GIC Architecture Specification

Processor Division

Document number:	PRD03-GENC-010745 16.0
Date of Issue:	1 November 2013
Author:	Tony Jebson
Authorized by:	

Abstract

Keywords

Reviewer list

Name	Function	Name	Function

Contents

Confidential Proprietary Notice

This document is **CONFIDENTIAL** and any use by you is subject to the terms of the agreement between you and ARM or the terms of the agreement between you and the party authorised by ARM to disclose this document to you.

This document is protected by copyright and other related rights and the practice or implementation of the information contained in this document may be protected by one or more patents or pending patent applications. No part of this document may be reproduced in any form by any means without the express prior written permission of ARM. **No license, express or implied, by estoppel or otherwise to any intellectual property rights is granted by this document unless specifically stated.**

Your access to the information in this document is conditional upon your acceptance that you will not use or permit others to use the information: **(i)** for the purposes of determining whether implementations infringe any third party patents; **(ii)** for developing technology or products which avoid any of ARM's intellectual property; or **(iii)** as a reference for modifying existing patents or patent applications or creating any continuation, continuation in part, or extension of existing patents or patent applications; or **(iv)** for generating data for publication or disclosure to third parties, which compares the performance or functionality of the ARM technology described in this document with any other products created by you or a third party, without obtaining ARM's prior written consent.

THIS DOCUMENT IS PROVIDED "AS IS". ARM PROVIDES NO REPRESENTATIONS AND NO WARRANTIES, EXPRESS, IMPLIED OR STATUTORY, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, SATISFACTORY QUALITY, NON-INFRINGEMENT OR FITNESS FOR A PARTICULAR PURPOSE WITH RESPECT TO THE DOCUMENT. For the avoidance of doubt, ARM makes no representation with respect to, and has undertaken no analysis to identify or understand the scope and content of, third party patents, copyrights, trade secrets, or other rights.

This document may include technical inaccuracies or typographical errors.

TO THE EXTENT NOT PROHIBITED BY LAW, IN NO EVENT WILL ARM BE LIABLE FOR ANY DAMAGES, INCLUDING WITHOUT LIMITATION ANY DIRECT, INDIRECT, SPECIAL, INCIDENTAL, PUNITIVE, OR CONSEQUENTIAL DAMAGES, HOWEVER CAUSED AND REGARDLESS OF THE THEORY OF LIABILITY, ARISING OUT OF ANY USE OF THIS DOCUMENT, EVEN IF ARM HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

This document consists solely of commercial items. You shall be responsible for ensuring that any use, duplication or disclosure of this document complies fully with any relevant export laws and regulations to assure that this document or any portion thereof is not exported, directly or indirectly, in violation of such export laws. Use of the word "partner" in reference to ARM's customers is not intended to create or refer to any partnership relationship with any other company. ARM may make changes to this document at any time and without notice.

If any of the provisions contained in these terms conflict with any of the provisions of any signed written agreement or click through End User Agreement covering this document with ARM, then the signed written agreement or End User Agreement prevails over and supersedes the conflicting provisions of these terms.

1 ABOUT THIS DOCUMENT

1.1 Change control

1.1.1 Current status and anticipated changes

Draft for review.

1.1.2 Change history

The change history for this document is managed by Domino.

1.2 References

This document refers to the following documents.

Ref	Doc No	Author(s)	Title
[1]	ARM DDI 0406C		ARM Architecture Reference Manual
[2]	ARM IHI 0048B		GIC Architecture Specification v2
[3]	ARM IHI 0051A		AMBA 4 AXI4 Stream Protocol v1.0
[4]	PRD03-GENC-0094324		ARMv8 Exception Model

1.3 Terms and abbreviations

This document uses the following terms and abbreviations.

Term	Meaning
Affinity	A term describing the locality relationship between two processors.
Affinity Hierarchy	A four-level tree structure that defines the locality of processors in a system.
Affinity Routing	A method of distributing interrupts based on the locality of the required destination processors.
ARE	Affinity Routing Enable. A register field that controls whether affinity routing is used for a security state.
Cluster	A group of processors that are in some way “local” to each other.
CPU interface	That part of the GIC responsible for the delivery of interrupt exceptions to a processor. The CPU interface comprises the physical interface (that delivers FIQ, IRQ and SError exceptions), the virtual interface (that delivers virtual FIQ, virtual IRQ and virtual SError exceptions) and the control interface (that configures the CPU interface and allows software to generate virtual interrupts). The CPU interface might provide memory-mapped and system register access to the physical, virtual or control interfaces.
CPU interface registers	The set of GIC registers associated with the CPU interface logic. Includes all the memory-mapped registers (GICC_*, GICV_* and GICH_*) and all the system registers (ICC_* and ICH_*).
DIR	Deactivate Interrupt Register. A register within the GIC that is written by software to clear the active status of the indicated interrupt. Writes to DIR are only allowed when EOI mode is set.
Direct virtual interrupt	An interrupt sent directly to a guest operating system without hypervisor intervention. ***Note***: this feature is only supported in GICv4.
Distributor	That part of the GIC responsible for the distribution of interrupts to target processors. The Distributor comprises the complete set of re-distributors for an Affinity Hierarchy.
EOI mode	This indicates whether handling of an interrupt by the processor is completed within a single interrupt service routine (ISR) or whether handling of the interrupt is split between an ISR and an interrupt service task (IST). When EOI mode is clear, an interrupt is handled solely in an ISR and writes to EOIR result clear the active status of an interrupt and no write to DIR is required. When EOI mode is set, interrupt handling is split between an ISR and an IST. The ISR software writes to EOIR to perform a “priority drop” which ensures the ISR is not re-entered for the same interrupt and the IST performs a write to DIR to clear the active status of the interrupt.
EOIR	End of Interrupt Register. A register within the GIC that is written by software to indicate that processing of the indicated interrupt has been completed. When EOI mode is clear, an interrupt is handled solely in an ISR and writes to EOIR result clear the active status of an interrupt and no write to DIR is required. When EOI mode is set, interrupt handling is split between an ISR and an IST. The ISR software writes to EOIR to perform a “priority drop” which ensures the ISR is not re-entered for the same interrupt and the IST performs a write to DIR to clear

	the active status of the interrupt.
FIQ	A processor interrupt exception normally used for to communicate Group 0 interrupts. When the processor is in a different security state to the target security state of a Group 1 interrupt, the interrupt is “promoted” and communicated using FIQ.
IAR	Interrupt Acknowledge Register. A register within the GIC that is read by software to indicate receipt of an interrupt by the processor. Reading this register normally clears the pending state and returns the identifier of the interrupt that has been received by the processor.
Interrupt Group	A collection of interrupts of differing priorities that are always handled by the same Interrupt Regime.
Interrupt Regime	A combination of the processor's current security state and exception level.
IRQ	A processor interrupt exception normally used to communicate Group 1 interrupts to the processor. When the processor is in a different security state to the target security state of a Group 1 interrupt, the interrupt is “promoted” and communicated using FIQ.
ISR	Interrupt Service Routine. Software invoked directly as a consequence of an interrupt exception. Generally, such software acknowledges interrupts and performs an end-of-interrupt operation.
IST	Interrupt Service Task. Software invoked indirectly when the processor takes an interrupt exception. Generally, such software is scheduled during an ISR and might perform an interrupt deactivation operation.
ITE	Interrupt Translation Table Entry.
ITS	Interrupt Translation Service. A hardware service that provides software with mechanisms to re-direct interrupts to different target processors and to re-map interrupt IDs without needing to understand the address map and how each peripheral device is addressed.
Locality	A group of processors that are in some way “local” to each other.
LPI	Locality-specific Peripheral Interrupt. A message based interrupt that is distributed to a specific processor within the affinity hierarchy.
Message based interrupt	An interrupt that is set pending by a peripheral device performing a write to a memory location rather than by assertion of a dedicated signal.
PPI	Private Peripheral Interrupt.
Processor Number	An integer value that uniquely identifies each processor within the system. There is an implementation defined mapping between this and the affinity path of the processor. This mapping may be discovered from GICR_TYPER in each affinity level 0 re-distributor. When ARE is zero for a security state, only processor numbers in the range zero to seven may be used and these numbers map onto a bit position within the GICD_ITARGETSRn register that corresponds to the processors that support GICv2 operation. The processor number is also used in ITS commands in monolithic implementations.

Re-Distributor	Logic corresponding to a node within the Affinity Hierarchy. Message based interrupts can target individual re-distributors to set LPIs as pending.
SEI	System Error Interrupt.
SGI	Software Generated Interrupt.
SPI	Shared Peripheral Interrupt. A wired interrupt from a peripheral device that connects to the top-level re-distributor. Each SPI has associated group, priority and routing information.
SRE	System Register Enable. A register field that controls whether system register access is enabled for an interrupt regime.
Top-Level Re-Distributor	The re-distributor at the apex of the Affinity Hierarchy. This top-level re-distributor generally corresponds to the GICv2 Distributor. All wired SPI interrupts are connected to the top-level re-distributor.
Virtual Access	An access to a register at non-secure EL1 where the interrupt group associated with that register has been virtualized (i.e. the corresponding HCR_EL2.{FMO, IMO, AMO} bit is one). For “common” registers (see section 4.6.6) and access is virtual if either HCR_EL2.FMO or HCR_EL2.IMO is one.
Wired interrupt	An interrupt set pending by a peripheral by asserting a dedicated signal.

2 SCOPE

This document specifies two GIC architecture versions beyond the existing GICv2 specification [2]. The versions specified are:

- GICv3. This version introduces affinity routing to support systems containing more than eight processors, interrupt translation for physical interrupts, system register access to the CPU interface registers, support for reporting system error interrupts, and better support for both interrupt scalability and distributed placement.
- GICv4. This version introduces support for direct injection of guest interrupts.

3 INTRODUCTION

The existing GICv2 specification [2] restricts the number of processors supported to a maximum of eight, is detached from the processor and has limited scalability as system size increases. This document addresses these limitations.

3.1 New Concepts in GICv3

GICv3 introduces the following concepts:

- The ability to route interrupts according to an affinity hierarchy (see section 4.2.1) that in some way matches “locality” within an implementation. This allows the GIC to support routing of interrupts to more than 8 processors (see the “>8 CPUs” column in Table 1: GIC versions and features below).
- Support for “message based interrupts”. That is, the ability to set interrupts pending by writing an interrupt ID to an address within the GIC (see the “Msg” column in Table 1: GIC versions and features below).
- Support for system register access (see section 4.4.1) to the GIC physical and virtual CPU interfaces and hypervisor control registers. Access is separately enabled for each physical interrupt regime (a combination of security state and exception level; see section 4.4.2). See the “System Register” column in Table 1: GIC versions and features below.
- Support for generation of System Error Interrupts (signaled using the SError exception with ARMv8 processors[4]; see the “SEI” column in Table 1: GIC versions and features below).
- Support for generation, distribution and forwarding of a new class of Locality-specific Peripheral Interrupts (see section 4.8) that provide better scalability of interrupts than SPIs (see the “Scalability” column in Table 1: GIC versions and features below).
- Support for generation, distribution and forwarding of Secure Group 1 interrupts (see section 4.1.3) to allow interrupts to directly target secure software operating at EL1 (see the “Groups” column in Table 1: GIC versions and features below).
- The ability to fully re-direct individual interrupts using an Interrupt Translation Service (see section 4.9) that can re-direct interrupts to different destination processors (see the “Translation” column in Table 1: GIC versions and features below).

GICv4 introduces the following concepts:

- The ability to forward direct virtual interrupts (see section 4.9.3) to a guest operating system without hypervisor intervention (see the “Direct” column in Table 1: GIC versions and features below).

3.2 GIC Version Applicability

The table below defines which GIC versions may be used with a processor architecture and which features are provided with each version:

GIC Version	Processor versions	CPU interface mapping		Supported features						
		Memory	System Register	>8 CPUs	Interrupt Features					
					SEI	Msg	Scalable	Translation	Direct	Groups
GICv2	ARMv7	Y	N	N	N	N	N	N	N	2
	ARMv8	Y	N	N	N	N	N	N	N	2
GICv3	ARMv8	*	Y	Y	Y	Y	Y	Y	N	3
GICv4	ARMv8	*	Y	Y	Y	Y	Y	Y	Y	3

Table 1: GIC versions and features

Note *: optional for GICv2 compatibility only. GICv3 implementations might not provide memory-mapped access to the CPU interface registers when it is known backwards compatibility is not required.

3.3 GICv3 Backwards Compatibility

For systems requiring GICv2 backwards compatibility, GICv3 will provide full backward compatibility, except the ability for software on one processor to access the virtual interface control registers of another processor (see section 5.2.7 of ref [2]; use of this feature in GICv2 is deprecated). ***Note***: see section 3.3.1 for additional restrictions on compatibility with legacy configurations including the removal of deprecated features such as AckCtl.

To support backward compatibility, the existing memory mapped interfaces to the Distributor registers, the CPU interface registers, the virtual CPU interface registers and the Hypervisor control registers will be provided, and these will be GICv2 compatible.

In implementations that support two security states, GICv3 features can be separately enabled for the secure state and the non-secure state. To enable these additional features for a security state, an “Affinity Routing Enable” (ARE) bit must be set in the Distributor control register for that security state. When ARE is not set, the Distributor follows the programming model of the GICv2 architecture, described in [2].

In systems with less than nine processors, the GIC will be able to route interrupts for a security state to any processor in the system when ARE is zero for that security state.

In a system with more than eight processors, interrupts for a security state can only be routed to a subset of the processors comprising eight or fewer processors when ARE is zero for that security state. This subset of processors to which interrupts may be routed is implementation defined.

In a system supporting two security states, ARE may be set independently for each security state, allowing interrupts for that security state to be routed to any processor in the system and providing that security state with access to the full set of GICv3 features.

Note: operation with ARE set to zero for a security state is deprecated in GICv3

Note: in GICv3 systems with more than eight processors that support two security states, it is expected that full backward compatibility with GICv2 might only be required for the secure state and that backward compatibility for the non-secure state might only be required for Guest Operating Systems.

To support systems that do not require support for operation with ARE set to zero for a security state, in some GICv3 implementations, ARE might be RAO/WI for a security state:

- To reduce validation complexity, if ARE for the secure state is RAO/WI, ARE for the non-secure state must also be RAO/WI.

When ARE is set for a security state, the CPU interface for physical interrupts handled by the Distributor is mapped into system register space. As a result, in systems where ARE is tied to one for both security states, there can be no memory-mapped physical CPU interface:

- A memory mapped interface to the virtual CPU interface registers might be provided for virtual interrupt regimes.
- The CPU interface registers that control physical interrupts will always be system register mapped
- The Hypervisor control registers will always be system register mapped
- The Distributor registers remain memory mapped but non-secure interrupts must use affinity routing.

3.3.1 Unsupported Legacy Configurations

The following legacy GICv1 / GICv2 configurations are not supported by GICv3:

- When GICC_CTLR.AckCtl == 1. This is DEPRECATED in GICv2 and is obsolete in GICv3. ***Note***: for virtualization, GICV_CTLR.AckCtl == 1 is still supported.
- Secure management of NS interrupts when ARE is set for one security state but not the other.
- Use of Locked SPIs.
- Use of CFGSDISABLE.

3.4 GICv3 System Register Access Rationale

This section outlines some of the reasons for providing system register access to GIC registers.

3.4.1 OS usage

Some OS interest has been shown in a simple mechanism to disable some groups of interrupts but still allow handling of higher priority groups of interrupts. This can be used to compensate for the lack of an FIQ at EL1 in systems where FIQ is used as a secure interrupt.

The GIC priority scheme provides exactly this capability but adoption in some operating systems is made harder when the priority mechanism is memory mapped.

Generally, the handling a memory-mapped device is performed in a device driver layer. However, the GIC must be handled by the kernel. So, system register access will help the adoption process by:

- Removing the need to discover the address of the (memory mapped) GIC registers
- Removing by construction the possibility of aborts when accessing GIC registers.

Additionally, there is a perception that memory-mapped access is slower, less predictable and less reliable than system register access. This perception has significantly increased resistance to the adoption of the GIC priority scheme.

3.4.2 Locality

Interrupts for a processor are local to that processor and the interface for handling such interrupts is best considered as a co-processor interface. As such, it is preferable that this interface is provided by system registers rather than being memory-mapped.

3.4.3 Scalability

To make software generated interrupts scalable to large systems, it is preferable that the mechanism for generating an SGI is local to the generating processor, preventing large round-trip latencies to any central Distributor resource and allowing easy virtualization of SGI requests.

System registers provide an obvious mechanism to do this and avoid the need to move SGI generation registers to a separate page of memory so they can be separately trapped for virtualization.

3.4.4 Simplicity

To provide adequate performance, implementations using the existing GICv2 memory-mapped interface must intercept memory transactions to the physical addresses allocated to the GIC before they leave the processor. This increases hardware complexity and makes it harder to achieve frequency goals.

System register access falls within the existing instruction decode scheme and is simpler.

Similarly, system register access avoids the need for memory address discovery, simplifying software.

3.5 Support for System Error Interrupts

GICv3 systems that support the system register interface to the GIC also include support for reporting System Error Interrupts (both physical and virtual) using the SError exception[4].

This simplifies the processor IP boundary and provides a single mechanism for reporting all classes of external interrupt.

3.6 IP Roadmap

This section outlines the features provided by each GIC version and the features that ARMv8 implementations must provide in order to support GICv3.

3.6.1 GICv3 Features

GICv3 supports the following features:

- Support for more than eight processors. That is, support for affinity routing is required.
- Message based interrupts. See Section 4.1.5.
- System Error Interrupts. See Section 4.7.
- Secure Group 1 interrupts (when SRE is set for Secure EL1).
- System Register access. See Section 4.4.1.
- Locality-specific Peripheral Interrupts. See Section 4.8.
- Interrupt Translation Service. See Section 4.9.

3.6.2 GICv4 Features

GICv4 supports the following features in addition to those in GICV3 above:

- Direct Virtual Interrupts. See Section 4.9.3.

3.6.3 ARMv8 features required to support GICv3

ARMv8 processors that require GICv3 support must implement the following features:

- System Register access. See Section 4.4.1.
- System Error Interrupts. See Section 4.7.

Note: In many systems it is useful to be able to drive IRQ and FIQ from other sources when the CPU Interface is disabled, and this is supported by the GICv3 architecture.

3.6.4 GIC Capability Discovery

Support for such GIC features is discoverable using GICD_TYPER. ***Note***: System Register access is a property of the processor and is discoverable from the processor identification registers (see section 5.7.31).

4 ARCHITECURE CONCEPTS

4.1 Communication of Interrupts to the GIC

In most systems, interrupts are used to communicate the occurrence of events to software. Such events can either be:

- Generated by software, or
- Generated by a peripheral device

To accelerate handling of interrupts, the GIC architecture associates an interrupt identifier with each distinct interrupt event. This identifier can be used by software to quickly identify the event that has occurred.

Because the software that is the target for a given event might run on any processor within a system, software expects the interrupt identifier for each distinct event to be invariant, regardless of the processor on which the software is running.

This requirement for interrupt identifier invariance requires GICv3 to introduce an Interrupt Translation Service (ITS) that manages the delivery of interrupts to processors as the software targets are moved and ensures that the identifier received by software for a given event is invariant.

4.1.1 Position of the GIC within a System

The diagram below illustrates the position of the GIC within the system.

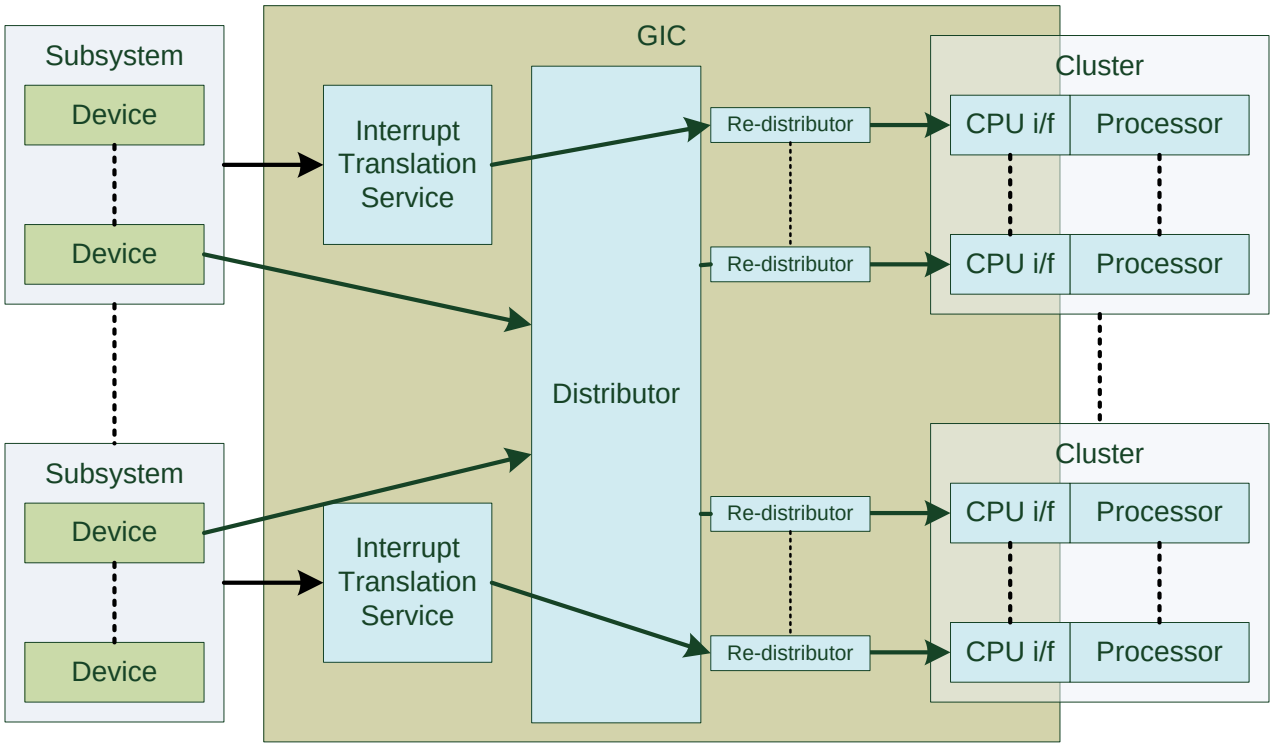


Figure 1: The position of the GIC within the system

Where:

- Interrupt requests from devices may be communicated directly to the Distributor. This is generally for interrupts that require distribution to any one processor in the system.

- Interrupt requests from devices may be communicated to an Interrupt Translation Service (ITS) which will communicate one or more interrupts to the appropriate re-distributors. This is generally for interrupts for a single software entity that is resident on a single processor but that processor might change dynamically such that the routing changes.

The typical flow of an interrupt communicated directly to the Distributor is:

- A device requests an interrupt and communicates this to the Distributor
- The Distributor forwards the interrupt to the appropriate re-distributor
- The re-distributor communicates this interrupt to the CPU interface for signaling to the processor
- The CPU interface generates the appropriate interrupt exception on the processor
- The processor takes the exception and software handles the interrupt, which includes further interaction with the CPU interface.

The typical flow of an interrupt communicated to the ITS is:

- A device requests an interrupt and communicates this to the ITS
- The ITS retrieves details of the interrupts to be generated for this request and where they are to be forwarded
- The ITS forwards the generated interrupts to the appropriate re-distributor
- The re-distributor communicates this interrupt to the CPU interface for signaling to the processor
- The CPU interface generates the appropriate interrupt exception on the processor
- The processor takes the exception and software handles the interrupt, which includes further interaction with the CPU interface.

4.1.2 Support for Two Security States

In GICv3, there are two primary components in a GIC implementation concerned with security: the Distributor, and the CPU interface. GICv3 allows each to independently provide support for a single security state or two security states.

- A CPU interface must support two security states whenever its associated processor implements EL3. ***Note***: a processor might implement EL3 but it may be “unused”. See section 4.6.9.
- A Distributor indicates support for two security states using a single “Single Security State” bit.
- This “Single Security State” bit is provided in the Distributor control register as GICD_CTLR.DS. An implementation might choose to make this bit read-only or programmable. When programmable, the bit may only be modified by secure accesses to GICD_CTLR:
    - When GICD_CTLR.DS is zero, two security states are supported.
    - When GICD_CTLR.DS is one, a single security state is supported.
    - See section 5.3.20 for the full definition of this bit, and see section 4.6.11 for other effects when this bit is one.

Because support for two security states is provided independently by the Distributor and each CPU interface, GICv3 supports the following types of system implementation:

- Systems where the Distributor and all processors support two security states
- Systems where the Distributor and all processors support a single security state. Such systems might be always in the secure state or always in the non-secure state.
- Systems where the Distributor supports two security states and processors support a mixture of one and two security states. The supported configurations and how they must be managed are specified in sections 4.6.9 and 4.6.10.

4.1.3 Interrupt Grouping and Security

In ARMv8 when EL3 is using AArch64 there is a separation between Secure EL1 and Secure EL3. For this reason, GICv3 provides the ability to distinguish between interrupts that should be handled at Non-Secure EL1/2, Secure EL1 and Secure EL3. In GICv3, the interrupt group is modified by an extra bit to provide this additional information as shown in the table below:

GRPMOD	GROUP	Definition	Short name
0	0	Secure Group 0	G0S
0	1	Non-Secure Group 1	G1NS
1	0	Secure Group 1	G1S
1	1	Reserved (treated as Non-Secure Group 1)	-

Table 2: Interrupt Group and Modifier

Note: this has been done by provision of a “group modifier” register for backwards compatibility.

In systems that support a single security state, there is no security distinction between Secure and Non-Secure interrupt groups. That is, “Secure” and “Non-Secure” interrupts are always accessible. ***Note***: such a system still provides GRPMOD information but Secure Group 1 interrupts will be treated as Secure Group 0 (see below).

Secure Group 1 interrupts will be treated by the CPU interfaces as Secure Group 0 if:

- The CPU does not implement EL3. ***Note***: a processor that does not implement EL3 might be connected to a Distributor that supports two security states (and includes GICD_IGRPMODRn).
- When the secure copy of ICC_SRE_EL1.SRE is zero or when GICD_CTLR.DS is one, the CPU interface will not receive Secure Group 1 interrupts if software has configured the Distributor correctly. However, to prevent unpredictable behaviour if the Distributor is incorrectly configured, the CPU interface should still treat Secure Group 1 interrupts as Secure Group 0 for these conditions. ***Note***: in implementations that support local generation of system errors, an implementation might generate an SEI (see section 4.7).

Secure Group 1 interrupts will be treated by the Distributor as Secure Group 0 if:

- Distributor security has been disabled (GICD_CTLR.DS == ‘1’. See section 5.3.20). ***Note***: the Distributor must never send a Secure Group 1 interrupt to a processor when security has been disabled.
- ARE is not set to one for the secure state. ***Note***: this covers the case where the CPU does not have SRE set to one for the Secure EL1 interrupt regime.

Note: GICv2 does not include a GRPMOD register and the value of GRPMOD is always considered to be zero when ARE is not set for the secure state.

4.1.4 Interrupt Categories and Management

GICv3 supports four categories of interrupt. Each category is allocated a dedicated range of interrupt identifiers as shown in the table below:

Interrupt Identifier	Full Name	Short name	Features supported		Programmability	
			Level	Translation	Always	When ARE is one
0 to 15	Software Generated Interrupt	SGI	N	N	G0S, G1NS	G1S
16 to 31	Private Peripheral Interrupt	PPI	Y	N	G0S, G1NS	G1S
32 to 1019	Shared Peripheral Interrupt	SPI	Y	N	G0S, G1NS	G1S
1020 to 1023	Special purpose identifiers	-	-	-	-	-
1024 to 8191	Reserved	-	-	-	-	-
8192 and greater	Locality-specific Peripheral Interrupt	LPI	N*	Y	-	G1NS

Table 3: Interrupt Identifiers and Categories

Where:

- As GICv2, 16 SGIs are provided per processor. SGIs may be generated directly by processors.
- As GICv2, 16 PPIs are provided per processor. PPIs may be generated directly by a peripheral device.
- As GICv2, SPIs are “global” and each is routed to a single processor. SPIs may be generated directly by a peripheral device.
- LPIs are new in GICv3, are ***always*** Non-Secure Group 1, and may be generated directly by a peripheral device or as a result of an interrupt translation. See section 4.8.
- “*” indicates that LPIs do not support the active state and never require a write to DIR. See section 4.3.2.

4.1.5 Message Based Interrupts

GICv3 supports message based interrupts for all peripherals for both edge and level triggered interrupts. This can be used for any peripheral that is capable of generating a memory message to signal an interrupt and is not specific to PCIe. This enables not just PCIe support but can be used to reduce the number of wires needed to signal interrupts for other peripherals.

4.1.6 PCIe Support

PCIe devices may support one of three different interrupt types:

- Legacy INT A/B/C/D. These are level based interrupts but are implemented as writes to set and clear addresses.
- MSI. These (edge triggered) interrupts provide devices with the ability to vector interrupts. A 16 bit value and a 32/64 bit address are programmed into each device. When the device generates an interrupt, it writes the 16 bit value (with the bottom 5 bits modified by a vector) to the address. The final 16 bit value identifies the source of the interrupt.
- MSI-X. These (edge triggered) interrupts remove a limitation of MSI. Instead of a single address / value pair per device, the device holds a table of address / value pairs where the size of the values is increased to 32 bits. This table is indexed by the device vector. This allows more than 32 vectors per device and allows interrupts to be written to different places.

The intent of MSI and MSI-X is that the reason for the interrupt can be fully identified without additional accesses to the peripheral, allowing for faster interrupt handling. This means the vector information encoded in the message must be transformed efficiently into an interrupt number.

Hence, to properly support PCIe, the GIC must support message based interrupts with a data payload indicating the interrupt number to be used (that isn't 1-hot encoded). This feature is introduced in GICv3.

PCIe systems also expect a large number of possible interrupt targets. For example, a PCIe device may need to generate 128 or more different interrupt vectors per processor. As the number of processors increases, this quickly exceeds the number of SPIs supported by the GIC. GICv3 adds a new class of Locality-specific Peripheral Interrupt (see Section 4.8 below) to overcome this limitation and which can support very large numbers of interrupts in a more scalable manner.

The diagram below shows an overview of a typical PCIe system.

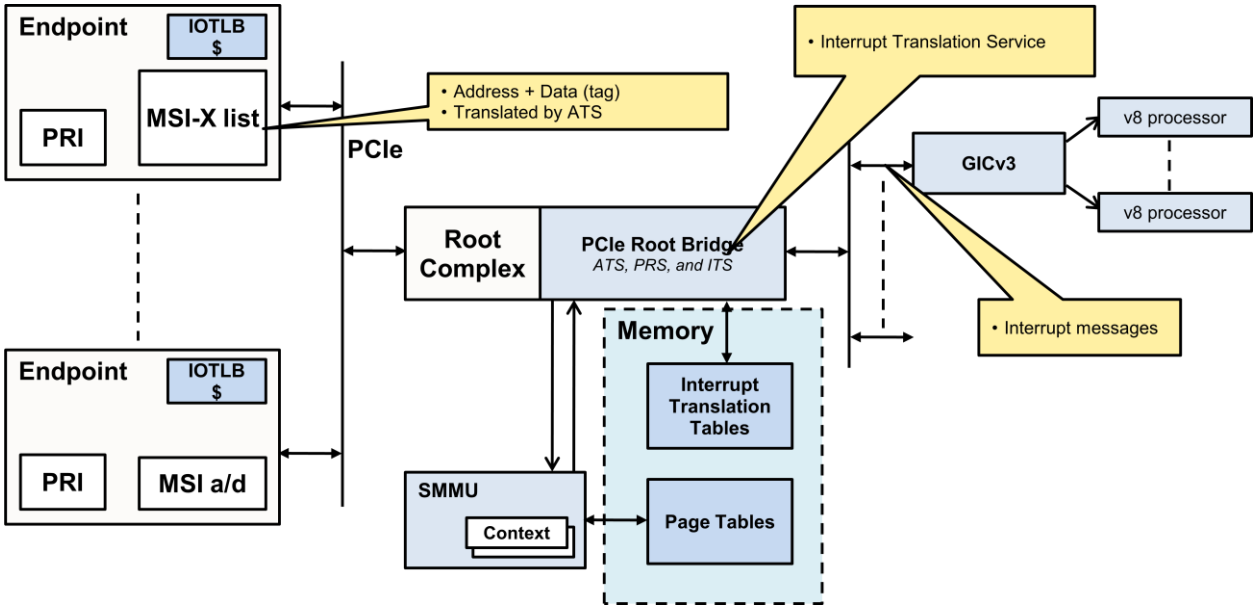


Figure 2: An example GIC based PCIe system

4.1.7 Peripheral Registers for Message Based Interrupts

The GICv3 support for message based interrupts registers provides an interrupt vector communicated as the data payload of a write message. Peripherals using message based interrupts should make address and data values used programmable by system software.

For new ARM peripherals to support message based interrupts, they must:

- Provide at least one address register capable of generating any 32 bit aligned address within the full physical address range of the system for which peripheral is intended.
- Provide at least one data register capable of holding an interrupt number. ***Note***: this register must implement sufficient bits to hold the maximum interrupt identifier supported by the GIC (see the "IDbits" field in sections 5.3.1 and 5.12.6)

Peripherals might provide more than this minimum capability:

- A peripheral may choose to modify the data register in a manner analogous to MSI, or
- A peripheral may choose to provide a table of address and data registers in a manner analogous to MSI-X, or
- A peripheral may choose to provide some different combination of address and data registers
- Regardless of the mechanism chosen, a separate enable must be provided for each unique interrupt number that can be generated.

4.1.8 Existing Peripherals and Message Based Interrupts

To allow existing ARM peripherals to use message based interrupts, a system might include one or more "Consolidators" that provide a wire replacement function for a number of peripherals. It is expected that such Consolidators provide the address / data / enable registers (as described above) for each supported interrupt wire.

4.1.9 Distributor Registers for Message Based Interrupts

To support generation of SPIs using message based interrupts, additional registers in the top-level re-distributor are required for setting / clearing interrupts. Separate registers are provided for secure and non-secure interrupts.

MSI and MSI-X are edge triggered and only require a mechanism for setting them as pending. However, message-based interrupts can also be used to reduce the number of wires required to signal SPI interrupts from peripherals. SPI interrupts may be level sensitive and require additional registers for clearing their pending state.

Hence GICv3 provides registers in the top-level re-distributor that allow fully programmable SPIs to be generated by message:

- GICD_SETSPI_NSR. Set an SPI as "pending" if permitted by the security state of the access and the GICD_NSACRn value for that SPI.
- GICD_CLRSPI_NSR. Clear the pending state for an SPI if permitted by the security state of the access and the GICD_NSACRn value for that SPI.
- GIGD_SETSPI_SR. Set a secure (or non-secure) SPI as "pending".
- GICD_CLRSPI_SR. Clear the pending state for a secure (or non-secure) SPI.

These registers are specified below and their offset in the top-level Re-Distributor address map is shown in Table 21: Distributor Register Map below.

In addition to these registers in the top-level re-distributor, similar registers are provided in each re-distributor to allow Locality-specific Peripheral Interrupts to be set pending by message. See section 4.8.7 below.

4.1.10 SPI Wire Replacement

When an SPI is used as a message based interrupt, it is recommended that:

- A dedicated SPI that is not wired to a device is provided that may only be set pending by message, or
- If a device may set it as pending by asserting a signal, software must ensure that this is not possible before using the SPI as a general purpose message based interrupt.

4.1.11 MSI and MSI-X programming

In systems without Virtualization and no Interrupt Translation Service (see section 4.9), the address registers for message based interrupts (for example, MSI and MSI-X) must be programmed with the appropriate register address within the target Re-Distributor:

- Top-Level re-distributor: GICD_SETSPI_{S,NS}R (as appropriate for the security domain of the device).

The data registers must be programmed with the Interrupt ID of the interrupt (see section 4.8) to be generated. ***Note***: this must be a valid Interrupt ID for the addressed component otherwise an implementation might optionally generate a system error.

4.2 Distribution of Interrupts within the GIC

4.2.1 Introduction to Affinity Hierarchies

The ARM architecture provides an enumeration scheme for different processors in the system based on a scheme that encodes affinity information as part of the enumeration scheme. This enumeration scheme is also used for the GIC. In particular, GICv3 provides the capability for interrupts to be routed by affinity.

In ARMv8 (when operating in AArch64 mode), MPIDR provides four levels of affinity that define an affinity hierarchy. The GICv3 affinity routing information maps directly onto these four affinity levels. The diagram below shows an example affinity hierarchy:

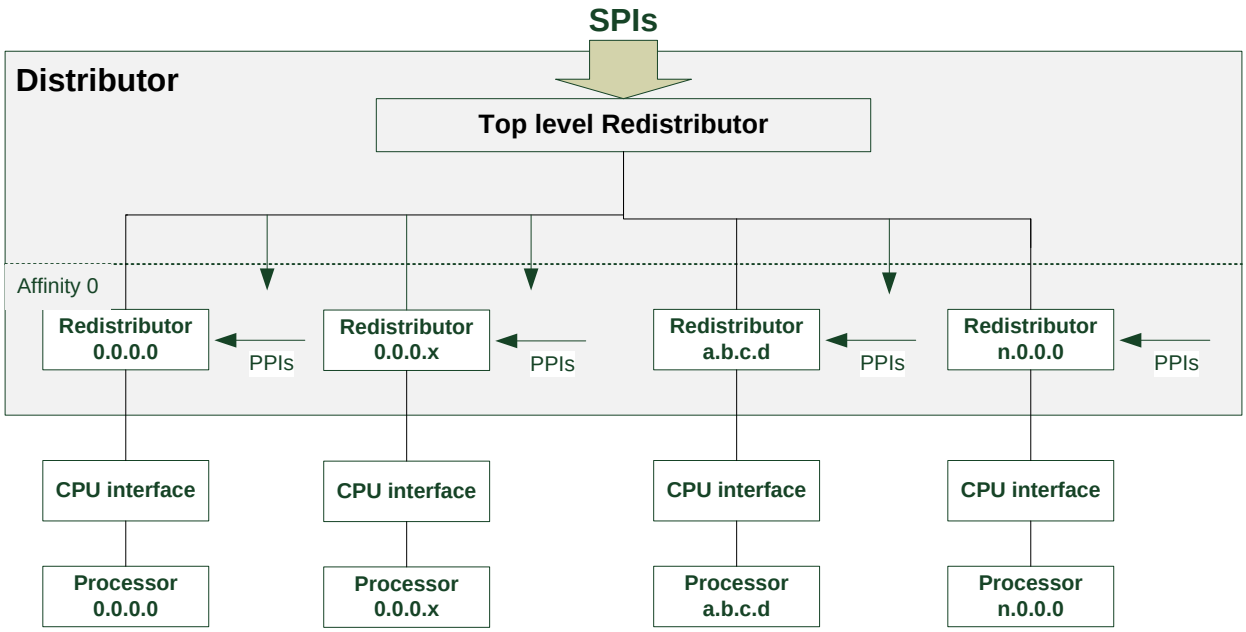


Figure 3: An example affinity hierarchy

In the figure above, there is a single re-distributor for each affinity level 0 node within the hierarchy. Each re-distributor is enumerated according to its path from the top-level re-distributor. Similarly, processors are also enumerated by their path from the top-level re-distributor.

Note: there is a one-to-one mapping between affinity level 0 re-distributors and processors.

4.2.2 Interrupt Routing in GICv3 Systems

In GICv3 when ARE is set for a security state, Shared Peripheral Interrupts (SPIs) can either be directed at a specified processor or can be distributed to one-of-N processors. Software Generated Interrupts are distributed to some-of-N processors that share the same affinity 1, affinity 2 and affinity 3 values.

This document expresses such affinity routing information as a path terminated by a wild-card indicator. Lower-case letters are used to signify a specific number (between 0 and 255) and "*" to signify a wild-card.

Affinity routing information for SPIs is expressed as below:

- "a.b.c.d". Send to the processor "a.b.c.d".
- "*". Send to any one processor below the top-level re-distributor.
- ***Note***: routing to other affinity paths, for example, "a.b.c.*" is not currently supported.

To ensure that only a single processor handles an SPI, GICv3 provides the following guarantees:

- The mechanism by which the target processor is selected is implementation defined.
    - ***Note***: an SPI might only be presented to one processor
- An interrupt will not be delivered to a processor that is not participating in the distribution scheme for an interrupt group (see section 4.5.4).

Affinity routing information for SGIs is expressed as below:

- Send to all in "a.b.c.{target list}" where "target list" is a one-hot set of bits specifying a set of processors within "a.b.c".
- Send to all in "*", excluding "self".

In systems supporting affinity routing, each affinity level might have between one and 256 children.

4.2.3 Permissible ARE settings

In GICv3, separate control is provided over ARE for the secure and non-secure state because it is expected that secure software is more difficult to change and will take longer to change than non-secure software. For this reason, only the following ARE settings are permitted:

- Both are zero. In this case, the GIC is operating in a fully GICv2 compatible mode and only eight or fewer processors are supported.
- Only the non-secure ARE bit is set. In this case, the secure side is limited to eight or fewer processors (that use the GIC) but the non-secure side supports an arbitrary number of processors. The restrictions on this case are outlined further below.
- Both are set. In this case the GIC is operating in a fully GICv3 compatible mode and an arbitrary number of processors is supported.

4.2.4 Restrictions when ARE_NS is set and ARE_S is zero

In GICv3, the following restrictions apply when the non-secure state is using affinity routing and the secure state is not using affinity routing:

- GICD_ITARGETSRn is RAZ/WI for any SPI where ARE is set for its security state. ***Note***: legacy secure software cannot re-route non-secure interrupts because GICD_IROUTERn is inaccessible to secure accesses (and is presumably not understood by "legacy" software). Secure software can however change the Group of the interrupt.
- GICD_NSACRn cannot be used (and wasn't implemented in GIC-400).
- Because routing is limited to at most eight processors when ARE is not set, the can be a maximum of eight secure processors that use the GIC in a system. The mapping between their bit position and their affinity is implementation defined.

- Generating SGIs from the non-secure side. If an SGI might target a secure processor, care must be taken to ensure the routing is correct.
- Generating SGIs from the secure side. When routed using the Targets List, SGIs must be delivered to the implementation defined processors where the appropriate bit is set. When routed using the Targets List Filter setting of "all but self", the SGI must be delivered to all processors, including processors not in the implementation defined set that support security. ***Note***: secure software that does not use affinity routing cannot use a non-secure alias to GICD_SGIR (by setting NS in the page tables) to generate non-secure SGIs as this would result in a non-secure write to GICD_SGIR which is RAZ/WI when ARE is set for the non-secure state.

4.2.5 Changing Affinity Routing Enables

The ARE settings for a security state should only be changed in controlled circumstances (otherwise the system behaviour is UNPREDICTABLE). Software must follow the follow sequence to ensure this:

- Interrupts must be disabled (and no interrupts may be active) for the target security state. This might be achieved by:
  - Clearing the appropriate enable bit(s) in GICD_CTLR and GICR_CTLR if ***all*** interrupts can safely be disabled
  - Software must poll GICD_CTLR.RWP / GICR_CTLR.RWP to ensure the effects are globally visible.
  - If all interrupts cannot be safely disabled, by clearing the enable bit(s) for all interrupts that target the security state
- Software must ensure the action to disable interrupts is visible by issuing an appropriate memory barrier.
- Software may then change the ARE setting for the target security state by setting the appropriate ARE bit(s) in GICD_CTLR
  - Software must GICD_CTLR.RWP / GICR_CTLR.RWP to ensure the effects are globally visible.
- Interrupts for the target security state may then be enabled.

Note: the same procedure may be followed to set GICD_CTLR.DS, except all interrupt groups must be disabled.

The ARE settings for a guest operating system (that is, software executing at non-secure EL0/1 under the control of software at EL2) may be changed without disabling interrupts if performed when executing at EL2 (or higher).

4.2.6 Interrupt Routing Modes

Registers GICD_IROUTERn, ICC_SGI0R_EL1, ICC_SGI1R_EL1 and ICC_ASGI1R_EL1 include a single bit "Interrupt Routing Mode" field that is used to determine how the generated interrupts should be distributed to processors.

The table below shows how these Interrupt Routing Mode fields are interpreted:

Mode	Routing for SPIs	Routing for SGIs
0	The processor specified by *a.b.c.d*	The processors specified by *a.b.c.{target list}*
1	Any one processor in the entire system	All processors in the system, excluding "self"

Table 4: Interrupt Routing Modes

4.2.7 Interrupt Ordering in GICv3

GICv3 supports distributed implementations, and makes the following guarantees about delivery of interrupts:

- the highest priority unmasked enabled interrupt will be delivered to a target processor in finite time.
- there is ***no*** guarantee that higher priority interrupts will always be taken before lower priority interrupts, though this will generally be the case (for example, when a processor is being powered off, a higher

priority interrupt might get rejected by the powering-off processor and may be taken by another processor after a later arriving lower priority interrupt).

4.2.8 Interrupt Handling in GICv3

In GICv2 systems which support two security states, Group 0 and Group 1 interrupts are handled using a single register which is banked by security and separate “alias” registers are provided to allow secure software to handle Group 1 interrupts. However, systems which support a single security state handle interrupts using separate registers for Group 0 and Group 1 interrupts (using the “alias” registers). ***Note***: for example, a Group 1 interrupt might be acknowledged using either a non-secure access to GICC_IAR or a secure access to GICC_AIAR.

In GICv3, both models are supported:

- Software might choose to use the “banked” model. ***Note***: a non-secure mapping might be required to allow secure software to generate non-secure accesses to a register to control Group 1 interrupts.
- Software might choose to always use the “alias” registers to handle Group 1 interrupt. ***Note***: non-secure access is always permitted to the “alias” registers.

Note: separate registers are also provided for Group 0 and Group 1 interrupts when using system register access.

4.2.9 SGI and PPI Configuration Registers

In GICv2 the registers that control the behaviour of interrupts 0 to 31 (SGIs and PPIs) are notionally in the top-level Re-Distributor though all such registers are banked per processor. These interrupts are logically associated with an affinity level 0 re-distributor. In GICv3, when ARE is set for a security state, these registers are accessed as affinity level 0 re-distributor registers.

When ARE is set for a security state, affinity level 0 re-distributor registers must be used to access the following registers (formerly banked in the top level re-distributor):

- GICD_ISPENDR0 must be accessed using GICR_ISPENDR0.
- GICD_ICPENDR0 must be accessed using GICR_ICPENDR0.
- GICD_ISENABLER0 must be accessed using GICR_ISENABLER0.
- GICD_ICENABLER0 must be accessed using GICR_ICENABLER0
- GICD_IPRIORITYR{0..7} must be accessed using GICR_IPRIORITYR{0..7}
- GICD_IGROUPR0 must be accessed using GICR_IGROUPR0
- GICD_IGRPMODR0 must be accessed using GICR_IGRPMODR0
- GICD_ISACTIVER0 must be accessed using GICR_ISACTIVER0
- GICD_ICACTIVER0 must be accessed using GICR_ICACTIVER0
- GICD_ICFGR0 and GICD_ICFGR1 must be accessed using GICR_ICFGR0 and GICR_ICFGR1 respectively.
- GICD_NSACR0 must be accessed using GICR_NSACR

For compatibility with GICv2, software on some processors might read or write the banked GICD_* registers and the Distributor must ensure such reads or writes access the correct state from the affinity level 0 re-distributor associated with the processor performing the access. An implementation defined set of eight processors might perform such accesses.

4.2.10 Software Generated Interrupts in GICv3

In GICv2, pending software generated interrupts were banked by source processor as well as destination processor. This does not scale well in large systems. GICv3 simplifies this so that, when ARE is set for a security state, pending SGIs are only banked by destination processor and:

- A “Source CPU ID” is no longer provided when reading an IAR
- A “Source CPU ID” is no longer required when writing to EOIR
- Only 16 SGI pending bits are required per re-distributor.

When ARE is set for a security state, some Distributor registers that are banked per processor are changed:

- GICD_SPENDSGIRn will be RES0. SGIs are no longer pending by source processor and the equivalent function is provided using GICR_ISPENDR0[0:15].
- GICD_CPENDSGIRn will be RES0. SGIs are no longer pending by source processor and the equivalent function is provided using GICR_ICPENDR0[0:15].

The GICD_SGIR register, used to generate SGIs, is disabled when ARE is set for a security state and a mechanism to generate SGIs is provided as part of the system register interface when SRE is set for an interrupt regime (see section 5.7.29):

- ICC_SGI1R_EL1 allows software operating in a given security state to generate Group 1 SGIs to the ***same*** security state on the target processors:
    - Non-Secure software can generate Non-Secure Group 1 interrupts to each target processor.
    - Secure software can generate Secure Group 1 interrupts to each target processor.
- ICC_ASGI1R_EL1 allows software operating in a given security state to generate Group 1 SGIs to the ***other*** security state on the target processors:
    - Non-Secure software can generate Secure Group 1 interrupts to each target processor, if permitted by GICR_NSACR at each target processor; see section 5.4.19).
    - Secure software can generate Non-Secure Group 1 interrupts to each target processor.
- ICC_SGI0R_EL1 allows secure software (and non-secure software if permitted by GICR_NSACR at each target processor; see section 5.4.19) to generate Secure Group 0 SGIs to each target processor.
    - ***Note***: it is expected that secure software might use ICC_SGI0R_EL1 to generate interrupts to secure software on each target processor.

4.3 Communication of Interrupts to the Processor

4.3.1 Processor Interrupt Exceptions

In the ARM architecture, processors provide three exceptions for the communication of events external to the processor. In systems that use GICv3, two of these are used for communication normal interrupt events to the processor:

- The **FIQ** interrupt exception. This is normally used for the communication of all Group 0 interrupts. In processors that support two security states, it is also used to communicate Group 1 interrupts when the processor is operating in a different security state than the target security state of the highest priority Group 1 interrupt. For example, if the processor is in the secure state and the highest priority Group 1 interrupt is for the non-secure state, this will be communicated using the FIQ interrupt exception. See section 4.6.4 for more details of how this switching of security states is achieved.
- The **IRQ** interrupt exception. This is normally used for the communication of all Group 1 interrupts.

In systems that use GICv3, the third of these (the SError exception) is used to communicate system error exceptions to the processor. This is specified in section 4.7.

Note: the GIC provides the ability to directly control the signaling of FIQ and IRQ. See section 4.6.2.

4.3.2 Overall Flow of Interrupt Handling

In the GIC architecture, there are two methods software might employ to handle an interrupt. The GIC provides a control called “EOI mode” that selects between these two methods:

- When EOI mode is zero, the interrupt may be completely handled in an Interrupt Service Routine (ISR) entered directly from a processor interrupt exception (FIQ or IRQ).
- When EOI mode is one, handling of the interrupt may be split between an ISR entered directly from a processor interrupt exception and an Interrupt Service Task (IST) schedule by software during execution of the ISR.

The GIC provides a separate EOI mode control for each security state. ***Note***: it is recommended that EOI mode is programmed to one if a security state supports any level sensitive interrupts.

When handling an interrupt software must follow the following sequence:

- After entry into the ISR, software must read an Interrupt Acknowledge Register (IAR). Reading this register will return an interrupt identifier to software. If the interrupt identifier returned to software identifies a normal interrupt, the pending state of that interrupt is cleared and the active state will be set for SGIs, PPIs and SPIs. ***Note***: LPIs do not have an active state.
- The read of IAR might return a special identifier rather than the identifier of a normal interrupt, so software must then check the interrupt identifier returned from the read of IAR. The following special identifiers might be received:
    - 1020. This identifier may only be received if an FIQ interrupt exception is taken at EL3. If this identifier is received, it indicates that a Group 1 interrupt for the secure state has been received. See section 4.6.4 for the expected behaviour.
    - 1021. This identifier may only be received if an FIQ interrupt exception is taken at EL3. If this identifier is received, it indicates that a Group 1 interrupt for the non-secure state has been received. See section 4.6.4 for the expected behaviour.
    - 1022.
    - 1023. This identifier indicates that a race condition occurred such that there is no valid interrupt to be handled. If this identifier is received, software should just perform a return from exception and should not follow the remainder of the sequence described below.
- If EOI mode is set for the current security state, software must schedule the appropriate IST.

- Software must write the identifier received when IAR was read to an End of Interrupt Register (EOIR). The behaviour of this write depends on the EOI mode for the current security state:
    - When EOI mode is zero, this clears the active state of the interrupt
    - When EOI mode is one, this performs a “priority drop” such that the interrupt exception will not be taken again (causing multiple entries into the ISR) for this interrupt. Except for LPIs (which do not have an active state), the active state of the interrupt must be cleared by a subsequent write to a Deactivate Interrupt (DIR) register.
- After entry into the IST (if any) and if the interrupt is level-sensitive, software must perform any actions required to ensure that the interrupt is de-asserted
- Except for LPIs, software must write to the DIR to clear the active state of the interrupt.

4.3.3 Handling of Edge Triggered Interrupts

The diagram below shows an overview of how an edge-triggered interrupt is processed. ***Note***: this assumes EOI mode is one so that “priority drop” and clearing the active state are separate actions.

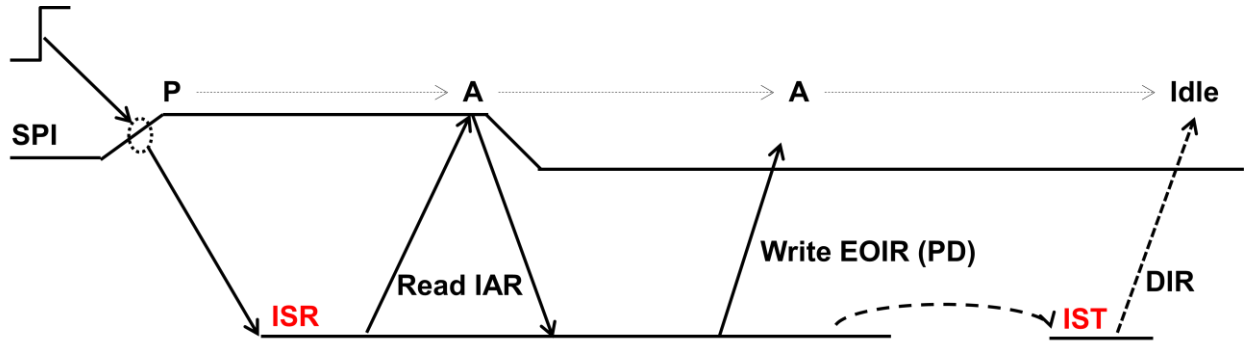


Figure 4: Handling edge-triggered interrupts

Note: the effects of a read of ICC_IAR{0,1}_EL1 on the signaling of interrupt exceptions to the processor must be observed when the instruction retires.

4.3.4 Handling of Level Sensitive Interrupts

The diagram below shows an overview of how a level-sensitive interrupt is processed. ***Note***: this assumes EOI mode is one so “priority drop” and clearing the active state are separate actions.

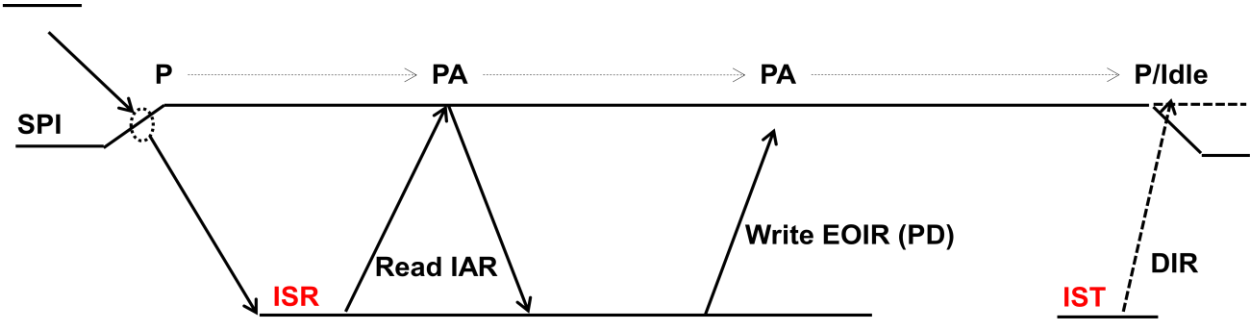


Figure 5: Handling level-sensitive interrupts

4.3.5 Processor Access to the GIC

In GICv2 all accesses to GIC registers were memory mapped and the GIC registers were split into categories. In GICv3, these categories are modified and the following categories are provided:

- GICD_* registers. A single page of registers accessible to all processors. These registers are used to control the configuration all interrupts. ***Note***: when ARE is zero for a security state, registers within this page associated with SGIs and PPIs are banked on a per-processor basis. See section 4.2.1.
- GICR_* registers. Multiple pages of registers per re-distributor accessible to all processors. These registers are used by a processor to control LPIs and to control power on and power-off. ***Note***: when ARE is one for a security state, GICR_* registers replace the GICD_* registers associated with SGIs and PPIs.
- GICC_* registers. Two pages of registers per-processor, accessible only by that processor. These registers are used by a processor to handle physical interrupts (using an IAR, EOIR and DIR register as described above) and to save and restore state. ***Note***: only used when ARE is zero for a security state.
- GICH_* registers. A single page of registers per-processor, accessible only by that processor. These registers are used to control the generation of virtual interrupts visible to a processor using the GICV_* registers and to save and restore the virtual interrupt state of a processor. ***Note***: only used when ARE is zero for a security state.
- GICV_* registers. Two pages of registers per-processor, accessible only by that processor. These registers are used by a processor to handle virtual interrupts (using an IAR, EOIR and DIR register as described above). ***Note***: only used by memory mapped guest operating systems. The hypervisor is responsible for ensuring that either GICC_* or GICV_* registers but not both are visible in the address map of a memory-mapped guest operating system. See sections 4.6.8 and 5.10.

In GICv3 when ARE is set for a security state, the GICC_* and GICH_* registers are not used and system registers are used instead. This is described in the section that follows.

4.4 Introduction to System Registers

4.4.1 System Register Access to CPU Interface Registers

GICv3 provides configuration bits that determine whether the CPU interface registers are memory-mapped (for backwards compatibility with GICv2) or system register mapped. Unless otherwise stated, the functionality of a system register is the same as the equivalent memory mapped register. The registers affected are shown in the table below:

Purpose	Memory mapped registers		System Registers	Notes
	Physical	Virtual		
Interrupt Acknowledgement	GICC_IAR* GICC_AIAR	GICV_IAR GICV_AIAR	ICC_IAR0_EL1 ICC_IAR1_EL1	
End of Interrupt	GICC_EOIR* GICC_AEOIR	GICV_EOIR GICV_AEOIR	ICC_EOIR0_EL1 ICC_EOIR1_EL1	
Highest Priority Pending Interrupt	GICC_HPPIR* GICC_AHPPIR	GICV_HPPIR GICV_AHPPIR	ICC_HPPIR0_EL1 ICC_HPPIR1_EL1	
Binary Point	GICC_BPR* GICC_ABPR	GICV_BPR GICV_ABPR	ICC_BPR0_EL1 ICC_BPR1_EL1*	
Priority Mask	GICC_PMR	GICV_PMR	ICC_PMR_EL1	
Running Priority	GICC_RPR	GICV_RPR	ICC_RPR_EL1	
Interrupt Deactivation	GICC_DIR	GICV_DIR	ICC_DIR_EL1	
Active Priorities	GICC_APR*n* GICC_NSAPR*n*	GICV_APR*n*	ICC_AP0R*n*_EL1 ICC_AP1R*n*_EL1*	
Control	GICC_CTLR*	GICV_CTLR	ICC_CTLR_EL1* ICC_CTLR_EL3	
Virtualisation Control	GICH_HCR GICH_VTR GICH_VMCR GICH_MISR GICH_EISR*n* GICH_ELSR*n* GICH_LR*n* GICH_APR*n*	-	ICH_HCR_EL2 ICH_VTR_EL2 ICH_VMCR_EL2 ICH_MISR_EL2 ICH_EISR_EL2 ICH_ELSR_EL2 ICH_LR*n*_EL2 ICH_AP0R*n*_EL2 ICH_AP1R*n*_EL2	

Table 5: CPU interface register mapping

Where:

- Memory-mapped registers marked "*" are banked by security. Non-secure accesses read / modify state for Group 1 interrupts and secure accesses read / modify state for Group 0 interrupts
- System registers marked "*" are banked by security. Non-secure accesses read / modify state for Non-secure Group 1 interrupts and secure accesses read / modify state for Secure Group 1 interrupts

When executing in Non-secure EL1, a single system register encoding can be configured to access either a GICC register or the equivalent GICV register. That way, as desired, a Guest OS cannot tell if it has access to physical or virtual resources.

Additionally, in GICv2 the register used to generate SGIs is memory mapped. For performance and scalability, it is desirable to make SGI generation system register accessible.

GICv3 provides configuration bits that determine whether the SGI generation registers are memory-mapped (for backwards compatibility with GICv2) or system register mapped:

Purpose	Memory mapped registers		System Registers	Notes
	Physical	Virtual		
Software Generated Interrupts	GICD_SGIR	-	ICC_SGI0R_EL1 ICC_SGI1R_EL1 ICC_ASGI1R_EL1	

Table 6: SGI generation register mapping

Where:

- ICC_SGI0R_EL1 is expected to be used when software wishes to communicate with secure software using SGIs.
- ICC_SGI1R_EL1 is expected to be used when software requires an interrupt to the same security state.
- ICC_ASGI1R_EL1 is expected to be used when software requires an interrupt to the other security state.

Note: though mapped to CPU system registers, writes to ICC_SGI0R_EL1, ICC_SGI1R_EL1 and ICC_ASGI1R_EL1 cause communication to all destination processors.

4.4.2 Interrupt Regimes

Interrupts may be handled at different exception levels and security states. A combination of exception level and security state at which an interrupt is handled is referred to as an ***Interrupt Regime***. The following interrupt regimes are defined when EL3 is using AArch64:

- (Secure) EL3
- (Non-Secure) EL2
- Non-Secure EL1
- Secure EL1 (only exists when EL3 is using AArch64)

When EL3 is using AArch32, the following additional interrupt regimes are defined:

- Monitor
- Secure non-Monitor

4.4.3 System Registers and Interrupt Regime

To allow compatibility for “legacy” memory-mapped software, separate control over use of system registers has been provided per interrupt regime. The table below specifies how the system register enable registers govern access to the other system registers:

Level	Control bits	Description
EL3	ICC_SRE_EL3.SRE	Controls whether ICC_* and ICH_* system registers (excluding ICC_SRE_EL2 and ICC_SRE_EL1) are accessible by EL3 or Monitor software ***Note***: ICC_SRE_EL3 is always accessible by EL3 and Monitor software.
	ICC_SRE_EL3.Enable	Controls whether software executed at exception levels below EL3 or Monitor can access ICC_SRE_EL1 / ICC_SRE_EL2 or whether accesses to those registers trap to EL3 ***Note***: when EL3 is using AArch32, an UNDEF_cfg exception will be

		generated instead. When ICC_SRE_EL3.SRE is zero, the value of Enable is ignored and is treated as one.
EL2	ICC_SRE_EL2.SRE	Controls whether ICC_* and ICH_* registers are accessible by EL2 software.
	ICC_SRE_EL2.Enable	Controls whether software executed at exception levels at non-secure EL1 can access ICC_SRE_EL1 or whether non-secure EL1 accesses to this register trap to EL2. When ICC_SRE_EL3.SRE is zero or ICC_SRE_EL2.SRE is zero, the value of Enable is ignored and is treated as one.
EL1	ICC_SRE_EL1.SRE	This register is banked by security and controls whether ICC_* registers are accessible by software executing at EL1 for the current security state.

Table 7: System Registers and Exception Levels

4.4.4 Access to System Register Enables

This section summarises the rules for access to the various SRE bits when access is permitted to the register and does not result in a trap (i.e. if the exception level of the access is less than the minimum required, or if an "Enable" bit is zero as specified in section 4.4.3).

The SRE bit for EL3 (ICC_SRE_EL3.SRE) has the following effects:

- When ICC_SRE_EL3.SRE is zero, all other SRE bits are RAZ/WI
- When ICC_SRE_EL3.SRE is changed from zero to one, all other SRE bits become UNKNOWN.

The SRE bit for EL2 (ICC_SRE_EL2.SRE) has the following behaviour:

- This bit is treated as zero (and is RAZ/WI) if ICC_SRE_EL3.SRE is zero. If ICC_SRE_EL3.SRE is changed from zero to one, this bit becomes UNKNOWN.
- Otherwise, this bit is treated as one (and is RAO/WI) if the secure copy of ICC_SRE_EL1.SRE is one. If the secure copy of ICC_SRE_EL1.SRE is changed from one to zero, this bit becomes UNKNOWN.

The SRE bit for secure EL1 (the secure copy of ICC_SRE_EL1.SRE) has the following behaviour:

- This bit is treated as zero (and is RAZ/WI) if ICC_SRE_EL3.SRE is zero.
- If ICC_SRE_EL3.SRE is changed from zero to one, the secure copy of this bit becomes UNKNOWN.

The SRE bit for non-secure EL1 (the non-secure copy of ICC_SRE_EL1.SRE) has the following behaviour:

- The non-secure copy of this bit is treated as one (and is RAO/WI) if the secure copy of the bit is one and any of HCR_EL2.{FMO, IMO, AMO} are zero.
- Otherwise (i.e. when the non-secure copy of this bit is not being treated as one), the non-secure copy of this bit is treated as zero (and is RAZ/WI) if ICC_SRE_EL2.SRE is zero and is not treated as one or if ICC_SRE_EL2.SRE is treated as zero.
- If the secure copy of this bit is changed from one to zero, or if the secure copy of the bit is one and all of HCR_EL2.{FMO, IMO, AMO} become one, or if the effective value of ICC_SRE_EL2.SRE is changed from zero, the non-secure copy of the bit becomes UNKNOWN.

Note: this summary does not cover the behaviour when EL2 or EL3 are not present. This is fully defined in sections 5.7.38, 5.7.39 and 5.7.40.

4.4.5 Relationship between ARE and SRE

The setting of ARE for a security state governs the register interface software must use for handling physical interrupts for that security state. In a system with two security states, the permitted ARE settings and their effect on which interface is used is shown below:

<table>
<tr><th colspan="2">ARE settings</th><th colspan="3">Interrupt Group</th></tr>
<tr><th>ARE_NS</th><th>ARE_S</th><th>G0S</th><th>G1NS</th><th>G1S</th></tr>
<tr><td>0</td><td>0</td><td rowspan="2">Handled using secure accesses to GICC_* memory-mapped registers
Must be handled by a secure interrupt regime and SRE must be zero for that interrupt regime.</td><td>Handled using non-secure accesses to GICC_* memory-mapped registers
Usually handled by a non-secure interrupt regime and SRE must be zero for that interrupt regime.</td><td>-</td></tr>
<tr><td>1</td><td>0</td><td rowspan="2">Handled using system registers ICC_IAR1_EL1, ICC_EOIR1_EL1 (and possibly ICC_DIR_EL1).
Usually handled by a non-secure interrupt regime and SRE must be one for that interrupt regime.
Note: a secure interrupt regime is permitted to handle G1NS interrupts but SRE must be one for the handling interrupt regime.</td><td>-</td></tr>
<tr><td>1</td><td>1</td><td>Handled using system registers ICC_IAR0_EL1, ICC_EOIR0_EL1 (and possibly ICC_DIR_EL1).
Must be handled by a secure interrupt regime and SRE must be one for that interrupt regime.</td><td>Handled using system registers ICC_IAR1_EL1, ICC_EOIR1_EL1 (and possibly ICC_DIR_EL1)
Must be handled by a secure interrupt regime and SRE must be one for that interrupt regime.</td></tr>
</table>

Table 8: How ARE affects interrupt handling

Note: ICC_SRE_EL3.SRE might be set to one when ARE is zero for only the secure state or for both security states. In this configuration, EL3 is only used for coordination of switching between the security states on receipt of an interrupt. As such, EL3 does not “handle” any interrupts, though it might read ICC_IAR{0,1}_EL1 and receive special identifiers indicating the target security state for an interrupt. See sections 4.6.4 and 4.6.5.

Note: in a configuration where only ICC_SRE_EL3.SRE is one, ARE must be zero for both security states. Hence, to generate SGIs EL3 software must write to GICD_SGIR. See sections 5.3.14 and 5.7.29.

4.4.6 Permissible System Register Enable settings

In GICv3 separate control is provided over SRE for EL3 and for secure EL1 and non-secure EL1/2 because it is expected that secure software is more difficult to change and will take longer to change than non-secure software.

The SRE settings are per-processor, and some processors may use memory mapped interfaces for an interrupt regime while others use the system register interface. In particular, this is envisioned for systems where ARMv7 and ARMv8 processors form a single coherency domain and share a single GIC.

The permitted SRE settings are shown below:

ICC_SRE_EL3.SRE	ICC_SRE_EL2.SRE	Non-secure ICC_SRE_EL1.SRE	Secure ICC_SRE_EL1.SRE	Notes
0	0	0	0	Memory mapped access only. SRE_EL2, SRE_EL1S and SRE_EL1NS cannot be modified (assumes ICC_SRE_EL3.Enable is zero). ***Note***: GICD_CTLR.ARE_S and GICD_CTLR.ARE_NS must both be zero. ***Note***: reporting of System Errors by message is not supported.
1	0	0	0	System register access for the secure EL3 interrupt regime only. ***Note***: generally this will be used with ICC_CTLR_EL3.RM == '1' SRE_EL1NS cannot be modified (assumes ICC_SRE_EL2.Enable is zero). Interrupts routed to Secure EL3 use ICC_* system registers ***Note***: GICD_CTLR.ARE_S and GICD_CTLR.ARE_NS must both be zero. ***Note***: when only EL3 is using system registers, it is expected the SError exceptions will be routed to EL3.

1	1	0	0	System register access for non-secure EL2 and the hypervisor must use ICH_* registers. Interrupts routed to Secure EL3 use ICC_* system registers Interrupts routed to Secure EL1 use GICC_* registers and GICD_CTLR.ARE_S must be zero. For non-secure physical interrupts, there are two options: • Interrupts routed to non-secure EL2 use ICC_* and GICD_CTLR.ARE_NS must be one, or • Interrupts routed to non-secure EL1 use GICC_* and guest operating systems cannot use system registers and GICD_CTLR.ARE_NS must be zero If GICD_CTLR.DS is zero and EL3 is present, FIQ must be routed to EL3 and the routing of IRQ governs how software must configure GICD_CTLR.ARE_NS. If GICD_CTLR.DS is one, then both interrupts must be routed to the same EL and this governs how software must configure GICD_CTLR.ARE. ***Note***: if software changes the routing of interrupts such that an ARE setting must be changed, the sequence in section 4.2.5 must be followed and the routing must be changed after ARE has been changed but before interrupts are re-enabled. ***Note***: when SRE_EL1NS is zero for a guest operating system, the hypervisor must ensure that the re-distributor 0 is programmed such that no virtual LPIs are forwarded to the CPU interface. That is, it must ensure that GICR_VPENDBASER.Valid is zero.
1	1	1	0	System register access for all non-secure interrupt regimes and for secure EL3. Interrupts routed to Secure EL3 use ICC_* system registers Interrupts routed to Secure EL1 use GICC_* and GICD_CTLR.ARE_S must be zero. Interrupts routed to non-secure EL1 and EL2 use ICC_* and GICD_CTLR.ARE_NS must be one.

1	1	*	1	System register access for all non-secure interrupt regimes and for secure EL3. Interrupts routed to Secure EL3 use ICC_* system registers Interrupts routed to secure EL1 use ICC_* Interrupts routed to non-secure EL2 use ICC_* If any physical interrupts are routed to non-secure EL1 (that is, if any of HCR_EL2.{FMO, IMO, AMO} is zero), then non-secure EL1 uses ICC_* (in this case, SRE_EL1NS is treated as one) If no physical interrupts are routed to non-secure EL1 and all non-secure EL1 interrupts are virtualized (that is, if all of HCR_EL2.{FMO, IMO, AMO} are one), the guest operating system uses GICC_* of ICC_*, depending on the value of SRE_EL1NS (i.e. the value of "*" in the SRE_EL1NS column) ***Note***: GICD_CTLR_ARE_NS and GICD_CTLR.ARE_S must be one. ***Note***: when SRE_EL1NS is zero for a guest operating system, the hypervisor must ensure that the re-distributor 0 is programmed such that no virtual LPIs are forwarded to the CPU interface. That is, it must ensure that GICR_VPENDBASER.Valid is zero.

Table 9: Supported SRE Settings

4.4.7 Implementations with Fixed System Register Enables

GICv3 implementations that do not require GICv2 compatibility might choose to make some system register enable bits RAO/WI. The following options are supported:

- ICC_SRE_EL3.SRE (see section 5.7.40) may be RAO/WI. This means EL3 software must always access the GIC using system registers but lower exception levels might use memory-mapped access to the GIC subject to the rules in section 4.4.6.
- ICC_SRE_EL2.SRE (see section 5.7.39) may be RAO/WI if ICC_SRE_EL3.SRE is also RAO/WI. This means EL2 software must always access the GIC using system registers not non-secure EL1 software might use memory-mapped access to the GIC subject to the rules in section 4.4.6.
- The non-secure copy of ICC_SRE_EL1.SRE (see section 5.7.38) may be RAO/WI if ICC_SRE_EL2.SRE is also RAO/WI. This means all non-secure software, including guest operating systems using only virtual interrupts, must access the GIC using system registers.
- The secure copy of ICC_SRE_EL1.SRE (see section 5.7.38) may be RAO/WI if ICC_SRE_EL3.SRE and ICC_SRE_EL2.SRE are also RAO/WI. This means that all secure software must access the GIC using system registers and all non-secure accesses to registers for physical interrupts must use system registers. ***Note***: a guest operating system using only virtual interrupts might still use memory-mapped access if the non-secure copy of ICC_SRE_EL1.SRE is ***not*** RAO/WI.

4.4.8 Changing System Register Enables

The System Register Enable settings for a physical interrupt regime should only be changed in controlled circumstances (otherwise the behaviour is UNPREDICTABLE). ***Note***: this includes any implicit change to the system register enables due to, for example, changes to the set of virtual interrupt exceptions visible to a guest operating system.

Software must follow the following sequence to ensure this:

- Interrupts must be disabled (and no interrupts may be active) for the target interrupt regime. This might be achieved by:
    - Clearing the appropriate enable bit(s) in GICD_CTLR and GICR_CTLR if ***all*** interrupts can safely be disabled
    - Software must poll GICD_CTLR.RWP / GICR_CTLR.RWP to ensure the effects are globally visible.
    - If all interrupts cannot be safely disabled and SRE is zero for the target interrupt regime, clearing the appropriate enable bit(s) in GICC_CTLR, or
    - If all interrupts cannot be safely disabled and SRE is set for the target interrupt regime, clearing the enable bit in ICC_IGRPEN{0,1}_EL1 as appropriate
    - ***Note***: when changing SRE for EL2 from one to zero, software must also ensure no virtual interrupts are active. That is, it must ensure the virtual active priorities registers are zero (GICH_APRn or ICH_AP{0,1}Rn_EL2 as appropriate).
- Software must ensure the action to disable interrupts is visible by issuing an appropriate memory barrier.
- Software may then change the SRE setting for the target interrupt regime.
    - See section 5.7.40 for the effects on the values of GICC_* and ICC_* if the SRE setting is changed for the EL3 interrupt regime
    - See sections 5.5.2 and 5.7.38 for the effects on the values of GICC_* and ICC_* if the SRE setting is changed for the secure EL1 interrupt regime
    - See section 5.7.39 for the effects on the values of the hypervisor control registers if the SRE setting is changed for the non-secure EL2 interrupt regime.
    - ***Note***: Software must also ensure all system registers accessible by the target interrupt regime are programmed to known values
- Interrupts for the target interrupt regime may then be enabled.

The System Register Enable settings for the non-secure EL1 interrupt regime may be changed without disabling interrupts if performed when executing at EL2 or higher (for example, when a hypervisor is changing the guest operating system at EL1). When executing at non-secure EL1, interrupts must be disabled (or otherwise guaranteed to never occur) before the System Register Enable settings are changed.

4.4.9 Banking of EL1 System Registers

In AArch64, most EL1 registers are not banked for security and EL3 software loads the correct security context when switching between security states.

Interrupts are physical events that might occur when the target security context is out of scope. This means that some GIC system registers are banked by security:

- ICC_SRE_EL1. See section 5.7.38.
- ICC_CTLR_EL1. See section5.7.33.
- ICC_IGRPEN1_EL1. See section 5.7.36.
- ICC_BPR1_EL1. See section 5.7.25.
- ICC_AP1Rn_EL1. See section 5.7.42.

4.4.10 System Registers and EL3 Access

When EL3 accesses system registers and EL3 system register access is enabled:

- When EL3 is configured as AArch32, EL3 access to EL2 registers is only permitted if SCR_EL3.NS == ‘1’ otherwise an UNDEF_und exception will be generated (see section 6.2)

- When EL3 is configured as AArch64, EL3 access to ICC_SRE_EL2 is only permitted if SCR_EL3.NS == ‘1’ otherwise an UNDEF_und exception will be generated (see section 6.2). ***Note***: this behaviour is expected for all registers that would be banked if a “Secure EL2” interrupt regime was introduced.
- EL3 accesses to EL1 registers that are banked by security access the secure copy when SCR_EL3.NS == ‘0’ and the non-secure copy when SCR_EL3.NS == ‘1’
- EL3 accesses to EL1 registers that are not banked by security access a common register.

4.4.11 Observability of GIC Register Accesses

To allow software to ensure appropriate observability of actions initiated by GIC register accesses, the processor and CPU interface logic must ensure that:

- Writes to ICC_PMR_EL1 are self-synchronising. This ensures that no interrupts below the written PMR value will be taken after a write to ICC_PMR_EL1 is architecturally executed. See section 5.7.27.
- Reads of ICC_IAR0_EL1 and ICC_IAR1_EL1 are self-synchronising when interrupts are masked by the processor (i.e. when PSTATE.{I,F} are one). This ensure that the effect of activating an interrupt on the signaling of interrupt exceptions is observed when a read of ICC_IAR{0,1}_EL1 is architecturally executed such that no spurious interrupt exception occurs if interrupts are unmasked by an instruction immediately following the read. See sections 5.7.11 and 5.7.13.
- Instructions that change the current exception level from 3 to a lower value (e.g. an ERET) must be synchronised with any corresponding change in the allocation of interrupts to FIQ and IRQ such that no spurious FIQ is taken after architectural execution of the instruction. See section 4.6.3.
- Architectural execution of a DSB instruction guarantees that:
    - The last value written to ICC_PMR_EL1 or GICC_PMR is observed by the associated re-distributor
    - The last value written to ICC_SGI{0,1}_EL1 is observed by the associated re-distributor
    - The last value written to ICC_ASGI1_EL1 is observed by the associated re-distributor
    - The last value written to ICC_IGRPEN{0,1}_EL1, ICC_IGRPEN1_EL3 or GICC_CTLR.EnableGrp* is observed by the associated re-distributor
    - The last value written to ICH_VMCR_EL2.VENG{0,1} or GICV_CTLR.EnableGrp* is observed by the associated re-distributor
    - The last SPI interrupt identifier read from ICC_IAR{0,1}_EL1 or GICC_{A}IAR is observable by the top-level Distributor and by accesses from any processor to the top-level Distributor
    - The last SGI/PPI/LPI interrupt identifier read from ICC_IAR{0,1}_EL1 or GICC_{A}IAR is observable by the associated re-distributor and by accesses from any processor to that re-distributor
    - The last Deactivate packet for an SPI generated by a write to ICC_EOIR{0,1}_EL1, GICC_{A}EOIR, ICC_DIR_EL1 or GICC_DIR is observable by the top-level Distributor and by accesses from any processor to the top-level Distributor
    - The last Deactivate packet for an SGI/PPI generated by a write to ICC_EOIR{0,1}_EL1, GICC_{A}EOIR, ICC_DIR_EL1 or GICC_DIR is observable by the associated re-distributor and by accesses from any processor to that re-distributor

4.5 Introduction to Interrupt Properties

4.5.1 Priorities and Visibility

GICv3 uses the same priority model as GICv2. That is:

- The priority of all interrupts is visible to Secure EL1 and EL3. Software may program any supported priority value. ***Note***: an implementation might choose to support fewer priorities than the maximum permitted by the architecture.
- In systems supporting two security states, Non-Secure EL1/2 can only see the priority of non-secure interrupts and cannot see priorities less than 128. ***Note***: in such systems, non-secure accesses to priority registers see the shifted value as defined for GICv2.

Note: in systems supporting a single security state (or when GICD_CTLR.DS is one; see section 5.3.20) all accesses see the full priority range.

4.5.2 Priority Masks

A single Priority Mask Register, ICC_PMR_EL1, specifies the priority mask when SRE is set for an interrupt regime. ***Note***: this register and its memory mapped equivalent, GICC_PMR might access the same state. See section 5.7.27 for details.

4.5.3 Binary Point and Priorities

In GICv3, there are three binary point registers. Secure Group 0 interrupts use ICC_BPR0_EL1 to determine how to interpret their priority. ICC_BPR1_EL1 remains banked by security and is used by Secure Group 1 and Non-secure Group 1 interrupts to determine how to interpret their respective priorities. The 128 highest priorities are allocated to Secure Group 0 and Secure Group 1. The remaining 128 may be shared between Secure Group 0, Secure Group 1 and Non-Secure Group 1. This priority scheme is shown below:

GICD_IPRIORITYR	Group 0	Group 1	
Bits[7:6]	Secure	Secure	Non-Secure
0b00	✓	✓	
0b01	✓	✓	
0b10	✓	✓	✓
0b11	✓	✓	✓

Table 10: Priority Visibility

Note: secure software might allocate a priority within the 128 highest priorities to a Non-secure Group 1 interrupt but the actual priority value allocated will not be visible to non-secure software.

The table below shows how, for example, priorities might be allocated to:

- Ensure there are some Secure EL3 interrupts that are higher priority than any other interrupts
- Ensure there are some Secure EL1 interrupts of higher priority than any Non-secure interrupt.

