<p>The value of this field controls how the 8-bit interrupt priority field is split into a group priority field, used to determine interrupt preemption, and a subpriority field. For how this field determines the interrupt priority bits assigned to the group priority field see:</p> <ul style="list-style-type: none"> <li>• <a href="#">Table 3-7 on page 3-57</a>, for the processing of Group 1 interrupts on a GIC that supports interrupt grouping, when the GICC_CTLR.CBPR bit is set to 1</li> <li>• <a href="#">Table 3-2 on page 3-46</a>, for all other cases.</li> </ul> <p>See <a href="#">Priority grouping on page 3-45</a> for more information.</p>

———— **Note** —————

Aliasing the Non-secure GICC\_BPR as the [GICC\\_ABPR](#) means that, in a multiprocessor system, a processor that can make only Secure accesses to the GIC can access the GICC\_ABPR, to configure the preemption setting for Group 1 interrupts.

The following pseudocode shows the effects of the GIC Security Extensions on accesses to this register:

```
// BinaryPointRegWrite()
// =====

BinaryPointRegWrite(bits(3) value)

    if NS_access && GICC_CTLR.CBPR == '1' then
        IgnoreWriteRequest();
    else
        GICC_BPR<2:0> = value;           // Banked register

bits(3) BinaryPointRegRead()

    read_value = GICC_BPR<2:0>;         // Banked register
    if NS_access && GICC_CTLR.CBPR == '1' then
        read_value = GICC_BPR_Secure; // The secure copy of the BPR
        if read_value != 7 then
            read_value = read_value + 1;
    return(read_value);
```

#### 4.4.4 Interrupt Acknowledge Register, GICC\_IAR

The GICC\_IAR characteristics are:

- Purpose** The processor reads this register to obtain the interrupt ID of the signaled interrupt. This read acts as an acknowledge for the interrupt.
- Usage constraints** When GICC\_CTLR.AckCtl is set to 0 in a GICv2 implementation that does not include the Security Extensions, if the highest priority pending interrupt is in Group 1, the interrupt ID 1022 is returned.
- Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions:
- this register is Common.
  - the GICC\_AIAR is an alias of the Non-secure view of this register.
- Attributes** See the register summary in [Table 4-2 on page 4-76](#).

[Figure 4-27](#) shows the IAR bit assignments.



**Figure 4-27 GICC\_IAR bit assignments**

[Table 4-34](#) shows the IAR bit assignments.

**Table 4-34 GICC\_IAR bit assignments**

Bit	Name	Function
[31:13]	-	Reserved.
[12:10]	CPUID	For SGIs in a multiprocessor implementation, this field identifies the processor that requested the interrupt. It returns the number of the CPU interface that made the request, for example a value of 3 means the request was generated by a write to the <a href="#">GICD_SGIR</a> on CPU interface 3. For all other interrupts this field is RAZ.
[9:0]	Interrupt ID	The interrupt ID.

A read of the GICC\_IAR returns the interrupt ID of the highest priority pending interrupt for the CPU interface. The read returns a spurious interrupt ID of 1023 if any of the following apply:

- forwarding of interrupts by the Distributor to the CPU interface is disabled
- signaling of interrupts by the CPU interface to the connected processor is disabled
- no pending interrupt on the CPU interface has sufficient priority for the interface to signal it to the processor.

———— **Note** ————

The following sequence of events is an example of when the GIC returns an interrupt ID of 1023, and shows how reads of the GICC\_IAR can be timing critical:

1. A peripheral asserts a level-sensitive interrupt.
2. The interrupt has sufficient priority and therefore the GIC signals it to a targeted processor.
3. The peripheral deasserts the interrupt. Because there is no other pending interrupt of sufficient priority, the GIC deasserts the interrupt request to the processor.

4. Before it has recognized the deassertion of the interrupt request from stage 3, the targeted processor reads the GICC\_IAR. Because there is no interrupt with sufficient priority to signal to the processor, the GIC returns the spurious ID value of 1023.

---

The determination of the returned interrupt ID is more complex if the GIC supports interrupt grouping, see *Effect of interrupt grouping on reads of the GICC\_IAR*.

A non-spurious interrupt ID returned by a read of the GICC\_IAR is called a valid interrupt ID.

When the GIC returns a valid interrupt ID to a read of the GICC\_IAR it treats the read as an acknowledge of that interrupt and, as a side-effect of the read, changes the interrupt status from pending to active, or to active and pending if the pending state of the interrupt persists. Normally, the pending state of an interrupt persists only if the interrupt is level-sensitive and remains asserted.

For every read of a valid Interrupt ID from the GICC\_IAR, the connected processor must perform a matching write to the GICC\_EOIR.

---

**Note**

- For compatibility with possible extensions to the GIC architecture specification, ARM recommends that software preserves the entire register value read from the GICC\_IAR, and writes that value back to the GICC\_EOIR when it has completed its processing of the interrupt.
  - Although multiple target processors might attempt to read the GICC\_IAR at any time, in GICv2 only one processor can obtain a valid interrupt ID, see *Implications of the 1-N model on page 3-41* for more information.
- 

### Effect of interrupt grouping on reads of the GICC\_IAR

---

**Note**

This section does not apply to GICV\_IAR, the corresponding register in the virtual CPU interface.

---

When a GIC implementation supports interrupt grouping, whether a read of the GICC\_IAR returns a valid interrupt ID depends on:

- whether there is a pending interrupt of sufficient priority for it to be signaled to the processor, and if so, whether:
  - the highest priority pending interrupt is a Group 0 or a Group 1 interrupt
  - interrupt signaling is enabled for that interrupt group.
- if the GIC implements the Security Extensions, whether the GICC\_IAR read access is Secure or Non-secure
- the value of the GICC\_CTLR.AckCtl bit.

Reads of the GICC\_IAR that do not return a valid interrupt ID returns a spurious interrupt ID, ID 1022 or 1023, see *Special interrupt numbers when a GIC supports interrupt grouping on page 3-50*. Table 4-35 shows all possible GICC\_IAR reads for a GIC that supports interrupt grouping on a CPU interface that implements the Security Extensions. For a GICv2 CPU interface that does not implement the Security Extensions, all entries except those for Non-secure GICC\_IAR reads apply.

**Table 4-35 Effect of interrupt grouping and the Security Extensions on reads of GICC\_IAR**

State	GICC_IAR read	GICC_CTLR.AckCtl	Returned interrupt ID
Highest priority pending interrupt <sup>a</sup> is Group 1	Non-secure	x	ID of Group 1 interrupt
	Secure	1	ID of Group 1 interrupt
		0	Interrupt ID 1022



**Table 4-35 Effect of interrupt grouping and the Security Extensions on reads of GICC\_IAR (continued)**

State	GICC_IAR read	GICC_CTLR.AckCtl	Returned interrupt ID
Highest priority pending interrupt <sup>a</sup> is Group 0	Non-secure	x	Interrupt ID 1023
	Secure	x	ID of Group 0 interrupt
No pending interrupts <sup>a</sup>	x	x	Interrupt ID 1023
Interrupt signaling of the required interrupt group by CPU interface disabled	x	x	Interrupt ID 1023

a. Of sufficient priority to be signaled to the processor if signaling by the CPU interface is enabled.

The following pseudocode shows the effects of the GIC Security Extensions on accesses to this register:

```
// ReadGICC_IAR()
// =====
//
// Value of GICC_IAR read by a CPU access
//

bits(32) ReadGICC_IAR(integer cpu_id)

    pendID = HighestPriorityPendingInterrupt(cpu_id);

    if ( IsGrp0Int(pendID) && (GICD_CTLR.EnableGrp0 == '0' || GICC_CTLR.EnableGrp0 == '0')) ||
        (!IsGrp0Int(pendID) && (GICD_CTLR.EnableGrp1 == '0' || GICC_CTLR.EnableGrp1 == '0'))
    then
        pendID = 1023; // If the highest priority isn't enabled, then no interrupt

    if pendID != 1023 then // An enabled interrupt is pending
        if IsGrp0Int(pendID) then // Highest priority is Group 0
            if NS_access then
                pendID = 1023;
            else // Highest priority is Group 1
                if !NS_access && (GICC_CTLR[cpu_id].AckCtl == '0') then
                    pendID = 1022;

        cpuID = 0; // Must be zero for non-SGI interrupts

    if pendID < 16 then // 0 .. 15 are Software Generated Interrupts
        sgiID = SGI_CpuID(pendID); // value is IMPLEMENTATION DEFINED

    if pendID < 1020 then // Check that it is not a spurious interrupt
        AcknowledgeInterrupt(pendID); // Set active and attempt to clear pending

    rval = 0;
    rval<12:10> = sgiID;
    rval<9:0> = pendID;

    return(rval);
```

#### 4.4.5 End of Interrupt Register, GICC\_EOIR

The GICC\_EOIR characteristics are:

**Purpose** A processor writes to this register to inform the CPU interface either:

- that it has completed the processing of the specified interrupt
- in a GICv2 implementation, when the appropriate [GICC\\_CTLR.EOImode](#) bit is set to 1, to indicate that the interface should perform priority drop for the specified interrupt.

See [Priority drop and interrupt deactivation on page 3-38](#) for more information.

**Usage constraints** A write to this register must correspond to the most recent valid read from an Interrupt Acknowledge Register. A valid read is a read that returns a valid interrupt ID, that is not a spurious interrupt ID.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.

**Attributes** See [Table 4-2 on page 4-76](#).

[Figure 4-28](#) shows the GICC\_EOIR bit assignments.



**Figure 4-28 GICC\_EOIR bit assignments**

[Table 4-36](#) shows the GICC\_EOIR bit assignments.

**Table 4-36 GICC\_EOIR bit assignments**

Bits	Name	Function
[31:13]	-	Reserved.
[12:10]	CPUID	On a multiprocessor implementation, if the write refers to an SGI, this field contains the CPUID value from the corresponding <a href="#">GICC_IAR</a> access. In all other cases this field SBZ.
[9:0]	EOINTID	The Interrupt ID value from the corresponding <a href="#">GICC_IAR</a> access.

See [Priority drop and interrupt deactivation on page 3-38](#) for more information about the effect of a write to this register.

For every read of a valid Interrupt ID from the [GICC\\_IAR](#), the connected processor must perform a matching write to the [GICC\\_EOIR](#). The value written to the [GICC\\_EOIR](#) must be the interrupt ID read from the [GICC\\_IAR](#).

If a read of the [GICC\\_IAR](#) returns the ID of a spurious interrupt, software does not have to make a corresponding write to the [GICC\\_EOIR](#). If software writes the ID of a spurious interrupt to the [GICC\\_EOIR](#), the GIC ignores that write.

**Note**

For compatibility with possible extensions to the GIC architecture specification, ARM recommends that software preserves the entire register value read from the [GICC\\_IAR](#) when it acknowledges the interrupt, and uses that entire value for its corresponding write to the [GICC\\_EOIR](#).

For nested interrupts, the order of writes to GICC\_EOIR must be the reverse of the order of interrupt acknowledgement. Behavior is UNPREDICTABLE if either:

- the ordering constraints on reads from the GICC\_IAR and writes to the GICC\_EOIR are not maintained
- the value in a write to the GICC\_EOIR does not match an active interrupt, or the ID of a spurious interrupt.

The effect of writing to GICC\_EOIR with a valid interrupt ID is UNPREDICTABLE if any of the following apply:

- the value written does not match the last valid interrupt value read from the Interrupt Acknowledge register
- there is no outstanding acknowledged interrupt
- the indicated interrupt has already been subject to an EOI request.

For any implementation other than a GICv1 implementation without the GIC Security Extensions, see one of the following sections for more information:

- [Behavior of writes to GICC\\_EOIR, GICv1 with Security Extensions.](#)
- [Behavior of writes to GICC\\_EOIR, GICv2 on page 4-140.](#)

### Behavior of writes to GICC\_EOIR, GICv1 with Security Extensions

If a CPU interface on a GICv1 implementation implements the GIC Security Extensions, whether a write to the GICC\_EOIR removes the active status of the identified interrupt depends on:

- whether the identified interrupt is Group 0 or Group 1
- whether the GICC\_EOIR write is Secure or Non-secure
- the value of the GICC\_CTLR.AckCtl bit.

Table 4-37 shows all possible results of a write to the GICC\_EOIR.

**Table 4-37 Effect of the Security Extensions on writes to GICC\_EOIR**

Interrupt status	GICC_EOIR write	GICC_CTLR.AckCtl	Active status removed
Group 0	Non-secure	x	No
	Secure	x	Yes
Group 1	Non-secure	x	Yes
	Secure	1	Yes
	Secure	0	UNPREDICTABLE

When GICC\_CTLR.AckCtl == 0, the ordering requirement for GICC\_EOIR writes relative to GICC\_IAR reads applies independently for Secure and Non-secure register accesses. This means:

- a Secure write to GICC\_EOIR must correspond to the most recent Secure read of GICC\_IAR
- a Non-secure write to GICC\_EOIR must correspond to the most recent Non-secure read of GICC\_IAR
- a Secure write to the GICC\_AEOIR must correspond to the most recent Secure read of the GICC\_AIAR.

When GICC\_CTLR.AckCtl == 1, the ordering requirement for Secure GICC\_EOIR writes relative to GICC\_IAR reads takes no account of the security level of the GICC\_IAR accesses. This means that a Secure write to GICC\_EOIR must correspond to the most recent read of GICC\_IAR, regardless of the security level of that read of GICC\_IAR.

#### ———— Note ————

The value of GICC\_CTLR.AckCtl has no effect on the behavior of Non-secure register accesses. Any Non-secure write to GICC\_EOIR must correspond to the most recent Non-secure read of GICC\_IAR. However, when GICC\_CTLR.AckCtl is set to 0, Non-secure software must not perform a GICC\_IAR write for an interrupt if Secure software has already performed the GICC\_IAR write for that interrupt. If it does, the effect of the write is UNPREDICTABLE.

## Behavior of writes to GICC\_EOIR, GICv2

See *Priority drop and interrupt deactivation* on page 3-38 for general information about the effect of writes to GICC\_EOIR, and about the possible separation of the GIC priority drop and interrupt deactivate operations in a GICv2 implementation.

In a GICv2 implementation, when GICC\_CTLR.AckCtl is set to 0:

- GICC\_EOIR is used for processing Group 0 interrupts
- GICC\_AEOIR is used for processing Group 1 interrupts.

In a GICv2 implementation that includes the GIC Security Extensions:

- GICC\_CTLR.EOImodeS controls the behavior of Secure accesses to GICC\_EOIR and GICC\_AEOIR
- GICC\_CTLR.EOImodeNS controls the behavior of Non-secure accesses to GICC\_EOIR
- when GICC\_CTLR.AckCtl is set to 0:
  - a Non-secure write to GICC\_EOIR must correspond to the most recent Non-secure read of GICC\_IAR
  - a Secure write to the GICC\_AEOIR must correspond to the most recent Secure read of the GICC\_AIAR.

Table 4-38 shows how, for a GICv2 implementation, the security level of the GICC\_EOR access, and the value of the GICC\_CTLR.AckCtl bit, determine the Priority drop effect of a valid GICC\_EOIR write. It also shows how, in a system that uses the suggested implementation for the Active Priorities Registers, the priority drop clears a bit in either the Secure Active Priorities Register, GICC\_APRn, or Non-secure Active Priorities Register, GICC\_NSAPRn. If the GIC does not implement the GIC Security Extensions, only the entries for the Secure GICC\_EOIR accesses apply.

**Table 4-38 Priority drop effect of GICC\_EOIR writes**

GICC_EOIR access	GICC_CTLR.AckCtl	Highest priority active interrupt	Effect
Non-secure	-	Group 1	Performs priority drop for Group 1 interrupts. In the Active Priorities registers, clears the highest active Group 1 priority level
Non-secure	-	Group 0	Architecturally UNPREDICTABLE. This access must not affect the set of active Group 0 priority levels.  <div style="text-align: center;"><b>Note</b></div> The write might have an IMPLEMENTATION DEFINED effect. For example, an implementation might clear the highest active Group 1 priority level in the Active Priorities registers.
Secure	0	-	Performs priority drop for Group 0 interrupts. In the Active Priorities registers, clears the highest active Group 0 priority level.
Secure	1	-	Performs priority drop. The running priority, and priority drop, take no account of interrupt grouping. In the Active Priorities registers, clears the highest active priority level. This can be either a Group 0 or a Group 1 active priority depending on which is the higher. If the highest active priority levels for both Group 0 and Group 1 are the same, the effect is UNDEFINED.

Table 4-39 shows how, for a GICv2 implementation, the security level of the GICC\_EOR access, and the values of the GICC\_CTLR control bits, determine whether a valid GICC\_EOIR write deactivates the identified interrupt. If the GIC does not implement the GIC Security Extensions, the entries for the Non-secure GICC\_EOIR accesses do not apply.

**Table 4-39 Deactivate interrupt effect of GICC\_EOIR writes**

GICC_EOIR access	GICC_CTLR.AckCtl	EOImode bit <sup>a</sup>	Identified interrupt	Effect
Non-secure	-	0	Group 1	Interrupt deactivated
Non-secure	-	0	Group 0	Access ignored
Secure	-	0	Group 0	Interrupt deactivated
Secure	1	0	Group 1	Interrupt deactivated
Secure	0	0	Group 1	UNPREDICTABLE
-	-	1	-	Interrupt remains active

a. For a GICv2 implementation that does not include the Security Extensions.

For a GICv2 implementation that includes the Security Extensions, GICC\_CTLR.EOImode is called:

- GICC\_CTLR.EOImodeS for Secure accesses to GICC\_EOIR. This setting also applies to Secure accesses to GICC\_AEOIR
- GICC\_CTLR.EOImodeNS for Non-secure accesses to GICC\_EOIR.

### 4.4.6 Running Priority Register, GICC\_RPR

The GICC\_RPR characteristics are:

- Purpose** Indicates the [Running priority](#) of the CPU interface.
- Usage constraints** If there is no active interrupt on the CPU interface, the value returned is the [Idle priority](#).

———— **Note** —————

Software cannot determine the number of implemented priority bits from a read of this register.

If the GIC implements the Security Extensions, the value returned by a Non-secure read of the Priority field is:

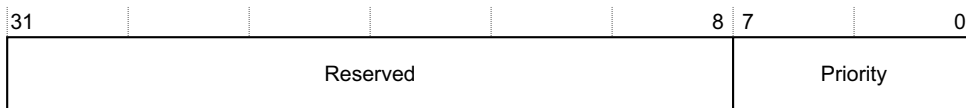
- 0x00 if the field value is less than 0x80
- the Non-secure view of the Priority value if the field value is 0x80 or more.

For more information see [Non-secure access to register fields for Group 0 interrupt priorities on page 4-81](#).

- Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.

- Attributes** See the register summary in [Table 4-2 on page 4-76](#).

[Figure 4-29](#) shows the GICC\_RPR bit assignments.



**Figure 4-29 GICC\_RPR bit assignments**

[Table 4-40](#) shows the GICC\_RPR bit assignments.

**Table 4-40 GICC\_RPR bit assignments**

Bit	Name	Description
[31:8]	-	Reserved.
[7:0]	Priority	The current running priority on the CPU interface.

The following pseudocode shows the effects of the GIC Security Extensions on accesses to this register:

```
// ReadGICC_RPR()
// =====
//
// Value of GICC_RPR read by a processor access
//
bits(8) ReadGICC_RPR()

    read_value = GICC_RPR<7:0>;
    if NS_access then // A non-secure GIC access,
        if read_value<7> == '0' then // therefore, adjust value.
            read_value = '00000000'; // A secure priority value, RAZ
        else
            read_value = LSL((read_value AND P_MASK), 1);
    return(read_value);
```

#### 4.4.7 Highest Priority Pending Interrupt Register, GICC\_HPIR

The GICC\_HPIR characteristics are:

<b>Purpose</b>	Indicates the Interrupt ID, and processor ID if appropriate, of the highest priority pending interrupt on the CPU interface.
<b>Usage constraints</b>	<p>Never returns the Interrupt ID of an interrupt that is active and pending. Returns a processor ID only for an SGI in a multiprocessor implementation.</p> <p>If the GIC supports interrupt grouping, the value returned by a read of GICC_HPIR can depend on:</p> <ul style="list-style-type: none"> <li>• the value of GICC_CTLR.AckCtl</li> <li>• if the GIC implements the Security Extensions, whether the register access is Secure or Non-secure:</li> </ul> <p>See <a href="#">Effect of interrupt grouping and the Security Extensions on reads of the GICC_HPIR</a>.</p>
<b>Configurations</b>	This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.
<b>Attributes</b>	See the register summary in <a href="#">Table 4-2 on page 4-76</a> .

[Figure 4-30](#) shows the GICC\_HPIR bit assignments.



**Figure 4-30 GICC\_HPIR bit assignments**

[Table 4-41](#) shows the GICC\_HPIR bit assignments.

**Table 4-41 GICC\_HPIR bit assignments**

Bit	Name	Description
[31:13]	-	Reserved.
[12:10]	CPUID	On a multiprocessor implementation, if the PENDINTID field returns the ID of an SGI, this field contains the CPUID value for that interrupt. This identifies the processor that generated the interrupt. In all other cases this field is RAZ.
[9:0]	PENDINTID	The interrupt ID of the highest priority pending interrupt. See <a href="#">Table 4-42 on page 4-144</a> for more information about the result of Non-secure reads of the GICC_HPIR when the GIC implements the Security Extensions.