GICD_IPRIORITYR	Group 0	Group 1	
Bits[7:6]	Secure	Secure	Non-Secure
0b00	✓		
0b01	✓	✓	
0b10		✓	✓

0b11			✓

Table 11: Example Priority Allocation

4.5.4 Interrupt Enables and Participation in Distribution

For each CPU Interface, there is an Enable bit that applies to each interrupt group that determines whether the interrupt is signaled to the processor:

- 0b0. The interrupt group is not enabled and will not be signaled to the processor by the CPU interface.
- 0b1. The interrupt group is enabled and will be signaled to the processor by the CPU interface.

This “Enable” bit is defined in the table below:

Enable	Behaviour
0	The processor is not participating in the distribution scheme for that interrupt group. The Distributor might never forward an interrupt for the group to the CPU interface and must not choose the processor as the target of a “one of N” interrupt.
1	The processor is participating in the distribution scheme for that interrupt group. The Distributor may forward interrupts for the group to the CPU interface and may choose the processor as the target of a “one of N” interrupt.

Table 12: Interrupt Enables and Participation

The table below shows how the “Enable” bits are mapped to registers:

Interrupt Group	SRE == ‘0’ for the interrupt regime		SRE == ‘1’ for the interrupt regime	
	Register	Notes	Register	Notes
Secure Group 0	GICC_CTLR	Secure accesses only See section 5.6.18.	ICC_IGRPEN0_EL1	See section 5.7.35.
Virtual Group 0	GICV_CTLR		Virtual accesses to ICC_IGRPEN0_EL1	See section 5.7.35.
Non-Secure Group 1	GICC_CTLR	See section 5.6.17.	ICC_IGRPEN1_EL1 (non-secure accesses)	See section 5.7.36.
Virtual Group 1	GICV_CTLR		Virtual accesses to ICC_IGRPEN1_EL1	See section 5.7.36.
Secure Group 1	-		ICC_IGRPEN1_EL1 (secure accesses)	See section 5.7.36.

Table 13: Access to Interrupt Group Enables

Note: whether access to the system registers ICC_IGRPEN0_EL1 and ICC_IGRPEN1_EL1 is permitted is governed by interrupt routing. See Table 17: CPU Interface System Register Access Behaviour for details.

4.5.5 SPI Interrupt Configuration

To configure an SPI interrupt, to ensure that interrupts are never distributed using partially updated configuration information, software must:

- Ensure the interrupt is not active
- Ensure that the interrupt is disabled
    - This might be done either by writing to GICD_CTLR to clear the enables for a group, or
    - By writing to GICD_ICENABLERn to clear the Enable bit of the interrupt (see section 5.3.11).
    - In both cases, software must poll GICD_CTLR.RWP to ensure the effects are visible (see section 5.3.20).
- Program the routing (if appropriate), priority and group
- Enable the interrupt (if required)

4.5.6 SGI and PPI Interrupt Configuration

To configure an SGI or PPI interrupt, to ensure that interrupts are never distributed using partially updated configuration information, software must:

- Ensure the interrupt is not active
- Ensure that the interrupt is disabled
    - This might be done either by writing to GICD_CTLR to clear the enables for a group and polling GICD_CTLR.RWP to ensure the effects are visible (see section 5.3.20), or
    - By writing to GICR_ICENABLER0 to clear the Enable bit of the interrupt (see section 5.4.14) and polling GICR_CTLR.RWP to ensure the effects are visible (see section 5.4.7).
- Program the routing (if appropriate), priority and group
- Enable the interrupt (if required)

4.5.7 LPI Interrupt Configuration

For LPI configuration, see sections 4.8.10 to 4.8.13 inclusive.

4.6 Interrupt Signaling and Routing

This section covers how the three interrupt groups are allocated to FIQ and IRQ, and how FIQ and IRQ are routed to an exception level.

It also covers how the exception level to which an interrupt is routed modifies system register accessibility and the behaviour of the interrupt acknowledge registers.

4.6.1 External Interrupts to the Processor

The ARM architecture supports two classes of interrupt exception: FIQ and IRQ. These can be generated either by the GIC in response to Group 0 or Group 1 interrupts, or can be driven by pins on the device if the CPU Interface is disabled.

The Interrupt Groups that are supported by the GIC are allocated to FIQ and IRQ as described in section 4.6.3 below.

4.6.2 Interrupt Bypass

A processor supporting GICv3 might need to bypass the CPU interface logic if the system uses a simple external interrupt controller during the boot process but uses the GIC during normal operation.

- To enable such systems when SRE is zero, FIQ and IRQ might be driven by an external interrupt controller when the appropriate Enable is zero (see section 4.5.4) and separate bypass disable bits are provided in GICC_CTLR (see section 5.6.18).
- To enable such systems when SRE is set, FIQ and IRQ might be driven by an external interrupt controller when the appropriate interrupt enable is zero (see section 4.5.4) and separate bypass disable bits are provided in ICC_SRE_EL3 (see section 5.7.40).

4.6.3 Allocation of Interrupt Groups to FIQ and IRQ

In systems supporting a single security state (i.e. when EL3 is not present or when GICD_CTLR.DS is one) or when SRE is zero for the Secure EL1 interrupt regime (i.e. when ICC_SRE_EL1S.SRE is zero), Secure Group 0 and Non-Secure Group 1 interrupts are allocated to FIQ and IRQ respectively. ***Note***: in cases where Secure Group 1 interrupts might be sent by the Distributor, they must be treated as Secure Group 0 by the CPU interface (see section 4.1.3).

The table below defines how interrupts are allocated to FIQ and IRQ in systems supporting a single security state or when SRE is zero for the Secure EL1 interrupt regime:

Security state and Exception Level	Group 0	Group 1	
	Secure	Secure	Non-Secure
Any	FIQ	-	IRQ

Table 14: Interrupt Allocation for a single security state

In systems supporting two security states (i.e. when EL3 is present and GICD_CTLR.DS is zero) and when SRE is one for the Secure EL1 interrupt regime (i.e. when ICC_SRE_EL1S.SRE is one), Secure Group 0 interrupts are allocated to FIQ and the current interrupt regime is used to determine whether Secure Group 1 and Non-Secure Group 1 interrupts are allocated to IRQ or to FIQ.

Note: setting SRE to one for the Secure EL1 interrupt regime (i.e. when ICC_SRE_EL1S.SRE is one) means all physical interrupt regimes are using system registers and Secure Group 1 interrupts are supported. See section 4.4.6 for permitted SRE settings.

The table below defines how interrupts are allocated to FIQ and IRQ when EL3 is using AArch64:

Security state and Exception Level	Group 0	Group 1	
	Secure	Secure	Non-Secure
Secure EL0/1	FIQ	IRQ	FIQ
Non-Secure EL0/1/2	FIQ	FIQ	IRQ
EL3	FIQ	FIQ	FIQ

Table 15: Interrupt Allocation for two security states when EL3 is using AArch64

Note: because the assertion and deassertion of IRQ and FIQ may be affected by the processor's current exception level and security, then as part of the context synchronization that occurs as the result of the architectural execution of an exception-level-changing instruction (such as an exception return), the processor must ensure that IRQ and FIQ are both appropriately asserted/deasserted according to the establishment of the new exception level and security, and recognized as such by the processor's interrupt mechanism, prior to the completion of the exception-level-changing instruction.

The table below defines how interrupts are allocated to FIQ and IRQ when EL3 is using AArch32:

Security state and Exception Level	Group 0	Group 1	
	Secure	Secure	Non-Secure
EL3 / Secure EL0	FIQ	IRQ	FIQ
Non-Secure EL0/1/2	FIQ	FIQ	IRQ

Table 16: Interrupt Allocation for two security states when EL3 is using AArch32

When Secure Group 1 or Non-Secure Group 1 interrupts are allocated to FIQ, the behavior of ICC_IAR0_EL1 is modified to return a special ID indicating the target interrupt regime (1020 for Secure EL1 or 1021 for Non-Secure EL1/2) when the processor is executing at EL3 (see section 4.6.5).

When SRE is zero for the Secure EL1 interrupt regime (i.e. when ICC_SRE_EL1S.SRE is zero), interrupts allocated to FIQ will be signalled as IRQ if GICC_CTLR.FIQEn == ‘0’. ***Note***: the setting of GICC_CTLR.FIQEn does not affect access to system registers as defined in section 4.6.6.

4.6.4 Switching Security States for Group 1 Interrupts

The diagram below shows the sequence when a Group 1 interrupt for the other security state is asserted (SRE must be set for all physical interrupt regimes)

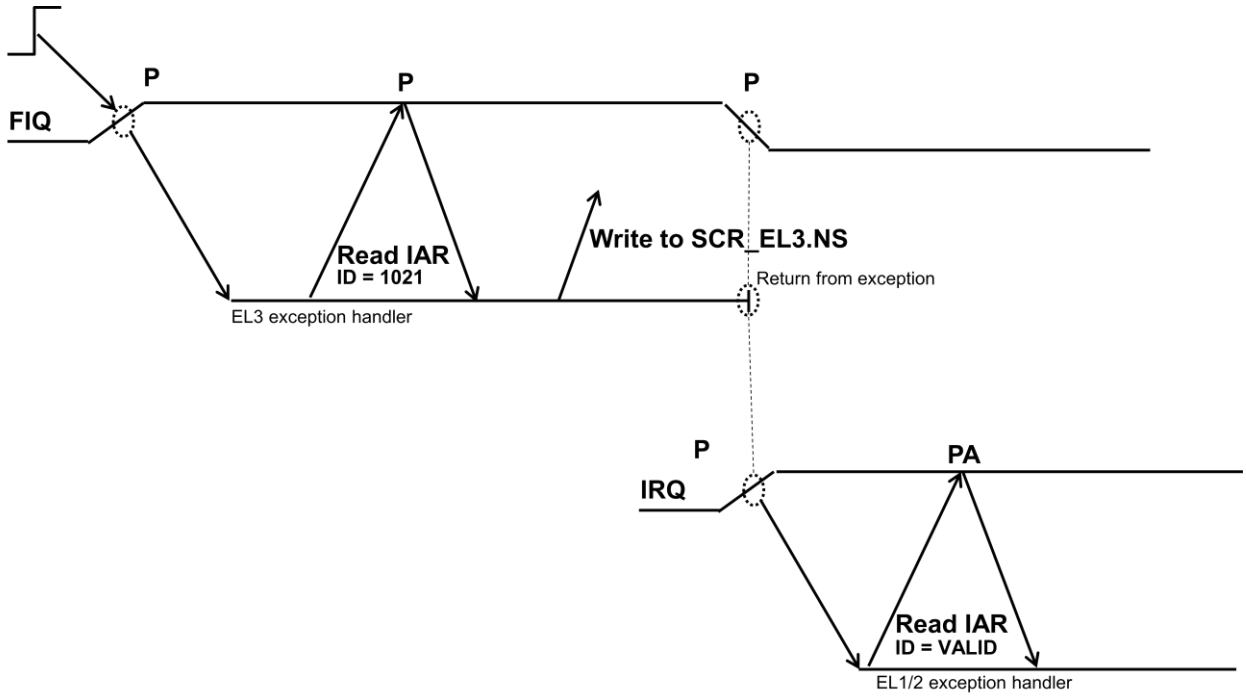


Figure 6: Handling Group 1 interrupts for the other security state

4.6.5 AArch32 Secure Software and Group Allocation

In systems where SRE is set for the Secure EL3 interrupt regime and SRE is zero for the Secure EL1 interrupt regime and Secure EL1 uses AArch32, the software using the Secure EL1 interrupt regime is assumed to be legacy software that expects its interrupt to be delivered as FIQ.

In such systems, only two interrupt groups are supported: Secure Group 0 and Non-Secure Group 1 and these are normally allocated to FIQ and IRQ respectively (by setting GICC_CTLR.FIQEn == '1').

Note: Secure Group 1 interrupts will be treated as Secure Group 0.

For such systems an additional "Routing Modifier" bit is provided (ICC_CTLR_EL3.RM) that modifies the behaviour of both ICC_IAR0_EL1 and ICC_IAR1_EL1:

- When ICC_CTLR_EL3.RM == '1', ICC_IAR0_EL1 when read at EL3 returns a special ID (1020) indicating the interrupt should be handled at Secure EL1.
- When ICC_CTLR_EL3.RM == '1', ICC_IAR1_EL1 when read at EL3 returns a special ID (1021) indicating the interrupt should be handled at Non-Secure EL1/2.

Note: when EL3 is using AArch32, the RM bit is accessed using ICC_MCTLR.RM.

This allows the secure EL3 monitor to control the routing of FIQ and IRQ such that interrupts are handled correctly. That is:

- When operating in the non-secure state, IRQ is routed to EL1/2 and will be handled correctly. FIQ is routed to EL3 and ICC_IAR0_EL1 returns a special ID indicating that an interrupt is pending that must handled at Secure EL1.
- When operating in Secure EL1, FIQ is routed to (secure) EL1 and will be handled correctly. IRQ is routed to EL3 and ICC_IAR1_EL1 (when read at EL3) returns a special ID indicating that an interrupt is pending that must handled at Non-secure EL1/2.
- When operating in Secure EL3, all interrupts are allocated to FIQ and software might choose to route FIQ to EL3 to allow software visibility of interrupts.

Note: this also applies when EL3 is using AArch32.

The diagram below shows how an interrupt for “legacy” secure software (when ICC_CTLR_EL3.RM is set to one) is handled. The example shown is when the processor is operating in the non-secure state, FIQ is routed to EL3 and IRQ is routed to EL1.

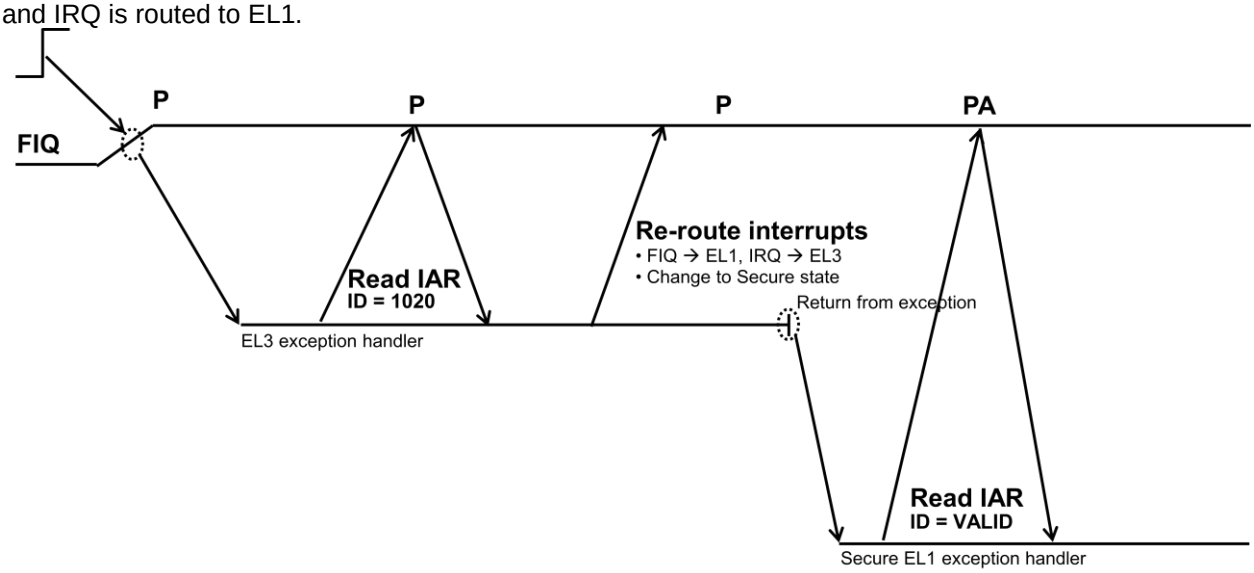


Figure 7: Handling interrupts when RM is one

4.6.6 Interrupt Routing and System Register Access

In ARMv8 systems, interrupt routing to an exception level is controlled by bits in SCR_EL3 and HCR_EL2:

- FIQ is controlled by SCR_EL3.FIQ, SCR_EL3.NS and HCR_EL2.FMO
- IRQ is controlled by SCR_EL3.IRQ, SCR_EL3.NS and HCR_EL2.IMO
- SError is controlled by SCR_EL3.EA, SCR_EL3.NS and HCR_EL2.AMO

The Exception Level from which the CPU interface EL1 system registers (that are associated with control and acknowledgement of interrupts) are accessible is also governed by this routing. The registers governed by this routing (and their associated interrupt exception) are shown below:

- **FIQ**: ICC_IAR0_EL1, ICC_EOIR0_EL1, ICC_HPPIR0_EL1, ICC_BPR0_EL1, ICC_AP0Rx_EL1, ICC_IGRPEN0_EL1. ***Note***: if ICH_HCR_EL2.TALL0 is one, then all non-secure EL1 accesses to these registers trap to EL2 (HYP_TRAP).
- **IRQ**: ICC_IAR1_EL1, ICC_EOIR1_EL1, ICC_HPPIR1_EL1, ICC_BPR1_EL1, ICC_AP1Rx_EL1, ICC_IGRPEN1_EL1. ***Note***: if ICH_HCR_EL2.TALL1 is one, then all non-secure EL1 accesses to these registers trap to EL2 (HYP_TRAP).
- **Common registers**: ICC_SGI0R_EL1, ICC_SGI1R_EL1, ICC_ASGI1R_EL1, ICC_CTLR_EL1, ICC_DIR_EL1, ICC_PMR_EL1 and ICC_RPR_EL1 (see Section 4.6.7 below). ***Note***: if ICH_HCR_EL2.TC is one, then all non-secure EL1 accesses to these registers trap to EL2 (HYP_TRAP).

The table below shows how these bits control accesses to the GICv3 system registers:

SCR_EL3		HCR_EL2	EL interrupt is routed to and physical state is accessible at	EL1 and EL2 behaviour
FIQ / IRQ / EA	NS	FMO / IMO / AMO		
0	0	X	EL1	
0	1	0	EL1	
0	1	1	EL2	EL1 accesses virtual register
1	0	X	EL3	EL1 accesses generate a MON_TRAP exception to EL3 when EL3 is using AArch64. EL3 accesses other than Monitor mode generate an UNDEF_cfg exception when EL3 is using AArch32.
1	1	0	EL3	EL1 and EL2 accesses generate a MON_TRAP exception to EL3 when EL3 is using AArch64. EL1, EL2 and EL3 accesses other than Monitor mode generate an UNDEF_cfg exception when EL3 is using AArch32.
1	1	1	EL3	EL1 accesses virtual register. EL2 accesses generate a MON_TRAP exception to EL3 when EL3 is using AArch64. EL2 and EL3 accesses other than Monitor mode generate an UNDEF_cfg exception when EL3 is using AArch32.

Table 17: CPU Interface System Register Access Behaviour

Note: for EL1 to access the full set of virtual registers, ICC_SRE_EL1 must first be set to one.

Note: MON_TRAP and UNDEF_cfg are as defined in Table 38: GIC Exception Behaviour below.

The CPU interface registers are described below. Unless otherwise stated, accesses to these registers are governed by Table 17: CPU Interface System Register Access Behaviour above.

4.6.7 Access to Common Registers

When SRE is set for the Non-Secure EL1 interrupt regime, Group 0 and Group 1 are separately virtualized. That is, a guest operating system operating at EL1 may control both physical interrupts and virtual interrupts.

For example, a guest operating system might be configured to handle:

- Virtual Group 0 interrupts by setting SCR_EL3.NS == '1' and HCR_EL2.FMO == '1'
- Physical Group 1 interrupts by setting SCR_EL3.NS == '1', SCR_EL3.IRQ == '0' and HCR_EL2.IMO == '0'

For most operations, this separate virtualisation is achieved by using different registers for handling Group 0 and Group 1 interrupts.

This cannot be done for registers that are common to both Group 0 and Group 1 interrupts, so the rules governing whether accesses to common registers (ICC_SGI0R_EL1, ICC_SGI1R_EL1, ICC_ASGI1R_EL1, ICC_CTLR_EL1, ICC_DIR_EL1, ICC_PMR_EL1 and ICC_RPR_EL1) are physical, virtual or trap are different:

- When ICH_HCR_EL2.TC == '1', non-secure accesses at EL1 trap to EL2
- When SCR_EL3.NS == '1' && (HCR_EL2.FMO == '1' || HCR_EL2.IMO == '1'), non-secure accesses at EL1 are virtual accesses.
    - Virtual accesses to the SGI generation registers ICC_SGI0R_EL1, ICC_SGI1R_EL1 and ICC_ASGI1R_EL1 always trap to EL2.
- Otherwise the lowest exception level it may be accessed at is the lowest of either:
    - That specified by SCR_EL3.FIQ, SCR_EL3.NS and HCR_EL2.FMO
    - Or, that specified by SCR_EL3.IRQ, SCR_EL3.NS and HCR_EL2.IMO

Note: it is expected that software will configure the GIC such that:

- ICH_HCR_EL2.TC == '1' (trap to EL2) when Group 0 and Group 1 configuration is asymmetric – i.e. EL1 accesses virtual state for one and physical state for the other. This allows the hypervisor perform the correct action.
- When the configuration is symmetric, ICH_HCR_EL2.TC == '0' and accesses to ICC_DIR_EL1 access the physical or virtual state for both Group 0 and Group 1.

4.6.8 Non-secure EL1 and Virtualisation

In GICv2, all accesses to the GIC were memory-mapped and interrupt virtualization was implemented by the hypervisor allowing a guest operating system, operating at non-secure EL1, to access either the GICC_* register or the GICV_* registers.

In GICv3, there might be a mixture of system register and memory-mapped access to the GIC, as defined in section 4.4.6. This has the following implications for how the hypervisor should manage interrupt handling by software operating at non-secure EL1:

- When EL2 is using memory-mapped access, non-secure EL1 **must** use memory-mapped access and the hypervisor must allow access by non-secure EL1 to either GICC_* or GICV_* registers, but not both.
    - When non-secure EL1 access to GICC_* registers is permitted, HCR_EL2.FMO and HCR_EL2.IMO must be zero. ***Note***: in this case, non-secure EL1 has no access to virtual state and there is no requirement for the hypervisor to program the GICH_* registers. Non-secure EL1 is also permitted to modify the GICv2 bypass controls in GICC_CTLR.
    - When non-secure EL1 access to GICV_* registers is permitted, HCR_EL2.FMO and HCR_EL2.IMO must be one. ***Note***: non-secure EL2 is permitted to modify the GICv2 bypass controls in GICC_CTLR.
    - ***Note***: if the hypervisor grants access to both GICC_* and GICV_* registers or does not program HCR_EL2.FMO and HCR_EL2.IMO correctly, this is a programming error and system behaviour might be unpredictable.
- When EL2 is using system register access and non-secure EL1 uses memory-mapped access, the hypervisor must follow the same rules as above. That is, it must allow access by non-secure EL1 to either GICC_* or GICV_* registers, but not both.
    - When non-secure EL1 access to GICC_* registers is permitted, HCR_EL2.FMO and HCR_EL2.IMO must be zero. ***Note***: in this case, non-secure EL1 has no access to virtual state and there is no requirement for the hypervisor to program the ICH_* registers. Non-secure EL1 is also permitted to modify the GICv2 bypass controls in GICC_CTLR.
    - When non-secure EL1 access to GICV_* registers is permitted, HCR_EL2.FMO and HCR_EL2.IMO must be one. ***Note***: non-secure EL2 cannot modify the GICv2 bypass controls in GICC_CTLR but might be permitted to modify ICC_SRE_EL2.DFB and ICC_SRE_EL2.DIB if EL3 is not present or if GICD_CTLR.DS is one.
    - ***Note***: if the hypervisor grants access to both GICC_* and GICV_* registers or does not program HCR_EL2.FMO and HCR_EL2.IMO correctly, this is a programming error and system behaviour might be unpredictable.
- When EL2 is using system register access and non-secure EL1 uses system register access, the hypervisor must not allow access by non-secure EL1 to GICC_* or GICV_* registers. The hypervisor might separately virtualise FIQ and IRQ:
    - When HCR_EL2.FMO is one, non-secure EL1 accesses to system registers controlling Group 0 interrupts are “virtual accesses”.
    - When HCR_EL2.IMO is one, non-secure EL1 accesses to system registers controlling Group 1 interrupts are “virtual accesses”.
    - When either of, or both of HCR_EL2.FMO and HCR_EL2.IMO are one, accesses to common system registers are “virtual accesses”.
    - ***Note***: non-secure EL2 cannot modify the GICv2 bypass controls in GICC_CTLR but might be permitted to modify ICC_SRE_EL2.DFB and ICC_SRE_EL2.DIB if EL3 is not present or if GICD_CTLR.DS is one.
    - See section 4.6.6.

Typically, a hypervisor will set all of HCR_EL2.{FMO, IMO, AMO} but might never generate certain virtual interrupt exceptions (for example, it might never generate virtual group 0 interrupts). When non-secure EL1 is virtualised in this way, it provides the equivalent of a (non-virtualised) physical environment where:

- HCR_EL2.{FMO, IMO, AMO} are all zero.
- SCR_EL3.{FIQ, IRQ, EA} are all zero.
- SCR_EL3.NS and GICD_CTLR.DS are all one.

4.6.9 Interrupt Routing and “Unused” Exception Levels

In GICv3 systems where a processor supports two security states, software might choose not to use EL3 and / or EL2.

Similarly, where a Distributor supports two security states, the system might not support two security states and software might set GICD_CTLR.DS to disable security.

The following configurations are supported:

Distributor	EL3 used	EL2 used	State	Description
One security state or, Two security states and GICD_CTLR.DS is zero	Yes	-	Both	The processor fully supports two security states and can receive interrupts for any interrupt group
Two security states and GICD_CTLR.DS is zero	No	Yes	NS	The processor is always non-secure and can only receive Non-Secure Group 1 interrupts Before EL3 is disabled and control is passed to EL2, EL3 software must: • Set ICC_SRE_EL3.SRE to one • Set ICC_SRE_EL3.Enable to one to allow EL2 to use system registers, if required. • Set ICC_SRE_EL3.DFB to one • Set SCR_EL3.FIQ to one • Set SCR_EL3.IRQ to zero • Set SCR_EL3.NS to one • Set ICC_IGRPEN0_EL1.Enable to zero to disable signaling of Group 0 interrupts to this processor • Set (secure) ICC_IGRPEN1_EL1.Enable to zero to disable signaling of Secure Group 1 interrupts to this processor ***Note***: any interrupts required by this processor must either be programmed to be Non-Secure Group 1 prior to disabling EL3 or (for SPIs and LPIs) must be programmed to be Non-Secure Group 1 by another processor.

Two security states and GICD_CTLR.DS is zero	No	No	NS	The processor is always non-secure and can only receive Non-Secure Group 1 interrupts Before EL3 and EL2 are disabled and control is passed to Non-Secure EL1, EL3 software must: • Set ICC_SRE_EL3.SRE to one • Set ICC_SRE_EL3.Enable to one • Set ICC_SRE_EL3.DFB to one • If EL2 is present, set ICC_SRE_EL2.Enable to one • Set SCR_EL3.FIQ to one • Set SCR_EL3.IRQ to zero • Set HCR_EL2.IMO to zero • Set SCR_EL3.NS to one • Set ICC_IGRPEN0_EL1.Enable to zero to disable signaling of Group 0 interrupts to the processor • Set (secure) ICC_IGRPEN1_EL1.Enable to zero to disable signaling of Secure Group 1 to the processor ***Note***: any interrupts required by this processor must either be programmed to be Non-Secure Group 1 prior to disabling EL3 or (for SPIs and LPIs) must be programmed to be Non-Secure Group 1 by another processor.
Two security states and GICD_CTLR.DS is zero	No	No	S	The processor is always secure and can only receive Secure Group 0 and Secure Group 1 interrupts Before EL3 and EL2 are disabled and control is passed to Secure EL1, EL3 software must: • Set ICC_SRE_EL3.SRE to one • Set ICC_SRE_EL3.Enable to one • Set SCR_EL3.FIQ to zero • Set SCR_EL3.IRQ to zero • Set SCR_EL3.NS to zero • Set (non-secure) ICC_IGRPEN1_EL3.Enable to zero to disable signaling of Non-Secure Group 1 to the processor

One security state or, Two security states and GICD_CTLR.DS is set.	No	-	-	The distributor and all processors are always in a single security state and can receive Group 0 and Group 1 interrupts Before EL3 is disabled and control is passed to EL2, EL3 software must: • Set GICD_CTLR.DS to one if the GIC implementation provides support for two security states. • For all processors o Set ICC_SRE_EL3.SRE to one o Set ICC_SRE_EL3.Enable to one o Set SCR_EL3.FIQ to zero o Set SCR_EL3.IRQ to zero o Set SCR_EL3.NS to one

Figure 8: Supported "Unused" EL Configurations

4.6.10 Interrupt Routing and Processors where EL3 is Not Present

In GICv3 systems where the Distributor supports two security states, a processor might not implement EL3 and / or EL2 and the following configurations are supported:

Distributor	EL3 exists	EL2 exists	State	Description
Two security states and GICD_CTLR.DS is zero	Yes	-	Both	The processor fully supports two security states and can receive interrupts for any interrupt group ***Note***: this is the “normal” case where all exception levels are present
Two security states and GICD_CTLR.DS is zero	No	-	NS	The processor is always non-secure and can only receive Non-Secure Group 1 interrupts The processor must behave as if software had: • Set ICC_SRE_EL3.Enable to one to allow EL2 to use system registers, if required. • Set ICC_SRE_EL3.DFB to one • Set SCR_EL3.FIQ to one • Set SCR_EL3.IRQ to zero • Set SCR_EL3.NS to one • Set ICC_IGRPEN0_EL1.Enable to zero to disable signaling of Group 0 interrupts to this processor • Set (secure) ICC_IGRPEN1_EL1.Enable to zero to disable signaling of Secure Group 1 interrupts to this processor This means the GIC CPU interface will report: • EN0S as always zero (see section 7.4.7). • EN1S as always zero (see section 7.4.7). • NS as always one (in all interface packets; see section 7.4).
Two security states and GICD_CTLR.DS is zero	No	No	S	The processor is always secure and can only receive Secure Group 0 and Secure Group 1 interrupts The processor must behave as if software had: • Set ICC_SRE_EL3.Enable to one • Set SCR_EL3.FIQ to zero • Set SCR_EL3.IRQ to zero • Set SCR_EL3.NS to zero • Set (non-secure) ICC_IGRPEN1_EL3.Enable to zero to disable signaling of Non-Secure Group 1 to the processor This means the GIC CPU interface will report: • EN1NS as always zero • NS as always zero

One security state, or Two security states and GICD_CTLR.DS is one	No	-	-	The distributor and all processors are always in a single security state and can receive Group 0 and Group 1 interrupts All processor must behave as if software had: • Set ICC_SRE_EL3.Enable to one • Set SCR_EL3.FIQ to zero • Set SCR_EL3.IRQ to zero • Set SCR_EL3.NS to one This means all GIC CPU interfaces will report: • EN1S as always zero • NS as always one

Figure 9: Configurations when EL3 not present

4.6.11 Disabling Security

To disable security, secure software must:

- Disable all interrupt groups in the Distributor
- Ensure no physical interrupts are pending or active
- Write to GICD_CTLR to set DS to one
- Wait until GICD_CTLR.RWP is zero.

If this sequence is not followed, system behaviour is unpredictable.

Once DS is set to one it can only be set to zero by reset.

After power on, the value of GICD_CTLR.DS is communicated to a processor by the affinity level 0 re-distributor. See section 4.10.6.

Note: when security is disabled, control over the bypass disable bits (ICC_SRE_EL3.{DFB, DIB}; see section 5.7.40) is handed over to non-secure software. If secure software wishes to prevent non-secure software from using the bypasses, it must disable them before disabling security by writing to the appropriate disables in GICC_CTLR (see section 5.6.18).

4.6.12 Other Effects of Disabling Security

In addition to those specified above, setting GICD_CTLR.DS to one has the following effects:

- The memory-mapped interface behaves as defined for GICv2 ***without the security extensions***
- The system register interface is unaffected, except as noted below:
    - ICC_CTLR_EL1.PMHE and ICC_CTLR_EL1.CBPR are writeable at EL1/EL2. See section 5.7.33.
    - The bypass disable bits (DFB and DIB) in ICC_SRE_EL1 and ICC_SRE_EL2 behave as if EL3 is not present. That is, they are writeable at the highest implemented exception level below EL3.
- ***Note***: this ensures that Group 0 interrupts can be handled in non-secure interrupt regimes
- Section 4.1.3 specifies the effects of GICD_CTLR.DS on Interrupt Grouping.
- Section 4.6.3 specifies the effects of GICD_CTLR.DS on the allocation of interrupts to FIQ and IRQ.
- GICR_NSACR checks on SGIs always pass. See section 5.7.29.

Note: in processors with EL3, Group 0 interrupts cannot be successfully handled by non-secure software unless the Distributor does not support security or Distributor security has been disabled (i.e. GICD_CTLR.DS == '1'). This means SCR_EL3.FIQ must be programmed to one in processors with EL3 whenever the Distributor supports

security, otherwise system behaviour is unpredictable. Secure-only processors can set SCR_EL3.FIQ to zero, regardless of the value of GICD_CTLR.DS.

4.7 System Error Interrupt and External Abort Support

The GICv3 architecture provides a mechanism to signal System Errors to the processor. The system errors can be signaled to the GIC by asserting edge sensitive error pins on each processor in a GICv3 implementation.

To this end, the GICv3 architecture specifies the ISS field used for system errors (exception class 0x2F) in the AArch64 Exception Syndrome register of the ARMv8 processor that implements GICv3.

For system errors reported in AArch32, system errors are reported using asynchronous data aborts.

4.7.1 System Error Handling in AArch32

When the SError exception is routed to an interrupt regime that uses AArch32, a value is provided in DFSR.FS field to distinguish the source:

- Asynchronous parity error on memory access. This value is used when an REI pin is asserted (see section 4.7.3 below).
- Asynchronous external abort. This value is used when reporting all other system errors.
- ***Note***: an implementation might choose to use bits [15:14] of DFSR to record additional implementation defined information about the system error (for example, they might be used to record data about the attribution and containability of system errors).

4.7.2 System Error Handling in AArch64

When the SError exception is routed to an interrupt regime that uses AArch64, a value is provided in the appropriate ESR_ELx to distinguish the source as defined in section 3.11.26 of [4]. That is, bits [31:26] contain an Exception Class that is set to 0x2F and a 25 bit instruction specific syndrome (ISS) in bits [24:0].

The ISS for system errors must follow the format below:

- Bits [24]. Syndrome Valid.
- Bits [23:22]. System Error Source.
    - 0b00. Processor Specific Error. System Error Interrupts generated internally by the processor, including asynchronous External Aborts returned as a response to a memory read or write with the data response.
    - 0b01. System Specific Error. System Error Interrupts generated internally by the GIC CPU interface or externally by asserting a pin on the processor.
    - 0b10 - 0b11. Reserved.
- Bits [21:0]. Source Specific Syndrome.
    - For Processor Specific Errors, this field is IMPLEMENTATION DEFINED
    - For System Specific Errors, this field is IMPLEMENTATION DEFINED. For ARM implementations, this field is defined in section 4.7.3.

4.7.3 System Error Pins for ARM Implementations

In ARM implementations, two interface pins per processor are provided for reporting System Errors:

- **SEI**. This signal is used for reporting Asynchronous External Aborts. If this signal is asserted, a system error exception will be generated. If the target exception level uses AArch64 the syndrome value in the appropriate ESR_ELx will be:
  - Bits [31:26]. EC == 0x2F.
  - Bits [25]. IL == '1'.
  - Bits [24]. Syndrome Valid == '1'.
  - Bits [23:22] System Error Source == 0b01
  - Bits [21:1]. 0x00000.
  - Bit [0]. Reason == '0' (External Abort)
  - ***Note***: if the target exception level uses AArch32, the value of DFSR.FS will indicate "Asynchronous external abort" (0b10110 when using short descriptors; 0b010001 when using long descriptors).
- **REI**. This signal is used for reporting Asynchronous RAM Errors. If this signal is asserted, a system error exception will be generated. If the target exception level uses AArch64 the syndrome value in the appropriate ESR_ELx will be:
  - Bits [31:26]. EC == 0x2F.
  - Bits [25]. IL == '1'.
  - Bits [24]. Syndrome Valid == '1'.
  - Bits [23:22] System Error Source == 0b01
  - Bits [21:1]. 0x00000.
  - Bit [0]. Reason == '1' (RAM Error)
  - ***Note***: if the target exception level uses AArch32, the value of DFSR.FS will indicate "Asynchronous parity error on memory access" (0b11000 when using short descriptors; 0b011001 when using long descriptors).

See section 7.1.1 for details of these signals and their polarity.

4.7.4 System Error Interrupt to the Processor

On taking an SEI when the target interrupt regime uses AArch32, the accumulated syndrome held in the CPU interface will be simplified to "Asynchronous parity error on memory access" or "Asynchronous external abort" by the processor and will be presented as DFSR.FS as defined in section 4.7.1.

On taking an SEI when the target interrupt regime uses AArch64, the accumulated syndrome held in the CPU interface will be presented in ESR_ELx and depends on the System Error Source (see above) in bits [24:23]:

- Bits [31:26]. EC. Must be 0x2F.
- Bits [25]. IL. Must be set to '1'.
- Bits [24]. Syndrome Valid. Must be set to '1'.
- Bits [23:22] == 0b00
    - Bits [21:0]. Implementation Defined.
- Bits [23:22] == 0b01
    - Bits [21:0]. Implementation Defined.

4.7.5 Locally Generated System Error Interrupts

GICv3 optionally supports locally generated system error interrupts by the CPU interface logic. Such errors might be associated with a physical interrupt or with a virtual interrupt.

Support for locally generated SEIs associated with physical interrupts is discoverable from ICC_CTLR_EL1.SEIS (see section 5.7.33) or ICC_CTLR_EL3.SEIS (see section 5.7.34). When supported, any such SEIs are reported using the SError exception.

Support for locally generated SEIs associated with virtual interrupts is discoverable from ICH_VTR_EL2.SEIS (see section 5.9.4). When supported, any such SEIs caused by accesses at non-secure EL1 are reported using either the SError exception or the ***virtual*** SError exception:

- If HCR_EL2.AMO is zero, locally generated SEIs from non-secure EL1 are reported using the SError exception. That is, they are taken in the interrupt regime to which SError is routed.
- If HCR_EL2.AMO is one, locally generated SEIs from non-secure EL1 are reported using the ***virtual*** SError exception. That is, they are taken at non-secure EL1.
- ***Note***: if HCR_EL2.AMO is one, a hypervisor may intercept locally generated SEIs by the GIC that would otherwise be taken at non-secure EL1 by setting ICH_HCR_EL2.TSEI to one (see section 5.9.2).

4.7.6 Locally Generated SEI Syndrome Values for ARM Implementations

In ARM systems that support locally generated SEIS, the syndrome presented in the ESR follows the format shown below:

- Bits [31:26]. EC == 0x2F.
- Bits [25]. IL == '1'.
- Bits [24]. Syndrome Valid == '1'.

- Bits [23:22] System Error Source == 0b01
- Bits [21:0]. Reason.

The table below defines the implemented syndrome values.

Reason	Name	Description	Section
0x00_1000	INVALID_INTERRUPT_IDENTIFIER	An invalid interrupt identifier was detected when accessing a system register.	5.7.9 5.7.10
0x00_1001	EOI0_HIGHEST_IS_G1	When writing to ICC_EOIR0_EL1, the interrupt identifier presented represents a Group 1 interrupt.	5.7.19 5.7.20
0x00_1002	EOI0_NO_INTS_ACTIVE	When writing to ICC_EOIR0_EL1, there were no active priorities.	5.7.19
0x00_1003	EOI0_NOT_HIGHEST_PRIORITY	When writing to ICC_EOIR0_EL1, the interrupt identifier presented is not the highest active priority.	5.7.20
0x00_1009	EOI1_HIGHEST_IS_G0	When writing to ICC_EOIR1_EL1, the interrupt identifier presented represents a Group 0 interrupt.	5.7.21 5.7.22
0x00_100A	EOI1_NO_INTS_ACTIVE	When writing to ICC_EOIR1_EL1, there were no active priorities.	5.7.21
0x00_100B	EOI1_NOT_HIGHEST_PRIORITY	When writing to ICC_EOIR1_EL1, the interrupt identifier presented is not the highest active priority.	5.7.22
0x00_1010	DIR_EOIMODE_NOT_SET	When writing to ICC_DIR_EL1, EOI mode was not set.	5.7.23 5.7.24

Table 18: Locally Generated SEI Syndrome Values

4.8 Locality-specific Peripheral Interrupts

To allow a large number of interrupt targets, a new class of Locality-specific Peripheral Interrupts (LPIs) has been added to GICv3.

These interrupts are forwarded to a single processor within the affinity hierarchy.

Note: 1 of N routing capability can be provided using SPIs.

LPIs are only supported using message based interrupts.

4.8.1 Interrupt Identifiers

In GICv3, the following ID spaces are supported:

- A shared ID space for SGIs, PPIs, and SPIs (IDs in the range 0 to 8191). ***Note***: IDs in the range 1024 – 8191 are used to support the banking by source CPU of SGIs when ARE is zero for a security state.
- An ID space for physical LPIs (IDs above 8191).

In GICv4, each software entity has a separate interrupt identifier space, where a given identifier always has the same meaning for a given software entity and different software entities can assign different meanings to the same identifier. Hence, GICv4 also supports:

- An ID space for virtual LPIs that is banked by virtual machine (IDs above 8191)

When coupled with an interrupt translation service that provides interrupt isolation between devices, translation of virtual to physical identifiers and (in GICv4) direct injection of virtual interrupts, this allows a software entity that "owns" a device to program that device directly without the need for costly hypervisor intervention.

4.8.2 State for Large Interrupt Identifier Spaces

In GICv3, the pending state of each interrupt identifier and the configuration data associated with each interrupt identifier is held in memory:

- A single contiguous region of physically addressed memory per re-distributor holds the pending state. ***Note***: because an LPI can only be directed to a single re-distributor at any given time, the need for coherent accesses to the pending state is avoided by providing the pending state per re-distributor
- A single contiguous region of physically addressed memory, which is accessed by all re-distributors, holds the configuration data.

In GICv3, interrupt identifiers are defined to be 32 bits or less. The maximum number of bits of interrupt identifier supported is implementation defined but implementations of the GIC CPU interface must implement at least 16 bits of interrupt identifier. The number of bits supported by an implementation is fully discoverable (see section 5.11 for details).

4.8.3 Properties of LPIs

LPIs always have the following properties:

- Fully configurable Enable. Each LPI might be enabled or disabled.
- Each LPI has a configurable Priority (eight bits, where bits [1:0] are always zero)
- Each LPI has a configurable Group.
- Edge triggered only. This removes the need for an "Active" state to be held.
    - ***Note***: this means that, regardless of the setting of EOImode for an interrupt regime, software must not explicitly Deactivate LPIs or a system error might be generated. The pending bit within the Re-Distributor is cleared on receipt of an Activate.
    - ***Note***: when the IAR returns a valid LPI ID, the LPI becomes active in the active priorities register and software must issue a write to EOI to clear the active priorities register, hence the CPU interface still requires an "Active" state for LPIs, even though this is not necessary within the re-distributor.

- Always routed to a specific processor that is the child of an affinity level 0 re-distributor.
- In GICv3, LPIs are allocated IDs in the range 8,192 - (2^{32} - 1) inclusive.

Note: LPIs (including virtual LPIs in GICv4) are only supported when ARE is set for the non-secure state and the ITS is enabled. See sections 5.12.5, 5.4.25 and 5.4.34.

Note: an ITS or Distributor implementation might choose to support any size of LPI identifier field up to and including 32 bits. For example, an implementation might choose to support 14 bits. Because IDs 0 to 8191 are used for other classes of interrupt, a 14 bit identifier provides support for 8192 LPIs. The number supported by software is configured writing a value to the "IDbits" field in GICR_PROPBASER (see section 5.4.23), subject to the maximum supported by the implementation (see section 5.11).

4.8.4 LPI Configuration Tables

The configuration of each LPI is determined by data held in memory. Each LPI is controlled by a byte of data:

- Bit [7:2]. Priority [7:2]. The priority of the LPI. ***Note***: Priority[1:0] is zero.
    - In systems supporting two security states (i.e. when GICD_CTLR.DS is zero), this priority value is shifted right by one such that LPIs may only have priorities in the lower half of the priority range {128 .. 254}.
    - In systems supporting one security state (i.e. when GICD_CTLR.DS is one), the raw priority value is used and LPIs may have any priority in the range {0 … 252}.
- Bit [1]. Group. The interrupt group of the LPI. RES1. ***Note***: LPIs are always non-secure Group 1
- Bit [0]. Enable. The enables for the LPI.

The configuration for an LPI with ID *N* is determined as shown below:

- The base address is determined from:
    - For physical LPIs, GICR_PROPBASER (see section 5.4.23)
    - For virtual LPIs from GICR_VPROPBASER (see section 5.4.32)
    - Which provide bits [47:16] of the physical address. Bits [15:0] of the base address are zero. ***Note***: an implementation might support less than 48 bits of physical address and unimplemented bits (starting at bit [47]) are RES0.
- The byte of data are held in memory address (base address + (*N* – 8192))

A re-distributor might cache this configuration data but it must ensure that:

- The effects of this caching are not visible to software except when reconfiguring an LPI, in which case an explicit invalidate command must be issued (e.g. an ITS INV command or a write to GICR_INVLPIR)
- ***Note***: this means Configuration data may not be speculatively cached unless hardware ensures the effects of this speculative caching are not visible to software.
- ***Note***: this means hardware must manage its caches automatically when moving interrupts

4.8.5 LPI Pending Tables

Each LPI Pending Table contains a single bit per LPI that contains the pending state of that LPI:

- 0b0. The LPI is not pending
- 0b1. The LPI is pending

The first 1kB of this table corresponds to other classes of interrupts (PPIs, SGIs and SPIs) and is expected to be used by implementations to hold a coarse-grained map of which LPIs, if any, are in the "pending" state.

During normal operation, the LPI Pending Table is maintained solely by a re-distributor. The copy in memory is made consistent with memory when GICR_WAKER.Sleep is written to one (see section 5.4.22).

4.8.6 Re-Distributor Registers

In a distributed implementation (see section 4.9.1) where an external ITS requires the ability to manage LPIs, GICv3 adds optional, programmable registers to each Re-Distributor at affinity level 0. See section 5.4 for details of these registers.

4.8.7 Generating and Clearing LPIs

The registers in each Re-Distributor that are used to generate LPIs follow the format of the registers defined in Section 4.1.9 above. To allow software to clear pending LPIs, a “clear” register is also required. The following registers are provided:

- GICR_SETLPIR. Set a physical LPI as “pending”. See section 5.4.25.
- GICR_CLRLPIR. Clear the pending state of a physical LPI. See section 5.4.26.

These registers are specified below and their offset in the Re-Distributor address map is shown in Table 23: Re-Distributor Physical LPI Register Address Map below.

4.8.8 Virtual LPIs, Virtual Processors and Residency

In GICv3, physical LPIs are virtualised by use of the GIC virtual interface (see sections 5.9 and 5.10) and the ITS MAPVI command (see sections 4.9.22 and 5.13.14). It is expected that:

- Each ID is mapped to a unique physical ID using the MAPVI command
- On receiving the unique physical ID corresponding to a virtual ID for particular guest operating system, the hypervisor will present the virtual ID to the guest operating system by programming the List Registers (see section 5.9.6).

GICv4 virtualises physical LPIs by providing support for a separate virtual ID space for each virtual machine and each virtual machine may have multiple virtual processors. This means that for each virtual machine, memory must be allocated for:

- A single contiguous region of physically addressed memory that contains the Virtual Configuration Table for that virtual machine.
- A single contiguous region of physically addressed memory per virtual processor that contains the Virtual Pending Table for that virtual processor.

Each virtual processor of a virtual machine is always associated with a single physical processor. All interrupts for a virtual processor are forwarded to the affinity level 0 re-distributor corresponding to the associated physical processor. A virtual processor is either:

- In-scope and executing (i.e. “resident”) on a physical processor, or,
- Out-of-scope and not executing (i.e. “not resident”) on a physical processor

A Virtual Processor is considered to be resident on a physical processor when the associated re-distributor is programmed with its Virtual Configuration Table (see section 5.4.32) and Virtual Pending Table (see section 5.4.33).

4.8.9 Configuring the Configuration and Pending Tables

The group, pending and priority state for LPIs is held in tables in memory. Each re-distributor shares a table containing the LPI configuration (group, priority and enables) and has a separate table containing the pending state for each LPI. Software configures these tables using the following registers:

- GICR_PROPBASER. A descriptor to the configuration table for physical LPIs. See section 5.4.23.
- GICR_PENDBASER. A descriptor to the pending table for physical LPIs. See section 5.4.24.
- GICR_VPROPBASER. In GICv4, a descriptor to the configuration table for the virtual LPIs for the current virtual machine. See section 5.4.32.
- GICR_VPENDBASER. In GICv4, a descriptor to the pending table for the virtual LPIs for the current virtual machine. See section 5.4.33.

Each “descriptor” contains a valid bit, the physical address of the start of the associated table and the size of the table (expressed as the number of bits of ID supported).

4.8.10 Initial Configuration of Physical LPIs

To initialise the configuration of physical LPIs before LPIs are enabled by setting GICR_CTLR.EnableLPIs (see section 5.4.7) has been written in each re-distributor, software must:

- Update the configuration data in the physical LPI configuration table in memory and ensure any updates are globally observable.
- Write the physical address and size (number of ID bits) of the configuration data table to GICR_PROPBASER (see section 5.4.23) in each re-distributor.
- Write the physical address of the pending table to GICR_PENDBASER (see section 5.4.24).

When physical LPIs are enabled by setting GICR_CTLR.EnableLPIs to one:

- The re-distributor must read any pending bits from the pending table and if any interrupts are pending, their configuration must be read from the properties table.
- Software must assume the ITS or re-distributor hierarchy might contain cached copies of the physical LPI configuration and must follow the sequence described in section 4.8.11 below.

If GICR_PROPBASER or GICR_PENDBASER are updated when GICR_CTLR.EnableLPIs is one and GICR_WAKER.Quiescent is zero, the effects are unpredictable.

4.8.11 Configuring Physical LPIs with Interrupts Enabled

To update the configuration (including changing the enable) of a physical LPI while interrupts are enabled software must:

- Update the configuration in the LPI configuration table memory. ***Note***: software must ensure any updates are globally observable.
- Ensure that the ITS and re-distributor hierarchy discards the old configuration from any caches. This may be done by:
    - If a small number of LPIs have been re-configured, issuing an INV commands for the re-configured LPIs (see section 5.13.17)
    - If a large number of LPIs have been re-configured, issuing an INVALL command (see section 5.13.19)
- ***Note***: interrupts using the old configuration might occur at any time before completion of the ITS command(s).

In distributed implementations (see section 4.9.1), the following re-distributor registers are provided to allow the ITS to manage the configuration of physical LPIs:

- GICR_INVLPIR. When written, the re-distributor will ensure that any cached data associated with the configuration of the specified LPI is invalidated (and might be reloaded from memory).. See section 5.4.29.
- GICR_INVALLR. When written, the re-distributor will ensure that any cached data associated with the configuration of LPIs is invalidated (and might be reloaded from memory). See section 5.4.30.

4.8.12 Configuring Virtual LPIs when the Virtual Processor is Not Resident

In GICv4, to initialise the configuration (including changing the enable) of a virtual LPI when the target virtual processor is not resident, software must

- Update the configuration in the virtual LPI configuration table memory and ensure any updates are globally observable.

When the target virtual processor is resident, software must assume the ITS or re-distributor hierarchy might contain cached copies of the virtual LPI configuration and must follow the sequence described in section 4.8.13 below.

4.8.13 Configuring Virtual LPIs when the Virtual Processor is Resident

In GICv4, to update the configuration (including changing the enable) of a virtual LPI when the target virtual processor is resident software must:

- Update the configuration in the virtual LPI configuration table memory
- Ensure that the ITS and re-distributor hierarchy discards the old configuration from any caches. This may be done by:
    - If a small number of LPIs have been re-configured, issuing an INV commands for the re-configured LPIs (see section 5.13.17)
    - If a large number of LPIs have been re-configured, issuing an VINVALL command (see section 5.13.26)
- ***Note***: interrupts using the old configuration might occur at any time before completion of the ITS command(s).

In distributed implementations (see section 4.9.1), the following re-distributor registers are provided to allow the ITS to manage the configuration of virtual LPIs:

- GICR_VINVLPIR. When written, the re-distributor will ensure the pending table in memory is updated for the specified ID and that the configuration the specified ID held in the cache are reloaded from the configuration table for a specified virtual processor. See section 5.4.40.
- GICR_VINVALLR. When written, the re-distributor will ensure the pending table in memory is updated for all IDs and that the configuration any IDs held in the cache are reloaded from the configuration table for a specified virtual processor. See section 5.4.41.

4.8.14 Changing the Residency of a Virtual Processor

In GICv4, to make a virtual processor “not resident” on a re-distributor, software must:

- Write GICR_VPENDBASER.Valid to zero.

In GICv4, to make a virtual processor “resident” on a re-distributor, software must:

- Ensure no other virtual processor is resident (as above)
- Write the physical address of the virtual configuration data table to GICR_VPROPBASER (see section 5.4.32).
- Write the physical address and size of the virtual pending table to GICR_VPENDBASER (see section 5.4.33).
- Write GICR_VPENDBASER.Valid to one. ***Note***: this might be combined with the write(s) that sets the address and size of the pending table.

4.8.15 Generating Virtual LPIs

Virtual LPIs may be generated in three distinct ways:

- In distributed GICv4 implementations (see section 4.9.1), the hypervisor might generate virtual LPIs by writing to the appropriate re-distributor register (GICR_VSETLPIR; see section 5.4.34).
- In GICv3 and GICv4, the hypervisor might generate a virtual LPI by writing to list registers (ICH_LRn_EL2; see section 5.9.6). ***Note***: a guest must use system registers to support virtual LPIs.
- In GICv4, an Interrupt Translation Service (see section 4.9) can generate virtual LPIs by writing to GICR_VSETLPIR (see section 5.4.34) of the target affinity level 0 Re-Distributor. ***Note***: generation of direct virtual interrupts using this mechanism is only supported in GICv4. Support for this feature is discoverable from the re-distributor type register (GICR_TYPER; see section 5.4.8).

Note: only guest operating systems that support GICv3 can receive virtual LPIs. That is, when the guest operating system is using system registers.

4.8.16 Virtual CPU Interface Operation with Virtual LPIs

In GICv4, when a GIC supports Virtual LPIs, the virtual CPU interface must send the highest priority pending virtual interrupt to the currently resident virtual processor. This highest priority virtual interrupt might be in the List Registers (if directly generated by the hypervisor) or it might be a Virtual LPI signaled directly from the affinity level 0 re-distributor.

To allow virtual interrupts to be set pending when a virtual processor is not resident, the address of a virtual pending and a physical LPI identifier must be provided when writing to the target re-distributor. This physical LPI is set as pending if the target virtual processor is not resident.

To support this, the GICR_VSETLPIR register (see section 5.4.34) is 64 bits and when generating a virtual LPI a target Virtual Pending Table address, a Physical LPI ID and Virtual LPI ID are provided in the data payload.

4.8.17 Software Actions for Swapping a Guest Operating System

In GICv3 and GICv4, the hypervisor must, at minimum, perform the following actions when it changes the guest operating system at non-secure EL1 on a given processor:

- Ensure the hypervisor is operating at EL2 and that no events can occur such that the outgoing guest is able to run at non-secure EL1
    - In GICv4, write to GICR_VPENDBASER with Valid set to zero. This ensures that any virtual LPIs are released by the CPU interface.
- Save the interrupt context of the outgoing guest operating system.
    - Save the contents of any List Registers that are either not in the “invalid” state (i.e. the List Register state is non-zero) or are in the “invalid” state and are signaling an EOI (i.e. the List Register state is zero, HW is zero and EOI is one). ***Note***: any List Registers not in the “invalid” state must be made “invalid” and EOI must be set to zero. See section 5.9.6.
    - Save the contents of ICH_AP0Rn_EL2
    - Save the contents of ICH_AP1Rn_EL2
    - Save the contents of ICH_VMCR_EL2
    - Save the contents of the non-secure copy of ICC_SRE_EL1. ***Note***: this must be done before HCR_EL2 is modified otherwise ICC_SRE_EL1.SRE might become treated as zero or one before the correct value has been saved.
    - Save the contents of HCR_EL2
- Restore the interrupt context of the incoming guest operating system.
    - Restore the value of HCR_EL2 for the incoming guest (see section 4.6.8)
    - Restore the value of the non-secure copy of ICC_SRE_EL1. ***Note***: this must be done after HCR_EL2 is restored otherwise ICC_SRE_EL1.SRE might be being treated as zero or one such that the correct value is not restored.
    - Restore the value of ICH_VMCR_EL2
    - Restore the value of ICH_AP1Rn_EL2
    - Restore the value of ICH_AP0Rn_EL2
    - Restore the value of the List Registers.
    - If the characteristics of the incoming guest operating system are unknown to the hypervisor, it might additionally set one or more of ICH_HCR_EL2.{TALL0,TALL1,TC} (see section 5.9.2) such that when the incoming guest attempts to change its configuration, it traps to the hypervisor, allowing the hypervisor to learn the guest characteristics.
- Ensure any events for the incoming guest operating system may now occur
    - In GICv4, if the incoming guest operating system uses system registers and LPIs, write to GICR_VPROPBASER with the address of the Virtual Configuration Table for the incoming guest.
    - In GICv4, if the incoming guest operating system uses system registers and LPIs, write to GICR_VPENDBASER with Valid set to one and the address of the Virtual Pending Table for the incoming guest. This ensures that any virtual LPIs for the incoming guest will now be taken once the guest is running at non-secure EL1.

4.9 Interrupt Translation Service

In GICv3, an Interrupt Translation Service (ITS) provides a simple software mechanism for re-targetting interrupts where:

- Peripherals (including PCIe endpoints) that support message based interrupts that require retargeting are programmed with the translation address of the ITS (see GITS_TRANSLATER in section 5.12.4)
- Each peripheral is programmed with an ID and the ITS determines whether this ID is a virtual ID or a physical ID
- When a peripheral generates an interrupt, the ID is written to the programmed address (see GITS_TRANSLATER in section 5.12.4)
- The ITS uses the Device ID to find an Interrupt Translation Table. ***Note***: the contents and format of an Interrupt Translation Table (ITT) are implementation defined. Software can discover the size of each entry from GITS_TYPER (see section 5.12.6).
- The ITS uses the ID to index this Interrupt Translation Table (ITT) and obtain an Interrupt Translation Entry (ITE)
- This ITE provides the ID of the Collection that the interrupt is a member of.
- The ITS uses the Collection ID to index a Collection Table and obtain a Collection Table Entry (CTE) that provides the routing of the interrupt.
- The ITE also provides details of how to translate the ID to a unique Physical ID which is presented to software on the processor obtained from the CTE. ***Note***: see section 4.9.3 below for direct injection of virtual interrupts added in GICv4.

Note: in systems without an ITS that support message based interrupts, it is relatively difficult to re-target interrupts as software must know the addresses of each individual peripheral's interrupt address and data registers.

The ITS is managed using commands issued to a (circular) command queue in memory. Commands are provided to enable software to re-target interrupts and to perform other operations (see section 4.9.7 for an overview of the commands provided).

Provision of an ITS is mandatory in GICv3 and GICv4 systems that support LPIs (see sections 5.3.1 and 5.4.8).

4.9.1 Supported Implementation Options

In both GICv3 and GICv4 systems, the following implementation options are supported:

- Monolithic with an integrated ITS. This option provides a simple solution for systems with a small number of processors and peripheral devices, where only the minimum set of re-distributor registers must be provided (and provision of the full set is optional).
- Distributed, with at least one ITS. This option provides a scalable solution for systems with a large number of processors and peripheral devices. All re-distributor registers must be provided.

4.9.2 Interrupt Collections

In both GICv3 and GICv4, the ITS considers all physical interrupts to be members of Interrupt Collections. This allows software to manage large numbers physical interrupts with a small number of commands rather than issuing commands per interrupt.

The data associated with an interrupt collection may either be held within the ITS or in external memory:

- The number of collections supported by an ITS where the data is held within the ITS is discoverable from GITS_TYPER.HCC (see section 5.12.6).
- The number of collections where the data is held in external memory is determined by:
    - Whether the ITS supports collections with data held in external memory (i.e. if a GITS_BASERn register has a "Type" field indicating "Interrupt Collections" (see section 5.12.12).

    - When the ITS does support such collections, the number supported is given by (GITS_BASERn.Size * 4096) / GITS_BASERn.EntrySize.
- Hence the total number supported is given by GITS_TYPER + the value encoded by the Size field from the appropriate GITS_BASERn (if any).
- When GITS_TYPER.HCC is non-zero, collections with identifiers in the range {0 .. GITS_TYPER.HCC - 1} are held within the ITS and collections with identifiers greater than GITS_TYPER.HCC are held in external memory (if supported)
- When GITS_TYPER.HCC is zero, an ITS ***must*** support collections in external memory and all collections are held in external memory.

4.9.3 Direct Injection of Virtual Interrupts

GICv4 adds the ability to directly inject virtual interrupts into guest operating systems without hypervisor involvement. This addition requires the following changes:

- A new set of ITS commands. See section 4.9.7 below.
- Provision of additional ITS tables (see section 4.9.24 below)
- In distributed implementations, a new set of re-distributor registers. See sections 5.4.32 to 5.4.42 inclusive.

When an interrupt is mapped as a direct virtual interrupt (using the VMAPI command; see section 5.13.24), the ITS will always mark it as pending in the appropriate virtual pending table. However, the target virtual processor might not be resident and hypervisor action might be needed to make the virtual processor resident so that it can handle the interrupt. To support this, GICv4 generates a physical “doorbell” interrupt if it detects the target virtual processor is not resident.

4.9.4 Address and Data Programming for Message Based Interrupts

When using an ITS, peripherals and consolidators contain a 64 bit address register and a 32 bit data register for each possible interrupt that contains the address of the ITS translation register (GITS_TRANSLATER; see section 5.12.4) and the ID. The format of the address register is shown below:

- Bits [63:48]. Reserved. RES0.
- Bits [47:16]. Base address of ITS. This must be the address of the ITS Translation Register page. See section 5.12.3.
- Bits [15:0]. Reserved. RES0.

The format of the data register is shown below:

- Bits [31:0]. ID. The identifier that will be used as the input to the interrupt translation process. ***Note***: in GICv3, it is expected that IDs for virtualized software will be mapped using a MAPVI command (see section 5.13.14) and will result in the presentation of a unique physical interrupt ID to the hypervisor.
    - ***Note***: it is implementation defined how many identifier bits are implemented, though at least 16 bits must be implemented. Unimplemented bits are RES0. See section 5.11 for details of how the number of implemented bits can be discovered.

4.9.5 Device Interrupt Isolation

Together, the ITS and the SMMU ensure that IO devices are isolated from each other and can only affect memory belonging to their owning software.

The SMMU ensures that devices can only access memory permitted by the owning software.

The ITS ensures that devices can only generate interrupts permitted by the owning software. To ensure this, the ITS provides:

- Separate register pages for programming by the processor and interrupt generation by devices
- A mapping from the device identifier to a set of translation tables for the owning software entity.
- Translation tables that ensure that only a permitted set of interrupts can be generated by devices belonging to a software entity.

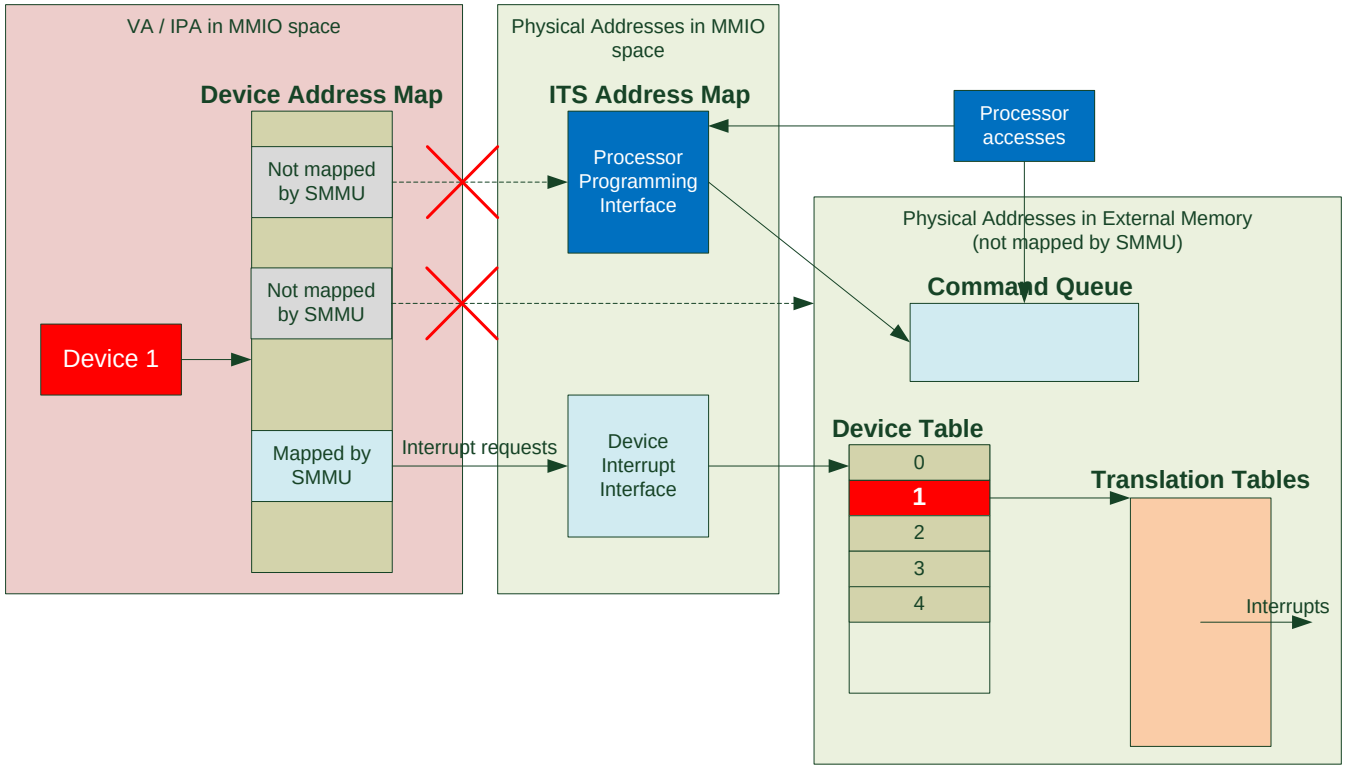


Figure 10: Device Isolation

The ITS provides a command interface for managing the device table and the translation tables. This interface is described below.

4.9.6 Device Identifiers

The SMMU must preserve a one-to-one mapping of device identifiers and a unique device identifier must be presented on writes to GITS_TRANSLATER (see section 5.12.4).

The values of device identifiers are implementation specific and discovery of these values is beyond the scope of the GIC architecture. The maximum size of device identifier supported by an ITS is discoverable from GITS_TYPER.Devbits (see section 5.12.6).

4.9.7 Overview of ITS Commands

The ITS is controlled by a sequence of commands from software. The table below provides a summary of the commands provided, both for normal operation and for maintenance of the ITS:

Operational Commands	Physical Command	Hypervisor Command (GICv4 only)	Description
Move Interrupt	MOVI	VMOVI	Move an individual interrupt to a different target processor
Move Processor	-	VMOVP	Move a virtual processor to a different physical processor
Move All	MOVALL		Move all interrupts from one physical processor to another
Generate Interrupt	INT		Generate an interrupt with a specified identifier.
Clear an Interrupt	CLEAR		Clear the pending state of a specified identifier
Synchronise	SYNC	VSYNC	Ensure all previous operations for a re-distributor or virtual processor have completed
Maintenance Commands	**Physical Command**	**Hypervisor Command (GICv4 only)**	**Description**
Map device	MAPD		Map a device to an interrupt translation table Multiple devices may share a single ITT or may have separate ITTs
Map collection	MAPC	-	Map a collection of interrupts to a physical processor
Map processor	-	VMAPP	Map a virtual processor to a physical processor and a virtual pending table
Map virtual interrupt	MAPVI	VMAPVI	Map an input identifier to an interrupt identifier that will be presented to the target software and an interrupt collection.
Map interrupt	MAPI	VMAPI	Map an interrupt to an interrupt collection
Clean interrupt	INV		Ensure any caching associated with an interrupt is consistent with memory
Clean all	INVALL	VINVALL	Ensure any caching associated with a collection or virtual processor is consistent with memory
Discard interrupt	DISCARD		Discard an interrupts received from a device with a specified ID.

Figure 11: An overview of the ITS commands

Note:

- Virtual Commands are only provided by GICv4.
- Commands are processed by the ITS in command queue order.
- Completion of a command does not guarantee visibility of the effects of a command. To ensure the effects of all previous commands are visible a SYNC or VSYNC command must be issued.

4.9.8 The ITS Command Queue

In normal operation, the ITS is controlled using a command queue in memory. This queue is defined by:

- A register that contains the base address and size of the queue. The queue is circular and wraps at base address + size. See section 5.12.9.
- A write offset register (GITS_CWRITER) that contains the offset from the base where software will write the next command. See section 5.12.10.
- A read offset register (GITS_CREADR) that contains the offset from the base where the ITS will read the next command. As the ITS processes each command, it updates the register taking account of the wrap address. See section 5.12.11.
- Commands are processed sequentially and the ITS may update the read pointer to the next command once the command has been read from the queue.
    - ***Note***: to ensure the result of commands are globally visible, an additional synchronisation command must be issued (see sections 5.13.22 and 5.13.27) and software must wait for completion of this command.
    - ***Note***: the value obtained by reading GITS_CREADR can only be used to guarantee the ITS has fully completed all required actions for the SYNC and VSYNC commands (see sections 5.13.22 and 5.13.27).
- All commands comprise 32 bytes.
- The queue is empty when the write offset is equal to the read offset.
- The queue is full when the write offset points at the command before the current read offset.

Note: all addresses are physical addresses.

This is shown in the diagram below:

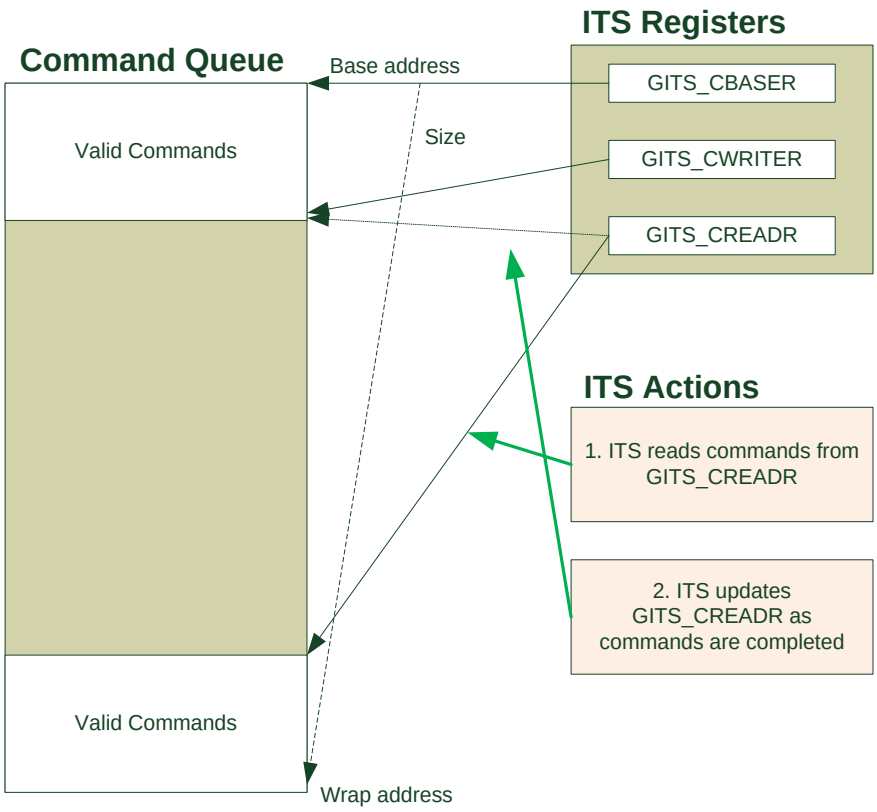


Figure 12: The ITS Command Queue

Where:

- Base address is always 64kB aligned and size is expressed as a multiple of 4kB.
- Wrap address is always 4kB aligned and is obtained from base address + (size * 4kB)

4.9.9 Adding New Commands to the Queue

The diagram below illustrates how software adds commands to the queue and can wait for completion.

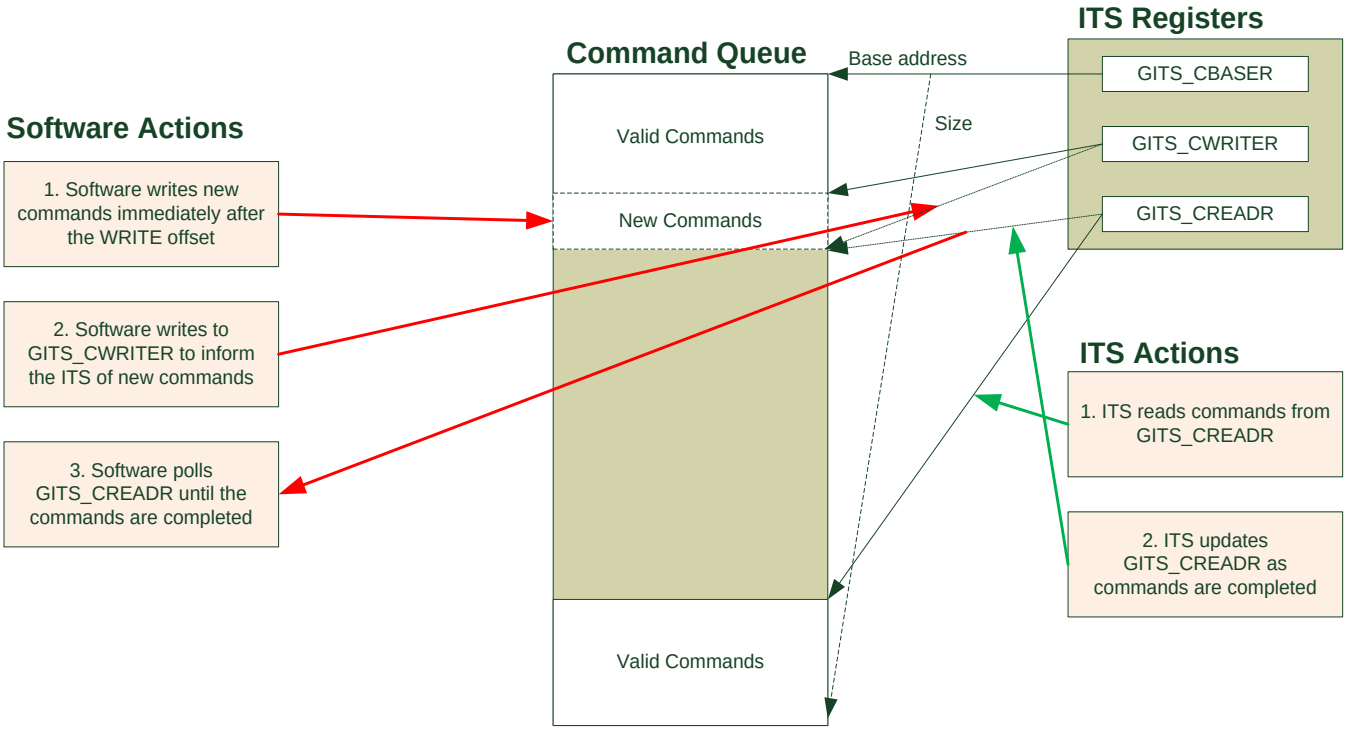


Figure 13: Adding Commands to the Command Queue

Notes:

- Software can write multiple commands to the queue before updating the write offset.

Instead of polling for completion of commands, software can request notification of completion by interrupt. This is described in section 4.9.10 below.

4.9.10 Notification of Command Completion by Interrupt

Instead of polling for completion of commands, software can request notification of completion of a set of commands by adding a “generate interrupt” command after a set of commands. Because commands are processed sequentially, this guarantees the previous commands have been completed before the notification interrupt is generated.

The diagram below illustrates this:

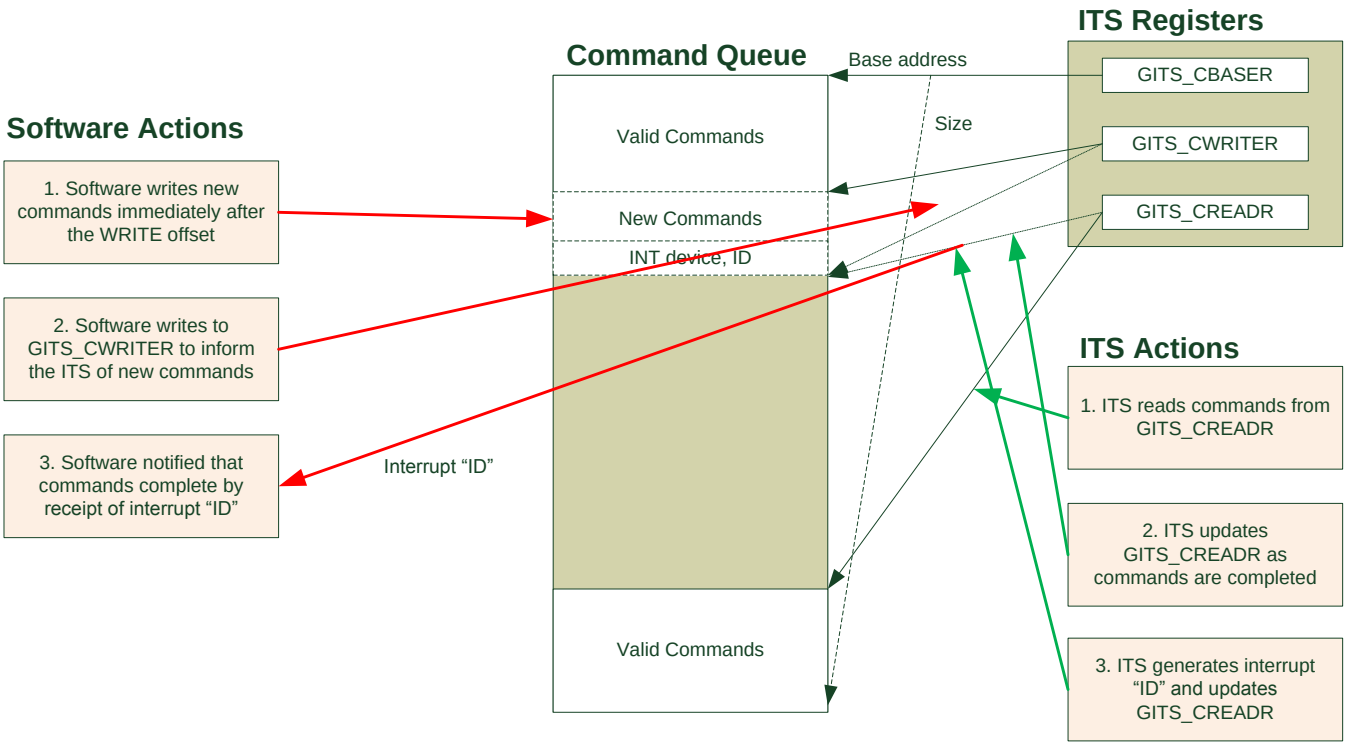


Figure 14: Notification of Command Completion by Interrupt

Notes:

- The notification interrupt can have any ID
- A “device” number must be provided to allow the ITS to find an appropriate ITT which specifies how the interrupt is to be treated and the destination of the interrupt
- It is expected that software might map a “synthetic” device (that does not correspond to a physical device) in order to avoid potential conflict with interrupts for physical devices.

4.9.11 Virtualising the ITS Command Queue

In GICv3 and GICv4, no explicit hardware support is provided for virtualising the ITS. The expected model is that a virtualized guest operating system will write to what it considers to be an ITS command queue and will either poll GITS_CREADR or expect notification by interrupt.

The hypervisor must ensure that virtualized guest accesses to the (virtual) ITS registers trap and that the hypervisor performs appropriate actions.

Note: it is expected that the guest operating system writes to its command queue in normal memory and that the hypervisor reads this guest command queue and inserts appropriate commands into the physical command queue.

The diagram below illustrates the expected interactions:

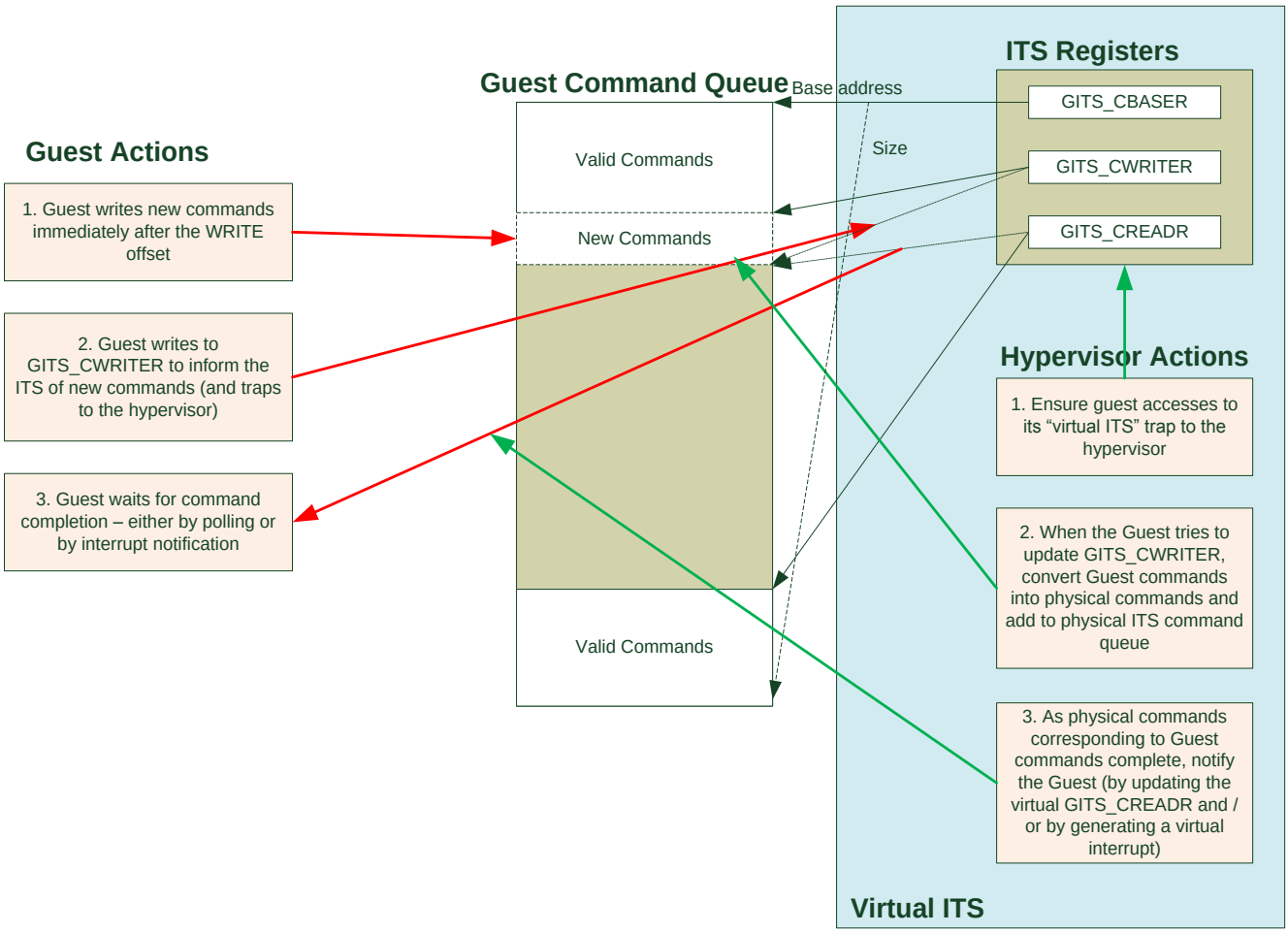


Figure 15: The Virtual ITS

Care has been taken to ensure that guest operating system commands can be easily virtualized and the transformation of guest commands to physical commands for both GICv3 and GICv4 is addressed below.

4.9.12 Notional ITS Table Structure

The diagram below shows a notional ITS table structure. This structure is used in the description of operation of the ITS commands and is illustrative only. An implementation may choose any table structure that achieves the same overall functionality.

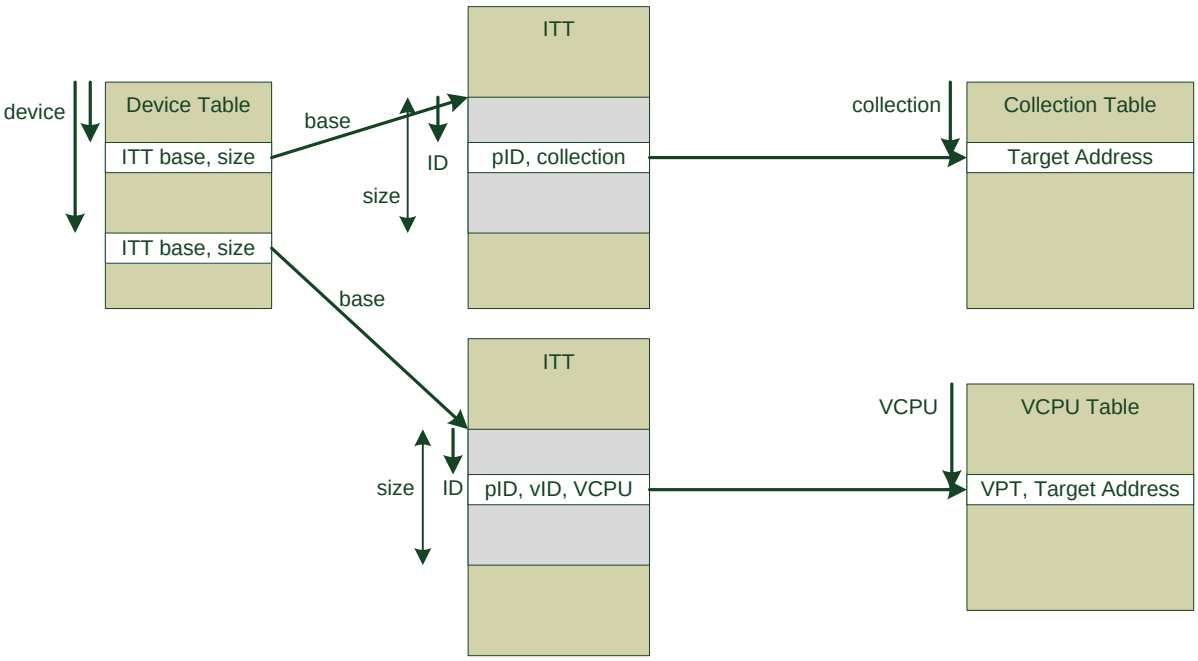


Figure 16: Notional ITS Table Structure

4.9.13 Introduction to Interrupt Mapping

Interrupt translation requests to the ITS can be mapped to generate interrupts in three ways, as described in the table below:

Mapping Command	Section	Usage
MAPD device, ITT base, size	5.13.11	Map a device to an interrupt translation table.
MAPC collection, TA	5.13.12	Map an interrupt collection to a target re-distributor
VMAPP VCPU, TA, VPT	5.13.23	Map a virtual processor to a virtual pending table and a target re-distributor. Redirected using a VMOVP VCPU, new TA. ***Note***: GICv4 only.
MAPI device, ID, collection	5.13.13	Map an interrupt to an interrupt collection. Interrupt identifier ID will be presented to the target software. Redirected using MOVI device, ID, new collection.
MAPVI device, ID, pID, collection	5.13.14	Map an input identifier to a physical interrupt and an interrupt collection. Interrupt identifier pID will be presented to the target software. Redirected using MOVI device, ID, new collection.
VMAPI device, ID, pID, VCPU	5.13.24	Map a virtual interrupt to a virtual processor and assign a physical doorbell interrupt for when the virtual processor is not resident. Interrupt identifier ID will be presented to the virtual processor. Interrupt identifier pID will be presented to the hypervisor when the virtual processor is not resident. Redirected using VMOVI device, ID, new VCPU. ***Note***: GICv4 only.
VMAPVI device, ID, vID, pID, VCPU	5.13.25	Map an input identifier to a virtual interrupt on a virtual processor and assign a physical doorbell interrupt for when the virtual processor is not resident. Interrupt identifier vID will be presented to the virtual processor. Interrupt identifier pID will be presented to the hypervisor when the virtual processor is not resident. Redirected using VMOVI device, ID, new VCPU. ***Note***: GICv4 only.

Figure 17: Introduction to Interrupt Mapping

Note:

- If a VMOVI command is received for an interrupt request mapped using MAPI or MAPVI, then a System Error might be generated.
- If a MOVI command is received for an interrupt request mapped using VMAPI or VMAPVI, then a System Error might be generated.

4.9.14 Mapping Devices to Interrupt Translation Tables

Devices are expected to be mapped to Interrupt Translation Tables (using the MAPD command) such that each interrupt request from each device maps to a unique Interrupt Translation Table Entry (ITE). In addition, software must ensure that:

- Each device maps to a dedicated section of an ITT (i.e. no other device is mapped to that section of ITT), or
- When multiple devices share a section of an ITT, each ITE within that section is associated with one device only and that all ITS commands that affect an ITE are performed using the associated device.

Note: to simplify software, it is recommended that each device maps to a dedicated section of an ITT.
Note: ITS accesses to an ITT use the same shareability and cachability attributes as specified for the Device Table (see section 5.12.12).

4.9.15 Mapping of Interrupts to Interrupt Collections

Interrupts are expected to be mapped to collections such that:

- Each collection corresponds to a single physical processor in a fully powered-up system
- When a processor is powered off, all its interrupts can be moved to a different processor using a single MAPC command (see section 5.13.12) and a single MOVALL command (see section 5.13.18)
- When a processor is powered on, software can map a new collection to this processor (see the MAPC command in section 5.13.12) and can migrate individual interrupts to this collection (see the MOVI command in section 5.13.15).
- ***Note***: this means that in a system with ***N*** physical processors, at least ***N+1*** collections must be supported to allow interrupt migration and software must ensure there is always an unused collection available when a processor is powered off

4.9.16 Target Addresses

Many ITS commands require a Target Address that specifies a re-distributor that an interrupt identifier (or set if identifiers) is to be associated with. An implementation may choose two different formats for this Target Address:

- When GITS_TYPER.PTA is one (see section 5.12.6), these Target Addresses must correspond to the base address in memory of the required re-distributor.
- When GITS_TYPER.PTA is zero, Target Addresses must correspond to a unique “processor number”. The mapping between this processor number and the processor affinity value is discoverable from GICR_TYPER.ProcessorNumber (see section 5.4.8).

4.9.17 Re-mapping and Un-mapping Interrupts

Interrupts can be re-mapped or un-mapped when the ITS is enabled by performing the following sequence of operations:

- Ensure the old target interrupt is disabled
- Issue a mapping command (MAPI, MAPVI, VMAPI or VMAPVI) or an un-mapping command (DISCARD)
- Wait for the command to be completed
- Following completion of the command, no more interrupts will be delivered to the old target
- ***Note***: software might still need to handle a pending interrupt to the old target.

4.9.18 Re-mapping and Un-mapping Devices

Devices can be re-mapped or un-mapped when the ITS is enabled by performing the following sequence of operations:

- Ensure all interrupts for the old mapping are disabled
- Issue a mapping command (MAPD; see section 5.13.11) or an un-mapping command (MAPD with “Valid” set to zero)

- Wait for the command to be completed
- Following completion of the command, no more interrupts will be delivered to the old mapping

Note: software might still need to handle pending interrupts for the old mapping.

4.9.19 Re-mapping and Un-mapping Interrupt Collections

Interrupt collections can be re-mapped or un-mapped when the ITS is enabled by performing the following sequence of operations:

- Issue a mapping command (MAPC; see section 5.13.12) or an un-mapping command (MAPC with “Valid” set to zero)
- If an interrupt collection has been re-mapped, issue a MOVALL command (see section 5.13.18).
- If an interrupt collection has been re-mapped, issue a SYNC command (see section 5.13.22).
- Wait for the last command issued to be completed.
- Following completion of the last command, no more interrupts will be delivered to the old mapping

4.9.20 Re-mapping and Un-mapping Virtual Processors

Virtual processors can be re-mapped or un-mapped when the ITS is enabled by performing the following sequence of operations:

- Ensure all interrupts for the old mapping are disabled.
- Issue a mapping command (VMAPP; see section 5.13.23) or an un-mapping command (VMAPP with “Valid” set to zero)
- Wait for the command to be completed
- Following completion of the command, no more interrupts will be delivered to the old mapping.

4.9.21 Usage of ITS Commands without Virtualisation

It is expected that, in both GICv3 and GICv4 systems, the ITS will be used by software that is not virtualized. The table below shows the available ITS commands and their usage:

Command	Section	Usage
MAPD device, ITT PA, ITT size	5.13.11	Map a physical device to an ITT. ***Note***: this allocates memory to the ITS that is used for an ITT. See section 5.12.6 for discovery of the size of each entry.
MAPC collection, TA	5.13.12	Map an interrupt collection to a target processor
MAPI device, pID, collection	5.13.13	Map a physical interrupt to an interrupt collection. Interrupt identifier “pID” is presented to the target processor. ***Note***: used when the input identifier programmed into the device is identical to the interrupt identifier expected by software.
MAPVI device, ID, pID, collection	5.13.14	Map an input identifier to a physical interrupt to an interrupt collection. Interrupt identifier “pID” is presented to the target processor. ***Note***: used when the input identifier programmed into the device is different to the interrupt identifier expected by software, for example where a device only has access to a section of a single large ITT.
MOVI device, pID, collection	5.13.15	Re-direct new interrupts with identifier “pID” to an interrupt collection
DISCARD device, pID	5.13.16	Discard interrupt requests with identifier “pID”
INV device, pID	5.13.17	Clean any caches associated with identifier “pID”
INVALL collection	5.13.19	Clean any caches associated with an interrupt collection
INT device, pID	5.13.20	Generate an interrupt with identifier “pID”
CLEAR device, pID	5.13.21	Clear the pending state of the interrupt with identifier “pID”
SYNC TA	5.13.22	Wait for completion of any outstanding ITS actions for collection re-distributor (for example, any prior MOVI or INV commands).

Figure 18: Use of ITS command without virtualisation

Where:

- “pID” specifies a unique physical interrupt identifier between 8192 and the maximum value allowed by the number of identifier bits supported. The number of identifier bits supported is determined from GITS_TYPER.IDbits (see section 5.12.6).
- “device” specifies a unique physical device identifier. The number of device bits supported is determined from GITS_TYPER.Devbits (see section 5.12.6).
- “collection” specifies a unique interrupt collection identifier
- “TA” is a physical address that specifies the physical re-distributor that corresponds to the target physical processor.

4.9.22 Command Mapping for a Guest with a GICv3 ITS

In GICv3, direct injection of virtual interrupts into a guest operating system is ***not*** supported.
In GICv3 systems, where a guest virtual device corresponds to some physical device (i.e. the device is not emulated by the hypervisor), the table below shows the expected mapping of virtual guest ITS commands for this device to physical ITS commands:

Guest Command	Section	Physical Command
MAPD device, ITT IPA, ITT size	5.13.11	MAPD dev, ITT PA, ITT size
MAPC vCID, vTA	5.13.12	MAPC pCID, pTA
MAPI device, vID, vCID	5.13.13	MAPVI dev, vID, pID, pCID ***Note***: on receipt of interrupt pID the hypervisor must generate interrupt vID to the guest. ***Note***: it is expected that the hypervisor will allocate a globally unique pID to each (device, vID) pair.
MAPVI device, ID, vID, vCID	5.13.14	MAPVI dev, ID, pID, pCID ***Note***: on receipt of interrupt pID the hypervisor must generate interrupt vID to the guest. ***Note***: it is expected that the hypervisor will allocate a globally unique pID to each (device, vID) pair.
MOVI device, vID, vCID	5.13.15	MOVI dev, vID, pCID
DISCARD device, vID	5.13.16	DISCARD dev, vID
INV device, vID	5.13.17	INV dev, vID
INVALL vCID	5.13.19	INVALL pCID
INT device, vID	5.13.20	INT dev, vID ***Note***: interrupt vID must have been previously mapped by the guest operating system using a MAPI command (resulting in MAPVI command generated by the hypervisor).
CLEAR device, vID	5.13.21	CLEAR dev, vID ***Note***: interrupt vID must have been previously mapped by the guest operating system using a MAPI command (resulting in MAPVI command generated by the hypervisor).
SYNC vTA	5.13.22	SYNC pTA

Figure 19: GICv3 Virtual Command Mapping

Where:

- “ID” specifies an input identifier with any value up to the maximum value allowed by the number of identifier bits supported.
- “vID” specifies a virtual interrupt identifier between 8192 and the maximum value allowed by the number of identifier bits supported. The number of identifier bits supported is determined from GITS_TYPER.IDbits (see section 5.12.6). The identifier must be unique for the software that owns the specified device.

- “pID” specifies a unique physical interrupt identifier between 8192 and the maximum value allowed by the number of identifier bits supported. The number of identifier bits supported is determined from GITS_TYPER.IDbits (see section 5.12.6).
- “dev” is the physical device that corresponds to the guest’s virtual “device”. The number of device bits supported is determined from GITS_TYPER.Devbits (see section 5.12.6).
- “vTA” is an intermediate physical address that specifies a virtual re-distributor
- “pTA” is a physical address that specifies the physical re-distributor that maps to the virtual re-distributor specified by “vTA”

4.9.23 Command Mapping for a Guest using a GICv4 ITS

In GICv4, direct injection of virtual interrupts into a guest operating system is supported.
This allows the command mapping shown in section 4.9.22 above to be enhanced as shown in the table below:

Guest Command	Section	Physical Command	Section
MAPD device, ITT IPA, ITT size	5.13.11	MAPD dev, ITT PA, ITT size	5.13.11
MAPC vCID, vTA	5.13.12	VMAPP VCPU, pTA, VPT	5.13.23
MAPI device, vID, vCID	5.13.13	VMAPI dev, vID, pID, VCPU Where pID is a doorbell interrupt that will be generated if the target virtual processor is not resident. ***Note***: on receipt of interrupt pID the hypervisor might re-schedule the guest if it is not currently running.	5.13.24
MAPVI device, ID, vID, vCID	5.13.14	VMAPVI dev, ID, vID, pID, VCPU Where pID is a doorbell interrupt that will be generated if the target virtual processor is not resident. ***Note***: on receipt of interrupt pID the hypervisor might re-schedule the guest if it is not currently running.	5.13.25
MOVI device, vID, vCID	5.13.15	VMOVI dev, vID, VCPU	5.13.29
DISCARD device, vID	5.13.16	DISCARD dev, vID	5.13.16
INV device, vID	5.13.17	INV dev, vID	5.13.17
INVALL vCID	5.13.19	VINVALL VCPU	5.13.26
INT device, vID	5.13.20	INT dev, vID	5.13.20
CLEAR device, vID	5.13.21	CLEAR dev, vID	5.13.21
SYNC vTA	5.13.22	VSYNC VCPU	5.13.27

Figure 20: GICv4 Virtual Command Mapping

Where:

- “vID” specifies a virtual interrupt identifier between 8192 and the maximum value allowed by the number of identifier bits supported. The number of identifier bits supported is determined from GITS_TYPER.IDbits (see section 5.12.6). The identifier must be unique for the software that owns the specified device
- “pID” specifies a unique physical interrupt identifier between 8192 and the maximum value allowed by the number of identifier bits supported. The number of identifier bits supported is determined from GITS_TYPER.IDbits (see section 5.12.6).
- “dev” is the physical device that corresponds to the guest’s virtual “device”. The number of device bits supported is determined from GITS_TYPER.Devbits (see section 5.12.6).
- “vTA” is an intermediate physical address that specifies a virtual re-distributor

- "VCPU" specifies the virtual processor that corresponds to the virtual re-distributor specified by "vTA"

4.9.24 Provision of Memory for ITS Tables

To allow software to provision memory for ITS private tables, it provides a set of registers that allow discovery of the number of private tables required (for example, a "device table"), the size of each entry in each table and the scaling factor for each table.

The following types of scaling factor are expected:

- The number of bits of unique device identifier presented to an ITS.
- The number of physical processors.
- The number of virtual processors.
- The number of interrupts.

These registers (GITS_BASERn; see section 5.12.12) must be provisioned before the ITS is enabled.

Note: an ITS might be powered-off dynamically. This means that, when the ITS is re-initialised following a subsequent power-on, the tables might contain valid mapping data.

4.9.25 ITS Initialisation

To initialize an ITS, software must perform the following actions:

- Provision memory for any private ITS tables. See section 4.9.24.
- Provision memory for the command queue. See section 4.9.8.
- Set the command queue write pointer (GITS_CWRITER) to the start of the queue. See section 5.12.10. ***Note***: the read pointer, GITS_CREADR does not require initialization because it is updated by hardware when GITS_CBASER is written (see section 5.12.9).
- Enable the ITS by writing to GITS_CTLR. See section 5.12.5.

4.9.26 Generating an interrupt mapped using MAPI

In both GICv3 and GICv4, the identifiers of interrupts mapped using MAPI (see MAPI in section 4.9.7 and section 5.13.13) are not translated and these interrupts correspond to physical LPIs. The ITS only manages the routing of such interrupts to a target processor.

The diagram below shows an overview of how such an interrupt is handled:

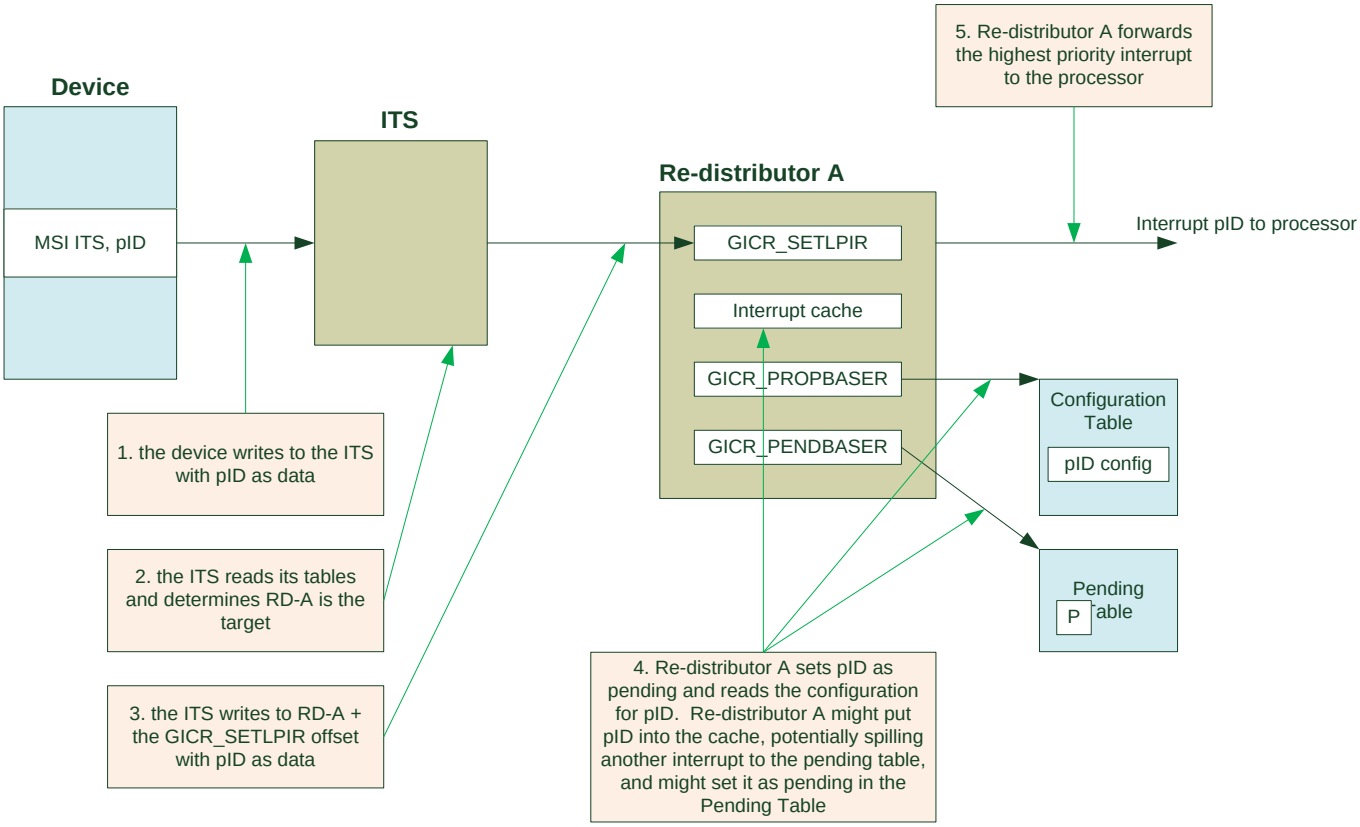


Figure 21: Generating an interrupt mapped using MAPI

Where:

- “pID” is the identifier of an interrupt for the device that has mapped as a physical LPI using the MAPI command (see section 5.13.13).
- “RD-A” is the base address of Re-distributor A

4.9.27 Generating an interrupt mapped using MAPVI

In both GICv3 and GICv4, the identifiers of interrupts mapped using MAPVI (see MAPVI in section 4.9.7 and section 5.13.13) are translated and these interrupts correspond to virtual LPIs. The ITS manages the translation of the incoming virtual identifier to a physical identifier and the routing of the physical interrupt to a target processor.

Note: in GICv4, it is expected that most virtual LPIs will be mapped using VMAPI (see sections 5.13.24 and 4.9.7) and will be directly injected into the target virtual processor.

The diagram below shows an overview of how such an interrupt is handled:

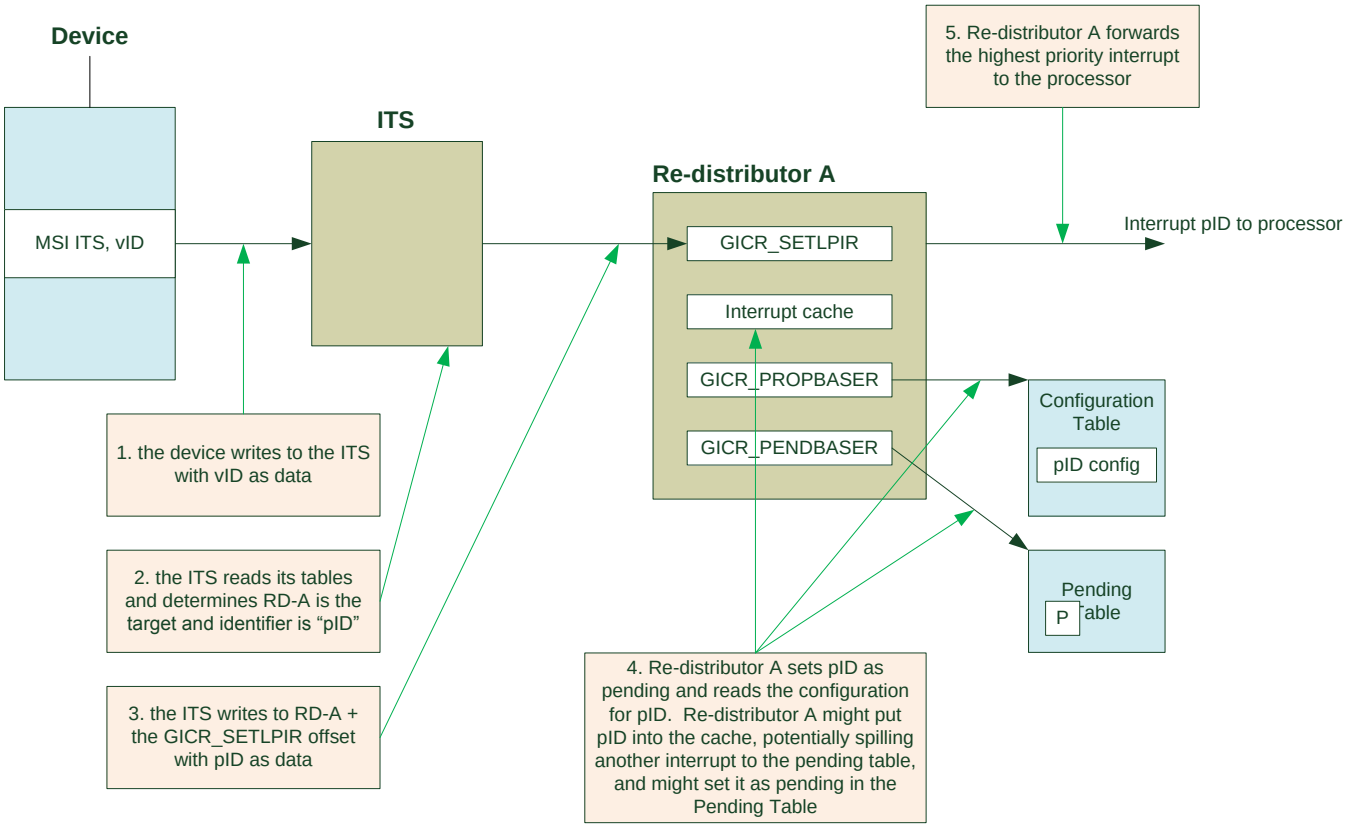


Figure 22: Generating an interrupt mapped using MAPVI

Where:

- “vID” is the identifier of an interrupt for the device that has been mapped using the MAPVI command (see section 5.13.14).
- “pID” is the identifier of the physical interrupt that will be generated
- “RD-A” is the base address of Re-distributor A

4.9.28 Generating an interrupt mapped using VMAPI

In GICv4, the identifiers of interrupts mapped using VMAPI (see VMAPI in section 4.9.7 and section 5.13.24) are not translated and these interrupts correspond to virtual LPIs. The ITS manages the delivery of the incoming virtual identifier directly to the target virtual processor and the routing of the virtual interrupt to a target processor.

For interrupts mapped using VMAPI, an interrupt is handled by the target re-distributor as below:

- If the target virtual processor is “Resident” (see section 4.8.8) the virtual LPI will be set pending and is injected into the target virtual processor when it becomes the highest priority interrupt.
- If the target virtual processor is “Not Resident” the virtual LPI will be set pending and is injected into the target virtual processor when it becomes resident and it is the highest priority interrupt. The physical doorbell interrupt will be set pending and is forwarded to the target physical processor when it becomes the highest priority interrupt.

Note: in GICv3, virtual LPIs will be mapped using MAPVI (see sections 5.13.14 and 4.9.7) and will be translated to unique physical interrupts presented to the hypervisor.

The diagram below shows an overview of how such an interrupt is handled:

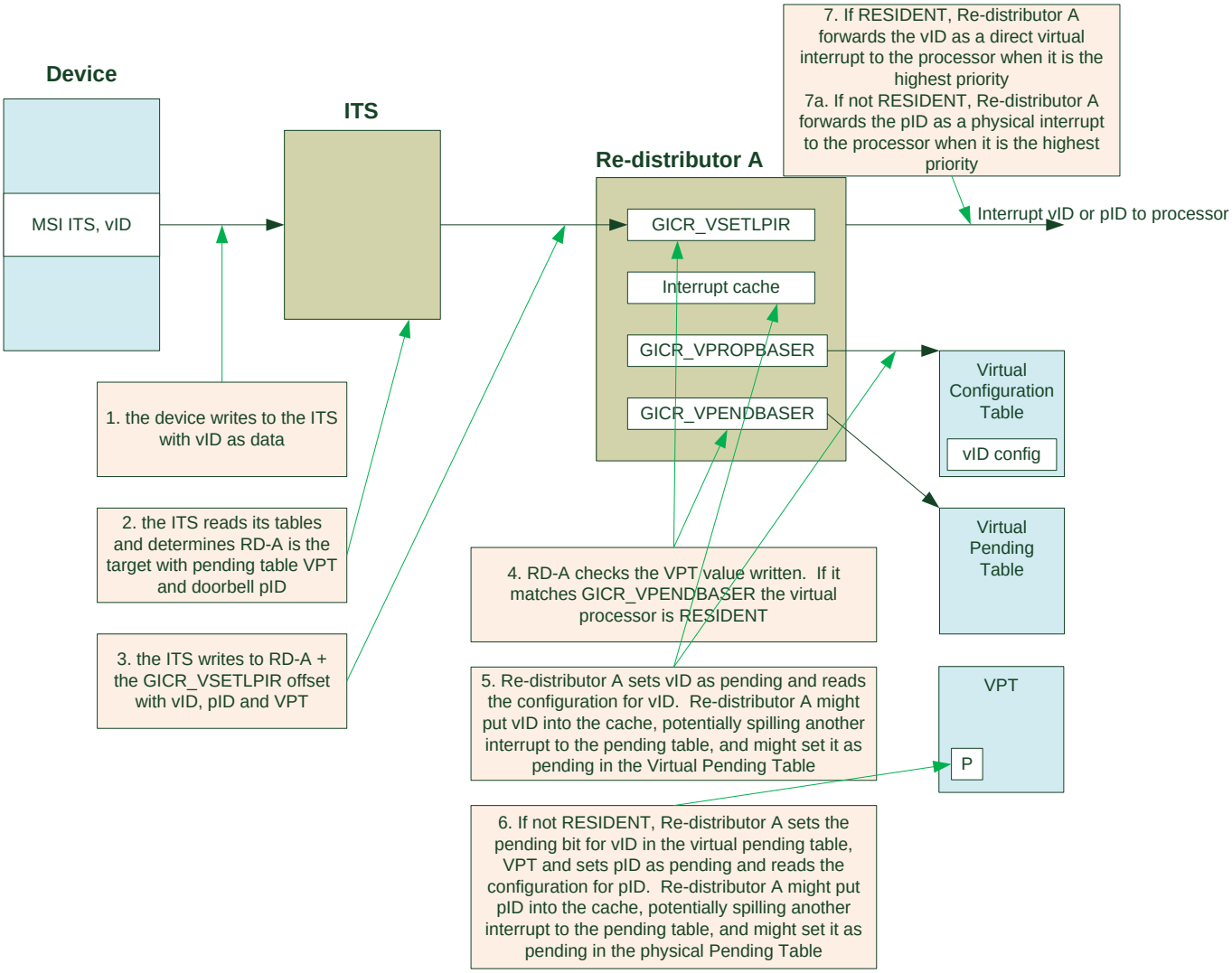


Figure 23: Generating an interrupt mapped using VMAPI

Where:

- “vID” is the identifier of the interrupt that has been mapped using the VMAPI command (see section 5.13.24)
- “pID” is the identifier of the physical doorbell interrupt
- VPT is the base address of the Virtual Pending Table for the target virtual processor, VCPU.
- RD-A is the base address of Re-distributor A

4.9.29 Moving an interrupt mapped using MAPI or MAPVI

In both GICv3 and GICv4, interrupts mapped using both MAPI and MAPVI are moved to a different target processor using the MOVI command (see sections 4.9.7 and 5.13.15).
Note: in GICv4, it is expected that most virtual LPIs will be mapped using VMAPI (see sections 5.13.24 and 4.9.7) and will be moved using VMOVI (see section 4.9.30).
The diagram below shows an overview of how movement of such interrupts is handled:

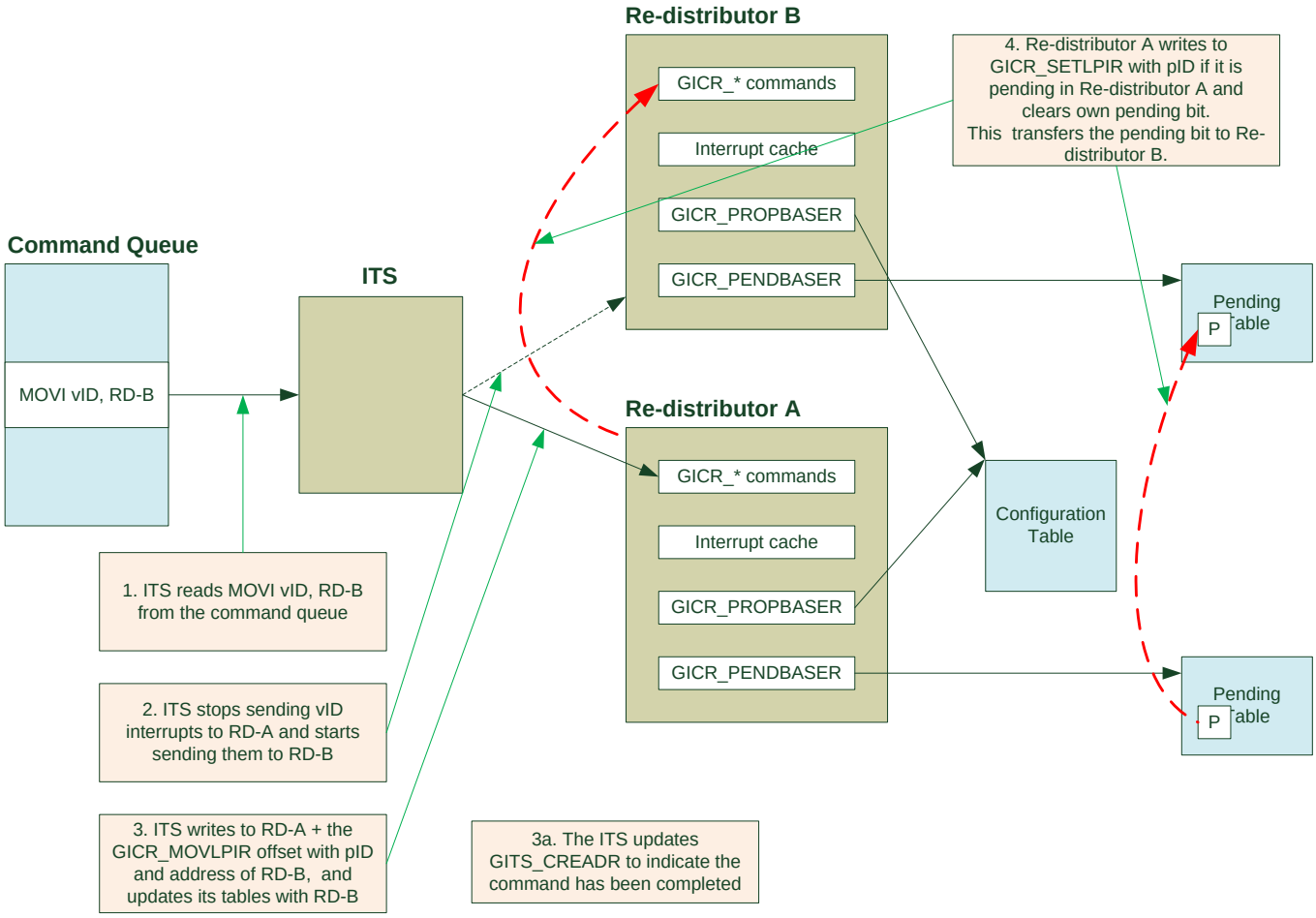


Figure 24: Moving an interrupt mapped using MAPI or MAPVI

Where:

- "vID" is the identifier programmed into the device
- "pID" is the corresponding physical identifier.
    - For interrupts mapped using MAPI (see section 5.13.13) this will be the same as "vID".
    - For interrupts mapped using MAPVI (see section 5.13.14), this is the physical identifier provided in the MAPVI command
- RD-A is the base address of the current target re-distributor held in the ITS tables.
- RD-B is the base address of the new target re-distributor.

4.9.30 Moving an interrupt mapped using VMAPI or VMAPVI

In GICv4, interrupts mapped using VMAPI or VMAPVI are moved to a different target processor using the VMOVI command (see sections 4.9.7 and 5.13.29).

Note: in GICv3, virtual LPIs will be mapped using MAPVI (see sections 5.13.14 and 4.9.7) and will be moved using MOVI (see section 4.9.29).

The diagram below shows an overview of how movement of such interrupts is handled:

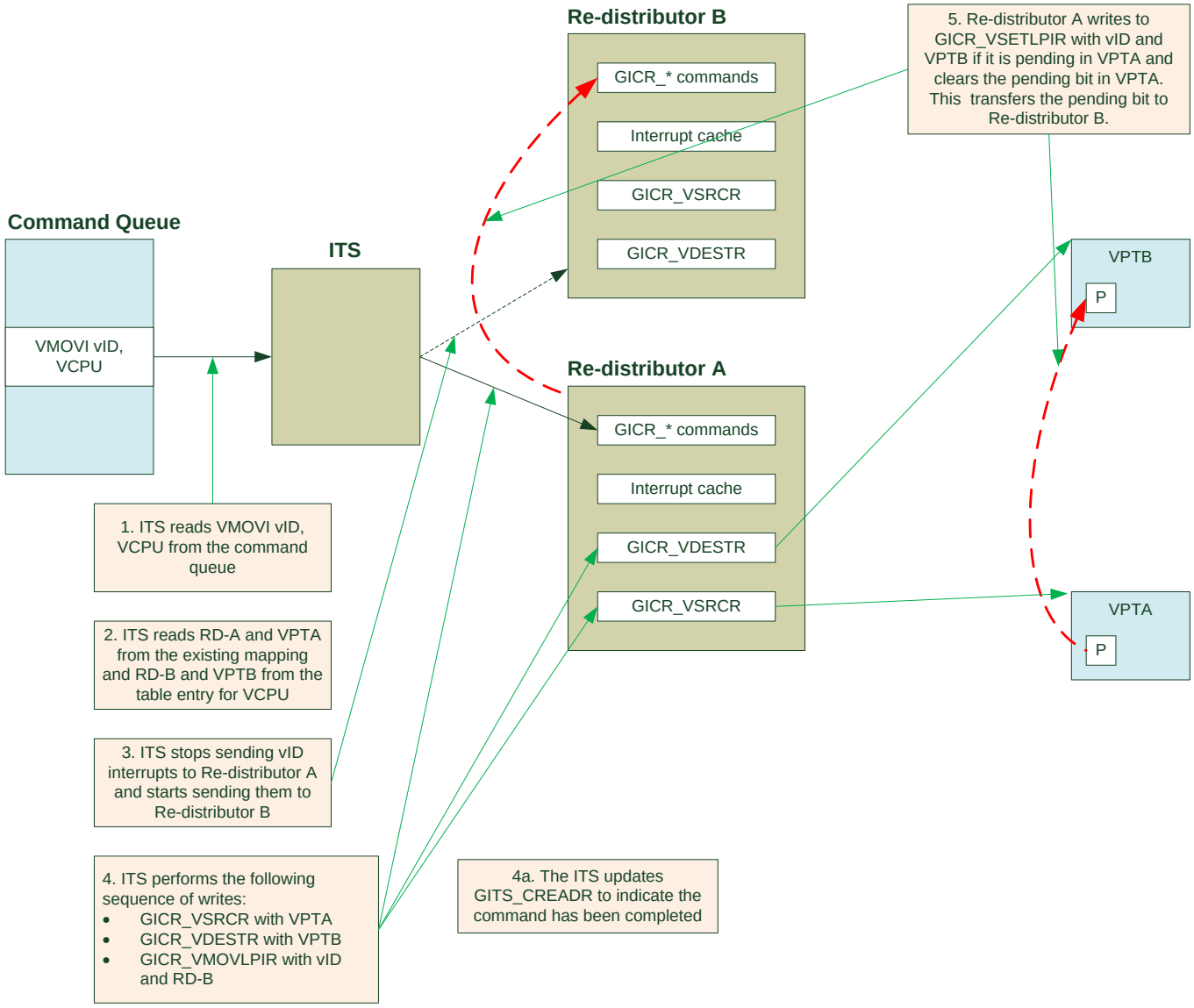


Figure 25: Moving an interrupt mapped using VMAPI or VMAPVI

Where:

- “vID” is the virtual identifier of the interrupt being moved and “VCPU” specifies the new target virtual processor
- RD-A and VPTA are the addresses of the target re-distributor and virtual pending table for the current target virtual processor
- RD-B and VPTB are the addresses of the target re-distributor and virtual pending table for the new target virtual processor

4.9.31 Moving a Virtual Processor

In GICv4, a virtual processor can be moved from one target re-distributor to another using the VMOVP command (see sections 4.9.7 and 5.13.28).
The diagram below shows an overview of how movement of such interrupts is handled:

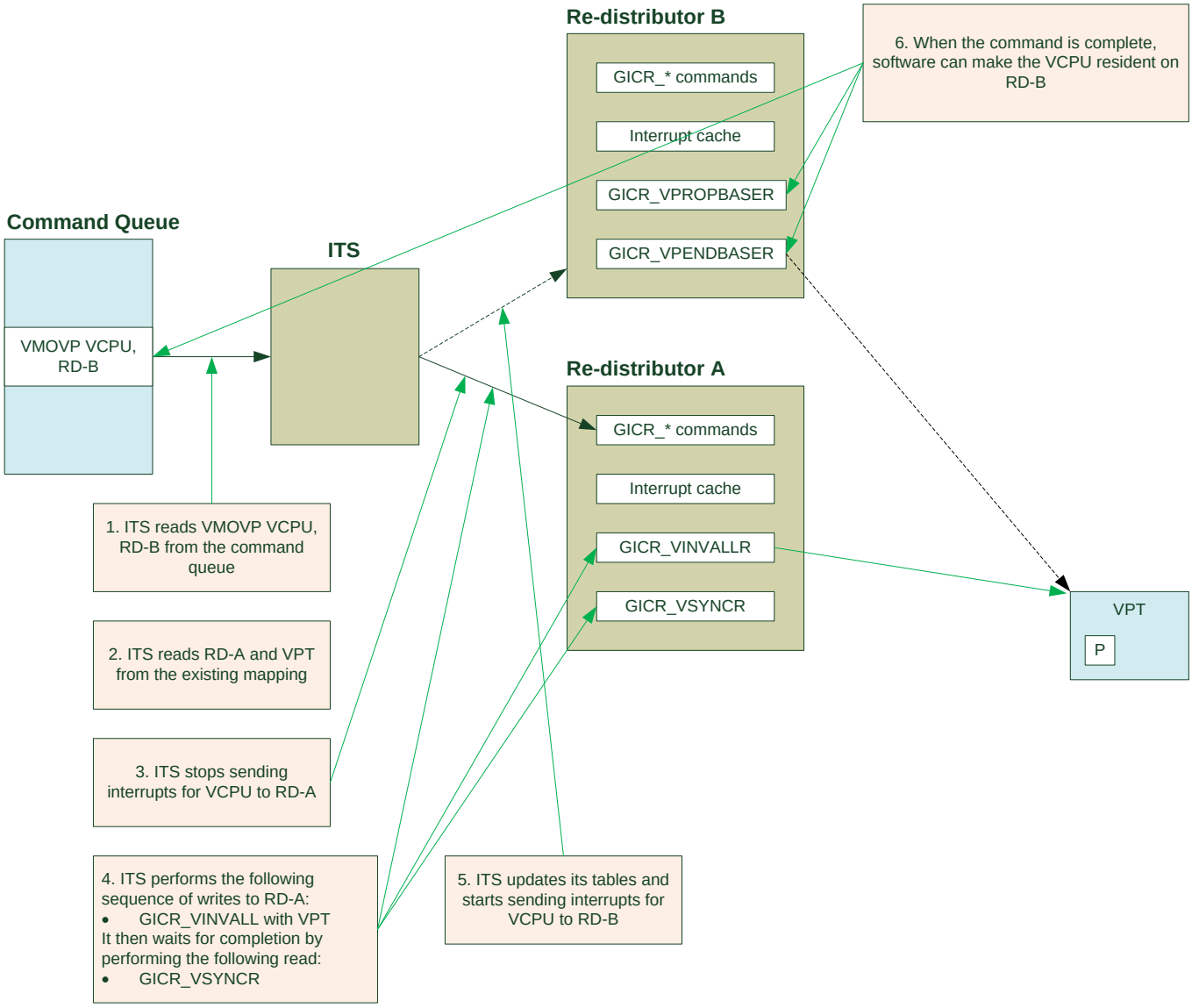


Figure 26: Moving a virtual processor

Where:

- “VCPU” is the identifier of the virtual processor
- “RD-B” is the base address of the new target re-distributor
- “RD-A” is the base address of the current target re-distributor
  “VPT” is the base address of the virtual pending table for the virtual processor.

4.10 Introduction to Power Management

4.10.1 Power Management Interface

To allow a Re-distributor to “wake up” a processor that is unable to receive interrupts, it is expected that each Re-Distributor must supply an external wake up signal, as specified below.

Signal	Input / Output	Description
WakeRequest	Output	Wake up request for the down-stream re-distributor or processor. • For Re-Distributor 0, there will only be a single signal to the power controller that indicates that the attached processor must be made able to process interrupts.

Table 19: Re-Distributor Wake-Up Signals

If a re-distributor wishes to forward an interrupt or system error to a downsteam processor:

- If GICR_WAKER.ProcessorSleep == ‘0’ it may forward the interrupt if the interrupt group is enabled

  If GICR_WAKER.ProcessorSleep == ‘1’, it must assert the WakeRequest signal until GICR_WAKER.ProcessorSleep becomes zero, at which point it may forward the interrupt if the interrupt group is enabled

Note: a processor cannot be selected as the target of a “1 of N” interrupt if the associated enable is zero.

4.10.2 Processor Power Management

Each affinity level 0 re-distributor provides a WakeRequest signal, which can be used to request that a processor which is currently powered down is powered up to allow it to handle interrupts.

When ARE is not set for a security state, WakeRequest is asserted for a processor when an interrupt is pending for any interrupt group of that security state for that processor. ***Note***: if more than one bit is set to one in the corresponding GICD_ITARGETSRn register for a pending SPI, the Distributor must assert at least one WakeRequest and eventually ***all*** WakeRequests for processors where the corresponding GICD_ITARGETSRn bit is one, until a ProcessorSleep bit has been set to zero (and the interrupt sent to the corresponding CPU interface). This ensures compatibility with GICv2 behaviour.

When ARE is set for a security state, the Enable bits that govern whether a processor is participating in distribution for an interrupt group are also used (see section 4.5.4), in conjunction with a “ProcessorSleep” bit, for power management. The table below summarizes this behaviour:

ProcessorSleep	Behaviour
0	The Distributor is allowed to forward interrupts for any interrupt group permitted by the participation rules in section 4.5.4 to the CPU interface. The Distributor is allowed to forward system errors permitted by the participation rules in section 4.5.4 to the CPU interface.
1	The processor must be woken before the Distributor can forward interrupts for any interrupt group permitted by the participation rules in section 4.5.4 to this processor. The processor must be woken before the Distributor can forward system errors permitted by the participation rules in section 4.5.4 to this processor. Additionally, this processor must not be chosen as the target of a “1 of N” system error. When a re-distributor wishes to forward an interrupt to a processor in this state, it must: • Assert WakeRequest for that processor. See section 4.10.1. • Retain the interrupt as pending until it can be forwarded to this processor (i.e. when

	software sets “ProcessorSleep” to zero following a wake-up; see section 4.10.6).

Table 20: GICv3 Processor WakeRequest Behaviour

Note: the “ProcessorSleep” bit for each re-distributor is accessible using memory mapped registers via GICR_WAKER.ProcessorSleep (see section 5.4.22) and the value is communicated to the distributor.

Note: when a re-distributor receives a write that sets the “ProcessorSleep” bit to one, it must ensure that any interrupts that are pending within the CPU interface are released (see section 7.5.3).

4.10.3 Software Requirements for Power Down

Software must ensure that any pending and active interrupts within the CPU interface and virtual CPU interface are saved and restored across power down.

Software must ensure no actions that result in traffic to the Distributor are performed after it has set GICR_WAKER.ProcessorSleep to one otherwise the effects are unpredictable.

The sections that follow specify the actions software must perform for power down and power up.

Note: it is recommended that GICR_WAKER is normally accessed by the processor associated with an affinity level 0 re-distributor. If software accesses a GICR_WAKER register to power-down a different processor, then system error interrupts for that other processor might be lost unless there is coordination between the processors to ensure that any outstanding system error interrupts have been taken prior to power off.

4.10.4 Power Down Sequencing

The concept for powering down a processor is outlined below.

First, software must ensure that the processor only receives interrupts that should wake the processor from power-down. That is, software must re-target any such interrupts either by writing directly to the peripheral devices that could generate such interrupts or by modifying the Interrupt Translation Service tables.

Second, software must set the “ProcessorSleep” bit in the affinity level 0 re-distributor for the processor that will be powered off.

When the “ProcessorSleep” bit is set in a re-distributor, it must ensure that any pending interrupts within the CPU interface are released. To achieve this, the re-distributor performs a simple request / acknowledgement sequence for the processor (using the Quiesce packet defined in section 7.3.4). The re-distributor must also ensure that any caches within the re-distributor are consistent with external memory, including the contents of any implementation defined coarse-map (see section 4.8.5).

Finally, software may power down the processor.

The detailed sequence for powering-off a processor is given below.

4.10.5 Processor Power Down With Power-On by Interrupt

To power down a processor using GICv3 and allow automatic power-on if an interrupt must be sent to a processor, software must follow the following sequence:

- Set Enable to zero for all interrupt groups (by writing to GICC_CTL or ICC_IGRPEN{0,1}_EL1/3 as appropriate). Configure the power control logic to power-on the processor when WakeRequest is asserted for by the processor’s connected affinity level 0 re-distributor
- Configure the affinity level 0 re-distributor wake up controls:
    - Set GICR_WAKER.ProcessorSleep to one in the affinity level 0 re-distributor.
    - After GICR_WAKER.ProcessorSleep is set to one, software ***must*** wait for GICR_WAKER.ChildrenAsleep to be set to one and ***must*** ensure no further interrupts are processed that would require communication with the Distributor, otherwise the system behaviour is unpredictable.
    - The affinity level 0 re-distributor ensures any pending interrupts that have been sent to the processor are released (see section 7.5.3).

    - Poll the value of GICR_WAKER.ChildrenAsleep until it is set to one to ensure the CPU interface is quiescent.
    - ***Note***: this procedure must be followed regardless of ARE settings.
- Ensure the remainder of the processor is ready to power off.
- Perform the operations required such that the power control logic powers down the processor
- Perform a WFI instruction to inform the power controller that the processor may be powered off.

When an interrupt arrives for an interrupt group where automatic power-on is required:

- The affinity level 0 re-distributor will assert WakeRequest
- The power control logic will perform a power-on sequence for the processor
- The processor will execute code starting from its initial reset vector
- Configure the affinity level 0 re-distributor wake up controls:
    - Set GICR_WAKER.ProcessorSleep to zero in the connected affinity level 0 re-distributor
    - Wait for GICR_WAKER.ChildrenAsleep to be zero.
    - ***Note***: this procedure must be followed regardless of ARE settings.
- Eventually, software will write to a register that enables interrupts (GICC_CTLR or ICC_IGRPEN{0,1}_EL1) and an Upstream Write packet will be sent to the affinity level 0 re-distributor.

See section 7.6.5 for a diagram illustrating processor power down sequencing.

Note: once software has written GICR_WAKER.ProcessorSleep to one, it ***must*** wait until GICR_WAKER.ChildrenAsleep is one before re-writing GICR_WAKER.ProcessorSleep to zero. Otherwise the effects are unpredictable (in particular, an implementation might choose to ignore such a write).

4.10.6 Processor Power-Up Sequencing

The concept of powering up a processor is outlined below. ***Note***: this procedure must also be followed if the power-down sequence above is aborted after setting GICR_ProcessorSleep to one, except that the first step (i.e. power on and reset) will not occur.

If an interrupt becomes pending for a powered-off processor, the parent affinity level 0 re-distributor must assert WakeRequest for that processor.

Following assertion of WakeRequest by the parent, the following actions will occur:

- The processor will be powered on and perform a hardware reset
- Software must restore the configuration of all PPIs and SGIs in the affinity level 0 re-distributor. ***Note***: the affinity level 0 re-distributor cannot process any Generate SGI packets until it has received confirmation that the processor has applied its settings (as described below).
- Software must write to the GICR_WAKER of the affinity level 0 re-distributor to reset the “ProcessorSleep” bit.
- On receipt of this write, the affinity level 0 re-distributor must inform the CPU interface of its settings:
    - That is, the value of the GICD_CTLR.DS bit (disable security).
    - The processor must acknowledge that is has applied its settings
    - ***Note***: GICR_WAKER.ChildrenAsleep will read as one until the re-distributor receives this acknowledgement. Software must ensure the value of the GICR_WAKER.ChildrenAsleep bit is zero before accessing other GIC registers or the effects might be unpredictable.
    - ***Note***: in implementations that use the Distributor interface defined in section 7 communication of the CPU interface settings is achieved using the Downstream Write (see section 7.3.11) and Downstream Write Acknowledge (see section 7.4.9) packets.
- Software must poll GICR_WAKER and wait until the “ChildrenAsleep” bit is zero.
    - Once “ChildrenAsleep” is zero, software should enable any interrupt groups (including system errors) for which it wants to receive interrupts.

    - ***Note***: to cover the case where the processor was powered-down with some interrupt groups enabled, software should write to the enables for all interrupt groups regardless of whether they will be enabled or disabled.
- On receipt of the acknowledgement that the processor has applied its settings, the affinity level 0 re-distributor may forward the interrupts for enabled interrupt groups to the processor.

See section 7.6.7 for a diagram illustrating the power-up sequencing.

4.10.7 Re-Distributor Power Down Sequencing

To power down a re-distributor in GICv3, the following sequence must be followed:

- Re-direct all interrupts away from the re-distributor being powered-down, otherwise system behavior might be unpredictable.
- ***Note***: in a monolithic implementation accesses to GICR_WAKER.Sleep in all re-distributors might access the same state. In such implementations software must ensure the ITS is made quiescent such that no re-distributor can receive transactions from the ITS before powering down the re-distributors.
- Ensure the associated processor is powered-down.
    - That is, software must have previously followed the procedure specified in section 4.10.5.
    - In monolithic implementations where all re-distributors share the same state for GICR_WAKER.Sleep, all processors must be powered-down before GICR_WAKER.Sleep may be written to one.
    - ***Note***: if software does not ensure the processor is powered-down, the effects might be unpredictable (for example, and implementation might ignore writes to GICR_WAKER.Sleep unless the processor has been powered off)
- Software must write GICR_WAKER.Sleep (see section 5.4.22) to one.
    - When GICR_WAKER.Sleep is written to one, the re-distributor must ensure that any caches within the re-distributor are consistent with external memory, including the contents of any implementation defined coarse-map (see section 4.8.5).
    - When GICR_WAKER.Sleep is written to one, the re-distributor must ensure that the value of “Sleep” is communicated to the top-level Distributor.
- Software must poll GICR_WAKER and wait until the “Quiescent” bit is one.

Note: once software has written GICR_WAKER.Sleep to one, it ***must*** wait until GICR_WAKER.Quiescent is one before re-writing GICR_WAKER.Sleep to zero. Otherwise the effects are unpredictable (in particular, an implementation might choose to ignore such a write).

4.10.8 Re-Distributor Power Up Sequencing

The concept of powering up an affinity level 0 re-distributor is outlined below.

If an interrupt becomes pending for a powered-off sub-tree, the distributor must assert WakeRequest for that sub-tree.

Following assertion of WakeRequest by the distributor, the following actions will occur:

- The re-distributor will be powered on and perform a hardware reset
- The powered-up re-distributor must inform the distributor that it is no longer “Asleep”.
- On receipt of this, the distributor must inform the powering-up re-distributor of its settings
    - These include all the interrupt group enables for the re-distributor and the value of the GICD_CTLR.DS bit (disable security)
    - The powered-up re-distributor must use these settings as the settings for all its children
    - The powered-up re-distributor must then acknowledge that is has applied its settings
- On receipt of the acknowledgement that the powered-up re-distributor has applied its settings, the distributor should forward the interrupt that caused WakeRequest to be asserted to the powered up re-distributor.
- The powered-up re-distributor must assert WakeRequest for the associated processor.

See section 7.6.8 for a diagram illustrating the power up sequencing.

Note: the full state of a re-distributor (e.g. LPI configuration) will be restored by a woken processor.

Note: following power on of a re-distributor, software must ensure GICR_WAKER.Quiescent is zero before any LPI is permitted to target the re-distributor. This includes issue of any “move” commands to the ITS.

4.10.9 ITS Power-Down Sequencing

To power down an ITS in GICv3, the following sequence must be followed:

- Re-direct all interrupts away from the ITS being powered-down or disable the generation of all interrupts to that ITS.
- Software must write GITS_CTLR.Enable (see section 5.12.5) to zero.
    - When GITS_CTLR.Enable is written to zero, the ITS ensures all outstanding operations are complete.
- Software must poll GITS_CTLR and wait until the “Quiescent” bit is one.
- The ITS may now be powered-off.

Note: once software has written GITS_CTLR.Enable from one to zero, it ***must*** wait until GITS_CTLR.Quiescent is one before re-writing GITS_CTLR.Enable to one. Otherwise the effects are unpredictable (in particular, an implementation might choose to ignore such a write).

4.11 The GIC Distributor and System Interconnect

In GICv3 and GICv4, the ITS and each re-distributor might contain caches or otherwise require access to normal memory. This section specifies the rules that an implementation must follow to ensure deadlock-free operation.

4.11.1 Memory Traffic in Monolithic Implementations

A monolithic implementation must ensure that the path to normal main memory is free-flowing from the combined logic that provides the GITS_*, GICD_* and GICR_* register interfaces such that any cache fill / eviction / table-walk actions required are guaranteed to complete regardless of any pending writes to any GITS_*, GICD_* or GICR_* registers.

4.11.2 Memory Traffic in Distributed Implementations

A distributed implementation must ensure that the path to normal main memory is free-flowing from each of the blocks that provide the GITS_*, GICD_* and GICR_* register interfaces such that any cache fill / eviction / table-walk actions required are guaranteed to complete regardless of any pending writes to any GITS_*, GICD_* or GICR_* registers within the block issuing the memory traffic.

4.11.3 Re-distributor to Re-distributor Traffic in Distributed Implementations

In a distributed implementation where a read of GICR_SYNCR or GICR_VSYNCR waits for completion of other actions before returning data, the implementation must further ensure that when a read of GICR_SYNCR (see section 5.4.31) or GICR_VSYNCR (see section 5.4.42) is awaiting completion of other actions before returning data, the re-distributor continues to accept further writes to GICR_* registers and that any writes to normal memory required to accept such writes are free-flowing as specified above.

5 REGISTER DEFINITIONS

Register names have been retained from GICv2 wherever possible. This means that many GIC register names do not follow the ARMv8 convention of appending the Exception Level that owns the register. However, the names of the system registers for accessing these GIC registers do follow the ARMv8 convention.

5.1 Memory Mapped Special Behaviour

5.1.1 Reserved Behaviour

If software accesses a reserved location, the location is RES0.

If an implementation detects a read access to a reserved location it might choose to set an “RRD” bit in an optional status register (e.g. GICD_STATUSR.RRD) and might report this in an implementation defined manner (for example by asserting a pin or by generating a system error).

If an implementation detects a write access to a reserved location it might choose to set a “WRD” bit in an optional status register (e.g. GICR_STATUSR.WRD) and might report this in an implementation defined manner (for example by asserting a pin or by generating a system error).

5.1.2 Read Only and Write only Behaviour

If software writes to a read-only register, the write must be ignored. If an implementation detects such an access it might choose to set a “WROD” bit in an optional status register (e.g. GICD_STATUSR.WROD) and might report this in an implementation defined manner (for example by asserting a pin or by generating a system error).

Similarly if software reads a write-only registers the location reads as zero. If an implementation detects such an access it might choose to set a “RWOD” bit in an optional status register (e.g. GICD_STATUSR.RWOD) and might report this in an implementation defined manner (for example by asserting a pin or by generating a system error).

5.2 Reset Values

Unless otherwise specified in the sections that follow (or in the GICv2 specification, if appropriate), all registers reset to an UNKNOWN value and software must ensure registers are initialized before use, or system behaviour might be unpredictable.

5.3 Distributor Registers

To support affinity routing, the Distributor register map is expanded to occupy a 64kB page:

Register Name	Type	Offset	Reset	Notes
GICD_CTLR	RW	0x0000	IMP DEF	
GICD_TYPER	RO	0x0004	IMP DEF	
GICD_IIDR	RO	0x0008	IMP DEF	
Reserved		0x000C	-	
GICD_STATUSR	RW	0x0010	0x00000000	Optional error reporting register
Reserved		0x0014 – 0x001C	-	
IMP DEF		0x0020 – 0x003C	IMP DEF	
GICD_SETSPI_NSR	WO	0x0040	-	Data contains ID
Reserved		0x0044	-	Possible 64b expansion
GICD_CLRSPI_NSR	WO	0x0048	-	Data contains ID
Reserved		0x004C	-	Possible 64b expansion
GICD_SETSPI_SR	WO	0x0050	-	Data contains ID
Reserved		0x0054	-	Possible 64b expansion
GICD_CLRSPI_SR	WO	0x0058	-	Data contains ID
Reserved		0x005C	-	Possible 64b expansion
Reserved		0x0060 – 0x007C	-	
GICD_IGROUPRn	RW	0x0080 – 0x00FC	IMP DEF	Reset values as defined for GICv2
GICD_ISENABLERn	RW	0x0100 – 0x017C	IMP DEF	Reset values as defined for GICv2
GICD_ICENABLERn	RW	0x0180 – 0x01FC	IMP DEF	Reset values as defined for GICv2
GICD_ISPENDRn	RW	0x0200 – 0x027C	0x00000000	
GICD_ICPENDRn	RW	0x0280 – 0x02FC	0x00000000	
GICD_ISACTIVERn	RW	0x0300 – 0x037C	0x00000000	
GICD_ICACTIVERn	RW	0x0380 – 0x03FC	0x00000000	
GICD_IPRIORITYRn	RW	0x0400 – 0x07F8	0x00000000	
GICD_ITARGETSRn	RW	0x0800 – 0x0BF8	IMP DEF	Reset values as defined for GICv2 When ARE is set, replaced by GICD_IROUTERn
GICD_ICFGR	RW	0x0C00 – 0x0CFC	IMP DEF	
GICD_IGRPMODRn	RW	0x0D00 – 0x0D7C	0x00000000	RES0 when ARE is zero for the secure state
GICD_NSACRn	RW	0x0E00 – 0x0EFC	0x00000000	

GICD_SGIR	WO	0x0F00	-	RAZ / WI when ARE is set for a security states. ***Note***: when SRE is set for an interrupt regime, SGIs can be generated using ICC_SGI0R_EL1, ICC_SGI1R_EL1 or ICC_ASGI1R_EL1 as appropriate (see section 5.7.29).
GICD_CPENDSGIRn	RW	0x0F10 – 0x0F1C	0x00000000	Replaced by GICR_ICPENDR0 when ARE is set
GICD_SPENDSGIRn	RW	0x0F20 – 0x0F2C	0x00000000	Replaced by GICR_ISPENDR0 when ARE is set
Reserved		0x0F30 – 0x5FFC	-	
GICD_IROUTERn	RW	0x6000 – 0x7FF8	0x00000000	
Reserved		0x8000 – 0xBFFC		
IMP DEF		0xC000 – 0xFFCC		
Identification Registers	RO	0xFFD0 – 0xFFFC	IMP DEF	Reserved for Implementation Defined PrimeCell ID registers. Includes registers GICD_PIDR{0..7} which are architecturally defined or are defined for ARM implementations. See section 5.3.23.

Table 21: Distributor Register Map

5.3.1 GICD_TYPER

This 32 bit read-only register has been updated to clarify the meaning of the CPUNumber field and to add capability discovery for Locality-specific Peripherals Interrupts, System Registers and Interrupt Translation.

- Bits [31:25]. Reserved. RES0.
- Bits [24]. A3V. Indicates whether the Distributor supports non-zero values of Affinity 3.
- Bits [23:19]. IDbits. The number if interrupt identifier bits supported by the GIC Stream Protocol Interface (see section 7), minus one.
- Bit [18]. DVIS. Direct Virtual LPI Injection supported. See section 4.9.3.
- Bit [17]. LPIS. Locality-specific Peripheral Interrupts supported.
- Bit [16]. MBIS. Message Based Interrupts supported (by writing to Distributor registers).
- Bits [15:11]. LSPI. Number of Lockable SPI Interrupts. Unmodified from GICv2. ***Note***: this is no longer supported in GICv3 and this field is RES0.
- Bit [10]. SecurityExtn. Unmodified from GICv2, except this bit is RAZ when GICD_CTLR.DS is one.
- Bits [9:8]. Reserved. RES0.
- Bits [7:5]. CPUNumber. The value (CPUNumber + 1) indicates how many processors may be used as interrupt targets when the ARE bit is zero. These processors must be numbered contiguously from zero, however, the relation between this “processor number” and the affinity hierarchy from MPIDR is IMPLEMENTATION DEFINED.
- Bits [4:0]. ITLinesNumber. Unmodified from GICv2. ***Note***: the value derived from this specifies the maximum number of SPIs. An implementation may implement “sparse” interrupts and may not implement all SPIs up to this maximum.

Note: this register is unaffected by ARE.

5.3.2 GICD_IIDR

This 32 bit read-only register behaves exactly as defined for GICv2.

5.3.3 GICD_ITARGETSRn

When ARE is zero for the security state of an associated interrupt, this register provides the routing information for the associated interrupt. The byte associated behaves as GICv2 (i.e. each bit maps to a processor) and the number of bits (starting from zero) that may be set is given by (GICD_TYPER.CPUNumber + 1).

The mapping between bit numbers and the affinity hierarchy is implementation defined. ***Note***: software can discover this mapping by reading GICR_TYPER for each affinity level 0 re-distributor, and processor numbers in the range zero to GICD_TYPER.CPUNumber correspond to bit positions in GICD_ITARGETSRn.

When ARE is set for the security state of an associated interrupt, the routing information for that interrupt is provided by GICD_IROUTERn (below) and the associated byte in GICD_ITARGETSRn is RES0.

In systems supporting two security states, the GICD_ITARGETSRn registers associated with secure interrupts may only be accessed by secure accesses and are RAZ/WI for non-secure accesses (unless permitted by the setting of the associated GICD_NSACRn register; see section 5.3.15).

Note: as for GICv2, byte accesses are permitted to this register.

Note: when ARE becomes zero for a security state (e.g. after reset or following a write to GICD_CTLR), the value of this register is UNKNOWN for that security state.

Note: when the group of an interrupt changes such that the ARE setting for the interrupt changes to zero, the value of this register is UNKNOWN for that interrupt.

Note: if software writes to this register without following the procedure in sections 4.5.5 and 4.5.6 implementations must ensure that an interrupt that is pending at the time of the write uses either the old value or the new value and must ensure that the interrupt is neither lost nor handled more than once.

5.3.4 GICD_IROUTERn

These registers provide the routing information when ARE is set for the security state of the associated interrupts. When ARE is not set, these registers are RES0.

This set of registers is allocated 8kB of the Distributor address space. Each register provides up to 64 bits of state to control the routing of its associated interrupt. The format of these registers is given below:

- Bits [63:40]. Reserved. RES0.
- Bits [39:32]. Affinity 3.
- Bits [31]. Interrupt Routing Mode (see section 4.2.6).
- Bits [30:24]. Reserved. RES0.
- Bits [23:16]. Affinity 2.
- Bits [15:8]. Affinity 1.
- Bits [7:0]. Affinity 0.

If this register is programmed to forward the corresponding interrupt to a specific processor (i.e. IRM is zero) and the affinity path does not correspond to an implemented processor, then if the corresponding interrupt becomes pending it will not be forwarded to ***any*** processor and will remain pending.

In systems supporting two security states, the GICD_IROUTERn registers associated with secure interrupts may only be accessed by secure accesses and are RAZ/WI for non-secure accesses (unless permitted by the setting of the associated GICD_NSACRn register; see section 5.3.15).

Note: when ARE becomes one for a security state (e.g. after reset or following a write to GICD_CTLR), the value of all writeable fields in this register is UNKNOWN for that security state.

Note: when the group of an interrupt changes such that the ARE setting for the interrupt changes to one, the value of this register is UNKNOWN for that interrupt.

Note: if software writes to this register without following the procedure in sections 4.5.5 and 4.5.6 implementations must ensure that an interrupt that is pending at the time of the write uses either the old value or the new value and must ensure that the interrupt is neither lost nor handled more than once.

Note: for each interrupt, an implementation might support less than 256 values for an affinity level and some bits of the appropriate affinity field might have a fixed, read-only value.

For SPI ID *m*:

- the corresponding GICD_IROUTERn number, *n*, is given by *n* = *m*
- the offset of the required GICD_IROUTERn is (0x6000 + (8**n*))
- GICD_IROUTER0 to GICD_IROUTER31 are RES0 (as they correspond to SGIs or PPIs)

5.3.5 GICD_IGROUPRn

When ARE is zero for the current security state of an interrupt, the bit in GICD_IGROUPRn for the interrupt behaves exactly as defined for GICv2.

When ARE is one for the current security state of an interrupt, the bit in GICD_IGROUPRn for the interrupt is concatenated with GICD_IGRPMODRn to form a two bit field for that interrupt that defines an interrupt group (as defined in section 4.1.3).

In systems supporting a single security state, the bit in GICD_IGROUPRn for an interrupt indicates whether the interrupt is Group 0 or Group 1.

In systems supporting two security states, GICD_IGROUPRn may only be accessed by secure accesses and is RAZ/WI for non-secure accesses.

For interrupts in GICD_IGROUPR0, when ARE is set for the security state of an interrupt, the bit in GICD_IGROUPR0 is RES0 for that interrupt and the equivalent function is provided by GICR_IGROUPR0 (see section 5.4.9).

Note: if software writes to this register without following the procedure in sections 4.5.5 and 4.5.6 implementations must ensure that an interrupt that is pending at the time of the write uses either the old value or the new value and must ensure that the interrupt is neither lost nor handled more than once.

5.3.6 GICD_IGRPMODRn

In GICv3, to enable the GIC to distinguish between Secure Group 0, Secure Group 1 and Non-Secure Group 1 interrupts, GICD_IGROUPRn is modified by a new register, GICD_IGRPMODRn.

The two bit field for an interrupt, formed by concatenating the appropriate bit from GICD_IGRPMODR and GICD_IGROUPR is decoded as shown in section 4.1.3.

In systems supporting two security states, GICD_IGRPMODRn may only be written by secure accesses and is RAZ/WI for non-secure accesses.

For interrupts in GICD_IGRPMODR0, when ARE is set for the security state of an interrupt, the bit in GICD_IGRPMODR0 is RES0 for that interrupt and the equivalent function is provided by GICR_IGRPMODR0 (see section 5.4.10).

Note: GICv2 does not include a GRPMOD register and the value of GICD_IGRPMODRn is always considered to be zero (and is RAZ/WI) when ARE is not set for the secure state or when GICD_CTLR.DS is set to one.

Note: if software writes to this register without following the procedure in sections 4.5.5 and 4.5.6 implementations must ensure that an interrupt that is pending at the time of the write uses either the old value or the new value and must ensure that the interrupt is neither lost nor handled more than once.

5.3.7 GICD_ISPENDRn

When ARE is zero for the security state of an interrupt, the bit in GICD_ISPENDRn for the interrupt behaves exactly as defined for GICv2. That is:

- Writing one to a bit sets the status of the corresponding peripheral interrupt to pending.
- Reading a bit identifies whether the interrupt is pending.

When ARE is one for the security state of an interrupt, the bit in GICD_ISPENDRn for the interrupt behaves exactly as defined for GICv2 except:

- For SGI and PPI interrupts, the bit in GICD_ISPENDR0 for the interrupt is RES0 and the equivalent function is provided by GICR_ISPENDR0 (see section 5.4.11). ***Note***: there is no sharing of the pending state for SGIs for the ARE settings.
- Pending bits corresponding to Group 0 and Group 1 Secure interrupts may only be set by secure accesses

5.3.8 GICD_ICPENDRn

When ARE is zero for the security state of an interrupt, the bit in GICD_ICPENDRn for the interrupt behaves exactly as defined for GICv2.

When ARE is one for the security state of an interrupt, the bit in GICD_ICPENDRn for the interrupt behaves exactly as defined for GICv2 except:

- For SGI and PPI interrupts, the bit in GICD_ICPENDR0 for the interrupt is RES0 and the equivalent function is provided by GICR_ICPENDR0 (see section 5.4.12). ***Note***: there is no sharing of the pending state for SGIs for the ARE settings.
- Pending bits corresponding to Group 0 and Group 1 Secure interrupts may only be cleared by secure accesses

5.3.9 GICD_SPENDSGIRn and GICD_CPENDSGIRn

When ARE is zero for the security state of an interrupt, the bits in GICD_SPENDSGIRn and GICD_CPENDSGIRn for the interrupt behave exactly as defined for GICv2.

Note: to clarify the GICv2 specification, these registers operate normally regardless of whether interrupts are disabled in GICD_CTLR.

When ARE is one for the security state of an interrupt, the bits in GICD_SPENDSGIRn and GICD_CPENDSGIRn for the interrupt are RES0. ***Note***: there is no sharing of the pending state for SGIs for the ARE settings.

5.3.10 GICD_ISENABLERn

When ARE is zero for the security state of an interrupt, the bit in GICD_ISENABLERn for the interrupt behaves exactly as defined for GICv2.

When ARE is one for the security state of an interrupt, the bit in GICD_ISENABLERn for the interrupt behaves exactly as defined for GICv2 except:

- For SGI and PPI interrupts, the bit in GICD_ISENABLER0 for the interrupt is RES0 and the equivalent function is provided by GICR_ISENABLER0 (see section 5.4.13).
- Enable bits corresponding to Group 0 and Group 1 Secure interrupts may only be set by secure accesses

5.3.11 GICD_ICENABLERn

When ARE is zero for the security state of an interrupt, the bit in GICD_ICENABLERn for the interrupt behaves exactly as defined for GICv2.

When ARE is one for the security state of an interrupt, the bit in GICD_ICENABLERn for the interrupt behaves exactly as defined for GICv2 except:

- For SGI and PPI interrupts, the bit in GICD_ICENABLER0 for the interrupt is RES0 and the equivalent function is provided by GICR_ICENABLER0 (see section 5.4.14).
- Enable bits corresponding to Group 0 and Group 1 Secure interrupts may only be cleared by secure accesses
- Writes to the register cannot be considered complete until the effects of the write are visible throughout the affinity hierarchy.
- ***Note***: to ensure an enable has been cleared, software must write to the register with bits set to clear the required enables. Software must then poll GICD_CTLR.RWP (register writes pending) until it has the value zero.

Note: when ARE is zero for the security state of an interrupt, there is no mechanism to know whether the effects of clearing an enable are visible throughout the hierarchy. Because GICv3 implementations can be distributed it might take much longer for such effects to become visible and software might still receive newly disabled interrupts during this time.

5.3.12 GICD_ISACTIVERn and GICD_ICACTIVERn

When ARE is zero for the security state of an interrupt, the bits in GICD_ISACTIVERn and GICD_ICACTIVERn for the interrupt behave exactly as defined for GICv2.

When ARE is one for the security state of an interrupt, the bits in GICD_ISACTIVERn and GICD_ICACTIVERn for the interrupt behave exactly as defined for GICv2 except:

- For SGI and PPI interrupts, the bits in GICD_ISACTIVER0 and GICD_ICACTIVER0 for the interrupt are RES0 and the equivalent functions are provided by GICR_ISACTIVER0 and GICR_ICACTIVER0 (see section 5.4.15).

5.3.13 GICD_IPRIORITYRn

When ARE is zero for the security state of an interrupt, the byte in GICD_IPRIORITYRn for the interrupt behaves exactly as defined for GICv2.

When ARE is one for the security state of an interrupt, the byte in GICD_IPRIORITYRn for the interrupt behaves exactly as defined for GICv2 except:

- For SGI and PPI interrupts, the byte in GICD_IPRIORITYR0 to GICD_IPRIORITYR7 for the interrupt is RES0 and the equivalent functions is provided by a byte in GICR_IPRIORITYR0 to GICR_IPRIORITYR7 (see section 5.4.16).

Note: when GICD_CTLR.DS is one, the full priority range may be programmed by non-secure accesses.

Note: as for GICv2, byte accesses are permitted to this register.

Note: if software writes to this register without following the procedure in sections 4.5.5 and 4.5.6 implementations must ensure that an interrupt that is pending at the time of the write uses either the old value or the new value and must ensure that the interrupt is neither lost nor handled more than once.

5.3.14 GICD_SGIR

When ARE is zero for a security state of an access, this register behaves as defined for GICv2.

When ARE is one for a security state of an access, the GICD_SGIR register is RES0 for that security state.

Note: when SRE is one for an interrupt regime, SGIs are generated using ICC_SGI0R_EL1, ICC_SGI1R_EL1 or ICC_ASGI1R_EL1 as appropriate (see section 5.7.29).

5.3.15 GICD_NSACRn

This secure register controls whether non-secure access is permitted to secure interrupt resources. Except for GICD_NSACR0, this register behaves as defined for GICv2 with the following clarifications to the NS_access field:

- 0b00. No Non-secure access is permitted to fields associated with the corresponding interrupt.
- 0b01. Non-secure read and write access is permitted to fields associated with the corresponding interrupt in the GICD_ISPENDRn registers. A Non-secure write access to GICD_SETSPI_NSR is permitted to set the pending state of the corresponding interrupt. A Non-secure write access to GICD_SGIR is permitted to generate a Group 0 SGI for the corresponding interrupt.
- 0b10. Adds Non-secure read and write access permission to fields associated with the corresponding interrupt in the GICD_ICPENDRn registers. A Non-secure write access to GICD_CLRSPI_NSR is permitted to clear the pending state of the corresponding interrupt. Also adds Non-secure read access permission to fields associated with the corresponding interrupt in the GICD_ISACTIVERn and GICD_ICACTIVERn registers.
- 0b11. Adds Non-secure read and write access permission to fields associated with the corresponding interrupt in the GICD_ITARGETSRn and GICD_IROUTERn registers.

This register is RAZ/WI for non-secure accesses.

When GICD_CTLR.DS is one, this register is RAZ/WI.

For GICD_NSACR0, the field in this register (which controls access to the SGI interrupt) is replaced by a field in GICR_NSACR (see section 5.4.19) when ARE is one for the security state of the corresponding interrupt. Otherwise it behaves as defined for GICv2.

Note: there is a mistake in the GICv2 specifiction which states that GICD_NSACR0 bits [15:0] control access to SGI configuration registers. Because each field in this register comprises two bits, the whole of GICD_NSACR0 controls access rights to SGI registers, GICD_NSACR1 controls access to PPI registers (and is always RAZ/WI) and GICD_NSACR2 and beyond control access to SPI registers.

Note: when ARE is zero for the security state of the associated interrupt, accesses to GICD_NSACR0 must access the appropriate GICR_NSACR state.

5.3.16 GICD_SETSPI_NSR

This 32 bit register is write-only and may be written regardless of the ARE settings. The value written to this register specifies which SPI to set to pending. The format of the value written is shown below:

- Bits [31:10]. Reserved. RES0.
- Bits [9:0]. The ID of the SPI. The value written must identify a valid SPI otherwise the write has no effect.

If the SPI is already pending, a write has no effect.

If the value specifies an invalid SPI, the write has no effect.

If this register is written using a non-secure access, and the value specifies a secure SPI and the value of the corresponding GICD_NSACR*n* register is zero (i.e. does not permit non-secure accesses to set the interrupt as pending), the write has no effect. ***Note***: secure accesses may set the pending state of any valid SPI.

If the value specifies a PPI or an SGI, the write has no effect.

Note: when GICD_CTLR.DS is one, the Distributor operates in a single security state and all interrupts are considered “non-secure”.

Note: 16 bit access to bits[15:0] of this register must be supported.

5.3.17 GICD_CLRSPI_NSR

This 32 bit register is write-only and may be written regardless of the ARE settings. The value written to this register specifies which SPI to clear from “pending”. The format of the value written is shown below:

- Bits [31:10]. Reserved
- Bits [9:0]. The ID of the SPI. The value written must identify a valid SPI otherwise the write has no effect.

If the SPI is not pending, the write has no effect.

If the value specifies an invalid SPI, the write has no effect.