#### Effect of interrupt grouping and the Security Extensions on reads of the GICC\_HPIR

If a CPU interface supports interrupt grouping, whether a read of the GICC\_HPIR returns a valid interrupt ID depends on:

- whether the highest priority pending interrupt is configured as a Group 0 or a Group 1 interrupt
- the value of the [GICC\\_CTLR.AckCtl](#) bit.
- if the GIC implements the Security Extensions, whether the GICC\_HPIR read access is Secure or Non-secure.

Reads of the GICC\_HPPIR that do not return a valid interrupt ID returns a spurious interrupt ID, ID 1022 or 1023, see *Special interrupt numbers when a GIC supports interrupt grouping* on page 3-50. Table 4-42 shows all possible GICC\_HPPIR reads for a CPU interface that implements the Security Extensions on a GIC that supports interrupt grouping. If the CPU interface does not implement the Security Extensions, the entries that apply to Secure GICC\_HPPIR reads describe the behavior.

**Table 4-42 Effect of the Security Extensions on GICC\_HPPIR reads**

Current state	GICC_HPPIR read	GICC_CTLR.AckCtl	Returned interrupt ID
Highest priority pending interrupt is Group 1	Non-secure	x	ID of Group 1 interrupt
	Secure	0	Spurious interrupt ID 1022
		1	ID of Group 1 interrupt
Highest priority pending interrupt is Group 0	Non-secure	x	Spurious interrupt ID 1023
	Secure	x	ID of Group 0 interrupt
No pending interrupts	x		Spurious interrupt ID 1023

The following pseudocode shows the effects of the GIC Security Extensions on accesses to this register:

```
// ReadGICC_HPPIR()
// =====
//
// Value of GICC_HPPIR read by a CPU access

bits(32) ReadGICC_HPPIR(integer cpu_id)
  // cpu_id identifies the accessed CPU interface
  // GICC_CTLR[cpu_id] is the GICC_CTLR register for that interface

  pendID = HighestPriorityPendingInterrupt(cpu_id);

  if ( IsGrp0Int(pendID) && GICC_CTLR.EnableGrp0 == '0' ) ||
    (!IsGrp0Int(pendID) && GICC_CTLR.EnableGrp1 == '0')
  then
    pendID = 1023; // If required group is not enabled, then no interrupt

  if GICC_MASK_HPPIR // GICC_MASK_HPPIR indicates the IMPLEMENTATION DEFINED
                    // choice whether GICC_CTLR.EnableGrp{0,1} being zero
  then // returns a spurious interrupt
    if ( IsGrp0Int(pendID) && GICC_CTLR[cpu_id].EnableGrp0 == '0' ) ||
      (!IsGrp0Int(pendID) && GICC_CTLR[cpu_id].EnableGrp1 == '0')
    then
      pendID = 1023; // If required group is not enabled, then no interrupt

  if pendID != 1023 then // An enabled interrupt is pending
    if IsGrp0Int(pendID) then // Highest priority is Group 0
      if NS_access then
        pendID = 1023;
      else // Highest priority is Group 1
        if !NS_access && (GICC_CTLR[cpu_id].AckCtl == '0') then
          pendID = 1022;

  cpuID = 0; // Must be zero for non-SGI interrupts

  if pendID < 16 then // 0 .. 15 are Software Generated Interrupts
    sgiID = SGI_CpuID(pendID); // value is IMPLEMENTATION DEFINED

  rval = 0;
  rval<12:10> = sgiID;
  rval<9:0> = pendID;

  return(rval);
```



#### 4.4.8 Aliased Binary Point Register, GICC\_ABPR

The GICC\_ABPR characteristics are:

- Purpose** A Binary Point Register for handling Group 1 interrupts.  
The reset value of this register is defined as (minimum [GICC\\_BPR.Binary point](#) + 1), resulting in a permitted range of 0x1-0x4.
- Usage constraints** If the GIC implements the Security Extensions, accessible by Secure accesses only.
- Configurations** This register is present only in GICv2, and in GICv1 implementations that include the Security Extensions,  
In a GIC implementation that includes the Security Extensions, GICC\_ABPR is an alias of the Non-secure [GICC\\_BPR](#), and when [GICC\\_CTLR.CBPR](#) is set to 0, a Secure access to this register is equivalent to a Non-secure access to [GICC\\_IAR](#).

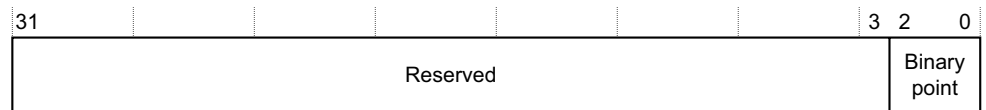
———— **Note** ————

- [GICC\\_ABPR](#) is redundant when [GICC\\_CTLR.CBPR](#) is set to 1. In a GIC implementation that includes the Security Extensions, when [GICC\\_CTLR.CBPR](#) is set to 1, the behavior of Secure accesses to [GICC\\_ABPR](#) is not identical to the behavior of Non-secure accesses to [GICC\\_BPR](#)
- Accesses to the [GICC\\_ABPR](#) are unaffected by the value of the [GICC\\_CTLR.CBPR](#) bit.

If the GIC implementation includes the Security Extensions, [GICC\\_ABPR](#) is a Secure register. If the GIC does not implement the [GICC\\_ABPR](#), the address is RAZ/WI.

**Attributes** See the register summary in [Table 4-2 on page 4-76](#).

[Figure 4-31](#) shows the [GICC\\_ABPR](#) bit assignments.



**Figure 4-31 GICC\_ABPR bit assignments**

[Table 4-43](#) shows the [GICC\\_ABPR](#) bit assignments.

**Table 4-43 GICC\_ABPR bit assignments**

Bits	Name	Function
[31:3]	-	Reserved.
[2:0]	Binary point	<p>The value of this field controls how the 8-bit interrupt priority field is split into a group priority field, used to determine interrupt preemption, and a subpriority field. For how this field determines the interrupt priority bits assigned to the group priority field see:</p> <ul style="list-style-type: none"> <li>• <a href="#">Table 3-7 on page 3-57</a>, for the processing of Group 1 interrupts on a GIC that supports interrupt grouping, when the <a href="#">GICC_CTLR.CBPR</a> bit is set to 1</li> <li>• <a href="#">Table 3-2 on page 3-46</a>, for all other cases.</li> </ul> <p>See <a href="#">Priority grouping on page 3-45</a> for more information.</p>

———— **Note** ————

In a GIC implementation that includes the Security Extensions, aliasing the Non-secure [GICC\\_BPR](#) as the [GICC\\_ABPR](#) means that, in a multiprocessor system, a processor that can make only Secure accesses to the GIC can access the [GICC\\_ABPR](#), to configure the preemption setting for Group 1 interrupts.

### 4.4.9 Aliased Interrupt Acknowledge Register, GICC\_AIAR

The GICC\_AIAR characteristics are:

- Purpose** An Interrupt Acknowledge register for handling Group 1 interrupts.  
 The processor reads this register to obtain the interrupt ID of the signaled Group 1 interrupt. This read acts as an acknowledge for the interrupt.
- Usage constraints** If the GIC implements the Security Extensions, accessible by Secure accesses only.
- Configurations** This register is present only in GICv2. If the GIC implements the Security Extensions, GICC\_AIAR is an alias of the Non-secure view of [GICC\\_IAR](#), and a Secure access to this register is identical to a Non-secure access to [GICC\\_IAR](#).  
 If the GIC implements the Security Extensions this is a Secure register.
- Attributes** See the register summary in [Table 4-2 on page 4-76](#).

[Figure 4-32](#) shows the GICC\_AIAR bit assignments.



**Figure 4-32 GICC\_AIAR bit assignments**

[Table 4-44](#) shows the GICC\_AIAR bit assignments.

**Table 4-44 GICC\_AIAR bit assignments**

Bit	Name	Function
[31:13]	-	Reserved, SBZ.
[12:10]	CPUID	For SGIs in a multiprocessor implementation, this field identifies the processor that requested the interrupt. It returns the number of the CPU interface that made the request, for example a value of 3 means the request was generated by a write to the <a href="#">GICD_SGIR</a> on CPU interface 3. For all other interrupts this field is RAZ.
[9:0]	Interrupt ID	The interrupt ID.

#### 4.4.10 Aliased End of Interrupt Register, GICC\_AEOIR

The GICC\_AEOIR characteristics are:

**Purpose** An end of interrupt register for handling Group 1 interrupts.  
When `GICC_CTLR.AckCtl` is set to 0, a write to this register performs priority drop for the identified Group 1 interrupt, and if the appropriate `GICC_CTLR.EOImode` bit is set to 0, also deactivates the interrupt. For more information see *Priority drop and interrupt deactivation* on page 3-38.

———— **Note** —————

In a GIC that implements interrupt grouping, when `GICC_CTLR.AckCtl` is set to 0 the Secure `GICC_EOIR` cannot be used for this purpose because a write to that register affects `GICC_APRn`, not `GICC_NSAPRn`.

**Usage constraints** A write to this register must correspond to the most recently acknowledged Group 1 interrupt.  
If the GIC implementation includes the Security Extensions, this is a Secure register.

**Configurations** This register is present only in GICv2.  
If the GIC implements the Security Extensions, `GICC_AEOIR` is effectively an alias of the Non-secure `GICC_EOIR`. A Secure access to this register is similar to a Non-secure access to `GICC_EOIR`, except that the `GICC_CTLR.EOImodeS` bit is used. See *End of Interrupt Register, GICC\_EOIR* on page 4-138 for more information.  
If the GIC implements the Security Extensions, this is a Secure register.

**Attributes** See the register summary in [Table 4-2 on page 4-76](#).

[Figure 4-33](#) shows the GICC\_AEOIR bit assignments



**Figure 4-33 GICC\_AEOIR bit assignments**

[Table 4-45](#) shows the GICC\_AEOIR bit assignments.

**Table 4-45 GICC\_AEOIR bit assignments**

Bit	Name	Function
[31:13]	-	Reserved, SBZ.
[12:10]	CPUID	On a multiprocessor implementation, when processing an SGI, this field must contain the CPUID value from the corresponding <code>GICC_AIAR</code> , or Non-secure <code>GICC_IAR</code> , access. In all other cases this field SBZ.
[9:0]	Interrupt ID	The Interrupt ID value from the corresponding <code>GICC_AIAR</code> , or Non-secure <code>GICC_IAR</code> , access.

The effect is UNPREDICTABLE if a value other than the last value read from the `GICC_AIAR` is written to `GICC_AEOIR`.

### 4.4.11 Aliased Highest Priority Pending Interrupt Register, GICC\_AHPPIR

The GICC\_AHPPIR characteristics are:

- Purpose** Provides a Highest Priority Pending Interrupt register for the handling of Group 1 interrupts. If the highest priority pending interrupt on the CPU interface is a Group 1 interrupt, returns the interrupt ID of that interrupt. Otherwise, returns a spurious interrupt ID of 1023.
- Usage constraints** Never returns the Interrupt ID of an interrupt that is active and pending. If the GIC implements the Security Extensions, accessible by Secure accesses only.
- Configurations** This register is present only in GICv2. If the GIC implements the Security Extensions, GICC\_AHPPIR is an alias of the Non-secure GICC\_HPPIR, and a Secure access to this register is equivalent to a Non-secure access to GICC\_HPPIR. If the GIC implements the Security Extensions this is a Secure register.
- Attributes** See the register summary in [Table 4-2 on page 4-76](#).

[Figure 4-34](#) shows the GICC\_AHPPIR bit assignments.



**Figure 4-34 GICC\_AHPPIR bit assignments**

[Table 4-46](#) shows the GICC\_AHPPIR bit assignments.

**Table 4-46 GICC\_AHPPIR bit assignments**

Bit	Name	Description
[31:13]	-	Reserved.
[12:10]	CUID	On a multiprocessor implementation, if the PENDINTID field returns the ID of an SGI, this field contains the CUID value for that interrupt. This identifies the processor that generated the interrupt. In all other cases this field is RAZ.
[9:0]	PENDINTID	The interrupt ID of the highest priority pending interrupt, if that interrupt is a Group 1 interrupt. Otherwise, the spurious interrupt ID, 1023.

#### 4.4.12 Active Priorities Registers, GICC\_APRn

The GICC\_APR characteristics are:

<b>Purpose</b>	These are IMPLEMENTATION DEFINED registers that provide support for preserving and restoring the active priority in power-management implementations.
<b>Usage constraints</b>	Although the format of these registers is IMPLEMENTATION DEFINED: <ul style="list-style-type: none"> <li>because GICv2 guarantees the ability to save and restore all GIC state, the GICC_APRn registers must be present in all GIC implementations</li> <li>in an implementation that includes the GIC Security Extensions, Non-secure accesses must not affect Secure operation, and the architecture requires that these registers are banked, to provide Secure and Non-secure copies of the registers.</li> </ul>
<b>Configurations</b>	These registers are present only in GICv2. The register locations are reserved in GICv1. The number of Active Priorities registers implemented depends on the number of <a href="#">Preemption levels</a> supported, see <a href="#">Table 4-47 on page 4-150</a> . If the GIC does not implement the Security Extensions, these registers hold the active priorities for the Group 0 interrupts.

———— **Note** ————

The GICC\_NSAPRn registers always hold the active priorities for the Group 1 interrupts.

If the GIC implements the Security Extensions, these registers are banked to provide Secure and Non-secure copies, see [Register banking on page 4-77](#). This ensures that:

- Non-secure accesses do not affect Secure operation
- the Non-secure copies of these registers provide a Non-secure view of the priorities of the Group 1 interrupts, see [Software views of interrupt priority in a GIC that includes the Security Extensions on page 3-53](#).
- the Secure copies of these registers track active priorities for Group 0 interrupts.

———— **Note** ————

The Secure copies of the GICC\_NSAPRn registers track active priorities for Group 1 interrupts.

**Attributes** See the register summary in [Table 4-2 on page 4-76](#).

———— **Note** ————

These registers are IMPLEMENTATION DEFINED, but ARM strongly recommends that, when these registers are implemented, they follow the guidelines in this section.

If the GIC implementation includes the Security Extensions, some of the Active Priorities Register space is visible to Non-secure accesses in a manner consistent with the Group 1 interrupt priority level remapping, see [Software views of interrupt priority in a GIC that includes the Security Extensions on page 3-53](#) and [The effect of interrupt grouping on priority grouping on page 3-57](#). This means only the lower half of the Preemption level space is visible, but remapped so it appears in the upper half of the preemption level space, as [Table 4-47 on page 4-150](#) shows.

See [Priority grouping on page 3-45](#) for more information about priority grouping and preemption levels.

Predictable system restoration is guaranteed if the GICC\_APRn registers are saved before powering down and restored after powering up. However, if a different value is written when restoring the register, or if any other value is written to the register during operation, behavior is UNPREDICTABLE.

To ensure stability:

- ARM recommends that software uses these registers only for the purpose of saving and restoring state
- both the GICC\_APRn and the GICC\_NSAPRn must include a bit for each preemption level that the system permits.

In an implementation that includes the GIC Security Extensions:

- Non-Secure register accesses must only access bits corresponding to preemption levels in the Non-Secure priority space, but return the Non-secure view of those priorities, see *Software views of interrupt priority in a GIC that includes the Security Extensions* on page 3-53
- the GICC\_NSAPRn registers must provide the Distributor view of the active priorities of the Group 1 interrupts.

Table 4-47 shows the GICC\_APR implementation.

**Table 4-47 Active Priorities register implementation**

Minimum value of Secure GICC_BPR	Minimum value of Non-secure GICC_BPR	Maximum number of group priority bits	Maximum number of preemption levels	GICC_APRn implementation	View of Active Priorities Registers for Non-secure accesses <sup>a</sup>
3	4	4	16	GICC_APR0[15:0]	GICC_NSAPR0[15:8] appears as GICC_APR0[7:0]
2	3	5	32	GICC_APR0[31:0]	GICC_NSAPR0[31:16] appears as GICC_APR0[15:0]
1	2	6	64	GICC_APR0-GICC_APR1	GICC_NSAPR1 appears as GICC_APR0
0	1	7	128	GICC_APR0-GICC_APR3	GICC_NSAPR2-GICC_NSAPR3 appear as GICC_APR0-GICC_APR1

a. In a GIC implementation that includes the GIC Security Extensions.

### 4.4.13 Non-secure Active Priorities Registers, GICC\_NSAPRn

The GICC\_NSAPR characteristics are:

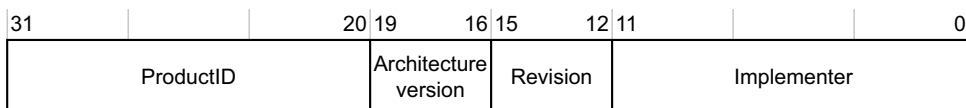
<b>Purpose</b>	<p>These are IMPLEMENTATION DEFINED registers that provide support for preserving and restoring the active priority in power-management implementation. These are separate registers for Group 1 interrupts.</p> <p>Software on the connected processor can save or restore the complete active priorities state by:</p> <ul style="list-style-type: none"><li>• using the GICC_APRn registers to save or restore the state for the Group 0 interrupts</li><li>• using the GICC_NSAPRn registers to save or restore the state for the Group 1 interrupts.</li></ul> <p>In an implementation that includes the Security Extensions:</p> <ul style="list-style-type: none"><li>• these registers ensure that Non-secure accesses cannot interfere with Secure operation</li><li>• Secure software on the connected processor can save or restore the complete active priorities state using the GICC_APRn and the GICC_NSAPRn registers.</li></ul>
<b>Usage constraints</b>	<p>Although the format of these registers is IMPLEMENTATION DEFINED:</p> <ul style="list-style-type: none"><li>• because GICv2 guarantees the ability to save and restore all GIC state, the GICC_NSAPRn registers must be present in all GIC implementations</li><li>• in an implementation that includes the Security Extensions, these registers are accessible only by Secure accesses.</li></ul>
<b>Configurations</b>	<p>These registers are present only in GICv2 implementations that include the GIC Security Extensions. The register locations are reserved in GICv1.</p> <p>These are Secure registers.</p> <p>The number of Active Priorities registers implemented depends on the number of levels of preemption level supported, see <a href="#">Table 4-47 on page 4-150</a>.</p>
<b>Attributes</b>	<p>See the register summary in <a href="#">Table 4-2 on page 4-76</a>.</p> <p>To support situations where Secure software elevates Group 1 interrupt priorities into Group 0 priority space, these registers must be large enough to contain the entire supported priority space. Therefore, the implemented GICC_NSAPRn register set is identical in format to the GICC_APRn register set, but contains the active priorities of Group 1 interrupts rather than Group 0 interrupts. If the GIC implementation includes the Security Extensions, the lower priority half of these registers can be accessed by Non-secure software accessing the Non-secure copies of the GICC_APRn registers, but these accesses return a Non-secure view of the interrupt priorities, as <a href="#">Table 4-47 on page 4-150</a> shows.</p> <p>See <a href="#">Active Priorities Registers, GICC_APRn on page 4-149</a> or more information about the implementation of these registers.</p>

#### 4.4.14 CPU Interface Identification Register, GICC\_IIDR

The GICC\_IIDR characteristics are:

- Purpose** Provides information about the implementer and revision of the CPU interface.
- Usage constraints** No usage constraints.
- Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.
- Attributes** See the register summary in [Table 4-2 on page 4-76](#).

[Figure 4-35](#) shows the GICC\_IIDR bit assignments.



**Figure 4-35 GICC\_IIDR bit assignments**

[Table 4-48](#) shows the GICC\_IIDR bit assignments.

**Table 4-48 GICC\_IIDR bit assignments**

Bit	Name	Description
[31:20]	ProductID	An IMPLEMENTATION DEFINED product identifier.
[19:16]	Architecture version	The value of this field depends on the GIC architecture version, as follows: <ul style="list-style-type: none"> <li>• 0x1 for GICv1</li> <li>• 0x2 for GICv2.</li> </ul>
[15:12]	Revision	An IMPLEMENTATION DEFINED revision number for the CPU interface.
[11:0]	Implementer	Contains the JEP106 code of the company that implemented the GIC CPU interface: <sup>a</sup> <ul style="list-style-type: none"> <li><b>Bits [11:8]</b> The JEP106 continuation code of the implementer.</li> <li><b>Bit [7]</b> Always 0.</li> <li><b>Bits [6:0]</b> The JEP106 identity code of the implementer.</li> </ul>

a. For an ARM implementation, the value of this field is 0x43B.



#### 4.4.15 Deactivate Interrupt Register, GICC\_DIR

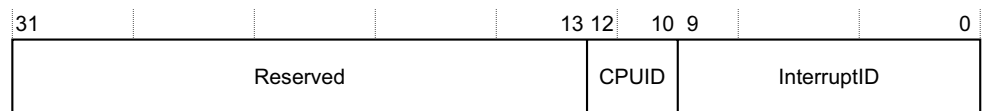
The GICC\_DIR characteristics are:

- Purpose** When interrupt priority drop is separated from interrupt deactivation, as described in [Priority drop and interrupt deactivation on page 3-38](#), a write to this register deactivates the specified interrupt.
- Usage constraints** Writes to this register only have an effect when:
- for a GIC implementation that does not include the Security Extensions, GICC\_CTLR.EOImode is set to 1
  - for a GIC implementation that includes the Security Extensions:
    - for Secure accesses to the register, GICC\_CTLR.EOImodeS is set to 1
    - for Non-secure accesses to the register, GICC\_CTLR.EOImodeNS is set to 1.
- If the relevant EOImode bit is 0 then the effect of this register access is UNPREDICTABLE.
- If the interrupt identified in the GICC\_DIR is not active, and is not a spurious interrupt, the effect of the register write is UNPREDICTABLE. This means any GICC\_DIR write must identify an interrupt for which there has been a valid GICC\_EOIR or GICC\_AEOIR write.
- Unlike GICC\_EOIR and GICC\_AEOIR writes, there is no ordering requirement for GICC\_DIR writes, provided they meet the other requirements given in this section.
- ARM recommends that the value written to GICC\_DIR is the 32-bit value returned by the corresponding GICC\_IAR or GICC\_AIAR read.
- Configurations** This register is present only in GICv2. The register location is reserved in GICv1. If the GIC implements the Security Extensions this register is Common.
- Attributes** See the register summary in [Table 4-2 on page 4-76](#).

———— **Note** —————

There is no requirement to deactivate interrupts in any particular order.

[Figure 4-36](#) shows the GICC\_DIR bit assignments.



**Figure 4-36 GICC\_DIR bit assignments**

[Table 4-49](#) shows the GICC\_DIR bit assignments.

**Table 4-49 GICC\_DIR bit assignments**

Bit	Name	Description
[31:13]	-	Reserved, SBZ
[12:10]	CPUID	For an SGI in a multiprocessor implementation, this field identifies the processor that requested the interrupt. For all other interrupts this field is RAZ.
[9:0]	Interrupt ID	The interrupt ID

## Behavior of writes to the GICC\_DIR

Regardless of the state of the `GICC_CTLR.AckCtl` bit:

- if the implementation does not include the Security Extensions, a valid write to `GICC_DIR` deactivates the specified interrupt, regardless of whether that interrupt is in Group 0 or Group 1
- if the implementation includes the Security Extensions, a valid:
  - Secure write to `GICC_DIR` deactivates the specified interrupt, regardless of whether that interrupt is in Group 0 or Group 1
  - Non-secure write to `GICC_DIR` deactivates the specified interrupt only if that interrupt is in Group 1. A valid write is one that specifies an interrupt that is active, and for which there has been a successful write to `GICC_EOIR` or `GICC_AEOIR`.

Table 4-50 shows the behavior of valid writes to `GICC_DIR`. In an implementation that does not include the Security Extensions, valid writes have the behavior shown for Secure `GICC_DIR` writes.

**Table 4-50 Behavior of GICC\_DIR writes**

<code>GICC_CTLR.AckCtl</code>	GICC_DIR write	Interrupt group	Effect
x	Non-secure	Group 1	Interrupt is deactivated.
x	Non-secure	Group 0	Write is ignored.
x	Secure	x	Interrupt is deactivated.

## 4.5 Preserving and restoring GIC state

Some situations require the system to save and restore the state of the GIC, or particular interrupts. For example:

- during system shutdowns
- when migrating the state of one CPU to another in a multiprocessor implementation
- in a system that supports processor virtualization, when changing between virtual machines.

To support this functionality, GICv2 provides:

- The ability to save and restore the active state held in [GICD\\_ISACTIVERn](#) and [GICD\\_ICACTIVERn](#).

———— **Note** —————

Implementations must not contain additional state, such as active bits per source processor for SGIs, because such additional state is not restored during these operations. For example, operations that preserve and restore GIC state by saving the Set-active bits from [GICD\\_ISACTIVERn](#) must be capable of saving and restoring all active state information.

- The ability to set and clear pending SGIs, provided by [GICD\\_CPENDSGIRn](#) and [GICD\\_SPENDSGIRn](#).
- Active Priorities registers, [GICC\\_APRn](#) and [GICC\\_NSAPRn](#), that provide the ability to access directly the associated GIC architecture state, and to save and restore that state.



# Chapter 5

## GIC Support for Virtualization

This chapter describes the GIC Virtualization Extensions introduced in GICv2, and using these to implement a GIC in a system with at least one processor that implements the ARM Virtualization Extensions. It contains the following sections:

- *About implementing a GIC in a system with processor virtualization on page 5-158*
- *Managing the GIC virtual CPU interface on page 5-160*
- *GIC virtual interface control registers on page 5-167*
- *The virtual CPU interface on page 5-178*
- *GIC virtual CPU interface registers on page 5-179.*

## 5.1 About implementing a GIC in a system with processor virtualization

With a GIC that includes the GIC Virtualization Extensions, a virtual machine running on a processor that includes the ARMv7-A Virtualization Extensions communicates with a virtual CPU interface on the GIC. The virtual machine receives virtual interrupts from this interface, and cannot distinguish these interrupts from physical interrupts.

A hypervisor handles all IRQs, translating those destined for a virtual machine into virtual interrupts, and, in conjunction with the GIC, manages the virtual interrupts and the associated physical interrupts. It also uses the GIC virtual interface control registers to manage the virtual CPU interface. As part of this control, the hypervisor updates the List registers, that are a subset of the GIC virtual interface control registers. In this way the hypervisor and GIC together provide a virtual distributor, that appears to a virtual machine as the physical GIC distributor.

The GIC virtual CPU interface signals virtual interrupts to the virtual machine, subject to the normal GIC handling and prioritization rules. [Figure 5-1 on page 5-159](#) shows an example of how the GIC handles interrupts in a implementation that supports processor virtualization.

---

### Note

- Any ARM processor implementation that includes the Virtualization Extensions must also include the Security Extensions. Such a processor is usually implemented with a GIC that implements both the GIC Security Extensions and GIC Virtualization Extensions. The examples in this chapter only describe such an implementation, for which:
  - Group 0 physical interrupts are Secure interrupts
  - Group 1 physical interrupts are Non-secure interrupts.
  - the hypervisor performs the initial processing of all physical IRQs, virtualizing them as required as virtual IRQs or virtual FIQs
  - Secure Monitor mode performs the initial processing of all physical FIQs.
 See [Security Extensions support on page 1-16](#) for more information.
- In descriptions of processor virtualization, a virtual machine runs a Guest OS, that runs applications. In many contexts, the terms *virtual machine* and *Guest OS* are synonymous.

---

In the ARM model for virtualizing Non-secure operation of a processor that implements the ARM Virtualization Extensions, Secure software on the processor must configure the system as described in [Using IRQs and FIQs to provide Non-secure and Secure interrupts on page 3-68](#), so that FIQs are used for Secure interrupts, and IRQs for Non-secure interrupts.

When the hypervisor receives an IRQ, it determines whether the interrupt is for itself, or for a virtual machine. If it is for a virtual machine it determines which virtual machine must handle the interrupt and generates a virtual interrupt, see [Managing the GIC virtual CPU interface on page 5-160](#).

The GIC Virtualization Extensions provide the following support for a virtual CPU interface:

- GIC virtual interface control registers. These are management registers, accessed by a hypervisor, or similar software. See [Managing the GIC virtual CPU interface on page 5-160](#) for more information.
- GIC virtual CPU interface registers. These registers provide the virtual CPU interface accessed by the current virtual machine on a connected processor. In general, they have the same format as the GIC physical CPU interface registers, but they operate on the interrupt view defined by the List registers.

A virtual machine communicates with the virtual CPU interface, but cannot detect that it is not communicating with a GIC physical CPU interface.

The virtual CPU interface and the GIC virtual interface control registers are both in the Non-secure memory map. A hypervisor uses the Non-secure stage 2 address translations to ensure that the virtual machine cannot access the GIC virtual interface control registers. To support this, the GIC architecture requires the GIC virtual CPU interface registers and the GIC virtual interface control registers to be in separate 4KB address regions. See the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* for more information about the ARM Virtualization Extensions and Non-secure address translation.

[GIC register names on page 4-74](#) describes the naming convention for the GIC registers, which means that:

- a name starting GICH\_ indicates a register described in [GIC virtual interface control registers on page 5-167](#)
- a name starting GICV\_ indicates a register described in [GIC virtual CPU interface registers on page 5-179](#).

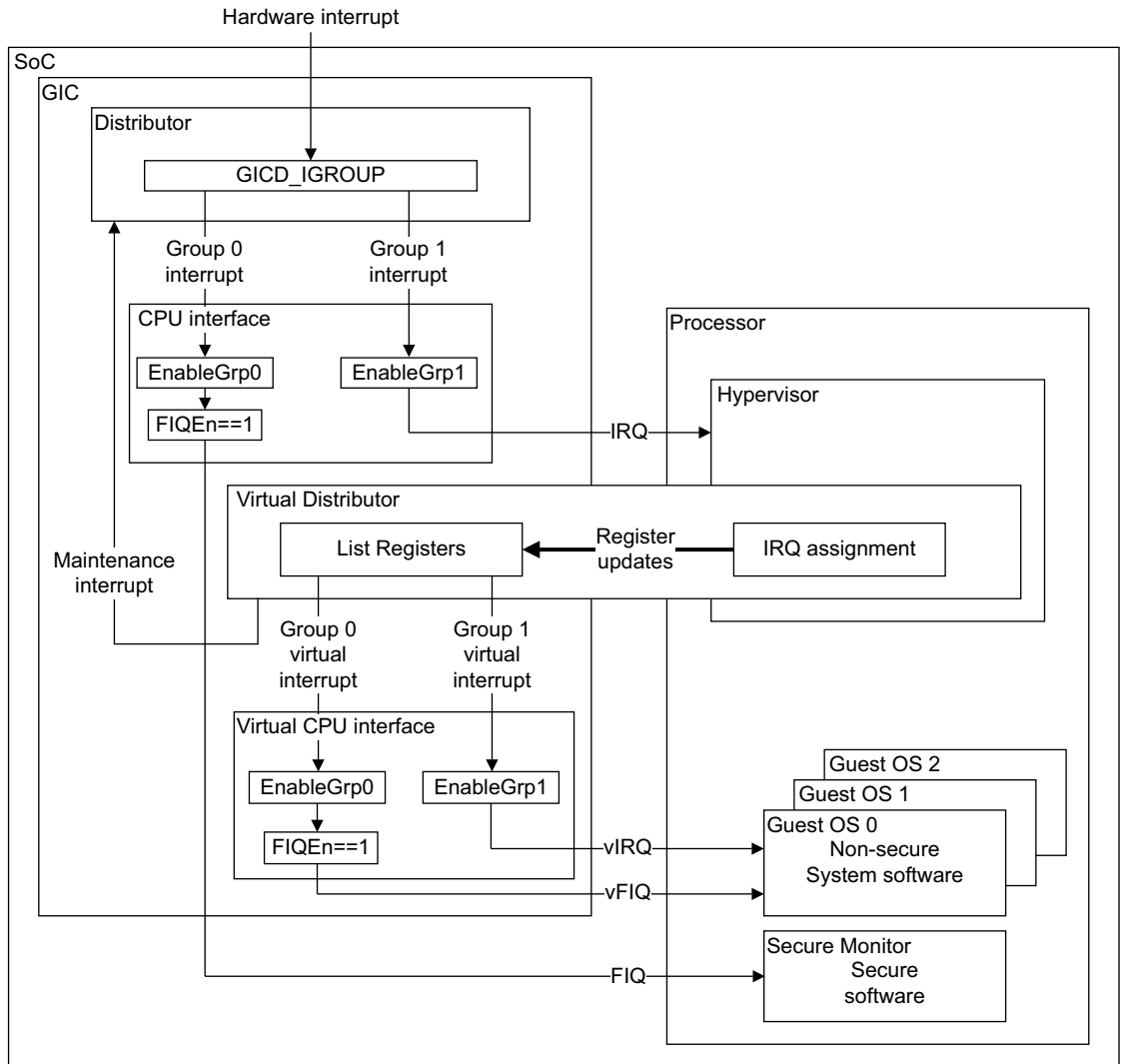
As with a physical CPU interface, the virtual CPU interface signals an interrupt if the highest priority pending interrupt has sufficient priority. However, the signaled interrupt is a virtual interrupt rather than a physical interrupt. The List registers indicate whether the interrupt is in Group 0 or Group 1 and therefore whether it is signaled as a virtual IRQ or a virtual FIQ.

The virtual machine:

- acknowledges a virtual interrupt by reading from the Interrupt Acknowledge Register.
- indicates when it has completed interrupt processing by writing to the End of Interrupt Register.

These writes update the List registers. See [Acknowledgement and completion of virtual interrupts on page 5-162](#) for more information.

[Figure 5-1](#) shows the model for implementing a GIC with an ARMv7-A processor that implements the processor Virtualization Extensions.



**Figure 5-1 Implementing the GIC with an ARM processor that supports virtualization**

## 5.2 Managing the GIC virtual CPU interface

This section describes the ARM model for virtualizing ARMv7-A processor that implements the processor Virtualization Extensions.

The hypervisor, or similar software, manages the GIC virtual interface control registers, consisting of:

**List registers** Used to define the active and pending virtual interrupts for the virtual CPU interface. The current virtual machine accesses these interrupts indirectly, using the virtual CPU interface.

### Management registers

Used to manage the virtual CPU interface, and to save and restore settings when switching between virtual machines.

On the processor, the hypervisor:

- configures IRQs to be taken in Hyp mode, so that it handles all IRQs itself
- uses the stage 2 Non-secure address translations to:
  - trap all Guest OS accesses to the GIC Distributor registers, so that it can determine the virtual distributor settings for each virtual machine
  - ensure that the virtual machines cannot access the GIC virtual interface control registers
  - remap the GIC CPU interface register address space to point to the GIC virtual CPU interface registers.
- configures the required maintenance interrupts from the virtual CPU interface, see [Maintenance interrupts on page 5-164](#).

The hypervisor controls, and switches between, the virtual machines. When it starts a virtual machine, it programs the List registers to define the interrupts that are visible to that virtual machine.

When it receives a physical IRQ, the hypervisor determines the required destination of the interrupt and then either:

- Processes the interrupt itself, for example if the IRQ is a maintenance interrupt from the virtual CPU interface. It then deactivates the physical interrupt.
- Generates a virtual interrupt. Depending on the interrupt priority and the targeted virtual machine, the hypervisor takes one of the following actions:
  - If the interrupt is for the current virtual machine, updates the List registers with details of the interrupt, redefining the interrupts that are visible to the current virtual machine. If there is no space in the List registers, it saves the context to memory so the details can be added at a later stage. See [List registers and virtual interrupt handling on page 5-161](#) for more information.
  - Records that the interrupt is for a different virtual machine by saving details of the interrupt as part of the hypervisor state associated with that virtual machine.
  - Switches to a different virtual machine that can handle the interrupt. In doing so it must save the interrupt state for the current virtual machine, using the information in the List registers, and reprogram the List registers, to indicate the interrupt state for the new virtual machine, including the state for the interrupt that has arrived.

The virtual machine accesses the GIC virtual CPU interface registers. These registers have the same general format as the physical CPU interface registers, and, in a typical implementation the virtual machine believes it is accessing a physical CPU interface. These accesses update the state and status bits in the List registers.

When the virtual machine handles a virtual interrupt, it writes to the virtual CPU interface to indicate when it has finished this processing. The virtual CPU interface signals this completion to the physical Distributor and the physical Distributor then deactivates the interrupt.

### ————— Note —————

The hypervisor is not part of the GIC architecture. It is supported by the ARMv7-A Architecture Virtualization Extensions. The hypervisor runs as Non-secure software in Hyp mode. See the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition* for more information.



The remainder of this section describes:

- [List registers and virtual interrupt handling](#)
- [Completion of virtualized physical interrupts](#)
- [Acknowledgement and completion of virtual interrupts on page 5-162](#)
- [GIC virtual interface control interface requirements on page 5-164](#)
- [Maintenance interrupts on page 5-164](#)
- [Software-generated interrupts on page 5-165](#)
- [GIC Virtualization Extensions register mapping on page 5-165.](#)

### 5.2.1 List registers and virtual interrupt handling

With a GIC implementation that includes the Virtualization Extensions, a hypervisor uses List registers to maintain the list of highest priority virtual interrupts. The total number of interrupts that are either pending, active, or active and pending, can exceed the number of List registers available. If this happens, the hypervisor can save one or more active interrupt entries to memory, and later restore them to the List registers, based on their priority. Therefore:

- The List registers might not include all active, or active and pending, interrupts. Virtual CPU interface accesses by the virtual machine update the List registers, and normally an EOI request from the virtual machine deactivates an interrupt in the list. However, the virtual machine can issue an EOI request for an interrupt before the hypervisor restores the associated active interrupt entry into a List register. In this case, the EOI request cannot update the List registers.

———— **Note** ————

Only hypervisor-generated interrupts can be active and pending.

- Although the List registers might include only active interrupts, with the hypervisor maintaining any pending interrupts in memory, a pending interrupt cannot be signaled to the virtual machine until the hypervisor adds it to the List registers. Therefore, to minimize interrupt latency and ensure efficient virtual machine operation, ARM strongly recommends that the List registers contain at least one pending interrupt, provided a List register is available for this interrupt.

To maintain the 1-N interrupt handling model, a hypervisor might have to migrate an interrupt from one virtual machine to another. An example of when this might be necessary is when enabling or disabling a virtual CPU interface.

The GIC Virtualization Extensions include the following features to support virtual interrupt handling:

- Priority drop functionality is separate from interrupt deactivation, see [Completion of virtualized physical interrupts](#).
- Maintenance interrupts signal key events, see [Maintenance interrupts on page 5-164](#).

### 5.2.2 Completion of virtualized physical interrupts

This section describes the procedure that ARM recommends for the completion of a physical interrupt that has been virtualized. Using these procedures reduces the amount of hypervisor intervention required, meaning the hypervisor can receive new physical interrupts as early as possible. This prevents physical interrupts from being prematurely rescheduled to the hypervisor.

When virtualizing physical interrupts, ARM recommends that, for each CPU interface that corresponds to a processor running virtual machines, the `GICC_CTLR.EOImodeNS` bit is set to 1. This means that hypervisor accesses to the `GICC_EOIR` register drops the running priority of the CPU interface but does not deactivate the interrupt. After writing to the EOI register, the running priority level on the CPU interface is lower, so that subsequent interrupts can be signaled to the processor.

ARM recommends that physical interrupt completion consists of the following separate steps:

1. EOI
2. interrupt deactivation.

These steps are explained in more detail as follows:

1. After receiving a physical interrupt, the hypervisor performs an EOI request for the physical interrupt by writing to the `GICC_EOIR` or `GICC_AEOIR` register. After EOI, although the virtual machine has not processed the virtual interrupt, the lower running priority of the CPU interface means that the hypervisor can still receive new physical interrupts.

This enables the hypervisor to set the priority of new interrupts as they arrive, providing more flexibility than the EOI procedure for non-virtualized physical interrupts described in [General handling of interrupts on page 3-37](#).

———— **Note** ————

The only interrupts that are not signaled to the hypervisor are the physical interrupts most recently subject to EOI. This is because the interrupts have not been deactivated. This prevents the interrupts from being re-signaled to the hypervisor before being processed by the virtual machine.

2. After the virtual machine completes processing the corresponding virtual interrupt, it writes to the `GICV_EOIR` or `GICV_AEOIR` to deactivate the interrupt. This deactivates both the virtual interrupt and the corresponding physical interrupt, provided that both of the following conditions are true:

- the `GICV_CTLR.EOImode` bit is set to 0
- the `GICH_LRn.HW` bit is set to 1.

Alternatively, if the `GICV_CTLR.EOImode` bit is set to 1, the virtual machine writes to the `GICV_DIR` register to deactivate the interrupt.

If the `GICH_LRn.HW` bit is set to 0, the hypervisor must deactivate the physical interrupt itself. ARM recommends one of the following methods for deactivating physical SGIs that are routed to a virtual machine:

- the hypervisor deactivates the SGI by writing to the `GICC_DIR` register after the virtual machine writes to `GICC_EOIR`
- the hypervisor uses an EOI maintenance interrupt to write to the `GICC_DIR` register after the virtual machine writes to `GICV_EOIR`, see [Maintenance interrupts on page 5-164](#) for more information.

### 5.2.3 Acknowledgement and completion of virtual interrupts

This section describes the relationship between use of the GIC virtual CPU interface registers and the GIC virtual interface control registers. See [The virtual CPU interface on page 5-178](#) for more information about the virtual CPU interface.

To ensure system correctness when handling virtual interrupts, one of the following conditions must be true:

- All Group 0 interrupts must have a higher priority than any Group 1 interrupt. That is, there is no overlap in the priorities allocated to Group 0 and Group 1 interrupts.
- The `GICV_CTLR.AckCtl` bit must be set to 0.