If this register is written using a non-secure access, and the value specifies a secure SPI and the value of the corresponding GICD_NSACR*n* register is less than 0b10 (i.e. does not permit non-secure accesses to clear the pending state of the interrupt), the write has no effect. ***Note***: secure accesses may clear the pending state of any valid SPI.

If the value specifies a PPI or an SGI, the write has no effect.

Note: when GICD_CTLR.DS is one, the Distributor operates in a single security state and all interrupts are considered “non-secure”.

Note: 16 bit access to bits[15:0] of this register must be supported.

5.3.18 GICD_SETSPI_SR

This 32 bit register is write-only and may be written regardless of the ARE settings, but may only be written using secure accesses. The value written to this register specifies which SPI to set to “pending”. The format of the value written is shown below:

- Bits [31:10]. Reserved
- Bits [9:0]. The ID of the SPI. The value written must identify a valid SPI otherwise the write has no effect.

If this register is written using a non-secure access, the write has no effect.

If the SPI is already pending, the write has no effect.

If the value specifies an invalid SPI, the write has no effect.

Note: when GICD_CTLR.DS is one, this register is RAZ/WI.

Note: 16 bit access to bits[15:0] of this register must be supported.

5.3.19 GICD_CLRSPI_SR

This 32 bit register is write-only and may be written regardless of the ARE settings, but may only be written using secure accesses. The value written to this register specifies which SPI to clear from “pending”. The format of the value written is shown below:

- Bits [31:10]. Reserved.
- Bits [9:0]. The ID of the SPI. The value written must identify a valid SPI otherwise the write has no effect.

If this register is written using a non-secure access, the write has no effect.

If the SPI is not pending, the write has no effect.

If the value specifies an invalid SPI, the write has no effect.

Note: when GICD_CTLR.DS is one, this register is RAZ/WI.

Note: 16 bit access to bits[15:0] of this register must be supported.

5.3.20 GICD_CTLR

This register remains banked by security. The format of the register is dependent on the security state of the access, the setting of ARE for that security state and the setting of the DS bit.

Note: when a write changes the value of ARE for a security state or the value of the DS bit, the format used for interpreting the remaining bits provided in the write data is the format that applied before the write takes effect.

In systems supporting two security states, the format of GICD_CTLR for secure accesses is as shown below:

- Bit [31]. RWP. Register Write Pending. Read only. This bit is common to both security states and indicates whether a register write is in progress or not.
    - 0b0. The effect of all register writes are visible to all descendants of the top-level re-distributor, including processors.
    - 0b1. The effects of all register writes are not visible to all descendants of the top-level re-distributor.
    - ***Note***: this field tracks completion of writes to GICD_CTLR that change the state of an interrupt group enable or an affinity routing enable setting and writes to GICD_ICENABLERn that clear the enable of one or more SPIs.
- Bits [30:7]. Reserved. RES0.
- Bit [6]. DS. Disable Security. When set, non-secure accesses are permitted to access and modify registers that control Group 0 interrupts. Resets to zero.
    - ***Note***: if DS becomes one when ARE_S is one, then ARE for the single security state is RAO/WI.
    - ***Note***: this bit is RAO/WI if the Distributor only supports a single security state (see below).
    - ***Note***: this bit is RAZ/WI if the Distributor only supports two security states.
    - ***Note***: when DS is set, all accesses to GICD_CTLR access the single security state view (below) and all bits are accessible
    - ***Note***: once set to one, DS may only be cleared by a hardware reset.
- Bit [5]. ARE_NS. When set to one, enables affinity routing for the non-secure state. In systems where GICv2 backwards compatibility for the non-secure state is implemented, this bit resets to zero; otherwise this bit is RAO/WI.
    - ***Note***: this bit is RAO/WI when ARE is one for the secure state.
    - ***Note***: once set to one, an implementation may choose to make the bit RAO/WI and only cleared by a hardware reset.
- Bit [4]. ARE_S. When set to one, enables affinity routing for the secure state. In systems where GICv2 backwards compatibility for the secure state is implemented, this bit resets to zero; otherwise this bit is RAO/WI.
    - ***Note***: if ARE_S is written from one to zero, ARE_NS becomes UNKNOWN.
    - ***Note***: once set to one, an implementation may choose to make the bit RAO/WI and only cleared by a hardware reset.
- Bit [3]. Reserved. RES0.
- Bit [2]. EnableGrp1, secure. Enables Secure Group 1 interrupts. RES0 if ARE_S is zero.
- Bit [1]. EnableGrp1, non-secure. Behaves as defined for GICv2.
- Bit [0]. EnableGrp0. Behaves as defined for GICv2.

In systems supporting two security states, the format of GICD_CTLR for non-secure accesses is as shown below:

- Bit [31]. RWP. Register Write Pending. Read only. This bit indicates whether a register write is in progress or not.
    - 0b0. The effect of all register writes are visible to all descendants of the top-level re-distributor, including processors.
    - 0b1. The effects of all register writes are not visible to all descendants of the top-level re-distributor.
    - ***Note***: this field tracks completion of writes to GICD_CTLR that change the state of an interrupt group enable or an affinity routing setting and writes to GICD_ICENABLERn that clear the enable of one or more SPIs.

- Bits [30:5]. Reserved. RES0.
- Bit [4]. ARE_NS. When set, enables affinity routing for the non-secure state. In systems where GICv2 backwards compatibility for the non-secure state is implemented, this bit resets to zero; otherwise this bit is RAO/WI.
    - ***Note***: this bit is RAO/WI when ARE is one for the secure state.
    - ***Note***: once set to one, an implementation may choose to make the bit RAO/WI and only cleared by a hardware reset.
- Bit [3:2]. Reserved. RES0.
- Bit [1]. EnableGrp1A. When ARE_NS is one, enables Non-secure Group 1 interrupts. Otherwise RES0.
- Bit [0]. EnableGrp1. When ARE_NS is zero, enables Non-secure Group 1 interrupts. Otherwise RES0.

Systems supporting a single security state, allow access to the enables for System Errors, Group 0 interrupts and Group 1 interrupts:

- Bit [31]. RWP. Register Write Pending. Read only. This bit indicates whether a register write is in progress or not.
    - 0b0. The effect of all register writes are visible to all descendants of the top-level re-distributor, including processors.
    - 0b1. The effects of all register writes are not visible to all descendants of the top-level re-distributor.
    - ***Note***: this field tracks completion of writes to GICD_CTLR that change the state of an interrupt group enable or an affinity routing setting and writes to GICD_ICENABLERn that clear the enable of one or more SPIs.
- Bits [30:7]. Reserved. RES0.
- Bit [6]. DS. Disable Security. This field is RAO/WI.
- Bit [5]. Reserved. RES0.
- Bit [4]. ARE. When set, enables affinity routing. In systems where GICv2 backwards compatibility is implemented, this bit resets to zero; otherwise this bit is RAO/WI.
    - ***Note***: in systems supporting two security states when GICD_CTLR.DS is one, this corresponds to the ARE_NS field.
    - ***Note***: once set to one, an implementation may choose to make the bit RAO/WI and only cleared by a hardware reset.
- Bit [3:2]. Reserved. RES0.
- Bit [1]. EnableGrp1. Behaves as defined for GICv2.
- Bit [0]. EnableGrp0. Behaves as defined for GICv2.

Note: this view is also accessed in systems supporting two security states when GICD_CTLR.DS is set to one.

To change an ARE bit, interrupts for that security state must be disabled in the Distributor by clearing the Enable bits defined above, otherwise a Distributor might choose to generate system errors (see sections 5.1.1 and 5.1.2).

5.3.21 GICD_ICFGRn

When ARE is zero for the security state of an interrupt, this register behaves as defined for GICv2.

When ARE is one for the security state of an interrupt, the fields in GICD_ICFGR0 and GICD_ICFGR1 for that interrupt are RES0 and the equivalent function is provided by GICR_ICFGR0 and GICR_ICFGR1 (see sections 5.4.17 and 5.4.18). For all other “n” values, the Int_config field is defined as:

- Bit [1]. Edge Triggered.
    - 0b0. The corresponding interrupt is level-sensitive.
    - 0b1. The corresponding interrupt is edge-triggered.
- Bit [0]. RES0.

5.3.22 GICD_STATUSR

This optional 32 bit register, which is banked by security, has been added to GICv3 to provide software with a mechanism to detect accesses to reserved locations, writes to read-only locations and reads of write-only locations.

If not implemented this register is RAZ / WI.

The register is at offset 0x0010 in the Distributor register map. Its format is shown below:

- Bits [31:4]. Reserved. RES0.
- Bit [3]. WROD. This bit is set if write to a read-only location is detected. Software must write a one to this it to clear it.
- Bit [2]. RWOD. This bit is set if read to a write-only location is detected. Software must write a one to this bit to clear it.
- Bit [1]. WRD. This bit is set if a write to a reserved location is detected. Software must write a one to this bit to clear it.
- Bit [0]. RRD. This bit is set if a read to a reserved location is detected. Software must write a one to this bit to clear it.

Note: the register is banked because the checks performed depend on the ARE settings for a security state.

5.3.23 Peripheral Identification Registers

The table below specified the values of the Distributor Peripheral Identification Registers:

Offset	Register	Description
0xFFF0	GICD_CIDR0	• Component ID0. Value 0x0D.
0xFFF4	GICD_CIDR1	• Component ID1. Value 0xF0.
0xFFF8	GICD_CIDR2	• Component ID2. Value 0x05.
0xFFFC	GICD_CIDR3	• Component ID3. Value 0xB1.
0xFFE0	GICD_PIDR0	• Bits [7:0]. Bits [7:0] of the ARM-defined DevID field. This field is 0x92 in ARM implementations of a GICv3 or later Distributor.
0xFFE4	GICD_PIDR1	• Bits [7:4]. Bits[3:0] of the ARM defined JEP identity field. This field is 0xB in ARM implementations. • Bits [3:0]. Bits [11:8] of the ARM-defined DevID field. This field is 0x4 in ARM implementations of a GICv3 or later Distributor.

0xFFE8	GICD_PIDR2	• Bits [7:4]. ArchRev. This field has the following implementation defined values:
		  ○ 0x1. GICv1
		  ○ 0x2. GICv2
		  ○ 0x3. GICv3.
		  ○ 0x4. GICv4.
		  ○ 0x5 – 0xf. Reserved
		• Bit [3]. UsesJEPcode. This field is 0b1 in ARM implementations.
		• Bits [2:0]. Bits[6:4] of the ARM defined JEP identity field. This field is 0x3 in ARM implementations.

0xFFEC	GICD_PIDR3	• Bits [7:4]. RevAnd. Manufacturer defined revision number. Normally zero. • Bits [3:0]. Customer modified.

0XFFD0	GICD_PIDR4	• Bits [7:4]. RES0. • Bits [3:0]. ContinuationCode. This field is 0x4 in ARM implementations.
0xFFD4	GICD_PIDR5	• Bits [7:0]. RES0.
0xFFD8	GICD_PIDR6	• Bits [7:0]. RES0.
0xFFDC	GICD_PIDR7	• Bits [7:0]. RES0.

Table 22: Distributor Peripheral Identification Registers

5.4 Re-Distributor Registers

It is expected that registers with the same function as Distributor registers will have the same offset. The Re-Distributor memory map splits into three portions:

- Registers that control the behaviour of a re-distributor and registers associated with physical LPIs that must be present in distributed implementations (see section 4.9.1) and are optional in monolithic implementations.
- Registers associated with virtual LPIs that must be present in distributed GICv4 implementations (see section 4.9.1) and are optional in monolithic GICv4 implementations.
- Registers associated with SGIs and PPIs.

These three portions are allocated separate (contiguous) pages as defined below.

5.4.1 Re-Distributor Addressing

Each re-distributor at affinity level 0 must be allocated at one page for controlling the overall behavior of the re-distributor and for controlling physical LPIs. The base address of this page is referred to as RD_base. In addition, each re-distributor must be also allocated the following additional pages:

- In GICv3, one additional page for control and generation of SGIs (referred to as SGI_base). The pages for each re-distributor must be contiguous and must be ordered as follows: RD_base followed by SGI_base.
- In GICv4, three additional pages: one for controlling virtual LPIs (referred to as VLPI_base), one for control and generation of SGIs (referred to as SGI_base), and one reserved page. The pages for each re-distributor must be contiguous and must be ordered as follows: RD_base, SGI_base, VLPI_base and reserved.

Note: all of these pages use the 64kB translation granule.

5.4.2 Re-Distributor Registers for Control and Physical LPIs

In GICv3 and GICv4 these registers start from RD_base and the offset of each register is defined in the table below.

Register Name	Type	PLPI Offset	Notes
GICR_CTLR	RW	0x0000	
GICR_IIDR	RO	0x0004	Version numbers
GICR_TYPER	RO	0x0008 – 0x000C	Indicates affinity value of the re-distributor and the level of LPI support
GICR_STATUSR	RW	0x0010	Optional error reporting register
GICR_WAKER	RW	0x0014	Wake Request control register
Reserved		0x0018	
IMP DEF		0x0020 – 0x003C	Implementation Defined
GICR_SETLPIR	WO	0x0040 – 0x0044	64b. Data contains the physical ID ***Note***: optional in Monolithic implementations.
GICR_CLRLPIR	WO	0x0048 – 0x004C	64b. Data contains the physical ID ***Note***: optional in Monolithic implementation.
Reserved		0x0050 – 0x006C	
GICR_PROPBASER	RW	0x0070 – 0x0074	64b.
GICR_PENDBASER	RW	0x0078 – 0x007C	64b.
Reserved		0x0080 – 0x009C	
GICR_INVLPIR	WO	0x00A0 – 0x00A4	64b. Data contains the physical ID. ***Note***: optional in Monolithic implementation.
Reserved		0x00A8 – 0x00AC	
GICR_INVALLR	WO	0x00B0 – 0x00B4	***Note***: optional in Monolithic implementation.
Reserved		0x00B8 – 0x00BC	
GICR_SYNCR	RO	0x00C0 – 0x00C4	***Note***: optional in Monolithic implementation.
Reserved		0x00C8 – 0x00CC	
GICR_MOVLPIR	WO	0x0100 – 0x0104	64b. Data contains the physical ID and Target Address ***Note***: optional in Monolithic implementation.
Reserved		0x0108 – 0x010C	
GICR_MOVALLR	WO	0x0110 – 0x0114	64b. Data contains the Target Address ***Note***: optional in Monolithic implementation.
Reserved		0x0118 – 0xBFFC	
IMP DEF		0xC000 – 0xFFCC	

Identification Registers	RO	0xFFD0 – 0xFFFC	Reserved for 64kB page identification registers. This includes registers GICR_PIDR{0..7} which are architecturally defined. See section 5.4.21.

Table 23: Re-Distributor Physical LPI Register Address Map

These registers behave as described below.

5.4.3 Re-Distributor Registers for Virtual LPIs

In GICv4 distributed implementations (see section 4.9.1) these registers must be provided for affinity level 0 re-distributors and might optionally be provided in monolithic implementations. These registers start from VLPI_base and the offset of each register is defined in the table below:

Register Name	Access	VLPI Offset	Notes
Reserved		0x0000 – 0x003C	
GICR_VSETLPIR	WO	0x0040 – 0x0044	64b. Data contains the virtual ID and VPT
GICR_VCLRLPIR	WO	0x0048 – 0x004C	64b. Data contains the virtual ID and VPT
Reserved		0x0050 – 0x006C	
GICR_VPROPBASER	RW	0x0070 – 0x0074	64b. ***Note***: this register must be provided in both Distributed and Monolithic implementations.
GICR_VPENDBASER	RW	0x0078 – 0x007C	64b. ***Note***: this register must be provided in both Distributed and Monolithic implementations.
Reserved		0x0080 – 0x009C	
GICR_VINVLPIR	WO	0x00A0 – 0x00A4	64b. Data contains the virtual ID and VPT
Reserved		0x00A8 – 0x00AC	
GICR_VINVALLR	WO	0x00B0 – 0x00B4	64b. Data contains the VPT
Reserved		0x00B8 – 0x00BC	
GICR_VSYNCR	RO	0x00C0 – 0x00C4	64b.
Reserved		0x00C8 – 0x00FC	
GICR_VSRCR	RW	0x0100 – 0x0104	64b. Data contains the source VPT
GICR_VSRCR*n*	RW	0x0108 – 0x017C	As above, where "*n*" is in the range 1..15 and corresponds to the ITS number (see section 5.12.6). Only provided when more than one ITS is required.
GICR_VDESTR	RW	0x0180 – 0x0184	64b. Data contains the destination VPT
GICR_VDESTR*n*	RW	0x0188 – 0x01FC	As above, where "*n*" is in the range 1..15 and corresponds to the ITS number (see section 5.12.6). Only provided when more than one ITS is required.

GICR_VMOVLPIR	WO	0x0200 – 0x0204	64b. Data contains the virtual ID and VPT
GICR_VMOVLPIR*n*	WO	0x0208 – 0x027C	As above, where "*n*" is in the range 1..15 and corresponds to the ITS number (see section 5.12.6). Only provided when more than one ITS is required.
GICR_VSETLPILR	WO	0x0280 – 0x0284	64b. Data contains the virtual ID and VPT
GICR_VSETLPILR*n*	WO	0x0288 – 0x02FC	As above, where "*n*" is in the range 1..15 and corresponds to the ITS number (see section 5.12.6). Only provided when more than one ITS is required.
Reserved		0x0300 – 0xBFFC	
IMP DEF		0xC000 – 0xFFCC	
Reserved		0xFFD0 – 0xFFFC	

Figure 27: Re-distributor Virtual LPI Address Map

5.4.4 Re-Distributor Registers for SGIs and PPIs.

These registers start from SGI_base and the offset of each register is defined in the table below.

Register Name	Access	SGI Offset	Notes
GICR_IGROUPR0	RW	0x0080	32b. Contains the Group bits for SGIs and PPIs.
GICR_IGRPMODR0	RW	0x0D00	32b. Contains the Group Modifier bits for SGIs and PPIs.
GICR_ISENABLER0	RW	0x0100	32b. Allows the Enable bits for SGIs and PPIs to be set atomically.
GICR_ICENABLER0	RW	0x0180	32b. Allows the Enable bits for SGIs and PPIs to be cleared atomically.
GICR_ISPENDR0	RW	0x0200	32b. Allows the Pending bits for SGIs and PPIs to be set atomically.
GICR_ICPENDR0	RW	0x0280	32b. Allows the Pending bits for SGIs and PPIs to be cleared atomically.
GICR_ISACTIVER0	RW	0x0300	32b. Allows the Active bits for SGIs and PPIs to be set atomically.
GICR_ICACTIVER0	RW	0x0380	32b. Allows the Active bits for SGIs and PPIs to be cleared atomically.
GICR_IPRIORITYR{0..7}	RW	0x0400 – 0x041C	32b. Contains the priorities for SGIs and PPIs
GICR_ICFGR0	RW	0x0C00	32b. Interrupt configuration register for SGIs
GICR_ICFGR1	RW	0x0C04	32b. Interrupt configuration register for PPIs
GICR_NSACR	RW	0x0E00	32b. Contains the NSACR value checked when setting an SGI pending on a target processor (replaces GICD_NSACR0).
Reserved		0x0E04 – 0xBFFC	

IMP DEF		0xC000 – 0xFFCC	
Reserved		0xFFD0 – 0xFFFC	

Figure 28: Re-Distributor SGI and PPI Register Address Map

Note: the offsets for each register are identical to those defined in the top-level re-distributor for SGIs and PPIs.

5.4.5 Ordering of Accesses to Registers for Physical and Virtual LPIs

In implementations that support the registers that control physical and virtual LPIs, accesses to these registers must obey the following rules:

- The effects of writes must appear to be as if the re-distributor handled the writes in arrival order

5.4.6 GICR_IIDR

This 32 bit read-only register behaves like GICD_IIDR (see section 5.3.2).

5.4.7 GICR_CTLR

This 32 bit register controls the operation of a given re-distributor. This register is banked by security and the format of the register is given below:

- Bit [31]. Upstream Write Pending. Common to both security states. Read-only.
  - 0b0. The effects of all upstream writes have been communicated to the distributor, including any Generate SGI packets
  - 0b1. The effects of all upstream writes have not been communicated to the distributor, including any Generate SGI packets
- Bits [30:4]. Reserved. RES0
- Bits [3]. RWP. Register Write Pending. Common to both security states. This bit indicates whether a register write for the current security state is in progress or not.
  - 0b0. The effect of all register writes are visible to all descendants of the re-distributor, including processors.
  - 0b1. The effects of all register writes are not visible to all descendants of the re-distributor.
  - ***Note***: this field tracks completion of writes to GICR_ICENABLERn that clear the enable of one or more interrupts. It also tracks completion of any cache operations initiated by clearing GICR_CTLR.Enable_Virtual_LPIs or clearing GICR_VPENDBASER.Valid.
- Bit [2]. Reserved. RES0.
- Bit [1]. Enable Virtual LPIs. Resets to zero. Common to both security states. When this bit is clear, writes to generate Virtual LPIs to GICR_VSETLPIR will be ignored. When ARE is zero for the non-secure state, this bit is RES0. ***Note***: in GICv3, this field is RES0.
- Bit [0]. Enable LPIs. Resets to zero. Common to both security states. When this bit is clear, writes to generate Physical LPIs to GICR_SETLPIR will be ignored. When ARE is zero for the non-secure state, this bit is RES0. When a write changes this bit from zero to one, this bit becomes RES1 and the re-distributor must load the pending table from memory to check for any pending interrupts.

5.4.8 GICR_TYPER

This 64 bit read-only register is used to discover the properties of the current re-distributor and is always accessible regardless of the ARE setting for a security state.

- Bits [63:32]. Affinity Value. The identity of the re-distributor.
  - Bits [63:56]. A3. The Affinity Level 3 value for this re-distributor.
  - Bits [55:48]. A2. The Affinity Level 2 value for this re-distributor.
  - Bits [47:40]. A1. The Affinity Level 1 value for this re-distributor.
  - Bits [39:32]. A0. The Affinity Level 0 value for this re-distributor.
- Bit [31:24]. Reserved. RES0
- Bit [23:8]. Processor Number. A unique identifier for the processor understood by the ITS
- Bit [7:5]. Reserved. RES0
- Bits [4]. Last. This bit is only set to one for the last re-distributor in a set of contiguous re-distributor register pages.
- Bits [3]. Distributed. .
  - 0b0. Monolithic implementation. The registers required to support a Distributed implementation are not provided and the re-distributor does not support a separate, remote ITS
  - 0b1. Distributed implementation. All the registers required to support a Distributed implementation are provided and the re-distributor supports writes to these by a separate, remote ITS
- Bits [2]. Reserved. RES0.
- Bits [1]. VLPIS. Virtual LPIs and Direct injection of Virtual LPIs supported. ***Note***: in GICv3, this field is RES0.
- Bits [0]. PLPIS. Physical LPIs supported.

5.4.9 GICR_IGROUPR0

Regardless of the ARE value for a security state, this register behaves identically to GICD_IGROUPR0 as defined above (see section 5.3.5). ***Note***: access is allowed when ARE is zero to permit secure boot code to configure PPIs to be group 1 in processors that do not support GICv2 operation in the secure state.

This register only applies to SGIs and PPIs. For SPIs, this functionality is provided by GICD_IGROUPRn.

Note: accesses to GICD_IGROUPR0 when ARE is zero for a security state access the same state as GICR_IGROUPR0 and must update affinity level 0 re-distributor state associated with the processor performing the accesses.

Note: if software writes to this register without following the procedure in section 4.5.6 implementations must ensure that an interrupt that is pending at the time of the write uses either the old value or the new value and must ensure that the interrupt is neither lost nor handled more than once.

5.4.10 GICR_IGRPMODR0

When ARE is zero for the security state of an interrupt or GICD_CTLR.DS is one, the bit for that interrupt in this register is RES0.

When ARE is one for the security state of an interrupt and GICD_CTLR.DS is zero, the bit for that interrupt in this register behaves identically to GICD_IGRPMODR0 as defined above (see section 5.3.6).

This register only applies to SGIs and PPIs. For SPIs, this functionality is provided by GICD_IGRPMODRn.

Note: if software writes to this register without following the procedure in section 4.5.6 implementations must ensure that an interrupt that is pending at the time of the write uses either the old value or the new value and must ensure that the interrupt is neither lost nor handled more than once.

5.4.11 GICR_ISPENDR0

When ARE is zero for the security state of an interrupt, this register is RES0.

When ARE is one for the security state of an interrupt:

- For PPI interrupts, bits in GICR_ISPENDR0[31:16] behave identically to GICD_ISPENDR0[31:16] as defined above (see section 5.3.7). That is:
    - Writing one to a bit sets the status of the corresponding peripheral interrupt to pending.
    - Reading a bit identifies whether the interrupt is pending.
- Because only a single bit is required for the pending state of each SGI in each CPU interface, GICR_ISPENDR0[15:0] may be used to set an SGI as pending. It may also be used to save and restore the pending state of an SGI.

In systems supporting two security states, non-secure accesses to bits corresponding to secure SGIs or PPIs are RAZ/WI.

Note: accesses to GICD_ISPENDR0[31:16] when ARE is zero for the security state of an interrupt access the same state as GICR_ISPENDR0[31:16] and must update affinity level 0 re-distributor state associated with the processor performing the access.

5.4.12 GICR_ICPENDR0

When ARE is zero for the security state of an interrupt, this register is RES0.

When ARE is one for the security state of an interrupt:

- For PPI interrupts, bits in GICR_ICPENDR0[31:16] behave identically to GICD_ICPENDR0[31:16] as defined above (see section 5.3.8).
- Because only a single bit is required for the pending state of each SGI in each CPU interface, GICR_ICPENDR0[15:0] may be used to clear the pending state of an SGI.

In systems supporting two security states, non-secure accesses to bits corresponding to secure SGIs or PPIs are RAZ/WI.

Note: accesses to GICD_ICPENDR0 when ARE is zero for the security state of an interrupt access the same state as GICR_ICPENDR0 and must update affinity level 0 re-distributor state associated with the processor performing the access.

5.4.13 GICR_ISENABLER0

When ARE is zero for a security state of an interrupt, the bit in this register for the interrupt is RES0.

When ARE is one for a security state of an interrupt, the bit in this register for the interrupt behaves identically to GICD_ISENABLER0 as defined above (see section 5.3.10).

In systems supporting two security states, non-secure accesses to bits corresponding to secure SGIs or PPIs are RAZ/WI.

Note: when ARE is one for a security state, unlike GICv2, the enable bit for implemented SGIs in this register must be programmable and must reset to zero.

Note: accesses to GICD_ISENABLER0 when ARE is zero for the security state of an interrupt access the same state as GICR_ISENABLER0 and must update affinity level 0 re-distributor state associated with the processor performing the access.

5.4.14 GICR_ICENABLER0

When ARE is zero for a security state of an interrupt, the bit in this register for the interrupt is RES0.

When ARE is one for a security state of an interrupt, the bit in this register for the interrupt behaves identically to GICD_ICENABLER0 as defined above (see section 5.3.11) except as noted below.

In systems supporting two security states, non-secure accesses to bits corresponding to secure SGIs or PPIs are RAZ/WI.

Note: accesses to GICD_ICENABLER0 when ARE is zero for the security state of an interrupt access the same state as GICR_ICENABLER0 and must update affinity level 0 re-distributor state associated with the processor performing the access.

Note: in GICv3, writes to GICR_ICENABLER0 cannot be considered complete until their effects are visible throughout the re-distributor hierarchy.

Note: to ensure an enable has been cleared, software must write to the register with bits set to clear the required enables. Software must then poll GICR_CTLR.RWP (register writes pending) until it has the value zero.

5.4.15 GICR_ISACTIVER0 and GICR_ICACTIVER0

When ARE is zero for a security state of an interrupt, the bits in these registers for the interrupt are RES0.

When ARE is one for a security state of an interrupt, the bits in these registers for the interrupt behave identically to GICD_ISACTIVER0 and GICD_ICACTIVER0 as defined above (see section 5.3.12).

In systems supporting two security states, non-secure accesses to bits corresponding to secure SGIs or PPIs are RAZ/WI.

Note: accesses to GICD_I{S,C}ACTIVER0 when ARE is zero for the security state of an interrupt access the same state as GICR_I{S,C}ACTIVER0 and must update affinity level 0 re-distributor state associated with the processor performing the access.

5.4.16 GICR_IPRIORTYR0 to GICR_IPRIORITYR7

When ARE is zero for a security state of an interrupt, the byte in this register for the interrupt is RES0.

When ARE is one for a security state of an interrupt, the bytes in these registers for the interrupt behave identically to GICD_IPRIORITYR0 and GICD_IPRIORITYR7 as defined above (see section 5.3.13).

In systems supporting two security states, non-secure accesses to bytes corresponding to secure SGIs or PPIs are RAZ/WI.

Note: byte accesses are permitted to these registers.

Note: accesses to GICD_IPRIORITYR{0..7} when ARE is zero for the security state of an interrupt access the same state as GICR_IPRIORITYR{0..7} and must update affinity level 0 re-distributor state associated with the processor performing the access.

Note: if software writes to this register without following the procedure in section 4.5.6 implementations must ensure that an interrupt that is pending at the time of the write uses either the old value or the new value and must ensure that the interrupt is neither lost nor handled more than once.

5.4.17 GICR_ICFGR0

When ARE is zero for a security state of an interrupt, bit field in this register for the interrupt is RES0.

When ARE is one for a security state of an interrupt, the Int_config field for the interrupt is defined as:

- Bit [1]. Edge-Triggered. Read-only and RAO. That is, SGIs are always edge-triggered.
- Bit [0]. Read-only. RES0.

5.4.18 GICR_ICFGR1

When ARE is zero for a security state of an interrupt, bit field in this register for the interrupt is RES0.

When ARE is one for a security state of an interrupt, the Int_config field for the interrupt is defined as:

- Bit [1]. Edge-triggered. It is implementation defined whether this bit is programmable.
    - 0b0. The corresponding interrupt is level-sensitive.
    - 0b1. The corresponding interrupt is edge-triggered.
- Bit [0]. Read-only. RES0.

5.4.19 GICR_NSACR

This secure register is used to check whether non-secure software is permitted to generate Secure interrupts at a target processor following a non-secure processor write to ICC_SGI1R_EL1, ICC_ASGI1R_EL1 or ICC_SGI0R_EL1.

When ARE is one for the security state of an SGI interrupt, this register is used instead of GICD_NSACR0.

This register is RAZ/WI for non-secure accesses.

When GICD_CTLR.DS is one, this register is RAZ/WI.

This register provides two bits per SGI. When the appropriate bits GICR_IGROUPR0 and GICR_IGRPMODR0 indicate the SGI is a Secure Group 0 or a Secure Group 1 interrupt, the meaning of the two bit field in GICR_NSACR is shown below:

- 0b00. No non-secure accesses
- 0b01. Non-secure writes are permitted to generate Secure Group 0 interrupts.
- 0b10. Non-secure writes are also permitted to generate Secure Group 1 interrupts.
- 0b11. Reserved, treated as 0b01.

Note: for compatibility with GICv2, writes to GICD_NSACR0 must be coordinated within the Distributor and must update the appropriate affinity level 0 re-distributor state.

Note: the privileges granted are additive with increasing value.

5.4.20 GICR_STATUSR

This optional 32 bit register, which is banked by security, has been added to GICv3 to provide software with a mechanism to detect accesses to reserved locations, writes to read-only locations and reads of write-only locations.

If not implemented this register is RAZ / WI.

The register is at offset 0x0010 in the Re-Distributor register map. Its format is shown below:

- Bits [31:4]. Reserved. RES0.
- Bit [3]. WROD. This bit is set if write to a read-only location is detected. Software must write a one to this bit to clear it.
- Bit [2]. RWOD. This bit is set if read to a write-only location is detected. Software must write a one to this bit to clear it.
- Bit [1]. WRD. This bit is set if a write to a reserved location is detected. Software must write a one to this bit to clear it.
- Bit [0]. RRD. This bit is set if a read to a reserved location is detected. Software must write a one to this bit to clear it.

5.4.21 Peripheral Identification Registers

The table below specified the values of the Re-distributor Peripheral Identification Registers:

Offset	Register	Description
0xFFF0	GICR_CIDR0	• Component ID0. Value 0x0D.

0xFFF4	GICR_CIDR1	• Component ID1. Value 0xF0.
0xFFF8	GICR_CIDR2	• Component ID2. Value 0x05.
0xFFFC	GICR_CIDR3	• Component ID3. Value 0xB1.
0xFFE0	GICR_PIDR0	• Bits [7:0]. Bits [7:0] of the ARM-defined DevID field. This field is 0x93 in ARM implementations of a GICv3 or later Re-distributor.
0xFFE4	GICR_PIDR1	• Bits [7:4]. Bits[3:0] of the ARM defined JEP identity field. This field is 0xB in ARM implementations. • Bits [3:0]. Bits [11:8] of the ARM-defined DevID field. This field is 0x4 in ARM implementations of a GICv3 or later Re-distributor.
0xFFE8	GICR_PIDR2	• Bits [7:4]. ArchRev. This field has the following implementation defined values: o 0x1. GICv1 o 0x2. GICv2 o 0x3. GICv3. o 0x4. GICv4. o 0x5 – 0xf. Reserved • Bit [3]. UsesJEPcode. This field is 0b1 in ARM implementations. • Bits [2:0]. Bits[6:4] of the ARM defined JEP identity field. This field is 0x3 in ARM implementations.
0xFFEC	GICR_PIDR3	• Bits [7:4]. RevAnd. Manufacturer defined revision number. Normally zero. • Bits [3:0]. Customer modified.
0XFFD0	GICR_PIDR4	• Bits [7:4]. RES0. • Bits [3:0]. ContinuationCode. This field is 0x4 in ARM implementations.
0xFFD4	GICR_PIDR5	• Bits [7:0]. RES0.
0xFFD8	GICR_PIDR6	• Bits [7:0]. RES0.
0xFFDC	GICR_PIDR7	• Bits [7:0]. RES0.

Table 24: Re-distributor Peripheral Identification Registers

5.4.22 GICR_WAKER

This 32 bit secure register allows software to control the behaviour of the WakeRequest signal that corresponds to this re-distributor.

Note: the WakeRequest signal that indicates to the power controller that a re-distributor must be powered on is asserted by the re-distributor's parent. Hence, changes to the value of GICR_WAKER must be communicated to the distributor, including after reset.

In systems that support a single security state and when GICD_CTLR.DS is one, this register is accessible by any access.

In systems that support two security states when GICD_CTLR.DS is zero, this register is RAZ/WI for non-secure accesses.

- Bit [31]. Quiescent. Read only. Resets to zero. This bit indicates the re-distributor is quiescent and can be powered off. It is used in conjunction with the "Sleep" and "ProcessorSleep" bits below.
- Bit [30:3]. Reserved. RES0.
- Bit [2]. ChildrenAsleep. Read only. Resets to one.
    - 0b0. An interface to a child might still be active.
    - 0b1. The interfaces to all children are quiescent.
    - ***Note***: when ProcessorSleep is one, the Distributor treats the interrupt group enables as zero until a subsequent update to the enables is received (in implementations that use the Distributor interface defined in section 7 this corresponds to receipt of an Upstream Write Packet; see section 7.4.7). This means software must explicitly write to these enables when ChildrenAsleep becomes zero. See section 4.10.6.
- Bit [1]. ProcessorSleep. Resets to one. Affinity Level 0 only; RAZ/WI for other affinity levels.
    - 0b0. This re-distributor never asserts WakeRequest.
    - 0b1. This re-distributor must assert WakeRequest and hold interrupts as pending if an Enable bit is zero for an interrupt group and there is a pending interrupt for that group. See 4.10.2.
    - ***Note***: when "ProcessorSleep" is set to one, the re-distributor must ensure that any interrupts that are pending within the CPU interface are released (see section 7.5.3).
    - ***Note***: software must set "ProcessorSleep" to one and wait for "ChildrenAsleep" to become one before processor power off.
    - ***Note***: software must set "ProcessorSleep" to zero and wait for "ChildrenAsleep" to become zero following a processor power-on (or "aborted" power off).
- Bit [0]. Sleep. Resets to zero.
    - 0b0. The distributor never asserts WakeRequest.
    - 0b1. The distributor must assert WakeRequest and hold interrupts as pending. See section 4.10.2. It must also ensure that any caches within the re-distributor are consistent with external memory, including the contents of any implementation defined coarse-map (see section 4.8.5).
    - When a write changes this bit from one to zero, the re-distributor must load the pending table from memory to check for any pending interrupts
    - See section 4.10.7 for the requirements for updating this bit.
    - ***Note***: software must set "Sleep" to one and wait for "Quiescent" to become one before re-distributor power off.
    - ***Note***: in a monolithic implementation, the "Sleep" bits in each re-distributor access common state. That is, there is a "Sleep" bit that is used by all re-distributors.

To power down an affinity sub-tree, software must:

- Ensure no interrupts can be generated that directly target re-distributors within the sub-tree

- Set the “Sleep” bit for the re-distributor at the top of the sub-tree that will be powered off
    - This ensures that any caches within the re-distributor are consistent with external memory, including the contents of any implementation defined coarse-map (see section 4.8.5).
- Wait for “Quiescent” to become one in that re-distributor

See section 4.10 provides details of the requirements on software and the expected sequences for power-on and power-off.

Note: whenever the value of the “Sleep” bit is modified, the new value is communicated to the distributor.

Note: when this register is written with “Sleep” set to one, the re-distributor must ensure the contents of any interrupt caches (including any coarse-grained map) are consistent with memory.

Note: to ensure a re-distributor is quiescent in a distributed implementation, software must write to the register with “ProcessorSleep” set to one. Software must then poll the register until “ChildrenAsleep” reads as one.

5.4.23 GICR_PROPBASER

This 64 bit register specifies the address of the memory used to hold the LPI configuration table. The format of this register is shown below:

- Bit [63:48]. Reserved. RES0.
- Bits [47:12]. Physical Address. Bits [47:12] of the physical address containing the LPI configuration table.
- Bits [11:10]. Shareability. The shareability attributes of accesses to the properties table:
    - 0b00. Accesses are non-shareable.
    - 0b01. Accesses are inner-shareable.
    - 0b10. Accesses are outer-shareable.
    - 0b11. Reserved. Treated as 0b00.
    - In implementations where re-distributors cannot express shareability on the downstream bus system, this field is RAZ/WI.
- Bits [9:7]. Cacheability. The cacheability attributes of accesses to the properties table:
    - 0b000. Non-cacheable, non-bufferable.
    - 0b001. Non-cacheable.
    - 0b010. Read-allocate, Write-through.
    - 0b011. Read-allocate, Write-back.
    - 0b100. Write-allocate, Write-through.
    - 0b101. Write-allocate, Write-back.
    - 0b110. Read-allocate, Write-allocate, Write-through.
    - 0b111. Read-allocate, Write-allocate, Write-back.
- Bits [6:5]. Reserved. RES0.
- Bits [4:0]. IDbits. The number of bits of interrupt identifier supported, minus one, by the table that starts at “Physical Address”. ***Note***: if the value written exceeds the value of GICD_TYPER.IDbits, the number of bits must be treated as the value defined by GICD_TYPER.IDbits (see section 5.3.1).

The format of the configuration data for physical LPIs is described in section 4.8.4.

A re-distributor might cache this configuration data but it must ensure that:

- The effects of this caching are not visible to software except when reconfiguring an LPI, in which case an explicit invalidate command must be issued (e.g. an ITS INV command or a write to GICR_INVLPIR).
- ***Note***: this means Configuration data may not be speculatively cached unless hardware ensures the effects of this speculative caching are not visible to software.
- ***Note***: this means hardware must manage its caches automatically when moving interrupts

Note: for initialization of this register, see section 4.8.10.
Note: in a monolithic implementation, the GICR_PROPBASER registers in each re-distributor might access common state. That is, there might be a single properties base address that is used by all re-distributors. In distributed implementations, software must ensure that GICR_PROPBASER is programmed consistently in all re-distributors or the system behaviour might be unpredictable.
Note: it is expected that all re-distributors will share a single physical configuration table.

Note: a re-distributor will only read and must never write to the physical configuration table.
Note: an implementation might choose to make this register read-only (for example, to point to a configuration table in read-only memory).
Note: in implementations where the GICR_PROPBASER registers in each re-distributor ***do not*** access common state, this register is read-only when GICR_CTLR.EnableLPIs is one and GICR_WAKER.Quiescent is zero. See section 5.4.7.
Note: in implementations where the GICR_PROPBASER registers in each re-distributor access common state, this register is read-only when GICR_CTLR.EnableLPIs is one and GICR_WAKER.Quiescent is zero in ***any*** re-distributor. See section 5.4.7.

5.4.24 GICR_PENDBASER

This 64 bit register specifies the address and size of the memory used to hold the LPI pending table. The format of this register is shown below:

- Bits [63]. Reserved. RES0.
- Bits [62]. Pending Table Zero. This write-only field indicates whether the pending table is zero when GICR_CTLR.EnableLPIs becomes one. For reads, this bit is zero.
    - 0b0. The coarse-grained map is valid. The re-distributor may choose to rely on the coarse-grained map in memory or might choose to scan the table and build a coarse-grained map.
    - 0b1. The pending table is zero. Software must ensure the pending table is zeroed and the re-distributor may assume the table is all zeros and build an empty coarse-grained map.
- Bit [61:48]. Reserved. RES0.
- Bits [47:16]. Physical Address. Bits [47:16] of the physical address containing the LPI pending table.
- Bits [15:12]. Reserved. RES0.
- Bits [11:10]. Shareability. The shareability attributes of accesses to the pending table:
    - 0b00. Accesses are non-shareable.
    - 0b01. Accesses are inner-shareable.
    - 0b10. Accesses are outer-shareable.
    - 0b11. Reserved. Treated as 0b00.
    - In implementations where re-distributors cannot express shareability on the downstream bus system, this field is RAZ/WI.
- Bits [9:7]. Cacheability. The cacheability attributes of accesses to the pending table:
    - 0b000. Non-cacheable, non-bufferable.
    - 0b001. Non-cacheable.
    - 0b010. Read-allocate, Write-through.
    - 0b011. Read-allocate, Write-back.
    - 0b100. Write-allocate, Write-through.
    - 0b101. Write-allocate, Write-back.
    - 0b110. Read-allocate, Write-allocate, Write-through.
    - 0b111. Read-allocate, Write-allocate, Write-back.
- Bits [6:0]. Reserved. RES0.

The format of the pending data for physical LPIs is described in section 4.8.5.
Note: for initialization of this register, see section 4.8.10.
Note: each re-distributor must be provided with a separate physical pending table by software.
Note: a re-distributor may read and write to the physical pending table.
Note: in a monolithic implementation, the Shareability and Cachability fields in GICR_PENDBASER registers in each re-distributor might access common state.
Note: when GICR_CTLR.EnableLPIs is one and GICR_WAKER.Quiescent is zero, this register is read-only. See section 5.4.7.

5.4.25 GICR_SETLPIR

This 64 bit register is write-only; when written with a 32 bit write the data is zero extended to 64 bits. The value written to this register specifies which physical LPI to set to pending. The format of the value written is shown below:

- Bits [63:32]. Reserved. RES0
- Bits [31:0]. Physical ID. The ID of the physical LPI to be generated.
    - ***Note***: the number of implemented Physical ID bits is implementation defined. Unimplemented bits are RES0. The number of implemented bits can be discovered from GICD_TYPER.IDbits (see section 5.3.1).

If the LPI is already pending, the write has no effect.

If the value specifies an unimplemented LPI, the write has no effect.

If GICR_CTLR.EnableLPIs is zero, the write has no effect.

Note: this register is provided to enable the generation of physical LPIs directly by peripherals and the ITS.

Note: this register is mandatory in distributed implementations and might optionally be provided in monolithic implementations (see section 4.9.1).

5.4.26 GICR_CLRLPIR

This 64 bit register is write-only; when written with a 32 bit write the data is zero extended to 64 bits. The value written to this register specifies which physical LPI to set as not pending. The format of the value written is shown below:

- Bits [63:32]. Reserved. RES0
- Bits [31:0]. Physical ID. The ID of the physical LPI to be set as “not pending”.
    - ***Note***: the number of implemented Physical ID bits is implementation defined. Unimplemented bits are RES0. The number of implemented bits can be discovered from GICD_TYPER.IDbits (see section 5.3.1).

If the LPI is already “not pending”, the write has no effect.

If the value specifies an unimplemented LPI, the write has no effect.

If GICR_CTLR.EnableLPIs is zero, the write has no effect.

Note: this register is mandatory in distributed implementations and might optionally be provided in monolithic implementations (see section 4.9.1).

5.4.27 GICR_MOVLPIR

This 64 bit register is write-only. The value written to this register specifies the base address of the re-distributor to which a physical LPI is to be moved. The value also specifies the ID of the physical LPI. The format of the value written is shown below:

- Bits [63:32]. Target Address[47:16]. The base address of the re-distributor to which the physical LPI is to be moved.
- Bits [31:0]. Physical ID. The ID of the physical LPI to be “moved” to the re-distributor at “Target Address”.
    - ***Note***: the number of implemented Physical ID bits is implementation defined. Unimplemented bits are RES0. The number of implemented bits can be discovered from GICD_TYPER.IDbits (see section 5.3.1).

If the LPI specified by Physical ID is “not pending”, the write invalidates any cached data associated with the configuration of Physical ID.

If the LPI specified by Physical ID is “pending”, the re-distributor performs the following actions:

- If an implementation supports speculative caching of Configuration data, it must issue a strongly-ordered write of “Physical ID” to the GICR_INVLPIR register in the re-distributor specified by “Target Address”. ***Note***: this relies on all re-distributors within the system providing a 64 bit GICR_INVLPIR register at offset 0x00A0.

- Issue a strongly-ordered write of “Physical ID” to the GICR_SETLPIR register in the re-distributor specified by “Target Address”. ***Note***: this relies on all re-distributors within the system providing a 32 bit GICR_SETLPIR register at offset 0x0040.
- Clear the “pending” state for Physical ID in the local re-distributor and invalidates any cached data associated with the configuration of Physical ID.

If the value specifies an unimplemented LPI, the write has no effect.

If GICR_CTLR.EnableLPIs is zero, the write has no effect.

Note: this register is mandatory in distributed implementations and might optionally be provided in monolithic implementations (see section 4.9.1).

5.4.28 GICR_MOVALLR

This 64 bit register is write-only. The value written to this register specifies the base address of the re-distributor to which all physical LPIs are to be moved. The format of the value written is shown below:

- Bits [63:32]. Target Address[47:16]. The base address of the re-distributor to which pending physical LPIs are to be moved.
- Bits [31:0]. Reserved. RES0.

For each physical LPI, if the LPI specified by Physical ID is “not pending”, the write invalidates any cached data associated with the configuration of physical LPIs.

For each physical LPI, if the LPI specified by Physical ID is “pending”, the re-distributor performs the following actions:

- If an implementation supports speculative caching of Configuration data, it must issue a strongly-ordered write of “Physical ID” to the GICR_INVLPIR register in the re-distributor specified by “Target Address”. ***Note***: this relies on all re-distributors within the system providing a 64 bit GICR_INVALLIR register at offset 0x00A0.
- Issue a strongly ordered write of “Physical ID” to the GICR_SETLPIR register in the re-distributor specified by “Target Address”. ***Note***: this relies on all re-distributors within the system providing a 32 bit GICR_SETLPIR register at offset 0x0040.
- Clear the “pending” state for Physical ID in the local re-distributor and invalidates any cached data associated with the configuration of physical LPIs.

If GICR_CTLR.EnableLPIs is zero, the write has no effect.

Note: this register is mandatory in distributed implementations and might optionally be provided in monolithic implementations (see section 4.9.1).

5.4.29 GICR_INVLPIR

This 64 bit register is write-only; when written with a 32 bit write the data is zero extended to 64 bits. The value written to this register specifies which physical LPI for which cached configuration data must be invalidated. The format of the value written is shown below:

- Bits [63:32]. Reserved. RES0
- Bits [31:0]. Physical ID. The ID of the physical LPI to be cleaned.
    - ***Note***: the number of implemented Physical ID bits is implementation defined. Unimplemented bits are RES0. The number of implemented bits can be discovered from GICD_TYPER.IDbits (see section 5.3.1).

If the LPI specified by Physical ID is not in any local re-distributor cache, the write has no effect.

If the LPI specified by Physical ID is in a local re-distributor cache, the write invalidates any cached data associated with the configuration of Physical ID. ***Note***: if the LPI is “pending” the configuration data might be immediately re-loaded from the physical configuration table (see section 5.4.23).

If the value specifies an unimplemented LPI, the write has no effect.

Note: this register is mandatory in distributed implementations and might optionally be provided in monolithic implementations (see section 4.9.1).

5.4.30 GICR_INVALLR

This 64 bit register is write-only; when written with a 32 bit write the data is zero extended to 64 bits. The value written to this register specifies that any caching associated with the configuration of physical LPIs must be invalidated. The format of the value written is shown below:

- Bits [63:32]. Reserved. RES0
- Bits [31:0]. Reserved. RES0.

The write only affects physical LPIs currently held within any local re-distributor cache.

For all physical LPIs in any local re-distributor cache, the write invalidates any cached data associated with the configuration of physical LPIs.

Note: this register is mandatory in distributed implementations and might optionally be provided in monolithic implementations (see section 4.9.1).

5.4.31 GICR_SYNCR

This 32 bit read-only register is used to ensure completion of physical re-distributor operations. The register has the following format:

- Bits [31:1]. Reserved. RES0.
- Bit [0]. Busy. RES0.

When this register is read, it will only return read-data with “Busy” as zero when none the following operations are in progress:

- Any writes to GICR_CLRLPIR (see section 5.4.26).
- Any writes to GICR_MOVLPIR (see section 5.4.27).
- Any writes to GICR_MOVALLR (see section 5.4.28).
- Any writes to GICR_INVLPIR (see section 5.4.29).
- Any writes to GICR_INVALLR (see section 5.4.30).
- Any writes to another re-distributor performed as a consequence of a previous write to GICR_MOVLPIR or GICR_MOVALLR have completed and arrived at the target re-distributor.
- ***Note***: this also includes completion of any operations initiated by writing to GICR_PENDBASER (see section 5.4.24) or GICR_PROPBASER (see section 5.4.23).

Note: this register is mandatory in distributed implementations and might optionally be provided in monolithic implementations (see section 4.9.1).

Note: an implementation might choose to wait until none of the actions listed above are still in progress and always return zero. Such an implementation must obey the rules specified in section 4.11.3.

5.4.32 GICR_VPROPBASER

In GICv4, this 64 bit register specifies the address and size of the memory used to hold the virtual LPI configuration table for the currently resident virtual machine. The format of this register is shown below:

- Bit [63:48]. Reserved. RES0.
- Bits [47:12]. Physical Address. Bits [47:12] of the physical address containing the virtual LPI configuration table.
- Bits [11:10]. Shareability. The shareability attributes of accesses to the virtual properties table:
    - 0b00. Accesses are non-shareable.
    - 0b01. Accesses are inner-shareable.
    - 0b10. Accesses are outer-shareable.
    - 0b11. Reserved. Treated as 0b00.
    - In implementations where re-distributors cannot express shareability on the downstream bus system, this field is RAZ/WI.
- Bits [9:7]. Cacheability. The cacheability attributes of accesses to the virtual properties table:
    - 0b000. Non-cacheable, non-bufferable.
    - 0b001. Non-cacheable.
    - 0b010. Read-allocate, Write-through.
    - 0b011. Read-allocate, Write-back.
    - 0b100. Write-allocate, Write-through.
    - 0b101. Write-allocate, Write-back.
    - 0b110. Read-allocate, Write-allocate, Write-through.
    - 0b111. Read-allocate, Write-allocate, Write-back.
- Bits [6:5]. Reserved. RES0.
- Bits [4:0]. Number of Bits. The number of bits of virtual LPI identifier supported, minus one.

The format of the configuration data for virtual LPIs is described in section 4.8.4.
Note: for initialization of this register see sections 4.8.12 and 4.8.13.
Note: a re-distributor will only read and must never write to the virtual configuration table.

Note: this register is only provided in GICv4 implementations (see section 4.9.3).

5.4.33 GICR_VPENDBASER

In GICv4, this 64 bit register specifies the address and size of the memory used to hold the virtual LPI pending table for the currently resident virtual machine. The format of this register is shown below:

- Bit [63]. Valid. Resets to zero.
  - 0b0. The virtual LPI pending table is not valid and no virtual machine is considered to be resident.
  - 0b1. The virtual LPI pending table is valid.
  - ***Note***: when this bit is written to zero, the re-distributor must ensure external memory is consistent with any cached data.
  - ***Note***: software must poll GICR_CTLR.RWP until the bit reads as zero to ensure external memory is consistent with any caches.
- Bits [62]. Coarse Grain Map Invalid. This field indicates whether the implementation defined coarse-grained map is valid when the register has been updated with a new physical address.
  - 0b0. The coarse-grained map is valid
  - 0b1. The coarse-grained map is invalid and the entire pending table must be read. Once the pending table has been read and a coarse=grained map construsted, this bit will become zero.
- Bits [61:48]. Reserved. RES0.
- Bits [47:16]. Physical Address. Bits [47:16] of the physical address containing the LPI pending table.
- Bits [15:12]. Reserved. RES0.
- Bits [11:10]. Shareability. The shareability attributes of accesses to the virtual pending table:
  - 0b00. Accesses are non-shareable.
  - 0b01. Accesses are inner-shareable.
  - 0b10. Accesses are outer-shareable.
  - 0b11. Reserved. Treated as 0b00.
  - In implementations where re-distributors cannot express shareability on the downstream bus system, this field is RAZ/WI.
- Bits [9:7]. Cacheability. The cacheability attributes of accesses to the virtual pending table:
  - 0b000. Non-cacheable, non-bufferable.
  - 0b001. Non-cacheable.
  - 0b010. Read-allocate, Write-through.
  - 0b011. Read-allocate, Write-back.
  - 0b100. Write-allocate, Write-through.
  - 0b101. Write-allocate, Write-back.
  - 0b110. Read-allocate, Write-allocate, Write-through.
  - 0b111. Read-allocate, Write-allocate, Write-back.
- Bits [6:0]. Reserved. RES0.

The format of the pending data for virtual LPIs is described in section 4.8.5.
When this register is written with the valid bit set to one, the re-distributor will perform a “clean all” operation, as specified for GICR_VINVALLR below (see section 5.4.41).
Note: unlike for physical LPIs a “valid” bit is required. The tables for Physical LPIs must always be valid when physical LPIs are enabled. Virtual LPIs might still be received and might update tables in memory even when no guest operating system is considered as resident.
Note: when the “Valid” bit is cleared or the register is re-written with a different (valid) physical address, the re-distributor must ensure any outstanding pending virtual interrupts are cleared from the CPU interface (by issuing appropriate VClear packets; see section 7.3.6) and that the associated doorbell interrupts are set pending.
Note: for initialization of this register see sections 4.8.12 and 4.8.13.
Note: a re-distributor may read and write to the virtual pending table.
Note: this register is only provided in GICv4 implementations (see section 4.9.3).

5.4.34 GICR_VSETLPIR

In GICv4, this 64 bit register is write-only. The value written to this register specifies the virtual machine and which virtual LPI to set to “pending” and which physical interrupt to set pending if the virtual machine is not resident. The format of the value written is shown below:

- Bits [63:32]. VPT. The base address of the Virtual Pending Table.
- Bits [31:16]. Physical ID. The ID of the physical interrupt to be generated if the virtual machine specified by VPT is not resident.
- Bits [15:0]. Virtual ID. The ID of the virtual LPI to be generated for the virtual machine specified by the VPT.

Note: if a Physical or Virtual ID greater than 65,535 is required, GICR_VSETLPILRn (and GICR_VDESTRn) must be used. See section 5.4.35.

If ARE is zero for the non-secure state, the write has no effect.

The target virtual machine is considered to be “resident” if the value of VPT is equal to GICR_VPENDBASER.PhysicalAddress and GICR_VPENDBASER.Valid is one.

Regardless of the residency of the virtual machine, the re-distributor performs the following actions:

- If the virtual LPI specified by Virtual ID is already pending, the write has no effect.
- If the Virtual ID specifies an unimplemented virtual LPI, the write has no effect.
- Otherwise, the virtual LPI corresponding to the Virtual ID is set as pending in the VPT

If the target virtual machine is not resident then the re-distributor performs the following actions:

- If the physical LPI specified by Physical ID is already pending, the write has no effect.
- If the Physical ID specifies a spurious interrupt (i.e. Physical ID is 1023), the write has no effect
- If the Physical ID specifies an unimplemented physical LPI, the write has no effect.
- Otherwise, the physical LPI corresponding to the Physical ID is set as pending

Note: writes to this register must also update any implementation defined coarse-map (see section 4.8.5)

Note: this register is provided to enable the direct injection of virtual interrupts by the ITS.

Note: this register is only provided in GICv4 distributed implementations (see section 4.9.1).

5.4.35 GICR_VSETLPILRn

In GICv4, this 64 bit register is write-only. The value written to this register specifies which virtual LPI to set to "pending" for the virtual machine specified by the value of GICR_VDESTRn and which physical interrupt to set pending if the virtual machine is not resident. The format of the value written is shown below:

- Bits [63:32]. Physical ID. The ID of the physical interrupt to be generated if the virtual machine specified by VPT is not resident.
    - ***Note***: the number of implemented Physical ID bits is implementation defined. Unimplemented bits are RES0. The number of implemented bits can be discovered from GICD_TYPER.IDbits (see section 5.3.1).
- Bits [31:0]. Virtual ID. The ID of the virtual LPI to be generated for the virtual machine specified by the VPT.
    - ***Note***: the number of implemented Virtual ID bits is implementation defined. Unimplemented bits are RES0. The number of implemented bits can be discovered from GICD_TYPER.IDbits (see section 5.3.1).

If ARE is zero for the non-secure state, the write has no effect.

The target virtual machine is considered to be "resident" if the value of GICR_VDESTRn is equal to GICR_VPENDBASER.PhysicalAddress and GICR_VPENDBASER.Valid is one.

Regardless of the residency of the virtual machine, the re-distributor performs the following actions:

- If the virtual LPI specified by Virtual ID is already pending, the write has no effect.
- If the Virtual ID specifies an unimplemented virtual LPI, the write has no effect.
- Otherwise, the virtual LPI corresponding to the Virtual ID is set as pending in the VPT

If the target virtual machine is not resident then the re-distributor performs the following actions:

- If the physical LPI specified by Physical ID is already pending, the write has no effect.
- If the Physical ID specifies a spurious interrupt (i.e. Physical ID is 1023), the write has no effect
- If the Physical ID specifies an unimplemented physical LPI, the write has no effect.
- Otherwise, the physical LPI corresponding to the Physical ID is set as pending

Note: writes to this register must also update any implementation defined coarse-map (see section 4.8.5)

Note: this register is provided to enable the direct injection of virtual interrupts by the ITS.

Note: this register is only provided in GICv4 distributed implementations (see section 4.9.1).

Note: in systems with more than one ITS, each ITS must use a unique set of GICR_VSETLPILRn and GICR_VDESTRn registers.

5.4.36 GICR_VCLRLPIR

In GICv4, this 64 bit register is write-only. The value written to this register specifies the virtual machine and which virtual LPI to set as “not pending”. The format of the value written is shown below:

- Bits [63:32]. VPT. The base address of the Virtual Pending Table.
- Bits [31:0]. Virtual ID. The ID of the virtual LPI to be generated for the virtual machine specified by the VPT.
    - ***Note***: the number of implemented Virtual ID bits is implementation defined. Unimplemented bits are RES0. The number of implemented bits can be discovered from GICD_TYPER.IDbits (see section 5.3.1).

If ARE is zero for the non-secure state, the write has no effect.

Regardless of the residency of the virtual machine, the re-distributor performs the following actions:

- If the virtual LPI specified by Virtual ID is already pending, the write has no effect.
- If the Virtual ID specifies an unimplemented virtual LPI, the write has no effect.
- Otherwise, the virtual LPI corresponding to the Virtual ID is set as pending in the VPT

Note: writes to this register must also update any implementation defined coarse-map (see section 4.8.5)

Note: this register is only provided in GICv4 distributed implementations (see section 4.9.1).

5.4.37 GICR_VSRCRn

In GICv4, this 64 bit register specifies the address of the memory used as the source Virtual Pending Table for “move” operations (see section 4.9.7); when written with a 32 bit write the data is zero extended to 64 bits. The format of this register is shown below:

- Bits [63:48]. Reserved. RES0.
- Bits [47:16]. Physical Address. Bits [47:16] of the physical address containing the Virtual Pending Table.
- Bits [15:0]. Reserved. RES0.

Note: this register is only provided in GICv4 distributed implementations (see section 4.9.1).
Note: in systems with more than one ITS, each ITS must use a unique set of GICR_VMOVLPIRn, GICR_VSRCRn and GICR_VDESTRn registers.

5.4.38 GICR_VDESTRn

In GICv4, this 64 bit register specifies the address of the memory used as the destination Virtual Pending Table for “move” operations (see section 4.9.7)); when written with a 32 bit write the data is zero extended to 64 bits. The format of this register is shown below:

- Bits [63:48]. Reserved. RES0.
- Bits [47:16]. Physical Address. Bits [47:16] of the physical address containing the Virtual Pending Table.
- Bits [15:0]. Reserved. RES0.

Note: this register is only provided in GICv4 distributed implementations (see section 4.9.1).
Note: in systems with more than one ITS, each ITS must use a unique set of GICR_VMOVLPIRn, GICR_VSRCRn and GICR_VDESTRn registers.

5.4.39 GICR_VMOVLPIRn

In GICv4, this 64 bit register is write-only. The value written to this register specifies the base address of the re-distributor to which a virtual LPI is to be moved. The value also specifies the ID of the virtual LPI. The format of the value written is shown below:

- Bits [63:32]. Target Address[47:16]. The base address of the re-distributor to which the virtual LPI is to be moved.
- Bits [31:0]. Virtual ID. The ID of the virtual LPI to be "moved" to the re-distributor at "Target Address".
    - ***Note***: the number of implemented Virtual ID bits is implementation defined. Unimplemented bits are RES0. The number of implemented bits can be discovered from GICD_TYPER.IDbits (see section 5.3.1).

If ARE is zero for the non-secure state, the write has no effect.

If the LPI specified by Virtual ID is "not pending" in the source VPT (see section 5.4.37), the write invalidates any cached data associated with the configuration of Virtual ID.

If the LPI specified by Virtual ID is "pending" in the source VPT (see section 5.4.37), the re-distributor performs the following actions:

- In implementations supporting 16 bit or smaller interrupt identifiers:
    - If an implementation supports speculative caching of Configuration data, it must issue a strongly-ordered write of "VPT" and "Virtual ID" to the GICR_VINVLPIR register in the re-distributor specified by "Target Address". ***Note***: this relies on all re-distributors within the system providing a 64 bit GICR_VINVLPIR register at offset 0x00A0.
    - Issue a write of the destination VPT (see section 5.4.38) and the "Virtual ID" to the GICR_VSETLPIR register in the re-distributor specified by "Target Address".
    - ***Note***: the "Physical ID" must be set to 1023
    - ***Note***: this relies on all re-distributors within the system providing a 64 bit GICR_VSETLPIR register at offset 0x0040.
- In implementations supporting interrupt identifiers larger than 16 bits:
    - If an implementation supports speculative caching of Configuration data, it must issue a strongly-ordered write of "VPT" and "Virtual ID" to the GICR_VINVLPIR register in the re-distributor specified by "Target Address". ***Note***: this relies on all re-distributors within the system providing a 64 bit GICR_VINVLPIR register at offset 0x00A0.
    - Issue a write of the "Virtual ID" to the GICR_VSETLPILRn register in the re-distributor specified by "Target Address".
    - ***Note***: the ITS must have previously set GICR_VDESTRn in the re-distributor specified by "Target Address"
    - ***Note***: the "Physical ID" must be set to 1023
- Clear the "pending" state for Virtual ID in the source VPT (see section 5.4.37) and invalidates any cached data associated with the configuration of Virtual ID.

If the value specifies an unimplemented virtual LPI, the write has no effect.

Note: this register is only provided in GICv4 distributed implementations (see section 4.9.1).

Note: in systems with more than one ITS, each ITS must use a unique set of GICR_VMOVLPIRn, GICR_VSRCRn and GICR_VDESTRn registers.

5.4.40 GICR_VINVLPIR

In GICv4, this 64 bit register is write-only. The value written to this register specifies which virtual LPI to be "cleaned" to a specified VPT. The format of the value written is shown below:

- Bits [63:32]. VPT. The base address of the Virtual Pending Table.

- Bits [31:0]. Virtual ID. The ID of the virtual LPI to be cleaned to VPT.
    - ***Note***: the number of implemented Virtual ID bits is implementation defined. Unimplemented bits are RES0. The number of implemented bits can be discovered from GICD_TYPER.IDbits (see section 5.3.1).

If ARE is zero for the non-secure state, the write has no effect.

If the LPI specified by Virtual ID is not in any local re-distributor cache, the write has no effect.

If the LPI specified by Virtual ID is in a local re-distributor cache, the write causes the pending state to be updated in the specified VPT, and causes the interrupt configuration to be re-loaded from the virtual configuration table (see section 5.4.32) if the virtual machine is resident.

If the Virtual ID specifies an unimplemented LPI, the write has no effect.

Note: the contents of any implementation defined coarse-map must also be made consistent with memory (see section 4.8.5).

Note: this register is only provided in GICv4 distributed implementations (see section 4.9.1).

5.4.41 GICR_VINVALLR

In GICv4, this 64 bit register is write-only. The value written to this register specifies that the state of all virtual LPIs is to be "cleaned" to a specified VPT. The format of the value written is shown below:

- Bits [63:32]. VPT. The base address of the Virtual Pending Table.
- Bits [31:0]. Reserved. RES0.

If ARE is zero for the non-secure state, the write has no effect.

The write only affects virtual LPIs currently held within any local re-distributor cache.

For all virtual LPIs for the specified VPT in any local re-distributor cache, the write causes the pending state to be updated in the specified VPT, and causes the interrupt configuration to be re-loaded from the virtual configuration table (see section 5.4.32) if the virtual machine is resident.

Note: this register is only provided in GICv4 distributed implementations (see section 4.9.1).

5.4.42 GICR_VSYNCR

In GICv4, this 32 bit read-only register is used to ensure completion of virtual re-distributor operations. The register has the following format:

- Bits [31:1]. Reserved. RES0.
- Bit [0]. Busy. RES0.

When this register is read, it will only return read-data with "Busy" as zero when all none the following operations are still in progress:

- Any writes to GICR_VCLRLPIR (see section 5.4.36).
- Any writes to GICR_VMOVLPIR (see section 5.4.39).
- Any writes to another re-distributor performed as a consequence of a previous write to GICR_VMOVLPIR have completed and arrived at the target re-distributor.
- Any writes to GICR_VINVLPIR (see section 5.4.40).
- Any writes to GICR_VINVALLR (see section 5.4.41). ***Note***: this also includes completion of any "clean all" operation initiated by writing to GICR_VPENDBASER (see section 5.4.33).

Note: this register is only provided in GICv4 distributed implementations (see section 4.9.1).

Note: an implementation might choose to wait until none of the actions listed above are still in progress and always return zero. Such an implementation must obey the rules specified in section 4.11.3.

5.4.43 GICR_ITSSYNCRn

In GICv4, this 64 bit register is used to enable the synchronization of multiple ITS.

- Bits [63:48]. Reserved. RES0.
- Bits [47:32]. Sequence Number. A value that uniquely identifies a set of VMOVP commands associated with moving a particular VCPU.
- Bits [31:16]. Reserved. RES0.
- Bits [15:0]. ITS List. A value specifying which ITS are included in the synchronization operation. Each bit corresponds to an ITS where bit ‘n’ corresponds to the ITS with GITS_CTLR.ITS_Number == ‘n’.

When this register is written, the following actions must be performed by the re-distributor:

- The storage corresponding to the “Sequence Number” and “ITS List” fields must be updated, then
- For each GICR_ITSSYNCRn register including the register being written, bit “n” must be set to zero if the “Sequence Number” is equal to the “Sequence Number” provided in the write data. ***Note***: this includes the bit corresponding to the register being written.

Note: this register is only provided in GICv4 implementations (see section 4.9.3).

5.5 Access to CPU Interface Registers

For performance, GICv3 provides system register access to the CPU interface registers. To allow for GICv2 compatibility, memory mapped access is still provided and this is fully compatible with GICv2 except as described in section 3.3.1.

System register access to the CPU interface registers is enabled by setting to the appropriate System Register Enable (SRE) bit in ICC_SRE_ELx to one (see below).

When the system register access to the CPU interface registers is enabled for an interrupt regime, the registers have been defined to follow the ARMv8 exception level convention.

The system registers that provide equivalent functions for each memory mapped register are shown below:

- GICC_IAR and GICC_AIAR. Equivalent function provided by ICC_IAR0_EL1 and ICC_IAR1_EL1 for Group 0 and Group 1 interrupts respectively.
- GICC_EOIR and GICC_AEOIR. Equivalent function provided by ICC_EOIR0_EL1 and ICC_EOIR1_EL1 for Group 0 and Group 1 interrupts respectively.
- GICC_HPPIR and GICC_AHPPIR. Equivalent function provided by ICC_HPPIR0_EL1 and ICC_HPPIR1_EL1 for Group 0 and Group 1 interrupts respectively.
- GICC_BPR and GICC_ABPR. Equivalent function provided by ICC_BPR0_EL1 and ICC_BPR1_EL1 for Group 0 and Group 1 interrupts respectively.
- GICC_PMR. Equivalent function provided by ICC_PMR_EL1. ***Note***: this register and its memory mapped equivalent, GICC_PMR might access the same state. See section 5.7.27 for details.
- GICC_DIR. Equivalent function provided by ICC_DIR_EL1.

Processor interrupt exceptions may be routed to different exception levels as shown in the table in section 3.5.3.3.1 of [4]. See 4.4.1 for how the exception level to which interrupts are routed affects the accessibility of system registers.

5.5.1 Affect of ARE on Interrupt ID in CPU Interface Registers

The setting of ARE for a security state governs the format of the Interrupt ID presented by the Distributor to each CPU interface. This, in turn, affects the values reads from CPU interface registers that contains an Interrupt ID. The affected registers are:

- GICC_IAR, GICC_AIAR, ICC_IAR0_EL1 and ICC_IAR1_EL1.
- GICC_EOIR, GICC_AEOIR, ICC_EOIR0_EL1 and ICC_EOIR1_EL1.
- GICC_HPPIR, GICC_AHPPIR, ICC_HPPIR0_EL1 and ICC_HPPIR1_EL1.
- GICC_DIR and ICC_DIR_EL1.

When ARE is zero for a security state, the format of these registers is:

- Bits [31:13]. Reserved. RES0.
- Bits [12:10]. Source CPU ID. Only valid for Interrupt IDs 0 to 15; otherwise RES0.
- Bits [9:0]. Interrupt ID. ***Note***: IDs 1020 – 1023 are reserved and convey additional information such as a spurious interrupts.

When ARE is set for a security state, the format of these registers is:

- Bits [31:24]. Reserved. RES0.
- Bits [23:0]. Interrupt ID.
    - The number of implemented bits is discoverable from ICC_CTLR_EL1/3 (see sections 5.7.33 and 5.7.34). Unimplemented bits are RES0.
    - ***Note***: IDs 1020 – 1023 are reserved and convey additional information such as a spurious interrupts.

Note: except for (secure) EL3, interrupt regimes where SRE is set but where ARE is zero for the security state is not supported.

Note: it is expected that the Distributor always communicates (and interprets) Interrupt IDs consistent with the ARE settings for a security state.

5.5.2 Access to Memory-Mapped Registers When SRE is Set

Because memory mapped accesses and system register accesses might not access the same state and are not guaranteed to be synchronized, when SRE is set for an interrupt regime, the memory mapped registers should not be used and the system register that provides equivalent function should be used instead.

In systems that support asymmetric SRE settings where the secure copy of ICC_SRE_EL1.SRE is programmable, the following state must be shared between system register access and memory mapped access to ensure that preemption operates correctly:

- GICC_PMR and ICC_PMR_EL1 must access the same state.
- GICC_APR / GICC_NSAPR and ICC_AP0Rn_EL1 / non-secure ICC_AP1Rn_EL1 must access the same state.
- GICC_CTLR.CBPR and ICC_CTLR_EL3.CBPR_EL1NS must access the same state.
- Secure access to GICC_BPR and ICC_BPR0_EL1 must access the same state when ICC_CTLR_EL3.CBPR_EL1NS is one.

Note: software must follow the rules specified in section 4.4.8 when changing any SRE settings.

Note: memory mapped accesses and system register accesses might access the same state for registers in addition to those listed above. Section 4.4.1 specifies the relationships between memory-mapped registers and system registers and state may only be shared between registers that perform the same function.

An implementation might choose to make accesses to the memory-mapped CPU Interface registers subject to the SRE settings:

- If the secure copy of ICC_SRE_EL1.SRE is one, then the physical (GICC) memory-mapped registers might not be accessible or might be RAZ/WI. ***Note***: when EL3 is configured to use AArch32, the secure EL1

interrupt regime is inaccessible but software may still set the secure copy of ICC_SRE_EL1.SRE to one (to enable support for Secure Group 1 interrupts. See section 4.6.5).

- If ICC_SRE_EL2.SRE is one or is treated as one, then the virtual control (GICH) memory-mapped registers might not be accessible or might be RAZ/WI.
- If the non-secure copy of ICC_SRE_EL1.SRE is one or is treated as one, then the virtual (GICV) memory-mapped registers might not be accessible or might be RAZ/WI.

Note: in implementations where the non-secure copy of ICC_SRE_EL1.SRE is programmable (i.e. is not RAO/WI), a memory-mapped virtual (GICV) interface must still be provided.

An implementation might choose to detect accesses to memory mapped registers which should not be accessed because an SRE bit is one and report them in an implementation defined manner.

5.5.3 CPU Interface Register Reset Domain

All registers (memory-mapped and system registers) in the CPU interface logic belong to the processor's Warm reset domain (see ref [4]).

5.6 Memory Mapped Access to CPU Interface Registers

This section defines the memory mapped CPU interface registers and their behavior.

Note: when SRE is set for an interrupt regime, an implementation might choose to make these registers inaccessible (see section 5.5.2).

5.6.1 Reset Values

The table below defines the reset values of all the CPU interface system registers

Register	Reset Value	Name
GICC_IAR	-	Interrupt Acknowledge Register
GICC_AIAR	-	Aliased Interrupt Acknowledge Register
GICC_EOIR	-	End Of Interrupt Register
GICC_AEOIR	-	Aliased End Of Interrupt Register
GICC_HPPIR	-	Highest Priority Pending Interrupt Register
GICC_AHPPIR	-	Aliased Highest Priority Pending Interrupt Register
GICC_BPR	Implementation Defined. As GICv2.	Binary Point Register
GICC_ABPR	Implementation Defined. As GICv2.	Aliased Binary Point Register
GICC_DIR	-	Deactivate Interrupt Register
GICC_PMR	0x00	Priority Mask Register
GICC_RPR	-	Running Priority Register
GICC_APRn	0x00000000	Active Priorities Register n
GICC_NSAPRn	0x00000000	Non-secure Active Priorities Register n
GICC_CTLR	Implementation Defined. See section 5.6.18	CPU Interface Control Register Resets to zero unless an EOI mode bit is implemented as RAO/WI.
GICC_STATUSR	0x00000000	CPU Interface Status Register
GICC_IIDR	Implementation Defined. See section 5.6.15	CPU Interface Identification Register

Table 25: CPU interface memory mapped register reset values

5.6.2 GICC_IAR

When SRE is zero for an interrupt regime, this register behaves as defined for GICv2. A summary is shown below:

Security states	Security of access	Value returned
1	-	As GICv2 GICC_IAR; IAR for Group 0 interrupt
2	Secure	As GICv2 GICC_IAR; IAR for Group 0 interrupt
	Non-Secure	As GICv2 GICC_IAR; IAR for Group 1 interrupt

When SRE is set for an interrupt regime, memory-mapped GICC_IAR must not be used for that interrupt regime, and system registers ICC_IAR0_EL1 and ICC_IAR1_EL1 should be used to handle Group 0 and Group 1 interrupts respectively.

5.6.3 GICC_AIAR

When SRE is zero for an interrupt regime, this register behaves as shown below:

Security states	Security of access	Value returned
1	-	As GICv2 GICC_AIAR; IAR for Group 1 interrupt
2	-	IAR for Group 1 interrupt

When SRE is set for an interrupt regime, memory-mapped GICC_AIAR must not be used for that interrupt regime, and system register ICC_IAR1_EL1 provides the equivalent functions.

5.6.4 GICC_HPPIR

When SRE is zero for an interrupt regime, this register behaves as defined for GICv2. A summary is shown below:

Security states	Security of access	Value returned
1	-	As GICv2 GICC_HPPIR; Highest priority pending Group 0 interrupt
2	Secure	As GICv2 GICC_HPPIR; Highest priority pending Group 0 interrupt
	Non-Secure	As GICv2 GICC_HPPIR; Highest priority pending Group 1 interrupt

When SRE is set for an interrupt regime, memory-mapped GICC_HPPIR must not be used for that interrupt regime, and system registers ICC_HPPIR0_EL1 and ICC_HPPIR1_EL1 provide the equivalent functions.

Note: in GICv2 it was implementation defined whether disabled interrupt groups affected GICC_HPPIR. In GICv3, only enabled interrupt groups affect GICC_HPPIR.

5.6.5 GICC_AHPPIR

When SRE is zero for an interrupt regime, this register behaves as shown below:

Security states	Security of access	Value returned
1	-	As GICv2 GICC_AHPPIR; Highest priority pending Group 1 interrupt
2	-	Highest priority pending Group 1 interrupt

When SRE is set for an interrupt regime, memory-mapped GICC_AHPPIR must not be used for that interrupt regime, and system register ICC_HPPIR1_EL1 provides the equivalent function.

Note: in GICv2 it was implementation defined whether disabled interrupt groups affected GICC_AHPPIR. In GICv3, only enabled interrupt groups affect GICC_AHPPIR.

5.6.6 GICC_EOIR

When SRE is zero for an interrupt regime, this register behaves as defined for GICv2. A summary is shown below:

Security states	Security of access	Effect of the write
1	-	As GICv2 GICC_EOIR; EOI for Group 0 interrupt. Affects active priorities as if a secure EL1 write to ICC_EOIR0_EL1 had occurred.
2	Secure	As GICv2 GICC_EOIR; EOI for Group 0 interrupt. Affects active priorities as if a secure EL1 write to ICC_EOIR0_EL1 had occurred.
	Non-Secure	As GICv2 GICC_EOIR; EOI for Group 1 interrupt. Affects active priorities as if a non-secure EL1 write to ICC_EOIR1_EL1 had occurred.

When this register is written, the checks specified in section 5.7.9 must be performed with parameter `lpiAllowed` set to `true`.

In GICv2, there were three unpredictable cases when writing to GICC_EOIR:

- A secure write when identified interrupt is Group 1. GICv3 Distributor implementations must ignore such writes. In systems supporting system error generation, an implementation might generate an SEI. This means the behaviour of the GIC is predictable and the active priority for the interrupt will be reset. However, the system behaviour is UNPREDICTABLE.
- A non-secure write when EOImodeS == '1' and the identified interrupt is Group 0. GICv3 Distributor implementations must ignore such writes. In systems supporting system error generation, an implementation might generate an SEI. This means the behaviour of the GIC is predictable and the active priority for the interrupt will be reset. However, the system behaviour is UNPREDICTABLE.
- A write when the identified interrupt is NOT the same as the last acknowledged interrupt. This remains UNPREDICTABLE but it is expected that the CPU Interface logic will perform a "priority drop" (if security checks allow).

When SRE is set for an interrupt regime, memory-mapped GICC_EOIR must not be used for that interrupt regime, and system registers ICC_EOIR0_EL1 and ICC_EOIR1_EL1 provide the equivalent functions.

5.6.7 GICC_AEOIR

When SRE is zero for an interrupt regime, this register behaves as shown below:

Security states	Security of access	Effect of the write
1	-	As GICv2 GICC_AEOIR; EOI for Group 1 interrupt Affects active priorities as if a non-secure EL1 write to ICC_EOIR1_EL1 had occurred.
2	-	EOI for Group 1 interrupt Affects active priorities as if a non-secure EL1 write to ICC_EOIR1_EL1 had occurred.

When this register is written, the checks specified in section 5.7.9 must be performed with parameter `lpiAllowed` set to `true`.

In GICv2, there was an unpredictable case when writing to GICC_AEOIR:

- A write when the identified interrupt is NOT the same as the last acknowledged interrupt. This remains UNPREDICTABLE but it is expected that the CPU Interface logic will perform a “priority drop” (if security checks allow).

When SRE is set for an interrupt regime, memory-mapped GICC_AEOIR must not be used for that interrupt regime, and system register ICC_EOIR1_EL1 provides the equivalent function.

5.6.8 GICC_BPR

When SRE is zero for an interrupt regime, this register behaves as defined for GICv2. A summary is shown below:

Security states	Security of access	Value returned
1	-	As GICv2 GICC_BPR; Binary Point for Group 0 interrupts
2	Secure	As GICv2 GICC_BPR; Binary Point for Group 0 interrupts
	Non-Secure	As GICv2 GICC_BPR; Binary Point for Group 1 interrupts

When SRE is set for an interrupt regime, memory-mapped GICC_BPR must not be used for that interrupt regime, and system registers ICC_BPR0_EL1 and ICC_BPR1_EL1 provide the equivalent functions.

5.6.9 GICC_ABPR

When SRE is zero for an interrupt regime, this register behaves as shown below:

Security states	Security of access	Value returned
1	-	As GICv2 GICC_ABPR; Binary Point for Group 1 interrupts
2	Secure	As GICv2 GICC_ABPR; Binary Point for Group 1 interrupts
	Non-Secure	Binary Point for Group 1 interrupts (shifted value)

Note: GICC_ABPR resets to (minimum GICC_BPR + 1) as defined for GICv2.

Note: in GICv3 systems that support two security states, if ICC_CTLR_EL3.CBPR_EL1NS is set, secure reads and writes to GICC_ABPR access (and might modify) ICC_BPR0_EL1.

When SRE is set for an interrupt regime, memory-mapped GICC_ABPR must not be used, and system register ICC_BPR1_EL1 provides the equivalent function.

5.6.10 GICC_DIR

Because this register is at offset 0x1000 (i.e. the start of a new 4kB page), memory mapped access to the GIC must always use 4kB pages to allow a hypervisor to separately trap accesses to GICC_DIR.

In GICv2, there are three unpredictable cases when writing to GICC_DIR:

- When EOImode == ‘0’. GICv3 implementations must ignore such writes. In systems supporting system error generation, an implementation might generate a system error.
- When EOImode == ‘1’ but the interrupt is not “active” in the Distributor. GICv3 Distributor implementations must ignore such writes. In systems supporting system error generation, an implementation might generate an SEI. ***Note***: in implementations using the Distributor Interface defined in 7, these writes will correspond to a Deactivate packet (see section 7.4.5) for an interrupt that is not “active”.
- When EOImode == ‘1’ but no EOI has been issued. The behaviour of the GIC is predictable and the active priority for the interrupt will remain set (because no EOI was issued). However, the system behaviour is UNPREDICTABLE.

When SRE is zero for an interrupt regime, this register behaves as defined for GICv2, except the checks specified in section 5.7.9 must be performed with parameter `lpiAllowed` set to `false`.

If SRE is set for an interrupt regime, then GICC_DIR must not be used, and system register ICC_DIR_EL1 provides the equivalent function.

5.6.11 GICC_PMR

When SRE is zero for an interrupt regime, this register behaves as GICv2 and is COMMON.

When SRE is set for an interrupt regime, memory-mapped GICC_PMR must not be used for that interrupt regime, and system register ICC_PMR_EL1 provides the equivalent functions.

5.6.12 GICC_RPR

When SRE is zero for an interrupt regime, this register operates exactly as GICv2. ***Note***: the GICv2 glossary incorrectly states that the "Running Priority" is the group priority of the highest priority active interrupt. It is actually the priority (not group priority) of the current active interrupt.

If SRE is set to one for an interrupt regime, memory-mapped GICC_RPR must not be used for that interrupt regime, and system register ICC_RPR_EL1 provides the equivalent function.

5.6.13 GICC_APRn

The data values of these registers are implementation defined in GICv2. In GICv3, these registers follow the principles defined in section 4.4.12 of [2] but the data values must be consistent with the physical views of ICC_AP0Rn_EL1 and ICC_AP1Rn_EL1 as described in section 5.7.43.

Resets to zero.

Note: if GICD_CTLR.DS is set, GICC_APRn always presents an unshifted view of the active priorities.

Note: if the value of GICD_CTLR.DS is modified when there are active interrupts, the value of GICC_APRn is not changed. Hence it is recommended that GICD_CTLR.DS is only changed when there are no active interrupts otherwise software might not be able to correctly handle interrupts.

5.6.14 GICC_NSAPRn

The data values of these registers are implementation defined in GICv2. In GICv3, these registers follow the principles defined in section 4.4.13 of [2] but the data values must be consistent with the physical views of ICC_AP0Rn_EL1 and ICC_AP1Rn_EL1 as described in section 5.7.43 such that secure accesses to this register access the unshifted view of non-secure Group 1 priorities.

Resets to zero.

If GICD_CTLR.DS is set to one, this register is always accessible (regardless of the security state of the access) and provides access to the (Non-secure) Group 1 active priorities and GICC_APRn presents an unshifted view of the (Secure) Group 0 active priorities.

Note: in systems supporting two security states when GICD_CTLR.DS is zero, unlike the “alias” registers (GICC_A*), this register remains as secure access only.

5.6.15 GICC_IIDR

This register behaves exactly as GICv2, except the ArchRev field has been updated:

- Bits [19:16]. ArchRev. This field has the following implementation defined values:
  - 0x1. GICv1
  - 0x2. GICv2
  - 0x3. Reserved.
  - 0x4. GICv4 memory-mapped interface supported. ***Note***: support for the GIC system register interface is discoverable from processor registers ID_PFR1 and ID_AA64PFR0_EL1 (see sections 5.7.32 and 5.7.31).
  - 0x5 – 0xf. Reserved

5.6.16 GICC_STATUSR

This optional 32 bit register, which is banked by security, has been added to GICv3 to provide software with a mechanism to detect accesses to reserved locations, writes to read-only locations and reads of write-only locations.

If not implemented this register is RAZ / WI.

The register is at offset 0x002C in the CPU interface register map. Its format is shown below:

- Bits [31:5]. Reserved. RES0.
- Bit [4]. ASV. Attempted Security Violation. This bit is set if a non-secure access to a secure location is detected. ***Note***: it does NOT apply to registers where some fields are secure but others are non-secure.
- Bit [3]. WROD. This bit is set if write to a read-only location is detected. Software must write a one to this it to clear it.
- Bit [2]. RWOD. This bit is set if read to a write-only location is detected. Software must write a one to this bit to clear it.
- Bit [0]. WRD. This bit is set if a write to a reserved location is detected. Software must write a one to this bit to clear it.
- Bit [0]. RRD. This bit is set if a read to a reserved location is detected. Software must write a one to this bit to clear it.

Note: the register is banked because the checks performed depend on the ARE settings for a security state.

If SRE is set for an interrupt regime, memory-mapped GICC_STATUSR is not updated and the equivalent functionality is provided for system register accesses by appropriate traps and exceptions.

Note: if GICC_STATUSR is implemented by a GIC, GICV_STATUSR must also be implemented. See section 5.10.1.

5.6.17 Non-Secure Access to GICC_CTLR

When SRE is zero for an interrupt regime, the format of non-secure accesses to this register is shown below:

- Bit [31:10]. Reserved. RES0.
- Bit [9]. EOImodeNS. Controls the behavior of non-secure accesses to the GICC_EOIR, GICC_AEOIR and GICC_DIR registers:
  - 0b0. Non-secure accesses to GICC_EOIR and GICC_AEOIR have both priority drop and deactivate interrupt functionality. Accesses to the GICC_DIR are UNPREDICTABLE.
  - 0b1. Non-secure accesses to GICC_EOIR and GICC_AEOIR have priority drop functionality only. The GICC_DIR register has deactivate interrupt functionality.
  - ***Note***: an implementation might choose to make this field RAO/WI. However, such implementations must still ensure the setting of this field is communicated to all processors following a processor reset. See section 7.2.1.
- Bit [8:7]. Reserved. RES0.
- Bit [6]. IRQBypDisGrp1. If SRE is set for the EL3 interrupt regime and ICC_SRE_EL3.DIB is set to one, this is bit is RAO/WI. Otherwise, behaves as GICv2. ***Note***: if SRE is one for the Secure EL1 interrupt regime, this bit is ignored.
- Bit [5]. FIQBypDisGrp1. If SRE is set for the EL3 interrupt regime and ICC_SRE_EL3.DFB is set to one, this is bit is RAO/WI. Otherwise, behaves as GICv2. ***Note***: if SRE is one for the Secure EL1 interrupt regime, this bit is ignored.
- Bit [4]. Reserved. RES0.
- Bit [3:1]. Reserved. RES0.
- Bit [0]. EnableGrp1, non-secure. Resets to zero.

When SRE is set for an interrupt regime, memory-mapped GICC_CTLR must not be used for that interrupt regime, and system registers ICC_CTLR_EL1, ICC_CTLR_EL3, ICC_IGRPEN0_EL1 and ICC_IGRPEN1_EL1 provide the equivalent functions.

5.6.18 Secure Access to GICC_CTLR

When SRE is zero for an interrupt regime, the format of secure accesses to this register is shown below:

- Bit [31:11]. Reserved. RES0.
- Bit [10]. EOImodeNS. Controls the behavior of non-secure accesses to the GICC_EOIR, GICC_AEOIR and GICC_DIR registers:
  - 0b0. Non-secure accesses to GICC_EOIR and GICC_AEOIR have both priority drop and deactivate interrupt functionality. Accesses to the GICC_DIR are UNPREDICTABLE.
  - 0b1. Non-secure accesses to GICC_EOIR and GICC_AEOIR have priority drop functionality only. The GICC_DIR register has deactivate interrupt functionality.
  - ***Note***: an implementation might choose to make this field RAO/WI.
- Bit [9]. EOImodeS. Controls the behavior of secure accesses to the GICC_EOIR and GICC_DIR registers:
  - 0b0. Secure accesses to GICC_EOIR and GICC_AEOIR have both priority drop and deactivate interrupt functionality. Accesses to the GICC_DIR are UNPREDICTABLE.
  - 0b1. Secure accesses to GICC_EOIR and GICC_AEOIR have priority drop functionality only. The GICC_DIR register has deactivate interrupt functionality.
  - ***Note***: an implementation might choose to make this field RAO/WI.
- Bit [8]. IRQBypDisGrp1. If SRE is set for the EL3 interrupt regime and ICC_SRE_EL3.DIB is set to one, this is bit is RAO/WI. Otherwise, behaves as GICv2. ***Note***: if SRE is one for the Secure EL1 interrupt regime, this bit is ignored.
- Bit [7]. FIQBypDisGrp1. If SRE is set for the EL3 interrupt regime and ICC_SRE_EL3.DFB is set to one, this is bit is RAO/WI. Otherwise, behaves as GICv2. ***Note***: if SRE is one for the Secure EL1 interrupt regime, this bit is ignored.
- Bit [6]. IRQBypDisGrp0. If SRE is set for the EL3 interrupt regime and ICC_SRE_EL3.DIB is set to one, this is bit is RAO/WI. Otherwise, behaves as GICv2. ***Note***: if SRE is one for the Secure EL1 interrupt regime, this bit is ignored.
- Bit [5]. FIQBypDisGrp0. If SRE is set for the EL3 interrupt regime and ICC_SRE_EL3.DFB is set to one, this is bit is RAO/WI. Otherwise, behaves as GICv2. ***Note***: if SRE is one for the Secure EL1 interrupt regime, this bit is ignored.
- Bit [4]. CBPR.
- Bit [3]. FIQEn. Resets to zero.
- Bit [2]. Reserved. RES0. ***Note***: this bit was AckCtl in GICv2 and is removed in GICv3.
- Bit [1]. EnableGrp1, non-secure. Resets to zero.
- Bit [0]. EnableGrp0. Resets to zero.

When SRE is set for an interrupt regime, memory-mapped GICC_CTLR must not be used for that interrupt regime, and system registers ICC_CTLR_EL1, ICC_CTLR_EL3, ICC_IGRPEN0_EL1 and ICC_IGRPEN1_EL1 provide the equivalent functions.

Note: GICC_CTLR.CBPR is an alias of ICC_CTLR_EL3.CBPR_EL1NS.

5.7 System Register Access to CPU Interface Registers

This section defines the behavior of the CPU interface system registers.

5.7.1 Reset Values

The table below defines the reset values of all the CPU interface system registers

System Register			
AArch64	AArch32	Reset Value	Name
ICC_IAR0_EL1	ICC_IAR0	-	Interrupt Acknowledge Register 0
ICC_IAR1_EL1	ICC_IAR1	-	Interrupt Acknowledge Register 1
ICC_EOIR0_EL1	ICC_EOIR0	-	End Of Interrupt Register 0
ICC_EOIR1_EL1	ICC_EOIR1	-	End Of Interrupt Register 1
ICC_HPPIR0_EL1	ICC_HPPIR0	-	Highest Priority Pending Interrupt Register 0
ICC_HPPIR1_EL1	ICC_HPPIR1	-	Highest Priority Pending Interrupt Register 1
ICC_BPR0_EL1	ICC_BPR0	Implementation Defined. See section 5.7.26.	Binary Point Register 0
ICC_BPR1_EL1	ICC_BPR1	Implementation Defined. See section 5.7.25.	Binary Point Register 1
ICC_DIR_EL1	ICC_DIR	-	Deactivate Interrupt Register
ICC_PMR_EL1	ICC_PMR	0x00	Priority Mask Register
ICC_RPR_EL1	ICC_RPR	-	Running Priority Register
ICC_AP0R0_EL1	ICC_AP0R0	0x00000000	Active Priorities 0 Register 0
ICC_AP0R1_EL1	ICC_AP0R1	0x00000000	Active Priorities 0 Register 1
ICC_AP0R2_EL1	ICC_AP0R2	0x00000000	Active Priorities 0 Register 2
ICC_AP0R3_EL1	ICC_AP0R3	0x00000000	Active Priorities 0 Register 3
ICC_AP1R0_EL1	ICC_AP1R0	0x00000000	Active Priorities 1 Register 0
ICC_AP1R1_EL1	ICC_AP1R1	0x00000000	Active Priorities 1 Register 1
ICC_AP1R2_EL1	ICC_AP1R2	0x00000000	Active Priorities 1 Register 2
ICC_AP1R3_EL1	ICC_AP1R3	0x00000000	Active Priorities 1 Register 3
ICC_IGRPEN0_EL1	ICC_IGRPEN0	0x0	Interrupt Group Enable Register 0
ICC_IGRPEN1_EL1	ICC_IGRPEN1	0x0	Interrupt Group Enable Register 1
ICC_IGRPEN1_EL3	ICC_IGRPEN3	0x0	Interrupt Group Enable Register 1 for EL3
ICC_SRE_EL1	ICC_SRE	Implementation Defined. See section 5.7.38.	System Register Enable Register for EL1
ICC_SRE_EL2	ICC_HSRE	Implementation Defined. See section 5.7.39.	System Register Enable Register for EL2
ICC_SRE_EL3	ICC_MSRE	Implementation Defined. See section 5.7.40.	System Register Enable Register for EL3
ICC_CTLR_EL1	ICC_CTLR	Implementation Defined. See section 5.7.33.	Interrupt Control Register for EL1
ICC_CTLR_EL3	ICC_MCTLR	Implementation Defined. See section 5.7.34.	Interrupt Control Register for EL3
ICC_SGI0R_EL1	ICC_SGI0R	-	SGI Generation Register 0
ICC_SGI1R_EL1	ICC_SGI1R	-	SGI Generation Register 1
ICC_ASGI1R_EL1	ICC_ASGI1R	-	Alternate SGI Generation Register 1

Table 26: CPU interface system register reset values

5.7.2 Helper Functions

The following “helper” functions are used in the pseudocode that follows:

```
const integer NI = IMPLEMENTATION_DEFINED;        // The number of implemented Interrupt ID bits

struct lrType {
    bits(2)  State;
    bit      HW;
    bit      Group;
    bits(8)  Priority;
    bits(10) PhysicalID;  // When HW is zero, PhysicalID[9] is the "EOI" bit (section 5.9.7)
    bits(NI) VirtualID;
}
enumeration intGroup { None, Group0, Group1NS, Group1S };
lrType GICH_VLPIR;                            // Holds virtual LPIs received from the Distributor

UndefinedException();                         // Generate an UNDEF_cfg exception
HypervisorTrap();                             // Generate a HYP_TRAP exception
MonitorTrap();                                // Generate a MON_TRAP exception
boolean EL3IsAArch32();                       // Returns true if EL3 is using AArch32
boolean IsEL3OrMon();                         // Returns true if EL3 is using AArch32 and in Mon or
                                              // if EL3 is using AArch64 and PSTATE.EL == 3
boolean ELusingSysRegs(elType EL);            // Returns true if exception level EL is using
                                              // system registers
PermissionFailureException(integer targetEL); // Generate a permission failure exception

boolean IsSecureInt(bits(NI) ID);             // Returns true if GICD_CTLR.DS is zero and interrupt ‘ID’
                                              // is a group 0 or group 1 secure interrupt
boolean IsGrp0Int(bits(NI) ID);               // Returns true if interrupt ‘ID’ is group 0.
bits(7) PriorityGroup(bits(8) priority, intGroup group); // returns the priority group field for the
                                              // minimum BPR value for the group

bits(7) VPriorityGroup(bits(8) priority, bit group); // returns the priority group field for the
                                              // minimum BPR value for the group
boolean PriorityDrop(bits(7) priority, bits(128) p0); // Clears the highest priority bit set in the
                                              // supplied register; returns false if no bits were set.
boolean PriorityDrop2(bits(7) priority, bits(128) p0, bits(128) p1); // Clears the highest priority
                                              // bit set in the supplied registers; returns
                                              // false if no bits were set.
bits(7) GetHighestActivePriority(bits(128) p0, bits(128) p1, bits(128) p2); // Returns the index of
                                              // the highest priority bit set from three registers.
                                              // Returns 0x7f if no bits are set.
intGroup GetHighestActiveGroup(bits(128) p0, bits(128) p1, bits(128) p2); // Returns a value
                                              // indicating the interrupt group of the highest priority
                                              // bit set from three registers.
                                              // Returns “None” if no bits are set.

AcknowledgeInterrupt(bits(NI) ID);            // Acknowledges interrupt ‘ID’, sends an Activate pkt
                                              // and sets the appropriate ICC_AP{0,1}R_EL1 active
                                              // priority bit
AcknowledgeVInterrupt(bits(NI) ID);           // Acknowledges virtual interrupt ‘ID’
Deactivate (bits(NI) ID);                     // Deactivates interrupt ‘ID’ and sends a Deactivate pkt
bits(8) P_MASK =  ((0xff << (8 – HaveEL(El3) ? ICC_CTLR_EL3.PRIbits : ICC_CTLR_EL1.PRIbits)) & 0xff;
bits(8) VP_MASK = ((0xff << (8 - ICH_VTR_EL2.PRIbits)) & 0xff;
```

Note: the GICH_VLPIR register holds the highest priority virtual LPI received from the Distributor (by a “VSet” packet; see section 7.3.5):

- The register is not visible to software and there is only a single register i.e. only a single virtual LPI can be pending within the CPU interface
- Software must ensure that a virtual LPI ID is not pending in GICH_VLPIR and the List Registers simultaneously or it is UNPREDICTABLE whether both virtual LPIs are handled correctly

5.7.3 Secure Configuration Register Access

The pseudocode below shows the code that returns the value of the secure configuration register:

```
SCRType SCR_GEN()
    // Check if we have EL3 or not
    if (HaveEL(EL3)) then
        // We have EL3 so return the register appropriate to the register width
        if ELUsingAArch32(EL3) then
            return SCR;
        else
            return SCR_EL3;
    else
        // No EL3 so return a default value
        SCRType defaultSCR = Zeroes();

        defaultSCR.NS = '1';

        return defaultSCR;

CTLRType CTLR_GEN()
    // Check if we have EL3 or not
    if (HaveEL(EL3)) then
        return ICC_CTLR_EL3;
    else
        return ICC_CTLR_EL1;
```

5.7.4 Hypervisor Configuration Register Access

The pseudocode below shows the code that returns the value of the hypervisor configuration register:

```
HCRType HCR_GEN()
    // Check if we have EL2 or not
    if (HaveEL(EL2)) then
        // We have EL2 so return the register appropriate to the register width
        if ELUsingAArch32(EL2) then
            return HCR;
        else
            return HCR_EL2;
    else
        // No EL2 so return a default value
        HCRType defaultHCR = Zeroes();

        return defaultHCR;
```

5.7.5 EOI Mode Determination

The pseudocode below shows how the EOI mode for the current interrupt regime is determined:

```
boolean EOImodeSet()
    // Check if we have EL3 or not
    if (HaveEL(EL3)) then
        // We have EL3 so return the value appropriate to the EL and security state
        if IsEL3OrMon() then
            // We are in EL3 and EL3 is AArch64 or EL3 is AArch32 and we are in Monitor mode
            return ICC_CTLR_EL3.EOImode_EL3;

        else if (PSTATE.EL == '3' || SCR_GEN().NS == '0')
            // We are secure
            return ICC_CTLR_EL3.EOImode_EL1S;

        else
            // We are non-secure
            return ICC_CTLR_EL3.EOImode_EL1NS;
```



```
else
    // No EL3 so return the value from ICC_CTLR_EL1
    return ICC_CTLR_EL1.EOImode;
```


5.7.6 System Register Enable Check

The following pseudocode checks whether access to a system register is permitted or not:

```
SystemRegisterAccessPermitted(integer group)
// The "group" parameter indicates which set of registers is being accessed
// as defined in section 4.2.5:
// 0   FIQ (Group 0) registers
// 1   IRQ (Group 1) registers
// 2   Common registers
// First check if any System Registers are enabled
    if (ICC_SRE_EL3.SRE == '0') then
        UndefinedException();              // System registers aren't enabled.

// Check that whether the access is to virtual or physical state
    boolean accessIsVirtual = false;

    if (group == 0 && PSTATE.EL == 1 && SCR_GEN().NS == '1' && HCR_GEN().FMO == '1') then
        accessIsVirtual = true;

    if (group == 1 && PSTATE.EL == 1 && SCR_GEN().NS == '1' && HCR_GEN().IMO == '1') then
        accessIsVirtual = true;

    if (group == 2 && PSTATE.EL == 1 && SCR_GEN().NS == '1' &&
        (HCR_GEN().FMO == '1' || HCR_GEN().IMO == '1')) then
        accessIsVirtual = true;

    boolean sreEL1S = HaveEL(EL3) && ICC_SRE_EL1S.SRE == '1';
    boolean sreEL2 = ICC_SRE_EL2.SRE == '1' || sreEL1S;
    boolean sreEL1NS = false;

// Check whether non-secure EL1 is using system registers or not
    if (HCR_GEN().FMO == '1' || HCR_GEN().IMO == '1' || HCR_GEN().AMO == '1') then
        // We have EL2 and at least one interrupt exception is virtualised
        sreEL1NS = (sreEL2  && ICC_SRE_EL1NS.SRE == '1') ||
                   (sreEL1S &&
                    (HCR_GEN().FMO == '0' || HCR_GEN().IMO == '0' || HCR_GEN().AMO == '0'));
    else if (HaveEL(EL2)) then
        sreEL1NS = (ICC_SRE_EL2.SRE == '1' && ICC_SRE_EL1NS.SRE == '1') || sreEL1S;
    else
        sreEL1NS = ICC_SRE_EL1NS.SRE == '1' || sreEL1S;

// Check if System Registers are enabled for the EL and security state
    if (PSTATE.EL == 2 && SCR_GEN().NS == '1' && !sreEL2) ||
       (PSTATE.EL == 1 && SCR_GEN().NS == '0' && ICC_SRE_EL1S.SRE == '0') ||
       (PSTATE.EL == 1 && SCR_GEN().NS == '1' && !sreEL1NS)
    then
        UndefinedException();              // System registers aren't enabled.

// Check if the access should trap to the hypervisor
    if (HaveEL(EL2)) then
        if (PSTATE.EL == 1 && SCR_GEN().NS == '1' && group == 0 && ICH_HCR_EL2.TALL0 == '1' then
            HypervisorTrap();

        if (PSTATE.EL == 1 && SCR_GEN().NS == '1' && group == 1 && ICH_HCR_EL2.TALL1 == '1' then
            HypervisorTrap();

        if (PSTATE.EL == 1 && SCR_GEN().NS == '1' && group == 2 && ICH_HCR_EL2.TC == '1' then
            HypervisorTrap();

// Check that access is allowed given the routing
    bits(2) lowestPhysicalEL == 3;

    if (group == 0 && SCR_GEN().FIQ == 0) then
        lowestPhysicalEL = 1;

    if (group == 1 && SCR_GEN().IRQ == 0) then
        lowestPhysicalEL = 1;

    if (group == 2 && (SCR_GEN().FIQ == 0 || SCR_GEN().IRQ == 0)) then
```



```
        lowestPhysicalEL = 1;

    if (!accessIsVirtual && PSTATE.EL < lowestPhysicalEL) then
        if (EL3IsAArch32()) then
            UndefinedException();
        else
            MonitorTrap();

    return;
```

5.7.7 Highest Priority Pending Interrupt

The pseudocode below shows how the highest priority pending interrupt is determined and whether the highest priority interrupt can preempt execution:

```
struct setType {
    bits(2)  State;
    intGroup Group;
    bits(8)  Priority;
    bits(NI) ID;
}
setType GICC_SETR; // Holds "Set" packets received from the Distributor

bits(NI) HighestPriorityPendingInterrupt()
    if ( ! GICC_SETR.State is "pending") then
        // No interrupt pending
        return 1023;

    if (GICC_SETR.Group == Group1NS) then
        if (ICC_IGRPEN1_EL1NS.Enable == '0') then return 1023;
    else if (GICC_SETR.Group == Group1S) then
        if (ICC_IGRPEN1_EL1S.Enable == '0') then return 1023;
    else if (GICC_SETR.Group == Group0) then
        if (ICC_IGRPEN0_EL1.Enable == '0') then return 1023;
    else // Reserved
        return 1023;

    return GICC_SETR.ID;


bool canSignalInterrupt()
    // Get the priority group of the current "Set" using the BPR appropriate to the group
    bits(7) setPriorityGroup = PriorityGroup(GICC_SETR.Priority, GICC_SETR.Group);

    bits(7) highestActive =  GetHighestActivePriority(ICC_AP0R_EL1, ICC_AP1R_EL1NS, ICC_AP1R_EL1S);
    intGroup highestGroup =  GetHighestActiveGroup(ICC_AP0R_EL1, ICC_AP1R_EL1NS, ICC_AP1R_EL1S);

    // Get the priority group of highest APR using the BPR appropriate to the APR group
    highestActive = PriorityGroup(highestActive << 1, highestGroup);

    if (GICC_SETR.State is "pending" && GICC_SETR.Priority < ICC_PMR_EL1) then
        // the "Set" is higher priority than PMR
        if (setPriorityGroup < highestActive) then
            // the “Set” can preempt
            return true;

    // Can’t preempt so no interrupt
    return false;
```


5.7.8 Highest Priority Virtual Interrupt

The pseudocode below shows how the highest priority pending list register interrupt is determined and whether the highest priority virtual interrupt can preempt execution:

```
integer HighestPriorityVirtualInterrupt()
    integer lrIndex = -1;
    bits(8) priority = 255;

// Find the List Register with the highest priority enabled pending interrupt
    for i = 0 to NUM_LIST_REGISTERS - 1
        if (listRegister[i].State is "pending") &&
            ( (listRegister[i].Group == 0 && ICH_VMCR_EL2.VENG0 == 1) ||
              (listRegister[i].Group == 1 && ICH_VMCR_EL2.VENG1 == 1) ) &&
            PriorityIsHigher( listRegister[i].Priority,  priority )
        then
            // Found an enabled pending list register with a higher priority
            priority = listRegister[i].Priority;
            lrIndex = i;

    return lrIndex;


boolean canSignalVirtualInt(lrType vInt)
    // First check whether the virtual interface is enabled
    if ( ICH_HCR_EL2.En == 0) then
        return false;

    // Get the priority group of "vInt" using the BPR appropriate to the group
    bits(7) setPriorityGroup = PriorityGroup(vInt.Priority, vInt.Group);

    bits(7) highestActive =  GetHighestActivePriority(ICH_AP0R_EL2, ICH_AP1R_EL2, Zeroes());
    intGroup highestGroup =  GetHighestActiveGroup(ICH_AP0R_EL2, ICH_AP1R_EL2, Zeroes);

    // Get the priority group of highest APR using the BPR appropriate to the APR group
    highestActive = PriorityGroup(highestActive << 1, highestGroup);

    if (vInt.State is "pending" && vInt.Priority < ICH_VMCR_EL2.VPMR) then
        // "vInt" is higher priority than PMR
        if (setPriorityGroup < highestActive) then
            // the "vInt" can preempt
            return true;

    // Can’t preempt so no interrupt
    return false;

boolean canSignalVirtualInterrupt()
    integer lrIndex = HighestPriorityVirtualInterrupt();

    if ( GICH_VLPIR.State == 0b01 &&
         ( lrIndex < 0 || PriorityIsHigher(GICH_VLPIR.Priority, listRegister[lrIndex].Priority) ) )
    then                                  // A virtual LPI is the highest priority
        return canSignalVirtualInt( GICH_VLPIR );


    else if ( lrIndex >= 0 ) then         // A list register is the highest priority
        return canSignalInt( listRegister[lrIndex] );

    return false;                         // There are no valid and enabled interrupts
```


5.7.9 Interrupt Identifier Valid Check

The following pseudocode checks whether an interrupt identifier is valid:

```
boolean InterruptIdentifierValid(integer data, boolean lpiAllowed)

// First check whether any out of range bits are set
    integer N;

    if (CTLR_GEN().IDbits == 0b000) then
        // 16 bit identifiers are supported
        N = 16;
    else
        // 24 bit identifiers are supported
        N = 24;

    if (data<31:N> != 0) then
        if (CTLR_GEN().SEIS) then
            // Reporting of locally generated SEIs is supported
            IMPLEMENTATION_DEFINED_SEI(data, INVALID_INTERRUPT_IDENTIFIER);

// Now check for LPI IDs

    if ( ! lpiAllowed) then
        // LPIs are not supported so check if the identifier is an LPI
        if (data<N-1:13> != 0) then
            if (CTLR_GEN().SEIS) then
                // Reporting of locally generated SEIs is supported
                IMPLEMENTATION_DEFINED_SEI(data, INVALID_INTERRUPT_IDENTIFIER);

        // Now check for special identifiers
        if (1020 <= data<9:0> <= 1023) then
            // it is a special ID
            return false;
    else
        // LPIs are supported so check if the identifier is a special identifier
        if (1020 <= data<N-1:0> <= 1023) then
            // it is a special ID
            return false;

// All the checks pass so the identifier is valid
    return true;
```

Note: this check also applies to memory mapped accesses to GICC_EOIR, GICC_AEOIR and GICC_DIR (see sections 5.6.6, 5.6.7 and 5.6.10 for the values of parameter `lpiAllowed`).

5.7.10 Virtual Interrupt Identifier Valid Check

The following pseudocode checks whether an interrupt identifier is valid:

```
boolean VirtualIdentifierValid(integer data, boolean lpiAllowed)

// First check whether any out of range bits are set
    integer N;

    if (ICH_VTR_EL2.IDbits == 0b000) then
        // 16 bit identifiers are supported
        N = 16;
    else
        // 24 bit identifiers are supported
        N = 24;

    if (data<31:N> != 0) then
        if (ICH_VTR_EL2.SEIS == '1') then
            // Reporting of locally generated virtual SEIs is supported
            IMPLEMENTATION_DEFINED_SEI(data, INVALID_INTERRUPT_IDENTIFIER);

// Now check for LPI IDs

    if ( ! lpiAllowed ) then
        // LPIs are not supported so check if the identifier is an LPI

        if (data<N-1:13> != 0) then
            if (ICH_VTR_EL2.SEIS == '1') then
                // Reporting of locally generated virtual SEIs is supported
                IMPLEMENTATION_DEFINED_SEI(data, INVALID_INTERRUPT_IDENTIFIER);
            return false;

        // Now check for special identifiers
        if (1020 <= data<9:0> <= 1023) then
            // it is a special ID
            return false;
    else
        // LPIs are supported so check if the identifier is a special identifier
        if (1020 <= data<N-1:0> <= 1023) then
            // it is a special ID
            return false;

// All the checks pass so the identifier is valid
    return true;
```

Note: this check also applies to memory mapped accesses to GICV_EOIR, GICV_AEOIR and GICV_DIR (see sections 5.10.2, 5.10.3 and 5.10.4 for the values of parameter `lpiAllowed`).

5.7.11 ICC_IAR0_EL1

This register is used instead of the memory mapped GICC_IAR register when SRE is set for an interrupt regime.

In systems with EL3, the intent is that the CPU receives a special purpose ID that indicates the target security domain if the interrupt is destined for EL1/EL2.

Note: to prevent spurious interrupts, when the FIQ interrupt exception to the processor is disabled any effects of reading this register on the signaling of interrupt exceptions to the processor must be observed when the instruction is architecturally executed.

Note: because direct reads of system registers may occur in any order, if software wishes to perform multiple accesses to this register without an intervening exception it must use an appropriate barrier to guarantee the ordering of these accesses.

```
bits(NI) CheckGroup0ForSpecialIdentifiers(bits(NI) pendID)
    if (!IsGrp0Int(pendID) && !IsEL3OrMon())
    then
        return 1023;                        // If the highest priority is group 1, then no interrupt

    if (IsSecureInt(pendID) && SCR_GEN().NS == '1' && !IsEL3OrMon()) then
        return 1023;                        // It is a secure interrupt the non-secure side can't see

    if pendID != 1023 && IsEL3OrMon() then // An enabled interrupt is pending
        if !IsGrp0Int(pendID) then
            if IsSecureInt(PendID) then    // A G1 interrupt for the other security state
                return 1020;               // Normally handled in secure EL1
            else
                return 1021;               // Normally handled in non-secure EL1/2
        else if ICC_CTLR_EL3.RM == '1' then
            return 1020;                   // A G0S interrupt destined for Secure EL1

    return pendID;

bits(32) ReadICC_IAR0_EL1()
// First check if System Registers are enabled
    SystemRegisterAccessPermitted(0);

// Check if the access is virtual
    if (PSTATE.EL == 1 && SCR_GEN().NS == '1' && HCR_GEN().FMO == '1') then
        return ReadICV_IAR0_EL1();        // Access the Virtual IAR0 register

// Now start handling the interrupt
    if ( ! canSignalInterrupt() ) then
        return 1023;

    // Gets the highest priority pending and enabled interrupt
    bits(NI) pendID = HighestPriorityPendingInterrupt();

    pendID = CheckGroup0ForSpecialIdentifiers(pendID);

    if pendID < 1020 || pendID > 1023 then // Check that it is not a special interrupt ID
        AcknowledgeInterrupt(pendID);      // Set active and attempt to clear pending

    bits(32) rval = Zeroes();
    rval<NI-1:0> = pendID;

    return rval;
```


5.7.12 Virtual Access to ICC_IAR0_EL1

The behaviour of virtual accesses to ICC_IAR0_EL1 is shown below:

```
bits(32) ReadICV_IAR0_EL1()
    int lrIndex = HighestPriorityVirtualInterrupt()

    if (! canSignalVirtualInterrupt()) then
        rval<NI-1:0> = 1023;
        return;

    if ( GICH_VLPIR.State == 0b01 &&
         ( lrIndex < 0 || PriorityIsHigher(GICH_VLPIR.Priority, listRegister[lrIndex].Priority) ) )
    then                                    // A virtual LPI is the highest priority
        rval<NI-1:0> = GICH_VLPIR.VirtualID;
        if ( GICH_VLPIR.Group == '0' ) then
            bits(7) vPriorityGroup = VPriorityGroup(GICH_VLPIR.Priority, 0);
            ICH_AP0R_EL2<vPriorityGroup> = '1';

            AcknowledgeVInterrupt(GICH_VLPIR.VirtualID);
            GICH_VLPIR.State = 0b00;        // Set the virtual LPI to Idle
        else
            rval<NI-1:0> = 1023;
        return rval ;

    // lrIndex must be valid (i.e. non-negative)
    lrType listReg = listRegister[lrIndex];

    bits(32) rval = Zeroes();
    bits(NI) vID = listReg.VirtualID;
    bits(NI) pID = listReg.PhysicalID;

    if (vID != 1023 && (listReg.Grp == '1' || listReg.State == 0b00)) then
        rval<NI-1:0>= 1023;                 // If the highest priority isn't group 0, then no interrupt
        return rval;

    rval<NI-1:0> = vID;

    if ( (vID < 1020 || vID > 1023) ) then // Check that it is not a spurious interrupt
        listReg.State = 0b10;               // Set the list register state to 'Active'
        bits(7) vPriorityGroup = VPriorityGroup(listReg.Priority, 0);
        ICH_AP0R_EL2<vPriorityGroup> = '1'; // Set the appropriate bit in the APR registers
    else                                    // It is a spurious ID, so
        listReg.State = 0b00;               // set the list register state to 'Not valid'

        boolean setEI = listReg.PhysicalID[9] == '1';

        // Generate a maintenance interrupt if required
        if (listReg.HW == '0' && setEI) then
            // Set the appropriate EISR bit to generate a maintenance interrupt
            ICH_EISR_EL2<lrIndex> = '1';
```



```
        if (listReg.HW == '1' || ! setEI ) then
            // Set the appropriate ELSR bit
            ICH_ELSR_EL2<lrIndex> = '1'
    return rval;
```

Note: the “GICH_VLPIR” register holds pending virtual LPIs (received as VSet packets; see section 7.3.5).

5.7.13 ICC_IAR1_EL1

This register is used instead of the memory-mapped GICC_IAR / GICC_AIAR register when SRE is set for an interrupt regime.

In systems with EL3, the intent is that the CPU will receive a spurious ID if the interrupt is a Secure Group 0 ID or if the interrupt is destined for the other security domain.

Note: to prevent spurious interrupts, when the IRQ interrupt exception to the processor is disabled any effects of reading this register on the signaling of interrupt exceptions to the processor must be observed when the instruction is architecturally executed.

Note: because direct reads of system registers may occur in any order, if software wishes to perform multiple accesses to this register without an intervening exception it must use an appropriate barrier to guarantee the ordering of these accesses.

Note: ICC_IAR1_EL1 may be read at EL3 if Secure EL1 software uses AArch32 and does not use system registers (see section 4.6.5).

```
bits(NI) CheckGroup1ForSpecialIdentifiers(bits(NI) pendID)
    if ( IsGrp0Int(pendID) && ! IsEL3OrMon() ) then
        return 1023;                        // If the highest priority is G0S
                                            // and not currently EL3 then spurious interrupt ID

    if pendID != 1023 then                  // An enabled interrupt is pending
        if (IsEL3OrMon() && ICC_CTLR_EL3.RM == '1') then
            // Read is at EL3 / Monitor and RM is set
            if !IsGrp0Int(pendID) then      // G1NS interrupts will be handled at Non-Secure EL1
                return 1021;
            else                            // We must be at EL3/Mon so indicate a G0 interrupt is pending
                return 1020;
        else if !IsGrp0Int(pendID)          // IRQ is routed to EL1/2 or RM is zero
            if IsSecureInt(pendID) then     // A secure G1 interrupt so check if cpu is secure
                if (SCR_GEN().NS == '1' && ! IsEL3OrMon()) then
                    return 1023;            // Not secure, so interrupt not visible
            else
                if SCR_GEN().NS == '0' && !IsEL3OrMon()) then
                    return 1023;            // It is a non-secure interrupt that secure EL1 can't see
        else
            return 1023;                    // It is a group 0 interrupt

    return pendID;

bits(32) ReadICC_IAR1_EL1()
// First check if System Registers are enabled
    SystemRegisterAccessPermitted(1);

// Check if the access is virtual
    if PSTATE.EL == 1 && SCR_GEN().NS == '1' && HCR_GEN().IMO == '1'
    then
        return ReadICV_IAR1_EL1();        // Access the Virtual IAR1 register

// Now start handling the interrupt
    if ( ! canSignalInterrupt() ) then
        return 1023;

    // Gets the highest priority pending and enabled interrupt
    bits(NI) pendID = HighestPriorityPendingInterrupt();

    pendID = CheckGroup1ForSpecialIdentifiers(pendID);
```



```
    if pendID < 1020 || pendID > 1023 then // Check that it is not a special interrupt ID
        AcknowledgeInterrupt(pendID);      // Set active and attempt to clear pending

    bits(32) rval = Zeroes();
    rval<NI-1:0> = pendID;

    return rval;
```

Because ICC_IAR1_EL1 will be mapped to a system register, if a higher priority group 0 interrupt becomes pending while a read of this register is in progress, the system might take the FIQ exception to EL3 and the spurious ID might never be visible for this case.

5.7.14 Virtual Access to ICC_IAR1_EL1

The behaviour of virtual accesses to ICC_IAR0_EL1 is shown below:

```
bits(32) ReadICV_IAR1_EL1()
    int lrIndex = HighestPriorityVirtualInterrupt();

    if (! canSignalVirtualInterrupt()) then
        rval<NI-1:0> = 1023;
        return;

    if ( GICH_VLPIR.State = 0b01 &&
         ( lrIndex < 0 || PriorityIsHigher(GICH_VLPIR.Priority, listRegister[lrIndex].Priority) ) )
    then                                    // A virtual LPI is the highest priority
        rval<NI-1:0> = GICH_VLPIR.VirtualID;
        if ( GICH_VLPIR.Group == '1' ) then
            bits(7) vPriorityGroup = VPriorityGroup(GICH_VLPIR.Priority, 1);
            ICH_AP1R_EL2<vPriorityGroup> = '1';

            AcknowledgeVInterrupt(GICH_VLPIR.VirtualID);
            GICH_VLPIR.State = 0b00;        // Set the virtual LPI to Idle
        else
            rval<NI-1:0> = 1023;
        return rval;

    // lrIndex must be valid (i.e. non-negative)
    lrType listReg = listRegister[lrIndex];

    bits(32) rval = Zeroes();
    bits(NI) vID = listReg.VirtualID;
    bits(NI) pID = listReg.PhysicalID;

    if ( listReg.Grp == '0' || listReg.State == '00' ) then
        rval<NI-1:0> = 1023;                // If the highest priority isn't group 1, then no interrupt
        return rval;

    rval<NI-1:0> = vID;

    if ( (vID < 1020 || vID > 1023) ) then // Check that it is not a spurious interrupt
        listReg.State = 0b10;               // Set the list register state to 'Active'
        bits(7) vPriorityGroup = VPriorityGroup(listReg.Priority, 1);
        ICH_AP1R_EL2<vPriorityGroup> = '1'; // Set the appropriate bit in the APR registers
    else                                    // It is a spurious ID, so
        listReg.State = 0b00;               // set the list register state to 'Not valid'

        boolean setEI = listReg.PhysicalID[9] == '1';

        // Generate a maintenance interrupt if required
        if (listReg.HW == '0' && setEI) then
            // Set the appropriate EISR bit to generate a maintenance interrupt
            ICH_EISR_EL2<lrIndex> = '1';

        if (listReg.HW == '1' || ! setEI) then
            // Set the appropriate ELSR bit
            ICH_ELSR_EL2<lrIndex> = '1'
```



```
    return rval;
```

Note: the “GICH_VLPIR” register holds pending virtual LPIs (received as VSet packets; see section 7.3.5).

5.7.15 ICC_HPPIR0_EL1

This register is used instead of the memory mapped GICC_HPPIR register when SRE is set for an interrupt regime.

The lowest exception level at which this register may be accessed is governed by the exception level to which FIQ is routed, as defined in Section 4.4.1.

```
bits(32) ReadICC_HPPIR0_EL1()
// First check if System Registers are enabled
    SystemRegisterAccessPermitted(0);

// Check if the access is virtual
    if PSTATE.EL == 1 && SCR_GEN().NS == '1' && HCR_GEN().FMO == '1' then
        return ReadICV_HPPIR0_EL1();        // Access the Virtual HPPRR0 register

// Now start handling the interrupt
    bits(NI) pendID = HighestPriorityPendingInterrupt();
    pendID = CheckGroup0ForSpecialIdentifiers(pendID);

    bits(32) rval = Zeroes();
    rval<NI-1:0> = pendID;

    return rval;
```

5.7.16 Virtual Access to ICC_HPPIR0_EL1

The behaviour of virtual accesses to ICC_HPPIR0_EL1 is shown below:

```
bits(32) ReadICV_HPPIR0_EL1()
    int lrIndex = HighestPriorityVirtualInterrupt();

    if ( GICH_VLPIR.State == 0b01 &&
         ( lrIndex < 0 || PriorityIsHigher(GICH_VLPIR.Priority, listRegister[lrIndex].Priority) ) )
    then                                    // A virtual LPI is the highest priority
        rval<NI-1:0> = GICH_VLPIR.VirtualID;
        if ( GICH_VLPIR.Group == '1' ) then
            rval<NI-1:0> = 1023;
        return rval ;

    // lrIndex must be valid (i.e. non-negative)
    lrType listReg = listRegister[lrIndex];

    bits(32) rval = Zeroes();
    bits(NI) vID = listReg.VirtualID;

    if (vID != 1023 && (listReg.Grp == '1' || listReg.State == '00')) then
        vID = 1023;                         // If the highest priority isn't group 0, then no interrupt

    rval<NI-1:0> = vID;

    return rval;
```


5.7.17 ICC_HPPIR1_EL1

This register is used instead of the memory mapped GICC_HPPIR / GICC_AHPPIR register when SRE is set for an interrupt regime.

The lowest exception level at which this register may be accessed is governed by the exception level to which IRQ is routed, as defined in Section 4.4.1.

```
bits(32) ReadICC_HPPIR1_EL1()
// First check if System Registers are enabled
    SystemRegisterAccessPermitted(1);

// Check if the access is virtual
    if PSTATE.EL == 1 && SCR_GEN().NS == '1' && HCR_GEN().IMO == '1'
    then
        return ReadICV_HPPIR1_EL1();        // Access the Virtual HPPIR1 register

// Now start handling the interrupt
    bits(NI) pendID = HighestPriorityPendingInterrupt();
    pendID = CheckGroup1ForSpecialIdentifiers(pendID);

    bits(32) rval = Zeroes();
    rval<NI-1:0> = pendID;

    return rval;
```

5.7.18 Virtual Access to ICC_HPPIR1_EL1

The behaviour of virtual accesses to ICC_HPPIR1_EL1 is shown below:

```
bits(32) ReadICV_HPPIR1_EL1()
    int lrIndex = HighestPriorityVirtualInterrupt();

    if ( GICH_VLPIR.State == 0b01 &&
         ( lrIndex < 0 || PriorityIsHigher(GICH_VLPIR.Priority, listRegister[lrIndex].Priority) ) )
    then                                    // A virtual LPI is the highest priority
        rval<NI-1:0> = GICH_VLPIR.VirtualID;
        if ( GICH_VLPIR.Group == '0' ) then
            rval<NI-1:0> = 1023;
        return rval ;

    // lrIndex must be valid (i.e. non-negative)
    lrType listReg = listRegister[lrIndex];

    bits(32) rval = Zeroes();
    bits(NI) vID = listReg.VirtualID;

    if (vID != 1023 && (listReg.Grp == '0' || listReg.State == '00')) then
        vID = 1023;                         // If the highest priority isn't group 1, then no interrupt

    rval<NI-1:0> = vID;

    return rval;
```


5.7.19 ICC_EOIR0_EL1

This register is used instead of secure accesses to the memory mapped GICC_EOIR register when SRE is set for an interrupt regime.

Software must ensure the interrupt identifier written to ICC_EOIR0_EL1 is identical to the identifier returned by the last read of an interrupt acknowledge register and that this identifier was read from ICC_IAR0_EL1 while operating in the same security state as that in which the write occurs, otherwise the system behaviour is UNPREDICTABLE.

The lowest exception level at which this register may be accessed is governed by the exception level to which FIQ is routed, as defined in Section 4.4.1.

```
WriteICC_EOIR0_EL1(integer data)
    bits(NI) eoiID = data<NI-1:0>;
// First check if System Registers are enabled
    SystemRegisterAccessPermitted(0);

// Check if the access is virtual
    if PSTATE.EL == 1 && SCR_GEN().NS == '1' && HCR_GEN().FMO == '1' then
        WriteICV_EOIR0_EL1(data);        // Access the Virtual EOIR0 register
        return;

// Check for spurious ID.  LPIs are allowed and the access is physical
    if (! InterruptIdentifierValid(data, true)) then
        return;

// Now start handling the interrupt
    // Is the highest priority G0S, G1S or G1NS
    intGroup pGroup = GetHighestActiveGroup(ICC_AP0R_EL1, ICC_AP1R_EL1NS, ICC_AP1R_EL1S);
    bits(7)  pPriority = GetHighestActivePriority(ICC_AP0R_EL1, ICC_AP1R_EL1NS, ICC_AP1R_EL1S);

    if (pGroup == None) then
        // There are no active interrupts
        if (CTLR_GEN().SEIS) then
            // Reporting of locally generated SEIs is supported
            IMPLEMENTATION_DEFINED_SEI(data, EOI0_NO_INTS_ACTIVE);
    else if (pGroup == Group0 && (SCR_GEN().NS == '0' || !IsSecureInt(data) || IsEL3OrMon())) then
        // Highest priority is Group 0
        // Drop the priority
        PriorityDrop(pPriority, ICC_AP0R_EL1);

        if (EOImodeSet()) then
           // EOI mode is set, so don't deactivate
        else
            // Deactivate the interrupt unless it is an LPI
            if (eoiID < 8192) then
                Deactivate(eoiID);
    else if (CTLR_GEN().SEIS) then // Locally generated SEIs are supported
        // Highest priority is Group 1
        IMPLEMENTATION_DEFINED_SEI(data, EOI0_HIGHEST_IS_G1);
```


5.7.20 Virtual Access to ICC_EOIR0_EL1

There are three cases when writing to ICC_EOIR0_EL1 that were unpredictable for a corresponding GICv2 write to GICV_EOIR:

- If the identified interrupt is in the List Registers and is Group 1. When EL2 is using system registers and ICH_VTR_EL2.SEIS is one, an implementation defined SEI might be generated but a virtual “priority drop” must still be performed.
- If the identified interrupt is in the List Registers and the HW bit is one and the “Physical ID” field indicates the interrupt to be deactivated is an SGI (i.e. the value of “Physical ID” is between 0 and 15). GICv3 implementations must perform the EOI operation (that is, the CPU interface must send an appropriate Deactivate packet). ***Note***: this means that GICv3 Distributor implementations must always ensure only a single GICv2 SGI is active for a processor such that no knowledge of the Source CPU ID is required.
- If the identified interrupt is in the List Registers and the HW bit is one and the “Physical ID” field is between 1020 and 1023, indicating a special purpose ID. GICv3 implementations must not perform a DIR operation but must still change the state of the List Register as appropriate. When EL2 is using system registers and ICH_VTR_EL2. SEIS is one, an implementation defined SEI might be generated.

The behaviour of virtual accesses to ICC_EOIR0_EL1 is shown below:

```
WriteICV_EOIR0_EL1(integer data)
    bits(NI) eoiID = data<NI-1:0>;
    bits(7)  vPriority = GetHighestActivePriority(ICH_AP0R_EL2, ICH_AP1R_EL2, Zeroes());

// Check the identifier is valid.  LPIs are allowed and the access is virtual
    if (! VirtualIdentifierValid(data, true)) then
        return;

// Now perform the priority drop
    boolean dropped = PriorityDrop2(vPriority, ICH_AP0R_EL2, ICH_AP1R_EL2);

    if ( ! dropped) then
        if (ICH_VTR_EL2.SEIS == '1') then
            // Reporting of locally generated virtual SEIs is supported
            IMPLEMENTATION_DEFINED_SEI(data, EOI0_HIGHEST_IS_G1);
        return;

    boolean lrFound = false;
    integer lrIndex = 0;

// Now find the matching List Register
    for i = 0 to NUM_LIST_REGISTERS - 1
        if (!lrFound && (listRegister[i].State & 0b10 != 0b00) &&
            listRegister[i].VirtualID == eoiID)
        then
            // Found an active or active+pending list register for the EOI ID
            lrFound = true;
            lrIndex = i;

    if (!lrFound && eoiID >= 8192) then
        // It is a virtual LPI not in the List Registers
        // so return without incrementing EOI count
        return;
    else if !lrFound then
        // No valid list register corresponds to the EOI ID, increment EOI count
        if (dropped && ICH_VMCR_EL2.VEOIM == '0') then
            ICH_HCR_EL2.EOIcount++;
        return;
```



```
// Start error checks
// When an error is detected return to avoid unpredictable behaviour
    if (listRegister[lrIndex].Group != '0') then
        // The EOI ID is not Group 0
        if (ICH_VTR_EL2.SEIS == '1') then
            // Reporting of locally generated virtual SEIs is supported
            IMPLEMENTATION_DEFINED_SEI(data, EOI0_HIGHEST_IS_G1);
        return;

    if (VPriorityGroup(listRegister[lrIndex].Priority, 0) != vPriority) then
        // The EOI ID is not the highest priority
        if (ICH_VTR_EL2.SEIS == '1') then
            // Reporting of locally generated virtual SEIs is supported
            IMPLEMENTATION_DEFINED_SEI(data, EOI0_NOT_HIGHEST_PRIORITY);
        return;
// End of error checks

    if (ICH_VMCR_EL2.VEOIM == '0' || eoiID >=8192) then
        boolean setEI = listRegister[lrIndex].PhysicalID[9] == '1';

        // EOI mode not set, or it is an LPI and no deactivate is expected
        // so clear the active state in the List Register
        if (listRegister[lrIndex].HW =='1') then
            // Deactivate the physical interrupt
            bits(10) pID = listRegister[lrIndex].PhysicalID;
            if (pID < 1020) then
                Deactivate(pID)
        else
            // Generate a maintenance interrupt if required
            if (setEI) then
                // Set the appropriate EISR bit to generate a maintenance interrupt
                ICH_EISR_EL2<lrIndex> = '1';
        // Clear the active state
        listRegister[lrIndex].State = listRegister[lrIndex].State & 0b01;

        if (listRegister[lrIndex].State == 0) then
            if (listRegister[lrIndex].HW == '1' || ! setEI) then
                // Set the appropriate ELSR bit
                ICH_ELSR_EL2<lrIndex> = '1';
```


5.7.21 ICC_EOIR1_EL1

This register is used instead of the memory mapped GICC_AEOIR and non-secure GICC_EOIR register when SRE is set for an interrupt regime.

Software must ensure the interrupt identifier written to ICC_EOIR1_EL1 is identical to the identifier returned by the last read of an interrupt acknowledge register and that this identifier was read from ICC_IAR1_EL1 while operating in the same security state as that in which the write occurs, otherwise the system behaviour is UNPREDICTABLE.

The lowest exception level at which this register may be accessed is governed by the exception level to which IRQ is routed, as defined in Section 4.4.1.

```
WriteICC_EOIR1_EL1(integer data)
    bits(NI) eoiID = data<NI-1:0>;
// First check if System Registers are enabled
    SystemRegisterAccessPermitted(1);

// Check if the access is virtual
    if PSTATE.EL == 1 && SCR_GEN().NS == '1' && HCR_GEN().IMO == '1' then
        WriteICV_EOIR1_EL1(data);  // Access the Virtual EOIR1 register

// Check for spurious ID.  LPIs are allowed and the access is physical
    if (! InterruptIdentifierValid(data, true)) then
        return;

// Now start handling the interrupt
    // Is the highest priority G0S, G1S or G1NS
    intGroup pGroup = GetHighestActiveGroup(ICC_AP0R_EL1, ICC_AP1R_EL1NS, ICC_AP1R_EL1S);
    bits(7)  pPriority = GetHighestActivePriority(ICC_AP0R_EL1, ICC_AP1R_EL1NS, ICC_AP1R_EL1S);

    if (pGroup == None) then
        // There are no active interrupts
        if (CTLR_GEN().SEIS) then
            // Reporting of locally generated SEIs is supported
            IMPLEMENTATION_DEFINED_SEI(data, EOI1_NO_INTS_ACTIVE);
    else if (pGroup == Group1NS && (SCR_GEN().NS == '1' || IsEL3OrMon())) then
        // Highest priority is Non-Secure Group 1
        // Drop the priority
        PriorityDrop(pPriority, ICC_AP1R_EL1NS);

        if (EOImodeSet()) then
           // EOI mode is set, so don't deactivate
        else
            // Deactivate the interrupt unless it is an LPI
            if (eoiID < 8192) then
                Deactivate(eoiID);
    else if (pGroup == Group1S && (SCR_GEN().NS == '0' || IsEL3OrMon())) then
        // Highest priority is Secure Group 1 and we are secure
        // Drop the priority
        boolean dropped = PriorityDrop(pPriority, ICC_AP1R_EL1S);

        if (EOImodeSet()) then
           // EOI mode is set, so don't deactivate
        else
            // Deactivate the interrupt unless it is an LPI
            if (eoiID < 8192) then
                Deactivate(eoiID);
    else if (CTLR_GEN().SEIS) then // Locally generated SEIs are supported
        // Highest priority is Group 0 or Secure Group 1 and we are not secure
        IMPLEMENTATION_DEFINED_SEI(data, EOI1_HIGHEST_NOT_ACCESSIBLE);
```


5.7.22 Virtual Access to ICC_EOIR1_EL1

There are three cases when writing to ICC_EOIR1_EL1 that were unpredictable for a corresponding GICv2 write to GICV_AEOIR:

- If the identified interrupt is in the List Registers and is Group 0. When EL2 is using system registers and ICH_VTR_EL2.SEIS is one, an implementation defined SEI might be generated but a virtual “priority drop” must still be performed.
- When EOI mode is zero, if the identified interrupt is in the List Registers and the HW bit is one and the “Physical ID” field indicates the interrupt to be deactivated is an SGI (i.e. the value of “Physical ID” is between 0 and 15). GICv3 implementations must perform the EOI operation (that is, the CPU interface must send an appropriate Deactivate packet). ***Note***: this means that GICv3 Distributor implementations must always ensure only a single GICv2 SGI is active for a processor such that no knowledge of the Source CPU ID is required.
- When EOI mode is zero, if the identified interrupt is in the List Registers and the HW bit is one and the “Physical ID” field is between 1020 and 1023, indicating a special purpose ID. GICv3 implementations must not perform a DIR operation but must still change the state of the List Register as appropriate. When EL2 is using system registers and ICH_VTR_EL2.SEIS is one, an implementation defined SEI might be generated.

The behaviour of virtual accesses to ICC_EOIR1_EL1 is shown below:

```
WriteICV_EOIR1_EL1(integer data)
    bits(NI) eoiID = data<NI-1:0>;
    bits(7)  vPriority = GetHighestActivePriority(ICH_AP0R_EL2, ICH_AP1R_EL2, Zeroes());

// Check for spurious ID.  LPIs are allowed and the access is virtual
    if (! VirtualIdentifierValid(data, true)) then
        return;

// Now perform the priority drop
    boolean dropped = PriorityDrop2(vPriority, ICH_AP0R_EL2, ICH_AP1R_EL2);

    if ( ! dropped) then
        if (ICH_VTR_EL2.SEIS == '1') then
            // Reporting of locally generated virtual SEIs is supported
            IMPLEMENTATION_DEFINED_SEI(data, EOI1_HIGHEST_IS_G0);
        return;

    lrFound = false;
    lrIndex = 0;

// Now find the matching List Register
    for i = 0 to NUM_LIST_REGISTERS - 1
        if (!lrFound && (listRegister[i].State & 0b10 != 0b00) &&
            listRegister[i].VirtualID == eoiID)
        then
            // Found an active or active+pending list register for the EOI ID
            lrFound = true;
            lrIndex = i;

    if (!lrFound && eoiID >= 8192) then
        // It is a virtual LPI not in the List Registers
        // so just priority drop and return without incrementing EOI count
        return;
    else if !lrFound then
        // No valid list register corresponds to the EOI ID, increment EOI count
        if (dropped && ICH_VMCR_EL2.VEOIM == '0') then
            ICH_HCR_EL2.EOIcount ++;
```



```
        return;

// Start error checks
// When an error is detected return to avoid unpredictable behaviour
    if (listRegister[lrIndex].Group != '1') then
        // The EOI ID is not Group 1
        if (ICH_VTR_EL2.SEIS == '1') then
            // Reporting of locally generated virtual SEIs is supported
            IMPLEMENTATION_DEFINED_SEI(data, EOI1_HIGHEST_IS_G0);
        return;

    if (VPriorityGroup(listRegister[lrIndex].Priority, 1) != vPriority) then
        // The EOI ID is not the highest priority
        if (ICH_VTR_EL2.SEIS == '1') then
            // Reporting of locally generated virtual SEIs is supported
            IMPLEMENTATION_DEFINED_SEI(data, EOI1_NOT_HIGHEST_PRIORITY);
        return;
// End of error checks

    if (ICH_VMCR_EL2.VEOIM == '0' || eoiID >=8192) then
        boolean setEI = listRegister[lrIndex].PhysicalID[9] == '1';

        // EOI mode not set, or it is an LPI and no deactivate is expected
        // so clear the active state in the List Register
        if (listRegister[lrIndex].HW =='1') then
            // Deactivate the physical interrupt
            bits(10) pID = listRegister[lrIndex].PhysicalID;
            if (pID < 1020) then
                Deactivate(pID)
        else
            // Generate a maintenance interrupt if required
            if (setEI) then
                // Set the appropriate EISR bit to generate a maintenance interrupt
                ICH_EISR_EL2<lrIndex> = '1';
        // Clear the active state
        listRegister[lrIndex].State = listRegister[lrIndex].State & 0b01;

        if (listRegister[lrIndex].State == 0) then
            if (listRegister[lrIndex].HW == '1' || ! setEI) then
                // Set the appropriate ELSR bit
                ICH_ELSR_EL2<lrIndex> = '1';
```


5.7.23 ICC_DIR_EL1

This register is used instead of the memory mapped GICC_DIR register when SRE is set for an interrupt regime.

The lowest exception level at which this register may be accessed is defined in Section 4.6.7.

Note: accesses to this register trap to EL2 (HYP_TRAP) if ICH_HCR_EL2.TC == '1';

There are three cases when writing to ICC_DIR_EL1 that were unpredictable for a corresponding GICv2 write to GICC_DIR:

- When EOImode == '0'. GICv3 implementations must ignore such writes. In systems supporting system error generation, an implementation might generate an SEI.
- When EOImode == '1' but the interrupt is not "active" in the Distributor. GICv3 Distributor implementations must ignore such writes. In systems supporting system error generation, an implementation might generate a system error. ***Note***: in implementations using the Distributor Interface defined in 7, these writes will correspond to a Deactivate packet (see section 7.4.5) for an interrupt that is not "active".
- When EOImode == '1' but no EOI has been issued. The interrupt will be de-activated by the Distributor however the active priority in the CPU interface for the interrupt will remain set (because no EOI was issued).

The pseudocode for writes to ICC_DIR_EL1 is shown below:

```
WriteICC_DIR_EL1(integer data)
// First check if System Registers are enabled
    SystemRegisterAccessPermitted(2);

// Check if the access is virtual
    if PSTATE.EL == 1 && SCR_GEN().NS == '1' && (HCR_GEN().FMO == '1' || HCR_GEN().IMO == '1') then
        WriteICV_DIR_EL1(data);  // Access the Virtual DIR register
        return;

// Check for spurious ID.  LPIs are not allowed and the access is physical
    if (! InterruptIdentifierValid(data, false)) then
        return;

// Now start handling the interrupt
    if (!EOImodeSet()) then
        // EOI mode is not set, so don't deactivate
        IMPLEMENTATION_DEFINED_SEI(data, DIR_EOIMODE_NOT_SET);
    else
        // Deactivate the interrupt
        Deactivate(data<NI-1:0>);
```

Note: in implementations that support the GIC Stream Protocol Interface specified in section 7, if the current interrupt regime permits deactivation of an interrupt group then the corresponding "Groups" bit is set in the deactivate packet sent to distributor. See section 7.4.5.

5.7.24 Virtual Access to ICC_DIR_EL1

There are four cases when writing to ICC_DIR_EL1 that were unpredictable for a corresponding GICv2 write to GICV_DIR:

- When EOImode == '0'. GICv3 implementations must ignore such writes. In systems supporting local generation of system errors, an implementation might generate an SEI.
- When EOImode == '1' but the interrupt is present in the List Registers but is not "active". GICv3 implementations must ignore such writes.
- Writes with a special ID (i.e. between 1020 and 1023 inclusive). GICv3 implementations must ignore such writes.
- If the identified interrupt is in the List Registers and ICH_LRn_EL2.HW == '1' and ICH_LRn_EL2.PhysicalID indicates the interrupt to be deactivated is an SGI (i.e. the value of PhysicalID is between 0 and 15). GICv3 implementations must perform the DIR operation. ***Note***: this means that GICv3 Distributor implementations must always ensure only a single GICv2 SGI is active for a processor such that no knowledge of the Source CPU ID is required.

The pseudocode for virtual accesses to ICC_DIR_EL1 is shown below:

```
WriteICV_DIR_EL1(integer data)
    bits(NI) eoiID = data<NI-1:0>;

    lrFound = false;
    lrIndex = 0;

// Start error checks
// When an error is detected return to avoid unpredictable behaviour
    if (ICH_VMCR_EL2.VEOIM == '0') then
        // EOI mode is not set so generate a maintenance interrupt
        if (ICH_VTR_EL2.SEIS == '1') then
            // Reporting of locally generated virtual SEIs is supported
            IMPLEMENTATION_DEFINED_SEI(data, DIR_EOIMODE_NOT_SET);
        return;
// End of error checks

// Check for spurious ID.  LPIs are not allowed and the access is virtual
    if (! VirtualIdentifierValid(data, false)) then
        return;

    for i = 0 to NUM_LIST_REGISTERS - 1
        if (!lrFound && (listRegister[i].State & 0b10 != 0b00) &&
            listRegister[i].VirtualID == eoiID)
        then
            // Found an active or active+pending list register for the EOI ID
            lrFound = true;
            lrIndex = i;

    if !lrFound then
        // No valid list register corresponds to the EOI ID, increment EOI count
        ICH_HCR_EL2.EOIcount ++;
        return;

    boolean setEI = listRegister[lrIndex].PhysicalID[9] == '1';

    if (listRegister[lrIndex].HW =='1') then
        // Deactivate the physical interrupt if EOI Mode is set
        bits(10) pID = listRegister[lrIndex].PhysicalID;
```



```
    if (pID < 1020) then
        Deactivate(pID)
else
    if (setEI) then
        // Set the appropriate EISR bit to generate a maintenance interrupt
        ICH_EISR_EL2<lrIndex> = '1';
listRegister[lrIndex].State = 0b00;

if (listRegister[lrIndex].HW == '1' || ! setEI) then
    // Set the appropriate ELSR bit
    ICH_ELSR_EL2<lrIndex> = '1';

return;
```


5.7.25 ICC_BPR1_EL1

This register is used instead of the memory mapped GICC_BPR (non-secure) / GICC_ABPR register when SRE is set for an interrupt regime. This register is banked by security and behaves as defined below:

- If ICC_CTLR_EL3.CBPR_EL1S is one:
  - Writes at secure EL1 will modify ICC_BPR0_EL1.
  - Writes at EL3 when EL3 is AArch32 and is not in Monitor mode will modify ICC_BPR0_EL1.
  - Reads at secure EL1 will return the value of ICC_BPR0_EL1.
  - Reads at EL3 when EL3 is AArch32 and is not in Monitor mode will return the value of ICC_BPR0_EL1.
- If ICC_CTLR_EL3.CBPR_EL1NS is one, non-secure accesses at EL1 or EL2 behave as defined in the table below.

SCR_EL3	HCR_EL2	
IRQ	IMO	Description
1	0	Inaccessible to non-secure EL1 / EL2
1	1	Non-secure EL1 access is virtual. Inaccessible to EL2.
0	1	Non-secure EL1 access is virtual. Non-secure EL2 reads return (ICC_BPR0_EL1 + 1) saturated to 0b111 Non-secure EL2 writes are ignored
0	0	Non-secure EL1 / EL2 reads return (ICC_BPR0_EL1 + 1) saturated to 0b111 Non-secure EL1 / EL2 writes are ignored

Table 27: Non-secure access to ICC_BPR1_EL1 when CBPR_EL1NS is one

The non-secure copy of this register resets to (the minimum value of ICC_BPR0_EL1 + 1) as defined for GICv2. ***Note***: if software writes to the binary point register with a value less than the reset value, the register is set to the reset value.

The secure copy of this register resets to (the minimum value of ICC_BPR0_EL1). ***Note***: if software writes to the binary point register with a value less than the reset value, the register is set to the reset value.

Note: when operating at EL3 and either EL3 is AArch64 or, EL3 is AArch32 and is in Monitor mode, the copy appropriate to the current setting of SCR_EL3.NS is accessed (see section 4.4.10) and ICC_CTLR_EL3.CBPR_EL1{S,NS} are ignored.

The lowest exception level at which this register may be accessed is governed by the exception level to which IRQ is routed, as defined in section 4.6.6.

Virtual accesses to this register update ICH_VMCR_EL2.VBPR1. ***Note***: virtual accesses to ICC_BPR1_EL1 are affected by ICH_VMCR_EL2.VCBPR. See section 5.9.5 for details.

5.7.26 ICC_BPR0_EL1

This register is used instead of the memory mapped GICC_BPR register when SRE is set for an interrupt regime.

The lowest exception level at which this register may be accessed is governed by the exception level to which FIQ is routed, as defined in section 4.6.6.

Virtual accesses to this register update ICH_VMCR_EL2.VBPR0.

5.7.27 ICC_PMR_EL1

This register is used instead of the memory mapped GICC_PMR register when SRE is set for an interrupt regime.

In order to allow software to use these registers to manage interrupt load balancing, writes to this register must be high performance and must ensure that no interrupt of lower priority than the written value occurs after the write, without requiring an ISB or an exception boundary. Hence, this system register is mapped into CP15, CRn = 4 space as defined in Section 3.10.3 of [4].

Virtual accesses to this register update ICH_VMCR_EL2.VPMR.

Note: accesses to this register trap to EL2 (HYP_TRAP) if ICH_HCR_EL2.TC == '1';

Note: an access is "virtual" when accessed at non-secure EL1 and either of FIQ and IRQ have been virtualized. That is, when (SCR_EL3.NS == '1' && (HCR_EL2.FMO == '1' || HCR_EL2.IMO == '1')).

The pseudocode for read access to ICC_PMR_EL1 is shown below:

```
bits(32) ReadICC_PMR_EL1()
// First check if System Registers are enabled
    SystemRegisterAccessPermitted(2); // Set group to 2 so "TC" bit is checked

    if (PSTATE.EL == 1 && SCR_GEN().NS == '1' &&
        (HCR_GEN().FMO == '1' || HCR_GEN().IMO == '1'))
    then
        // At least one interrupt is virtualised so return the virtual mask
        return ICH_VMCR_EL2.VPMR AND VP_MASK;

    bits(8) pPriority = ICC_PMR_EL1;

    if (!IsEL3OrMon() && SCR_GEN().NS == '1' && SCR_GEN().FIQ == '1') then
        // A non-secure GIC access and group 0 inaccessible to non-secure.
        if pPriority<7> == '0' then
            // Priority is in secure half and not visible to non-secure
            pPriority<7:0> = 0;
        else if pPriority != 255
            // Non secure access and not idle, so physical priority must be shifted
            pPriority<7:0> = LSL((pPriority AND P_MASK), 1);

    bits(32) rval = Zeroes();
    rval<7:0> = pPriority;

    return rval;
```


The pseudocode for write access to ICC_PMR_EL1 is shown below:

```
WriteICC_PMR_EL1(integer data)
// First check if System Registers are enabled
    SystemRegisterAccessPermitted(2); // Set group to 2 so "TC" bit is checked

    if (PSTATE.EL == 1 && SCR_GEN().NS == '1' &&
        (HCR_GEN().FMO == '1' || HCR_GEN().IMO == '1'))
    then
        // At least one interrupt is virtualised so update the virtual mask
        ICH_VMCR_EL2.VPMR = data<7:0> AND VP_MASK;
        return;

    if (!IsEL3OrMon() && SCR_GEN().NS == '1' &&  SCR_GEN().FIQ == '1') then
        // A non-secure GIC access and group 0 inaccessible to non-secure.
        mod_write_val = ('10000000' OR LSR(data<7:0>,1)) AND P_MASK;
        if ICC_PMR_EL1<7> == '1' then                   // Non-secure execution can only update the
            ICC_PMR_EL1<7:0> = mod_write_val;           // Priority Mask Register if the current
                                                        // value is in the range 0x80 to 0xFF
        else
            // PMR is between 0x00 and 0x7F and may not be written
            IgnoreWrite();
    else                                                // A secure GIC access
        ICC_PMR_EL1<7:0> = value AND P_MASK;
```


5.7.28 ICC_RPR_EL1

This register returns the priority of the current active interrupt. The priority returned is the group priority as if the BPR for the current interrupt regime was set to the minimum value of BPR for the number of implemented priority bits. ***Note***: if 8 bits of priority are implemented the group priority is bits[7:1] of the priority.

Note: virtual accesses to this register trap to EL2 (HYP_TRAP) if ICH_HCR_EL2.TC == '1'.

```
bits(32) ReadICC_RPR_EL1()
// First check if System Registers are enabled
    SystemRegisterAccessPermitted(2); // Set group to 2 so "TC" bit is checked

    bits(32) rval = Zeroes();

    if (PSTATE.EL == 1 && SCR_GEN().NS == '1' &&
        (HCR_GEN().FMO == '1' || HCR_GEN().IMO == '1'))
    then
        // At least one interrupt is virtualised so return the virtual priority
        bits(7) vPriority = GetHighestActivePriority(ICH_AP0R_EL2, ICH_AP1R_EL2, Zeroes());
        rval<7:1> = vPriority;
        rval<0> =  (vPriority == 0x7f); // return 0xff if no active priorities.
        return rval;

// Get physical priority.
    bits(8) pPriority = Zeroes();

    pPriority<7:1> = GetHighestActivePriority(ICC_AP0R_EL1, ICC_AP1R_EL1NS, ICC_AP1R_EL1S);
    pPriority<0> == (pPriority<7:1> == 0x7f); // 0xff if no active priorities

    if (!IsEL3OrMon() && SCR_GEN().NS == '1' &&  SCR_GEN().FIQ == '1') then
        // A non-secure GIC access and group 0 inaccessible to non-secure.
        if pPriority<7> == '0' then
            // Priority is in secure half and not visible to non-secure
            pPriority<7:0> = 0;
        else if pPriority != 255
            // Non secure access and not idle, so physical priority must be shifted
            pPriority<7:0> = LSL((pPriority AND P_MASK), 1);

    rval<7:0> = pPriority;

    return rval;
```


5.7.29 ICC_SGI0R_EL1, ICC_SGI1R_EL1 and ICC_ASGI1R_EL1

When SRE is set for an interrupt regime, these registers are used instead of the memory mapped GICD_SGIR register.

When any of ICC_SGI0R_EL1, ICC_SGI1R_EL1 or ICC_ASGI1R_EL1 are written, this results in the generation of interrupts to a set of processors “a.b.c.{target list}”, where “target list” is a set of up to sixteen processors within the affinity cluster defines by “a.b.c.*” (see section 4.2.1).

Note: if SRE is set ***only*** for secure EL3, software executing at EL3 might use the system register interface to generate SGIs. Hence, the Distributor must always be able to receive and acknowledge Generate SGI packets received from CPU interface regardless of the ARE settings for a security state. However, the Distributor may discard such packets.

Note: virtual accesses to these registers always trap to EL2 (HYP_TRAP). See section 4.6.7.

Note: accesses to these registers trap to EL2 (HYP_TRAP) if ICH_HCR_EL2.TC == ‘1’;

These registers provided the following capabilities:

- ICC_SGI0R_EL1 provides software the ability to generate secure group 0 SGIs, including from the non-secure state when permitted by GICR_NSACR.
- ICC_SGI1R_EL1 provides software the ability to generate group 1 SGIs for its own security state. That is, non-secure EL1 / EL2 accesses may generate non-secure group 1 interrupts and secure EL1 / EL3 may generate secure group 1 interrupts.
- ICC_ASGI1R_EL1 provides software the ability to generate group 1 SGIs for the other security state. That is, non-secure EL1 / EL2 accesses may generate secure group 1 interrupts and secure EL1 / EL3 may generate non-secure group 1 interrupts.

The table below shows the behaviour of these registers:

Access type	Register written	SGI configuration for target processor	Forward SGI to the destination?
EL3 / Secure EL1	ICC_SGI1R_EL1 (see note below)	Secure Group 0	Yes, if GICD_CTLR.DS is one.
		Secure Group 1	Yes
		Non-Secure Group 1	No
	ICC_ASGI1R_EL1	Secure Group 0	No
		Secure Group 1	No
		Non-Secure Group 1	Yes
Non-secure EL1 / EL2	ICC_SGI1R_EL1	Secure Group 0	Yes, if permitted by the appropriate GICR_NSACR field at each target processor, or if GICD_CTLR.DS is one.
		Secure Group 1	Yes, if permitted by the appropriate GICR_NSACR field at each target processor.
		Non-secure Group 1	Yes
	ICC_ASGI1R_EL1 (see note below)	Secure Group 0	Yes, if permitted by the appropriate GICR_NSACR field at each target processor, or if GICD_CTLR.DS is one.
		Secure Group 1	Yes, if permitted by the appropriate GICR_NSACR field at each target processor.

		Non-Secure Group 1	No
-	ICC_SGI0R_EL1	Secure Group 0	For non-secure EL1 / EL2 accesses, if permitted by the appropriate GICR_NSACR field at each target processor, or if GICD_CTLR.DS is set to one
		Group 1	No

Table 28: Effect of security on system register SGI generation

Notes:

- When SRE is zero for the Secure EL1 interrupt regime or GICD_CTLR.DS is set to one, Secure Group 1 interrupts are treated as Group 0 by the Distributor and when the table above indicates a Secure Group 1 interrupt should be generated, the Distributor must send a Secure Group 0 interrupt to the CPU interface logic

The format of these registers is as defined below:

- Bits [63:56]. Reserved. RES0.
- Bits [55:48]. Affinity 3. The affinity 3 value of the affinity path of the cluster for which SGI interrupts will be generated (see section 4.2.1)
    - ***Note***: see section 5.7.34 for whether non-zero affinity 3 values are supported.
- Bits [47:41]. Reserved. RES0.
- Bit [40]. Interrupt Routing Mode. See Table 4 above. ***Note***: when this field is one, the Target List, Affinity 1, Affinity 2 and Affinity 3 fields are RES0.
- Bits [39:32]. Affinity 2. The affinity 2 value of the affinity path of the cluster for which SGI interrupts will be generated (see section 4.2.1).
- Bits [31:28]. Reserved. RES0.
- Bits [27:24]. SGI Interrupt ID.
- Bit [23:16]. Affinity 1. The affinity 1 value of the affinity path of the cluster for which SGI interrupts will be generated (see section 4.2.1).
- Bits [15:0]. Target List. The set of processor for which SGI interrupts will be generated. Each bit corresponds to the processor within a cluster with an Affinity 0 value equal to the bit number. ***Note***: if a bit is one and the bit does not correspond to a valid target processor, the bit must be ignored by the Distributor. In such cases, a Distributor may optionally generate an SEI.
    - ***Note***: this restricts distribution of SGIs to the first 16 processors of an affinity 1 cluster.

Note: if software specifies Affinity 3 (if appropriate), Affinity 2 and Affinity 1 values that do not correspond to a valid set of target processors the Distributor must acknowledge and discard the Generate SGI packet. It may optionally generate an SEI.

Note: Secure Group 1 interrupts are only supported when SRE is set for both the Secure EL3 and the Secure EL1 interrupt regimes.

5.7.30 ICC_SGI0R, ICC_SGI1R and ICC_ASGI1R

When SRE is set for an interrupt regime and the interrupt regime is configured to be AArch32, these registers are used instead of the memory mapped GICD_SGIR register. These registers operate exactly as the AArch64 registers defined above and the format of the registers is identical.

Note: because these registers require 64 bits of data, they must be accessed using MCRR instructions.

5.7.31 ID_AA64PFR0_EL1

This AArch64 processor register has been extended to allow discovery of support for the GIC system register interface. The following bits have been added:

- Bits [27:24]. GIC System Register support
  - 0x0. No GIC system registers are supported
  - 0x1. GICv3 and GICv4 system registers are supported
  - 0x2 – 0xf. Reserved.

5.7.32 ID_PFR1

This AArch32 processor register has been extended to allow discovery of support for the GIC CP15 interface. The following bits have been added:

- Bits [31:28]. GIC CP15 support
  - 0x0. No GIC CP15 registers are supported
  - 0x1. GICv3 and GICv4 CP15 registers are supported
  - 0x2 – 0xf. Reserved

5.7.33 ICC_CTLR_EL1

This 32 bit register is banked by security. It controls aspects of the behaviour of the GIC CPU interface and provides information about the features implemented. The format of this register is shown below:

- Bits [31:16]. Reserved. RES0.
- Bit [15]. A3V. Read-only and writes are ignored.
  - 0b0. The CPU interface logic does not support non-zero values of Affinity 3 in SGI generation system registers. See section 5.7.29.
  - 0b1. The CPU interface logic supports non-zero values of Affinity 3 in SGI generation system registers. See section 5.7.29.
  - ***Note***: virtual accesses return the value from ICH_VTR_EL2.A3V.
- Bit [14]. SEIS. Read-only and writes are ignored.
  - 0b0. The CPU interface logic does not support local generation of SEIs by the CPU interface.
  - 0b1. The CPU interface logic supports local generation of SEIs by the CPU interface.
  - ***Note***: virtual accesses return the value from ICH_VTR_EL2.SEIS.
- Bits [13:11]. IDbits. Read-only and writes are ignored. The number of physical interrupt identifier bits supported:
  - 0b000. 16 bits.
  - 0b001. 24 bits.
  - 0b010 – 0b111. Reserved.
  - ***Note***: virtual accesses return the value from ICH_VTR_EL2.IDbits.
- Bits [10:8]. PRIbits. Read-only and writes are ignored. The number of priority bits implemented, minus one.
  - ***Note***: for physical accesses, this field determines the minimum value of ICC_BPR0_EL1.
  - ***Note:*** this field always returns the number of bits implemented, regardless of the security state of the access and value of GICD_CTLR.DS.
  - ***Note***: virtual accesses return the value from ICH_VTR_EL2.PRIbits.
- Bits [7]. Reserved. RES0.
- Bit [6]. PMHE. Priority Mask Hint Enable.
  - If EL3 is present this bit is an alias of ICC_CTLR_EL3.PMHE
    - When accessed at EL1, EL2 or EL3 and GICD_CTLR.DS == ‘0’, this bit is read-only
    - If ICC_CTLR_EL3.PMHE is writable, when accessed at EL1, EL2 or EL3 and GICD_CTLR.DS == ‘1’ this bit is read-write
  - If EL3 is not present and the field is writeable, this field resets to zero.
  - ***Note***: if EL3 is not present, an implementation might choose to make this field RAO/WI.
  - See section 4.6.11 for the effects of disabling security (GICD_CTLR.DS == ‘1’).
  - ***Note***: this field is RES0 for virtual accesses.
- Bits [5:2]. Reserved. RES0.
- Bit [1]. EOImode. The EOI mode for the current security state
  - If EL3 is present, this bit is a read/write alias of ICC_CTLR_EL3.EOImode_EL1{S,NS} as appropriate.
  - ***Note***: if EL3 is not present, an implementation might choose to make this field RAO/WI.
  - Virtual accesses modify ICH_VMCR_EL2.VEOIM.
- Bit [0]. CBPR. Common Binary Point Register.
  - If EL3 is present this bit is an alias of ICC_CTLR_EL3.CBPR_EL1{S,NS} as appropriate
    - When accessed at EL1, EL2 or EL3 and GICD_CTLR.DS == ‘0’, this bit is read-only
    - When accessed at EL1, EL2 or EL3 and GICD_CTLR.DS == ‘1’, this bit is read-write
    - The actual bit accessed depends on the current value of SCR_EL3.NS.
  - If EL3 is not present, this field resets to zero.
  - Virtual accesses modify ICH_VMCR_EL2.VCBPR. ***Note***: this bit affects virtual access to ICC_BPR1_EL1. See section 5.9.5 for details.
  - See section 4.6.11 for the effects of disabling security (GICD_CTLR.DS == ‘1’).