These conditions apply, also, to physical interrupts and the `GICC_CTLR.AckCtl` bit, see [The effect of interrupt grouping on interrupt acknowledgement on page 3-50](#).

———— **Note** ————

ARM deprecates the use of `GICC_CTLR.AckCtl` and `GICV_CTLR.AckCtl`, and strongly recommends using a software model where `GICC_CTLR.AckCtl` and `GICV_CTLR.AckCtl` are set to 0.

In GICv2, ARM recommends that separate registers are used to manage Group 0 and Group 1 interrupts:

- `GICV_IAR`, `GICV_EOIR`, and `GICV_HPPIR` for Group 0 interrupts
- `GICV_AIAR`, `GICV_AEOIR`, and `GICV_AHPPIR` for Group 1 interrupts.

The operation of these registers is:

### GICV\_IAR and GICV\_AIAR

The virtual machine reads [GICV\\_IAR](#) or [GICV\\_AIAR](#) to acknowledge an interrupt. A spurious interrupt ID is returned when:

- there is no interrupt to acknowledge
- a higher priority interrupt is ready to be acknowledged in the other group.

The normal GIC rule that interrupts must complete in the order that they are acknowledged on a CPU interface applies to both the physical and virtual CPU interfaces.

### GICV\_EOIR and GICV\_AEOIR

The EOI request must write the Interrupt ID and CPUID values read when the interrupt was acknowledged. A write to the appropriate register, [GICV\\_EOIR](#) or [GICV\\_AEOIR](#) clears the preemption bit associated with the highest priority active interrupt in the Active Priorities Register, [GICH\\_APR](#):

- When the highest priority active interrupt is a Group 0 interrupt, writing the appropriate value read from [GICV\\_IAR](#) to [GICV\\_EOIR](#):
  - clears the preemption bit in [GICH\\_APR](#)
  - if [GICV\\_CTLR.EOImode](#) is cleared to 0, removes the active state in the corresponding List register
  - if [GICV\\_CTLR.EOImode](#) is cleared to 0 and the [GICH\\_LRn.HW](#) bit is set to 1, deactivates the corresponding physical interrupt in the Distributor

When the highest priority active interrupt is a Group 0 interrupt, the effect of writing to [GICV\\_AEOIR](#) is UNPREDICTABLE.

- When the highest priority active interrupt is a Group 1 interrupt, writing the appropriate value read from [GICV\\_AIAR](#) to [GICV\\_AEOIR](#):
  - clears the preemption bit in [GICH\\_APR](#), and
  - if [GICV\\_CTLR.EOImode](#) is cleared to 0, removes the active state in the corresponding List register
  - if [GICV\\_CTLR.EOImode](#) is cleared to 0 and the [GICH\\_LRn.HW](#) bit == 1, deactivates the corresponding physical interrupt in the Distributor

When the highest priority active interrupt is a Group 1 interrupt, the effect of writing to [GICV\\_EOIR](#) is UNPREDICTABLE.

[Table 4-37 on page 4-139](#) shows how [GICV\\_AEOIR](#) is affected by [GICV\\_CTLR.AckCtl](#).

### GICV\_HPPIR and GICV\_AHPPIR

For the virtual CPU interface:

- a read of [GICV\\_HPPIR](#) returns the Group 0 pending interrupt with the highest priority
- a read of [GICV\\_AHPPIR](#) returns the Group 1 pending interrupt with the highest priority.

[Table 4-42 on page 4-144](#) shows how [GICV\\_HPPIR](#) is affected by [GICV\\_CTLR.AckCtl](#).

The hypervisor uses the [GICC\\_CTLR.EOImode](#) bit to separate priority drop in the physical CPU interface and interrupt deactivation in the Distributor. The hypervisor can use [GICC\\_DIR](#) to deactivate interrupts, to retire them from the Distributor. The [GICC\\_DIR](#) is used to deactivate hardware interrupts in certain cases, and usually the [GICC\\_DIR](#) operation is required for deactivating SGIs:

- in the SGI N-N handling model
- where a hypervisor-generated interrupt exists to support a virtual device.

A virtual machine can deactivate interrupts in the following ways:

#### **GICV\_CTLR.EOImode == 0**

The GIC deactivates hardware interrupts directly, that is, writing to `GICV_EOIR` drops the priority of an interrupt and deactivates it simultaneously. The `GICH_LRn.HW` bit indicates whether an interrupt is related to hardware or software, and therefore whether to forward the deactivate to the Distributor.

See *List Registers, GICH\_LRn* on page 5-176 for more information.

#### **GICV\_CTLR.EOImode == 1**

Writing to `GICV_EOIR` performs priority drop operation and writing to `GICV_DIR` performs the deactivate interrupt operation.

#### **Note**

- The limited context information available when a hypervisor handles a maintenance interrupt means that, if a hypervisor maintains more than one active interrupt in memory, instead of in the List registers, it must also trap virtual machine accesses to `GICV_DIR`, so that it can deactivate interrupts for the virtual machine.  
ARM recommends that, as far as possible, the hypervisor manages active interrupts for the current virtual machine using the List registers.
- The GIC architecture requires that writes to `GICV_EOIR` are ordered so that a write to `GICV_EOIR` always refers to the same interrupt as the most recent read of `GICV_IAR`. However, there is no requirement for writes to `GICV_DIR` to deactivate interrupts in any particular order.

### **5.2.4 GIC virtual interface control interface requirements**

The following cases are considered software programming errors and result in UNPREDICTABLE behavior:

- Having two or more copies of the same interrupt in the List registers.
- Having two or more interrupts with the same PhysicalID on one virtual CPU interface. This includes having interrupts with the same PhysicalID that correspond to a physical SPI.
- Having a hardware interrupt in active state or in pending state in the List registers if the Distributor does not have the corresponding physical interrupt in either the active state or the active and pending state.
- If `GICV_CTLR.EOImode` is set to 0, then either:
  - having an active interrupt in the List registers with a priority that is not set in the corresponding Active Priorities register
  - having two interrupts in the List registers in the active state with the same preemption priority.
- Writing an EOI request with the InterruptID of an interrupt that the List registers show as being in the pending state.

### **5.2.5 Maintenance interrupts**

Maintenance interrupts can signal key events in the operation of a GIC that implements the Virtualization Extensions. Typically, these events are processed by the hypervisor.

#### **Note**

- Maintenance interrupts are generated only when the global interrupt enable bit, `GICH_HCR.En`, is set to 1.
- Maintenance interrupt routing is outside the scope of this specification.

Maintenance interrupts are level-sensitive interrupts. Configuration bits in the `GICH_HCR` can be set to 1 to enable maintenance interrupt generation when:

- Group 0 virtual interrupts are enabled.

- Group 1 virtual interrupts are enabled.
- Group 0 virtual interrupts are disabled.
- Group 1 virtual interrupts are disabled.
- There are no pending interrupts in the List registers.
- At least one EOI request occurs with no valid List register entry for the corresponding interrupt.
- There are no valid entries, or only one valid entry, in the List registers. This is an underflow condition.
- At least one List register entry has received an EOI request.

See [Maintenance Interrupt Status Register, GICH\\_MISR](#) on page 5-172 for more information about the control and status reporting of maintenance interrupts.

## 5.2.6 Software-generated interrupts

The following types of software-generated interrupt exist:

### Hypervisor-generated interrupts

A hypervisor can generate virtual interrupts that do not have a corresponding physical interrupt, by creating an entry in the List registers with the GICH\_LRn.HW bit cleared to 0. The hypervisor can control how the interrupt appears to a virtual machine reading the GICV\_IAR or GICV\_AIAR register to acknowledge the interrupt, by presenting the interrupt as:

- an SGI, with a CPUID value provided in addition to the interrupt ID
- a PPI or SPI, with the CPUID value set to 0.

The hypervisor can virtualize the CPUID value, but it must be consistent with the type of interrupt indicated by the GICH\_LRn.VirtualID field. When the EOI notification is sent to the virtual CPU interface, only the List registers are affected, and no notification is sent to the Distributor. See [List Registers, GICH\\_LRn](#) on page 5-176 for more information.

### Distributor-generated interrupts

Because the hardware interrupt deactivation mechanism does not support SGIs, the hypervisor must virtualize SGIs originating from the Distributor in the same way as hypervisor-generated interrupts. The hypervisor can virtualize the GICH\_LRn.CPUID field, because this field is not required to be the same as that of the original SGI. See [Completion of virtualized physical interrupts](#) on page 5-161 for more information about deactivating virtualized SGIs.

## 5.2.7 GIC Virtualization Extensions register mapping

In a GIC implementation that includes the Virtualization Extensions, the GIC provides a virtual CPU interface, with a complete set of virtual interface control registers, for each processor in the system. The GIC must make these virtual interface control registers accessible in the following ways:

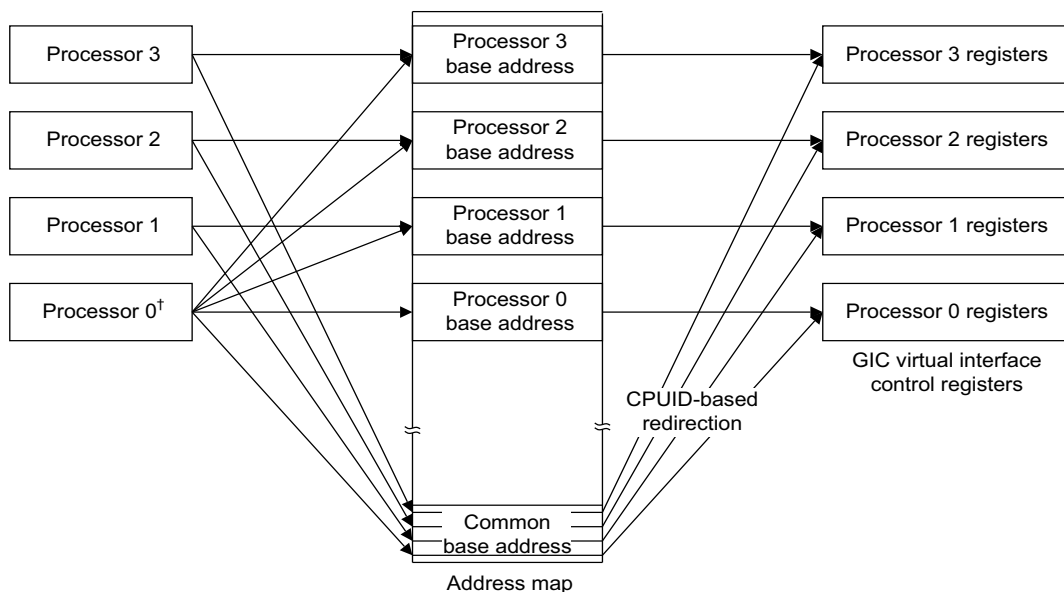
### Redirection through a common base address

The memory map includes a *common base address* for the virtual interface control registers. Each processor in the system can access its own GIC virtual interface control registers through this base address. The CPUID of the processor requesting access redirects the access to the GIC virtual interface control registers for that processor.

### Processor-specific base addresses

In addition to the common base address, the memory map contains, for each processor in the system, a processor-specific base address for the GIC virtual interface control registers. Any processor can use these addresses to access its own GIC virtual interface control registers, or to access the GIC virtual interface control registers of any other processor in the system.

[Figure 5-2](#) on page 5-166 shows this implementation.



† Use of the processor-specific base addresses is shown in full only for accesses by processor 0

**Figure 5-2 GIC virtual interface control register mappings**

## 5.3 GIC virtual interface control registers

The GIC virtual interface control registers are management registers. Configuration software on the processor must ensure they are accessible only by a hypervisor, or similar software.

———— **Note** ————

All GIC registers are 32-bits wide. Reserved register addresses are RAZ/WI.

Table 5-1 shows the register map for the GIC virtual interface control registers.

**Table 5-1 GIC virtual interface control register map**

Offset	Name	Type	Reset	Description
0x00	<a href="#">GICH_HCR</a>	RW	0x00000000	Hypervisor Control Register
0x04	<a href="#">GICH_VTR</a>	RO	IMPLEMENTATION DEFINED	VGIC Type Register
0x08	<a href="#">GICH_VMCR</a>	RW	IMPLEMENTATION DEFINED	Virtual Machine Control Register
0x0C	-	-	-	Reserved
0x10	<a href="#">GICH_MISR</a>	RO	0x00000000	Maintenance Interrupt Status Register
0x14-0x1C	-	-	-	Reserved
0x20	GICH_EISR0	RO	0x00000000	End of Interrupt Status Registers 0 and 1, see <a href="#">GICH_EISRn</a>
0x24	GICH_EISR1	RO	0x00000000	
0x28-0x2C	-	-	-	Reserved
0x30	GICH_ELSR0	RO	IMPLEMENTATION DEFINED <sup>a</sup>	Empty List Register Status Registers 0 and 1, see <a href="#">GICH_ELSRn</a>
0x34	GICH_ELSR1	RO	IMPLEMENTATION DEFINED <sup>a</sup>	
0x38-0xEC	-	-	-	Reserved
0xF0	<a href="#">GICH_APR</a>	RW	0x00000000	Active Priorities Register
0xF4-0xFC	-	-	RAZ/WI	Reserved for GICH_APR1-GICH_APR3
0x100	GICH_LR0	RW	0x00000000	List Registers 0-63, see <a href="#">GICH_LRn</a>
...	-	-	-	
0x1FC	GICH_LR63	RW	0x00000000	

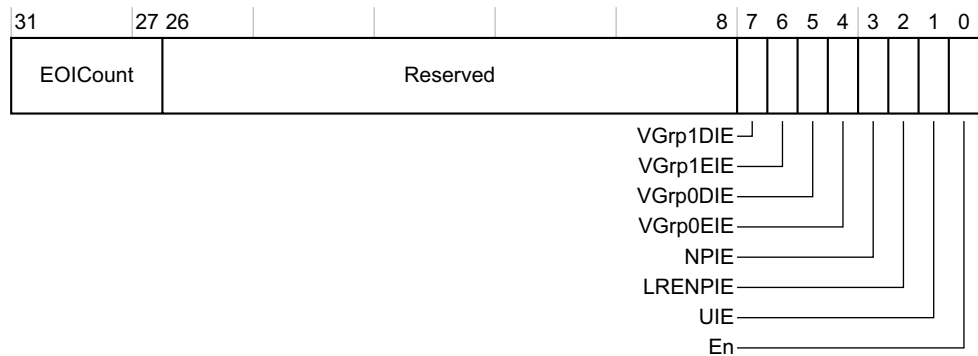
- a. Each bit that has a corresponding List register resets to 1, meaning that the reset value of the register depends on the number of List registers implemented.

### 5.3.1 Hypervisor Control Register, GICH\_HCR

The GICH\_HCR characteristics are:

- Purpose** This register contains control bits for the virtual CPU interface.
- Usage constraints** The GICH\_HCR.En bit must be set to 1 for any virtual or maintenance interrupt to be asserted.
- Configurations** This register is part of the GIC Virtualization Extensions.
- Attributes** See the register summary in [Table 5-1 on page 5-167](#).

[Figure 5-3](#) shows the GICH\_HCR bit assignments.



**Figure 5-3 GICH\_HCR bit assignments**

[Table 5-2](#) shows the GICH\_HCR bit assignments.

**Table 5-2 GICH\_HCR bit assignments**

Bit	Name	Description
[31:27]	EOICount	Counts the number of EOIs received that do not have a corresponding entry in the List registers. The virtual CPU interface increments this field automatically when a matching EOI is received. EOIs that do not clear a bit in the Active Priorities register, <a href="#">GICH_APR</a> do not cause an increment. Although not possible under correct operation, if an EOI occurs when the value of this field is 31, this field wraps to 0. The maintenance interrupt is asserted whenever this field is non-zero and the LRENPIE bit is set to 1.
[26:8]	-	Reserved.
[7]	VGrp1DIE	VM Disable Group 1 Interrupt Enable. Enables the signaling of a maintenance interrupt while signaling of Group 1 interrupts from the virtual CPU interface to the connected virtual machine is disabled: <b>0</b> Maintenance interrupt disabled. <b>1</b> Maintenance interrupt signaled while <a href="#">GICV_CTLR.EnableGrp1</a> is set to 0.
[6]	VGrp1EIE	VM Enable Group 1 Interrupt Enable. Enables the signaling of a maintenance interrupt while signaling of Group 1 interrupts from the virtual CPU interface to the connected virtual machine is enabled: <b>0</b> Maintenance interrupt disabled. <b>1</b> Maintenance interrupt signaled while <a href="#">GICV_CTLR.EnableGrp1</a> is set to 1.



**Table 5-2 GICH\_HCR bit assignments (continued)**

Bit	Name	Description
[5]	VGrp0DIE	VM Disable Group 0 Interrupt Enable. Enables the signaling of a maintenance interrupt while signaling of Group 0 interrupts from the virtual CPU interface to the connected virtual machine is disabled: <b>0</b> Maintenance interrupt disabled. <b>1</b> Maintenance interrupt signaled while <a href="#">GICV_CTLR.EnableGrp0</a> is set to 0.
[4]	VGrp0EIE	VM Disable Group 0 Interrupt Enable. Enables the signaling of a maintenance interrupt while signaling of Group 0 interrupts from the virtual CPU interface to the connected virtual machine is enabled: <b>0</b> Maintenance interrupt disabled. <b>1</b> Maintenance interrupt signaled while <a href="#">GICV_CTLR.EnableGrp0</a> is set to 1.
[3]	NPIE	No Pending Interrupt Enable. Enables the signaling of a maintenance interrupt while no pending interrupts are present in the List registers; <b>0</b> Maintenance interrupt disabled. <b>1</b> Maintenance interrupt signaled while the List registers contain no interrupts in the pending state.
[2]	LRENPIE	List Register Entry Not Present Interrupt Enable. Enables the signaling of a maintenance interrupt while the virtual CPU interface does not have a corresponding valid List register entry for an EOI request: <b>0</b> Maintenance interrupt disabled. <b>1</b> A maintenance interrupt is asserted while the EOICount field is not 0.
[1]	UIE	Underflow Interrupt Enable. Enables the signaling of a maintenance interrupt when the List registers are empty, or hold only one valid entry: <b>0</b> Maintenance interrupt disabled. <b>1</b> A maintenance interrupt is asserted if none, or only one, of the List register entries is marked as a valid interrupt.
[0]	En	Enable. Global enable bit for the virtual CPU interface: <b>0</b> Virtual CPU interface operation disabled. <b>1</b> Virtual CPU interface operation enabled. When this field is set to 0: <ul style="list-style-type: none"> <li>• the virtual CPU interface does not signal any maintenance interrupts</li> <li>• the virtual CPU interface does not signal any virtual interrupts</li> <li>• a read of <a href="#">GICV_IAR</a> or <a href="#">GICV_AIAR</a> returns a spurious interrupt ID.</li> </ul>

The VGrp1DIE, VGrp1EIE, VGrp0DIE, and VGrp0EIE bits enable the hypervisor to track the virtual CPU interfaces that are enabled. The hypervisor can then route interrupts that have more than one target correctly and efficiently, without having to read the virtual CPU interface status.

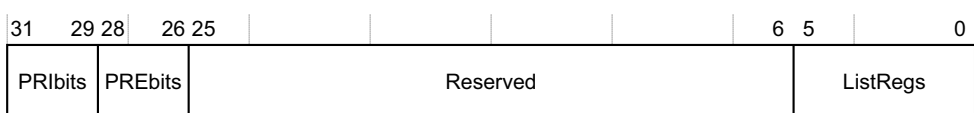
See [Maintenance interrupts on page 5-164](#) and [Maintenance Interrupt Status Register, GICH\\_MISR on page 5-172](#) for more information.

### 5.3.2 VGIC Type Register, GICH\_VTR

The GICH\_VTR characteristics are:

- Purpose** This is a read-only register that provides the following information about the implementation of the GIC Virtualization Extensions:
- number of priority levels supported
  - number of preemption levels supported
  - number of implemented List registers.
- Usage constraints** There are no usage constraints.
- Configurations** This register is part of the GIC Virtualization Extensions.
- Attributes** See the register summary in [Table 5-1 on page 5-167](#).

[Figure 5-4](#) shows the GICH\_VTR bit assignments.



**Figure 5-4 GICH\_VTR bit assignments**

[Table 5-3](#) shows the GICH\_VTR bit assignments.

**Table 5-3 GICH\_VTR bit assignments**

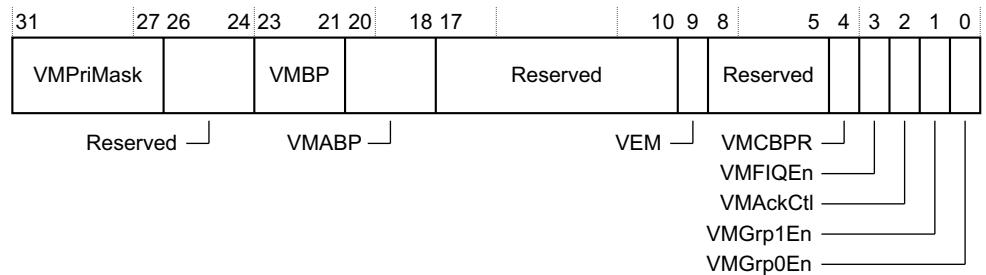
Bit	Name	Description
[31:29]	PRlbits	The number of priority bits implemented, minus one. In GICv2, the only valid value is 5 bits: <b>100</b> 32 priority levels.
[28:26]	PREbits	The number of preemption bits implemented, minus one. In GICv2, the only valid value is 5 bits: <b>100</b> 32 preemption levels
[25:6]	-	Reserved, RAZ
[5:0]	ListRegs	The number of implemented List registers, minus one. For example, a value of 0b111111 indicates that the maximum 64 List registers are implemented.

### 5.3.3 Virtual Machine Control Register, GICH\_VMCR

The GICH\_VMCR characteristics are:

- Purpose** Enables the hypervisor to save and restore the virtual machine view of the GIC state.
- Usage constraints** There are no usage constraints.
- Configurations** This register is part of the GIC Virtualization Extensions.
- Attributes** See the register summary in [Table 5-1 on page 5-167](#).

[Figure 5-5](#) shows the GICH\_VMCR bit assignments.



**Figure 5-5 GICH\_VMCR bit assignments**

[Table 5-2 on page 5-168](#) shows the GICH\_VMCR bit assignments.

**Table 5-4 GICH\_VMCR bit assignments**

Bit	Name	Description
[31:27]	VMPriMask	Alias of <a href="#">GICV_PMR</a> .Priority.
[26:24]	-	Reserved.
[23:21]	VMBP	Alias of <a href="#">GICV_BPR</a> .Binary point. On reset, this bit is set to the minimum supported value of <a href="#">GICV_BPR</a> .
[20:18]	VMABP <sup>a</sup>	Alias of <a href="#">GICV_ABPR</a> .Binary point.
[17:10]	-	Reserved.
[9]	VEM	Alias of <a href="#">GICV_CTLR</a> .EOImode.
[8:5]	-	Reserved.
[4]	VMCBPR	Alias of <a href="#">GICV_CTLR</a> .CBPR.
[3]	VMFIQEn	Alias of <a href="#">GICV_CTLR</a> .FIQEn.
[2]	VMAckCtl	Alias of <a href="#">GICV_CTLR</a> .AckCtl.
[1]	VMGrp1En	Alias of <a href="#">GICV_CTLR</a> .EnableGrp1.
[0]	VMGrp0En	Alias of <a href="#">GICV_CTLR</a> .EnableGrp0.

a. On reset, this is set to the minimum Group 1 binary point value, that is, the minimum of VMBP+1, saturated to 7.

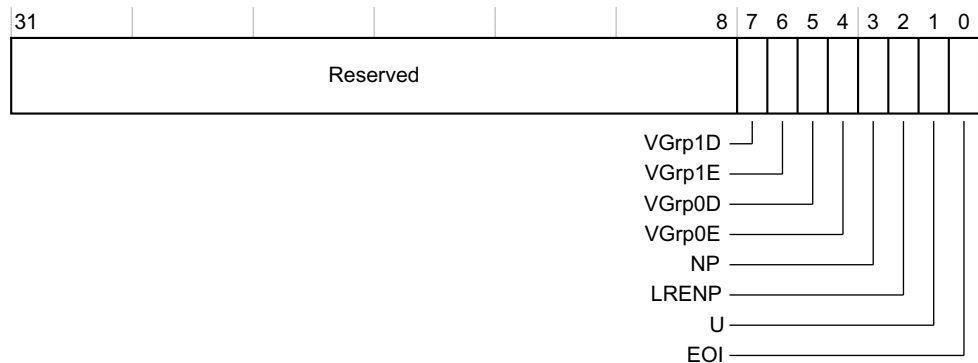
The GICH\_VMCR is a control register that contains read and write aliases of architecture state in the virtual machine view, enabling the hypervisor to save and restore this state with a single read or write, without accessing the GIC virtual CPU interface registers individually.

### 5.3.4 Maintenance Interrupt Status Register, GICH\_MISR

The GICH\_MISR characteristics are:

- Purpose** Indicates which maintenance interrupts are asserted.
- Usage constraints** A maintenance interrupt is asserted only if at least one bit is set in this register and if the global enable bit, GICH.HCR.En, is set to 1.
- Configurations** This register is part of the GIC Virtualization Extensions.
- Attributes** See the register summary in [Table 5-1 on page 5-167](#).

[Figure 5-6](#) shows the GICH\_MISR bit assignments.



**Figure 5-6 GICH\_MISR bit assignments**

[Table 5-5](#) shows the GICH\_MISR bit assignments.

**Table 5-5 GICH\_MISR bit assignments**

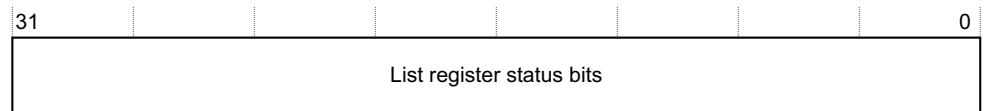
Bit	Name	Description
[31:8]	-	Reserved.
[7]	VGrp1D	Disabled Group 1 maintenance interrupt. Asserted whenever <a href="#">GICH_HCR.VGrp1DIE</a> is set and <a href="#">GICH_VMCR.VMGrp1En==0</a> .
[6]	VGrp1E	Enabled Group 1 maintenance interrupt. Asserted whenever <a href="#">GICH_HCR.VGrp1EIE</a> is set and <a href="#">GICH_VMCR.VMGrp1En==1</a> .
[5]	VGrp0D	Disabled Group 0 maintenance interrupt. Asserted whenever <a href="#">GICH_HCR.VGrp0DIE</a> is set and <a href="#">GICH_VMCR.VMGrp0En==0</a> .
[4]	VGrp0E	Enabled Group 0 maintenance interrupt. Asserted whenever <a href="#">GICH_HCR.VGrp0EIE</a> is set and <a href="#">GICH_VMCR.VMGrp0En==1</a> .
[3]	NP	No Pending maintenance interrupt. Asserted whenever <a href="#">GICH_HCR.NPIE==1</a> and no List register is in pending state.
[2]	LRENp	List Register Entry Not Present maintenance interrupt. Asserted whenever <a href="#">GICH_HCR.LRENPIE==1</a> and <a href="#">GICH_HCR.EOICount</a> is non-zero.
[1]	U	Underflow maintenance interrupt. Asserted whenever <a href="#">GICH_HCR.UIE</a> is set and if none, or only one, of the List register entries are marked as a valid interrupt, that is, if the corresponding <a href="#">GICH_LRn.State</a> bits do not equal 0x0.
[0]	EOI	EOI maintenance interrupt. Asserted whenever at least one List register is asserting an EOI Interrupt. At least one bit in <a href="#">GICH_EISRn==1</a> .

### 5.3.5 End of Interrupt Status Registers, GICH\_EISR0 and GICH\_EISR1

The GICH\_EISR characteristics are:

- Purpose** When a maintenance interrupt is received, these registers help determine which List registers have outstanding EOI interrupts that require servicing.
- Usage constraints** Bits corresponding to unimplemented List registers always RAZ.
- Configurations** These registers are part of the GIC Virtualization Extensions. The number of GICH\_EISRs depends on the number of List registers implemented. GICH\_EISR0 corresponds to List registers 0-31 and GICH\_EISR1 corresponds to List registers 32-63.
- Attributes** See the register summary in [Table 5-1 on page 5-167](#).

[Figure 5-7](#) shows the GICH\_EISR0 bit assignments.



**Figure 5-7 GICH\_EISR0 bit assignments**

[Table 5-6](#) shows the GICH\_EISR0 bit assignments.

**Table 5-6 GICH\_EISR0 bit assignments**

Bits	Name	Function
[31:0]	List register EOI status bits 0-31	For each bit: <b>0</b> Corresponding List register does not have an EOI. <b>1</b> Corresponding List register has an EOI. <sup>a</sup> See <a href="#">List Registers, GICH_LRn on page 5-176</a> for more information.

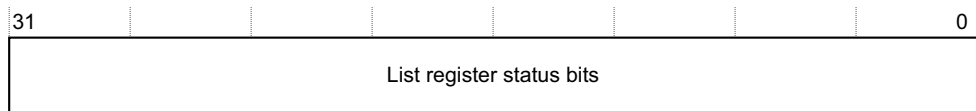
a. For any [GICH\\_LRn](#), the corresponding status bit is set to 1 if ([GICH\\_LRn.State==00](#) && [GICH\\_LRnn.HW ==0](#) && [GICH\\_LRn.EOI==1](#)).

### 5.3.6 Empty List Register Status Registers, GICH\_ELRSR0 and GICH\_ELRSR1

The GICH\_ELRSR characteristics are:

- Purpose** These registers can be used to locate a usable List register when the hypervisor is delivering an interrupt to a Guest OS.
- Usage constraints** Bits corresponding to unimplemented List registers always RAZ.
- Configurations** These registers are part of the GIC Virtualization Extensions. The number of GICH\_ELRSRs depends on the number of List registers implemented. GICH\_ELRSR0 corresponds to List registers 0-31 and GICH\_ELRSR1 corresponds to List registers 32-63.
- Attributes** See the register summary in [Table 5-1 on page 5-167](#).

[Figure 5-8](#) shows the GICH\_ELRSR0 bit assignments.



**Figure 5-8 GICH\_ELRSR0 bit assignments**

[Table 5-7](#) shows the GICH\_ELRSR0 bit assignments.

**Table 5-7 GICH\_ELRSR0 bit assignments**

Bits	Name	Function
[31:0]	List register status bits 0-31	<p>For each bit:</p> <p><b>0</b> The corresponding List register, if implemented, contains a valid interrupt. Using this List register can result in overwriting a valid interrupt.</p> <p><b>1</b> The corresponding List register does not contain a valid interrupt. The List register is empty and can be used without overwriting a valid interrupt or losing an EOI maintenance interrupt.<sup>a</sup></p> <p>See <a href="#">List Registers, GICH_LRn on page 5-176</a> for more information.</p>

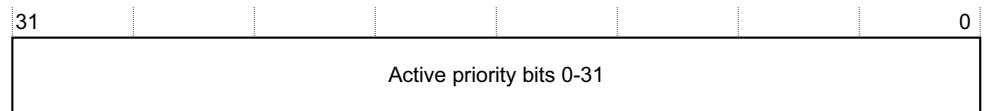
a. For any [GICH\\_LRn](#), the corresponding status bit is set to 1 if ([GICH\\_LRn.State==00](#) && ([GICH\\_LRn.HW==1](#) || [GICH\\_LRn.EOI==0](#))).

### 5.3.7 Active Priorities Register, GICH\_APR

The GICH\_APR characteristics are:

- Purpose** This register tracks which preemption levels are active in the virtual CPU interface, and is used to determine the current active priority. Corresponding bits are set in this register when an interrupt is acknowledged, based on [GICH\\_LRn.Priority](#), and the least significant set bit is cleared on EOI.
- Usage constraints** The bit to be set is determined by the top five bits of the interrupt priority.
- Configurations** This register is part of the GIC Virtualization Extensions.
- Attributes** See the register summary in [Table 5-1 on page 5-167](#).

[Figure 5-9](#) shows the GICH\_APR bit assignments.



**Figure 5-9 GICH\_APR bit assignments**

[Table 5-8](#) shows the GICH\_APR bit assignments.

**Table 5-8 GICH\_APR bit assignments**

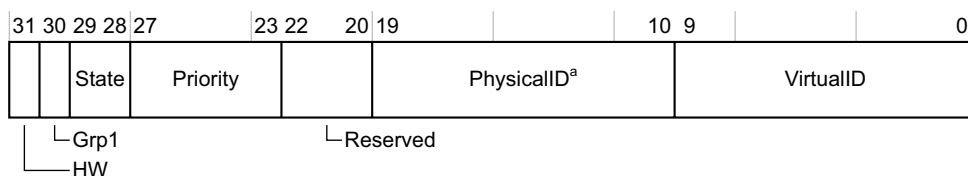
Bit	Name	Description
[31:0]	Active priority bits 0-31	Determines whether the corresponding preemption level is active: <b>0</b> the preemption level is not active <b>1</b> the preemption level is active.

### 5.3.8 List Registers, GICH\_LRn

The GICH\_LR characteristics are:

- Purpose** Provides interrupt context information for the virtual CPU interface.
- Usage constraints** There are no usage constraints.
- Configurations** These registers are part of the GIC Virtualization Extensions.  
A maximum of 64 List registers can be provided. The `GICH_VTR.ListRegs` bit defines the actual number implemented. All higher numbered List registers are RAZ/WI.  
Any unused bits in this register are RAZ/WI.
- Attributes** See the register summary in [Table 5-1 on page 5-167](#).

[Figure 5-10](#) shows the GICH\_LR bit assignments.



a These bits have different meaning when `GICH_LRn.HW==0`.

**Figure 5-10 GICH\_LR bit assignments**

[Table 5-9](#) shows the GICH\_LR bit assignments.

**Table 5-9 GICH\_LR bit assignments**

Bit	Name	Description
[31]	HW	Indicates whether this virtual interrupt is a hardware interrupt, meaning that it corresponds to a physical interrupt. Deactivation of the virtual interrupt also causes the deactivation of the physical interrupt with the ID that the PhysicalID field indicates.  <b>0</b> The interrupt is triggered entirely in software. No notification is sent to the Distributor when the virtual interrupt is deactivated. <b>1</b> A hardware interrupt. A deactivate interrupt request is sent to the Distributor when the virtual interrupt is deactivated, using bits [19:10], the PhysicalID, to indicate the physical interrupt ID. If <code>GICV_CTLR.EOImode == 0</code> , this request corresponds to a write to the <code>GICV_EOIR</code> or <code>GICV_AEOIR</code> , otherwise it corresponds to a write to the <code>GICV_DIR</code> .
[30]	Grp1	Indicates whether this virtual interrupt is a Group 1 virtual interrupt.  <b>0</b> This is a Group 0 virtual interrupt. <code>GICV_CTLR.FIQEn</code> determines whether it is signaled as a virtual IRQ or as a virtual FIQ, and <code>GICV_CTLR.EnableGrp0</code> enables signaling of this interrupt to the virtual machine. <b>1</b> This is a Group 1 virtual interrupt, signaled as a virtual IRQ. <code>GICV_CTLR.EnableGrp1</code> enables the signaling of this interrupt to the virtual machine.  ——— <b>Note</b> ——— The <code>GICV_CTLR.CBPR</code> bit controls whether <code>GICV_BPR</code> or <code>GICV_ABPR</code> is used to determine if a pending Group 1 interrupt has sufficient priority to preempt current execution.



**Table 5-9 GICH\_LR bit assignments (continued)**

Bit	Name	Description
[29:28]	State	<p>The state of the interrupt. This has one of the following values:</p> <p><b>00</b> invalid <b>01</b> pending <b>10</b> active <b>11</b> pending and active.</p> <p>The GIC updates these state bits as virtual interrupts proceed through the interrupt life cycle. Entries in the invalid state are ignored, except for the purpose of generating virtual maintenance interrupts.</p> <p>———— <b>Note</b> ————</p> <p>For hardware interrupts, the pending and active state is held in the physical Distributor rather than the virtual CPU interface. A hypervisor must only use the pending and active state for software originated interrupts, which are typically associated with virtual devices, or SGIs.</p>
[27:23]	Priority	The priority of this interrupt.
[22:20]	-	Reserved.
[19:10]	PhysicalID	<p>The function of this bit depends on the value of the GICH_LR.HW bit, as follows.</p> <p><b>0</b> When GICH_LR.HW is set to 0, bits [19:10] have the following meanings:</p> <p style="padding-left: 2em;"><b>[19]</b> EOI Indicates whether this interrupt triggers an EOI maintenance interrupt. <b>0</b> No maintenance interrupt is asserted. <b>1</b> A maintenance interrupt is asserted to signal EOI when the interrupt state is invalid, which typically occurs when the interrupt is deactivated.</p> <p style="padding-left: 2em;"><b>[18:13]</b> Reserved, SBZ</p> <p style="padding-left: 2em;"><b>[12:10]</b> CPUID If the interrupt has the VirtualID for an SGI, that is, 0-15, this field shows the requesting CPU ID. This appears in the relevant field of the VM Interrupt Acknowledge register, <a href="#">GICV_IAR</a> or <a href="#">GICV_AIAR</a>. Otherwise, this field must be set to 0.</p> <p><b>1</b> When GICH_LR.HW is set to 1, this field indicates the physical interrupt ID that the hypervisor forwards to the Distributor.</p> <p>———— <b>Note</b> ————</p> <p>When used to indicate the physical interrupt ID, this field is only required to implement enough bits to hold a valid value for the configuration used. Any unused higher order bits are RAZ/WI.</p> <p>—————</p> <p>If the value of PhysicalID is 0-15, or 1020-1023, behavior is UNPREDICTABLE. If the value of PhysicalID is 16-31, this field applies to the PPI associated with the same physical CPUID as the virtual CPU interface requesting the deactivation.</p>
[9:0]	VirtualID	<p>This ID is returned to the Guest OS when the interrupt is acknowledged through the VM Interrupt Acknowledge register, <a href="#">GICV_IAR</a>.</p> <p>Each valid interrupt stored in the List registers must have a unique VirtualID for that virtual CPU interface. If the value of VirtualID is 1020-1023, behavior is UNPREDICTABLE.</p>

## 5.4 The virtual CPU interface

A GIC virtual CPU interface signals virtual interrupts to a connected processor, subject to the normal GIC handling and prioritization rules. The GIC virtual CPU interface registers have the same general format as the GIC physical CPU interface registers and expected behavior is that a virtual machine cannot distinguish between them. The virtual interface control registers control virtual CPU interface operation, and in particular, the virtual CPU interface uses the contents of the List registers to determine when to signal virtual interrupts. When a processor accesses the virtual CPU interface the List registers are updated.

---

### Note

- Virtual interrupts are always handled through the virtual CPU interfaces.
  - On the connected processor, if the processor is in a Non-secure PL1 or PL0 mode, virtual interrupts are signaled to the current virtual machine.
  - In addition, a virtual machine can receive virtual IRQs and virtual FIQs signaled directly by the hypervisor. These exceptions are outside the scope of this specification. A virtual machine cannot distinguish:
    - A virtual exception signaled by the GIC from a corresponding virtual exception signaled directly by the hypervisor.
    - A virtual exception from the corresponding physical exception.
  - A virtual CPU interface does not require power management support, and therefore `GICV_CTLR` does not implement the `IRQByDisGrp1`, `FIQByDisGrp1`, `IRQByDisGrp0`, and `FIQByDisGrp0` bits that are supported by `GICC_CTLR`.
- 

### 5.4.1 Enabling and disabling virtual interrupts

The `GICV_CTLR` `EnableGrp1` and `EnableGrp0` bits control the signaling of Group 0 and Group 1 virtual interrupts to the connected virtual machine. When virtual interrupt signaling is disabled, the virtual CPU interface returns a spurious interrupt ID to any corresponding `GICV_IAR` or `GICV_AIAR` access. It is IMPLEMENTATION DEFINED whether disabling virtual interrupt signaling has the same effect on `GICV_HPPIR` and `GICV_AHPPIR`.

When enabling and disabling virtual interrupt generation, it might be necessary to reroute one or more interrupts, see *Maintenance interrupts* on page 5-164 for more information about associated events.

## 5.5 GIC virtual CPU interface registers

These registers provide the virtual CPU interface accessed by the virtual machine. Typically, a virtual machine is unaware of any difference between virtual interrupts and physical interrupts. This means the programmers' model for handling virtual interrupts must be identical to that for handling physical interrupts. In general, these registers have the same format as the GIC physical CPU interface registers, but they operate on the interrupt view defined primarily by the List registers.

These registers are memory-mapped, with defined offsets from an IMPLEMENTATION DEFINED GICV\_\* register base address.

———— **Note** ————

The offset of each GICV\_\* register is the same as the offset of the corresponding register for the physical CPU interface. For example, GICV\_PMR is at offset 0x0004 from the GICV\_\* register base address, and GICC\_PMR is at the same offset from the GICC\_\* register base address.

This means that:

- the hypervisor can use the stage 2 address translations to map the virtual CPU interface accesses to the correct physical addresses.
- software, whether accessing the registers of a physical CPU interface or of a virtual CPU interface, uses the same register addresses.

Table 5-10 shows the register map for the GIC virtual CPU interface registers.