Note: an access is “virtual” when accessed at non-secure EL1 and either of FIQ or IRQ has been virtualized. That is, when (SCR_EL3.NS == ‘1’ && (HCR_EL2.FMO == ‘1’ || HCR_EL2.IMO == ‘1’)).

Note: when operating at EL3 and either EL3 is AArch64 or, EL3 is AArch32 and is in Monitor mode, the copy appropriate to the current setting of SCR_EL3.NS is accessed (see section 4.4.10).

5.7.34 ICC_CTLR_EL3

This 32 bit register is not present in systems without EL3. It controls aspects of the behaviour of the GIC CPU interface and provides information about the features implemented. The format of this register is shown below:

- Bits [31:16]. Reserved. RES0.
- Bit [15]. A3V. Read-only and writes are ignored.
  - 0b0. The CPU interface logic does not support non-zero values of Affinity 3 in SGI generation system registers. See section 5.7.29.
  - 0b1. The CPU interface logic supports non-zero values of Affinity 3 in SGI generation system registers. See section 5.7.29.
- Bit [14]. SEIS. Read-only and writes are ignored.
  - 0b0. The CPU interface logic does not support generation of SEIs.
  - 0b1. The CPU interface logic supports generation of SEIs.
- Bits [13:11]. IDbits. Read-only and writes are ignored. The number of physical interrupt identifier bits supported:
  - 0b000. 16 bits.
  - 0b001. 24 bits.
  - 0b010 – 0b111. Reserved.
- Bits [10:8]. PRIbits. Read-only and writes are ignored. The number of priority bits implemented, minus one.
  - ***Note***: this field determines the minimum value of ICC_BPR0_EL1.
- Bit [7]. Reserved. RES0.
- Bit [6]. PMHE. Priority Mask Hint Enable. Resets to zero if the field is writeable. When set, enables use of the PMR as a hint for interrupt distribution.
  - ***Note***: when changing PMHE from zero to one, software must write to ICC_PMR_EL1 to ensure the distributor is informed of the value of PMR.
  - ***Note***: when changing PMHE from one to zero, software must write ICC_PMR_EL1 to 0xff prior to writing PMHE to zero.
  - ***Note***: an implementation might choose to make this field RAO/WI
- Bit [5]. RM. Routing Modifier. ***Note***: in systems without EL3 this bit is RES0.
  - This bit is used to modify the behaviour of ICC_IAR0_EL1 and ICC_IAR1_EL1 such that systems with “legacy” secure software may be supported correctly (as described in Section 4.6.5). ***Note***: to ensure correct system behaviour, software ***must*** ensure this bit is zero when the secure copy of ICC_SRE_EL1.SRE is one.
  - 0b0. Reading ICC_IAR0_EL1 and ICC_IAR1_EL1 at EL3 in AArch64 or Monitor mode in AArch32 acknowledges interrupts normally.
  - 0b1. Reading ICC_IAR0_EL1 and ICC_IAR1_EL1 at EL3 in AArch64 or Monitor mode in AArch32 returns special values as defined in Section 4.6.5.
  - The pseudocode for ICC_IAR0_EL1 (section 5.7.11) and ICC_IAR1_EL1 (see section 5.7.13) provides further details of the special values returned.
- Bit [4]. EOImode_EL1NS. EOI mode for interrupts handled at non-secure EL1/2 (i.e. the accesses to EOIR and DIR are performed at non-secure EL1/2).
  - ***Note***: an implementation might choose to make this field RAO/WI.
- Bit [3]. EOImode_EL1S. EOI mode for interrupts handled at secure EL1 or EL3 when EL3 is using AArch32 and not in monitor mode.
  - This bit covers accesses to EOIR and DIR performed in one of the above interrupt regimes.
  - ***Note***: an implementation might choose to make this field RAO/WI.
- Bit [2]. EOImode_EL3. EOI mode for interrupts handled at EL3 when EL3 is using AArch64 or when EL3 is using AArch32 and in monitor mode.
  - This bit covers accesses to EOIR and DIR performed in one of the above interrupt regimes.
  - ***Note***: an implementation might choose to make this field RAO/WI.
- Bit [1]. CBPR_EL1NS. When set, non-secure EL1 and non-secure EL2 accesses to GICC_BPR (see section 5.6.8) and ICC_BPR1_EL1 (see section 5.7.25) access the state of ICC_BPR0_EL1 (see section 5.7.26). ICC_BPR0_EL1 is used to determine the pre-emption group for non-secure Group 1 interrupts.

- Bit [0]. CBPR_EL1S. When set, secure EL1 and EL3 accesses when EL3 is using AArch32 and not in monitor mode accesses to ICC_BPR1_EL1 access the state of ICC_BPR0_EL1. ICC_BPR0_EL1 is used to determine the pre-emption group for secure Group 1 interrupts.

5.7.35 ICC_IGRPEN0_EL1

This register controls whether Group 0 interrupts are enabled or not. The format of this register is shown below:

- Bit [0]. Enable. Resets to zero.

The lowest exception level at which this register may be accessed is governed by the exception level to which FIQ is routed, as defined in Section 4.6.6.

Virtual accesses to this register update ICH_VMCR_EL2.VENG0.

Note: if an interrupt is pending within the CPU interface when software writes Enable from one to zero, the interrupt must be immediately released by the CPU interface logic to allow the Distributor to forward the interrupt to a different processor (see section 7.4.3).

5.7.36 ICC_IGRPEN1_EL1

This register is banked by security and controls whether Group 1 interrupts are enabled for the security state determined from SCR_EL3.NS. The format of this register is shown below:

- Bit [0]. Enable. Resets to zero.

The lowest exception level at which this register may be accessed is governed by the exception level to which IRQ is routed, as defined in Section 4.6.6.

Virtual accesses to this register update ICH_VMCR_EL2.VENG1.

Note: if an interrupt is pending within the CPU interface when software write Enable from one to zero, the interrupt must be immediately released by the CPU interface logic to allow the Distributor to forward the interrupt to a different processor (see section 7.4.3).

Note: when operating at EL3 and either EL3 is AArch64 or, EL3 is AArch32 and is in Monitor mode, the copy appropriate to the current setting of SCR_EL3.NS is accessed (see section 4.4.10).

5.7.37 ICC_IGRPEN1_EL3

This register provides access to the group 1 enables for both security states. The format of this register is shown below:

- Bit [1]. EnableGrp1, secure. Resets to zero.
- Bit [0]. EnableGrp1, non-secure. Resets to zero.

Note: if an interrupt is pending within the CPU interface when software writes an Enable from one to zero, the interrupt must be immediately released by the CPU interface logic to allow the Distributor to forward the interrupt to a different processor (see section 7.4.3).

5.7.38 ICC_SRE_EL1

This register is banked by security and governs whether the system register interface or the memory mapped interface to the GIC CPU interface is to be used for EL0/1. The format of this register is shown below:

- Bit [31:3]. Reserved.
- Bit [2]. DIB. Disable IRQ bypass.
  - If EL3 is present and GICD_CTLR.DS is zero, this field is a read-only alias of ICC_SRE_EL3.DIB.
  - If EL3 is not present or GICD_CTLR.DS is one, and EL2 is present, this field is a read-only alias of ICC_SRE_EL2.DIB.
  - If EL3 is present and GICD_CTLR.DS is one, and EL2 is not present, this field is a read-write alias of ICC_SRE_EL3.DIB.
  - If EL3 is not present and EL2 is not present, this field resets to zero.
- Bit [1]. DFB. Disable FIQ bypass.
  - If EL3 is present and GICD_CTLR.DS is zero, this field is a read-only alias of ICC_SRE_EL3.DFB.
  - If EL3 is not present or GICD_CTLR.DS is one, and EL2 is present, this field is a read-only alias of ICC_SRE_EL2.DFB.
  - If EL3 is present and GICD_CTLR.DS is one, and EL2 is not present, this field is a read-write alias of ICC_SRE_EL3.DFB.
  - If EL3 is not present and EL2 is not present, this field resets to zero.
- Bit [0]. SRE. System Register Enable. Resets to zero unless the field is RAO/WI (see section 4.4.7).
  - 0b0. The memory mapped interface must be used. Access to any other ICC_* system register results in an undefined exception
  - 0b1. The system register interface for the current security state is enabled.
  - If EL3 is present, the secure copy of this bit is treated as zero (and is RAZ/WI) if ICC_SRE_EL3.SRE is zero. If ICC_SRE_EL3.SRE is changed from zero to one, the secure copy of this bit becomes UNKNOWN.
  - If EL3 is present, the non-secure copy of this bit is treated as one (and is RAO/WI) if the secure copy of the bit is one and any of HCR_EL2.{FMO, IMO, AMO} are zero.
  - When the non-secure copy of this bit is not being treated as one (i.e. when EL3 is not present or when the secure copy of this bit is zero), the non-secure copy of this bit is treated as zero (and is RAZ/WI) if EL2 is present and ICC_SRE_EL2.SRE is zero and is not treated as one or if EL2 is present and ICC_SRE_EL2.SRE is treated as zero.
  - If EL3 is present, if the secure copy of this bit is changed from one to zero, or if the secure copy of the bit is one and all of HCR_EL2.{FMO, IMO, AMO} become one, or if the effective value of ICC_SRE_EL2.SRE is changed from zero, the non-secure copy of the bit becomes UNKNOWN.

If EL2 is present and ICC_SRE_EL2.Enable is zero, non-secure EL1 accesses to this register will trap to EL2.

If EL2 is present and ICC_SRE_EL2.Enable is one, and EL3 is present and ICC_SRE_EL3.Enable is zero, non-secure EL1 accesses to this register will trap to EL3.

If EL2 is not present and ICC_SRE_EL3.Enable is zero, non-secure EL1 accesses to this register will trap to EL3.

If EL3 is present, is using AArch64 and ICC_SRE_EL3.Enable is zero, secure EL1 and non-secure EL2 accesses to this register will trap to EL3.

If EL3 is present, is using AArch32 and ICC_SRE_EL3.Enable is zero, non-Monitor EL3 and non-secure EL2 accesses to this register will generate an UNDEF_cfg exception.

Changes to the value of the secure copy of ICC_SRE_EL1.SRE have the following effects on the CPU interface registers:

- If the secure copy of ICC_SRE_EL1.SRE changes from zero to one, all ICC_* state, except the shared state defined in section 5.5.2 and any state that has a defined reset value, becomes UNKNOWN.
- If the secure copy of ICC_SRE_EL1.SRE changes from one to zero, all GICC_* state, except the shared state defined in section 5.5.2, becomes UNKNOWN. ***Note***: if the secure copy of ICC_SRE_EL1.SRE changes from one to zero, the reset value of any ICC_* state may no longer be relied upon and any such state becomes UNKNOWN.

- ***Note***: if the secure copy of ICC_SRE_EL1.SRE changes from one to zero, this might change the treated value of ICC_SRE_EL2.SRE or the treated value of non-secure copy of ICC_SRE_EL1.SRE. Hence it is strongly recommended that the secure copy of ICC_SRE_EL1.SRE is NEVER changed from one to zero.

Note: when operating at EL3 and either EL3 is AArch64, or EL3 is AArch32 and is in Monitor mode, the copy appropriate to the current setting of SCR_EL3.NS is accessed (see section 4.4.10).

5.7.39 ICC_SRE_EL2

If EL3 is not present this register is always system register accessible. This register governs whether the system register interface or the memory mapped interface to the GIC CPU interface is to be used for EL2. The format of this register is shown below:

- Bit [31:4]. Reserved.
- Bit [3]. Enable. Enables lower exception level access to ICC_SRE_EL1. Resets to zero.
  - 0b0. Non-secure EL1 accesses to ICC_SRE_EL1 trap to EL2
  - 0b1. Non-secure EL1 accesses to ICC_SRE_EL1 are permitted if EL3 is not present or ICC_SRE_EL3.Enable is one, otherwise Non-secure EL1 accesses to ICC_SRE_EL1 trap to EL3.
  - If ICC_SRE_EL2.SRE is zero or if EL3 is present and ICC_SRE_EL3.SRE is zero, this bit is treated as one.
- Bit [2]. DIB. Disable IRQ bypass.
  - If EL3 is present and GICD_CTLR.DS is zero, this field is a read-only alias of ICC_SRE_EL3.DIB
  - If EL3 is present and GICD_CTLR.DS is one, this field is a read-write alias of ICC_SRE_EL3.DIB
  - If EL3 is not present this bit resets to zero.
- Bit [1]. DFB. Disable FIQ bypass.
  - If EL3 is present and GICD_CTLR.DS is zero, this field is a read-only alias of ICC_SRE_EL3.DFB
  - If EL3 is present and GICD_CTLR.DS is one, this field is a read-write alias of ICC_SRE_EL3.DFB
  - If EL3 is not present this bit resets to zero.
- Bit [0]. SRE. System Register Enable. Resets to zero unless the field is RAO/WI (see section 4.4.7).
  - 0b0. The memory mapped interface must be used. Access to any ICH_* system register results in an undefined exception
  - 0b1. The system register interface to ICC_*_EL1 and ICH_* is enabled for EL2.
  - If EL3 is present, this bit is treated as zero (and is RAZ/WI) if ICC_SRE_EL3.SRE is zero. If ICC_SRE_EL3.SRE is changed from zero to one, this bit becomes UNKNOWN.
  - Otherwise, this bit is treated as one (and is RAO/WI) if EL3 is present and the secure copy of ICC_SRE_EL1.SRE is one. If the secure copy of ICC_SRE_EL1.SRE is changed from one to zero, this bit becomes UNKNOWN.

If EL3 is present, is using AArch64 and ICC_SRE_EL3.Enable is zero, EL2 accesses to this register will trap to EL3.

If EL3 is present, is using AArch32 and ICC_SRE_EL3.Enable is zero, EL2 accesses to this register will generate an UNDEF_cfg exception.

Changes to the value or treatment of ICC_SRE_EL2.SRE have the following effects on the registers that control virtual interrupts for the non-secure EL1 interrupt regime:

- If ICC_SRE_EL2.SRE or its treated value changes from zero to one, all ICH_* state becomes UNKNOWN. ***Note***: this means that software must write to the system registers that control virtual interrupts for the non-secure EL1 regime to ensure they are in a valid state before execution at non-secure EL1 is allowed.
- If ICC_SRE_EL2.SRE or its treated value changes from one to zero, all GICH_* state becomes UNKNOWN. ***Note***: this means that software must write to the memory-mapped registers that control virtual interrupts for the non-secure EL1 regime to ensure they are in a valid state before execution at non-secure EL1 is allowed.

Note: if a hypervisor wishes to dynamically change its SRE setting, in addition to the rules in section 4.4.8, it must ensure it has saved the state of the current guest operating system before changing SRE. See sections 4.6.8 and 4.8.17.

5.7.40 ICC_SRE_EL3

This register is always system register accessible and governs whether the system register interface or the memory mapped interface to the GIC CPU interface is to be used for EL3. The format of this register is shown below:

- Bit [31:4]. Reserved.
- Bit [3]. Enable. Enables lower exception level access to ICC_SRE_EL1/2. Resets to zero.
  - 0b0. EL1/2 accesses to ICC_SRE_EL1/2 trap to EL3
  - 0b1. EL2 accesses to ICC_SRE_EL2 are permitted
  - If ICC_SRE_EL3.SRE is zero, this bit is treated as one.
- Bit [2]. DIB. Disable IRQ bypass. Resets to zero. When ICC_SRE_EL3.SRE is zero, the IRQ bypass controls are as GICv2. Otherwise, when ICC_SRE_EL3.SRE is one:
  - 0b0. The bypass input is selected according to the GICv2 logic (see Table 2.2 of [2]).
  - 0b1. The bypass input is never selected.
  - ***Note***: when all interrupt regimes are using system registers it is expected that software will not program any of the GICv2 bypass controls before setting the secure copy of ICC_SRE_EL1.SRE to one, and the bypass input is selected when no supported and enabled (see section 5.7.36) interrupt group is allocated to IRQ and DIB is zero (see section 4.6.3 for how interrupts are allocated).
  - ***Note***: in systems when one or more system register enables are RAO/WI, some conditions in the GICv2 bypass logic are inaccessible to software and can be assumed to always have their reset values.
  - ***Note***: in systems which do not support bypass, this bit may be RAO/WI.
- Bit [1]. DFB. Disable FIQ bypass. Resets to zero. When ICC_SRE_EL3.SRE is zero, the FIQ bypass controls are as GICv2. Otherwise, when ICC_SRE_EL3.SRE is one:
  - 0b0. The bypass input is selected according to the GICv2 logic (see Table 2.3 of [2]).
  - 0b1. The bypass input is never selected.
  - ***Note***: when all interrupt regimes are using system registers it is expected that software will not program any of the GICv2 bypass controls before setting the secure copy of ICC_SRE_EL1.SRE to one, and the bypass input is selected when no supported and enabled (see sections 5.7.35 and 5.7.36) interrupt group is allocated to FIQ and DFB is zero (see section 4.6.3 for how interrupts are allocated).
  - ***Note***: in systems when one or more system register enables are RAO/WI, some conditions in the GICv2 bypass logic are inaccessible to software and can be assumed to always have their reset values.
  - ***Note***: in systems which do not support bypass, this bit may be RAO/WI.
- Bit [0]. SRE. Secure EL3 System Register Enable. Resets to zero unless the field is RAO/WI (see section 4.4.7).
  - ***Note***: once set to one, an implementation may choose to make the bit RAO/WI and only cleared by a hardware reset.

Changes to the value of ICC_SRE_EL3.SRE have the following effects on the CPU interface registers:

- If the value of ICC_SRE_EL3.SRE changes from zero to one, all ICC_* state becomes UNKNOWN except for any state that has defined a reset value.
- If the value of ICC_SRE_EL3.SRE changes from one to zero, all GICC_* state becomes UNKNOWN. ***Note***: if ICC_SRE_EL3.SRE changes from one to zero, the reset value of any ICC_* state may no longer be relied upon and any such state becomes UNKNOWN.

Note: when EL3 is using AArch32, the AArch32 mapping of this register (ICC_MSRE) is subject to CP15SDISABLE.

5.7.41 ICC_AP0R{0..3}_EL1

These 32 bit system registers provide information about the Group 0 active priorities for the current interrupt regime. These registers reset to zero. Access to this register is governed by the rules in section 4.6.6 and following. See section 5.7.43 for the rules governing the data values of this register.

The bits in ICC_AP0R{0..3}_EL1 correspond to priority groups according to the following relationships:

- Bit *N* corresponds to priority group *M* divided by $2^{(U+1)}$, where *U* is the number of unimplemented bits.
- ICC_AP0R*n*_EL1 contains bits (*n* * 32) to (*n* * 32) + 31.

Note: in implementations supporting fewer than 8 bits of priority, some of these registers correspond to unimplemented priority levels. Access to such registers will generate an UNDEF_und exception (see Table 38: GIC Exception Behaviour).

5.7.42 ICC_AP1R{0..3}_EL1

These 32 bit system registers, which are banked by security, provide information about the Group 1 active priorities for the current interrupt regime. These registers reset to zero. Access to this register is governed by the rules in section 4.6.6 and following. See section 5.7.43 for the rules governing the data values of this register.

The bits in ICC_AP1R{0..3}_EL1 correspond to priority groups according to the following relationships:

- Bit *N* corresponds to priority group *M* divided by $2^{(U+1)}$, where *U* is the number of unimplemented bits.
- ICC_AP1R*n*_EL1 contains bits (*n* * 32) to (*n* * 32) + 31

Note: in systems supporting two security states where GICD_CTLR.DS is zero, non-secure accesses see a shifted view of priorities.

Note: in implementations supporting fewer than 8 bits of priority, some of these registers correspond to unimplemented priority levels. Access to such registers will generate an UNDEF_und exception (see Table 38: GIC Exception Behaviour).

5.7.43 Interrupt Regime effects on ICC_AP0R*n*_EL1 and ICC_AP1R*n*_EL1

Accesses to these registers from a given interrupt regime provide a view of the active priorities appropriate for that interrupt regime. That is, to allow save and restore of the appropriate state.

Note: only a single interrupt group can be active for any active priority bit and non-secure accesses cannot "overwrite" active priorities associated with secure interrupt groups.

The table below shows the effect of interrupt regime and GICD_CTLR.DS on accesses to these registers. Unless otherwise stated in an entry, when a bit is successfully set to one, this will clear any other active priorities corresponding to the bit:

Current Exception Level and Security State	ICC_AP0R*n*_EL1		ICC_AP1R*n*_EL1	
	Read accesses	Write accesses	Read accesses	Write accesses
EL3 when AArch64 or, AArch32 and Monitor mode	Permitted Accesses Group 0 Secure active priorities	Permitted Accesses Group 0 Secure active priorities	Permitted When SCR_EL3.NS is zero, accesses Group 1 Secure active priorities When SCR_EL3.NS is one, accesses Group 1 Non-secure active priorities (unshifted)	Permitted When SCR_EL3.NS is zero, accesses Group 1 Secure active priorities and bits are only updated if the corresponding Group 0 Secure active priorities are zero. When SCR_EL3.NS is one, accesses Group 1 Non-secure active priorities (unshifted) and bits are only updated if the corresponding Group 0 Secure and Group 1 Secure active priorities are zero.
Secure EL1 or EL3 when AArch32 and not Monitor mode	Permitted Accesses Group 0 Secure active priorities	Permitted Accesses Group 0 Secure active priorities	Permitted Accesses Group 1 Secure active priorities (unshifted)	Permitted Accesses Group 1 Secure active priorities (unshifted). When a bit is written, the bit is only updated if the corresponding Group 0 Secure active priority is zero.
Non-secure EL1/2 when EL3 is present and GICD_CTLR.DS is zero	Permitted Accesses Group 0 Secure active priorities	Permitted Accesses Group 0 Secure active priorities	Permitted Accesses Group 1 Non-Secure active priorities (shifted)	Permitted Accesses Group 1 Non-Secure active priorities (shifted) When a bit is written, the bit is only updated if the corresponding Group 0 Secure and Group 1 Secure active priority is zero.
Non-Secure EL1 virtual access	ICH_AP0Rn_EL2	ICH_AP0Rn_EL2	ICH_AP1Rn_EL2	ICH_AP1Rn_EL2 When a bit is written, the bit is only updated if the corresponding priority in ICH_AP0Rn_EL2 is zero.
Non-secure EL1/2 when EL3 not present or GICD_CTLR.DS is	Permitted Accesses Group 0 active priorities	Permitted Accesses Group 0 active priorities	Permitted Accesses Group 1 Non-Secure active priorities (unshifted)	Permitted Accesses Group 1 Non-Secure active priorities (unshifted) When a bit is set, the bit is only

one				updated if the Group 0 active priority is zero.

Table 29: Interrupt regime and view of Active Priorities

Note: non-secure EL1/2 access to ICC_AP0Rn_EL1 is permitted whenever SCR_EL3.FIQ is zero, regardless of the value of GICD_CTLR.DS. However, it is expected that non-secure access to ICC_AP0Rn_EL1 will not be permitted in systems that support security and that software will set SCR_EL3.FIQ to one in such systems.

Note: when Secure Group 1 interrupts are treated as Secure Group 0 (see section 4.1.3), software must ensure that the secure copy of ICC_AP1Rn_EL1 is zero otherwise system behaviour might be unpredictable.

5.8 Memory Mapped Hypervisor Control Registers

The intent of the changes to the Hypervisor Control registers is to:

- Allow virtualization of guest operating systems using GICv2.
- Allow system register access, when enabled by setting ICC_SRE_EL2.SRE for the non-secure state.

When system register access has not been enabled (i.e. when ICC_SRE_EL2.SRE == '0' for the non-secure state) the Hypervisor Control Registers must be memory mapped, and behave as defined in this section. When system registers access is enabled (i.e. when ICC_SRE_EL2.SRE == '1') the memory mapped registers are replaced by the registers defined in section 5.9.

Note: it is implementation defined whether an access to a Hypervisor control register using the memory-mapped interface accesses the same state as an access using the system register interface or whether the two interfaces access different state.

Register	Offset	Reset Value	Register Name
GICH_HCR	0x0000	0x00000000	Hypervisor Control Register
GICH_VTR	0x0004	IMP DEF	Virtual Type register
GICH_VMCR	0x0008	-	Virtual Machine Control Register
Reserved	0x000C	-	
GICH_MISR	0x0010	0x00000000	Maintenance Interrupt Status Register
Reserved	0x0014 – 0x001C	-	
GICH_EISR	0x0020	0x00000000	End Interrupt Status Register
Reserved	0x0024 – 0x002C	-	
GICH_ELSR	0x0030	IMP DEF	Empty List Register Status Register
Reserved	0x0034 – 0x00EC	-	
GICH_APR{0..3}	0x00F0 – 0x00FC	0x00000000	Active Priorities Registers
GICH_LR{0..15}	0x0100 – 0x013C	0x00000000	List Register {0..15} bits [31:0].

Table 30: Hypervisor Control Register Map

5.8.1 GICH_HCR

This register is unmodified from GICv2:

- Bit [31:27]. EOIcount. As GICv2.
- Bit [26:8]. Reserved. RES0.
- Bit [7]. VGrp1DIE. As GICv2.
- Bit [6]. VGrp1EIE. As GICv2.
- Bit [5]. VGrp0DIE. As GICv2.
- Bit [4]. VGrp0EIE. As GICv2.
- Bit [3]. NPIE. As GICv2.
- Bit [2]. LRENPIE. As GICv2.
- Bit [1]. UIE. As GICv2.
- Bit [0]. En. As GICv2.

5.8.2 GICH_VMCR

This register allows the hypervisor to save and restore the virtual machine view of the GIC state. The format of this register is shown below:

- Bit [31:24]. VPMR. Virtual Priority Mask. Visible to the guest as GICV_PMR.
- Bit [23:21]. VBPR0. Virtual BPR0. Visible to the guest as GICV_BPR.
- Bit [20:18]. VBPR1. Virtual BPR1. Visible to the guest as GICV_ABPR.
- Bit [17:10]. Reserved. RES0.
- Bit [9]. VEOIM. Virtual EOImode. Visible to the guest as GICV_CTLR.EOImode.
  - ***Note***: an implementation might choose to make this field RAO/WI.
- Bit [8:5]. Reserved. RES0.
- Bit [4]. VCBPR. Visible to the guest as GICV_CTLR.CBPR.
- Bit [3]. VFIQEn. Visible to the guest as GICV_CTLR.FIQEn.
- Bit [2]. VAckCtl. Visible to the guest as GICV_CTLR.AckCtl.
- Bit [1]. VENG1. Virtual group 1 interrupt enable. Visible to the guest as GICV_CTLR.EnableGrp1.
- Bit [0]. VENG0. Virtual group 0 interrupt enable. Visible to the guest as GICV_CTLR.EnableGrp0.

Note: when the hypervisor is memory-mapped, only memory mapped guests are supported.

5.8.3 GICH_MISR

This register is unmodified from GICv2:

- Bit [7]. VGrp1D. As GICv2.
- Bit [6]. VGrp1E. As GICv2.
- Bit [5]. VGrp0D. As GICv2.
- Bit [4]. VGrp0E. As GICv2.
- Bit [3]. NP. As GICv2. ***Note***: a List Register is in the pending state only if GICH_LRn_EL2.State is “pending”. The state “pending and active” is not included.
- Bit [2]. LRENP. As GICv2.
- Bit [1]. U. As GICv2.
- Bit [0]. EOI. As GICv2.

5.8.4 GICH_LRn

The GICH_LRn registers behave exactly as GICv2 (except there can be at most 16):

- Bit [31]. HW. As GICv2.
- Bit [30]. Group. As GICv2.
- Bits [29:28]. State. As GICv2.
- Bits [27:23]. Priority. As GICv2.
- Bits [22:20]. Reserved. RES0.
- Bits [19:10]. Physical ID. As GICv2.
- Bits [9:0]. Virtual ID. As GICv2.

Note: the number of List Registers supported has been reduced to 16.

5.8.5 GICH_APRn

These registers operate exactly as for GICv2.

5.9 System Register Access to Hypervisor Control Registers

The hypervisor control registers are accessible at EL2 when ICC_SRE_EL2.SRE == ‘1’.

The hypervisor control registers are accessible at EL3 when ICC_SRE_EL3.SRE == '1' and:

- EL3 is configured as AArch32, access is only permitted if SCR_EL3.NS == ‘1’ otherwise an UNDEF_und exception will be generated (see section 6.2).
- EL3 is configured as AArch64, access is always permitted
- ***Note***: because it is implementation defined whether the memory mapped GICH_* registers and the EL2 system registers access the same state, EL3 software must be aware of whether EL2 is using system registers (i.e. when the treated value of ICC_SRE_EL2.SRE is one) when determining whether to access either the GICH_* registers or the EL2 system registers.

5.9.1 Reset Values

The table below specifies the reset values for the Hypervisor Control system registers:

System Register Name		
AArch64	AArch32	Reset Value
ICH_HCR_EL2	ICH_HCR	0x00000000
ICH_VTR_EL2	ICH_VTR	Implementation defined. See section 5.9.4.
ICH_VMCR_EL2	ICH_VMCR	Implementation defined. See section 5.9.5.
ICH_MISR_EL2	ICH_MISR	0x00000000
ICH_EISR_EL2	ICH_EISR	0x00000000
ICH_ELSR_EL2	ICH_ELSR	Implementation defined. See section 5.9.12.
ICH_AP0R0_EL2	ICH_AP0R0	0x00000000
ICH_AP0R1_EL2	ICH_AP0R1	0x00000000
ICH_AP0R2_EL2	ICH_AP0R2	0x00000000
ICH_AP0R3_EL2	ICH_AP0R3	0x00000000
ICH_AP1R0_EL2	ICH_AP1R0	0x00000000
ICH_AP1R1_EL2	ICH_AP1R1	0x00000000
ICH_AP1R2_EL2	ICH_AP1R2	0x00000000
ICH_AP1R3_EL2	ICH_AP1R3	0x00000000
ICH_LR{0:15}_EL2	ICH_LR[0:15]	0x00000000_00000000
	ICH_LRC[0:15]	0x00000000

Table 31: Hypervisor Control System Registers Reset Values

5.9.2 ICH_HCR_EL2

This 32 bit register controls the environment for guest operating systems. The format of this register is shown below:

- Bits [31:27]. EOIcount. This field is incremented whenever a successful write to a virtual EOIR or DIR register would have resulted in a virtual interrupt deactivation. That is:
  - A virtual write to EOIR with a valid interrupt identifier that is not in the LPI range (i.e. < 8192) when EOI mode is zero and no List Register was found, or
  - A virtual write to DIR with a valid interrupt identifier that is not in the LPI range (i.e. < 8192) when EOI mode is one and no List Register was found
  - This allows software to manage more active interrupts than there are implemented List Registers.
- Bits [26:14]. Reserved. RES0.
- Bits [13]. TSEI. (“trap all locally generated SEIs”). See section 4.7.5. ***Note***: this bit allows the hypervisor to intercept locally generated SEIs that would otherwise be taken by a guest operating system at non-secure EL1.
  - 0b0: Locally generated SEIs do not cause a trap to EL2.
  - 0b1: Locally generated SEIs trap to EL2.
  - ***Note***: this bit is RES0 when ICH_VTR_EL2.SEIS is zero.
- Bits [12]. TALL1. (“trap all Non-Secure EL1 accesses to ICC_* system registers for group 1 interrupts”).
  - 0b0: Non-Secure EL1 accesses to ICC_* registers for group 1 interrupts proceed as normal.
  - 0b1: Any Non-Secure EL1 accesses to ICC_* registers for group 1 interrupts trap to EL2.
- Bits [11]. TALL0. (“trap all Non-Secure EL1 accesses to ICC_* system registers for group 0 interrupts”).
  - 0b0: Non-Secure EL1 accesses to ICC_* registers for group 0 interrupts proceed as normal.
  - 0b1: Any Non-Secure EL1 accesses to ICC_* registers for group 0 interrupts trap to EL2.
- Bit [10]. TC (“trap all Non-Secure EL1 accesses to system register common to Group 0 and Group 1”).
  - 0b0: Non-Secure EL1 accesses to common registers proceed as normal.
  - 0b1: Any Non-Secure EL1 access to common registers trap to EL2.
  - ***Note***: affects accesses to ICC_SGI0R_EL1, ICC_SGI1R_EL1, ICC_ASGI1R_EL1, ICC_CTLR_EL1, ICC_DIR_EL1, ICC_PMR_EL1 and ICC_RPR_EL1.
- Bit [9]. VARE. Virtual ARE.
  - 0b0: The guest operating system does not use affinity routing and expects a Source CPU ID for SGIs.
  - 0b1: The guest operating system uses affinity routing.
  - ***Note***: when a guest operating system is memory-mapped, it does not support LPIs and software must ensure that no LPIs are presented to the guest using either the List Registers or from the Distributor.
  - ***Note***: this bit has no hardware effect and an implementation might choose to make this bit RAZ/WI.
- Bit [8]. RES0.
- Bit [7]. VGrp1DIE. As GICv2
- Bit [6]. VGrp1EIE. As GICv2
- Bit [5]. VGrp0DIE. As GICv2
- Bit [4]. VGrp0EIE. As GICv2
- Bit [3]. NPIE. As GICv2
- Bit [2]. LRENPIE. As GICv2, except no maintenance interrupt is generated for an EOI that specifies an ID corresponding to the highest priority virtual LPI sent from the Distributor.
- Bit [1]. UIE. As GICv2

- Bit [0]. En. Enable. Global enable bit for the virtual CPU interface:
  - 0b0. Virtual CPU interface operation disabled.
  - 0b1. Virtual CPU interface operation enabled.
  - When this field is set to 0:
    - the virtual CPU interface does not signal any maintenance interrupts
    - the virtual CPU interface does not signal any virtual interrupts (including virtual system errors)
    - a virtual access to an interrupt acknowledge register returns a spurious interrupt ID.

5.9.3 ICH_MISR_EL2

This 32 bit register is used to determine the status of maintenance interrupts. The format of this register is as shown below:

- Bit [7]. VGrp1D. As GICv2.
- Bit [6]. VGrp1E. As GICv2.
- Bit [5]. VGrp0D. As GICv2.
- Bit [4]. VGrp0E. As GICv2.
- Bit [3]. NP. As GICv2. ***Note***: a List Register is in the pending state only if ICH_LRn_EL2.State is “pending”. The state “pending and active” is not included.
- Bit [2]. LRENP. As GICv2.
- Bit [1]. U. As GICv2.
- Bit [0]. EOI. As GICv2.

Note: the U and NP bits do not include the status of any pending / active VSET packets because these bits control generation of interrupts that allow software management of the contents of the List Registers (which are not affected by VSET packets).

5.9.4 ICH_VTR_EL2

This 32 bit read-only register describes the number of implemented virtual priority bits and List Registers. The format of this register is shown below:

- Bits [31:29]. PRIbits. The number of priority bits implemented, minus one.
- Bits [28:26]. PREbits. The number of preemption bits implemented, minus one. The value of this field must be less than or equal to PRIbits (above). This field determines the minimum value of ICH_VMCR_EL2.VBPR0.
- Bits [25:23]. IDbits. The number of virtual interrupt identifier bits supported:
  - 0b000. 16 bits.
  - 0b001. 24 bits.
  - 0b010 – 0b111. Reserved.
- Bit [22]. SEIS. Read-only and writes are ignored.
  - 0b0. The virtual CPU interface logic does not support local generation of SEIs by the CPU interface.
  - 0b1. The virtual CPU interface logic supports local generation of SEIs by the CPU interface.
- Bit [21]. A3V. Read-only and writes are ignored.
  - 0b0. The CPU interface logic does not support non-zero values of Affinity 3 in SGI generation system registers. See section 5.7.29.
  - 0b1. The CPU interface logic supports non-zero values of Affinity 3 in SGI generation system registers. See section 5.7.29.
- Bits [20:5]. Reserved.
- Bits [4:0]. ListRegs. The number of implemented List registers, minus one. For example, a value of 0xf indicates that the maximum of 16 List registers are implemented.

5.9.5 ICH_VMCR_EL2

This register allows the hypervisor to save and restore the virtual machine view of the GIC state. The format of this register is shown below:

- Bit [31:24]. VPMR. Virtual Priority Mask. Visible to the guest as ICC_PMR_EL1 / GICV_PMR.
- Bit [23:21]. VBPR0. Virtual BPR0. Visible to the guest as ICC_BPR0_EL1 / GICV_BPR.
- Bit [20:18]. VBPR1. Virtual BPR1. Visible to the guest as ICC_BPR1_EL1 / GICV_ABPR. ***Note***: this field is always accessible to EL2 accesses, regardless of the setting of the VCBPR field.
- Bit [17:10]. Reserved. RES0.
- Bit [9]. VEOIM. Virtual EOImode. Visible to the guest as ICC_CTLR_EL1.EOImode / GICV_CTLR.EOImode
  - ***Note***: an implementation might choose to make this field RAO/WI.
- Bit [8:5]. Reserved. RES0
- Bit [4]. VCBPR. Visible to the guest as ICC_CTLR_EL1.CBPR / GICV_CTLR.CBPR.
  - 0b0. Virtual reads and writes to ICC_BPR1_EL1 access ICH_VMCR_EL2.VBPR1.
  - 0b1. Virtual reads of ICC_BPR1_EL1 return (ICH_VMCR_EL2.VBPR0 + 1, saturated to 0b111) and virtual writes to ICC_BPR1_EL1 are ignored.
  - ***Note***: this bit has no effect on accesses to GICV_ABPR but does affect preemption.
- Bit [3]. VFIQEn. Visible to the guest as GICV_CTLR.FIQEn.
  - When the non-secure copy of ICC_SRE_EL1.SRE is one or is treated as one (see section 5.7.38), this bit is treated as one and is RAO. ***Note***: in this case Group 0 virtual interrupts will always be signaled to non-secure EL1 using the FIQ exception.
  - ***Note***: in implementations where the non-secure copy of ICC_SRE_EL1.SRE is always one, this bit is RES1.
- Bit [2]. VAckCtl. Visible to the guest as GICV_CTLR.AckCtl.
  - When the non-secure copy of ICC_SRE_EL1.SRE is one or is treated as one (see section 5.7.38), this bit is treated as zero and is RAZ.
  - ***Note***: this bit only affects accesses to GICV_IAR and must have no effect on virtual accesses to ICC_IAR0_EL1 or ICC_IAR1_EL1.
  - ***Note***: in implementations where the non-secure copy of ICC_SRE_EL1.SRE is always one, this bit is RES0.
- Bit [1]. VENG1. Virtual group 1 interrupt enable. Visible to the guest as ICC_IGRPEN1_EL1.Enable / GICV_CTLR.EnableGrp1.
- Bit [0]. VENG0. Virtual group 0 interrupt enable. Visible to the guest as ICC_IGRPEN0_EL1.Enable / GICV_CTLR.EnableGrp0.

Note: when the hypervisor is using system registers, guests using either system register or memory-mapped access must be supported.

5.9.6 ICH_LRn_EL2

When SRE is set for the non-secure EL2 interrupt regime and when EL2 is operating as AArch64, these registers replace the memory mapped GICH_LRn registers. To support the larger fields size required for LPIs, the List Registers are extended to 64-bits:

- Bit [63:62]. State. As GICv2
- Bit [61]. HW. As GICv2.
    - ***Note***: it is a programming error if the hypervisor sets this bit to one and uses the “Active and Pending” State. If this happens, interrupt corresponding to Physical ID will be deactivated twice and system behaviour might be unpredictable.
- Bit [60]. Group. As GICv2.
- Bits [59:56]. Reserved. RES0.
- Bits [55:48]. Priority[7:0].
    - ***Note***: it is implementation defined how many bits are implemented, though at least five bits must be implemented. Unimplemented bits are RES0 and start from bit [48]. Number of implemented bits determines how many GICH_APRn registers exist.
    - ***Note***: it is a programming error if the priority is set to the “lowest” implemented priority (i.e. the highest numeric value). Interrupts with this value can never assert an interrupt request as the value can never preempt and the value effectively means “idle”.
- Bits [47:42]. Reserved. RES0.
- Bits [41:32]. Physical ID. When HW is zero (i.e. there is no corresponding physical interrupt), some of these bits have a special meaning:
    - Bit [41]. EOI. Indicates whether this interrupt triggers a maintenance interrupt. As defined for GICv2.
    - Bits [40:32]. Reserved. RES0.
    - ***Note***: in GICv2, if the Physical ID specified an SGI behaviour was unpredictable. In GICv3, hardware deactivation of SGIs is fully supported.
    - ***Note***: a hardware physical identifier is only required in List Registers for interrupts that require an EOI or Deactivate. Hence, only 10 bits of Physical ID are required, regardless of the number specified by ICC_CTLR_EL1.IDbits.
- Bits [31:0]. Virtual ID.
    - ***Note***: when a guest operating system is using memory mapped access to the GIC, software must ensure the correct Source CPU ID is provided in bits [12:10].
    - ***Note***: software must ensure there is only a single valid entry for a given Virtual ID.
    - ***Note***: it is implementation defined how many bits are implemented, though at least 16 bits must be implemented. Unimplemented bits are RES0. The number of implemented bits can be discovered from ICH_VTR_EL2.IDbits.

The number of implemented Virtual ID bits may be discovered from ICH_VTR_EL2.IDbits. Unimplemented bits are RES0.

The number of implemented priority bits may be discovered from ICH_VTR_EL2.PRIbits.

The “HW” bit indicates whether an entry corresponds to a physical interrupt for which a de-activation is required. If the “Virtual ID” indicates that the virtual interrupt is an LPI, because no de-activation is required for LPIs, the de-activation of the physical interrupt must be sent when the appropriate EOI register is written, regardless of the setting of ICH_VMCR_EL2.VEOIM. ***Note***: virtual LPIs that correspond to a physical interrupt must always reside in the List Registers otherwise the hardware deactivation will not occur and it is recommended that software ensures entries with a Virtual ID > 8191 are kept resident in the list registers until the entry is no longer valid.

Note: the number of List Registers supported has been reduced to 16 to prevent excessive use of system register space.

Note: when multiple list registers contain interrupts with the same priority, the lower numbered list register is considered to be of a higher priority and is selected to be signaled.

Note: an implementation might choose to support fewer than 16 List Registers. Access to such registers will generate an UNDEF_und exception (see Table 38: GIC Exception Behaviour).

5.9.7 ICH_LRn

When SRE is set for the non-secure EL2 interrupt regime and when EL2 is operating as AArch32, these register provide access to ICH_LRn_EL2 bits [31:0].

Note: ICH_LRn and ICH_LRCn may be updated independently.

5.9.8 ICH_LRCn

When SRE is set for the non-secure EL2 interrupt regime and when EL2 is operating as AArch32, these register provide access to ICH_LRn_EL2 bits [63:32].

Note: ICH_LRn and ICH_LRCn may be updated independently.

5.9.9 ICH_AP0Rn_EL2

These 32 bit system registers provide the access to the virtual active priorities for Group 0 interrupts. The number of register implemented depends on how many bits of Priority are implemented in ICH_LRn_EL2 (this number can be discovered from ICH_VTR_EL2.PRIbits):

- 5-bits. ICH_AP0R0_EL2 must be implemented. Accesses to ICH_AP0R{1..3}_EL2 will generate an UNDEF_und exception.
- 6-bits. ICH_AP0R{0..1}_EL2 must be implemented. Accesses to ICH_AP0R{2..3}_EL2 will generate an UNDEF_und exception.
- 7-bits. ICH_AP0R{0..3}_EL2 must be implemented.

Only a single interrupt group can be active for any active priority bit in ICH_AP0Rn_EL2 and ICH_AP1Rn_EL2. This means that:

- Writes to ICH_AP0Rn_EL2 with bits set to one will set bits to one in ICH_AP0Rn_EL2 and will set the corresponding bits to zero in ICH_AP1Rn_EL2
- Writes to ICH_AP0Rn_EL2 with bits set to zero will set bits to zero bits in ICH_AP0Rn_EL2 and will not affect ICH_AP1Rn_EL2.

Note: software must ensure that ICH_AP0Rn_EL2 is zero for guest operating systems that use memory-mapped access to the GIC otherwise the effects are unpredictable.

Note: A maximum of 7 bits of priority (rather than 8) are supported by the ICC_AP0R{0..3}_EL1 and ICC_AP1R{0..3}_EL1 registers because when the binary point value is zero, the “group priority” field is bits [7:1] of the priority and bit [0] is never used for pre-emption (see section 3.3.3 of [2]).

Note: the active priorities for Group 0 and Group 1 interrupts for memory mapped guests (i.e. when non-secure EL1 is not using system registers as defined by the pseudocode in section 5.7.6) are held in ICH_AP1Rn_EL2 and virtual non-secure EL1 (i.e. guest) reads and writes to GICV_APR access ICH_AP1Rn_EL2. This means this register is inaccessible to memory mapped guests. It is recommended that EL2 software ensures this register is written to zero for memory mapped guests.

5.9.10 ICH_AP1Rn_EL2

These 32 bit system registers provide the access to the virtual active priorities for Group 1 interrupts. The number of register implemented depends on how many bits of Priority are implemented in ICH_LRn_EL2 (this number can be discovered from ICH_VTR_EL2.PRIbits):

- 5-bits. ICH_AP1R0_EL2 must be implemented. Accesses to ICH_AP1R{1..3}_EL2 will generate an UNDEF_und exception.
- 6-bits. ICH_AP1R{0..1}_EL2 must be implemented. Accesses to ICH_AP2R{2..3}_EL2 will generate an UNDEF_und exception.
- 7-bits. ICH_AP1R{0..3}_EL2 must be implemented.

Only a single interrupt group can be active for any active priority bit in ICH_AP0Rn_EL2 and ICH_AP1Rn_EL2. This means that:

- Writes to ICH_AP1Rn_EL2 with bits set to one will set bits to one in ICH_AP1Rn_EL2 only if the corresponding bits are zero in ICH_AP0Rn_EL2

Note: always used for memory mapped guests (i.e. when non-secure EL1 is not using system registers as defined by the pseudocode in section 5.7.6) regardless of the group of the virtual interrupt. Virtual non-secure EL1 (i.e. guest) reads and writes to GICV_APR access ICH_AP1Rn_EL2.

5.9.11 ICH_EISR_EL2

When SRE is set for the non-secure EL2 interrupt regime this register replaces the memory mapped GICH_EISRn registers. Only 16 system register accessible list registers are provided so only a single EISR register is required.

This register operates exactly as the GICv2 GICH_EISRn registers.

5.9.12 ICH_ELSR_EL2

When SRE is set for the non-secure EL2 interrupt regime this register replaces the memory mapped GICH_ELSRn registers. Only 16 system register accessible list registers are provided so only a single ELSR register is required.

After reset, the bits corresponding to implemented list registers will be one. All other bits are RES0.

This register operates exactly as the GICv2 GICH_ELSRn registers.

5.10 Virtual CPU Interface Register Changes

When a guest operating system uses system registers (as specified by the pseudocode in section 5.7.6), guest operating system accesses to the system register interface to the CPU interface registers access virtual not physical state as defined in section 4.4.1.

The memory-mapped registers must match the format of the physical CPU interface registers (GICC_*) defined above. This affects the following registers:

- GICV_IAR and GICV_AIAR. When SRE is set for the guest interrupt regime, the memory mapped registers should not be used and equivalent functions are provided by ICC_IAR0_EL1 and ICC_IAR1_EL1 respectively.
- GICV_EOIR and GICV_AEOIR. When SRE is set for the guest interrupt regime, the memory mapped registers should not be used and equivalent functions are provided by ICC_EOIR0_EL1 and ICC_EOIR1_EL1 respectively.
- GICV_HPPIR and GICV_AHPPIR. When SRE is set for the guest interrupt regime, the memory mapped registers should not be used and equivalent functions are provided by ICC_HPPIR0_EL1 and ICC_HPPIR1_EL1 respectively. ***Note***: in GICv3, only enabled interrupt groups affect GICV_HPPIR and GICV_AHPPIR.
- GICV_BPR and GICV_ABPR. When SRE is set for the guest interrupt regime, the memory mapped registers should not be used and equivalent functions are provided by ICC_BPR0_EL1 and ICC_BPR1_EL1 respectively.
- GICV_PMR. When SRE is set for the guest interrupt regime, the memory mapped register should not be used and equivalent function is provided by ICC_PMR_EL1.
- GICV_DIR. When SRE is set for the guest interrupt regime, the memory mapped register should not be used and equivalent function is provided by ICC_DIR_EL1.
- GICV_CTLR. When SRE is set for the guest interrupt regime, the memory mapped register should not be used and equivalent function is provided by ICC_CTLR_EL1. See the definition of virtual accesses to ICC_CTLR_EL1 in section 5.7.32.
- GICV_STATUSR. When SRE is set for the guest interrupt regime, this optional memory mapped register is not updated and the equivalent function is provided by appropriate traps and exceptions.

The new bit field definitions are similarly modified to match the CPU interface registers when ICH_HCR_EL2.VARE is set:

- Bits [31:0]. Interrupt ID.
    - ***Note***: interrupts 1020 – 1023 are reserved and convey additional information such as a spurious interrupts.
    - ***Note***: it is implementation defined how many bits are implemented, though at least 16 bits must be implemented. Unimplemented bits are RES0. The number of implemented bits can be discovered from ICH_VTR_EL2.IDbits.

To enable use of 64kB pages, the GICV_* memory map must ensure that:

- The base address of the GICV_* registers is 64kB aligned.
- An alias of the GICV_* registers is provided starting at offset 0xF000 from the start of this page such that a second copy of GICV_DIR exists at the start of the next 64kB page.

This provides support for both 4kB and 64kB pages.

Note: where a GICC_* register is banked by security, the GICV_* registers matches the format for secure accesses.

5.10.1 GICV_STATUSR

This optional 32 bit register has been added to GICv3 to provide software with a mechanism to detect accesses to reserved locations, writes to read-only locations and reads of write-only locations.

If not implemented this register is RAZ / WI.

The register is at offset 0x002C in the virtual CPU interface register map. Its format is shown below:

- Bits [31:4]. Reserved. RES0.
- Bit [3]. WROD. This bit is set if write to a read-only location is detected. Software must write a one to this it to clear it.
- Bit [2]. RWOD. This bit is set if read to a write-only location is detected. Software must write a one to this bit to clear it.
- Bit [1]. WRD. This bit is set if a write to a reserved location is detected. Software must write a one to this bit to clear it.
- Bit [0]. RRD. This bit is set if a read to a reserved location is detected. Software must write a one to this bit to clear it.

If SRE is set for the guest interrupt regime, memory-mapped GICV_STATUSR is not updated and the equivalent functionality is provided for system register accesses by appropriate traps and exceptions.

Note: if GICV_STATUSR is implemented by a GIC, GICC_STATUSR must also be implemented. See section 5.6.16.

5.10.2 GICV_EOIR

When SRE is zero for an interrupt regime, this register behaves as defined for GICv2 except when this register is written, the checks specified in section 5.7.10 must be performed with parameter `lpiAllowed` set to `true`.

In GICv2, there were three unpredictable cases when writing to GICV_EOIR:

- If highest active priority is Group 1 and the identified interrupt is in the List Registers and it matches the highest active priority, GICv3 implementations must perform the deactivate operation (that is, the CPU interface must send an appropriate Deactivate packet if EOI mode is zero).
- If the identified interrupt is in the List Registers and the HW bit is one and the “Physical ID” field indicates the interrupt to be deactivated is an SGI (i.e. the value of “Physical ID” is between 0 and 15). GICv3 implementations must perform the deactivate operation (that is, the CPU interface must send an appropriate Deactivate packet if EOI mode is zero). ***Note***: this means that GICv3 Distributor implementations must always ensure only a single GICv2 SGI is active for a processor such that no knowledge of the Source CPU ID is required.
- If the identified interrupt is in the List Registers and the HW is one and the value of the “Physical ID” field is between 1020 and 1023, indicating a special purpose ID. GICv3 implementations must not perform a deactivate operation but must still change the state of the List Register as appropriate. When EL2 is using system registers and ICH_VTR_EL2.SEIS is one, an implementation defined SEI might be generated.

When SRE is set for an interrupt regime, memory-mapped GICV_EOIR must not be used for that interrupt regime, and system register ICC_EOIR0_EL1 provides the equivalent functions.

5.10.3 GICV_AEOIR

When SRE is zero for an interrupt regime, this register behaves as defined for GICv2 except when this register is written, the checks specified in section 5.7.10 must be performed with parameter `lpiAllowed` set to `true`.

In GICv2, there were three unpredictable cases when writing to GICV_AEOIR:

- If highest active priority is Group 0 and the identified interrupt is in the List Registers and it matches the highest active priority. When EL2 is using system registers and ICH_VTR_EL2.SEIS is one, an implementation defined SEI might be generated, otherwise GICv3 implementations must ignore such writes.

- If the identified interrupt is in the List Registers and the HW bit is one and the “Physical ID” field indicates the interrupt to be deactivated is an SGI (i.e. the value of “Physical ID” is between 0 and 15). GICv3 implementations must perform the deactivate operation (that is, the CPU interface must send an appropriate Deactivate packet if EOI mode is zero). ***Note***: this means that GICv3 Distributor implementations must always ensure only a single GICv2 SGI is active for a processor such that no knowledge of the Source CPU ID is required.
- If the identified interrupt is in the List Registers and the HW bit is one and the “Physical ID” field is between 1020 and 1023, indicating a special purpose ID. GICv3 implementations must not perform a deactivate operation but must still change the state of the List Register as appropriate. When EL2 is using system registers and ICH_VTR_EL2.SEIS is one, an implementation defined SEI might be generated.

When SRE is set for an interrupt regime, memory-mapped GICV_AEOIR must not be used for that interrupt regime, and system register ICC_EOIR1_EL1 provides the equivalent function.

5.10.4 GICV_DIR

In GICv2, there were four unpredictable cases when writing to GICV_DIR:

- When EOImode == ‘0’. When EL2 is using system registers and ICH_VTR_EL2.SEIS is one, an implementation defined SEI might be generated, otherwise GICv3 implementations must ignore such writes.
- When EOImode == ‘1’ but the interrupt is present in the List Registers but is not “active”. When EL2 is using system registers and ICH_VTR_EL2.SEIS is one, an implementation defined SEI might be generated; otherwise GICv3 implementations must ignore such writes.
- Writes with a special ID (i.e. between 1020 and 1023 inclusive). GICv3 implementations must ignore such writes.
- If the identified interrupt is in the List Registers and the HW bit is one and the “Physical ID” field indicates the interrupt to be deactivated is an SGI (i.e. the value of “Physical ID” is between 0 and 15). GICv3 implementations must perform the deactivate operation. ***Note***: this means that GICv3 Distributor implementations must always ensure only a single GICv2 SGI is active for a processor such that no knowledge of the Source CPU ID is required.

When SRE is zero for the non-secure EL1 interrupt regime, this register behaves as defined for GICv2, except the checks specified in section 5.7.10 must be performed with parameter `lpiAllowed` set to `false`.

If SRE is set for the non-secure EL1 interrupt regime, then GICV_DIR must not be used, and system register ICC_DIR_EL1 provides the equivalent function.

5.10.5 GICV_PMR

This register behaves exactly as GICv2, except an implementation might choose to implement more than 5 bits of priority. That is, the priority field covers bits [7:0] and, if an implementation implements fewer than 256 priorities, some bits are RAZ / WI as defined in section 4.4.2 of [2].

5.10.6 GICV_APRn

This register behaves exactly as GICv2, except the data values are no longer implementation defined:

- When SRE is zero for EL2, this register accesses GICH_APRn and all active priorities for memory mapped guests are held in GICH_APRn, regardless of interrupt group.
- When SRE is one for EL2, this register accesses ICH_AP1Rn_EL2 and all active priorities for memory mapped guests are held in ICH_AP1Rn_EL2, regardless of interrupt group.

5.11 Discovery and Identification

It is expected that discovery and identification of systems will be achieved using a combination of the following registers:

- **Processor Affinity identification**. It is expected that the affinity values in the CPU MPIDR register uniquely identify the processor within the system.
- **Memory-mapped GIC Version identification**. In systems using GICv3, it is expected that over time systems may exist with different Re-distributor and CPU interface versions. These versions may be identified as below:
    - Distributor version from GICD_PIDR2.ArchRev and GICD_IIDR.Revision.
    - Re-distributor version from GICR_PIDR2.ArchRev and GICR_IIDR.Revision.
    - When system registers are not used, CPU interface version from GICC_IIDR.ArchRev and GICC_IIDR.Revision
- **System-register GIC Version identification**. Support for GIC system registers can be discovered from:
    - For AArch64, from ID_AA64PRF0_EL1[27:24]
    - For AArch32, from ID_PFR1[31:28].
- **Affinity Hierarchy**. The affinity hierarchy may be constructed by reading all the GICR_TYPER registers.
    - This also provides the mapping between the Processor Number and the affinity of a processor.
    - Support for non-zero values of Affinity 3 is discoverable.
        - For the Distributor, from GICD_TYPER.A3V
        - For the physical CPU interface, from ICC_CTLR_EL1/2.A3V
        - For the virtual CPU interface, from ICH_VTR_EL2.A3V
- **Device Identifier Size.** The maximum supported size of device identifiers is discovered from:
    - From GITS_TYPER.Devbits.
- **Interrupt Identifier Size**. The maximum supported size of interrupt identifiers is discovered from:
    - For the Distributor, from GICD_TYPER.IDbits
    - For the Interrupt Translation Service, from GITS_TYPER.IDbits
    - For the physical CPU interface, from ICC_CTLR_EL1/3.IDbits
    - For the virtual CPU interface, from ICH_VTR_EL2.IDbits
    - ***Note***: software must ensure that no interrupt identifier exceeds the minimum number of bits supported.
- **Feature Support**. The GICD_TYPER register identities the support level for the following features:
    - System Registers
    - Locality-specific Peripheral Interrupts
    - Direct Virtual Interrupts
    - Interrupt Translation.

5.11.1 Implementations with Mixed Interrupt Identifier Sizes

Implementations might choose to implement different interrupt identifier sizes for different parts of the GIC subject to the following rules:

- Processors may implement either 16 or 24 bits of interrupt identifier. ***Note***: a system might include a mixture of processors which support 16 bits of interrupt identifier and processors that support 24 bits.
- The Distributor (GICD_* registers) and all re-distributors (GICR_* registers) must all implement the same number of interrupt identifier bits.

- In systems that include one or more ITS (i.e. LPIs are supported) the Distributor and all re-distributors must implement at least 14 bits of interrupt identifier; the number of bits implemented must not exceed the minimum number implemented any processor in the system. ***Note***: because interrupts might target any processor, any processor must be able to receive the maximum ID that can be sent by a re-distributor. Hence, the interrupt identifier size supported by re-distributors cannot exceed the minimum supported by any processor.
- In systems that do not include an ITS (i.e. LPIs are not supported), the Distributor and all re-distributors must implement at least 5 bits of interrupt identifier and may not implement more than 10 bits of interrupt identifier.
- In systems that include one or more ITS, an ITS may implement any value up to and including the number of bits supported by the Distributor and re-distributors down to a minimum of 14 bits (the minimum number required for LPI support).

5.12 ITS Registers

This section specifies the format of the Interrupt Translation Service registers.

5.12.1 ITS Address Map

The ITS requires two separate 64kB pages starting from an implementation defined ITS_Base. This base address must be aligned to a 64kB boundary – i.e. bits [15:0] must be zero. This can be considered in two parts:

- Control registers. Located in a single page at ITS_Base + 0x000000. ***Note***: the registers in this page are expected to be used by processor software for configuration of the ITS and should never be mapped for IO devices.
- Interrupt translation space. Located in a single page at ITS_Base + 0x010000. ***Note***: the register in this page is expected to be targeted by device interrupt request and it is not expected that processor software will access the registers in this page (see section 4.9.5).

5.12.2 ITS Control Register Address Map

The address map of the ITS Control registers is shown below:

Register Name	Offset	Notes
GITS_CTLR	0x0000	
GITS_IIDR	0x0004	Version numbers
GITS_TYPER	0x0008 – 0x000C	Indicates the supported ITS features
Reserved	0x000C – 0x007C	
IMP DEF	0x0020 – 0x003C	Implementation Defined
Reserved	0x0040 – 0x007C	
GITS_CBASER	0x0080 – 0x0084	The command queue descriptor
GITS_CWRITER	0x0088 – 0x008C	The command queue write pointer
GITS_CREADR	0x0090 – 0x0094	The command queue read pointer
Reserved	0x0098 – 0x009C	
Reserved	0x00A0 – 0x000FC	
GITS_BASERn	0x0100 – 0x013C	A set of up to eight table descriptors that must be provisioned by software.
Reserved	0x0140 – 0xBFFC	
IMP DEF	0xC000 – 0xFFCC	
Identification Registers	0xFFD0 – 0xFFFC	Reserved for 64kB page identification registers. This includes registers GITS_PIDR{0..7} which are architecturally defined . See section 5.12.13.

Table 32: ITS Control Register Address Map

5.12.3 ITS Translation Register Address Map

The address map of the ITS Translation registers is shown below:

Register Name	Offset	Notes
Reserved	0x0000 – 0x003C	
GITS_TRANSLATER	0x0040	
Reserved	0x0044 – 0xFFFC	

Table 33: ITS Translation Register Address Map

5.12.4 GITS_TRANSLATER

This 32 bit register is write-only. The value written to this register specifies an interrupt identifier to be translated for the requesting device. The format of the value written is shown below:

- Bits [31:0]. Interrupt ID. The ID of interrupt to be translated for the requesting device. ***Note***: the number of interrupt identifier bits is defined by GITS_TYPER.IDbits. Non-zero identifier bits outside this range are ignored.

Implementations must ensure that a unique “device identifier” is provided for each requesting device and this is presented to the ITS on writes to this register. This “device identifier” is used to index a “device table” (see section 4.9.5) that maps the incoming device identifier to an interrupt translation table for that device.

For example, it is expected that the 16 bit “Requester ID” from a PCIe Root Complex will be presented to an ITS.

If GITS_CTLR.Enable is zero, the write has no effect (see section 5.12.5).

Note: the “device identifier” presented to the ITS on writes to this register corresponds to the “device” field in the ITS commands defined below.

Note: writes to this register with a “device identifier” that has not been mapped (see MAPD in section 5.13.11 below) will be ignored.

Note: writes to this register with a “device identifier” that exceed the supported device identifier size will be ignored (see section 4.9.5).

Note: this register is provided to enable the generation (and translation) of message-based interrupts from devices (e.g. MSI or MSI-X writes from PCIe devices). The register is at the same offset as GICD_SETSPI_NSR in the Distributor and GICR_SETLPIR in the re-distributor to allow virtualization of guest operating systems that directly program devices simply by ensuring the address programmed by the guest can be translated by an SMMU to target GITS_TRANSLATER.

Note: 16 bit access to bits [15:0] of this register must be supported. When written by a 16 bit transaction, bits [31:16] are written as zero.

5.12.5 GITS_CTLR

This 32 bit register controls the behaviour of the ITS. Its format is shown below:

- Bits [31]. Quiescent. Resets to one. This bit indicates whether the ITS has completed all operations following a write of “Enable” to zero.
    - 0b0. The ITS is not quiescent
    - 0b1. The ITS is quiescent, has no translations in progress and has completed all operations required to make any mapping data consistent with external memory and may be powered off. ***Note***: in Distributed implementations, the ITS must also have forwarded any required operations to the re-distributors and received confirmation that they have reached the appropriate re-distributor.
- Bits [30:8]. Reserved. RES0.
- Bits [7:4]. ITS number. This field is RES0 in GICv3.
    - In GICv4, if GITS_TYPER.Distributed is one, this indicates which set of GICR_VMOVLPIRn registers will be used by the ITS
    - In GICv4, if GITS_TYPER.Distributed is zero, this field is RES0.
- Bits [3:2]. Reserved. RES0.
- Bits [1]. Virtual LPI Enable. Resets to zero. In GICv3, this bit is RES0. In GICv4, this bit means:
    - 0b0. The ITS will ignore any virtual commands and will not cause direct injection of virtual LPIs.
    - 0b1. The ITS will handle virtual commands and may cause direct injection of virtual LPIs.
    - ***Note***: this bit is required to support distributed implementations with a GICv4 ITS and a GICv3 Distributor.
    - ***Note***: a GICv4 implementation might choose to make this field RES1.
- Bits [0]. Enabled. Resets to zero.
    - 0b0. The ITS is disabled. Writes to the Interrupt Translation Space will be ignored and no further command queue entries will be processed.
    - 0b1. The ITS is enabled. Writes to the Interrupt translation space will result in interrupt translations and the command queue will be processed.
    - If a write to this register changes “Enabled” from one to zero, the ITS must ensure that any caches containing mapping data must be made consistent with external memory and “Quiescent” must read as zero until this has been completed. See section 4.10.9.

5.12.6 GITS_TYPER

This 64 bit read-only register specifies the features supported by the ITS:

- Bits [63:32]. Reserved.
- Bits [31:24]. HCC. Hardware Collection Count. The number of collections supported by the ITS without provisioning of external memory.
    - If this field is non-zero, collections in the range zero to (HCC minus one) are solely maintained in storage within the ITS.
    - ***Note***: when this field is non-zero and an ITS is dynamically powered-off and back on, software must ensure that any hardware collections are re-mapped following power-on.
- Bits [23:20]. Reserved.
- Bits [19]. PTA. Physical Target Addresses supported. See section 4.9.16.
    - 0b0. Target Addresses correspond to linear processor numbers. See section 5.4.8.
    - 0b1. Target Addresses correspond to the base physical address of re-distributors

- Bits [18]. SEIS. Locally generated System Error Interrupts supported.
- Bits [17:13]. Devbits. The number of device identifier bits implemented, minus one.
- Bits [12:8]. IDbits. The number of interrupt identifier bits implemented, minus one.
- Bits [7:4]. ITT Entry size (see section 5.13.11). This field is read-only and specifies the number of byes per entry, minus one.
- Bits [3]. Distributed. Distributed implementation supported.
- Bits [2]. Reserved. RES0.
- Bits [1]. Virtual. Virtual LPIs and Direct injection of Virtual LPIs supported. ***Note***: this field is RES0 in GICv3 implementations.
- Bits [0]. Physical. Physical LPIs supported. This field is RES1.

5.12.7 GITS_IIDR

This 32 bit register is read-only behaves like GICD_IIDR (see section 5.3.2).

5.12.8 The ITS Command Queue

This queue is controlled by the GITS_CBASER, GITS_CWRITER and GITS_CREADR registers defined below. Each command within the queue comprises 32 bytes. This ensures the ITS commands can be extended to support:

- More than 16 bits of Device ID
- More than 48 bits of physical address.

The commands supported are described in section 5.13.

5.12.9 GITS_CBASER

This 64 bit register specifies the base address and size of the ITS command queue. The format of this register is specified below:

- Bits [63]. Valid. Resets to zero.
  - When set to one, indicates that memory has been allocated by software for the command queue
  - When set to zero, no memory has been allocated to the command queue and the ITS discards any writes to the interrupt translation page.
- Bits [62]. Reserved. RES0
- Bits [61:59]. Cacheability. The cacheability attributes of accesses to the table.
  - 0b000. Non-cacheable, non-bufferable.
  - 0b001. Non-cacheable.
  - 0b010. Read-allocate, Write-through.
  - 0b011. Read-allocate, Write-back.
  - 0b100. Write-allocate, Write-through.
  - 0b101. Write-allocate, Write-back.
  - 0b110. Read-allocate, Write-allocate, Write-through.
  - 0b111. Read-allocate, Write-allocate, Write-back.
- Bits [58:48]. Reserved. RES0
- Bits [47:12]. Physical Address. Provides bits [47:12] of the physical address of the memory containing the command queue. Bits [11:0] of the base address of the queue are zero.
- Bits [11:10]. Shareability. The shareability attributes of accesses to the table.
  - 0b00. Accesses are non-shareable.
  - 0b01. Accesses are inner-shareable.
  - 0b10. Accesses are outer-shareable.
  - 0b11. Reserved. Treated as 0b00.
  - In implementations where re-distributors cannot express shareability on the downstream bus system, this field is RAZ/WI.
- Bits [9:8]. Reserved.
- Bits [7:0]. Size. The number of 4kB pages of physical memory provided for the command queue, minus one.

The command queue is a circular buffer and wraps at Physical Address [47:0] + (4096 * (Size + 1)).

Note: when GITS_CBASER is successfully written, the value of GITS_CREADR is set to zero. See section 4.9.25 for details of the ITS initialization sequence.

Note: bits [63:32] and bits [31:0] may be accessed independently.

Note: when GITS_CTLR.Enable is one or GITS_CTLR.Quiescent is zero, this register is read-only. See section 5.12.5.

5.12.10 GITS_CWRITER

This 64 bit register specifies the offset from GITS_CBASER (see section 5.12.9) where software will write the next ITS command.

- Bits [63:20]. Reserved. RES0
- Bits [19:5]. Offset. Provides bits [19:5] of the offset from GITS_CBASER where software will write the next command. Bits [4:0] of the offset are zero.
- Bits [4:0]. Reserved. RES0

The command queue is considered to be empty when GITS_CWRITER is equal to GITS_CREADR.

The command queue is considered to be full when GITS_CWRITER is equal to (GITS_CREADR minus 32), taking wrapping into account.

Each command in the queue comprises 32 bytes. See section 5.13 for details of the commands supported and the format of each command.

Note: See section 4.9.25 for details of the ITS initialization sequence.

Note: bits [63:32] and bits [31:0] may be accessed independently.

5.12.11 GITS_CREADR

This 64 bit register is read-only and specifies the offset from GITS_CBASER (see section 5.12.9) where the ITS will read the next ITS command.

- Bits [63:20]. Reserved. RES0
- Bits [19:5]. Offset. Provides bits [19:5] of the offset from GITS_CBASER where the ITS will read the next command. Bits [4:0] of the offset are zero.
- Bits [4:0]. Reserved. RES0

The command queue is considered to be empty when GITS_CWRITER is equal to GITS_CREADR.

The command queue is considered to be full when GITS_CWRITER is equal to (GITS_CREADR minus 32), taking wrapping into account.

Note: when GITS_CBASER is written, the value of GITS_CREADR is set to zero (see section 5.12.9) . See section 4.9.25 for details of the ITS initialization sequence.

Note: bits [63:32] and bits [31:0] may be accessed independently.

5.12.12 GITS_BASERn

This set of 64 bit registers specify the base address and size of a number of implementation defined tables required by the ITS:

- An implementation can provide up to eight such registers.
- Where a register is not implemented, it is RES0.

The format of each implemented register is as shown below:

- Bits [63]. Valid. Resets to zero. If the Type field is zero this field is RAZ/WI.
    - When set to one, indicates that memory has been allocated by software for the table
    - When set to zero, no memory has been allocated to the table:
        - If the Type field specifies a valid table type other than Interrupt Collections, the ITS discards any writes to the interrupt translation page.
        - If the Type field specifies the Interrupt Collections table and GITS_TYPER.HCC is zero, the ITS discards any writes to the interrupt translation page.
- Bits [62]. Indirect. This field indicates whether an implemented register specifies a single, flat table or a two-level table where the first level contains a list of descriptors. ***Note***: this field is RAZ/WI for implementations that only support flat tables.
    - 0b0. Single Level. The Size field indicates a number of pages used by the ITS to store data associated with each table entry.
    - 0b1. Two Level. The Size field indicates a number of pages which contain an array of 64 bit descriptors to pages that are used to store the data associated with each table entry. Each 64 bit descriptor has the following format:
        - Bits[63]. Valid.
        - Bits [62:48]. Reserved. RES0
        - Bits[47:N]. Physical Address.
        - Bits [N-1:0]. Reserved. RES0.
        - Where N is the number of bits required to specify the page size.
        - ***Note***: software must ensure that each pointer in the first level table specifies a unique physical address otherwise the effects are unpredictable.
- Bits [61:59]. Cacheability. The cacheability attributes of accesses to the table. If the Type field is zero this field is RAZ/WI.
    - 0b000. Non-cacheable, non-bufferable.
    - 0b001. Non-cacheable.
    - 0b010. Read-allocate, Write-through.
    - 0b011. Read-allocate, Write-back.
    - 0b100. Write-allocate, Write-through.
    - 0b101. Write-allocate, Write-back.
    - 0b110. Read-allocate, Write-allocate, Write-through.
    - 0b111. Read-allocate, Write-allocate, Write-back.
- Bits [58:56]. Type. This field is read-only and specifies the type of entity that requires entries in the associated table. The field may have the following values:
    - 0x0. Unimplemented. This register does not correspond to an ITS table and requires no memory.
    - 0x1. Devices. This register corresponds to a table that scales according to the number of devices serviced by the ITS and requires (Entry-size * number-of-devices) bytes of memory.
    - 0x2. Virtual Processors. This register corresponds to a table that scales according to the number of virtual processors in the system and requires (Entry-size * number-of-processors) bytes of memory.

    - 0x3. Physical Processors. This register corresponds to a table that scales according to the number of physical processors in the system and requires (Entry-size * number-of-processors) bytes of memory.
    - 0x4. Interrupt Collections. This register corresponds to a table that scales according to the number of interrupt collections in the system and requires (Entry-size * number-of-collections) bytes of memory. ***Note***: at least (number-of-physical-processors + 1) entries must be supported.
    - 0x5 – 0x7. Reserved. Treated as 0x0.
- Bits [55:48]. Entry size. This field is read-only and specifies the number of bytes per entry, minus one.
- Bits [47:12]. Physical Address. If the Type field is zero this field is RAZ/WI.
    - This field provided bits [47:12] of the base physical address of the table. Bits [11:0] of the base physical address are zero.
- Bits [11:10]. Shareability. The shareability attributes of accesses to the table. If the Type field is zero this field is RAZ/WI.
    - 0b00. Accesses are non-shareable.
    - 0b01. Accesses are inner-shareable.
    - 0b10. Accesses are outer-shareable.
    - 0b11. Reserved. Treated as 0b00.
    - In implementations where re-distributors cannot express shareability on the downstream bus system, this field is RAZ/WI.
- Bits [9:8]. Page Size. The size of the pages used for this table:
    - 0b00. 4kB pages.
    - 0b01. 16kB pages.
    - 0b10. 64kB pages.
    - 0b11. Reserved. Treated as 64kB pages.
    - ***Note***: this field might be read-only if an implementation only supports a single, fixed page size.
- Bits [7:0]. Size. The number of pages (of the size indicated by the Page Size field) of memory allocated to the table, minus one. If the Type field is zero this field is RAZ/WI.

Note: bits [63:32] and bits [31:0] may be accessed independently.

Note: when GITS_CTLR.Enable is one or GITS_CTLR.Quiescent is zero, this register is read-only. See section 5.12.5.

5.12.13 Peripheral Identification Registers

The table below specified the values of the ITS Peripheral Identification Registers:

Offset	Register	Description
0xFFF0	GITS_CIDR0	• Component ID0. Value 0x0D.
0xFFF4	GITS_CIDR1	• Component ID1. Value 0xF0.
0xFFF8	GITS_CIDR2	• Component ID2. Value 0x05.
0xFFFC	GITS_CIDR3	• Component ID3. Value 0xB1.
0xFFE0	GITS_PIDR0	• Bits [7:0]. Bits [7:0] of the ARM-defined DevID field. This field is 0x94 in ARM implementations of a GICv3 or later ITS.
0xFFE4	GITS_PIDR1	• Bits [7:4]. Bits[3:0] of the ARM defined JEP identity field. This field is 0xB in ARM implementations. • Bits [3:0]. Bits [11:8] of the ARM-defined DevID field. This field is 0x4 in ARM implementations of a GICv3 or later ITS.

0xFFE8	GITS_PIDR2	• Bits [7:4]. ArchRev. This field has the following implementation defined values:
		  ○ 0x1. GICv1
		  ○ 0x2. GICv2
		  ○ 0x3. GICv3.
		  ○ 0x4. GICv4.
		  ○ 0x5 – 0xf. Reserved
		• Bit [3]. UsesJEPcode. This field is 0b1 in ARM implementations.
		• Bits [2:0]. Bits[6:4] of the ARM defined JEP identity field. This field is 0x3 in ARM implementations.

0xFFEC	GITS_PIDR3	• Bits [7:4]. RevAnd. Manufacturer defined revision number. Normally zero. • Bits [3:0]. Customer modified.
0XFFD0	GITS_PIDR4	• Bits [7:4]. RES0. • Bits [3:0]. ContinuationCode. This field is 0x4 in ARM implementations.
0xFFD4	GITS_PIDR5	• Bits [7:0]. RES0.
0xFFD8	GITS_PIDR6	• Bits [7:0]. RES0.
0xFFDC	GITS_PIDR7	• Bits [7:0]. RES0.

Table 34: ITS Peripheral Identification Registers

5.13 ITS Commands

This section specifies the commands supported by the ITS.

5.13.1 Supported Commands for GICv3

The table below specifies the commands supported by the ITS for GICv3:

CMD	Syntax	Description
0x01	MOVI device, ID, collection	This command specifies that the interrupt with identifier “ID” from “device” is now a member of the interrupt collection specified by “collection”
0x03	INT device, ID	This command specifies that interrupt “ID” must be generated for the device.
0x04	CLEAR device, ID	This command specifies that the pending state of interrupt “ID” for the specified device must be cleared.
0x05	SYNC Target Address	This command specifies that all actions for the specified re-distributor must be completed.
0x08	MAPD device, ITT descriptor	This command describes the Interrupt Translation Table to be used for a specified device. The descriptor comprises a 256 byte aligned physical address and the number of bits of interrupt ID supported.
0x09	MAPC collection, Target Address	This command specifies where interrupts belonging to the collection will be forwarded. This is specified by Target Address (the base address of a re-distributor)
0x0A	MAPVI device, ID, pID, collection	This command specifies that the interrupt with identifier “ID” from “device” will generate physical interrupts with identifier “pID” which is a member of the interrupt collection specified by “collection”
0x0B	MAPI device, ID, collection	This command specifies that the interrupt with identifier “ID” from “device” is a member of the interrupt collection specified by “collection”
0x0C	INV device, ID	This command specifies that the ITS must ensure that the re-distributor that currently owns interrupt “ID” for the device must ensure any caching associated with this index is consistent with the configuration and pending tables held in memory.
0x0D	INVALL collection	This command specifies that the ITS must ensure any caching associated with the specified interrupt collection is consistent with the configuration and pending tables held in memory for all re-distributors.
0x0E	MOVALL Target Address 1, Target Address 2	This command specifies that all pending interrupts must be moved from the pending table of the re-distributor specified by Target Address 1 to the pending table of the re-distributor specified by Target Address 2.
0x0F	DISCARD device, ID	This command specifies that incoming requests with an identifier of “ID” will be silently discarded.

Table 35: ITS commands for GICv3

Each command is described in the sections that follow.

5.13.2 Additional Supported Commands for GICv4

In addition to the commands specified in section 5.13.1 above, GICv4 adds the following commands:

CMD	Syntax	Description
0x21	VMOVI device, ID, VCPU	This command specifies that interrupts with identifier “ID” for the device must be sent to the virtual processor specified by VCPU.
0x22	VMOVP VCPU, Target Address	This command specifies that interrupts for the virtual processor specified by “VCPU” must be sent to the re-distributor specified by Target Address.
0x25	VSYNC VCPU	This command specifies that all actions for the virtual processor prior to this command must be completed.
0x29	VMAPP VCPU, Target Address, VPT descriptor	This command specifies that the virtual processor specified by VCPU is mapped to the re-distributor specified by Target Address and uses a Virtual Pending Table described by the VPT descriptor. The VPT descriptor comprises a 64kB aligned physical address and the number of bits of interrupt ID supported.
0x2A	VMAPVI device, ID, vID, pID, VCPU	This command specifies that virtual interrupts with identifier “vID” will be generated on the virtual processor specified by VCPU. If the virtual processor specified by VCPU is not resident then the physical interrupt “pID” that will be generated on the re-distributor that VCPU is mapped to.
0x2B	VMAPI device, ID, pID, VCPU	This command specifies that virtual interrupts with identifier “ID” will be generated on the virtual processor specified by VCPU. If the virtual processor specified by VCPU is not resident then the physical interrupt “pID” that will be generated on the re-distributor that VCPU is mapped to.
0x2D	VINVALL VCPU	This command specifies that the ITS must ensure any caching associated with the virtual processor specified by VCPU is consistent with the configuration and pending tables held in memory for all re-distributors.

Table 36: Additional ITS commands for GICv4

Note: a separate VINV command is not required as the physical INV command performs the same function, regardless of whether the interrupt is mapped as physical or virtual.

5.13.3 Identifier and Address Fields

The commands below include several fields with implementation defined sizes. These are:

- Device Identifiers. Up to and including 32 bits. Unimplemented bits are RES0. The number of implemented buts is discoverable from GITS_TYPER.Devbits (see section 5.12.6).
- Interrupt Identifiers. Up to and including 32 bits. Unimplemented bits are RES0. The number of implemented buts is discoverable from GITS_TYPER.IDbits (see section 5.12.6).
- Collection Identifiers. Up to and including 16 bits. Unimplemented bits are RES0.
- Target Addresses. Up to and including 48 bits. Unimplemented bits are RES0. ***Note***: Target Addresses must be 64kB aligned. Hence bits [15:0] must be zero and only bits [47:16] of Target Address need to be provided in commands that include a Target Address field.

5.13.4 ITS Command Errors

If the ITS detects an error in the data supplied to a command, it must:

- Ignore the command. That is, it must perform no actions that alter the handling of interrupts.
- Increment GITS_CREADR (wrapping if necessary; see section 5.12.11) to point to the next command
- If GITS_TYPER.SEIS is one (see section 5.12.6), generate a System Error. ***Note***: it is implementation defined how this system error is recorded and how it is reported to a processor.
- If the command queue is not empty, it must continue processing commands.