**Table 5-10 GIC virtual CPU interface register map**

Offset	Name	Type	Reset	Description
0x0000	GICV_CTLR	RW	0x00000000	Virtual Machine Control Register
0x0004	GICV_PMR	RW	0x00000000	VM Priority Mask Register
0x0008	GICV_BPR	RW	0x00000002	VM Binary Point Register
0x000C	GICV_IAR	RO	0x000003FF	VM Interrupt Acknowledge Register
0x0010	GICV_EOIR	WO	-	VM End of Interrupt Register
0x0014	GICV_RPR	RO	0x000000FF	VM Running Priority Register
0x0018	GICV_HPPIR	RO	0x000003FF	VM Highest Priority Pending Interrupt Register
0x001C	GICV_ABPR	RW	0x00000003	VM Aliased Binary Point Register
0x0020	GICV_AIAR	RO	0x000003FF	VM Aliased Interrupt Acknowledge Register
0x0024	GICV_AEOIR	WO	-	VM Aliased End of Interrupt Register
0x0028	GICV_AHPPIR	RO	0x000003FF	VM Aliased Highest Priority Pending Interrupt Register
0x002C-0x003C	-	-	-	Reserved
0x0040-0x00CC	-	-	-	IMPLEMENTATION DEFINED
0x00D0-0x00DC	GICV_APRn	RW	IMPLEMENTATION DEFINED	VM Active Priorities Registers
0x00E0-0x00EC	-	-	RAZ/WI	Reserved for second set of Active Priorities Registers, see the Note in the GICV_APRn description.
0x00F0-0x00F8	-	-	-	Reserved

**Table 5-10 GIC virtual CPU interface register map (continued)**

Offset	Name	Type	Reset	Description
0x00FC	<a href="#">GICV_IIDR</a>	RO	IMPLEMENTATION DEFINED	VM CPU Interface Identification Register
0x00FC-0x0FFC	-	-	-	Reserved
0x1000	<a href="#">GICV_DIR</a>	WO	-	VM Deactivate Interrupt Register
0x1004-0x1FFC	-	-	-	Reserved

### 5.5.1 Virtual Machine Control Register, GICV\_CTLR

The GICV\_CTLR characteristics are:

**Purpose** Enables and disables Group 0 and Group 1 virtual interrupts.

———— **Note** —————

[GICH\\_LRn.Grp1](#) determines whether a virtual interrupt is Group 0 or Group 1.

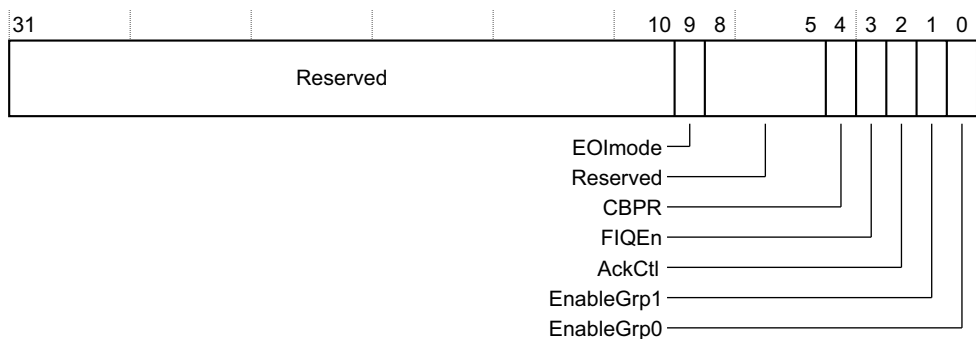
This register corresponds to the [GICC\\_CTLR](#) in the physical CPU interface.

**Usage constraints** There are no usage constraints.

**Configurations** This register is part of the GIC Virtualization Extensions.

**Attributes** See the register summary in [Table 5-10 on page 5-179](#).

[Figure 5-11](#) shows the GICV\_CTLR bit assignments:



**Figure 5-11 GICV\_CTLR bit assignments**

[Table 5-11 on page 5-181](#) shows the GICV\_CTLR bit assignments.

**Table 5-11 GICV\_CTLR bit assignments**

Bits	Name	Function
[31:10]	-	Reserved.
[9]	EOImode	<p>Controls the behavior associated with the <a href="#">GICV_EOIR</a>, <a href="#">GICV_AEOIR</a>, and <a href="#">GICV_DIR</a> registers:</p> <p><b>0</b>      <a href="#">GICV_EOIR</a> and <a href="#">GICV_AEOIR</a> perform priority drop and deactivate interrupt operations simultaneously. <a href="#">GICV_DIR</a> is UNPREDICTABLE.</p> <p>When it has completed processing the interrupt, the virtual machine writes to <a href="#">GICV_EOIR</a> or <a href="#">GICV_AEOIR</a>, deactivating the interrupt. The write:</p> <ul style="list-style-type: none"> <li>• updates the List registers</li> <li>• causes the virtual CPU interface to signal the interrupt completion to the physical Distributor.</li> </ul> <p><b>1</b>      <a href="#">GICV_EOIR</a> and <a href="#">GICV_AEOIR</a> perform priority drop operation only. <a href="#">GICV_DIR</a> performs deactivate interrupt operation only.</p> <p>At some point during its interrupt processing, the virtual machine writes to <a href="#">GICV_EOIR</a> or <a href="#">GICV_AEOIR</a>. This write drops the priority of the virtual interrupt, by updating its entry in the List registers.</p> <p>When it has completed processing the interrupt, the virtual machine writes to <a href="#">GICV_DIR</a>. This write deactivates the virtual interrupt and:</p> <ul style="list-style-type: none"> <li>• updates the List registers</li> <li>• causes the virtual CPU interface to signal the interrupt completion to the physical Distributor.</li> </ul>
[8:5]	-	Reserved
[4]	CBPR	<p>Controls whether the <a href="#">GICV_BPR</a> controls both Group 0 and Group 1 virtual interrupts.</p> <p><b>0</b>      <a href="#">GICV_BPR</a> controls Group 0 virtual interrupts, and <a href="#">GICV_ABPR</a> controls Group 1 virtual interrupts</p> <p><b>1</b>      <a href="#">GICV_BPR</a> controls Group 0 and Group 1 virtual interrupts.</p> <p>See <i>The effect of interrupt grouping on priority grouping</i> on page 3-57 for more information about how <a href="#">GICC_CTLR.CBPR</a> affects accesses to <a href="#">GICC_BPR</a> and <a href="#">GICC_ABPR</a>.</p>
[3]	FIQEn	<p>Controls whether interrupts marked as Group 0 are presented as virtual FIQs:</p> <p><b>0</b>      Group 0 interrupts are presented as virtual IRQs</p> <p><b>1</b>      Group 0 interrupts are presented as virtual FIQs.</p>
[2]	AckCtl	<p>ARM deprecates use of this bit. ARM strongly recommends that software is written to operate with this bit always set to 0.</p> <p>Controls whether a read of the <a href="#">GICV_IAR</a>, when the highest priority pending interrupt is a Group 1 interrupt, causes the CPU interface to acknowledge the interrupt.</p> <p><b>0</b>      If the highest priority pending interrupt is a Group 1 interrupt, a read of the <a href="#">GICV_IAR</a> returns an Interrupt ID of 1022. The read does not acknowledge the interrupt, and the pending status of the interrupt is unchanged.</p> <p><b>1</b>      If the highest priority pending interrupt is a Group 1 interrupt, a read of the <a href="#">GICV_IAR</a> returns the Interrupt ID of the Group 1 interrupt. The read acknowledges the interrupt, and the status of the interrupt becomes active, or active and pending.</p> <p style="text-align: center;">———— <b>Note</b> ————</p> <p style="text-align: center;">Only hypervisor-generated interrupts can be active and pending.</p>
[1]	EnableGrp1	<p>Enables the signaling of Group 1 virtual interrupts by the virtual CPU interface to the virtual machine:</p> <p><b>0</b>      Signaling of Group 1 interrupts disabled.</p> <p><b>1</b>      Signaling of Group 1 interrupts enabled.</p>

**Table 5-11 GICV\_CTLR bit assignments (continued)**

<b>Bits</b>	<b>Name</b>	<b>Function</b>
[0]	EnableGrp0	Enables the signaling of Group 0 virtual interrupts by the virtual CPU interface to the virtual machine: <b>0</b> Signaling of Group 0 interrupts disabled. <b>1</b> Signaling of Group 0 interrupts enabled.

## 5.5.2 VM Priority Mask Register, GICV\_PMR

The GICV\_PMR characteristics are:

**Purpose** Provides a virtual interrupt priority filter. Only virtual interrupts with higher priority than the value in this register can be signaled to the processor.

———— **Note** —————

Higher priority corresponds to a lower Priority field value.

The Priority field of this register is aliased to the VMPriMask field in [GICH\\_VMCR](#), to enable the state to be switched easily between virtual machines during context-switching.

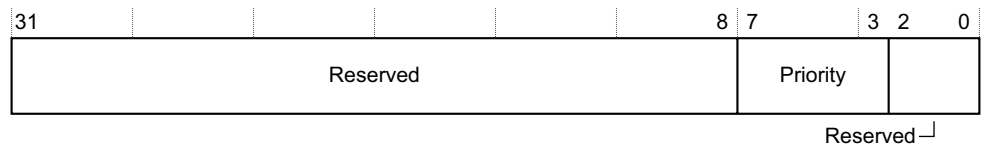
This register corresponds to the [GICC\\_PMR](#) in the physical CPU interface.

**Usage constraints** There are no usage constraints.

**Configurations** This register is part of the GIC Virtualization Extensions.

**Attributes** See the register summary in [Table 5-10 on page 5-179](#).

[Figure 5-12](#) shows the GICV\_PMR bit assignments.



**Figure 5-12 GICV\_PMR bit assignments**

GICV\_PMR is similar to [GICC\\_PMR](#), the corresponding register in the GIC physical CPU interface, except that bits [2:0] are reserved. This is because the virtual CPU interface supports fewer priority values than the maximum number of values that the physical CPU interface can support. See the [GICC\\_PMR](#) description for more information about the bit assignments.

### 5.5.3 VM Binary Point Register, GICV\_BPR

The GICV\_BPR characteristics are:

**Purpose** The register defines the point at which the priority value fields for the virtual interrupts split into two parts, the *group priority* field and the *subpriority* field. The group priority field is used to determine interrupt preemption. For more information see [Preemption on page 3-45](#) and [Priority grouping on page 3-45](#).

This register is used to determine the priority grouping for Group 0 interrupts and, if the GICV\_CTLR.CBPR bit is 1, for Group 1 interrupts also. This register corresponds to the [GICC\\_BPR](#) in the physical CPU interface.

**Usage constraints** The minimum binary point value is determined by the value of [GICH\\_VTR.PRE](#) bits. A GIC that includes the Virtualization Extensions supports a maximum of 32 preemption levels, corresponding to a minimum binary point value of 2. An attempt to program the binary point field to a value less than the minimum value sets the field to the minimum value. On a reset, the binary point field is set to the minimum supported value.

**Configurations** This register is part of the GIC Virtualization Extensions.

**Attributes** See the register summary in [Table 5-10 on page 5-179](#).

[Figure 5-13](#) shows the GICV\_BPR bit assignments.



**Figure 5-13 GICV\_BPR bit assignments**

The GICV\_BPR bit assignments are the same as assignments for the [GICC\\_BPR](#), the corresponding register in the physical CPU interface. See the [GICC\\_BPR](#) description for more information.

The Binary point field of this register is aliased to the [GICH\\_VMCR.VMBP](#) field, to enable the state to be switched easily between virtual machines during context-switching.



### 5.5.4 VM Interrupt Acknowledge Register, GICV\_IAR

<b>Purpose</b>	The virtual machine reads this register to obtain the interrupt ID of the signaled virtual interrupt. This read acts as an acknowledge for the interrupt. This register corresponds to the <a href="#">GICC_IAR</a> in the physical CPU interface.
<b>Usage constraints</b>	There are no usage constraints.
<b>Configurations</b>	This register is part of the GIC Virtualization Extensions.
<b>Attributes</b>	See the register summary in <a href="#">Table 5-10 on page 5-179</a> .

[Figure 5-14](#) shows the GICV\_IAR bit assignments.



**Figure 5-14** GICV\_IAR bit assignments

The GICV\_IAR bit assignments are the same as the bit assignments for the [GICC\\_IAR](#), the corresponding register in the physical CPU interface. See the [GICC\\_IAR](#) description for more information.

When the processor reads this register, the virtual CPU interface acknowledges the highest priority pending virtual interrupt and sets the state in the corresponding List register to active. The appropriate bit in the Active Priorities register, [GICH\\_APR](#) is set to 1.

If the [GICH\\_LRn.HW](#) bit is set to 0, indicating that the interrupt is triggered in software, then bits [12:10] of the [GICH\\_LRn](#), that indicate the CPU ID, are returned in the GICV\_IAR.CPUID field. Otherwise GICV\_IAR.CPUID field reads as zero.

[Table 5-12](#) shows all possible GICV\_IAR reads for a virtual CPU interface.

**Table 5-12** Effect of reads of GICV\_IAR

Interrupt status	GICV_CTLR.AckCtl	Returned interrupt ID
Highest priority pending interrupt <sup>a</sup> is Group 1	1	ID of Non-secure interrupt
	0	Interrupt ID 1022
Highest priority pending interrupt <sup>a</sup> is Group 0	x	ID of Secure interrupt

**Table 5-12 Effect of reads of GICV\_IAR (continued)**

<b>Interrupt status</b>	<b>GICV_CTLR.AckCtl</b>	<b>Returned interrupt ID</b>
No pending interrupts <sup>a</sup>	x	Interrupt ID 1023
Interrupt signaling by virtual CPU interface disabled	x	Interrupt ID 1023

- a. Of sufficient priority to be signaled to the processor with the virtual CPU interface enabled and the GICH\_HCR.En bit set to 1

### 5.5.5 VM End of Interrupt Register, GICV\_EOIR

<b>Purpose</b>	The virtual machine writes to this register to inform the virtual CPU interface that it has completed its interrupt service routine for the specified virtual interrupt. This register corresponds to the <a href="#">GICC_EOIR</a> in the physical CPU interface.
<b>Usage constraints</b>	There are no usage constraints.
<b>Configurations</b>	This register is part of the GIC Virtualization Extensions.
<b>Attributes</b>	See the register summary in <a href="#">Table 5-10 on page 5-179</a> .

[Figure 5-15](#) shows the GICV\_EOIR bit assignments.



**Figure 5-15 GICV\_EOIR bit assignments**

The GICV\_EOIR bit assignments are the same as the bit assignments for the [GICC\\_EOIR](#), the corresponding register in the physical CPU interface. See the [GICC\\_EOIR](#) description for more information. This section describes how the behavior of GICV\_EOIR differs from the behavior of [GICC\\_EOIR](#).

The behavior of GICV\_EOIR depends on the setting of [GICV\\_CTLR.EOImode](#):

- 0** Both the priority drop and the deactivate interrupt effects occur.
- 1** Only the priority drop effect occurs.

If the [GICH\\_LLRn.HW](#) bit in the matching List register is set to 1, indicating a hardware interrupt, then a deactivate request is sent to the physical Distributor, identifying the Physical ID from the corresponding field in the List register. This effect is identical to a Non-secure write to [GICC\\_DIR](#) from the processor having that physical ID. This means that if the corresponding physical interrupt is in Group 0 the request is ignored.

See [Behavior of writes to GICC\\_EOIR, GICv2 on page 4-140](#) for more information.

A successful EOI request means that:

- The highest priority bit in the [GICH\\_APR](#) is cleared, causing the running priority to drop
- If [GICC\\_CTLR.EOImode](#) == 0, the interrupt is deactivated in the corresponding List register. If the interrupt corresponds to a hardware interrupt, the interrupt is also deactivated in the Distributor.

**———— Note ————**

The only interrupts that can target the hypervisor are Group 1 interrupts and therefore only Group 1 interrupts are deactivated in the Distributor.

[Table 5-13](#) provides a summary of GICV\_EOIR operation.

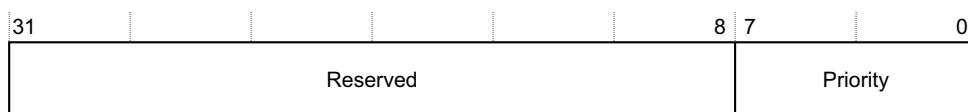
**Table 5-13 GICV\_EOIR operation**

Interrupt status	<a href="#">GICV_CTLR.AckCtl</a>	Effect
Group 0 interrupt	x	EOI operation performed
Group 1 interrupt	0	UNPREDICTABLE
Group 1 interrupt	1	EOI operation performed

### 5.5.6 VM Running Priority Register, GICV\_RPR

<b>Purpose</b>	Indicates the priority of the highest priority virtual interrupt that is active on the virtual CPU interface.  This register corresponds to the <a href="#">GICC_RPR</a> in the physical CPU interface.
<b>Usage constraints</b>	Depending on the implementation, if no bits are set in the Active Priorities register, <a href="#">GICH_APR</a> , indicating no active interrupts in the virtual CPU interface, the priority reads as 0xFF, or 0xF8 to reflect the number of supported interrupt priority bits, see <a href="#">VGIC Type Register</a> , <a href="#">GICH_VTR</a> on page 5-170 and <a href="#">Active Priorities Register</a> , <a href="#">GICH_APR</a> on page 5-175.
<b>Configurations</b>	This register is part of the GIC Virtualization Extensions.
<b>Attributes</b>	See the register summary in <a href="#">Table 5-10 on page 5-179</a> .

[Figure 5-16](#) shows the GICV\_RPR bit assignments.



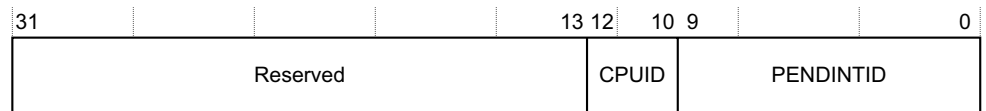
**Figure 5-16 GICV\_RPR bit assignments**

The GICV\_RPR bit assignments are the same as the bit assignments for the [GICC\\_RPR](#), the corresponding register in the physical CPU interface. See the [GICC\\_RPR](#) description for more information.

### 5.5.7 VM Highest Priority Pending Interrupt Register, GICV\_HPPIR

<b>Purpose</b>	Indicates the Interrupt ID of the pending virtual interrupt with the highest priority on the virtual CPU interface. Also returns the CPU ID for a software interrupt, that is, if the <a href="#">GICH_LRn.HW</a> bit==0.  This register corresponds to the <a href="#">GICC_HPPIR</a> in the physical CPU interface.
<b>Usage constraints</b>	Never returns the Interrupt ID of an interrupt that is active and pending. Returns a CPU ID only for interrupts triggered in software.
<b>Configurations</b>	This register is part of the GIC Virtualization Extensions.
<b>Attributes</b>	See the register summary in <a href="#">Table 5-10</a> on page 5-179.

[Figure 5-17](#) shows the GICV\_HPPIR bit assignments.



**Figure 5-17 GICV\_HPPIR bit assignments**

The GICV\_HPPIR bit assignments are the same as the bit assignments for the [GICC\\_HPPIR](#), the corresponding register in the physical CPU interface. See the [GICC\\_HPPIR](#) description for more information. This section describes how the behavior of GICV\_HPPIR differs from the behavior of [GICC\\_HPPIR](#).

In certain situations, such as when there are pending interrupts that are not stored in the List registers, this register returns an inappropriate spurious interrupt value, 1023. This is unlikely to cause any problems because:

- An implementation might not use this register
- The register works correctly even if it is inaccurate some of the time. A low priority pending interrupt is unlikely to affect the operating system if higher priority interrupts are active.

However, to guarantee no problems, ensure that the hypervisor always maintains the highest priority pending interrupt in the List registers, if one exists.

[Table 5-14](#) shows GICV\_HPPIR operation.

**Table 5-14 GICV\_HPPIR operation**

Interrupt status	GICV_CTLR.AckCtl	Returned interrupt ID
Highest priority pending interrupt is Group 0	x	ID of Group 0 interrupt
Highest priority pending interrupt is Group 1	0	Interrupt ID 1022
	1	ID of Group 1 interrupt
No pending interrupts	x	Interrupt ID 1023

### 5.5.8 VM Aliased Binary Point Register, GICV\_ABPR

The GICV\_ABPR characteristics are:

**Purpose** Provides a binary point register for the Group1 virtual interrupts.

———— **Note** —————

[GICH\\_LRn.Grp1](#) determines whether a virtual interrupt is Group 0 or Group 1.

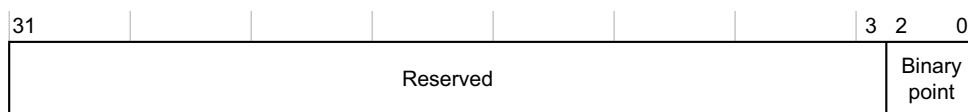
This register corresponds to the [GICC\\_ABPR](#) in the physical CPU interface.

**Usage constraints** The value contained in this register is one greater than the actual applied binary point value, see [The effect of interrupt grouping on priority grouping on page 3-57](#) for more information.

**Configurations** This register is part of the GIC Virtualization Extensions.

**Attributes** See the register summary in [Table 5-10 on page 5-179](#).

[Figure 5-18](#) shows the GICV\_ABPR bit assignments.



**Figure 5-18 GICV\_ABPR bit assignments**

The GICV\_ABPR bit assignments are the same as the bit assignments for the [GICC\\_ABPR](#), the corresponding register in the physical CPU interface. See the [GICC\\_ABPR](#) description for more information.

### 5.5.9 VM Aliased Interrupt Acknowledge Register, GICV\_AIAR

The GICV\_AIAR characteristics are:

- Purpose** A virtual machine reads this register to obtain the interrupt ID of the signaled virtual interrupt. This read acts as an acknowledge for a Group 1 virtual interrupt. Operation is similar to [GICV\\_IAR](#), except that the virtual machine only uses this register to acknowledge Group 1 interrupts. Group 0 interrupts are treated as spurious interrupts. This register corresponds to the [GICC\\_AIAR](#) in the physical CPU interface.
- Usage constraints** There are no usage constraints.
- Configurations** This register is part of the GIC Virtualization Extensions.
- Attributes** See the register summary in [Table 5-10 on page 5-179](#).

[Figure 5-19](#) shows the GICV\_AIAR bit assignments.



**Figure 5-19 GICV\_AIAR bit assignments**

The GICV\_AIAR bit assignments are the same as the bit assignments for the [GICC\\_AIAR](#), the corresponding register in the physical CPU interface. See the [GICC\\_AIAR](#) description for more information.

The operation of this register is similar to the operation of [GICV\\_IAR](#). When the virtual machine reads GICV\_AIAR, the corresponding interrupt List register is checked to determine whether the interrupt is Group 0 or Group 1:

- If the [GICH\\_LRn.Grp1](#) bit is 0, the interrupt is Group 0. The spurious interrupt ID 1023 is returned and the interrupt is not acknowledged.
- If the [GICH\\_LRn.Grp1](#) bit is 1, the interrupt is Group 1. The interrupt ID is returned, and if [GICH\\_LRn.HW](#) is 0, indicating that the interrupt is generated in software, the CPUID is returned also.

The List register entry is updated to active state, and the appropriate bit in the [GICH\\_APR](#), is set to 1.

If there is no pending virtual interrupt with sufficient priority to be signaled to the processor, then the spurious interrupt ID 1023 is returned.

[Table 5-15](#) shows GICV\_AIAR operation.

**Table 5-15 GICV\_AIAR operation**

Interrupt status	GICV_CTLR.AckCtl	Returned interrupt ID
Highest priority pending interrupt <sup>a</sup> is Group 0	x	Interrupt ID 1023
Highest priority pending interrupt is Group 1	x	ID of Group 1 interrupt
No pending interrupts of sufficient priority to be signaled	x	Interrupt ID 1023

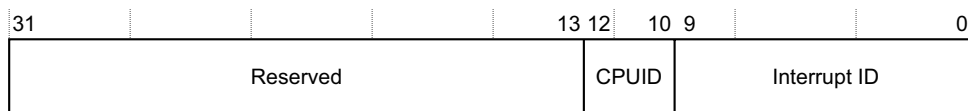
a. Of sufficient priority to be signaled to the processor if the virtual CPU interface is enabled and the [GICH\\_HCR.En](#) bit is set to 1.

### 5.5.10 VM Aliased End of Interrupt Register, GICV\_AEOIR

The GICV\_AEOIR characteristics are:

- Purpose** A virtual machine writes to this register to indicate completion of a Group 1 virtual interrupt. Operation is similar to [GICV\\_EOIR](#), except that the virtual machine only uses this register to indicate completion of Group 1 interrupts.  
This register corresponds to the [GICC\\_AEOIR](#) in the physical CPU interface.
- Usage constraints** There are no usage constraints.
- Configurations** This register is part of the GIC Virtualization Extensions.
- Attributes** See the register summary in [Table 5-10 on page 5-179](#).

[Figure 5-20](#) shows the GICV\_AEOIR bit assignments.



**Figure 5-20 GICV\_AEOIR bit assignments**

A successful EOI request means that:

- The highest priority bit in the GICH\_APR is cleared, causing the running priority to drop
- If GICC\_CTLR.EOImode == 0, the interrupt is deactivated in the corresponding List register. If the interrupt corresponds to a hardware interrupt, the interrupt is also deactivated in the Distributor.

———— **Note** —————

The only interrupts that can target the hypervisor are Group 1 interrupts and therefore only Group 1 interrupts are deactivated in the Distributor.

[Table 5-16](#) shows GICV\_AEOIR operation.

**Table 5-16 GICV\_AEOIR operation**

Interrupt status	GICV_CTLR.AckCtl	Effect
Group 0 interrupt	x	UNPREDICTABLE
Group 1 interrupt	x	EOI operation is performed



### 5.5.11 VM Aliased Highest Priority Pending Interrupt Register, GICV\_AHPPIR

The GICV\_AHPPIR characteristics are:

- Purpose** Returns the interrupt ID of the highest priority pending Group 1 virtual interrupt in the List registers.  
This register corresponds to the [GICC\\_AHPPIR](#) in the physical CPU interface.
- Usage constraints** Never returns the Interrupt ID of an interrupt that is active and pending.
- Configurations** This register is part of the GIC Virtualization Extensions.
- Attributes** See the register summary in [Table 5-10 on page 5-179](#).

[Figure 5-21](#) shows the GICV\_AHPPIR bit assignments.



**Figure 5-21 GICV\_AHPPIR bit assignments**

The GICV\_AHPPIR bit assignments are the same as the bit assignments of the [GICC\\_AHPPIR](#), the corresponding register in the physical CPU interface. See the [Aliased Highest Priority Pending Interrupt Register, GICC\\_AHPPIR on page 4-148](#) description for more information.

[Table 5-17](#) shows GICV\_AHPPIR operation.

**Table 5-17 GICV\_AHPPIR operation**

Interrupt status	GICV_CTLR.AckCtl	Returned interrupt ID
Highest priority pending interrupt is Group 0	x	Interrupt ID 1023
Highest priority pending interrupt is Group 1	x	ID of Group 1 interrupt
No pending interrupts of sufficient priority to be signaled	x	Interrupt ID 1023

## 5.5.12 VM Active Priorities Registers, GICV\_APRn

The GICV\_APR characteristics are:

**Purpose** For software compatibility, these registers are present in the virtual CPU interface. However, a virtual machine is not required to preserve and restore state during power down, and therefore does not have to use these registers.

———— **Note** —————

Instead, the hypervisor uses the [GICH\\_APR](#) register to save the GIC state for each virtual machine.

These registers correspond to the [GICC\\_APRn](#) registers in the physical CPU interface.

**Usage constraints** Because these registers are not required for preserving and restoring state, their content is IMPLEMENTATION DEFINED. Reading the content of these registers and then writing the same values does not change any state.

**Configurations** These registers are part of the GIC Virtualization Extensions.

ARM suggests implementing:

- [GICV\\_APR0](#) as an alias of [GICH\\_APR](#)
- the remaining GICV\_APRn registers as RAZ/WI.

**Attributes** See the register summary in [Table 5-10 on page 5-179](#).

The GICV\_APRn bit assignments are the same as the bit assignments of the [GICC\\_APRn](#) registers, the corresponding registers in the physical CPU interface. See the [GICC\\_APRn](#) description for more information.

———— **Note** —————

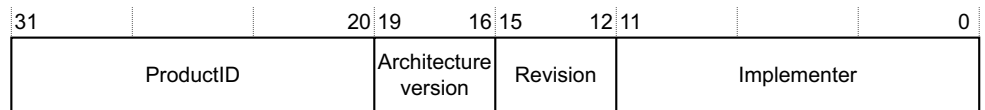
A virtualized processor does not require separate Secure and Non-secure APRs, and only a single set of Active Priorities registers, GICV\_APRn are defined. However, the register map allocates space for both sets of registers, to maximise software compatibility. The register space corresponding to the Non-secure APRs is RAZ/WI.

### 5.5.13 VM CPU Interface Identification Register, GICV\_IIDR

The GICV\_IIDR characteristics are:

- Purpose** Provides information about the implementer and revision of the virtual CPU interface.  
This register corresponds to the [GICC\\_IIDR](#) register in the physical CPU interface.
- Usage constraints** No usage constraints.
- Configurations** This register is part of the GIC Virtualization Extensions.
- Attributes** See the register summary in [Table 5-10 on page 5-179](#).

[Figure 5-22](#) shows the GICV\_IIDR bit assignments.



**Figure 5-22 GICV\_IIDR bit assignments**

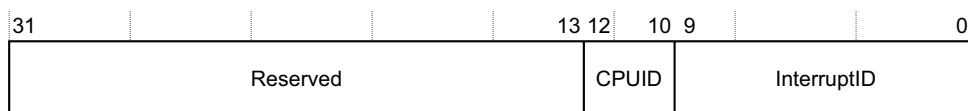
The GICV\_IIDR bit assignments are the same as the bit assignments of the [GICC\\_IIDR](#), the corresponding register in the physical CPU interface. See the [CPU Interface Identification Register, GICC\\_IIDR on page 4-152](#) description for more information.

## 5.5.14 VM Deactivate Interrupt Register, GICV\_DIR

The GICV\_DIR characteristics are:

<b>Purpose</b>	This register deactivates the virtual interrupt with the specified interrupt ID in the List registers.  This register corresponds to the <a href="#">GICC_DIR</a> register in the physical CPU interface.
<b>Usage constraints</b>	Writes to this register are valid only when <a href="#">GICV_CTLR.EOImode</a> is set to 1. If <a href="#">GICV_CTLR.EOImode</a> is set to 0, any write to this register is UNPREDICTABLE.
<b>Configurations</b>	This register is part of the GIC Virtualization Extensions.
<b>Attributes</b>	See the register summary in <a href="#">Table 5-10 on page 5-179</a> .

[Figure 5-23](#) shows the GICV\_DIR bit assignments.



**Figure 5-23 GICV\_DIR bit assignments**

The GICV\_DIR bit assignments are the same as the bit assignments for the [GICC\\_DIR](#), the corresponding register in the physical CPU interface. See the [GICC\\_DIR](#) description for more information. This section describes the behavior of the GICV\_DIR differs from the behavior of the [GICC\\_DIR](#).

When the virtual machine writes to this register, the specified interrupt in the List registers is changed from active to invalid, or from active and pending to pending. If the specified interrupt is present in the List registers but not in the active or pending and active states, the effect is UNPREDICTABLE. If the specified Interrupt does not exist in the List registers, the [GICH\\_HCR.EOIcount](#) field is incremented, potentially generating a maintenance interrupt.

———— **Note** —————

If the specified interrupt does not exist in the List registers, the virtual machine cannot recover the interrupt ID. Therefore, the hypervisor must ensure that, when [GICV\\_CTLR.EOImode](#) is set to 1, no more than one active interrupt is transferred from the List registers into a software list. If more than one active interrupt that is not stored in the List registers exists, the hypervisor must handle accesses to GICV\_DIR in software, typically by trapping these accesses.

If the [GICH\\_LRn.HW](#) bit in the matching List register is set to 1, indicating a hardware interrupt, then a deactivate request is sent to the physical Distributor, identifying the Physical ID from the corresponding field in the List register. This effect is identical to a Non-secure write to [GICC\\_DIR](#) from the processor having that physical ID. This means that if the corresponding physical interrupt is marked as Group 0, the request is ignored.

———— **Note** —————

Interrupt deactivation using GICV\_DIR is based on the provided interrupt ID, with no requirement to deactivate interrupts in any particular order. A single register is therefore used to deactivate Group 0 and Group 1 interrupts.

# Appendix A

## Pseudocode Index

This appendix gives an index of the pseudocode functions defined in this specification. It contains the following section:

- [Index of pseudocode functions on page A-198.](#)

## A.1 Index of pseudocode functions

Table A-1 is an index of the pseudocode functions defined in this specification. Where different forms of the function are used to support the architecture with and without the Security Extensions, the index refers to both forms.

———— **Note** —————

The pseudocode in this document follows the ARM architecture pseudocode conventions. For more information, see *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*.

**Table A-1 Pseudocode functions and procedures**

Function	Meaning	See
AcknowledgeInterrupt()	Set the active state and attempt to clear the pending state for the interrupt associated with argument InterruptID.	<i>General helper functions and definitions on page 3-61</i>
AnyActiveInterrupts()	Return TRUE if any interrupt is in the active state.	<i>General helper functions and definitions on page 3-61</i>
BinaryPointRegWrite()	Write behavior of accesses to GICC_BPR when the Security Extensions are implemented.	<i>Binary Point Register; GICC_BPR on page 4-133</i>
GIC_GenerateExceptions()	Exception generation by the CPU interface using the GIC prioritization scheme.	<i>Exception generation pseudocode on page 3-64</i> <i>Exception generation pseudocode, with interrupt grouping on page 3-64</i>
GIC_PriorityMask()	Return the priority mask to be used for priority grouping as part of interrupt prioritization	<i>General helper functions and definitions on page 3-61</i>
HighestPriorityPendingInterrupt()	Returns the ID of the highest priority interrupt that is pending. If no interrupts are pending, returns a spurious interrupt ID.	<i>General helper functions and definitions on page 3-61</i>
IgnoreWriteRequest()	No operation. Indicates cases where the GIC ignores a write to a register.	<i>General helper functions and definitions on page 3-61</i>
IsEnabled()	Return TRUE if the interrupt is enabled.	<i>General helper functions and definitions on page 3-61</i>
IsGrp0Int()	Return TRUE if the interrupt identified by the function argument is configured as a Group 0 interrupt.	<i>General helper functions and definitions on page 3-61</i>
IsPending()	Return TRUE if the interrupt identified by the function argument is pending.	<i>General helper functions and definitions on page 3-61</i>
MaskRegRead()	Read behavior of accesses to GICC_PMR when the Security Extensions are implemented.	<i>Interrupt Priority Mask Register; GICC_PMR on page 4-131</i>
MaskRegWrite()	Write behavior of accesses to GICC_PMR when the Security Extensions are implemented.	<i>Interrupt Priority Mask Register; GICC_PMR on page 4-131</i>

**Table A-1 Pseudocode functions and procedures (continued)**

<b>Function</b>	<b>Meaning</b>	<b>See</b>
PriorityIsHigher()	Return TRUE if the first argument of the function has a higher priority than the second argument.	<i>General helper functions and definitions on page 3-61</i>
PriorityRegRead()	Read behavior of accesses to the <a href="#">GICD_IPRIORITYRn</a> , <a href="#">GICC_PMR</a> and <a href="#">GICC_RPR</a> when the Security Extensions are implemented.	<i>The effect of the GIC Security Extensions on accesses to prioritization registers on page 3-66</i>
PriorityRegWrite()	Write behavior of accesses to the <a href="#">GICD_IPRIORITYRn</a> and <a href="#">GICC_PMR</a> when the Security Extensions are implemented.	<i>The effect of the GIC Security Extensions on accesses to prioritization registers on page 3-66</i>
ReadGICC_HPPIR()	Returns the value of <a href="#">GICC_HPPIR</a> read by a CPU access.	<i>Highest Priority Pending Interrupt Register, GICC_HPPIR on page 4-143</i>
ReadGICC_IAR()	Returns the value of <a href="#">GICC_IAR</a> read by a CPU access.	<i>Interrupt Acknowledge Register, GICC_IAR on page 4-135</i>
ReadGICC_RPR()	Returns the value of <a href="#">GICC_RPR</a> read by a CPU access.	<i>Running Priority Register, GICC_RPR on page 4-142</i>
ReadGICD_IPRIORITYR()	Return the priority value of the interrupt identified by the function argument, by reading the appropriate <a href="#">GICD_IPRIORITYRn</a> .	<i>General helper functions and definitions on page 3-61</i>
ReadGICD_ITARGETSR()	Returns an 8-bit field specifying which processors are to receive the interrupt specified by argument InterruptID.	<i>General helper functions and definitions on page 3-61</i>
SGI_CpuID()	Returns the ID of the highest priority processor for the software generated interrupt specified by InterruptID.	<i>General helper functions and definitions on page 3-61</i>
SignalFIQ()	If the input parameter is TRUE, signal the target processor to request an FIQ exception.	<i>General helper functions and definitions on page 3-61</i>
SignalIRQ()	If the input parameter is TRUE, signal the target processor to request an IRQ exception.	<i>General helper functions and definitions on page 3-61</i>
UpdateExceptionState()	GIC exception prioritization scheme used by the CPU interface	<i>Exception generation pseudocode on page 3-64</i>
WriteGICD_IPRIORITYR()	Set the priority value of the interrupt identified by the function argument, by writing to the appropriate <a href="#">GICD_IPRIORITYRn</a> .	<i>General helper functions and definitions on page 3-61</i>





# Appendix B

## Register Names

This appendix describes the relationship between the architectural names of the registers described in this specification, and their legacy aliases. It also provides an index of the architectural names. It contains the following sections:

- [Alternative register names](#) on page B-202
- [Register name aliases](#) on page B-203
- [Index of architectural names](#) on page B-204.

## B.1 Alternative register names

GICv2 suggests replacement register names for GICv1 registers. Table B-1 shows the GICv1 names and the GICv2 suggested replacement names for the registers in the Distributor.

**Table B-1 Replacement names for the registers in the Distributor**

Register	GICv2 name	GICv1 name
Distributor Control	GICD_CTLR	ICDDCR
Interrupt Controller Type	GICD_TYPER	ICDICTR
Distributor Implementer Identification	GICD_IIDR	ICDIIDR
Interrupt Group	GICD_IGROUPR <sub>n</sub>	ICDISR <sub>n</sub>
Interrupt Set-Active	GICD_ISACTIVER <sub>n</sub>	ICDABR <sub>n</sub>
Interrupt Set-Enable	GICD_ISENBALER <sub>n</sub>	ICDISER <sub>n</sub>
Interrupt Clear-Enable	GICD_ICENABLER <sub>n</sub>	ICDICER <sub>n</sub>
Interrupt Set-Pending	GICD_ISPENDR <sub>n</sub>	ICDISPR <sub>n</sub>
Interrupt Clear-Pending	GICD_ICPENDR <sub>n</sub>	ICDICPR <sub>n</sub>
Interrupt Priority	GICD_IPRIORITYR <sub>n</sub>	ICDIPR <sub>n</sub>
Interrupt Processor Targets	GICD_ITARGETSR <sub>n</sub>	ICDIPTR <sub>n</sub>
Interrupt Configuration	GICD_ICFGR <sub>n</sub>	ICDICR <sub>n</sub>
Software Generated Interrupt	GICD_SGIR	ICDSGIR
Identification	-	-

Table B-2 shows the GICv1 names and the GICv2 suggested replacement names for the registers in the CPU interface.

**Table B-2 Replacement names for the registers in the CPU interface**

Register	GICv2 name	GICv1 name
CPU Interface Control	GICC_CTLR	ICCICR
Priority Mask	GICC_PMR	ICCPMR
Binary Point Register	GICC_BPR	ICCBPR
Interrupt Acknowledge	GICC_IAR	ICCIAR
End of Interrupt	GICC_EOIR	ICCEOIR
Running Priority	GICC_RPR	ICCRPR
Aliased Binary Point	GICC_ABPR	ICCABPR
Highest Priority Pending Interrupt	GICC_HPPIR	ICCHPIR
CPU Implementer Identification	GICC_IIDR	ICCIIDR

## B.2 Register name aliases

Some implementations of this GIC architecture, for historical reasons, do not use the architectural names of the registers described in this specification. Developers must not rely on this distinction being maintained in future versions of the ARM GIC architecture. Table B-3 shows the alias names that are sometimes used for the registers in the Distributor.

**Table B-3 Alias names for the registers in the Distributor**

Register	Name	Alias
Distributor Control	GICD_CTLR	enable_s, enable_ns
Interrupt Controller Type	GICD_TYPER	ic_type_reg
Distributor Implementer Identification	GICD_IIDR	dist_ident_reg
Interrupt Group	GICD_IGROUPRn	int_security
Interrupt Set-Enable	GICD_ISENABLERn	enable_set
Interrupt Clear-Enable	GICD_ICENABLERn	enable_clr
Interrupt Set-Pending	GICD_ISPENDRn	pending_set
Interrupt Clear-Pending	GICD_ICPENDRn	pending_clr
Interrupt Priority	GICD_IPRIORITYRn	priority_level
Interrupt Processor Targets	GICD_ITARGETSRn	target
Interrupt Configuration	GICD_ICFGRn	int_config
Software Generated Interrupt	GICD_SGIR	sti_control
Identification	-	-

Table B-4 shows the alias names that are sometimes used for the registers in the CPU interface.

**Table B-4 Alias names for the registers in the CPU interface**

Register	Name	Alias
CPU Interface Control	GICC_CTLR	control_s, control_ns
Priority Mask	GICC_PMR	priority_mask
Binary Point Register	GICC_BPR	bin_pt_s, bin_pt_ns
Interrupt Acknowledge	GICC_IAR	int_ack
End of Interrupt	GICC_EOIR	EOI
Running Priority	GICC_RPR	run_priority
Aliased Binary Point	GICC_ABPR	alias_bin_pt_ns
Highest Priority Pending Interrupt	GICC_HPPIR	hi_pending
CPU Implementer Identification	GICC_IIDR	cpu_ident

## B.3 Index of architectural names

Table B-5 is an alphabetic index of the GIC register names, indexing the description of each register. An n at the end of a register name, as in `GICC_APRn`, shows that there are multiple instances of the register.