5.13.5 ITS Command Error Encodings

The pseudocode below uses the commend error encodings below. In an implementation that supports reporting of system errors, it is implementation defined how these encodings are recorded and reported to software.

Encoding	Mnemonic	Description	Section
0x01_0801	MAPD_DEVICE_OOR	“Device ID” was out of range for a MAPD	5.13.11
0x01_0802	MAPD_ITTSIZE_OOR		5.13.11
0x01_0902	MAPC_PROCNUM_OOR	Linear Processor Number was out of range for a MAPC	5.13.12
0x01_0903	MAPC_COLLECTION_OOR		5.13.12
0x01_0b01	MAPI_DEVICE_OOR		5.13.13
0x01_0b03	MAPI_COLLECTION_OOR		5.13.13
0x01_0b04	MAPI_UNMAPPED_DEVICE		5.13.13
0x01_0b05	MAPI_ID_OOR		5.13.13
0x01_0a01	MAPVI_DEVICE_OOR		5.13.14
0x01_0a03	MAPVI_COLLECTION_OOR		5.13.14
0x01_0a04	MAPVI_UNMAPPED_DEVICE		5.13.14
0x01_0a05	MAPVI_ID_OOR		5.13.14
0x01_0a06	MAPVI_PHYSICALID_OOR		5.13.14
0x01_0101	MOVI_DEVICE_OOR		5.13.15
0x01_0103	MOVI_COLLECTION_OOR		5.13.15
0x01_0104	MOVI_UNMAPPED_DEVICE		5.13.15
0x01_0105	MOVI_ID_OOR		5.13.15
0x01_0107	MOVI_UNMAPPED_INTERRUPT		5.13.15
0x01_0108	MOVI_ID_IS_VIRTUAL		5.13.15
0x01_0109	MOVI_UNMAPPED_COLLECTION		5.13.15

0x01_0f01	DISCARD_DEVICE_OOR		5.13.16
0x01_0f04	DISCARD_UNMAPPED_DEVICE		5.13.16
0x01_0f05	DISCARD_ID_OOR		5.13.16
0x01_0f10	DISCARD_ITE_INVALID		5.13.16
0x01_0c01	INV_DEVICE_OOR		5.13.17
0x01_0c04	INV_UNMAPPED_DEVICE		5.13.17
0x01_0c05	INV_ID_OOR		5.13.17
0x01_0c07	INV_UNMAPPED_INTERRUPT		5.13.17
0x01_0c10	INV_ITE_INVALID		5.13.17
0x01_0d03	INVALL_COLLECTION_OOR		5.13.19
0x01_0d09	INVALL_UNMAPPED_COLLECTION		5.13.19
0x01_0e01	MOVALL_PROCNUM_OOR		5.13.18
0x01_0301	INT_DEVICE_OOR		5.13.20
0x01_0304	INT_UNMAPPED_DEVICE		5.13.20
0x01_0305	INT_ID_OOR		5.13.20
0x01_0307	INT_UNMAPPED_INTERRUPT		5.13.20
0x01_0310	INT_ITE_INVALID		5.13.20
0x01_0501	CLEAR_DEVICE_OOR		5.13.21
0x01_0504	CLEAR_UNMAPPED_DEVICE		5.13.21
0x01_0505	CLEAR_ID_OOR		5.13.21
0x01_0507	CLEAR_UNMAPPED_INTERRUPT		5.13.21
0x01_0510	CLEAR_ITE_INVALID		5.13.21
0x01_2911	VMAPP_VCPU_OOR		5.13.23
0x01_2912	VMAPP_VPTSIZE_OOR		5.13.23
0x01_2913	VMAPP_PROCNUM_OOR		5.13.23
0x01_2b01	VMAPI_DEVICE_OOR		5.13.24
0x01_2b11	VMAPI_VCPU_OOR		5.13.24
0x01_2b04	VMAPI_UNMAPPED_DEVICE		5.13.24
0x01_2b05	VMAPI_ID_OOR		5.13.24

0x01_2b06	VMAPI_PHYSICALID_OOR		5.13.24
0x01_2a01	VMAPVI_DEVICE_OOR		5.13.25
0x01_2a11	VMAPVI_VCPU_OOR		5.13.25
0x01_2a04	VMAPVI_UNMAPPED_DEVICE		5.13.25
0x01_2a05	VMAPVI_ID_OOR		5.13.25
0x01_2a13	VMAPVI_VIRTUALID_OOR		5.13.25
0x01_2a06	VMAPVI_PHYSICALID_OOR		5.13.25
0x01_2d11	VINVALL_VCPU_OOR		5.13.26
0x01_2d14	VINVALL_VCPU_INVALID		5.13.26
0x01_2511	VSYNC_VCPU_OOR		5.13.27
0x01_2514	VSYNC_VCPU_INVALID		5.13.27
0x01_2211	VMOVP_VCPU_OOR		5.13.28
0x01_2214	VMOVP_VCPU_INVALID		5.13.28
0x01_2101	VMOVI_DEVICE_OOR		5.13.29
0x01_2103	VMOVI_COLLECTION_OOR		5.13.29
0x01_2105	VMOVI_ID_OOR		5.13.29
0x01_2107	VMOVI_UNMAPPED_INTERRUPT		5.13.29
0x01_2115	VMOVI_ID_IS_PHYSICAL		5.13.29
0x01_2116	VMOVI_ITEVCPU_INVALID		5.13.29
0x01_2117	VMOVI_CMDVCPU_INVALID		5.13.29

Table 37: ITS Command Error Encodings

5.13.6 ITS Helper Functions

The following “helper” functions are used in the ITS pseudo-code that follows:

```
bool DeviceOutOfRange((32)bit device);     // Returns true if the value supplied
                                           // has bits above the implemented range

bool CollectionOutOfRange((32)bit collection); // Returns true if the value supplied
                                           // has bits above the implemented range

bool ProcessorNumberOutOfRange((32)bit number); // Returns true if the value supplied
                                           // is not in the IMP DEF set of supported
                                           // processors

bool VCPUOutOfRange((32)bit vcpu);         // Returns true if the value supplied
                                           // has bits above the implemented range

bool SizeOutOfRange((8)bit ITT_size);      // Returns true if the value supplied
                                           // exceeds the maximum allowed by
                                           // GITS_TYPER.IDbits

bool IdOutOfRange((32)bit ID, (8)bit ITT_size); // Returns true if the value supplied
                                           // has bits above the implemented size
                                           // or above the ITT_size

bool LPIOutOfRange((32)bit ID);            // Returns true if the value supplied
                                           // is larger than that permitted by
                                           // GITS_TYPER.IDbits or not in the LPI
                                           // range (i.e. if it is less than 8192).

bool SpeculativeCachingSupported();        // Returns true if the redistributors support
                                           // speculative caching of LPI configuration data

void IncrementReadPointer();               // Increments GITS_CREADR, wrapping if
                                           // appropriate

DTE ReadDeviceTable(int index);            // Reads a device table entry from
                                           // memory
void WriteDeviceTable(int index, DTE dte); // Writes a device table entry to memory

CTE ReadCollectionTable(int index);        // Reads a collection table entry from
                                           // memory
void WriteCollectionTable(int index, CTE cte); // Writes a collection table entry
                                           // to memory

ITE ReadTranslationTable(Address base, int index); // Reads an ITT table entry from
                                           // memory
void WriteTranslationTable(Address base, int index, ITE cte); // Writes an ITT table entry
                                           // to memory

VTE ReadVCPUTable(int index);              // Reads a VCPU table entry from memory
void WriteVCPUTable(int index, VTE vte);   // Writes a VCPU table entry to memory

//
// Terminology:
// - "Interrupt Translation" an action that causes the ITS to attempt to set a
//    particular pending bit in a particular table.
```



```
// - "Pending Interrupt" a particular pending bit is set in a particular table
//
// The pseudocode below makes the following assumptions:
// 1. Each "perform_*" function must be performed as an atomic operation. In
//    reality, it is expected that operations might proceed in parallel, but
//    implementations must ensure that the observed behaviour is consistent with
//    that from a strictly atomic implementation.
// 2. Where the pseudocode issues a sequence of read and write operations to a
//    particular re-distributor, these operations must be performed in the order
//    they were generated.
// 3. Where the pseudocode issues a write to a particular re-distributor, operation
//    does not need to wait for completion of the write.  For example, a call
//    to WriteGICR_SETLPIR( rd1, ... ) will generate a write to "rd1" that will
//    eventually become visible to "rd1" however the function may return before
//    this write has become visible to "rd1".  That is, writes may be buffered.
// 4. Where the pseudocode issues writes to memory to update a table, operation
//    does not need to wait for completion of these writes.  Unlike for 3), a write
//    to memory might never become visible to an external observer.  However,
//    the effect of any such writes must be observed by any subsequent ITS
//    operations including handling of interrupt translations. There are no ordering
//    rules other than the standard rule that memory must appear as if writes
//    to each location occurred in program order.
//    That is, table data held in memory may be cached and any such caches are
//    updated by table write operations.
// 5. Because each "perform_*" function is performed as an atomic operation, any
//    new interrupt translation that occurs after the function must be subject to
//    effects of that function.  For example, after a successful "perform_mapi"
//    an interrupt translation for the newly mapped interrupt will be generated with
//    the new mapping.
// 6. Because each "perform_*" function is performed as an atomic operation, an
//    interrupt translation that occurs before the function must NOT be subject to
//    any effects of that function.
// 7. The effects on caching of re-distributor configuration and pending tables
//    are specified explicitly in the pseudocode.
// 8. During the period when an interrupt is pending but has not been acknowledged
//    by software (through the IAR), it MIGHT be affected as below:
//    a. If any part of its translation (DTE/ITE/CTE) is updated by a subsequent
//       "perform_*" function, it MIGHT use the new translation.
//       For example, INT X followed by a MAPI X might or might not result in the
//       movement of the pending bit set by the INT X command.
//    b. If there are other ITE translations that produce the same physical LPI ID,
//       it MIGHT use one of these other ITE translations. For example, if INT X and
//       INT Y would both generate the same LPI ID then either of these interrupts might
//       be sent to the collection of the other, even if only one is pending.
// NOTE: there is a notional point of synchronisation between command processing
//    and the handling of interrupt requests and "before" and "after" are defined
//    relative to this point.
//
CommandError((32)bit syndrome)
    if (GITS_TYPER.SEIS) then
        // The ITS supports generation of system errors
        Generate_SEI( syndrome );
    IncrementReadPointer();
```


5.13.7 Invalidate Interrupt Translation Table Entry

The pseudo-code below shows the actions that must be performed to invalidate the caches associated with an interrupt in a distributed implementation:

```
boolean invalidateByITE( ITE ite )
    if ( ite.Type == physical_interrupt ) then
        CTE cte = ReadCollectionTable( ite.Collection );

        if ( ! cte.Valid ) then
            return false;

        Address rd1 = cte.TargetAddress;

        WriteGICR_INVLPIR( rd1, ite.OutputID );
    else
        VTE vte = ReadVCPUTable( ite.VCPU );

        if ( ! vte.Valid ) then        {
            return false;

        Address rd1 = vte.TargetAddress;
        Address vpt = vte.VPT_base;

        WriteGICR_VINVLPIR( rd1, vpt, ite.OutputID );
    }

    return true;
```


5.13.8 Set the Pending State for an Interrupt Translation Table Entry

The pseudo-code below shows the actions required to set an interrupt as “pending” once the Interrupt Translation Table entry has been found:

```
boolean SetPendingState(ITE ite)
    if ( ite.physical_interrupt ) then
        CTE cte = ReadCollectionTable( ite.Collection );

        if ( ! cte.Valid ) then
            return false;

        Address rd1 = cte.TargetAddress;

        if ( GITS_TYPER.Distributed ) then
            WriteGICR_SETLPIR( rd1, ite.OutputID );
        else
            SetPendingState( rd1.PendBaseR, ite.OutputID );
    else
        VTE vte = ReadVCPUTable( ite.VCPU );

        if ( ! vte.Valid ) then
            return false;

        Address rd1 = vte.TargetAddress;
        Address vpt = vte.VPT_base;

        if ( GITS_TYPER.Distributed && GITS.TYPER.IDbits < 16 ) then
            // Interrupt identifier size is <= 16 bits
            WriteGICR_VSETLPIR( rd1, vpt, ite.OutputID, ite.HypervisorID );
        else if ( GITS_TYPER.Distributed ) then
            // Interrupt identifier size is > 16 bits
            // Note: the ITS must ensure the following sequence is performed
            // with no intervening writes to the same re-distributor by this ITS
            WriteGICR_VDESTRn( rd1, vpt );
            WriteGICR_VSETLPIRn( rd1, ite.OutputID, ite.HypervisorID );
        else
            SetPendingStateLocal( vpt, ite.OutputID );

            if ( rd1.VPendBaseR.Valid && rd1.VPendBaser.Address != vpt ) then
                if ( ite.HypervisorID != 1023 ) then
                    // Not resident so set the hypervisor interrupt pending as well
                    SetPendingStateLocal( rd1.PendBaseR, ite.HypervisorID );

    return true;
```

Note: the pseudo-code returns true if the operation was successful.

5.13.9 Clear the Pending State for an Interrupt Translation Table Entry

The pseudo-code below shows the actions required to clear the “pending” state of an interrupt once the Interrupt Translation Table entry has been found:

```
boolean ClearPendingState(ITE ite)
    if ( ite.physical_interrupt ) then
        CTE cte = ReadCollectionTable( ite.Collection );

        if ( ! cte.Valid ) then
            return false;

        Address rd1 = cte.TargetAddress;

        if ( GITS_TYPER.Distributed ) then
            WriteGICR_CLRLPIR( rd1, ite.OutputID );
        else
            ClearPendingStateLocal( rd1.PendBaseR, ite.OutputID );
    else
        VTE vte = ReadVCPUTable( ite.VCPU );

        if ( ! vte.Valid ) then
            return false;

        Address rd1 = vte.TargetAddress;
        Address vpt = vte.VPT_base;

        if ( GITS_TYPER.Distributed ) then
            WriteGICR_VCLRLPIR( rd1, vpt, ite.OutputID );
        else
            ClearPendingStateLocal( vpt, ite.OutputID );

    return true;
```

5.13.10 Move Virtual Pending State

The pseudo-code below shows the actions required to move the “pending” state of an interrupt in a monolithic implementation:

```
MoveVirtualPendingState(Address rd1, Address vpt1, Address vpt2, identifier ID)
    if ( IsPending(vpt1, ID) ) then
        // The interrupt is pending in the source re-distributor

        // Make sure the interrupt is released or taken by the processor for
        // example by sending a VClear and waiting for the response
        EnsureVirtualInterruptNotPendingOnProcessor( rd1, vpt1, ID );

        if ( IsPending(vpt1, ID) ) then
            // The CPU released the interrupt and it is still pending
            // Note: the following must be done without any possibility of the
            // source re-distributor re-forwarding the interrupt to the processor
            ClearPendingState( vpt1, ID );
            SetPendingState( vpt2, ID );
    return;
```


5.13.11 MAPD device, ITT descriptor

This command describes the Interrupt Translation Table to be used for a specified device.

The descriptor comprises a 256 byte aligned physical address and the number of bits of interrupt ID supported.

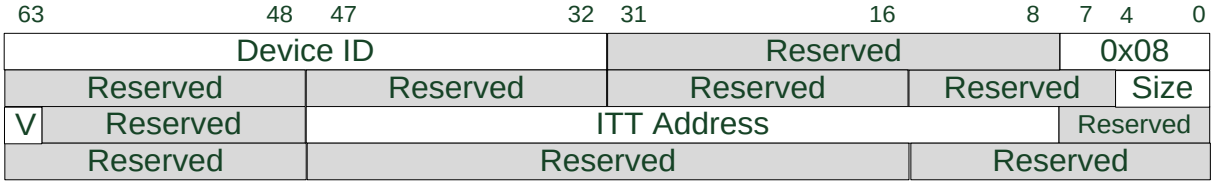


Figure 29: MAPD command format

Where:

- "Device ID" specifies the identifier of a device which will use the provided Interrupt Translation Table (ITT). ***Note***: more than one device might share an ITT.
- "V" is a "Valid" bit that specifies whether the ITT Address and Size are valid. When "Valid" is set to zero, this command un-maps the specified device and translation request from that device will be discarded.
- "ITT Address" specifies bits [47:8] of the physical address of the Interrupt Translation Table. Bits [7:0] of the physical address are zero. ***Note***: the format of ITT entries is implementation defined but assuming an entry size of 8 bytes, this allows allocation of identifiers to devices in multiples of 32 interrupts.
- "Size" is a 5 bit number that specifies the number of bits of interrupt ID supported for this device, minus one.

It is a command error (see section 5.13.4) if:

- "Device ID" exceeds the maximum value supported by the ITS.
- "Size" exceeds the maximum value permitted by GITS_TYPER.IDbits.

Note: the number of bits of "Device ID" supported by an implementation is discoverable from GITS_TYPER (see section 5.12.6). Unimplemented bits are RES0.

Note: software might issue a MAPD command to re-map an already mapped device and the ITS must invalidate all cached data for that device.

Note: ITS accesses to an ITT use the same shareability and cachability attributes as specified for the Device Table (see section 5.12.12).

The pseudo-code below shows the code for the MAPD command:

```
perform_mapd(command cmd)
    if ( DeviceOutOfRange( cmd.device ) ) then
        CommandError( MAPD_DEVICE_OOR );
        return;

    if ( SizeOutOfRange( cmd.ITT_size ) ) then
        CommandError( MAPD_ITTSIZE_OOR );
        return;

    // If a device is RE-mapped software must perform the following actions
    // to ensure the LPI configuration is up to date:
    // 1.  Ensure that the device is quiescent and that all interrupts have
    //     been handled.
    // 2.  Remap the device with the new (empty) ITT
    //
    DTE dte;

    dte.Valid = cmd.V;
    dte.ITT_base = cmd.ITT_base;
    dte.ITT_size = cmd.ITT_size;

    WriteDeviceTable( cmd.device, dte );
```



```
    IncrementReadPointer();
```

5.13.12 MAPC collection, Target Address

This command maps the interrupt collection specified by “collection” to the re-distributor specified by “Target Address”.

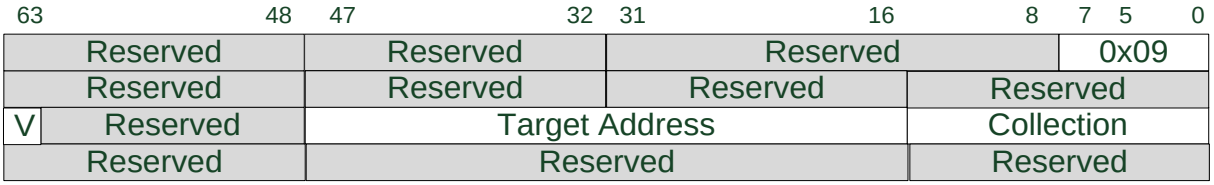


Figure 30: MAPC command format

Where:

- “V” is a “Valid” bit that specifies whether the Target Address is valid for the collection. When “Valid” is set to zero, this command un-maps the specified interrupt collection and interrupts for that collection will be discarded.
- “Collection” specifies the interrupt collection.
- “Target Address” specifies the physical address of the re-distributor to which interrupts for the collection will be forwarded. ***Note***: if GITS_TYPER.PTA is zero, Target Address specifies a linear processor number. See section 5.12.6.

It is a command error (see section 5.13.4) if:

- The “Collection” exceeds the maximum number supported by the ITS (see section 4.9.2)
- If GITS_TYPER.PTA is zero, if “Target Address” specifies an invalid linear processor number

The pseudo-code below shows the code for the MAPC command:

```
perform_mapc(command cmd)
    if ( CollectionOutOfRange( cmd.collection ) ) then
        CommandError( MAPC_COLLECTION_OOR );
        return;

    if (GITS_TYPER.PTA == 0 && ProcessorNumberOutOfRange( cmd.TargetAddress) then
        CommandError( MAPC_PROCNUM_OOR );
        Return;

    CTE cte;

    cte.Valid = cmd.V;
    cte.TargetAddress = cmd.TargetAddress;

    WriteCollectionTable( cmd.collection, cte );

    if ( GITS_TYPER.Distributed && cmd.V == '1' ) then
        // This invalidation is required because the INV command only invalidates
        // caches in the currently targetted re-distributor.  As a consequence,
        // when a collection is mapped to a re-distributor, that re-distributor
        // might still have cached data of an old configuration.
        //
        WriteGICR_INVALLR( cmd.TargetAddress );

    IncrementReadPointer();
```


5.13.13 MAPI device, ID, collection

This command specifies that the interrupt with identifier “ID” from the device specified by “device” is mapped to the interrupt collection specified by “collection”.

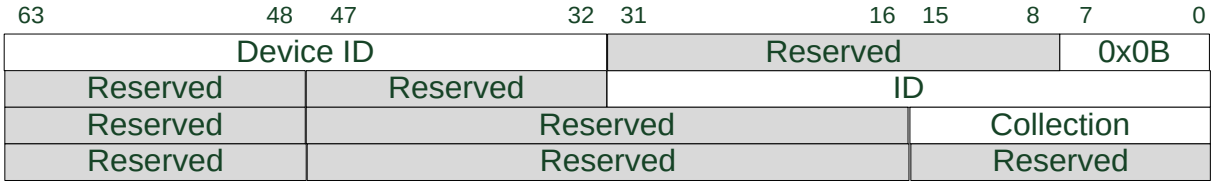


Figure 31: MAPI command format

Where:

- “ID” is the identifier of the interrupt that is presented to the software that controls the device specified by “Device ID”.
- “Collection” specifies the interrupt collection of which the interrupt with identifier “ID” is a member.

It is a command error (see section 5.13.4) if:

- The device specified by “Device ID” has not been mapped to an ITT (using MAPD in section 5.13.11), or
- The “Collection” exceeds the maximum number supported by the ITS (see section 4.9.2), or
- The “ID” exceeds the maximum value allowed by the “Size” field in the MAPD command

Note: the number of bits of “Device ID” and “ID” supported by an implementation are discoverable from GITS_TYPER (see section 5.12.6). Unimplemented bits are RES0.

The pseudo-code below shows the code for the MAPI command:

```
perform_mapi(command cmd)
    if ( DeviceOutOfRange( cmd.device ) ) then
        CommandError( MAPI_DEVICE_OOR );
        return;

    if ( CollectionOutOfRange( cmd.collection ) ) then
        CommandError( MAPI_COLLECTION_OOR );
        return;

    DTE dte = ReadDeviceTable( cmd.device );

    if ( ! dte.Valid) then
        CommandError( MAPI_UNMAPPED_DEVICE );
        return;

    if ( IdOutOfRange( cmd.ID, dte.ITT_size ) ) then
        CommandError( MAPI_ID_OOR );
        return;

    if ( LPIOutOfRange( cmd.ID ) ) then
        CommandError( MAPI_ID_OOR );
        return;

    ITE ite;

    ite.Valid = true;
    ite.Type = physical_interrupt;
    ite.OutputID = cmd.ID;
    ite.HypervisorID = 1023; // Don't generate a doorbell
```



```
    ite.Collection = cmd.collection;

    if (GITS_TYPER.Distributed) then
        // This invalidation is required because the INV command only invalidates
        // caches in the currently targetted re-distributor.  As a consequence,
        // when an interrupt is mapped to a re-distributor, that re-distributor
        // might still have cached data of an old configuration.
        //
        boolean success = invalidateByITE( ite );

    WriteTranslationTable( dte.ITT_base, cmd.ID, ite );

    IncrementReadPointer();
```

5.13.14 MAPVI device, ID, pID, collection

This command specifies that the interrupt with identifier “ID” from the device specified by “device” is mapped to the interrupt with identifier “pID” that belongs to the interrupt collection specified by “collection”.

Note: it is expected that in GICv3 systems this command will only be used by hypervisor software.

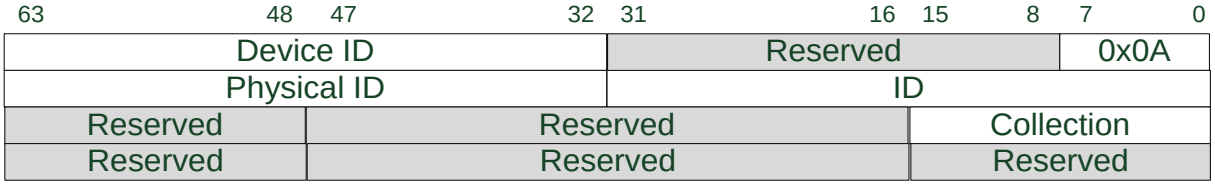


Figure 32: MAPVI command format

Where:

- “ID” is the input identifier that is used for the translation process.
- “Physical ID” the identifier of the interrupt that is presented to software.
- “Collection” specifies the interrupt collection of which the interrupt with identifier “Physical ID” is a member.

It is a command error (see section 5.13.4) if:

- The device specified by “Device ID” has not been mapped to an ITT (using MAPD in section 5.13.11), or
- The “Collection” exceeds the maximum number supported by the ITS (see section 4.9.2), or
- The “ID” exceeds the maximum value allowed by the “Size” field in the MAPD command, or
- The “Physical ID” does not specify a valid LPI identifier (i.e. it is <8192 or exceeds the maximum supported).

Note: the number of bits of “Device ID”, “ID” and “Physical ID” supported by an implementation are discoverable from GITS_TYPER (see section 5.12.6). Unimplemented bits are RES0.

The pseudo-code below shows the code for the MAPI command:

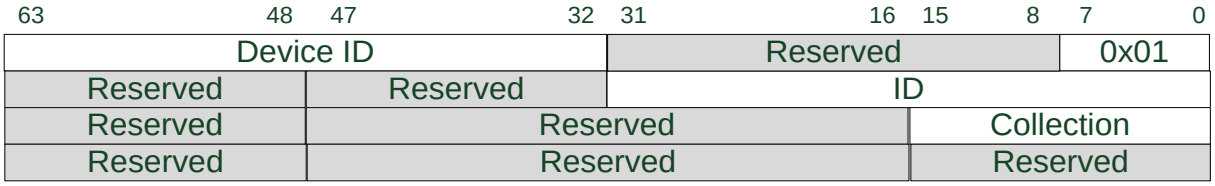
```
perform_mapvi(command cmd)
    if ( DeviceOutOfRange( cmd.device ) ) then
        CommandError( MAPVI_DEVICE_OOR );
        return;

    if ( CollectionOutOfRange( cmd.collection ) ) then
        CommandError( MAPVI_COLLECTION_OOR );
        return;

    DTE dte = ReadDeviceTable( cmd.device );

    if ( ! dte.Valid) then
```



```
        CommandError( MAPVI_UNMAPPED_DEVICE );
        return;

    if ( IdOutOfRange( cmd.ID, dte.ITT_size ) ) then
        CommandError( MAPVI_ID_OOR );
        return;

    if ( LPIOutOfRange( cmd.PhysicalID ) ) then
        CommandError( MAPVI_PHYSICALID_OOR );
        return;

    ITE ite;

    ite.Valid = true;
    ite.Type = physical_interrupt;
    ite.OutputID = cmd.PhysicalID;
    ite.HypervisorID = 1023; // Don't generate a doorbell
    ite.Collection = cmd.collection;

    if (GITS_TYPER.Distributed) then
        // This invalidation is required because the INV command only invalidates
        // caches in the currently targetted re-distributor.  As a consequence,
        // when an interrupt is mapped to a re-distributor, that re-distributor
        // might still have cached data of an old configuration.
        //
        boolean success = invalidateByITE( ite );

    WriteTranslationTable( dte.ITT_base, cmd.ID, ite );

    IncrementReadPointer();
```


5.13.15 MOVI device, ID, collection

This command specifies that interrupts with the specified identifier for the device now belong to the specified interrupt collection.



Where:

- "ID" is the input identifier of the interrupt to be re-directed from the device specified by "Device ID".
- "Collection" specifies the new collection of which the interrupt is to be a member.

It is a command error (see section 5.13.4) if:

- The device specified by "Device ID" has not been mapped to an ITT (using MAPD in section 5.13.11), or
- The "Collection" exceeds the maximum number supported by the ITS (see section 4.9.2), or
- The "Collection" has not been mapped to a Target Address (see MAPC in section 5.13.12), or
- The "ID" has not been mapped to a valid collection for the specified device (see MAPI and MAPVI in sections 5.13.13 and 5.13.14), or
- The "ID" corresponds to a virtual LPI

Note: the number of bits of "Device ID" and "ID" supported by an implementation are discoverable from GITS_TYPER (see section 5.12.6). Unimplemented bits are RES0.

Note: if software wishes to move an interrupt from collection A to B and then from B to C it must issue a "SYNC" command to the re-distributor for collection A between these moves to ensure correct behaviour. Otherwise, the interrupt might still occur on processor associated with collection B even after completion of the second MOVI command.

The pseudo-code below shows the code for the MOVI command:

```
MovePendingState(Address rd1, Address rd2, identifier ID)
    if ( IsPending( rd1.PendBaseR, ID ) ) then
        // The interrupt is pending in the source re-distributor

        // Make sure the interrupt is released or taken by the processor for
        // example by sending a clear and waiting for the response
        EnsureInterruptNotPendingOnProcessor( rd1, ID );

        if ( IsPending( rd1.PendBaseR, ID ) ) then
            // The CPU released the interrupt and it is still pending
            // Note: the following must be done without any possibility of the
            // source re-distributor re-forwarding the interrupt to the processor
            ClearPendingState( rd1.PendBaseR, ID );
            SetPendingState( rd2.PendBaseR, ID );

perform_movi(command cmd)
    if ( DeviceOutOfRange( cmd.device ) ) then
        CommandError( MOVI_DEVICE_OOR );
        return;

    if ( CollectionOutOfRange( cmd.collection ) ) then
        CommandError( MOVI_COLLECTION_OOR );
        return;

    DTE dte = ReadDeviceTable( cmd.device );

    if ( ! dte.Valid ) then
        CommandError( MOVI_UNMAPPED_DEVICE );
```



```
    return;

if ( IdOutOfRange( cmd.ID, dte.ITT_size ) ) then
    CommandError( MOVI_ID_OOR );
    return;

ITE ite = ReadTranslationTable( dte.ITT_base, cmd.ID );

if ( ! ite.Valid ) then
    CommandError( MOVI_UNMAPPED_INTERRUPT );
    return;

if ( ite.Type == virtual_interrupt ) then
    CommandError( MOVI_ID_IS_VIRTUAL );
    return;

CTE cte1 = ReadCollectionTable( ite.Collection );

if ( ! cte1.Valid ) then
    CommandError( MOVI_UNMAPPED_COLLECTION );
    return;

CTE cte2 = ReadCollectionTable( cmd.collection );

if ( ! cte2.Valid) then
    CommandError( MOVI_UNMAPPED_COLLECTION );
    return;

Address rd1 = cte1.TargetAddress;
Address rd2 = cte2.TargetAddress;

if ( rd1 != rd2 ) then
    if ( GITS_TYPER.Distributed ) then
        if ( SpeculativeCachingSupported() ) then
            // The redistributors support speculative caching and an invalidate
            // must be issued to rd2.  Any subsequent LPIs as a consequence of
            // writes to GITS_TRANSLATER must occur after this.
            WriteGICR_INVLPIR( rd2, ite.OutputID );

        // Issue a command to rd1 to move the pending state to rd2 if set
        // rd1 performs the equivalent of MovePendingState
        WriteGICR_MOVLPIR( rd1, rd2, ite.OutputID );
    else
        // Move the move the pending state to rd2 if set taking care of
        // any races where the interrupt has been forwarded to the processor
        MovePendingState( rd1, rd2, ite.OutputID );

ite.Collection = cmd.collection;

WriteTranslationTable( dte.ITT_base, cmd.ID, ite );
IncrementReadPointer();
```


5.13.16 DISCARD device, ID

This command specifies that incoming interrupt requests with a specified ID from the device will be silently discarded.

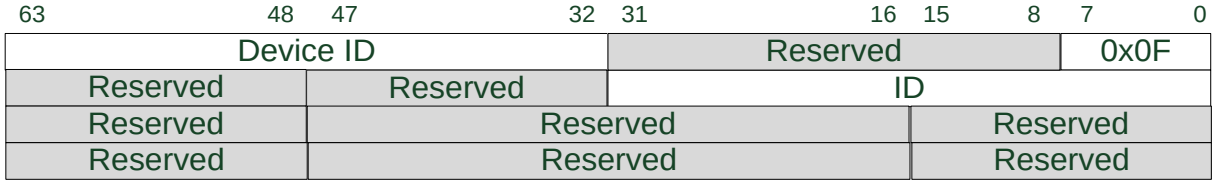


Figure 33: DISCARD command

Where:

- "ID" is the input identifier of the interrupt to be discarded from the device specified by "Device ID".

Note: the number of bits of "Device ID" and "ID" supported by an implementation are discoverable from GITS_TYPER (see section 5.12.6). Unimplemented bits are RES0.

It is a command error (see section 5.13.4) if:

- The device specified by "Device ID" has not been mapped to an ITT (using MAPD in section 5.13.11), or
- The "ID" has not been mapped to a Collection (see MAPI and MAPVI in sections 5.13.13 and 5.13.14), or
- The "ID" belongs to an unmapped Collection has not been mapped to a Target Address (see MAPC in section 5.13.12)

Note: this command will also perform an implicit Clear command to ensure that the interrupt is not in the pending state and must ensure any caching of LPI configuration data for the specified LPI is discarded.

The pseudo-code below shows the code for the DISCARD command:

```
perform_discard(command cmd)
    if (DeviceOutOfRange( cmd.device ) ) then
        CommandError( DISCARD_DEVICE_OOR );
        return;

    DTE dte = ReadDeviceTable( cmd.device );

    if ( ! dte.Valid ) then
        CommandError( DISCARD_UNMAPPED_DEVICE );
        return;

    if (IdOutOfRange( cmd.ID, dte.ITT_size ) ) then
        CommandError( DISCARD_ID_OOR );
        return;

    ITE ite = ReadTranslationTable( dte.ITT_base, cmd.ID );
    if ( ite.Valid ) then
        bool success = ClearPendingState( ite );
        if ( ! success ) then
            CommandError( DISCARD_ITE_INVALID );
            return;

    if ( GITS_TYPER.Distributed ) then
        boolean success = invalidateByITE( ite );

        if ( ! success ) then
            CommandError( DISCARD_ITE_INVALID );
            return;
```



```
else
    // This invalidates any caches containing the configuration data for the
    // specified interrupt within the collection.
    // NOTE: over invalidation is permitted.
    InvalidateInterruptConfigurationCaches( cmd.ID, ite.Collection );

ite.Valid = false;
WriteTranslationTable( dte.ITT_base, cmd.ID, ite );

IncrementReadPointer();
```


5.13.17 INV device, ID

This command specifies that the ITS must ensure that the re-distributor that currently owns a specified ID for the device must ensure any caching associated with this index is consistent with the configuration tables held in memory.

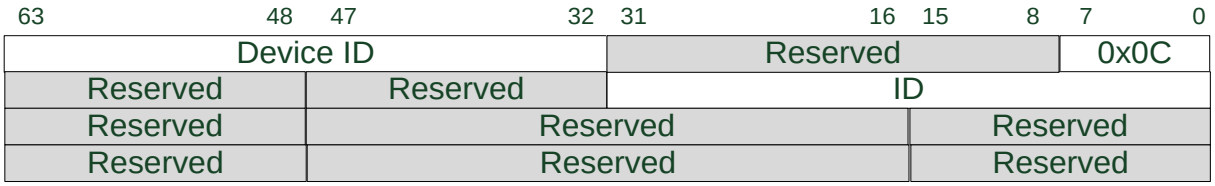


Figure 34: INV command format

Where:

- "ID" is the input identifier of the interrupt to be "cleaned" for the device specified by "Device ID".

It is a command error (see section 5.13.4) if:

- The device specified by "Device ID" has not been mapped to an ITT (using MAPD in section 5.13.11), or
- The "ID" has not been mapped (see MAPI and MAPVI in sections 5.13.13 and 5.13.14; VMAPI and VMAPVI in sections 5.13.24 and 5.13.25), or
- The "ID" corresponds to a physical LPI and belongs to an unmapped Collection (see MAPC in section 5.13.12)
- The "ID" corresponds to a virtual LPI and belongs to an unmapped VCPU (see VMAPP in section 5.13.23)

Note: the number of bits of "Device ID" and "ID" supported by an implementation are discoverable from GITS_TYPER (see section 5.12.6). Unimplemented bits are RES0.

Note: this command is expected to be used by software when it changed the re-configuration of an LPI in memory to ensure any cached copies of the old configuration are discarded. See section 4.8.11.

The pseudo-code below shows the code for the INV command:

```
perform_inv(command cmd)
    if ( DeviceOutOfRange( cmd.device ) ) then
        CommandError( INV_DEVICE_OOR );
        return;

    DTE dte = ReadDeviceTable( cmd.device );

    if ( ! dte.Valid ) then
        CommandError( INV_UNMAPPED_DEVICE );
        return;

    if ( IdOutOfRange( cmd.ID, dte.ITT_size ) ) then
        CommandError( INV_ID_OOR );
        return;

    ITE ite = ReadTranslationTable( dte.ITT_base, cmd.ID );

    if ( ! ite.Valid ) then
        // Monolithic implementations might not use the ITE data and might
        // choose not to implement this check
        CommandError( INV_UNMAPPED_INTERRUPT );
        return;
```



```
    if ( GITS_TYPER.Distributed ) then
        boolean success = invalidateByITE( ite );

        if ( ! success ) then
            CommandError( INV_ITE_INVALID );
            return;
    else
        // This invalidates any caches containing the configuration data for the
        // specified interrupt within the collection.
        // NOTE: over invalidation is permitted.
        InvalidateInterruptConfigurationCaches( cmd.ID, ite.Collection );

    IncrementReadPointer();
```

5.13.18 MOVALL Target Address 1, Target Address 2

This command specifies that the ITS must move all interrupts associated with the re-distributor at Target Address 1 to the re-distributor at Target Address 2.

Note: both the mapping of interrupts to collections and the mapping of collections to re-distributors are unaffected by this command. Software must ensure that any interrupts that might be affected by the MOVALL command target the re-distributor specified by Target Address 2 otherwise system behaviour might be unpredictable. In particular, an implementation might choose to re-map all affected collections to Target Address 2.

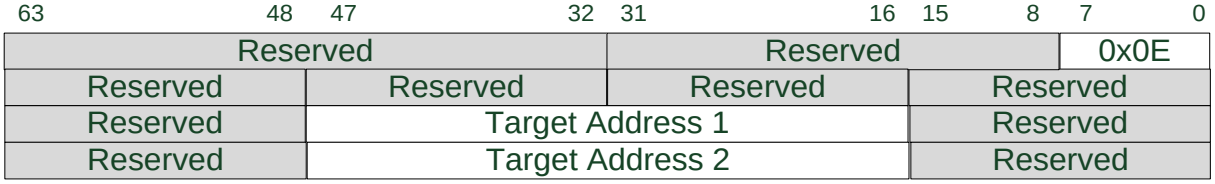


Figure 35: MOVALL command format

Where:

- "Target Address 1" specifies the old re-distributor
- "Target Address 2" specifies the new re-distributor.

It is a command error (see section 5.13.4) if:

- If GITS_TYPER.PTA is zero, if either of "Target Address1" or "TargetAddress2" specifies an invalid linear processor number

The pseudo-code below shows the code for the MOVALL command:

```
perform_movall(command cmd)
    Address rd1 = cmd.TargetAddress1;
    Address rd2 = cmd.TargetAddress2;

    if (GITS_TYPER.PTA == 0) then
        if (ProcessorNumberOutOfRange( rd1 ) || ProcessorNumberOutOfRange( rd2 ) ) then
            CommandError( MOVALL_PROCNUM_OOR );
            return;

    if ( rd1 != rd2 ) then
        if ( GITS_TYPER.Distributed ) then
            if ( SpeculativeCachingSupported() ) then
                // The redistributors support speculative caching and an invalidate
                // must be issued to rd2.  Any subsequent LPIs as a consequence of
                // writes to GITS_TRANSLATER must occur after this.
```



```
            WriteGICR_INVALLR( rd2 );

        WriteGICR_MOVALLR( rd1, rd2 );
    else
        MoveAllPendingState( rd1, rd2 );

IncrementReadPointer();
```


5.13.19 INVALL collection

This command specifies that the ITS must ensure any caching associated with the specified interrupt collection is consistent with the configuration tables held in memory in all re-distributors.

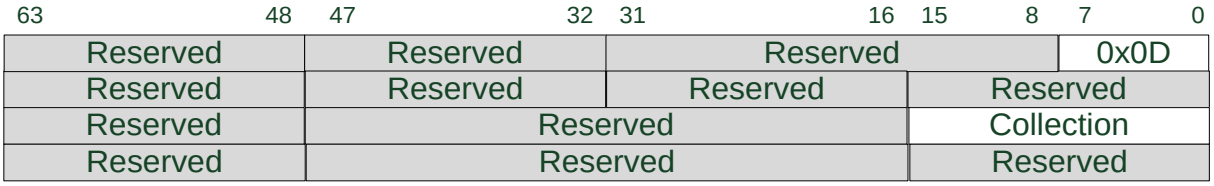


Figure 36: INVALL command format

Where:

- "Collection" specifies the interrupt collection.

It is a command error (see section 5.13.4) if:

- The "Collection" has not been mapped to a Target Address (see MAPC in section 5.13.12)

Note: this command is expected to be used by software when it changed the re-configuration of an LPI in memory to ensure any cached copies of the old configuration are discarded. See section 4.8.11.

The pseudo-code below shows the code for the INVALL command:

```
perform_invall(command cmd)
    if ( CollectionOutOfRange( cmd.collection ) ) then
        CommandError( INVALL_COLLECTION_OOR );
        return;

    CTE cte = ReadCollectionTable( cmd.collection );

    if ( ! cte.Valid ) then
        // Monolithic implementations might not use the CTE data and might
        // choose not to implement this check
        CommandError ( INVALL_UNMAPPED_COLLECTION );
        return;

    if ( GITS_TYPER.Distributed ) then
        Address rd1 = cte.TargetAddress;

        WriteGICR_INVALLR( rd1 );
    else
        // This invalidates any caches containing the configuration data for all
        // interrupts in the collection.  NOTE: over invalidation is permitted.
        InvalidateCollectionCaches( cmd.collection );

    IncrementReadPointer();
```


5.13.20 INT device, ID

This command specifies that interrupt with a specified ID must be generated for the device.

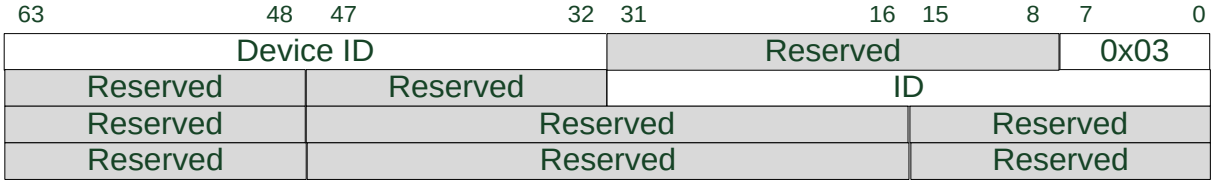


Figure 37: INT command format

Where:

- "ID" is the input identifier of the interrupt to be generated for the device specified by "Device ID".

It is a command error (see section 5.13.4) if:

- The device specified by "Device ID" has not been mapped to an ITT (using MAPD in section 5.13.11), or
- The "ID" has not been mapped to a Collection (see MAPI and MAPVI in sections 5.13.13 and 5.13.14), or
- The ID belongs to an unmapped Collection has not been mapped to a Target Address (see MAPC in section 5.13.12)

Note: the number of bits of "Device ID" and "ID" supported by an implementation are discoverable from GITS_TYPER (see section 5.12.6). Unimplemented bits are RES0.

The pseudo-code below shows the operation of the INT command:

```
perform_int(command cmd)
    if ( DeviceOutOfRange( cmd.device ) ) then
        CommandError ( INT_DEVICE_OOR );
        return;

    DTE dte = ReadDeviceTable( cmd.device );

    if ( ! dte.Valid ) then
        CommandError ( INT_UNMAPPED_DEVICE );
        return;

    if ( IdOutOfRange( cmd.ID, dte.ITT_size ) ) then
        CommandError ( INT_ID_OOR );
        return;

    ITE ite = ReadTranslationTable( dte.ITT_base, cmd.ID );

    if ( ! ite.Valid ) then
        CommandError ( INT_UNMAPPED_INTERRUPT );
        return;

    bool success = SetPendingState( ite );
    if ( ! success ) then
        CommandError ( INT_ITE_INVALID );
        return;
    IncrementReadPointer();
```


5.13.21 CLEAR device, ID

This command specifies that the pending state for an interrupt with a specified ID must be cleared.

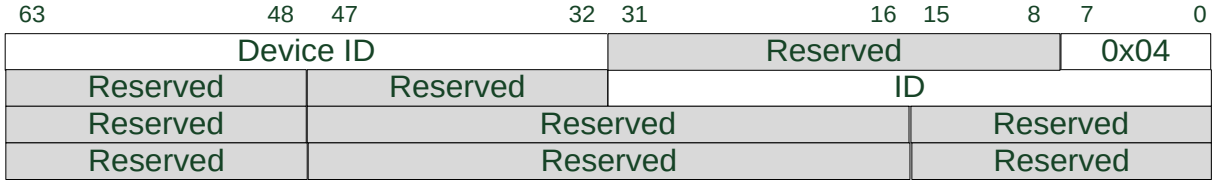


Figure 38: CLEAR command format

Where:

- "ID" is the input identifier of the interrupt for which the pending state is to be cleared for the device specified by "Device ID".

It is a command error (see section 5.13.4) if:

- The device specified by "Device ID" has not been mapped to an ITT (using MAPD in section 5.13.11), or
- The "ID" has not been mapped to a Collection (see MAPI and MAPVI in sections 5.13.13 and 5.13.14), or
- The ID belongs to an unmapped Collection has not been mapped to a Target Address (see MAPC in section 5.13.12)

Note: the number of bits of "Device ID" and "ID" supported by an implementation are discoverable from GITS_TYPER (see section 5.12.6). Unimplemented bits are RES0.

The pseudo-code below shows the operation of the CLEAR command:

```
perform_clear(command cmd)
    if ( DeviceOutOfRange( cmd.device ) ) then
        CommandError( CLEAR_DEVICE_OOR );
        return;

    DTE dte = ReadDeviceTable( cmd.device );
    if ( ! dte.Valid ) then
        CommandError( CLEAR_UNMAPPED_DEVICE );
        return;

    if (IdOutOfRange( cmd.ID, dte.ITT_size ) ) then
        GenerateError( CLEAR_ID_OOR );
        return;

    ITE ite = ReadTranslationTable( dte.ITT_base, cmd.ID );
    if ( ! ite.Valid ) then
        CommandError( CLEAR_UNMAPPED_INTERRUPT );
        return;

    bool success = ClearPendingState( ite );
    if ( ! success ) then
        CommandError( CLEAR_ITE_INVALID );
        return;
    IncrementReadPointer();
```


5.13.22 SYNC Target Address

This command specifies that the ITS must wait for completion of all outstanding actions relating to the specified re-distributor.

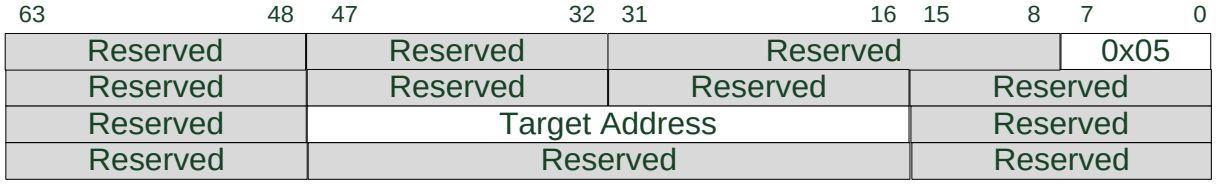


Figure 39: SYNC command format

Where:

- "Target Address" specifies the physical address of the re-distributor.

The pseudo-code below shows the operation of the SYNC command:

```
perform_sync(command cmd)
    // Wait for the external effects of any physical commands to be observable
    // by all re-distributors and ensure the internal effects of any previous
    // physical commands affect any subsequent physical interrupt requests or commands
    WaitForCompletion( cmd.TargetAddress );

    if ( GITS_TYPER.Distributed ) then
        // It is implementation defined whether a re-distributor stalls until
        // completion of a write to GICR_SYNCR or returns 0x0000_0001 until
        // the SYNC is complete.
        //
        while ( ReadGICR_SYNCR( cmd.TargetAddress ) != 0)
            ; // Do nothing

    IncrementReadPointer();
```


5.13.23 VMAPP VCPU, Target Address, VPT descriptor

In GICv4, this command specifies that a specified virtual processor is mapped to a specified re-distributor and also specifies the base address and size of the Virtual Pending Table for the virtual processor.

The VPT descriptor comprises a 64kB aligned physical address and the number of bits of interrupt ID supported.

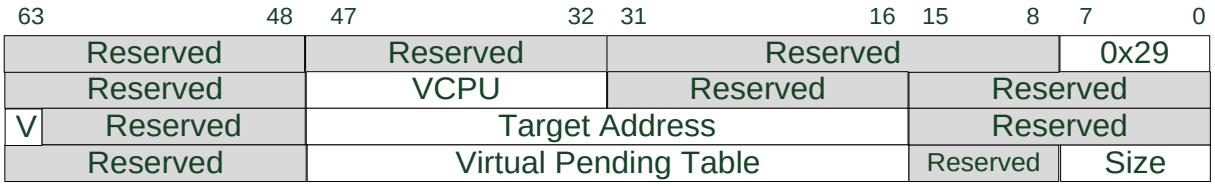


Figure 40: VMAPP command format

Where:

- "V" is a "Valid" bit that specifies whether the Target Address and Virtual Pending Table are valid for the VCPU. When "Valid" is set to zero, this command un-maps the specified VCPU and interrupts for that VCPU will be discarded.
- "Target Address" specifies the re-distributor that owns the virtual processor and to which the ITS will direct all commands for the virtual processor.
- "Virtual Pending Table" specifies bits [47:16] of the physical address of the Virtual Pending Table for the virtual processor. Bits [15:0] of the physical address are zero.
- "Size" specifies the number of bits of Virtual ID supported, minus one.

It is a command error (see section 5.13.4) if:

- "VCPU" exceeds the maximum number supported by the ITS (see GITS_BASERn in section 5.12.12)
- If GITS_TYPER.PTA is zero, if "TargetAddress" specifies an invalid linear processor number

The pseudo-code below shows the operation of the VMAPP command:

```
perform_vmapp(command cmd)
    if ( VCPUOutOfRange( cmd.vcpu ) ) then
        CommandError( VMAPP_VCPU_OOR );
        return;
    if ( SizeOutOfRange( cmd.VPT_size ) ) then
        CommandError( VMAPP_VPTSIZE_OOR );
        return;
    if ( ProcessorNumberOutOfRange( cmd.TargetAddress ) ) then
        CommandError( VMAPP_PROCNUM_OOR );
        return;

    VTE vte;
    vte.Valid = cmd.V;
    vte.TargetAddress = cmd.TargetAddress;
    vte.VPT = cmd.VirtualPendingTable;
    vte.VPT_size = cmd.VPT_size;

    WriteVCPUTable( cmd.vcpu, vte );

    if ( GITS_TYPER.Distributed && cmd.V == '1' ) then
        // This invalidation is required because the INV command only invalidates
        // caches in the currently targetted re-distributor.  As a consequence,
        // when a VCPU is mapped to a re-distributor, that re-distributor
        // might still have cached data of an old configuration.
        //
        WriteGICR_VINVALLR( cmd.vcpu );
```



```
    IncrementReadPointer();
```


5.13.24 VMAPI device, ID, pID, VCPU

In GICv4, this command specifies that virtual interrupts with a specified identifier will be generated on a specified virtual processor.

If the specified virtual processor is not resident then a specified physical interrupt will be generated on the re-distributor that the virtual processor is mapped to.

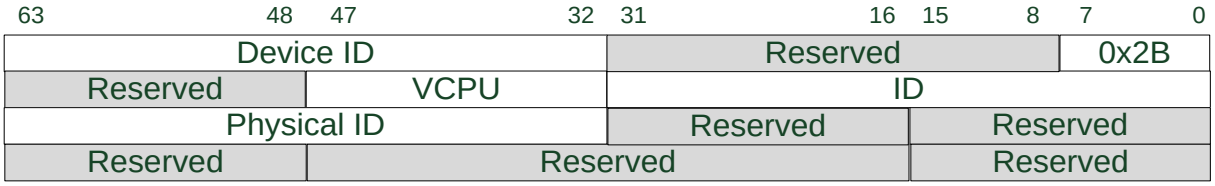


Figure 41: VMAPI command format

Where

- "Device ID" specifies a device owned by the virtual processor specified by "VCPU"
- "ID" specified the identifier of the interrupt that will be presented to the guest operating system
- "Physical ID" specifies the identifier that will be presented to the hypervisor if the virtual processor is not resident. ***Note***: if the Physical ID is a spurious ID (i.e. 1023) then no physical interrupt will be generated.

It is a command error (see section 5.13.4) if:

- The device specified by "Device ID" has not been mapped to an ITT (using MAPD in section 5.13.11), or
- The "ID" exceeds the maximum value allowed by the "Size" field in the MAPD command, or
- The "ID" does not specify a valid LPI identifier (i.e. it is <8192 or exceeds the maximum supported)
- The "Physical ID" does not specify a valid doorbell identifier (i.e. it is not 1023 and is <8192 or exceeds the maximum supported).

Note: the number of bits of "Device ID", "ID" and "Physical ID" supported by an implementation are discoverable from GITS_TYPER (see section 5.12.6). Unimplemented bits are RES0***.***

The pseudo-code below shows the operation of the VMAPI command:

```
perform_vmapi(command cmd)
    if ( DeviceOutOfRange( cmd.device ) ) then
        CommandError( VMAPI_DEVICE_OOR );
        return;

    if ( VCPUOutOfRange( cmd.vcpu ) ) then
        CommandError( VMAPI_VCPU_OOR );
        return;

    DTE dte = ReadDeviceTable( cmd.device );

    if ( ! dte.Valid) then
        CommandError( VMAPI_UNMAPPED_DEVICE );
        return;

    if ( IdOutOfRange( cmd.ID, dte.ITT_size ) ) then
        CommandError( VMAPI_ID_OOR );
        return;

    if ( LPIOutOfRange( cmd.ID ) ) then
        CommandError( VMAPI_ID_OOR );
        return;
```



```
if ( LPIOutOfRange( cmd.PhysicalID ) && cmd.PhysicalID != 1023 ) then
    CommandError( VMAPI_PHYSICALID_OOR );
    return;

ITE ite = ReadTranslationTable( dte.ITT_base, cmd.ID );

ite.Valid = true;
ite.Type = virtual_interrupt;
ite.OutputID = cmd.ID;
ite.HypervisorID = cmd.PhysicalID;
ite.VCPU = cmd.vcpu;

if (GITS_TYPER.Distributed) then
    // This invalidation is required because the INV command only invalidates
    // caches in the currently targetted re-distributor.  As a consequence,
    // when an interrupt is mapped to a re-distributor, that re-distributor
    // might still have cached data of an old configuration.
    //
    boolean success = invalidateByITE( ite );

WriteTranslationTable( dte.ITT_base, cmd.ID, ite );
IncrementReadPointer();
```


5.13.25 VMAPVI device, ID, vID, pID, VCPU

In GICv4, this command specifies that virtual interrupts with a specified identifier will be generated on a specified virtual processor.

If the specified virtual processor is not resident then a specified physical interrupt will be generated on the re-distributor that the virtual processor is mapped to.

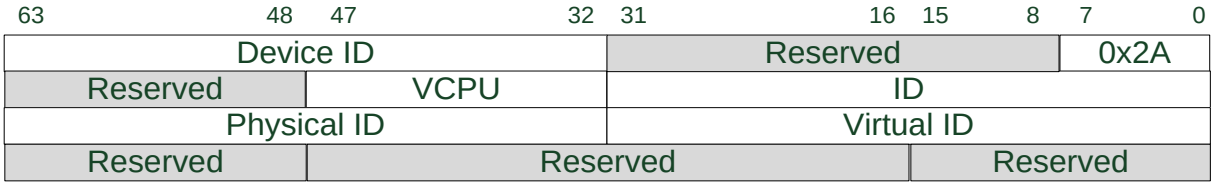


Figure 42: VMAPVI command format

Where

- "Device ID" specifies a device owned by the virtual processor specified by "VCPU"
- "ID" specified the input identifier that will be used for the translation process
- "Virtual ID" specified the identifier that is presented to the guest operating system that controls the device specified by "Device ID"
- "Physical ID" specifies the identifier that will be presented to the hypervisor if the virtual processor is not resident. ***Note***: if the Physical ID is a spurious ID (i.e. 1023) then no physical interrupt will be generated.

It is a command error (see section 5.13.4) if:

- The device specified by "Device ID" has not been mapped to an ITT (using MAPD in section 5.13.11), or
- The "ID" exceeds the maximum value allowed by the "Size" field in the MAPD command, or
- The "Virtual ID" does not specify a valid LPI identifier (i.e. it is <8192 or exceeds the maximum supported), or
- The "Physical ID" does not specify a valid doorbell identifier (i.e. it is not 1023 and is <8192 or exceeds the maximum supported).

Note: the number of bits of "Device ID", "ID", "Virtual ID" and "Physical ID" supported by an implementation are discoverable from GITS_TYPER (see section 5.12.6). Unimplemented bits are RES0.

The pseudo-code below shows the operation of the VMAPVI command:

```
perform_vmapvi(command cmd)
    if ( DeviceOutOfRange( cmd.device ) ) then
        CommandError( VMAPVI_DEVICE_OOR );
        return;

    if ( VCPUOutOfRange( cmd.vcpu ) ) then
        CommandError( VMAPVI_VCPU_OOR );
        return;

    DTE dte = ReadDeviceTable( cmd.device );

    if ( ! dte.Valid) then
        CommandError( VMAPVI_UNMAPPED_DEVICE );
        return;

    if ( IdOutOfRange( cmd.ID, dte.ITT_size ) ) then
        CommandError( VMAPVI_ID_OOR );
        return;
```



```
if ( LPIOutOfRange( cmd.VirtualID ) ) then
    CommandError( VMAPVI_VIRTUALID_OOR );
    return;

if ( LPIOutOfRange( cmd.PhysicalID ) && cmd.PhysicalID != 1023 ) then
    CommandError( VMAPVI_PHYSICALID_OOR );
    return;

ITE ite = ReadTranslationTable( dte.ITT_base, cmd.ID );

ite.Valid = true;
ite.Type = virtual_interrupt;
ite.OutputID = cmd.VirtualID;
ite.HypervisorID = cmd.PhysicalID;
ite.VCPU = cmd.vcpu;

if (GITS_TYPER.Distributed) then
    // This invalidation is required because the INV command only invalidates
    // caches in the currently targetted re-distributor.  As a consequence,
    // when an interrupt is mapped to a re-distributor, that re-distributor
    // might still have cached data of an old configuration.
    //
    boolean success = invalidateByITE( ite );

WriteTranslationTable( dte.ITT_base, cmd.ID, ite );
IncrementReadPointer();
```


5.13.26 VINVALL VCPU

In GICv4, this command specifies that the ITS must ensure any caching associated with a specified virtual processor is consistent with the configuration and pending tables held in memory in all re-distributors.

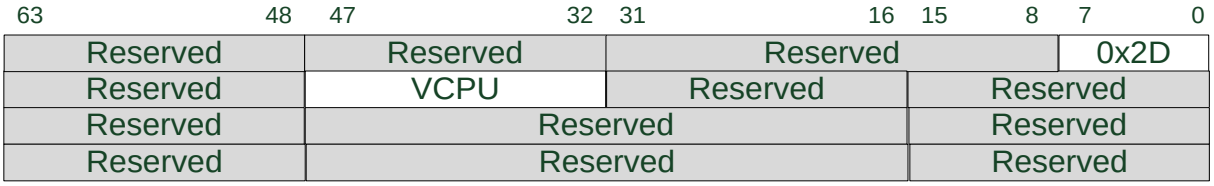


Figure 43: VINVALL command format

Where:

- "VCPU" specifies the virtual processor.

It is a command error (see section 5.13.4) if:

- The "VCPU" has not been mapped (see VMAPP in section 5.13.23)

Note: this command is expected to be used by software when it changed the re-configuration of an LPI in memory to ensure any cached copies of the old configuration are discarded. See section 4.8.11.

The pseudo-code below shows the operation of the VINVALL command:

```
perform_vinvall(command cmd)
    if ( VCPUOutOfRange( cmd.vcpu ) ) then
        CommandError( VINVALL_VCPU_OOR );
        return;

    VTE vte = ReadVCPUTable( cmd.vcpu );

    if ( ! vte.Valid ) then
        // Monolithic implementations might not use the VTE data and might
        // choose not to implement this check
        CommandError( VINVALL_VCPU_INVALID );
        return;


    if ( GITS_TYPER.Distributed ) then
        Address rd1 = vte.TargetAddress;
        Address vpt = vte.VPT_base;

        WriteGICR_VINVALLR( rd1, vpt );

    IncrementReadPointer();
```


5.13.27 VSYNC VCPU

In GICv4, this command specifies that all actions for the virtual processor prior to this command must be completed for the specified virtual processor.

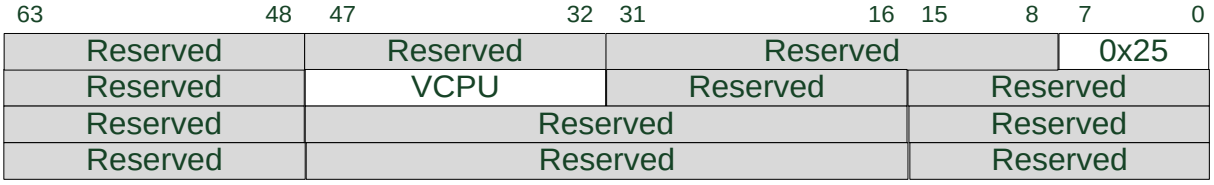


Figure 44: VSYNC command format

Where:

- "VCPU" specifies the target virtual processor for which commands must be synchronised.

It is a command error (see section 5.13.4) if:

- The "VCPU" has not been mapped (see VMAPP in section 5.13.23)

The pseudo-code below shows the operation of the VSYNC command:

```
perform_vsync(command cmd)
    if ( VCPUOutOfRange( cmd.vcpu ) ) then
        CommandError( VSYNC_VCPU_OOR );
        return;

    VTE vte = ReadVCPUTable( cmd.vcpu );

    if ( ! vte.Valid ) then
        CommandError( VSYNC_VCPU_INVALID );
        return;

    Address rd1 = vte.TargetAddress;

    // Wait for the external effects of any virtual commands to be observable
    // by all re-distributors and ensure the internal effects of any previous
    // virtual commands affect any subsequent virtual interrupt requests or commands
    WaitForVirtualCompletion( rd1 );

    if ( GITS_TYPER.Distributed ) then
        // It is implementation defined whether a re-distributor stalls until
        // completion of a write to GICR_VSYNCR or returns 0x0000_0001 until
        // the VSYNC is complete.
        //
        while ( ReadGICR_VSYNCR( rd1 ) != 0)
            ; // Do nothing

    IncrementReadPointer();
```


5.13.28 VMOVP VCPU, Target Address

In GICv4, this command specifies that interrupts for the specified virtual processor must be sent to the specified re-distributor.

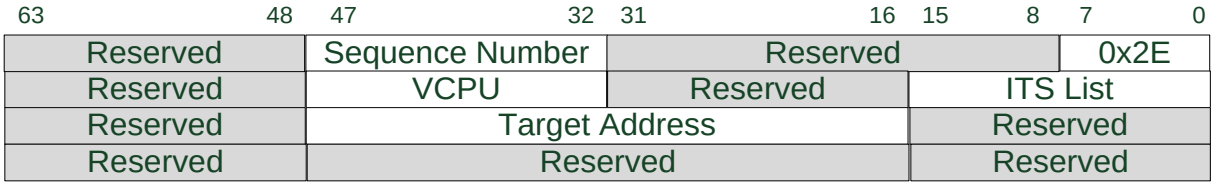


Figure 45: VMOVP command format

Where:

- "VCPU" specifies the virtual processor to be re-directed for the device specified by "Device ID"
- "Target Address" specifies the new target re-distributor.
- "Sequence Number" specifies the identity of the synchronization point that will be used by all ITS included in the "ITS List"
- "ITS List" specifies which ITS are included in the synchronization operation. Each bit of "ITS List" corresponds to an ITS where it [0] of "ITS List" corresponds to ITS 0, bit [1] to ITS [1] and so on. An ITS is included if its bit is one.

It is a command error (see section 5.13.4) if:

- The "VCPU" has not been mapped (see VMAPP in section 5.13.23)

The pseudo-code below shows the operation of the VMAPP command:

```
perform_vmovp(command cmd)
    if ( VCPUOutOfRange( cmd.vcpu ) ) then
        CommandError( VMOVP_VCPU_OOR );
        return;

    VTE vte = ReadVCPUTable( cmd.vcpu );

    if ( ! vte.Valid ) then
        CommandError( VMOVP_VCPU_INVALID );
        return;

    if ( GITS_TYPER.Distributed ) then
        Address rd1 = vte.TargetAddress;
        Address rd2 = cmd.TargetAddress;
        Address vpt = vte.VPT_base;

        if ( SpeculativeCachingSupported() ) then
            // The redistributors support speculative caching and an invalidate
            // must be issued to rd2.  Any subsequent LPIs as a consequence of
            // writes to GITS_TRANSLATER must occur after this.
            WriteGICR_VINVALLR( rd2, cmd.vcpu );

        // This invalidation causes the pending state to be made consistent
        // with external memory.
        WriteGICR_VINVALLR( rd1, vpt );

        // Wait for invalidation to complete
        // It is implementation defined whether a re-distributor stalls until
        // completion of a read of GICR_VSYNCR or returns 0x0000_0001 until
        // the VSYNC is complete.
```



```
        //
        while ( ReadGICR_VSYNCR( rd1 ) != 0)
            ; // Do nothing

        // This invalidation is required because the INV command only invalidates
        // caches in the currently targetted re-distributor.  As a consequence,
        // when an interrupt is mapped to a re-distributor, that re-distributor
        // might still have cached data of an old configuration.
        //
        WriteGICR_VINVALLR( rd2, vpt );

        if ( cmd.ITS_List != '0') then
            // Write to the re-distributor ITS synchronization register.
            WriteGICR_ITSSYNCRn( rd1, cmd.Sequence_Number, cmd.ITS_List, GITS_TYPER.ITS_NUMBER );

            // Wait for all included ITS to reach this point
            // As each ITS reaches the sync point, its bit in ITS List will become zero.
            // It is implementation defined whether a re-distributor stalls until
            // completion of a read of GICR_ITSSYNCRn or returns valid read data until
            // the ITS List field is zero.
            //
            //
            while ( ReadGICR_ITSSYNCRn( rd1 ).ITS_List != 0)
                ; // Do nothing


    vte.TargetAddress = cmd.TargetAddress;

    WriteVCPUTable( cmd.vcpu, vte );

    IncrementReadPointer();
```


5.13.29 VMOVI device, ID, VCPU

In GICv4, this command specifies that interrupts with the specified identifier for the device must be sent to the specified virtual processor.

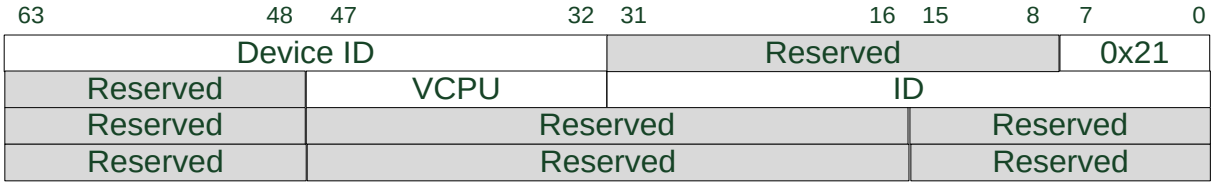


Figure 46: VMOVI command format

Where

- "ID" specifies the interrupt to be re-directed for the device specified by "Device ID"
- "VCPU" specifies the new target virtual processor.

It is a command error (see section 5.13.4) if:

- The device specified by "Device ID" has not been mapped to an ITT (using MAPD in section 5.13.11), or
- The "ID" exceeds the maximum value allowed by the "Size" field in the MAPD command, or
- The "ID" corresponds to a physical LPI, or
- The "VCPU" has not been mapped (see VMAPP in section 5.13.23)

Note: the number of bits of "Device ID" and "ID" supported by an implementation are discoverable from GITS_TYPER (see section 5.12.6). Unimplemented bits are RES0.

Note: if software wishes to move an interrupt from virtual processor A to B and then from B to C it must issue a "SYNC" command to the virtual processor A between these moves to ensure correct behaviour. Otherwise, the interrupt might still occur on virtual processor B even after completion of the second MOVI command.

The pseudo-code below shows the operation of the VMOVI command:

```
perform_vmovi(command cmd)
    if ( DeviceOutOfRange( cmd.device ) ) then
        CommandError( VMOVI_DEVICE_OOR );
        return;

    if ( VCPUOutOfRange( cmd.vcpu ) ) then
        CommandError( VMOVI_COLLECTION_OOR );
        return;

    DTE dte = ReadDeviceTable( cmd.device );

    if ( ! dte.Valid) then
        IncrementReadPointer();
        return;

    if (IdOutOfRange( cmd.ID, dte.ITT_size ) ) then
        CommandError( VMOVI_ID_OOR );
        return;

    ITE ite = ReadTranslationTable( dte.ITT_base, cmd.ID );

    if ( ! ite.Valid ) then
        CommandError( VMOVI_UNMAPPED_INTERRUPT );
        return;
```



```
if ( ite.Type == physical_interrupt ) then
    CommandError( VMOVI_ID_IS_PHYSICAL );
    return;

VTE vte1 = ReadVCPUTable( ite.vcpu );
VTE vte2 = ReadVCPUTable( cmd.vcpu );

if ( ! vte1.Valid ) then
    CommandError( VMOVI_ITEVCPU_INVALID );
    return;

if ( ! vte2.Valid ) then
    CommandError( VMOVI_CMDVCPU_INVALID );
    return;

// Start sending new interrupts to the new VCPU
RetargetVirtualInterrupt( cmd.device, cmd.ID, cmd.vcpu );

Address rd1 = vte1.TargetAddress;
Address vpt1 = vte1.VPT_base;
Address rd2 = vte2.TargetAddress;
Address vpt2 = vte2.VPT_base;

ite.VCPU = cmd.vcpu;

if ( GITS_TYPER.Distributed && vpt1 != vpt2 ) then
    if ( SpeculativeCachingSupported() ) then
        // The redistributors support speculative caching and an invalidate
        // must be issued to rd2.  Any subsequent LPIs as a consequence of
        // writes to GITS_TRANSLATER must occur after this.
        WriteGICR_VINVLPIR( rd2, vpt, ite.OutputID );

    // This invalidation is required because the INV command only invalidates
    // caches in the currently targetted re-distributor.  As a consequence,
    // when an interrupt is mapped to a re-distributor, that re-distributor
    // might still have cached data of an old configuration.
    //
    boolean success = invalidateByITE( ite );

    // Issue commands to rd1 to move the pending state to rd2 if pending
    // Note: rd1 performs the equivalent of MoveVirtualPendingState
    //
    // Note: the ITS must ensure the following sequence is performed
    // with no intervening writes to the same re-distributor by this ITS
    WriteGICR_VSRCRn( rd1, vpt1, GITS_TYPER.ITS_NUMBER );
    WriteGICR_VDESTRn( rd1, vpt2, GITS_TYPER.ITS_NUMBER );

    if ( GITS_TYPER.IDbits >= 16 ) then
        // Interrupt identifiers are > 16 bits so the re-distributor
        // will write to GICR_VSETLPILRn not GICR_VSETLPIR
        WriteGICR_VDESTRn( rd2, vpt2, ite.OutputID, GITS_TYPER.ITS_NUMBER );

    WriteGICR_VMOVLPILRn( rd1, rd2, ite.OutputID, GITS_TYPER.ITS_NUMBER );
else
    // Move the move the pending state to rd2 if set taking care of
    // any races where the interrupt has been forwarded to the processor
    MoveVirtualPendingState( rd1, vpt1, vpt2, ite.OutputID );
```



```
WriteTranslationTable( dte.ITT_base, cmd.ID, ite );
IncrementReadPointer();
```


6 SYSTEM REGISTER ACCESS

For performance, the CPU interface registers, the Hypervisor control registers and the Distributor registers that are banked by CPU interface have been mapped to system registers.

6.1 AArch32 and AArch64

The GIC registers will only be mapped to system registers in both AArch32 and AArch64. Generally, use of MRRC / MCRR in AArch32 has been avoided due to the extremely limited number of encodings available, and only ICC_SGI0R_EL1, ICC_SGI1R_EL1 and ICC_ASGI1R_EL1 use MCRR.

When EL3 is using AArch32, all AArch32 non-Monitor, secure privileged modes are treated as AArch64 Secure EL1 accesses. AArch32 Monitor mode accesses are treated as AArch64 EL3 accesses.

6.2 Undefined and Trap Behaviour

When attempting to access the GIC system registers, exceptions might be generated under some circumstances, as described below:

Exception name	Exception Priority	Description
UNDEF_und	7	An access was performed to a system register that is always undefined (e.g. EL0 access to ICC_PMR_EL1). ***Note***: this exception might also be generated for accesses to system registers where an implementation defined number might be implemented (see sections 5.7.41, 5.7.42, 5.9.9, 5.9.10 and 5.9.6).
UNDEF_cfg	8, 12, 18	An access was performed to a system register where access to that register at the current exception level is prohibited by the setting of a configuration register (e.g. EL1 access to ICC_SRE_EL1 when ICC_SRE_EL2.Enable is zero). The priority depends on the exception level to which the exception will be taken.
HYP_TRAP	11	An access was performed to a system register where access to that register at the current exception level will result in a Hyp trap to EL2.
MON_TRAP	17	An access was performed to a system register where access to that register at the current exception level will result in a Mon trap to EL3. ***Note***: when EL3 is configured as AArch32, MON_TRAP is replaced by UNDEF_cfg

Table 38: GIC Exception Behaviour

See ref [4] section 3.5.2.4 for further details of ARMv8 synchronous exception prioritization.

6.3 CPU interface System Register Access

The table below shows the system register mappings for the CPU interface registers and the state accessed:

System Register Name		Register or pseudocode function accessed	
AArch64	AArch32	Physical access	Virtual access
ICC_IAR1_EL1	ICC_IAR1	ReadICC_IAR1_EL1	ReadICV_IAR1_EL1
ICC_IAR0_EL1	ICC_IAR0	ReadICC_IAR0_EL1	ReadICV_IAR0_EL1
ICC_EOIR1_EL1	ICC_EOIR1	WriteICC_EOIR1_EL1	WriteICV_EOIR1_EL1
ICC_EOIR0_EL1	ICC_EOIR0	WriteICC_EOIR0_EL1	WriteICV_EOIR0_EL1
ICC_HPPIR1_EL1	ICC_HPPIR1	ReadICC_HPPIR1_EL1	ReadICV_HPPIR1_EL1
ICC_HPPIR0_EL1	ICC_HPPIR0	ReadICC_HPPIR0_EL1	ReadICV_HPPIR0_EL1
ICC_BPR1_EL1	ICC_BPR1	ICC_BPR1_EL1	ICH_VMCR_EL2.VBPR1
ICC_BPR0_EL1	ICC_BPR0	ICC_BPR0_EL1	ICH_VMCR_EL2.VBPR0
ICC_DIR_EL1	ICC_DIR	WriteICC_DIR_EL1	WriteICV_DIR_EL1
ICC_PMR_EL1	ICC_PMR	ReadICC_PMR_EL1	ICH_VMCR_EL2.PMR
ICC_RPR_EL1	ICC_RPR	ReadICC_RPR_EL1	ReadICV_RPR_EL1
ICC_AP0R0_EL1	ICC_AP0R0	ICC_AP0R0_EL1	ICH_AP0R0_EL2
ICC_AP0R1_EL1	ICC_AP0R1	ICC_AP0R1_EL1	ICH_AP0R1_EL2
ICC_AP0R2_EL1	ICC_AP0R2	ICC_AP0R2_EL1	ICH_AP0R2_EL2
ICC_AP0R3_EL1	ICC_AP0R3	ICC_AP0R3_EL1	ICH_AP0R3_EL2
ICC_AP1R0_EL1	ICC_AP1R0	ICC_AP1R0_EL1	ICH_AP1R0_EL2
ICC_AP1R1_EL1	ICC_AP1R1	ICC_AP1R1_EL1	ICH_AP1R1_EL2
ICC_AP1R2_EL1	ICC_AP1R2	ICC_AP1R2_EL1	ICH_AP1R2_EL2
ICC_AP1R3_EL1	ICC_AP1R3	ICC_AP1R3_EL1	ICH_AP1R3_EL2
ICC_IGRPEN0_EL1	ICC_IGRPEN0	ICC_IGRPEN0_EL1	ICH_VMCR_EL2.VENG0
ICC_IGRPEN1_EL1	ICC_IGRPEN1	ICC_IGRPEN1_EL1	ICH_VMCR_EL2.VENG1
ICC_IGRPEN1_EL3	ICC_IGRPEN3	ICC_IGRPEN1_EL3	Not allocated; UNDEF_und exception
ICC_SRE_EL1	ICC_SRE	ICC_SRE_EL1	See section 5.7.38.
ICC_SRE_EL2	ICC_HSRE	ICC_SRE_EL2	Not allocated; UNDEF_und exception
ICC_SRE_EL3	ICC_MSRE	ICC_SRE_EL3	
ICC_CTLR_EL1	ICC_CTLR	ICC_CTLR_EL1	ICH_VMCR_EL2.VEOIM and ICH_VMCR_EL2.VCBPR
ICC_CTLR_EL3	ICC_MCTLR	ICC_CTLR_EL3	Not allocated; UNDEF_und exception
ICC_SGI0R_EL1	ICC_SGI0R	ICC_SGI0R_EL1	Trap to EL2; HYP_TRAP
ICC_SGI1R_EL1	ICC_SGI1R	ICC_SGI1R_EL1	
ICC_ASGI1R_EL1	ICC_ASGI1R	ICC_ASGI1R_EL1	

Table 39: CPU Interface System Register State Accessed

Where:

- Physical and Virtual access as defined in Table 17: CPU Interface System Register Access Behaviour.
- The physical and virtual access columns list the register description or pseudocode function used.
- HYP_TRAP and UNDEF_und are as defined in Table 38: GIC Exception Behaviour above.
- HYP_TRAP might also be generated if ICH_HCR_EL2.TALL{0,1} or ICH_HCR_EL2.TC is set (see section 5.9.2).

6.4 Hypervisor System Register Access

The table below shows the system register mapping for the hypervisor control registers and how these are affected by Exception Level:

System Register Name		Register or pseudocode function accessed	
AArch64	AArch32	EL3, EL2	EL1
ICH_HCR_EL2	ICH_HCR	ICH_HCR_EL2	Not allocated; UNDEF_und exception
ICH_VTR_EL2	ICH_VTR	ICH_VTR_EL2	
ICH_VMCR_EL2	ICH_VMCR	ICH_VMCT_EL2	
ICH_MISR_EL2	ICH_MISR	ICH_MISR_EL2	
ICH_EISR_EL2	ICH_EISR	ICH_EISR_EL2	
ICH_ELSR_EL2	ICH_ELSR	ICH_ELSR_EL2	
ICH_AP0R0_EL2	ICH_AP0R0	ICH_AP0R0_EL2	
ICH_AP0R1_EL2	ICH_AP0R1	ICH_AP0R1_EL2	
ICH_AP0R2_EL2	ICH_AP0R2	ICH_AP0R2_EL2	
ICH_AP0R3_EL2	ICH_AP0R3	ICH_AP0R3_EL2	
ICH_AP1R0_EL2	ICH_AP1R0	ICH_AP1R0_EL2	
ICH_AP1R1_EL2	ICH_AP1R1	ICH_AP1R1_EL2	
ICH_AP1R2_EL2	ICH_AP1R2	ICH_AP1R2_EL2	
ICH_AP1R3_EL2	ICH_AP1R3	ICH_AP1R3_EL2	
ICH_LR{0:15}_EL2	ICH_LR[0:15]	GICH_LR{0 to 15}_EL2	
	ICH_LRC[0:15]	GICH_LR{0 to 15}_EL2	

Table 40: Hypervisor System Register State Accessed

Where:

- UNDEF_und is as defined in Table 38: GIC Exception Behaviour above.
- The EL3, EL2 column lists the register description appropriate to the access.

6.5 System Register Assignments

The GIC generally uses MRC and MCR due to the extremely limited space available for MRRC and MCRR. The following assignments have been designed to be easily separable and generally use currently unassigned CP15, CRn == c12 space. For AArch64, the usual convention of using the Op1 encoding to indicate lowest EL allowed to access the register has been followed (EL1 == 0, EL2 == 4, EL3 == 6).

The “Access” column provides detail of whether each register is read-only (RO), write-only (WO) or read-write (RW). Where a register is marked as “B*” then the register remains banked in AArch64.

Notes:

- When operating in EL3, accesses to banked EL1 registers access the copy appropriate to the current value of SCR_EL3.NS
- When operating in EL3 and EL3 is configured as AArch32, accesses to EL2 registers will result in an UNDEF_und trap if SCR_EL3.NS == ‘0’.

6.5.1 Read-Only and Write-Only System Register Accesses

If a read-only system register is written, this will result in an Undefined Exception (UNDEF_und as defined in Table 38: GIC Exception Behaviour above).

Similarly for write-only system registers, if such a register is read, this will result in an Undefined Exception (UNDEF_und as defined in Table 38: GIC Exception Behaviour above).

6.5.2 System Registers with Special Behaviour

The following registers have been assigned in CP15, CRn == 4 to ensure the effects of writes to these registers are visible without any explicit synchronization operations:

- ICC_PMR_EL1. No synchronization required to ensure no interrupts with priority below the PMR value will be taken.

See also section 3.10.3 of [4].