**Table B-5 Index of GIC register names**

Register name	Description
Component IDn	<i>Identification registers on page 4-119</i>
GICC_ABPR	<i>Aliased Binary Point Register, GICC_ABPR on page 4-145</i>
GICC_APRn	<i>Active Priorities Registers, GICC_APRn on page 4-149</i>
GICC_AEOIR	<i>Aliased End of Interrupt Register, GICC_AEOIR on page 4-147</i>
GICC_AIAR	<i>Aliased Interrupt Acknowledge Register, GICC_AIAR on page 4-146</i>
GICC_AHPPIR	<i>Aliased Highest Priority Pending Interrupt Register, GICC_AHPPIR on page 4-148</i>
GICC_BPR	<i>Binary Point Register, GICC_BPR on page 4-133</i>
GICC_CTLR	<i>CPU Interface Control Register, GICC_CTLR on page 4-125</i>
GICC_DIR	<i>Deactivate Interrupt Register, GICC_DIR on page 4-153</i>
GICC_EOIR	<i>End of Interrupt Register, GICC_EOIR on page 4-138</i>
GICC_HPPIR	<i>Highest Priority Pending Interrupt Register, GICC_HPPIR on page 4-143</i>
GICC_IAR	<i>Interrupt Acknowledge Register, GICC_IAR on page 4-135</i>
GICC_IIDR	<i>CPU Interface Identification Register, GICC_IIDR on page 4-152</i>
GICC_NSAPRn	<i>Non-secure Active Priorities Registers, GICC_NSAPRn on page 4-151</i>
GICC_PMR	<i>Interrupt Priority Mask Register, GICC_PMR on page 4-131</i>
GICC_RPR	<i>Running Priority Register, GICC_RPR on page 4-142</i>
GICD_CPENDESGIRn	<i>SGI Clear-Pending Registers, GICD_CPENDESGIRn on page 4-115</i>
GICD_CTLR	<i>Distributor Control Register, GICD_CTLR on page 4-85</i>
GICD_ICACTIVERn	<i>Interrupt Clear-Active Registers, GICD_ICACTIVERn on page 4-103</i>
GICD_ICENABLERn	<i>Interrupt Clear-Enable Registers, GICD_ICENABLERn on page 4-95</i>
GICD_ICFGRn	<i>Interrupt Configuration Registers, GICD_ICFGRn on page 4-109</i>
GICD_ICPENDRn	<i>Interrupt Clear-Pending Registers, GICD_ICPENDRn on page 4-99</i>
GICD_IGROUPRn	<i>Interrupt Group Registers, GICD_IGROUPRn on page 4-91</i>
GICD_IIDR	<i>Distributor Implementer Identification Register, GICD_IIDR on page 4-90</i>
GICD_IPRIORITYRn	<i>Interrupt Priority Registers, GICD_IPRIORITYRn on page 4-104</i>
GICD_ISACTIVERn	<i>Interrupt Set-Active Registers, GICD_ISACTIVERn on page 4-102</i>
GICD_ISENBALERn	<i>Interrupt Set-Enable Registers, GICD_ISENBALERn on page 4-93</i>
GICD_ISPENDRn	<i>Interrupt Set-Pending Registers, GICD_ISPENDRn on page 4-97</i>
GICD_ITARGETSRn	<i>Interrupt Processor Targets Registers, GICD_ITARGETSRn on page 4-106</i>

**Table B-5 Index of GIC register names (continued)**

<b>Register name</b>	<b>Description</b>
GICD_SGIR	<i>Software Generated Interrupt Register, GICD_SGIR on page 4-113</i>
GICD_NSACRn	<i>Non-secure Access Control Registers, GICD_NSACRn on page 4-111</i>
GICD_SPENDSGIRn	<i>SIG Set-Pending Registers, GICD_SPENDSGIRn on page 4-117</i>
GICD_TYPER	<i>Interrupt Controller Type Register, GICD_TYPER on page 4-88</i>
Peripheral IDn	<i>Identification registers on page 4-119</i>
GICH_APR	<i>Active Priorities Register, GICH_APR on page 5-175</i>
GICH_EISRn	<i>End of Interrupt Status Registers, GICH_EISR0 and GICH_EISR1 on page 5-173</i>
GICH_ELRSRn	<i>Empty List Register Status Registers, GICH_ELRSR0 and GICH_ELRSR1 on page 5-174</i>
GICH_HCR	<i>Hypervisor Control Register, GICH_HCR on page 5-168</i>
GICH_LRn	<i>List Registers, GICH_LRn on page 5-176</i>
GICH_MISR	<i>Maintenance Interrupt Status Register, GICH_MISR on page 5-172</i>
GICH_VMCR	<i>Virtual Machine Control Register, GICV_CTLR on page 5-180</i>
GICH_VTR	<i>VGIC Type Register, GICH_VTR on page 5-170</i>
GICV_ABPR	<i>VM Aliased Binary Point Register, GICV_ABPR on page 5-190</i>
GICV_AEOIR	<i>VM Aliased End of Interrupt Register, GICV_AEOIR on page 5-192</i>
GICV_AHPPIR	<i>VM Aliased Highest Priority Pending Interrupt Register, GICV_AHPPIR on page 5-193</i>
GICV_AIAR	<i>VM Aliased Interrupt Acknowledge Register, GICV_AIAR on page 5-191</i>
GICV_APRn	<i>VM Active Priorities Registers, GICV_APRn on page 5-194</i>
GICV_BPR	<i>VM Binary Point Register, GICV_BPR on page 5-184</i>
GICV_CTLR	<i>Virtual Machine Control Register, GICV_CTLR on page 5-180</i>
GICV_EOIR	<i>VM End of Interrupt Register, GICV_EOIR on page 5-187</i>
GICV_HPPIR	<i>VM Highest Priority Pending Interrupt Register, GICV_HPPIR on page 5-189</i>
GICV_IAR	<i>VM Interrupt Acknowledge Register, GICV_IAR on page 5-185</i>
GICV_PMR	<i>VM Priority Mask Register, GICV_PMR on page 5-183</i>
GICV_RPR	<i>VM Running Priority Register, GICV_RPR on page 5-188</i>
GICV_IIDR	<i>VM CPU Interface Identification Register, GICV_IIDR on page 5-195</i>
GICV_DIR	<i>VM Deactivate Interrupt Register, GICV_DIR on page 5-196</i>



# Appendix C

## Revisions

This appendix describes the main technical changes between released issues of this book. There are no technical changes between issue B and issue B.b, see the Status statement in the Release Information at the start of the book.

**Table C-1 Differences between issue A and issue B**

<b>Change</b>	<b>Location</b>
Section updated to describe interrupt grouping functionality	<i>About the Generic Interrupt Controller architecture on page 1-14</i>
Section updated	<i>Changes in version 2.0 of the Specification on page 1-15</i>
Section updated	<i>Security Extensions support on page 1-16</i>
Section added	<i>Virtualization support on page 1-17</i>
Section updated to clarify SGI description and to include virtual interrupts	<i>Interrupt types on page 1-18</i>
Section updated to describe virtual CPU interface	<i>About GIC partitioning on page 2-22</i>
Note added to clarify GICv1 functionality	<i>The Distributor on page 2-24</i>
Section updated to clarify GICv2 CPU interface behavior	<i>CPU interfaces on page 2-26</i>
Section added	<i>Interrupt signal bypass, and GICv2 bypass disable on page 2-27</i>
Section added	<i>Power management, GIC v2 on page 2-31</i>
GICC_CTLR.SBPR bit renamed to GICC_CTLR.CBPR to clarify terminology.	<ul style="list-style-type: none"><li>• Chapter 3 <i>Interrupt Handling and Prioritization</i></li><li>• Chapter 4 <i>Programmers' Model</i></li></ul>

Table C-1 Differences between issue A and issue B (continued)

Change	Location
Section updated to include interrupt grouping functionality	<ul style="list-style-type: none"> <li>• <i>About interrupt handling and prioritization on page 3-34</i></li> <li>• <i>Identifying the supported interrupts on page 3-35</i></li> </ul>
Section updated to clarify EOI behavior and scope.	<i>General handling of interrupts on page 3-37</i>
Section added	<i>Priority drop and interrupt deactivation on page 3-38</i>
Section updated to clarify EOI behavior and of Secure writes to GICD_SGIR	<i>Interrupt handling state machine on page 3-41</i>
Section updated to clarify scope of interrupt grouping functionality	<i>Interrupt prioritization on page 3-44</i>
Section updated to clarify preemption behavior and to include information about priority drop functionality	<i>Preemption on page 3-45</i>
Section updated to clarify functionality	<i>Priority grouping on page 3-45</i>
Added table to clarify priority grouping behavior	<i>Table 3-3 on page 3-46</i>
Section renamed and updated to clarify scope, and to describe interrupt grouping functionality	<i>The effect of interrupt grouping on interrupt handling on page 3-48</i>
Section updated to clarify interrupt handling	<i>The effect of interrupt grouping on interrupt acknowledgement on page 3-50</i>
Section added	<i>GIC power on or reset configuration on page 3-51</i>
Section renamed and updated to clarify scope, and to describe interrupt grouping functionality	<i>Interrupt grouping and interrupt prioritization on page 3-53</i>
Section updated to clarify functionality	<i>The effect of interrupt grouping on priority grouping on page 3-57</i>
Section added	<i>Additional features of the GIC Security Extensions on page 3-59</i>
Section added	<i>Access from processors not implementing the ARM Security Extensions on page 3-59</i>
Pseudocode updated	<i>Pseudocode details of interrupt handling and prioritization on page 3-61</i>
Section added	<i>The effect of the Virtualization Extensions on interrupt handling on page 3-67</i>
Section added	<i>Example GIC usage models on page 3-68</i>
Distributor and CPU interface register map tables updated	<ul style="list-style-type: none"> <li>• <i>Table 4-1 on page 4-75</i></li> <li>• <i>Table 4-2 on page 4-76</i></li> </ul>
Note added to clarify endianness	<i>GIC register access on page 4-77</i>
Section updated to clarify register banking in multiprocessor systems	<i>Register banking on page 4-77</i>
Section added	<i>Enabling and disabling the Distributor and CPU interfaces on page 4-77</i>
Section updated to include interrupt grouping functionality	<i>Effect of the GIC Security Extensions on the programmers' model on page 4-80</i>



Table C-1 Differences between issue A and issue B (continued)

Change	Location
Section updated to describe GICv1 and GICv2 differences and the effect of the Security Extensions	<i>Distributor Control Register, GICD_CTLR</i> on page 4-85
Registers renamed from Interrupt Security Registers, and section updated to clarify scope	<i>Interrupt Group Registers, GICD_IGROUPRn</i> on page 4-91
Sections updated to clarify register descriptions	<ul style="list-style-type: none"> <li><i>Interrupt Set-Enable Registers, GICD_ISENBLEn</i> on page 4-93</li> <li><i>Interrupt Clear-Enable Registers, GICD_ICENABLEn</i> on page 4-95</li> </ul>
Section updated to clarify level-sensitive interrupt information	<i>Interrupt Clear-Pending Registers, GICD_ICPENDRn</i> on page 4-99
Distributor register descriptions added	<ul style="list-style-type: none"> <li><i>Interrupt Set-Active Registers, GICD_ISACTIVERn</i> on page 4-102</li> <li><i>Interrupt Clear-Active Registers, GICD_ICACTIVERn</i> on page 4-103</li> <li><i>Non-secure Access Control Registers, GICD_NSACRn</i> on page 4-111</li> <li><i>SGI Clear-Pending Registers, GICD_CPENDSGIRn</i> on page 4-115</li> <li><i>SGI Set-Pending Registers, GICD_SPENDSGIRn</i> on page 4-117</li> </ul>
Pseudocode added to shows the effects of the GIC Security Extensions on accesses to these registers	<i>Interrupt Priority Registers, GICD_IPRIORITYRn</i> on page 4-104
Section updated to clarify usage constraints and change to bit name.	<i>Software Generated Interrupt Register, GICD_SGIR</i> on page 4-113
Section updated to describe effects of GICv2	<i>Identification registers</i> on page 4-119
Section updated to describe GICv1 and GICv2 registers and the effect of the Security Extensions	<i>CPU Interface Control Register, GICC_CTLR</i> on page 4-125
Pseudocode added to shows the effects of the GIC Security Extensions on accesses to this register	<i>Interrupt Priority Mask Register, GICC_PMR</i> on page 4-131
Section updated to include interrupt grouping functionality	<i>Binary Point Register, GICC_BPR</i> on page 4-133
Section updated to clarify interrupt acknowledgement behavior	<i>Interrupt Acknowledge Register, GICC_IAR</i> on page 4-135
Section updated to clarify EOI behavior	<i>End of Interrupt Register, GICC_EOIR</i> on page 4-138
Section updated to describe effect of Security Extensions	<i>Behavior of writes to GICC_EOIR, GICv1 with Security Extensions</i> on page 4-139
Section added	<i>Behavior of writes to GICC_EOIR, GICv2</i> on page 4-140
Section updated to clarify effect of interrupt grouping functionality	<i>Highest Priority Pending Interrupt Register, GICC_HPPIR</i> on page 4-143
Pseudocode added to show the effects of the GIC Security Extensions on accesses to this register	
Section updated to clarify behavior	<i>Aliased Binary Point Register, GICC_ABPR</i> on page 4-145

Table C-1 Differences between issue A and issue B (continued)

Change	Location
CPU interface register descriptions added	<ul style="list-style-type: none"> <li>• <i>Aliased Interrupt Acknowledge Register, GICC_AIAR</i> on page 4-146</li> <li>• <i>Aliased End of Interrupt Register, GICC_AEOIR</i> on page 4-147</li> <li>• <i>Aliased Highest Priority Pending Interrupt Register, GICC_AHPPIR</i> on page 4-148</li> <li>• <i>Active Priorities Registers, GICC_APRn</i> on page 4-149</li> <li>• <i>Non-secure Active Priorities Registers, GICC_NSAPRn</i> on page 4-151</li> <li>• <i>Deactivate Interrupt Register, GICC_DIR</i> on page 4-153</li> </ul>
Section added	<i>Preserving and restoring GIC state</i> on page 4-155
Chapter added	Chapter 5 <i>GIC Support for Virtualization</i>
Appendix removed <sup>a</sup>	Appendix B <i>Software Examples for the GIC</i>
Section added	<i>Alternative register names</i> on page B-202
Section updated to include GICv2 registers	<i>Index of architectural names</i> on page B-204

a. This content is outside the scope of the Architecture Specification.

# Glossary

**Activate** An interrupt is activated when its state changes either:

- from pending to active
- from pending to active and pending.

For more information see [Interrupt handling state machine on page 3-41](#).

**Banked interrupt**

In a multiprocessor implementation, a banked interrupt is one of multiple PPIs or SGIs that have the same interrupt ID, but target different connected processors and have independent states corresponding to each connected processor.

**Banked register**

A register that has multiple instances. A property of the state of the device determines which instance is in use. For more information about register banking in the GIC see [Register banking on page 4-77](#).

**Deactivate**

An interrupt is deactivated when its state changes either:

- from active to inactive
- from active and pending to pending.

For more information see [Interrupt handling state machine on page 3-41](#).

**Idle priority**

The lowest possible priority that can be assigned to an interrupt. In an implementation that supports eight-bit priority fields, the priority value of the idle priority is 0xFF. Otherwise, it is either the largest value with which a RW [GICD\\_IPRIORITYRn.Priority](#) field can be programmed, or 0xFF.

**IMP**

Is an abbreviation used in diagrams to indicate that the bit or bits concerned have IMPLEMENTATION DEFINED behavior.

**IMPLEMENTATION DEFINED**

Means that the behavior is not architecturally defined, but should be defined and documented by individual implementations.

**IMPLEMENTATION SPECIFIC**

Means that the behavior is not architecturally defined, and does not have to be documented by individual

implementations. Used when there are a number of implementation options available and the option chosen does not affect software compatibility.

### Interrupt grouping

The configuration of interrupts as either Group 0 or Group 1. One use of interrupt grouping is to manage Secure and Non-secure interrupts, using Group 0 for Secure interrupts and Group 1 for Non-secure interrupts.

### Local access

A local access to a particular GIC is an access from a processor with a CPU interface on that GIC. Remote and local access is permitted to SPIs, but SGIs only support local access. See also [Remote access](#).

### Observer

A processor or mechanism within the system, such as peripheral device, that is capable of generating reads from or writes to memory.

### Peripheral interrupt

An interrupt generated by the assertion of an interrupt request signal input to the GIC. The GIC architecture defines the following types of peripheral interrupt:

#### Private Peripheral Interrupt (PPI)

A peripheral interrupt that is specific to a single processor.

#### Shared Peripheral Interrupt (SPI)

A peripheral interrupt that the Distributor can route to a combination of processors, as specified by the corresponding `GICD_ITARGETSRn` register.

### PPI

See Peripheral Interrupt

### Preemption level

A preemption level is a supported group priority. For more information, see [Priority grouping on page 3-45](#).

### Priority drop

Priority drop is when the running priority of a CPU interface is set to the priority of the most recently acknowledged active interrupt, that has not been subject to an EOI request, but the interrupt remains active.

See also [Running priority](#).

### RAO

See Read-As-One.

### RAO/WI

Read-As-One, Writes Ignored. In any implementation, the bit must read as 1, or all 1s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 1, or all 1s for a bit field, and on writes being ignored.

### RAZ

See Read-As-Zero.

### RAZ/WI

Read-As-Zero, Writes Ignored. In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 0, or all 0s for a bit field, and on writes being ignored.

### Read-As-One (RAO)

In any implementation, the bit must read as 1, or all 1s for a bit field.

### Read-As-Zero (RAZ)

In any implementation, the bit must read as 0, or all 0s for a bit field.

### Remote access

A remote access to a particular GIC is an access from a processor without a CPU interface on that GIC. Remote and local access is permitted to SPIs, but SGIs only support local access. See also [Local access](#).

### Reserved

Registers that are reserved are RAZ/WI unless otherwise stated. Bit positions described as Reserved are UNK/SBZP.

### Running priority

The running priority of a CPU interface is either:

- the group priority of the highest priority active interrupt, on that interface, for which there has not been a valid write to an end of interrupt register

- if there is no active interrupt on the interface for which there has not been a valid write to an end of interrupt register, the running priority is the idle priority.

*See also* [Idle priority](#), [Priority drop and interrupt deactivation](#) on page 3-38.

**SBZ** *See* Should-Be-Zero.

**SBZP** *See* Should-Be-Zero-or-Preserved.

**Security hole** Is a mechanism that bypasses system protection.

**SGI** *See* Software-generated interrupt.

**Should-Be-Zero (SBZ)**

Should be written as 0 (or all 0s for a bit field) by software. Values other than 0 produce UNPREDICTABLE results.

**Should-Be-Zero-or-Preserved (SBZP)**

Must be written as 0, or all 0s for a bit field, by software if the value is being written without having been previously read, or if the register has not been initialized. Where the register was previously read on the same processor, since the processor was last reset, the value in the field should be preserved by writing the value that was previously read.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 0 (or all 0s for a bit field), nor a value previously read for the same field on the same processor, the result is UNPREDICTABLE.

**Software-generated interrupt (SGI)**

An interrupt generated by the GIC in response to software writing to a GIC register. In a multiprocessor implementation, an SGI is identified by the combination of its interrupt ID and the CPU ID of the processor that wrote to the GIC to generate the interrupt.

**SPI** *See* Peripheral Interrupt

**Spurious interrupt**

An interrupt that does not require servicing. Usually, refers to an interrupt ID returned by a GIC to a request from a connected processor. Returning a spurious interrupt ID indicates that there is no pending interrupt on the CPU interface that the requesting processor can service. For example, if a level-sensitive interrupt request signal to the GIC causes a CPU interface to signal an interrupt request to a processor, but by the time the processor reads the [GICC\\_IAR](#) to acknowledge the interrupt the request signal has been deasserted, the GIC returns a spurious interrupt ID of 1023, to indicate that there is no interrupt request to service.

**Sufficient priority**

To determine whether to signal an interrupt to its connected processor, a GIC CPU interface must determine whether the interrupt has *sufficient priority* to be signaled to the connected processor. It does this by comparing the interrupt priority with all of:

- the Priority Mask Register, [GICC\\_PMR](#)
- the preemption settings for the interface, as shown by [GICC\\_BPR](#) or [GICC\\_ABPR](#)
- the current running priority for the CPU interface.

If the interrupt has sufficient priority then an interrupt request is signaled to the connected processor.

*See also* [Running priority](#).

**UNK** Software must treat a field as containing an UNKNOWN value. In any implementation, the bit must read as 0, or all 0s for a bit field. Software must not rely on the field reading as zero.

**UNKNOWN** An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not be a security hole. UNKNOWN values must not be documented or promoted as having a defined value or effect.

**UNK/SBZP** UNKNOWN on reads, Should-Be-Zero-or-Preserved on writes.

In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the field must be ignored.

Software must not rely on the field reading as 0, or all 0s for a bit field, and must use an SBZP policy to write to the field.

## **UNPREDICTABLE**

The behavior cannot be relied upon. UNPREDICTABLE behavior must not represent security holes. UNPREDICTABLE behavior must not halt or hang the processor, or any parts of the system. UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

## **Valid interrupt ID**

An interrupt ID, as returned by a read of [GICC\\_IAR](#) or [GICC\\_AIAR](#), that is not a spurious interrupt ID. This means it is an interrupt ID with a value of 1019 or less. If the interrupt is an SGI, then unless the context indicates otherwise, the valid interrupt ID includes the associated CPUID.