6.5.3 CPU Interface Registers

System Register			AArch32				AArch64				
AArch64	AArch32	Access	CRn	Op1	CRm	Op2	Op0	CRn	Op1	CRm	Op2
ICC_IAR1_EL1	ICC_IAR1	RO	c12	0	c12	0	3	c12	0	c12	0
ICC_IAR0_EL1	ICC_IAR0	RO	c12	0	c8	0	3	c12	0	c8	0
ICC_EOIR1_EL1	ICC_EOIR1	WO	c12	0	c12	1	3	c12	0	c12	1
ICC_EOIR0_EL1	ICC_EOIR0	WO	c12	0	c8	1	3	c12	0	c8	1
ICC_HPPIR1_EL1	ICC_HPPIR1	RO	c12	0	c12	2	3	c12	0	c12	2
ICC_HPPIR0_EL1	ICC_HPPIR0	RO	c12	0	c8	2	3	c12	0	c8	2
ICC_BPR1_EL1	ICC_BPR1	RW, B*	c12	0	c12	3	3	c12	0	c12	3
ICC_BPR0_EL1	ICC_BPR0	RW	c12	0	c8	3	3	c12	0	c8	3
ICC_DIR_EL1	ICC_DIR	WO	c12	0	c11	1	3	c12	0	c11	1
ICC_PMR_EL1	ICC_PMR	RW	c4	0	c6	0	3	c4	0	c6	0
ICC_RPR_EL1	ICC_RPR	RO	c12	0	c11	3	3	c12	0	c11	3
ICC_CTLR_EL1	ICC_CTLR	RW, B*	c12	0	c12	4	3	c12	0	c12	4
ICC_CTLR_EL3	ICC_MCTLR	RW	c12	6	c12	4	3	c12	6	c12	4
ICC_SRE_EL1	ICC_SRE	RW, B*	c12	0	c12	5	3	c12	0	c12	5
ICC_SRE_EL2	ICC_HSRE	RW	c12	4	c9	5	3	c12	4	c9	5
ICC_SRE_EL3	ICC_MSRE	RW	c12	6	c12	5	3	c12	6	c12	5
ICC_IGRPEN0_EL1	ICC_IGRPEN0	RW	c12	0	c12	6	3	c12	0	c12	6
ICC_IGRPEN1_EL1	ICC_IGRPEN1	RW, B*	c12	0	c12	7	3	c12	0	c12	7
ICC_IGRPEN1_EL3	ICC_MGRPEN1	RW	c12	6	c12	7	3	c12	6	c12	7
ICC_SGI1R_EL1	ICC_SGI1R	WO	-	0	c12	-	3	c12	0	c11	5
ICC_ASGI1R_EL1	ICC_ASGI1R	WO	-	1	c12	-	3	c12	0	c11	6
ICC_SGI0R_EL1	ICC_SGI0R	WO	-	2	c12	-	3	c12	0	c11	7

Table 41: CPU Interface System Register Assignment

Notes:

- When operating in EL3, accesses to banked EL1 registers access the copy appropriate to the current value of SCR_EL3.NS. When EL3 is using AArch32, there is no Secure EL1 interrupt regime and accesses in any Secure EL3 mode except Mon access the secure copy.
- ICC_SGI{0,1}R_EL1 and ICC_ASGI1R_EL1 accessible using MCRR in AArch32
- No synchronization is required when changing the PMR value, hence ICC_PMR_EL1 is in CRn = 4

6.5.4 Active Priorities Registers

System Register			AArch32				AArch64				
AArch64	AArch32	Access	CRn	Op1	CRm	Op2	Op0	CRn	Op1	CRm	Op2
ICC_AP0R0_EL1	ICC_AP0R0	RW	c12	0	c8	4	3	c12	0	c8	4
ICC_AP0R1_EL1	ICC_AP0R1	RW	c12	0	c8	5	3	c12	0	c8	5
ICC_AP0R2_EL1	ICC_AP0R2	RW	c12	0	c8	6	3	c12	0	c8	6
ICC_AP0R3_EL1	ICC_AP0R3	RW	c12	0	c8	7	3	c12	0	c8	7
ICC_AP1R0_EL1	ICC_AP1R0	RW, B*	c12	0	c9	0	3	c12	0	c9	0
ICC_AP1R1_EL1	ICC_AP1R1	RW, B*	c12	0	c9	1	3	c12	0	c9	1
ICC_AP1R2_EL1	ICC_AP1R2	RW, B*	c12	0	c9	2	3	c12	0	c9	2
ICC_AP1R3_EL1	ICC_AP1R3	RW, B*	c12	0	c9	3	3	c12	0	c9	3
ICH_AP0R0_EL2	ICH_AP0R0	RW	c12	4	c8	0	3	c12	4	c8	0
ICH_AP0R1_EL2	ICH_AP0R1	RW	c12	4	c8	1	3	c12	4	c8	1
ICH_AP0R2_EL2	ICH_AP0R2	RW	c12	4	c8	2	3	c12	4	c8	2
ICH_AP0R3_EL2	ICH_AP0R3	RW	c12	4	c8	3	3	c12	4	c8	3
ICH_AP1R0_EL2	ICH_AP1R0	RW	c12	4	c9	0	3	c12	4	c9	0
ICH_AP1R1_EL2	ICH_AP1R1	RW	c12	4	c9	1	3	c12	4	c9	1
ICH_AP1R2_EL2	ICH_AP1R2	RW	c12	4	c9	2	3	c12	4	c9	2
ICH_AP1R3_EL2	ICH_AP1R3	RW	c12	4	c9	3	3	c12	4	c9	3

Table 42: APR System Register Assignment

6.5.5 Hypervisor Control Registers

The hypervisor control registers are accessible at EL3 when ICC_SRE_EL3.SRE == '1' and:

- EL3 is configured as AArch32, access is only permitted if SCR_EL3.NS == ‘1’ otherwise an UNDEF_und exception will be generated (see section 6.2)
- EL3 is configured as AArch64, access is always permitted.

System Register			AArch32				AArch64				
AArch64	AArch32		CRn	Op1	CRm	Op2	Op0	CRn	Op1	CRm	Op2
ICH_HCR_EL2	ICH_HCR	RW	c12	4	c11	0	3	c12	4	c11	0
ICH_VTR_EL2	ICH_VTR	RO	c12	4	c11	1	3	c12	4	c11	1
ICH_MISR_EL2	ICH_MISR	RO*	c12	4	c11	2	3	c12	4	c11	2
ICH_EISR_EL2	ICH_EISR	RO	c12	4	c11	3	3	c12	4	c11	3
ICH_ELSR_EL2	ICH_ELSR	RO	c12	4	c11	5	3	c12	4	c11	5
ICH_VMCR_EL2	ICH_VMCR	RW	c12	4	c11	7	3	c12	4	c11	7
ICH_LR{0..15}_EL2	ICH_LR{0..15}	RW	c12	4	{c12,c13}	0 - 7	3	c12	4	{c12, c13}	0 - 7
	ICH_LRC{0..15}	RW	c12	4	{c14,c15}	0 - 7					

Table 43: Hypervisor system Register Assignment

***Notes*:**

- When operating in EL3 and EL3 is configured as AArch32, accesses to EL2 registers will result in an UNDEF_und trap if SCR_EL3.NS == ‘0’.
- List Registers ICH_LR{0..7}_EL2 use CRm == c12 and List Registers ICH_LR{8..15}_EL2 use CRm == c13.
- In AArch32, ICH_LR{0..15} are 32 bit values and the other half is obtained using ICH_LRC{0..15}.
- In AArch64, ICH_LR{0..15}_EL2 are 64 bit values.
- In systems that support generation of SEIs by the CPU interface logic, ICH_MISR_EL2 is RW.

7 DISTRIBUTOR TO CPU INTERFACE

Because GICv3 is intended to support large numbers of processors, many implementations will separate the Distributor from the logic associated with the CPU interface. To support such implementations, this document specifies a standard interface between the Distributor and the CPU interface logic referred to as “The GIC Stream Protocol Interface”.

Additionally, because GICv3 supports system register access to CPU interface logic, it is expected that future CPU implementations will include the GIC CPU interface logic.

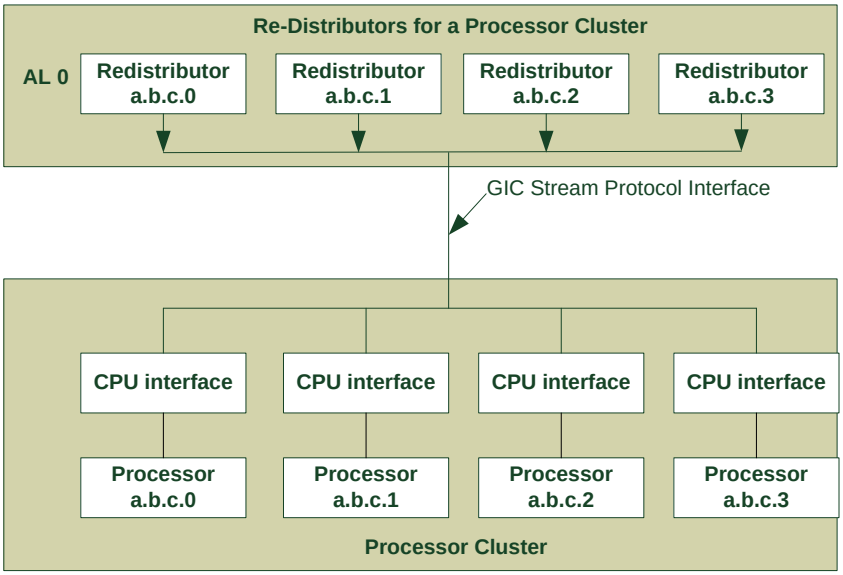


Figure 47: The GIC Stream Protocol Interface

Note: multiple Affinity Level 0 Re-Distributors may use a single multiplexed interface.

7.1 Signaling

The GIC Stream Protocol Interface is intended to be simple and comprises an AMBA 4 AXI4 Stream Protocol interface (see [3]) in each direction. The data on these interfaces is transferred as a sequence of packets, defined in the sections that follow. The interface comprises the following signals in each direction:

- TVALID
- TREADY
- TDATA
- TLAST

In addition, a global clock (ACLK) and reset (ARESETn) are required.

The width of TDATA is implementation defined (but must be an integral number of bytes).

Note: in systems where the processor and re-distributors implement a different width, use of the TSTRB and/or TKEEP signals might also be required.

7.1.1 Expected Interface Pins

The following pins are expected for each cluster of processors that supports a single GIC Stream Protocol Interface:

Processor Signal	Input / Output	Description
nFIQ[CN:0]	Input	Fast interrupt request (active-low). This signal is only used when: • When FIQ is in bypass mode when it is used as "Legacy FIQ"
nIRQ[CN:0]	Input	Interrupt request (active-low). This signal is only used when: • When IRQ is in bypass mode when it is used as "Legacy IRQ"
nSEI[CN:0]	Input	System Error Interrupt request (active-low; edge-sensitive). This signal is to signal that an external system error interrupt is to be generated. This signal is used to: • In ARM implementations, report Asynchronous External Aborts (see section 4.7.3).
nREI[CN:0]	Input	Optional RAM Error SEI request (active-low; edge-sensitive). This signal is to signal that an external system error interrupt is to be generated with a syndrome indicating the reason is a RAM error. See section 4.7.3. ***Note***: this signal is expected in all ARM implementations.
nVCPUMNTIRQ[CN:0]	Output	External virtual CPU interface maintenance interrupt request. This is used when the maintenance interrupt must be reported to an external Distributor.
GIC Stream Protocol Signal	Input / Output	Description
ICDTVALID	Input	AXI4 Stream Protocol signal. Distributor to GIC CPU Interface packets. **TVALID** indicates that the master is driving a valid transfer.
ICDTREADY	Output	AXI4 Stream Protocol signal. Distributer to GIC CPU Interface packets. **TREADY** indicates that the slave can accept a transfer in the current cycle.
ICDTDATA[BN:0]	Input	AXI4 Stream Protocol signal. Distributer to GIC CPU Interface packets. **TDATA** is the primary payload that is used to provide the data that is passing across the interface.
ICDTLAST	Input	AXI4 Stream Protocol signal. Distributer to GIC CPU Interface packets. **TLAST** indicates the boundary of a packet.
ICDTDEST[N:0]	Input	AXI4 Stream Protocol signal. Distributer to GIC CPU Interface packets. **TDEST** provides routing information for the data stream.
ICCTVALID	Output	AXI4 Stream Protocol signal. GIC CPU Interface to distributer packets. **TVALID** indicates that the master is driving a valid transfer.

ICCTREADY	Input	AXI4 Stream Protocol signal. GIC CPU Interface to distributer packets. **TREADY** indicates that the slave can accept a transfer in the current cycle.
ICCTDATA[BN:0]	Output	AXI4 Stream Protocol signal. GIC CPU Interface to Distributer packets. **TDATA** is the primary payload that is used to provide the data that is passing across the interface
ICCTLAST	Output	AXI4 Stream Protocol signal. GIC CPU Interface to Distributer packets. **TLAST** indicates the boundary of a packet
ICCTID[N:0]	Output	AXI4 Stream Protocol signal. GIC CPU Interface to Distributer. **TID** is the data stream identifier that indicates different streams of data.

Table 44: Interface Pins Required for a Processor Cluster

Where:

- CN is the number of processors minus one
- BN:0 is the smallest bit range that supports the implementation defined AXI4 Stream Interface width.
- N:0 is the smallest bit range that can encode a value of CN+1

Note: current ARM implementations use a width of two bytes and no byte strobes are necessary. If other widths are chosen, byte strobes may be required.

7.1.2 Cluster Addressing

An implementation may optionally provide a single interface for a cluster of processors. In such cases, the following additional signals defined in the AXI4-Stream Protocol must be provided:

- The interface from the Distributor to the cluster must include the TDEST signals. This is used to indicate the destination processor of a packet sent to the cluster.
- The interface to the Distributor from the cluster must include the TID signals. This is used to indicate the source processor of a packet received from the cluster.

The width of these signals is determined by the number of elements in the Affinity 0 locality: if this can contain ***N*** processors, then ***M*** bits must be provided, where ***M*** is the smallest integer such that $2^{M} \geq N$.

7.1.3 PPI Mapping

Some PPIs are reserved for ARM use. These PPIs are defined for the A-profile as shown in the table below.

Interrupt ID	Physical pin	Description
30	nCNTPNSIRQ	Non-secure physical timer interrupt
29	nCNTPSIRQ	Secure Physical timer interrupt
27	nCNTVIRQ	Virtual timer interrupt
26	nCNTHPIRQ	Hypervisor timer interrupt
25	nVCPUMNTIRQ (output)	Virtual CPU interface Maintenance interrupt
24	nCTIIRQ	CTI (Cross Trigger Interface) interrupt
23	nPMUIRQ	PMU (performance counter) overflow
22	nDCCIRQ	DCC (comms channel) interrupt

Table 45: PPI Mapping

Note: These pins are all outputs from each CPU and inputs to the Distributor as dedicated signals. Anywhere indicated is a direct output from the CPU interface logic.

It is implementation defined whether the remaining PPIs (IDs 16 – 21, 28, and 31) are made available for other purposes.

7.2 Packets and Protocol

Each packet comprises an initial command byte followed by zero or more data bytes.

The protocol comprises several (semi-)independent streams of packets:

- **Physical Interrupts**. Packets: Set, Deactivate, Deactivate Acknowledge, Clear, Clear Acknowledge (with V == 0), Release, Activate
- **Virtual Interrupts**: VSet, VClear, Clear Acknowledge (with V==1), Release (with V == 1), Activate (with V == 1)
- **Software Generated Interrupts**: Generate SGI, Generate SGI Acknowledge
- **Control**. Packets involved: Quiesce, Quiesce Acknowledge, Upstream Write, Upstream Write Acknowledge, Downstream Write, Downstream Write Acknowledge

The CPU interface logic and the Distributor must only transmit defined packets across the GIC Stream Protocol Interface. In addition, to ensure that the interface is deadlock free, the rules outlined below must be followed.

7.2.1 Distributor rules

The Distributor must always be able to sink the following packets:

- **Interrupt**: Release
- **Control**: Downstream Write Acknowledge

The Distributor has special rules for generating:

- **Set:** the Distributor can have at most two “Set” packets outstanding per processor at any time that are awaiting a response (an “Activate” or a “Release”). In addition, the Distributor may only issue a “Set” packet if it is able to sink an “Activate” packet with V == ‘0’.
- **VSet:** the Distributor can have at most two “VSet” packets outstanding per processor at any time that are awaiting a response (an “Activate” or a “Release”). In addition, the Distributor may only issue a “VSet” packet if it is able to sink an “Activate” packet with V == ‘1’.
- **Clear**: the Distributor may only issue a single Clear to the CPU interface and must wait for a Clear Acknowledge (with V == 0) before another may be issued. While a Clear is outstanding, no further “Physical Interrupt” packets except acknowledgements can be issued to the CPU interface
- **VClear (GICv4)**: the Distributor may only issue a single VClear to the CPU interface and must wait for a Clear Acknowledge (with V == 1) before another may be issued. While a Clear is outstanding, no further “Virtual Interrupt” packets can be issued to the CPU interface
- **Quiesce**: the Distributor may only issue a single Quiesce to the CPU interface and must wait for a Quiesce Acknowledge before another may be issued. While a Quiesce is outstanding, no further packets can be issued to the CPU interface (except for acknowledgements) but the Distributor must be able to sink packets from the CPU interface as normal
- **Downstream Write**: the Distributor may only issue a single Downstream Write to the CPU interface and must wait for a Downstream Write Acknowledge before another may be issued. While a Downstream Write is outstanding, no further packets can be issued to the CPU interface (except for acknowledgements) but the Distributor must be able to sink packets from the CPU interface as normal including an Upstream Write.

The Distributor must also ensure that:

- The first packet issued to each CPU interface (that is not an Acknowledge packet) after a Distributor (or re-distributor) reset or after power-on of the associated processor (see section 4.10.6) is a Downstream Write with the security and ID length settings.
- This must be sent whenever GICR_WAKER.ProcessorSleep is set to zero. This must be done regardless of whether the processor has been powered off.

Except as restricted by the special rules above the Distributor can always generate the following packets:

- **Physical Interrupts**: Set (up to two)
- **Virtual Interrupts (GICv4)**: VSet (up to two)

7.2.2 CPU interface rules

The CPU interface must always be able to sink the following packets:

- **Physical Interrupts**: Set, Activate Acknowledge, Deactivate Acknowledge, Upstream Write Acknowledge
- **Virtual Interrupts (GICv4)**: VSet
- **Software Generated Interrupts**: Generate SGI Acknowledge

The CPU interface has special rules for:

- **Clear**: the CPU interface must issue any Release packet(s) provoked by receipt of a Clear before issuing a Clear Acknowledge (with V == 0)
- **VClear (GICv4)**: the CPU interface must issue any Release packet(s) provoked by receipt of a VClear before issuing a Clear Acknowledge (with V == 1)
- **Quiesce**: the CPU interface must issue any Release packet(s) provoked by receipt of a Quiesce and ensure any outstanding Activate packets (including receipt of the Acknowledge packet), Deactivate packets (including receipt of the Acknowledge packet), Generate SGI packets (including receipt of the Acknowledge packet), Upstream Write packets (including receipt of the Acknowledge packet), and Clear Acknowledge packets have been sent to the Distributor before issuing a Quiesce Acknowledge.
- **Activate**: the CPU interface has a separate implementation defined “Activate credit” (that resets to this implementation defined value) for physical and virtual Activate packets. Transmission of an Activate packet to the Distributor consumes a credit and receipt of an Activate Acknowledge returns a credit. The CPU interface must have non-zero credit before it can transmit an Activate packet. When the credit is zero, the CPU interface may continue to send other packets.
- **Deactivate**: the CPU interface may only issue a single Deactivate to the Distributor and must wait for an Deactivate Acknowledge before another may be issued. While a Deactivate is outstanding, the CPU interface may continue to send packets.
- **Generate SGI**: the CPU interface may only issue a single Generate SGI to the Distributor and must wait for a Generate SGI Acknowledge before another may be issued. While a Generate SGI is outstanding, the CPU interface may continue to send packets.
- **Upstream Write**: the CPU Interface may only issue a single Upstream Write to the Distributor and must wait for a Upstream Write Acknowledge before another may be issued.

Except as restricted by the special rules above, the CPU interface can always generate the following packets:

- **Physical Interrupts**: Activate, Release
- **Virtual Interrupts (GICv4)**: Activate, Release.

7.2.3 Packet Interleaving

The AXI4 Stream Interface allows implementations to limit interleaving to packet boundaries. To ensure interoperability, implementations must only interleave traffic at the packet level. ***Note***: care must be taken to ensure any fabric does not reorder transfers.

7.2.4 Packet Length

The last transfer of each packet (of BN+1 bits) is indicated by the assertion of a “last” signal:

- ICDTLAST. Indicates the last transfer of packets from the Distributor.
- ICCTLAST. Indicates the last transfer of packets from the CPU interface logic.

The Distributor and GIC CPU interface logic may support:

- 16 bit interrupt identifiers, or
- 16 bit and 24 bit interrupt identifiers

Implementations might choose not to support affinity 3 values other than zero. See section 7.4.6.

To support implementations which support both 16 bit and 24 bit interrupt identifiers a two bit “ID length” field is included in all packets which include an interrupt identifier:

- 0b00. The packet includes a 16 bit identifier.
- 0b01. The packet includes a 24 bit identifier.
- 0b10 – 0b11. Reserved. It is a protocol error if a packet uses either of these encodings.

Implementations must ensure that no excess data beyond the minimum required to communicate the supported interrupt identifier length are transmitted across the GIC Stream Protocol Interface.

As specified in section 7.2.1, the first packet sent by the distributor communicates the number of security states supported and the physical and virtual interrupt identifier lengths supported by the distributor to the CPU interface logic.

The Downstream Write Acknowledge packet from the CPU interface logic returns the minimum interrupt identifier lengths that are supported by both the distributor and the CPU interface logic. The distributor and the CPU interface logic must ensure no packet uses an ID length value that exceeds these values.

When both the distributor and the CPU interface logic support an interrupt identifier length larger than 16 bits but the value of an interrupt identifier in a particular packet can be encoded using only 16 bits, it is permitted that an implementation might send 16 bits of data with “ID length” set to 0b00.

Similarly, when a CPU interface supports the affinity 3 field but a write to an SGI generation register specifies the value zero for affinity 3, it is permitted for the resulting packet to set A3V to zero.

Note: in systems where the Distributor only supports 16 bit identifiers but the processor supports more than 16 bit identifiers, the Distributor might choose to generate a system error on receipt of a packet with an identifier with non-zero bits above bit [15]. This applies to the Activate, Release and Deactivate packets (see sections 7.4.2, 7.4.3, and 7.4.5).

7.2.5 Protocol Errors

It must not be possible for any mis-programming by software to cause a hardware protocol error that results in the processor or GIC becoming non-operational. However, errors might result in the values of fields within packets. See below.

7.2.6 Detection and Reporting of Packet Errors

If the Distributor or a processor detects an error in the value of a field in a packet, the preferred solution is that such errors should be explicitly reported, however in the short term this might not be possible for all errors and an implementation might chose instead to ignore an error and continue.

If an implementation chooses to report an error, this should be done by generating an implementation defined System Error.

7.3 Commands to the CPU Interface

The table below shows a summary of all the packets from the Distributor to a CPU interface block.

Command	ID	Parameters	Data	Description
Set	0x1	Bit[7:6]: ID length Bit[5]: GrpMod Bit[4]: Group Bit[15:8]: Priority	Interrupt ID	Set the highest priority pending interrupt to Interrupt ID. The CPU interface block has control of the interrupt and reads of the IAR may immediately return Interrupt ID (and send an Activate to the Distributor). If the CPU interface block cannot handle the interrupt because the interrupt group is

				disabled, it will issue a Release packet for Interrupt ID.
Clear	0x3	Bit[7:6]: ID length	Interrupt ID	Clears the interrupt specifies by Interrupt ID if it is pending. The CPU interface must generate a Clear Acknowledge with “V” set to zero.
Quiesce	0x4	-	-	Request the CPU interface to enter the quiescent state. The CPU interface must acknowledge any outstanding Downstream Writes and must release any pending interrupts. The CPU interface must issue a Quiesce Acknowledge once the interface has entered the quiescent state.
VSet	0x6	Bit[7:6]: ID length Bit[4]: Group Bit[15:8]: Priority	Virtual ID	Set the virtual LPI specified by Virtual ID as pending for the guest operating system.
VClear	0x7	Bit[7:6]: ID length	Virtual ID	Clears the highest priority pending virtual interrupt. ***Note***: Virtual ID must match the ID of that interrupt. The CPU interface must generate a Clear Acknowledge with “V” set to one.
Downstream Write	0x8	Bit[15:12]: Length Bit[11:4]: Identifier	“Length” bytes of data	Write “length” bytes of data to the data value specified by “Identifier”. Constraints: 0 < length < 9
Generate SGI Acknowledge	0x9	-	-	Indicates the previous Generate SGI packet has been accepted and that the CPU interface can send a new Generate SGI.
Deactivate Acknowledge	0xA	-	-	Indicates the previous Deactivate packet has been accepted and that the CPU interface can send a new Deactivate.
Upstream Write Acknowledge	0xB	-	-	Indicates the previous Upstream Write packet has been accepted and that the CPU interface can send a new Upstream Write.
Activate Acknowledge	0xC	Bit [4]: V	-	Indicates a previous Activate packet with the same value of “V” has been accepted.

Table 46: Distributor to CPU commands

7.3.1 Distributor Interrupt Identifier Fields

The Interrupt Identifiers supported by the Distributor have an implementation defined size. This is discoverable from GICD_TYPER.IDbits (see sections 4.8.1 and 5.3.1).
A Distributor implementation may support 16 bits or less of Interrupt ID. Such implementations must provide Interrupt ID bits [15:0]. If less than 16 bits are supported, identifiers must be zero-extended to 16 bits.

When connected to a processor that supports more than 16 bits, the Interrupt ID must be zero extended to the number of bits supported by the processor.
See section 5.11.1 for the details of the rules governing interrupt identifier sizes.

7.3.2 Set

This command sets the highest priority pending interrupt for a processor. The processor has control of the interrupt and may respond to a read of the IAR with the Interrupt ID. The format of this packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]			Bit[3:0]
Priority		ID length	Mod	Grp	0x1
Interrupt ID [15:0]					
Interrupt ID [31:16]					

Where:

- “Priority” indicates the actual priority of the interrupt (i.e. the secure, unshifted view).
- “ID length” indicates how many bits of “Interrupt ID” are included in the packet. See section 7.2.4.
- “Interrupt ID” is the value returned by a valid read of an IAR. That is, it includes the Source CPU ID for SGI interrupts when ARE is zero (as defined for GICv2). ***Note***: if a packet includes Interrupt ID [31:16], then Interrupt ID [31:24] are RES0.
- “Mod” modifies the Grp bit as defined by GICD_IGRPMODRn.
- “Grp” indicates group of an interrupt as defined by GICD_IGROUPRn.

If the Distributor sends a Set command, the interrupt in the new command always displaces any previous interrupt – i.e. it sets a new highest priority pending interrupt. If still valid and pending, the previous interrupt must be released back to the Distributor.

The Distributor must obey the following rules:

- It can have at most two “Set” packets outstanding per processor at any time that are awaiting a response (an “Activate” or a “Release”).
- It may only issue a “Set” packet if it is able to accept an “Activate” with V == 0. ***Note***: an implementation might guarantee this by considering a “Set” packet to be outstanding until a “Release” packet has been received for the “Set” or an “Activate Acknowledge” packet has been sent for the corresponding “Activate”.
- It must never issue a “Set” with a special ID (1020 – 1023)
- If ARE is one for an interrupt group, it must never issue a “Set” with an ID in the range 1023 < ID < 8192
- If ARE is zero for an interrupt group, it must never issue a “Set” with an ID in the range 1023 < ID < 8192 where bits [9:4] are non-zero. That is, it may issue “Set” packets for SGIs with a “CPU Number” in bits [12:10].
- If ARE is zero for an interrupt group, it must never issue a “Set” with an ID greater than 8191 (i.e. LPIs are not permitted when ARE is zero).
- It may only issue another "Set" with the same “Interrupt ID” after receipt of an "Activate" or "Release" for that “Interrupt ID”.

Note: the Distributor will also not "Set" the same interrupt again until it is no longer “active” or has been released. For IDs < 8192, this typically requires a "Deactivate" packet, but can be accomplished by explicitly clearing the “active” state in the Distributor.

Note: this means the Distributor will always see responses to “Set” packets in the order they were issued.

If the CPU interface block cannot handle the interrupt because the interrupt group is (or becomes) disabled, it must issue a Release packet for Interrupt ID.

7.3.3 Clear

This command resets a specified pending interrupt. The format of this packet is shown below.

Bit[15:12]	Bit[11:8]	Bit[7:4]			Bit[3:0]
Reserved		ID length			0x3
Interrupt ID [15:0]					
Interrupt ID [31:16]					

Where:

- "ID length" indicates how many bits of "Interrupt ID" are included in the packet. See section 7.2.4.
- "Interrupt ID" identifies the interrupt that should be cleared. If the identified interrupt is in the pending state in the CPU interface, this pending state should be cleared and a Release packet must be issued before a Clear Acknowledge packet is issued. Otherwise the command has no effect but a Clear Acknowledge must still be issued after any outstanding "Activate" packet for the interrupt. ***Note***: if a packet includes Interrupt ID [31:16], then Interrupt ID [31:24] are RES0.

Note: the CPU interface must always respond to a Clear with a Clear Acknowledge with "V" set to zero.

7.3.4 Quiesce

This command requests that the CPU interface enters the quiescent state. The format of this packet is shown below.

Bit[15:12]	Bit[11:8]	Bit[7:4]				Bit[3:0]
Reserved						0x4

The CPU interface must issue responses to any outstanding Clear operations and must Release any pending interrupts and must drain any outstanding Generate SGI requests. The CPU interface is in the quiescent state when there are no pending interrupts and any outstanding Activate packets (including receipt of the Acknowledge packet), Deactivate packets (including receipt of the Acknowledge packet), Generate SGI packets (including receipt of the Acknowledge packet), Upstream Write packets (including receipt of the Acknowledge packet), and Clear Acknowledge packets have been sent to the Distributor.

Software must ensure no further traffic can be sent to the Distributor after the Quiesce Acknowledge packet has issued or the effects are unpredictable. ***Note***: because the timing of this is not predictable, this means software must ensure no further traffic is generated after GICR_WAKER.ProcessorSleep is set to one (see section 4.10.5). ***Note***: to ensure a Quiesce Acknowledge packet is issued if software continues to generate new traffic, an implementation must transmit any packets queued before receipt of a Quiesce packet but might prevent the transmission of any new (i.e. not already queued) packets generated by the processor after receipt of a Quiesce packet until the Quiesce Acknowledge packet has been successfully transmitted.

Note: the CPU interface must always respond to a Quiesce with a Quiesce Acknowledge when the CPU interface enters the quiescent state.

Note: this has no effect on outstanding SEIs and the value of any SEI Syndrome bits accumulated within the CPU interface will be discarded on power down.

Note: the CPU interface cannot receive a Quiesce packet while a Downstream Write Acknowledge is outstanding. See section 7.2.1.

7.3.5 VSet

In GICv4, this command sets a virtual interrupt pending for a virtual processor. The processor has control of the interrupt and may respond to a read of the IAR with the Virtual ID. The format of this packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]			Bit[3:0]
Priority		ID length		Grp	0x6
Virtual ID [15:0]					
Virtual ID [31:16]					

Where:

- "Priority" indicates the actual priority of the interrupt (i.e. the secure, unshifted view).
- "ID length" indicates how many bits of "Virtual ID" are included in the packet. See section 7.2.4.
- "Virtual ID" is the value returned by a valid virtual read of an IAR. ***Note***: if a packet includes Interrupt ID [31:16], then Interrupt ID [31:24] are RES0.
- "Grp" indicates group of the interrupt as defined by GICD_IGROUPRn. ***Note***: virtual interrupts are always non-secure.

The Distributor will send a VSet command when the virtual interrupt specified by Virtual ID is set as pending in the resident VPT.

The CPU interface must either issue an "Activate (Virtual ID)" with the "V" bit set or a "Release (Virtual ID)" to the Distributor.

If the Distributor sends a VSet command, the interrupt in the new command always displaces any previous interrupt – i.e. it sets a new highest priority pending virtual interrupt. If still valid and pending, the previous virtual interrupt must be released back to the Distributor.

Note: this means the Distributor will always see responses to "VSet" packets in the order the order they were issued.

Note: the Distributor must follow the following rules:

- It can have at most two "VSet" packets outstanding per processor at any time that are awaiting a response (an "Activate" or a "Release").
- It may only issue a "VSet" packet if it is able to accept an "Activate" with V == 1. ***Note***: an implementation might guarantee this by considering a "VSet" packet to be outstanding until a "Release" packet has been received for the "VSet" or an "Activate Acknowledge" packet has been sent for the corresponding "Activate".
- It must never issue a "VSet" with an identifier less than 8192
- It may only issue another "VSet" for a given identifier after receipt of either an "Activate" or a "Release" packet for that identifier.

If the CPU interface block cannot handle the interrupt because the interrupt group is (or becomes) disabled, it must issue a Release packet for Interrupt ID with V set to one. ***Note***: this includes when the virtual processor interface is disabled (i.e. when GICH_HCR.En is zero or ICH_HCR_EL2.En is zero, as appropriate).

If the CPU interface block cannot handle the interrupt because the hypervisor is not using system registers, it must issue a Release packet for the Interrupt ID with V set to one (i.e. when ICC_SRE_EL2.SRE is zero).

Note: when the non-secure copy of ICC_SRE_EL1.SRE is zero, an implementation might or might not factor the specified virtual interrupt into priority calculations and memory-mapped virtual register reads (e.g. GICV_IAR) at non-secure EL1.

Note: no “MOD” bit is required as only non-secure interrupts may be virtualized.

7.3.6 VClear

In GICv4, this command resets a specified pending virtual interrupt. The format of this packet is shown below.

Bit[15:12]	Bit[11:8]	Bit[7:4]			Bit[3:0]
Reserved		ID length			0x7
Virtual ID [15:0]					
Virtual ID [31:16]					

Where:

- "ID length" indicates how many bits of "Virtual ID" are included in the packet. See section 7.2.4.
- "Virtual ID" identifies the virtual interrupt that should be cleared. If the identified virtual interrupt is pending, it should be cleared and a Release packet must be issued before the VClear is Acknowledged. Otherwise the command has no effect but a Clear Acknowledge must still be issued. ***Note***: if a packet includes Interrupt ID [31:16], then Interrupt ID [31:24] are RES0.

Note: the CPU interface must always respond to a VClear with a Clear Acknowledge with "V" set to one.

Note: this packet has no effect on LPIs held in the List Registers.

7.3.7 Generate SGI Acknowledge

This command is sent by the Distributor in response to a Generate SGI Command when the effects of that Generate SGI packet are guaranteed to become visible to other processors. ***Note***: receipt of this packet does **not** guarantees that the corresponding SGI "pending" state has been set but it does guarantee that this pending state **will** become set. The format of the packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]	Bit[3:0]
Reserved	Reserved		0x9

7.3.8 Deactivate Acknowledge

This command is sent by the Distributor in response to a Deactivate Command when the effects of that Deactivate are observable by the Distributor and other processors. The format of the packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]	Bit[3:0]
Reserved	Reserved		0xA

7.3.9 Activate Acknowledge

This command is sent by the Distributor in response to an Activate Command when the effects of that Activate are observable by the Distributor and other processors. The format of the packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]		Bit[3:0]
Reserved	Reserved		V	0xC

Where:

- "V" indicates acknowledgement corresponds to an Activate for:
  - 0b0. The Activate was for a Set.
  - 0b1. The Activate was for a VSet.

7.3.10 Upstream Write Acknowledge

This command is sent by the Distributor in response to an Upstream Write Command when the effects of that Upstream Write are observable by the Distributor. The format of the packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]	Bit[3:0]
Reserved	Reserved		0xB

7.3.11 Downstream Write

This command writes the number of bytes data specified by the “Length” field to the Data Value specified by “Identifier”. The format of this packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]	Bit[3:0]
Length	Identifier		0x8
Data[1]		Data[0]	
…			
Data[Length-1]		Data[Length-2]	

The CPU interface will respond to this command with a Downstream Write Acknowledge command.

Where:

- “Length” indicates the number of bytes of valid data. If this field specifies a number of bytes that is not exactly divisible by the interface width (see section 7.1), any remaining bytes in the last transfer beyond this specified length must be zero and ignored by the CPU interface logic.
- “Identifier” is a value that specifies the format of the supplied data. ***Note***: each unique “Identifier” value might have a different length (but a given “Identifier” value must always have the same length)

At present, the following “Identifier” values have been defined:

Data Value Name	Identifier	Length	Contents
Settings	0x00	0x1	This variable holds the Distributor global settings: • Bit [7:6]. VL. Virtual Interrupt ID length supported. See section 7.2.4. • Bit [5:4]. PL. Physical Interrupt ID length supported. See section 7.2.4. • Bit [3:1]. Reserved RES0. • Bit [0]. DS. Disable security. The value of GICD_CTLR.DS. ***Note***: this bit will also be set if the Distributor only supports a single security state ***Note***: VL and PL are static fields and a CPU interface implementation may choose to use these bits only on the first packet received after reset.
Reserved	0x01 – 0x7f	*	

Implementation Defined	0x80 – 0xff	*	These identifiers are reserved for implementation defined variables. ***Note***: each identifier might have a different value of “Length”

Note: after the CPU interface has received a Downstream Write packet was sent with the DS set to one, it is a packet protocol violation if a Downstream Write packet to have DS set to zero is received without an intervening hardware reset.

7.4 Commands to the Distributor

The table below shows a summary of all the packets from a CPU interface block to the Distributor.

Command	ID	Parameters	Data	Description
Activate	0x1	Bit[7:6]: ID length Bit[4]: V	Interrupt ID	This acknowledges specified by Interrupt ID, sets it to the “active” state.
Release	0x3	Bit[7:6]: ID length Bit[4]: V	Interrupt ID	Release the interrupt specified by ID. Performed when a CPU can no longer process an ID for which it has received a Set. For example, if the CPU interface is disabled for a class of interrupts.
Clear Acknowledge	0x4	Bit [4]: V	-	Acknowledgement that the CPU has completed a Clear or VClear command (as indicated by the “V” bit).
Deactivate	0x6	Bits[10:8]: Groups Bit[7:6]: ID length	Interrupt ID	Deactivate Interrupt. Performs deactivation if the initiating interrupt regime is permitted access to the interrupt group of the Interrupt ID.
Generate SGI	0x7	Bit[15:12] SGInum Bit[7]: IRM Bit[6]: NS Bit[5:4]: SGT	Routing: A3, A2, A1 and Target List	The routing data is interpreted as defined for ICC_SGI{0,1}R_EL1 writes above. ***Note***: the A3 field is optional and might not be supported by an implementation. When not supported, A3 is zero.
Upstream Write	0x8	Bit[15:12]: Length Bit[11:4]: Identifier	“Length” bytes of data	Write “length” bytes of data to the data value specified by “Identifier”. Constraints: 0 < length < 9
Quiesce Acknowledge	0x9	-	-	The AXI4 Stream Interface to the CPU interface is in the quiescent state.
Downstream Write Acknowledge	0xB	-	-	Indicates the previous Downstream Write from the Distributor has completed and all its effects are visible.

Table 47: CPU to Distributor commands

7.4.1 CPU Interface Interrupt Identifier Fields

The Interrupt Identifiers supported by the GIC CPU interface have an implementation defined size (see section 4.8.1). This is discoverable from GICC_CTLR (see section 5.6.17), ICC_CTLR_EL1 (see section 5.7.33) and ICC_CTLR_EL3 (see section 5.7.34) as appropriate.
See section 5.11.1 for the details of the rules governing interrupt identifier sizes.

7.4.2 Activate

This command is sent when the CPU interface logic acknowledges an interrupt. On receipt of an Activate command, the Distributor sets the interrupt to the "active" state. The format of the packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]			Bit[3:0]
Reserved		ID length		V	0x1
Interrupt ID [15:0]					
Interrupt ID [31:16]					

The Distributor will respond to this command with an Activate Acknowledge command.

Where:

- "ID length" indicates how many bits of "Interrupt ID" are included in the packet. See section 7.2.4.
- "Interrupt ID" is the value that was returned when the IAR was read. That is, it includes the Source CPU ID for SGI interrupts when ARE is not set. ***Note***: if a packet includes Interrupt ID [31:16], then Interrupt ID [31:24] are RES0.
- "V" indicates the original command the Activate corresponds to:
  - 0b0. The Activate is for a Set.
  - 0b1. The Activate is for a VSet.

Note: an Activate should only be sent when the Distributor must take some action. That is:

- for PPIs, SGIs and SPIs where the Distributor must clear the pending bit and set the active bit
- for LPIs where the Distributor must clear the pending bit.

7.4.3 Release

This command is sent when the CPU interface logic cannot handle a particular interrupt. The format of the packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]			Bit[3:0]
Reserved		ID length		V	0x3
Interrupt ID [15:0]					
Interrupt ID [31:16]					

Where:

- "ID length" indicates how many bits of "Interrupt ID" are included in the packet. See section 7.2.4.
- "V" indicates the original command the Release corresponds to:
  - 0b0. The Release is for a Set.
  - 0b1. The Release is for a VSet.

Note: if a "Release" occurs as a consequence of disabling an interrupt group, it is recommended that the "Upstream Write" packet conveying the new interrupt group enables is sent before the "Release".

Note: if the "Interrupt ID" corresponds to a "one of N" interrupt, the Distributor might choose to forward the interrupt to a different processor or it might choose to re-send the interrupt to the same processor.

Note: if a packet includes Interrupt ID [31:16], then Interrupt ID [31:24] are RES0.

7.4.4 Clear Acknowledge

This command is sent to acknowledge the receipt of a Clear Interrupt command. The format of the packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]				Bit[3:0]
Reserved					V	0x4

Where:

- V" indicates the original command the Clear Acknowledge corresponds to:
  - 0b0. The Clear Acknowledge is for Clear.
  - 0b1. The Clear Acknowledge is for a VClear.

7.4.5 Deactivate

This command is sent to perform a Deactivate operation for an interrupt. The format of this packet is shown below:

Bit[15:12]	Bit[11:8]		Bit[7:4]			Bit[3:0]
Reserved		Groups	ID length			0x6
Interrupt ID [15:0]						

Where:

- "ID length" indicates how many bits of "Interrupt ID" are included in the packet. See section 7.2.4.
  - This field must be zero, indicating 16 bits of "Interrupt ID" are provided.
  - ***Note***: Deactivate packets may only be issued for SPIs, SGIs, and PPIs. Hence, the "Interrupt ID" must always be less than 8192 and the "ID length" field ***must*** be zero.
- "Groups" indicates the interrupt groups that the initiating interrupt regime is permitted to modify
  - Bit [2]. When set to one, Secure Group 1 interrupts may be modified.
  - Bit [1]. When set to one, Non-Secure Group 1 interrupts may be modified.
  - Bit [0]. When set to one, Group 0 interrupts may be modified.
  - When SRE is one for the initiating interrupt regime, a bit may only be set if the initiating interrupt regime has access to the physical interrupt group as defined in Table 17: CPU Interface System Register Access Behaviour. ***Note***: when the initiating interrupt regime is secure EL1 this means that if secure EL1 has permission to handle group 1 interrupts (i.e. IRQ is not routed to EL3), bit [1] will be one, and bit[2] will be one if GICD_CTLR.DS is zero.
  - When SRE is zero for the initiating interrupt regime, the bits are set according to the security state of the initiating interrupt regime. That is, bit [1] is set for non-secure writes, and bits [2:0] are all set for secure writes. ***Note***: when GICD_CTLR.DS is one, writes that result in the generation of a Deactivate packet are treated as if they were "secure".

    - If the Deactivate packet is due to handling of a virtual interrupt that is mapped to a physical interrupt in the List Registers (i.e. the “HW” bit is set and a virtual write resulted in deactivation of the physical interrupt; see section 5.9.6), the bits are set as if an equivalent write had been performed at EL2.
    - ***Note***: if GICD_CTLR.DS is zero, Group 0 and Group 1 Secure interrupts may only be modified if the initiating interrupt regime is secure (this includes EL3 regardless of the setting of SCR_EL3.NS).
    - ***Note***: if GICD_CLTR.DS is one, the bit for Secure Group 1 interrupts can be ignored by the Distributor.
    - ***Note***: a Deactivate packet must have at least one the “Groups” bits set to one. If none of the Group bits are set to one, it is a protocol error.
- “Interrupt ID” is the data value presented for the write to the EOI registers (with no masking).

The Distributor will respond to this command with a Deactivate Acknowledge command.

Note: the Distributor must follow the following rules:

- If ARE is one for an interrupt group, it must acknowledge but otherwise ignore any “Deactivate” with an ID in the range 1023 < ID < 8192
- If ARE is zero for an interrupt group, it must acknowledge but otherwise ignore any “Deactivate” with an ID in the range 1023 < ID < 8192 where bits [9:4] are non-zero. That is, it may issue “Set” packets for SGIs with a non-zero “CPU Number” in bits [12:10].
- If ARE is zero for an interrupt group and the ID specifies an SGI and the processor specified by “CPU Number” in bits [12:10] does not support operation with ARE as zero, the “Deactivate” packet must be ignored.

7.4.6 Generate SGI

This command is set to the Distributor to generate an SGI. The format of the packet is shown below:

Bit[15:12]	Bit[11:8]		Bit[7:4]			Bit[3:0]
SGInum		A3V	IRM	NS	SGT	0x7
Target List						
A2			A1			
			A3			

Where:

- "SGInum" indicates the SGI to be generated.
- "NS" indicates whether the packet was generated from a non-secure interrupt regime.
  - 0b0. The packet was generated from a secure interrupt regime (including EL3 regardless of the value of SCR_EL3.NS)
  - 0b1. The packet was generated from a non-secure interrupt regime.
- "IRM" indicates the Interrupt Routing Mode to be used.
  - When IRM is one, the Target List, A1, A2 and A3 fields are ignored.
- "A3V" indicates whether the packet includes an "A3" field. When zero, the packet does not include an A3 field and the Distributor must use zero as the value of A3.
- "A1", "A2" and "A3" provide the affinity values to be used for selecting the set of target processors.
- "Target List" provides the list of processors within affinity A3.A2.A1 that will receive the SGI.

The Distributor will respond to this command with a Generate SGI Acknowledge command.

A GICv3 SGI is to be generated and the IRM, affinity values and target list are to be treated as defined for ICC_SGI{0,1}R_EL1 and ICC_ASGI1R_EL1 (see section 5.7.29). The NS bit indicates the security state that generated the SGI and the SGT field indicates the command used to generate the SGI:

- 0b00. ICC_SGI0R_EL1
- 0b01. ICC_SGI1R_EL1
- 0b10. ICC_ASGI1R_EL1
- 0b11. Reserved.

Note: it is implementation defined whether the A3 field is supported (indicated by the value of the A3V field). When not supported, the Distributor must use zero as the value of A3.

Note: in systems where the Distributor does not support non-zero values of the A3 field but the processor supports fully supports this field, the Distributor might choose to generate an SEI on receipt of a packet with a non-zero A3 field.

7.4.7 Upstream Write

This command writes the number of bytes data specified by the “Length” field to the Data Value specified by “Identifier”. The format of this packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]	Bit[3:0]
Length	Identifier		0x8
Data[1]		Data[0]	
…			
Data[Length-1]		Data[Length-2]	

The Distributor will respond to this command with an Upstream Write Acknowledge command.

Where:

- “Length” indicates the number of bytes of valid data. If this field specifies a number of bytes that is not exactly divisible by the interface width (see section 7.1), any remaining bytes in the last transfer beyond this specified length must be zero and must be ignored by the Distributor.
- “Identifier” is a value that specifies the format of the supplied data.

At present, the following ”Identifier” values have been defined:

Data Value Name	Identifier	Length	Contents
Physical interface enables	0x00	0x1	This variable holds the physical CPU interface enables that must be communicated to the Distributor: • Bit [2]. EnableGrp1, secure. The value of the secure copy of ICC_IGRPEN1_EL1.Enable. • Bit [1]. EnableGrp1, non-secure. The value of the non-secure copy of ICC_IGRPEN1_EL1.Enable (or GICC_CTLR.EnableGrp1 when SRE is not set for an interrupt regime). • Bit [0]. EnableGrp0, secure. The value of ICC_IGRPEN0_EL1.Enable (or GICC_CTLR.EnableGrp0 when SRE is not set for an interrupt regime). ***Note***: for processors without EL3, or when GICD_CTLR.DS is set to one, the value of some of these bits is determined from section 4.6.10. To ensure the state of the enables is easily communicated to the Distributor following power-up (see section 4.10.6) this packet must be generated by any write to an enable register (as specified above for each bit). If multiple writes to an enable register occur before the packet is generated, an implementation can coalesce these into a single packet.
Virtual interface enables	0x01	0x1	This variable holds the virtual CPU interface enables that must be communicated to the Distributor: • Bit [1]. EnableGrp1. The value of ICH_VMCR_EL2.VENG1. • Bit [0]. EnableGrp0. The value of ICH_VMCR_EL2.VENG0.

			To ensure the state of the enables is easily communicated to the Distributor following power-up (see section 4.10.6) this packet must be generated by any write to a virtual enable bit. If multiple writes to a virtual enable bit occur before the packet is generated, an implementation can coalesce these into a single packet. ***Note***: if EL2 is memory mapped and using GICH_VMCR, then the guest must be memory mapped and using GICV_* accesses. If an implementation shares state between GICH_* and the ICH_* system registers it can chose whether or not to communicate any change to the virtual enables. ***Note***: similarly, if a guest is memory mapped but EL2 is using system registers, an implementation can chose whether or not to communicate any change to the virtual enables.
Physical Priority	0x02	0x1	This variable contains the current value of the Priority Mask Register. ***Note***: in distributor implementations that use this value, the distributor copy of the variable must reset to the idle priority (e.g. 0xf8 if 5 bits of priority are implemented). • Bit [7:0]. Priority. The value written to the PMR. This packet must be generated when the processor successfully writes to the Priority Mask Register and ICC_CTLR_EL3.PMHE is set to one. The packet must be generated by any successful write that changes the value of the Priority Mask Registers. If multiple successful writes occur before the packet is transmitted, an implementation can coalesce these into a single packet. ***Note***: an implementation might choose to generate this packet on any write to the Priority Mask Register. ***Note***: if a new Set packet is received with a priority lower than the current PMR value during the window when the Upstream Write Acknowledge for that PMR value has not been received, an implementation might choose to Release that Set packet.

Note**:** an implementation might choose to generate a system error if the Length or Identifier field is invalid.

7.4.8 Quiesce Acknowledge

This command is sent by the CPU Interface in response to a Quiesce Command. The format of the packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]	Bit[3:0]
Reserved	Reserved		0x9

This packet will be issued once the CPU interface has released any pending interrupts and any outstanding Activate packets (including receipt of the Acknowledge packet), Deactivate packets (including receipt of the Acknowledge packet), Generate SGI packets (including receipt of the Acknowledge packet), Upstream Write packets (including receipt of the Acknowledge packet), and Clear Acknowledge packets have been sent to the Distributor before issuing a Quiesce Acknowledge.

7.4.9 Downstream Write Acknowledge

This command is sent by the CPU Interface in response to a Downstream Write Command. The format of the packet is shown below:

Bit[15:12]	Bit[11:8]	Bit[7:4]			Bit[3:0]
Reserved	Reserved	VL	PL	Reserved	0xB

Where:

- VL indicates the Virtual Interrupt Identifier length supported (see section 7.2.4).
  - The value returned to the distributor must be set to the minimum of that received from the first Downstream Write in the VL field received after reset and the value of ICH_VTR_EL2.IDbits.
- PL indicates the Physical Interrupt Identifier length supported (see section 7.2.4).
  - The value returned to the distributor must be set to the minimum of that received from the first Downstream Write in the PL field received after reset and the value of ICC_CTLR_EL1/3.IDbits.

7.5 Distributor Implementations

The protocol has been designed to allow for different styles of Distributor implementation.

7.5.1 Handling Explicitly Routed Interrupts

When the Distributor receives an interrupt that explicit targets a particular processor, this must be sent to the affinity level 0 re-distributor (and subsequently the processor) using a “Set” packet.

This interrupt will remain pending in the processor’s CPU interface until one of the following occurs:

- The interrupt is acknowledged (by reading IAR)
- A new interrupt is sent to the CPU interface, resulting in a Release packet from the CPU interface
- A Quiese Interface packet is sent to the CPU interface, resulting in a Release packet from the CPU interface.

7.5.2 Handling Interrupts Routed to 1 of N Processors

When the Distributor receives an interrupt that targets 1 of N processor, it must distribute this interrupt as shown below:

- A processor with the interrupt group enabled is selected (ensuring “fairness”) and the interrupt is forwarded to that processor using a “Set” packet.
- If the Distributor determines that the processor has not Activated the interrupt in a timely manner, it may optionally issue a Clear, wait for the interrupt to be Released, wait for the Clear Acknowledge and issue the interrupt to another processor using any permitted method.

7.5.3 Releasing Pending Interrupts

When a re-distributor “Sleep” bit is set by a write to GICR_WAKER, this is communicated to the distributor. The re-distributor must:

- Send an Upstream Write packet to set “Sleep” to ‘1’ in the distributor
- Release any (transient) pending interrupts within the re-distributor. This is achieved by:
    - Issue a Quiesce packet to the processor if its “ProcessorSleep” bit is not set. ***Note***: the “ProcessorSleep” bit is set when the re-distributor issues the Quiesce packet.
    - Wait for a Quiesce Acknowledge packet from the processor (if a Quiesce packet was sent).
    - Forward any Release packets from the processor to the distributor.
    - ***Note***: the re-distributor must also release any new interrupts received from the distributor after sending the Upstream Write packet with sleep set to one.
- Discard (and acknowledge) any Generate SGI packets received from its children. ***Note***: this is to ensure deadlock-free operation. Software can check that any outstanding Generate SGI packets have been sent upstream by reading GICR_CTLR.UWP (see section 5.4.7).
- Send an Upstream Write packet to the distributor indicating that this re-distributor and its children are now quiescent.
- The re-distributor interface is now quiescent and “ChildrenAsleep” will read as one.

On receipt of an Upstream Write packet that sets “Sleep” for a child re-distributor, the distributor must:

- Ensure no more packets are sent to that re-distributor.
- The distributor might still receive a Clear packet for the child re-distributor after this point
- If so, it must wait for receipt of the Upstream Write packet from the child indicating it has completed the “Sleep” operation before responding with a Clear Acknowledge.

7.5.4 Combined Re-Distributor Implementations

An implementation might choose to combine multiple re-distributors into a single logical block (for example to reduce communications overheads.

Such implementations must:

- Provide WakeRequest signals for all (direct) child re-distributors outside of the combined block

Where such an implementation supports the (optional) Q-Channel interface, it must:

- Provide a single Q-Channel interface for power management where PWRQACTIVE indicates the combined set of re-distributors are quiescent and may be requested to power down

7.5.5 Implementations With Priority-based Routing

An implementation might choose to use the value of PMR to inform how the distributor picks the target of a “1 of N” SPI. In such implementations:

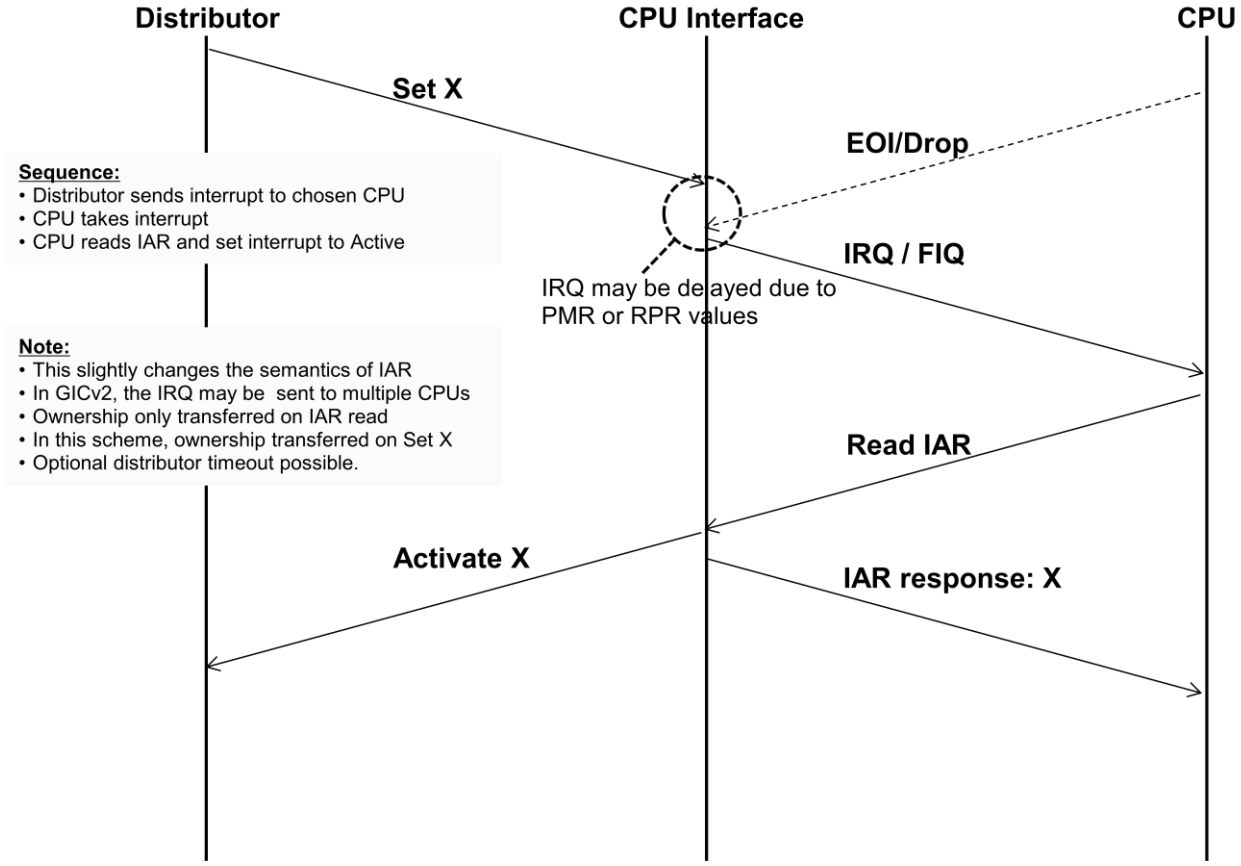
- If ICC_CTLR_EL{1, 3}.PMHE is zero, the re-distributor will always forward interrupts to the CPU interface regardless of the PMR value and CPU interface must not self-initiate a release even if it has an interrupt below the current PMR threshold.
- If ICC_CTLR_EL{1, 3}.PMHE is one, distributor implementations might or might not be using PMR to inform “1 of N” routing decisions and the CPU interface must:
    - If it receives an interrupt below the current PMR value during the window between updating PMR and receiving the Upstream Write Acknowledge that indicates this has been observed by the distributor, it must immediately issue a Release packet for this interrupt.
    - If it receives an interrupt below the current PMR value outside the above window, it must assume the distributor does not use PMR and it must not release the interrupt.

7.6 Example sequences

The intent of this section is to show how normal interrupts are handled and to highlight important race conditions.

7.6.1 Normal “Set” Interrupt

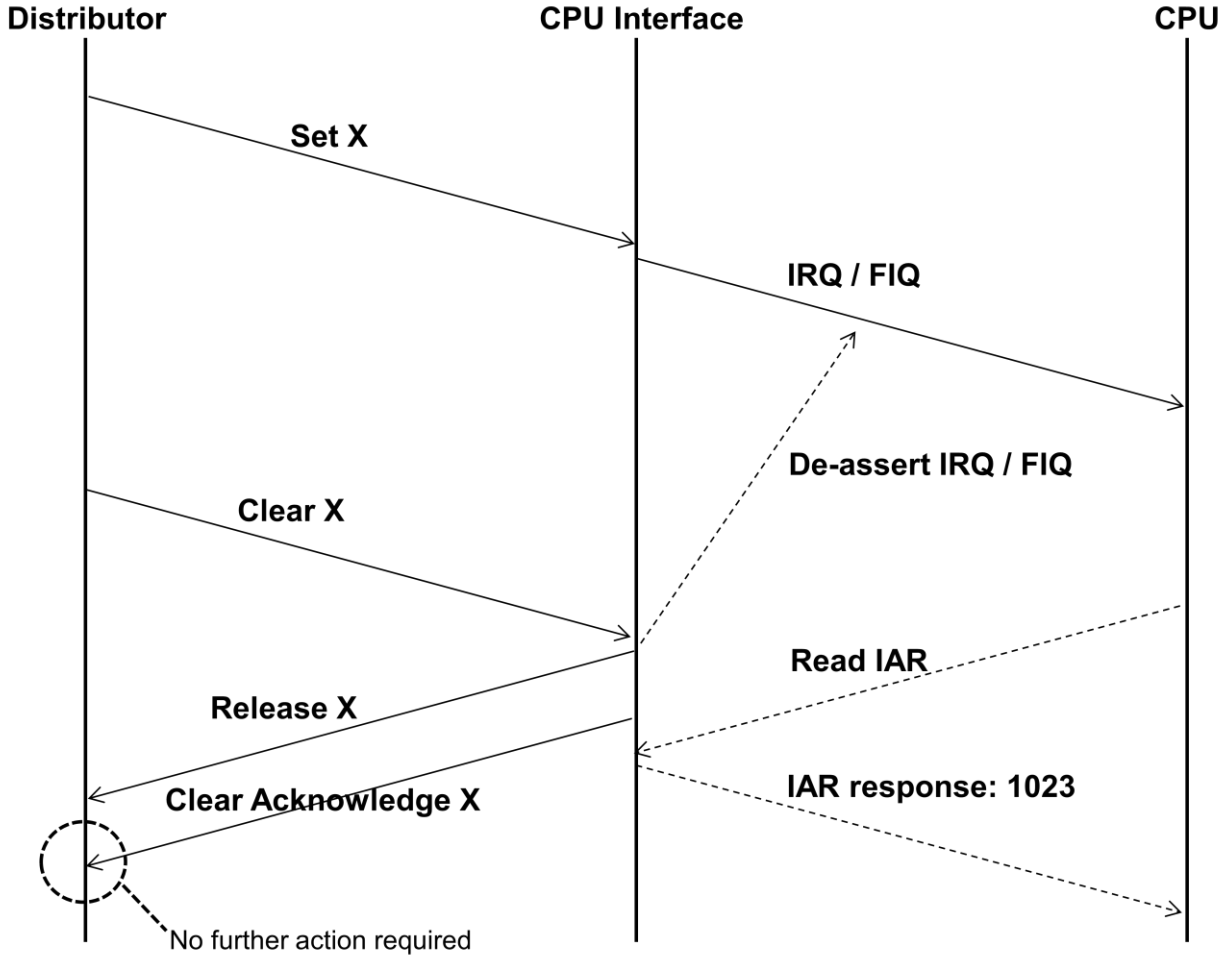
The diagram below shows the sequence of commands for a “normal” handling of an interrupt in implementations where the Distributor sends an interrupt to only a single processor.



In this scenario, a “Set” is sent indicating that the CPU interface may immediately return a valid Interrupt ID on a read of the IAR.

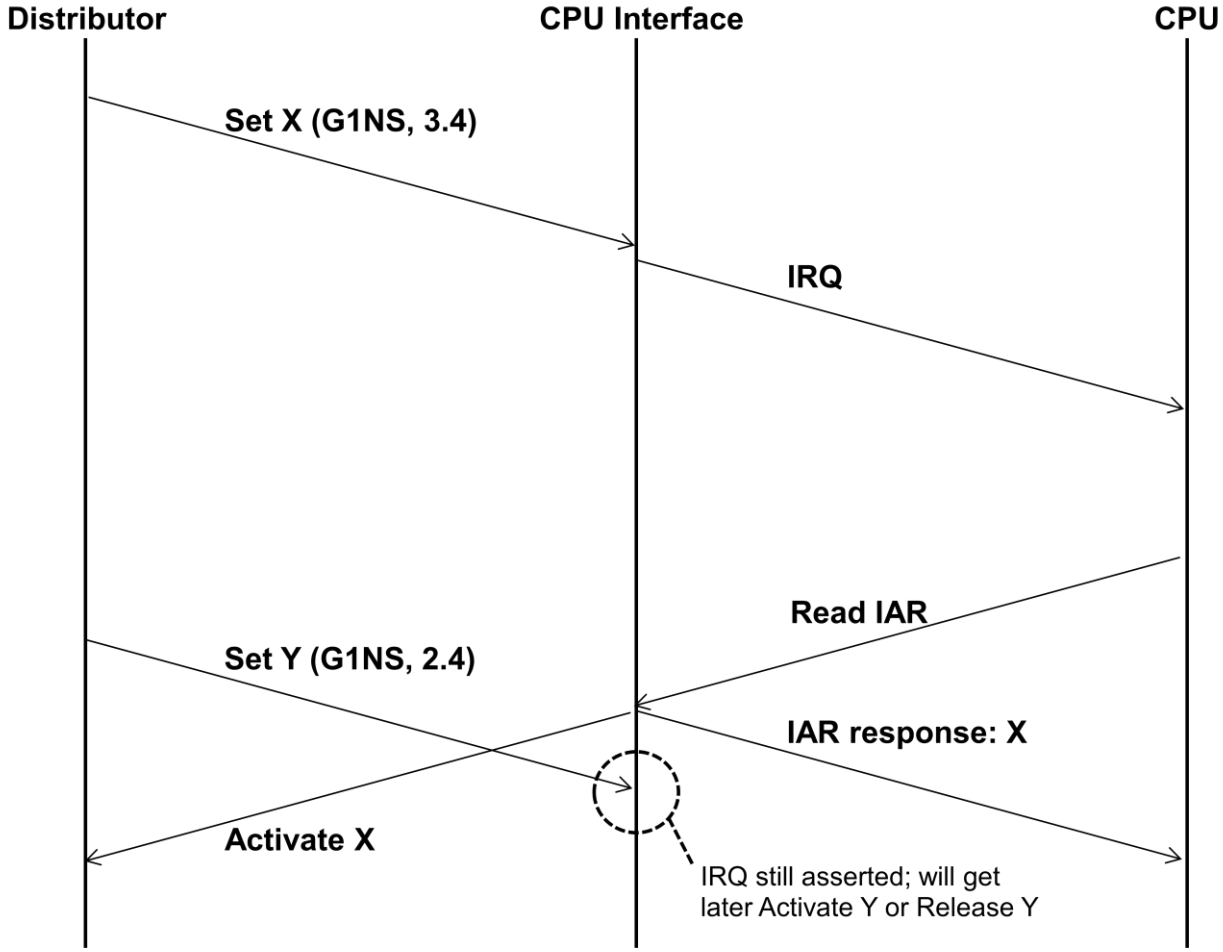
7.6.2 De-asserted Interrupts

A level sensitive interrupt may be de-asserted before or after the CPU reads the IAR. If the interrupt is de-asserted before the read, then the IAR read contains a spurious ID.



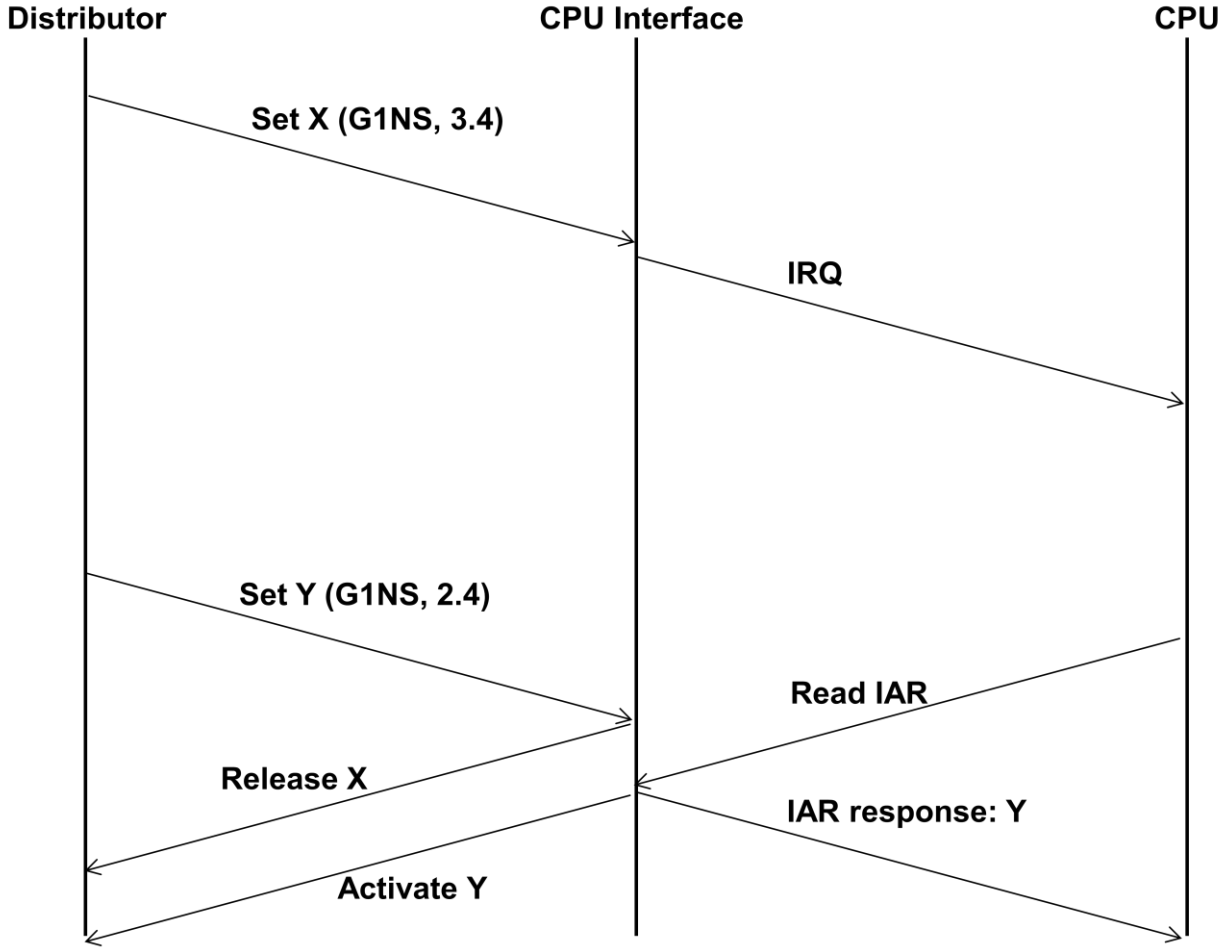
7.6.3 Higher Priority Interrupt Arrival #1

The diagram below shows the sequence of commands if a higher priority interrupt arrives just after the read of the IAR for a lower priority one.



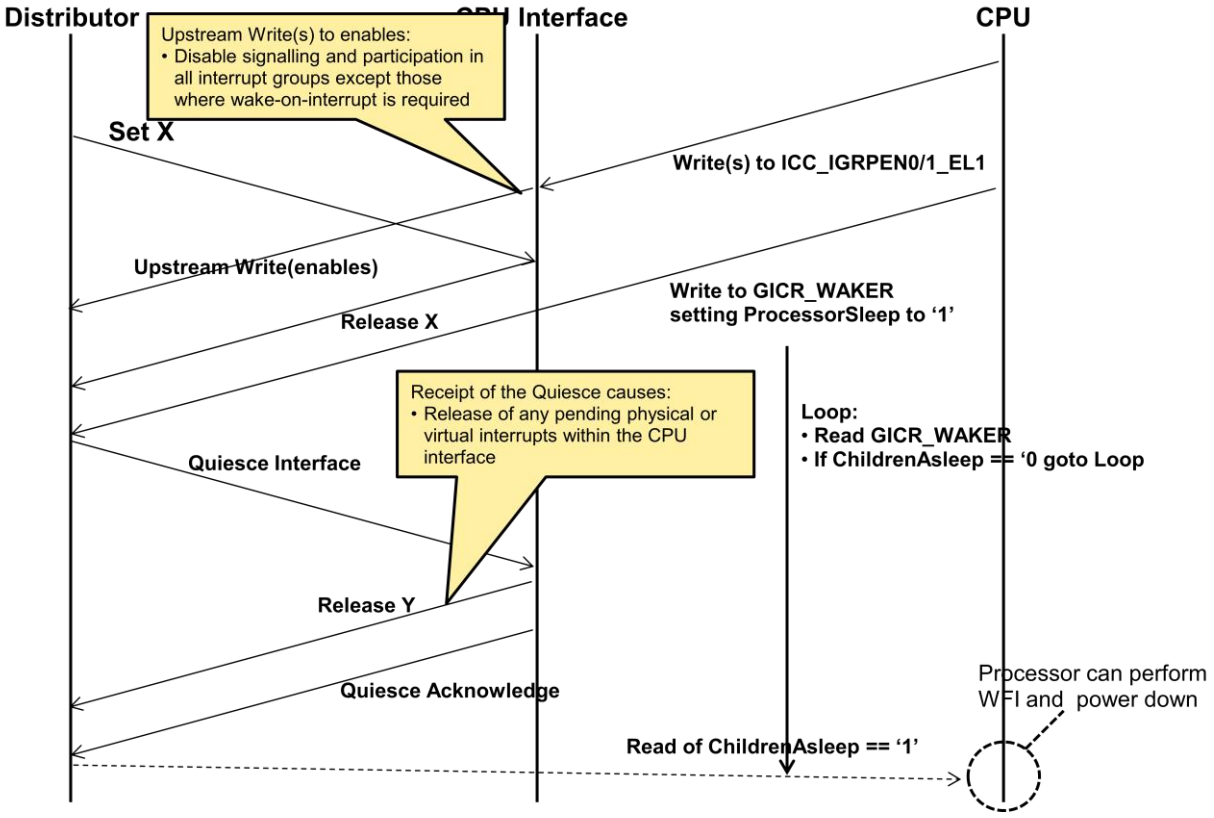
7.6.4 Higher Priority Interrupt Arrival #2

The diagram below shows the sequence of commands if a higher priority interrupt arrives just before the read of the IAR for a lower priority one.



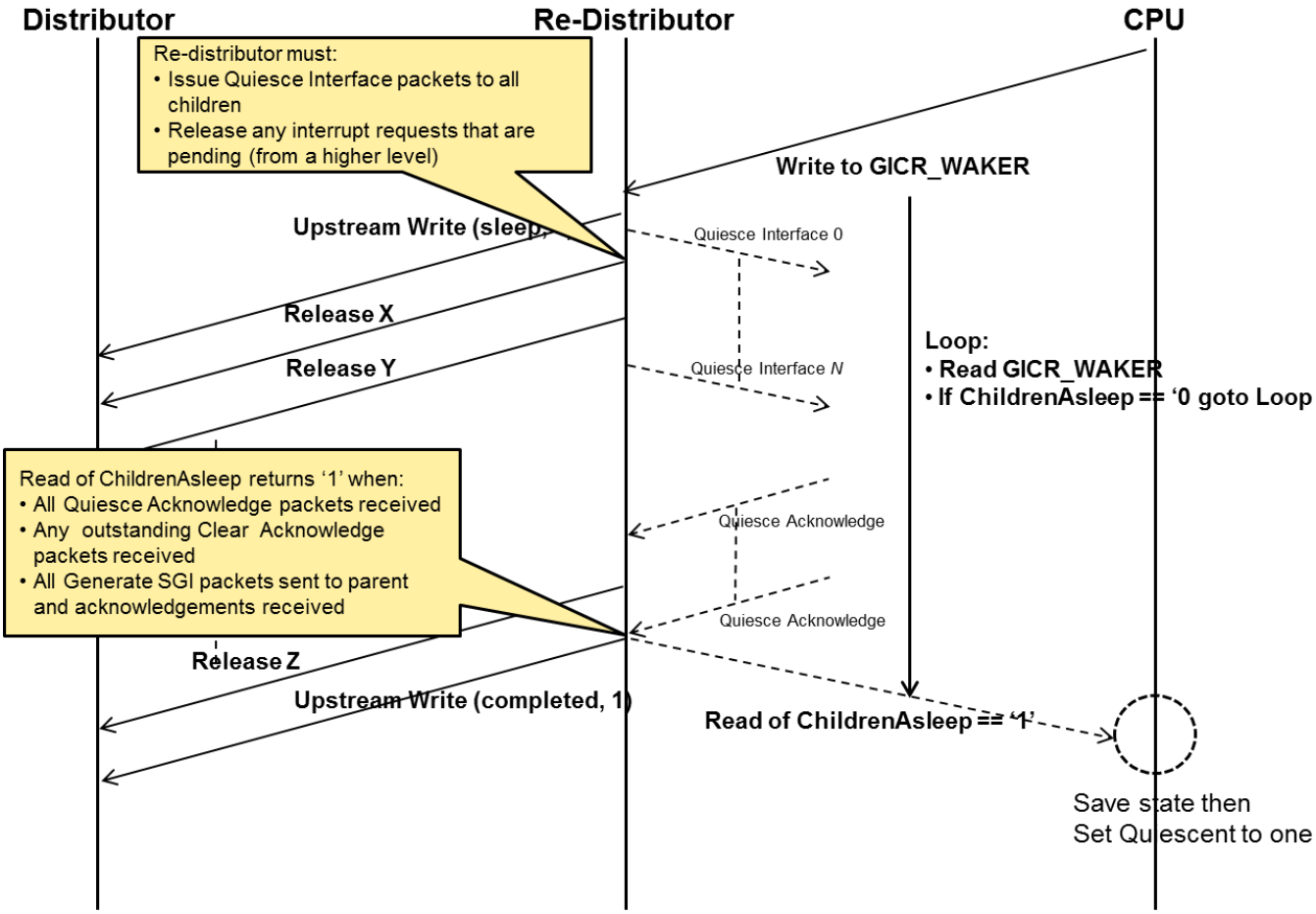
7.6.5 Processor Power Down

The diagram below shows the sequence of commands if an interrupt arrives during the power-down sequence.



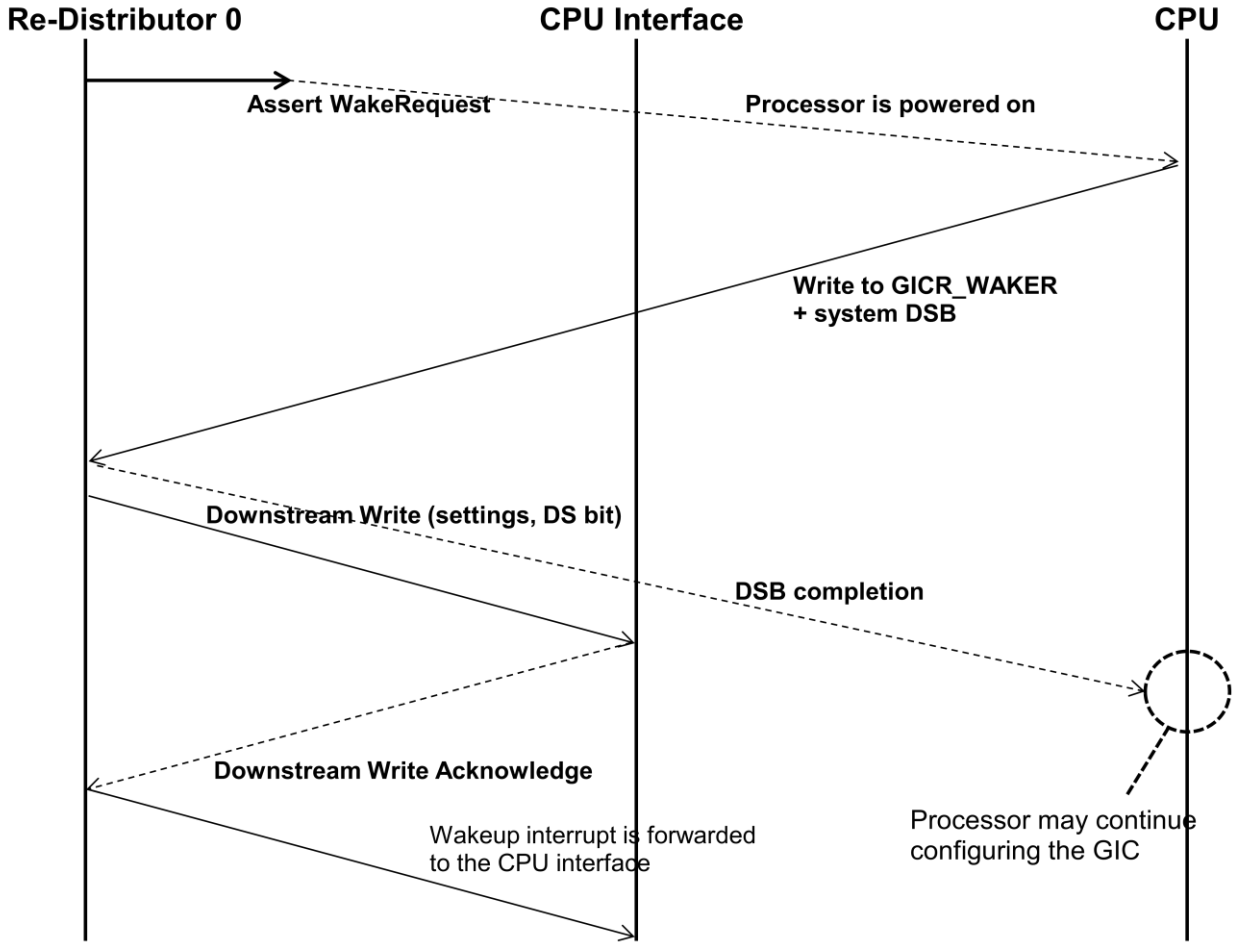
7.6.6 Re-Distributor Power Down

The diagram below shows the sequence of operations required to power down a re-distributor:



7.6.7 Processor Power Up

The diagram below shows the expected processor power-up sequence:



7.6.8 Re-Distributor Power Up

The diagram below shows the expected re-distributor power up sequence: