# ARM® Generic Interrupt Controller

**Architecture version 1.0**

# Architecture Specification

ARM IHI 0048A

# ARM Generic Interrupt Controller

## Release Information

The following changes have been made to this document.

**Change History**

| Date | Issue | Confidentiality | Change |
|---|---|---|---|
| 23 September 2008 | A | Non-Confidential Unrestricted Access | First release for version 1.0 |

## Proprietary Notice

110 Fulbourn Road Cambridge, England CB1 9NJ

# Contents
# ARM Generic Interrupt Controller Architecture Specification

## Preface

## Chapter 1 Introduction

## Chapter 2 GIC Partitioning

## Chapter 3 Interrupt Handling and Prioritization

# Preface

This preface summarizes the contents of this specification and lists the conventions it uses. It contains the following sections:

# About this specification

This is version 1.0 of the *Architecture Specification* for the *ARM Generic Interrupt Controller* (GIC).

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

## Intended audience

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

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

## Using this specification

This specification is organized into the following chapters:

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

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

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

Read this for a description of the major interfaces and components of the GIC. The chapter also describes how they operate.

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

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

**Chapter 4** ***Programmers Model***

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

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

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

**Appendix B** ***Software Examples for the GIC***

Read this for a description of non-architectural, non-prescriptive, methods of using the GIC, and of how the GIC manages interrupts.

**Appendix C** ***Register Shortform Names***

Read this for a description of relationship between the architectural shortform names of the registers described in this specification and their legacy shortform aliases, and for an alphabetic index of the architectural shortform names.

***Glossary*** Read the Glossary for definitions of terms used in this document.

# Conventions

The following sections describe conventions that this book can use:

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

## General typographic conventions

`monospace` Used for assembler syntax descriptions, pseudocode, and source code examples.

Also used in the main text for instruction mnemonics and for references to other items appearing in assembler syntax descriptions, pseudocode, and source code examples.

***italic*** Highlights important notes, introduces special terminology, and denotes internal cross-references and citations.

**bold** Denotes signal names and is used for terms in descriptive lists, where appropriate.

**SMALL CAPITALS** Used for a few terms that have specific technical meanings, that are included in the Glossary.

## Signals

The signal conventions are:

**Signal level** The level of an asserted signal depends on whether the signal is active-HIGH or active-LOW. Asserted means:

- HIGH for active-HIGH signals
- LOW for active-LOW signals.

**Lower-case n** At the start or end of a signal name denotes an active-LOW signal.

## Numbers

Numbers are normally written in decimal. Binary numbers are preceded by 0b, and hexadecimal numbers by `0x` and written in a `monospace` font.

## Pseudocode descriptions

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

# Further reading

This section lists publications by ARM and by third parties.

See http://infocenter.arm.com/ for access to ARM documentation.

## ARM publications

This specification contains information that is specific to the GIC. See the following documents for other relevant information:

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

## External publications

This section lists relevant documents published by third parties:

- JEP106M, *Standard Manufacture's Identification Code, JEDEC Solid State Technology Association*.

# Feedback

ARM welcomes feedback on this documentation.

## Feedback on this specification

If you have any comments or suggestions about this documentation, contact your supplier and give:

- the document title
- the document number
- an explanation with as much information as you can provide.

ARM also welcomes general suggestions for additions and improvements.

# Chapter 1
# Introduction

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

- *About the Generic Interrupt Controller architecture* on page 1-2
- *Security Extensions support* on page 1-3
- *Terminology* on page 1-4.

## 1.1 About the Generic Interrupt Controller architecture

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

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

**Note**

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

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

- registers for managing interrupt sources, interrupt behavior, and interrupt routing to one or more processors
- support for:
  - the ARM architecture Security Extensions
  - enabling, disabling, and generating processor interrupts from hardware (peripheral) interrupt sources
  - generating software interrupts
  - interrupt masking and prioritization
  - uniprocessor and multiprocessor environments.

The GIC takes interrupts asserted at the system level and signals them to each connected processor as appropriate. If the GIC implements the Security Extensions it can implement two interrupt requests to a connected processor. The architecture identifies these two requests as IRQ and FIQ.

**Note**

In many implementations the IRQ and FIQ interrupt requests correspond to the IRQ and FIQ asynchronous exceptions that are supported by all variants of the ARM architecture except the *Microcontroller profile* (M-profile). For more information about IRQ, FIQ, and asynchronous exceptions, see the *ARM Architecture Reference Manual, ARMv7-A and ARMv7-R edition*.

### 1.1.1 GIC architecture specification version

This specification describes version 1.0 of the GIC architecture. Some products have been implemented using preliminary versions of this specification and might not fully comply with this specification.

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

## 1.2 Security Extensions support

The ARM GIC architecture Security Extensions support:

- configuring each interrupt as either Secure or Non-secure
- signalling Secure interrupts to the target processor using either the IRQ or the FIQ exception request
- a unified scheme for handling the priority of Secure and Non-secure interrupts
- optional lockdown of the configuration of some Secure interrupts.

In an implementation that includes the Security Extensions:

- System software individually defines each implemented interrupt as either Secure or Non-secure.
- The behavior of processor accesses to registers in the GIC depends on whether the access is Secure or Non-secure, see *Processor security state and Secure and Non-secure GIC accesses* on page 1-5.

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

  — a Non-secure read of a register field holding state information for a Secure interrupt returns zero

  — the GIC ignores any Non-secure write to a register field holding state information for a Secure interrupt.

  Non-secure accesses can only read or set information corresponding to Non-secure interrupts. Secure accesses can read or set information corresponding to both Non-secure and Secure interrupts.
- A Non-secure interrupt signals an IRQ interrupt request to a target processor.
- A Secure interrupt can signal either an IRQ or an FIQ interrupt request to a target processor.

## 1.3 Terminology

The following sections define architectural terms used in this specification:

- *Interrupt states*
- *Interrupt types*
- *Models for handling interrupts* on page 1-5
- *Spurious interrupts* on page 1-5
- *Processor security state and Secure and Non-secure GIC accesses* on page 1-5
- *Banking* on page 1-6.

### 1.3.1 Interrupt states

The following states apply at the interface between the GIC and each processor supported in the system:

**Inactive** An interrupt that is not active or pending.

**Pending** An interrupt from a source to the GIC that is recognized as asserted in hardware or generated by software and is waiting to be serviced by a target processor.

**Active** An interrupt from a source to the GIC that has been acknowledged by a processor, and is being serviced but has not completed.

**Active and pending** A processor is servicing the interrupt and the GIC has a pending interrupt from the same source.

### 1.3.2 Interrupt types

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

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

**Private peripheral interrupt (PPI)**

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

**Shared peripheral interrupt (SPI)**

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

Each peripheral interrupt is either:

**Edge-triggered**

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

**Level-sensitive**

This is an interrupt that is asserted whenever the interrupt signal level is HIGH, and deasserted whenever the level is LOW.

**Note**

While a level-sensitive interrupt is asserted its state in the GIC is pending, or active and pending. If the peripheral deasserts the interrupt signal for any reason the GIC removes the pending state from the interrupt. For more information see *Interrupt handling state machine* on page 3-10.

**Software-generated interrupt (SGI)**

This is an interrupt generated by software writing to a specific register in the GIC. The system uses SGIs for interprocessor communication.

A software interrupt has edge-triggered properties. The software triggering of the interrupt is equivalent to the edge transition of the interrupt signal on a peripheral input.

### 1.3.3 Models for handling interrupts

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

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

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

### 1.3.4 Spurious interrupts

It is possible that an interrupt that the GIC has signaled to a processor is no longer required. If this happens, when the processor acknowledges the interrupt, the GIC returns a special Interrupt ID that identifies the interrupt as a *spurious interrupt*. This can happen because:

- the state of the interrupt has changed
- software has re-programmed the GIC to change the processing requirements for the interrupt
- the interrupt is handled using the 1-N model and another processor has acknowledged the interrupt.

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

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

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

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

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

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

### 1.3.6 Banking

**Interrupt banking**

In a multiprocessor implementation, for PPIs and SGIs, the GIC can have multiple interrupts with the same interrupt ID. Such an interrupt is called a *banked interrupt*, and is identified uniquely by the combination of its interrupt ID and its associated CPU interface. For more information see *Interrupt IDs* on page 2-4.

**Register banking**

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

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

For more information see *Register banking* on page 4-5.

# Chapter 2
# GIC Partitioning

This chapter describes the architectural partitioning of the GIC. It contains the following sections:

## 2.1 About GIC partitioning

This GIC architecture splits logically into a Distributor block and one or more CPU interface blocks, as Figure 2-1 on page 2-3 shows:

**Distributor** This performs interrupt prioritization and distribution to the CPU interfaces that connect to the processors in the system.

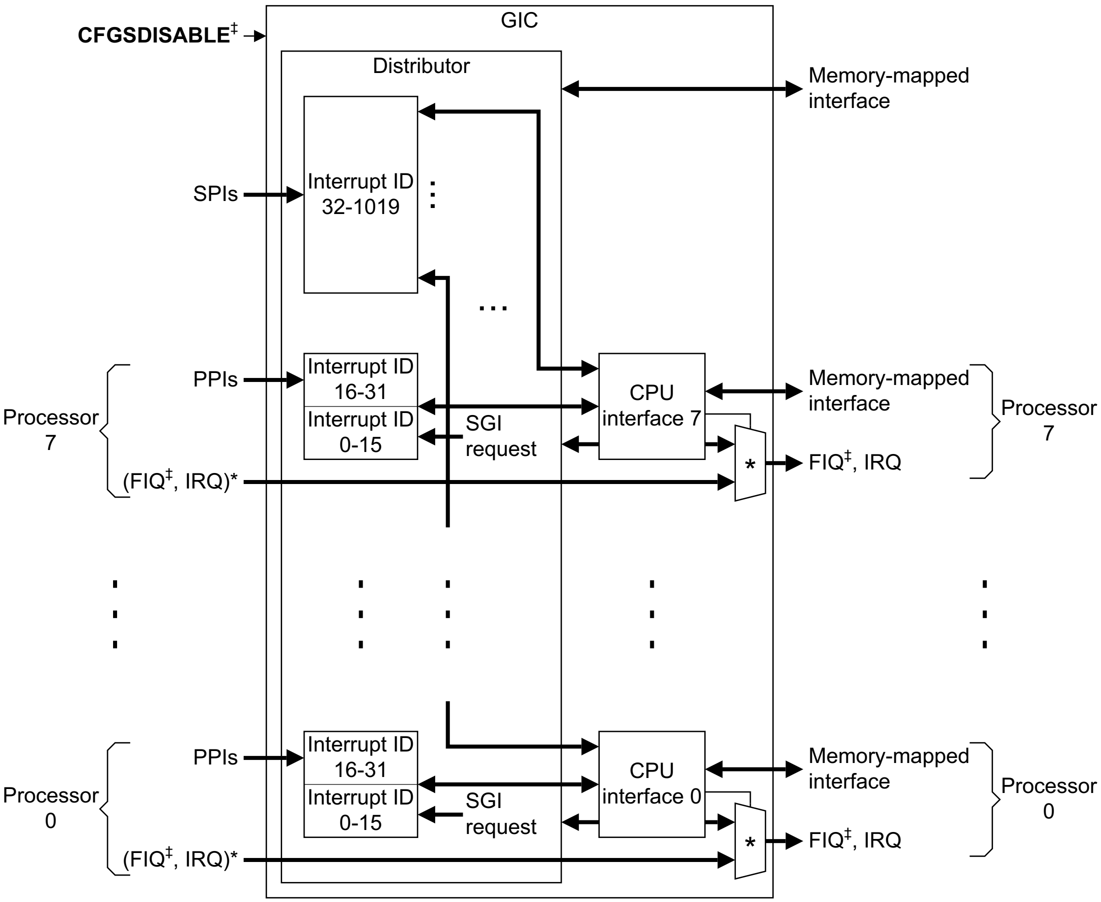
**CPU interfaces** Each CPU interface performs priority masking and preemption handling for a connected processor in the system.

Each block provides part of the GIC programmers model, and the programmers model is generally the same for each implemented CPU interface. This model supports implementation of the GIC in uniprocessing or multiprocessing environments.

**Note**

- The split of the GIC into the Distributor and CPU interface blocks is an architectural abstraction. It is IMPLEMENTATION DEFINED whether these blocks are implemented separately or combined.
- In a GIC that implements the Security Extensions in a multiprocessor system, a CPU interface can be implemented so that it receives:
  - both Secure and Non-secure accesses
  - only Secure accesses
  - only Non-secure accesses.

A GIC can implement up to eight CPU interfaces, numbered from 0-7.



‡ Only if Security Extensions are implemented

* Optional input and bypass multiplexer, see text

**Figure 2-1 GIC logical partitioning into Distributor and CPU interfaces**

## 2.2 The Distributor

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

The Distributor provides a programming interface for:

- globally enabling the forwarding of interrupts to the CPU interfaces
- enabling or disabling each interrupt
- setting the priority level of each interrupt
- setting the target processor list of each interrupt
- setting each peripheral interrupt to be level-sensitive or edge-triggered
- if the GIC implements the Security Extensions, setting each interrupt as either Secure or Non-secure
- sending an SGI to one or more target processors.

In addition, the Distributor provides:

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

### 2.2.1 Interrupt IDs

Interrupts from sources are identified using *ID numbers*. Each CPU interface can see up to 1020 interrupts. Because of banking of SPIs and PPIs this means the distributor supports up to 1244 interrupts.

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

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

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

  — ID0-ID15 are used for SGIs

  — ID16-ID31 are used for PPIs

  In a multiprocessor system:

  — A PPI is signalled to a particular CPU interface, and is private to that interface. In prioritizing interrupts for a CPU interface the distributor considers only the PPIs signalled to that interface.

  — Each connected processor issues an SGI by writing to the ICDSGIR in the Distributor, see *Software Generated Interrupt Register (ICDSGIR)* on page 4-39. Each SGI can target multiple processors. In the distributor and in a targeted processor, an SGI is identified uniquely by the combination of its interrupt number, ID0-ID15, and the *processor source ID*, CPUID0-CPUID7, of the processor that issued the SGI. Banking SGIs means the GIC can handle multiple software interrupts simultaneously without resource conflicts.

The Distributor ignores any write to the ICDSGIR that is not from a processor that is connected to one of the CPU interfaces. How the Distributor determines the processor source ID of a processor writing to the ICDSGIR is IMPLEMENTATION DEFINED.

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

- Interrupt numbers ID1020-ID1023 are reserved for special purposes, see *Special interrupt numbers* on page 3-11.

System software sets the priority of each interrupt independent of its interrupt number.

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

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

For more information about message passing see *Message passing between processors* on page B-20.

## 2.3 CPU interfaces

Each CPU interface block provides the interface for a processor that operates with the GIC. Each CPU interface provides a programming interface for:

- enabling the signalling of interrupt requests by the CPU interface
- acknowledging an interrupt
- indicating completion of the processing of an interrupt
- setting an interrupt priority mask for the processor
- defining the preemption policy for the processor
- determining the highest priority pending interrupt for the processor.

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

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

**Note**

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

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

- The ID number of the highest priority pending interrupt, if that interrupt is of sufficient priority to generate an interrupt exception on the processor. This is the normal response to an interrupt acknowledge.
- Exceptionally, an ID number that indicates a spurious interrupt.

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

When the interrupt handler on the processor has completed the processing of an interrupt, it writes to the CPU interface to indicate interrupt completion. When this happens, the distributor changes the status of the interrupt either:

- from active to inactive
- from active and pending to pending.

# Chapter 3
# Interrupt Handling and Prioritization

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

## 3.1 About interrupt handling and prioritization

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

- *Handling different interrupt types in a multiprocessor system* on page 3-3
- *Identifying the supported interrupts* on page 3-3.

The remainder of the chapter describes interrupt handling and prioritization.

Interrupt handling describes:

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

Prioritization describes:

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

The GIC architecture supports uniprocessor and multiprocessor systems. In either a uniprocessor or a multiprocessor system, a GIC can implement the ARM Security Extensions. A GIC that implements the Security Extensions:

- recognizes that a connected processor that implements the Security Extensions makes either Secure accesses or Non-secure accesses to the GIC registers
- supports:
  - — the configuration of interrupts as either Secure or Non-secure
  - — the handling of Secure and Non-secure interrupts.
- in a multiprocessor system, might implement the Security Extensions on only some of its CPU interfaces.

Support for Secure and Non-secure interrupts makes interrupt handling and prioritization more complex. This chapter describes interrupt handling and prioritization in a GIC that does not implement the Security Extensions, and then describes the effect of the Security Extensions, in the following sections:

- *General handling of interrupts* on page 3-5
- *Interrupt prioritization* on page 3-12
- *The effect of the Security Extensions on interrupt handling* on page 3-15
- *The effect of the Security Extensions on interrupt prioritization* on page 3-18
- *Pseudocode details of interrupt handling and prioritization* on page 3-25.

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

A GIC supports *peripheral interrupts* and *software-generated interrupts*, see *Interrupt types* on page 1-4.

In a multiprocessor implementation the GIC handles:

- software generated interrupts (SGIs) using an N-N model
- peripheral (hardware) interrupts using a 1-N model.

See *Models for handling interrupts* on page 1-5 for definitions of the two models.

### 3.1.2 Identifying the supported interrupts

The GIC architecture defines different ID values for the different types of interrupt, see *Interrupt IDs* on page 2-4. However, there is no requirement for the GIC to implement a continuous block of interrupt IDs for any interrupt type.

**Note**

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

To correctly handle interrupts, software must know what interrupt IDs are supported by the GIC. Software can use the ICDISERs to discover this information, see *Interrupt Set-Enable Registers (ICDISERn)* on page 4-19. If the processor implements the Security Extensions, Secure software determines which interrupts are visible to Non-secure software. The Non-secure software must know which interrupts it can see, and might use this discovery process to find this information.

ICDISER0 provides the Set-enables bits for:

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

The remaining ICDISERs, from ICDISER1, provide the Set-enable bits for the SPIs, starting at interrupt ID 32.

If an interrupt is:

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

Software discovers which interrupts are supported as follows:

- Read the ICDICTR, see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. The ITLinesNumber field identifies the number of implemented ICDISERs, and therefore the maximum number of SPIs that might be supported.
- Write 0 to the ICDDCR.Enable bit, to disable forwarding of interrupts to CPU interfaces, see *Distributor Control Register (ICDDCR)* on page 4-12.

**Note**

When ICDDCR.Enable is set to 0 the GIC ignores the state of peripheral signals. This means it might miss edge-triggered interrupts.

- For each implemented ICDISER, starting with ICDISER0:
  - Write `0xFFFFFFFF` to the ICDISER.
  - Read the value of the ICDISER. Bits that read as 1 correspond to supported interrupt IDs.

Software uses the ICDICERs to discover which interrupts are permanently enabled, see *Interrupt Clear-Enable Registers (ICDICERn)* on page 4-21. It does this discovery as follows. For each implemented ICDICER, starting with ICDICER0:

- Write `0xFFFFFFFF` to the ICDICER. This disables all interrupts that can be disabled.
- Read the value of the ICDICER. Bits that read as 1 correspond to interrupts that are permanently enabled.
- Write 1 to any bits in the ICDICER that correspond to interrupts that must be re-enabled.

The GIC implements the same number of ICDISERs and ICDICERs.

When software has completed its discovery, it writes 1 to the ICDDCR.Enable bit, to enable forwarding of interrupts to CPU interfaces.

If the GIC implements the Security Extensions, software can use Secure accesses to:

- discover all the supported interrupt IDs.
- write to the ICDISRs, to configure interrupts as Secure or Non-secure, see *Interrupt Security Registers (ICDISRn)* on page 4-17.

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

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

## 3.2 General handling of interrupts

The Distributor maintains a state machine for each supported interrupt on each CPU interface. *Interrupt handling state machine* on page 3-10 describes this state machine and its state transitions. The possible states of an interrupt are:

- inactive
- pending
- active
- active and pending.

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

The GIC operates on interrupts as follows:

1. The GIC determines whether each interrupt is enabled. An interrupt that is not enabled has no further effect on the GIC.
2. For each enabled interrupt that is pending, the Distributor determines the targeted processor or processors.
3. For each processor, the Distributor determines the highest priority pending interrupt, based on the priority information it holds for each interrupt, and forwards the interrupt to the CPU interface.
4. The CPU interface compares the interrupt priority with the current interrupt priority for the processor, determined by a combination of the Priority Mask Register, the current preemption settings, and the highest priority active interrupt for the processor. If the interrupt has sufficient priority, the GIC signals an interrupt exception request to the processor.

   **Note**

   Throughout this document, an interrupt is described as having *sufficient priority* if its priority value, compared with the Priority Mask Register value, the preemption settings for the interface, and the priority of the highest priority active interrupt on the processor, mean that the CPU interface must signal the interrupt request to the processor.

5. When the processor takes the interrupt exception, it reads the ICCIAR in its CPU interface to acknowledge the interrupt, see *Interrupt Acknowledge Register (ICCIAR)* on page 4-56. This read returns an Interrupt ID that the processor uses to select the correct interrupt handler. When it recognizes this read, the GIC changes the state of the interrupt:
   - if the pending state of the interrupt persists when the interrupt becomes active, or if the interrupt is generated again, from pending to active and pending.
   - otherwise, from pending to active

**Note**

- A level-sensitive peripheral interrupt persists when it is acknowledged by the processor, because the interrupt signal to the GIC remains asserted until the interrupt service routine (ISR) running on the processor accesses the peripheral asserting the signal.
- In a multiprocessor implementation, the GIC handles:
  - SGIs using an N-N model, where the acknowledgement of an interrupt by one processor has no effect on the state of the interrupt on other CPU interfaces
  - peripheral interrupts using a 1-N model, where the acknowledgement of an interrupt by one processor removes the pending status of the interrupt on any other targeted processors, see *Implications of the 1-N model* on page 3-8.

6. When the processor has completed handling the interrupt, it signals this completion by writing to the ICCEOIR in the GIC, see *End of Interrupt Register (ICCEOIR)* on page 4-59.

   The GIC requires the order of completion of interrupts by a particular processor to be the reverse of the order of acknowledgement, so the last interrupt acknowledged must be the first interrupt completed.

   When the processor writes to the ICCEOIR, the GIC changes the state of the interrupt, for the corresponding CPU interface, either:

   - from active to inactive
   - from active and pending to pending.

   If there is no pending interrupt of sufficient priority for the CPU interface to signal it to the processor, the interface deasserts the interrupt exception request to the processor.

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

   - for SPIs, this means the interface never signals any interrupt that is active and pending on any CPU interface
   - for SGIs, the interface never signals any interrupt that is active and pending on this interface, but does not consider whether the interrupt is active and pending on any other interface
   - any PPI is private to this interface and the interface does not signal it if it is active and pending.

For more information about the steps in this process see:

- *Interrupt prioritization* on page 3-12,
- for a GIC that implements the Security Extensions:
  - *The effect of the Security Extensions on interrupt handling* on page 3-15
  - *The effect of the Security Extensions on interrupt prioritization* on page 3-18.

### 3.2.1 Interrupt controls in the GIC

The following sections describe the interrupt controls in the GIC:

- *Interrupt enables*
- *Setting and clearing pending state of an interrupt*
- *Finding the active or pending state of an interrupt* on page 3-8
- *Generating an SGI* on page 3-8.

#### Interrupt enables

For peripheral interrupts, a processor:

- enables an interrupt by writing to the appropriate ICDISER bit, see *Interrupt Set-Enable Registers (ICDISERn)* on page 4-19
- disables an interrupt by writing to the appropriate ICDICER bit, see *Interrupt Clear-Enable Registers (ICDICERn)* on page 4-21.

Whether SGIs are permanently enabled, or can be enabled and disabled by writes to the ICDISER and ICDICER, is IMPLEMENTATION DEFINED.

Writes to the ICDISERs and ICDICERs control whether the Distributor forwards interrupts to the CPU interfaces. Disabling an interrupt by writing to the appropriate ICDICER does not prevent that interrupt from changing state, for example becoming pending.

#### Setting and clearing pending state of an interrupt

For peripheral interrupts, a processor can:

- set the pending state by writing to the appropriate ICDISPR bit, see *Interrupt Set-Pending Registers (ICDISPRn)* on page 4-23
- clear the pending state by writing to the appropriate ICDICPR bit, see *Interrupt Clear-Pending Registers (ICDICPRn)* on page 4-26.

For a level-sensitive interrupt:

- If the hardware signal of an interrupt is asserted when a processor writes to the corresponding ICDICPR bit then the write to the register has no effect on the pending state of the interrupt.
- If a processor writes a 1 to an ICDISPR bit then the corresponding interrupt becomes pending regardless of the state of the hardware signal of that interrupt, and remains pending regardless of the assertion or deassertion of the signal.

For more information about the control of the pending state of a level-sensitive interrupt see *Control of the pending status of level-sensitive interrupts* on page 4-28.

For SGIs, the GIC ignores writes to the corresponding ICDISPR and ICDISCR bits. A processor cannot change the state of a software-generated interrupt by writing to these registers.

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

A processor can find:

- the pending state of an interrupt by reading the corresponding ICDISPR or ICDICPR bit
- the active state of an interrupt by reading the corresponding ICDABR bit, see *Active Bit Registers (ICDABRn)* on page 4-29.

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

For an SGI, the corresponding ICDISPR and ICDICPR bits RAO if there is a pending interrupt from at least one generating processor that targets the processor reading the ICDISPR or ICDICPR.

### Generating an SGI

A processor generates an SGI by writing to an ICDSGIR, see *Software Generated Interrupt Register (ICDSGIR)* on page 4-39. An SGI can target multiple processors, and the ICDSGIR write specifies the target processor list. The ICDSGIR includes optimization for:

- interrupting only the processor that writes to the ICDSGIR
- interrupting all processors other than the one that writes to the ICDSGIR.

SGIs from different processors use the same interrupt IDs. Therefore, any target processor can receive SGIs with the same interrupt ID from different processors. On the CPU interface of the target processor, the pending status of each of these interrupts is independent of the pending status of any other interrupt, but only one interrupt with this ID can be active. Reading the ICCIAR for an SGI returns both the interrupt ID and the CPU ID of the processor that generated the interrupt, uniquely identifying the interrupt.

In a multiprocessor implementation, the interrupt priority of each SGI interrupt ID is defined independently for each CPU interface, see *Interrupt Priority Registers (ICDIPRn)* on page 4-31. This means that, for each CPU interface, all SGIs with a particular interrupt ID that are pending on that interface have the same priority and must be handled serially. How the CPU interface serializes these SGIs is IMPLEMENTATION DEFINED.

## 3.2.2 Implications of the 1-N model

In a multiprocessor implementation, the GIC uses a 1-N model to handle peripheral interrupts that target more than one processor. This means that when the GIC recognizes an interrupt acknowledge from one of the target processors it clears the pending state of the interrupt on all the other targeted processors. This model means an interrupt can be handled by the first available processor. However, the interrupt might generate an interrupt exception on more than one of the targeted processors, for example if two of the targeted processors recognize the interrupt exception request from the GIC at similar times.

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

- A processor reads the ICCIAR and obtains the interrupt ID of the interrupt to be serviced, see *Interrupt Acknowledge Register (ICCIAR)* on page 4-56. More than one target processor might have obtained this interrupt ID, if the processors read their ICCIARs at very similar times. The system might require software on the target processors to ensure that only one processor runs its interrupt service routine. A typical mechanism to achieve this is implementing a lock on the *interrupt service routine* (ISR), in shared memory. This might operate as follows:

- — each target processor that obtains the interrupt ID from its read of the ICCIAR runs a semaphore routine, attempting to obtain a lock on the ISR corresponding to the specified ID value
- — if a processor fails to obtain the lock it does no further processing of the interrupt, but writes the interrupt ID to its ICCEOIR, see *End of Interrupt Register (ICCEOIR)* on page 4-59
- — the processor that obtains the lock handles the interrupt and then writes the interrupt ID to its ICCEOIR.

- A processor reads the ICCIAR and obtains the interrupt ID 1023, indicating a spurious interrupt. The processor can return from its interrupt service routine without writing to its ICCEOIR.

  The spurious interrupt ID indicates that the original interrupt is no longer pending, typically because another target processor is handling it.

**Note**

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

### 3.2.3 Interrupt handling state machine

The distributor maintains a state machine for each supported interrupt on each CPU interface. Figure 3-1 shows an instance of this state machine, and the possible state transitions.

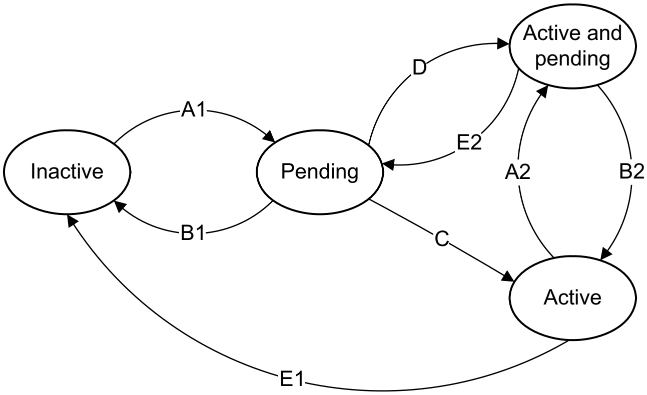


**Figure 3-1 Interrupt handling state machine**

**Note**

SGIs are generated only by writes to an ICDSGIR. Peripheral interrupts are generated by either a peripheral indicating it requires service, or by a write to an ICDISPR.

When the Distributor and CPU interfaces are enabled, the conditions that cause each of the state transitions are as follows:

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

For an SGI:

- Occurs on a write to an ICDSGIR that specifies the processor as a target.
- If the GIC implements the Security Extensions and the write to the ICDSGIR is Secure, the transition occurs only if the security configuration of the specified SGI, for the appropriate CPU interface, corresponds to the ICDSGIR.SATT bit value.

For an SPI or PPI, occurs if either:

- a peripheral asserts an interrupt signal
- software writes to an ICDISPR.

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

Not applicable to SGIs:

- a pending SGI must transition through the active state, or reset, to remove its pending status.
- an active and pending SGI must transition through the pending state, or reset, to remove its pending status.

For an SPI or PPI, occurs if either:

- the level-sensitive interrupt is pending only because of the assertion of an input signal, and that signal is deasserted
- the interrupt is pending only because of the assertion of an edge-triggered interrupt signal, or a write to an ICDISPR, and software writes to the corresponding ICDICPR.

**Transition C** If the interrupt is enabled and of sufficient priority to be signalled to the processor, occurs when software reads from the ICCIAR.

**Transition D** For an SGI, occurs if the associated SGI is enabled and the Distributor forwards it to the CPU interface at the same time that the processor reads the ICCIAR to acknowledge a previous instance of the SGI. Whether this transition occurs depends on the timing of the read of the ICCIAR relative to the reforwarding of the SGI.

For an SPI or PPI:

- Occurs if all the following apply:
  - The interrupt is enabled.
  - Software reads from the ICCIAR. This read adds the active state to the interrupt.
  - For a level-sensitive interrupt, the interrupt signal remains asserted. This is usually the case, because the peripheral does not deassert the interrupt until the processor has serviced the interrupt.
- For an edge-triggered interrupt, whether this transition occurs depends on the timing of the read of the ICCIAR relative to the detection of the reassertion of the interrupt. Otherwise the read of the ICCIAR causes transition C, possibly followed by transition A2.

**Transition E1 or E2, remove active status**

Occurs when software writes to the ICCEOIR.

### 3.2.4 Special interrupt numbers

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

**Note**

A race condition can cause a spurious interrupt. For example, a spurious interrupt can occur if a processor writes a 1 to a field in an ICDICER that corresponds to a pending interrupt after the CPU interface has signalled the interrupt to the processor and the processor has recognized the interrupt, but before the processor has read from the ICCIAR.

For more information about the special interrupt numbers see *Special interrupt numbers when the Security Extensions are implemented* on page 3-16.

## 3.3 Interrupt prioritization

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

- *Preemption* on page 3-13
- *Priority masking* on page 3-13
- *Priority grouping* on page 3-14
- *Interrupt generation* on page 3-14.

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

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

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

In the GIC prioritization scheme, lower numbers have higher priority, that is, the lower the assigned priority value the higher the priority of the interrupt. The highest interrupt priority always has priority field value 0, and the lowest value depends on the number of implemented priority levels, as Table 3-1 shows.

The ICDIPRs hold the priority value for each supported interrupt, see *Interrupt Priority Registers (ICDIPRn)* on page 4-31. To determine the number of priority bits implemented write `0xFF` to an ICDIPR priority field and read back the value stored.

**Note**

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

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

An implementation might reserve an interrupt for a particular purpose and assign a fixed priority to that interrupt, meaning the priority value for that interrupt is read-only.

This model aligns with the priority grouping mechanism described in *Priority grouping* on page 3-14.

If, on a particular CPU interface, multiple pending interrupts have the same priority, and have sufficient priority that the interface must signal them to the processor, it is IMPLEMENTATION DEFINED how the interface selects which interrupt to signal.

When an interrupt is active on a CPU interface, the GIC might signal a higher-priority interrupt on that CPU interface, see *Preemption*.

Software sets the priority of each interrupt in the appropriate ICDIPR, see *Interrupt Priority Registers (ICDIPRn)* on page 4-31. It is IMPLEMENTATION DEFINED whether a write to the ICDIPR changes the priority of any active interrupt.

### 3.3.1 Preemption

A CPU interface supports forwarding of higher priority pending interrupts to a target processor before an active interrupt completes. A pending interrupt is only forwarded if it has a higher priority than all of:

- the priority of the highest priority active interrupt on the target processor, the *running priority* for the processor, see *Running Priority Register (ICCRPR)* on page 4-61
- the priority mask, see *Priority masking*
- the priority group, see *Priority grouping* on page 3-14.

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

**Note**

For a processor that complies with the ARM architecture:

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

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

### 3.3.2 Priority masking

The ICCPMR for a CPU interface defines a priority threshold for the target processor, see *Interrupt Priority Mask Register (ICCPMR)* on page 4-52. The GIC only signals pending interrupts with a higher priority than this threshold value to the target processor. A value of zero, the register reset value, masks all interrupts to the associated processor.

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

**Note**

Writing 255 to the ICCPMR always sets it to the largest supported priority field value. Table 3-1 on page 3-12 shows how the largest supported field value varies with the number of implemented priority bits.

### 3.3.3 Priority grouping

Priority grouping splits each priority value into two fields, the *group priority* and the *subpriority* fields. The GIC uses the group priority field to determine whether a pending interrupt has sufficient priority to preempt a currently active interrupt.

The binary point field in the ICCBPR controls the split of the priority bits into the two parts. This 3-bit field specifies how many of the least significant bits of the 8-bit interrupt priority field are excluded from the group priority field, as Table 3-2 shows.

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

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

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

For more information about the ICCBPR see *Binary Point Register (ICCBPR)* on page 4-54.

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

### 3.3.4 Interrupt generation

The pseudocode in *Exception generation pseudocode, without the Security Extensions* on page 3-27 describes the generation of interrupts by the GIC.

## 3.4 The effect of the Security Extensions on interrupt handling

If a GIC CPU interface implements the Security Extensions, it provides two interrupt output signals, IRQ and FIQ:

- The CPU interface always uses the IRQ exception request for Non-secure interrupts
- Software can configure the CPU interface to use either IRQ or FIQ exception requests for Secure interrupts.

**Note**

In a GIC that does not support the Security Extensions, each CPU interface implements only a single port for signalling interrupts to a target processor. The alternatives for providing GIC interrupt routing for both IRQ and FIQ exception requests in a processor environment that does not include the Security Extensions are:

- Implement two instances of the GIC, without the Security Extensions. One instance routes FIQs and the other instance routes IRQs.
- Implement a single GIC that implements the Security Extensions, and implements the CPU interface to the processor so that all accesses appear to the GIC as Secure accesses. In an implementation of this model, software configures an interrupt as Secure to assign it to the generation of FIQ exception requests to the processor.

The remainder of this section describes a GIC that implements the Security Extensions.

### 3.4.1 Security Extensions support

Software can detect support for the Security Extensions by reading the ICDICTR.SecurityExtn bit, see *Interrupt Controller Type Register (ICDICTR)* on page 4-14.

Secure software makes Secure writes to the ICDISRs to configure each interrupt as Secure or Non-secure, see *Interrupt Security Registers (ICDISRn)* on page 4-17.

In addition:

- The banking of registers provides independent control of Secure and Non-secure interrupts, see *Effect of the Security Extensions on the programmers model* on page 4-7.
- The Secure copy of the ICCICR has additional fields to control the processing of Secure and Non-secure interrupts, see *CPU Interface Control Register (ICCICR)* on page 4-47. These fields are:
  - — the SBPR bit, that affects the preemption of Non-secure interrupts, see *Control of preemption by Non-secure interrupts* on page 3-22
  - — the FIQEn bit, that controls whether the interface signals Secure interrupts to the processor using the IRQ or FIQ interrupt exception requests
  - — the AckCtl bit, that affects the acknowledgment of Non-secure interrupts, see *Effect of the Security Extensions on interrupt acknowledgement* on page 3-16

- — the EnableNS bit, that controls whether Non-secure interrupts are signaled to the processor, and is an alias of the Enable bit in the Non-secure ICCICR.

- The Non-secure copy of the ICCBPR is aliased as the ICCABPR, see *Aliased Binary Point Register (ICCABPR)* on page 4-62. This is a Secure register, meaning it is only accessible by Secure accesses.

### 3.4.2 Special interrupt numbers when the Security Extensions are implemented

*Special interrupt numbers* on page 3-11 describes the use of interrupt ID 1023 to indicate a *spurious interrupt*. The full list of the interrupt ID numbers the GIC architecture reserves for special purposes is as follows:

**1020-1021** Reserved.

**1022** Used only if the GIC implements the Security Extensions.

The GIC returns this value to a processor in response to an interrupt acknowledge only when all of the following apply:

- the interrupt acknowledge is a Secure read
- the highest priority pending interrupt is Non-secure
- the AckCtl bit in the Secure ICCICR is set to 0
- the priority of the interrupt is sufficient for it to be signalled to the processor.

**Note**

Interrupt ID 1022 informs Secure software that there is a Non-secure interrupt of sufficient priority to be signalled to the processor, that must be handled by Non-secure software. In this situation the Secure software might alter its schedule to permit Non-secure software to handle the interrupt, to minimize the interrupt latency.

**1023** This value is returned to a processor, in response to an interrupt acknowledge, if there is no pending interrupt with sufficient priority for it to be signalled to the processor.

On a processor that implements the Security Extensions, Secure software treats values of 1022 and 1023 as spurious interrupts.

### 3.4.3 Effect of the Security Extensions on interrupt acknowledgement

When a processor takes an interrupt, it acknowledges the interrupt by reading the ICCIAR, see *General handling of interrupts* on page 3-5. A read of the ICCIAR always acknowledges the highest priority pending interrupt for the processor performing the read.

If the highest priority pending interrupt is a Secure interrupt, the processor must make a Secure read of the ICCIAR to acknowledge it.

By default, the processor must make a Non-secure read of the ICCIAR to acknowledge a Non-secure interrupt. If he AckCtl bit in the Secure ICCICR is set to 1 the processor can make a Secure read of the ICCIAR to acknowledge a Non-secure interrupt.

If the read of the ICCIAR does not match the security of the interrupt, taking account of the AckCtl bit value for a Non-secure interrupt, the ICCIAR read does not acknowledge any interrupt and returns the value:

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

See *Effect of the Security Extensions on reads of the ICCIAR* on page 4-58 for more information.

## 3.5 The effect of the Security Extensions on interrupt prioritization

If the GIC supports the Security Extensions:

- Secure software must program the ICDISRs to configure each supported interrupt as either Secure or Non-secure, see *Interrupt Security Registers (ICDISRn)* on page 4-17
- the GIC provides Secure and Non-secure views of the interrupt priority settings
- the minimum number of priority values supported increases from 16 to 32.

**Note**

Non-secure accesses can see only half of the supported priority values. Therefore, if the GIC implements 32 priority values, Non-secure accesses see only 16 priority values.

### 3.5.1 Software views of interrupt priority

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

For a Secure access, the GIC implements a minimum of 32 and a maximum of 256 priority levels. This means it implements 5-8 bits of the 8-bit priority value fields. Unimplemented low-order bits of the priority fields are RAZ/WI, Figure 3-2 shows the Secure view of a priority value field for a Secure interrupt.

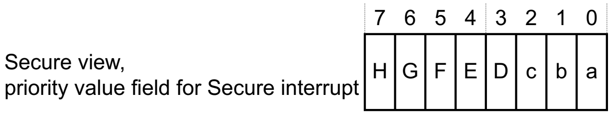


**Figure 3-2 Secure view of the priority field for a Secure interrupt**

In this view:

- bits H-D are the bits that the GIC must implement
- bits c-a are the bits the GIC might implement, that are RAZ/WI if not implemented.

A Non-secure access can only see a priority value field that corresponds to a Non-secure access. For a Non-secure access, the GIC supports half the priority levels it supports for a Secure access. Figure 3-3 shows the Non-secure view of a priority value field for a Non-secure interrupt.

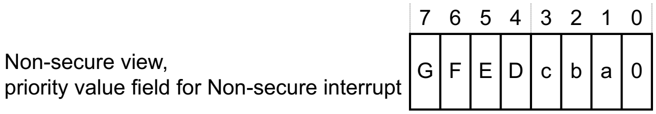


**Figure 3-3 Non-secure view of the priority field for a Non-secure interrupt**

In this view:

- bits G-D are the bits that the GIC must implement
- bits c-a are the bits the GIC might implement, that are RAZ/WI if not implemented
- bit [0] is RAZ/WI.

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

- right-shifts the value by one bit
- sets bit [7] of the value to 1.

This translation means the priority value for the Non-secure interrupt is in the top half of the possible value range, meaning the interrupt priority is in the bottom half of the priority range.

A Secure read of the priority value for a Non-secure interrupt returns the value stored in the distributor. Figure 3-4 shows this Secure view of the priority value field for a Non-secure interrupt that has had its priority value field set by a Non-secure access, or has had a priority value with bit [7] == 1 set by a Secure access:

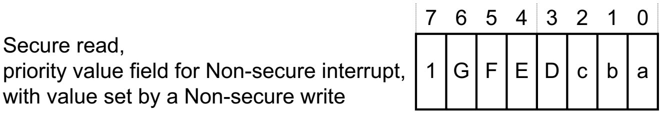

Secure read,
priority value field for Non-secure interrupt,
with value set by a Non-secure write
7 6 5 4 3 2 1 0
1 G F E D c b a

**Figure 3-4 Secure read of the priority field for a Non-secure interrupt**

A Secure write to the priority value field for a Non-secure interrupt can set bit [7] to 0, but see *Recommendations for managing priority values* on page 3-22. If a Secure write sets bit [7] to 0:

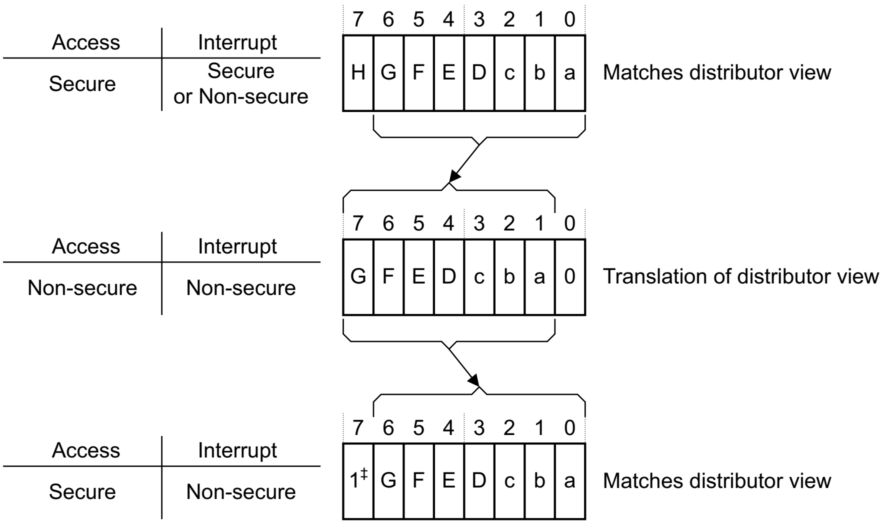
- A Non-secure read returns the value 0bGFEDcba0.
- A Non-secure write can change the value of the field, but only to a value that has bit [7] set to 1 in the distributor view of the field.

**Note**

This behavior of Non-secure accesses applies only to the Priority value fields in the ICDIPR, see *Interrupt Priority Registers (ICDIPRn)* on page 4-31:

- if the Priority field in the ICCPMR holds a value with bit [7] == 0, then the field is RAZ/WI to Non-secure accesses, see *Interrupt Priority Mask Register (ICCPMR)* on page 4-52
- if the Priority field in the ICCRPR holds a value with bit [7] == 0, then the field is RAZ to Non-secure reads, see *Running Priority Register (ICCRPR)* on page 4-61.

Figure 3-5 on page 3-20 shows the relationship between the views of the Priority value fields.



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

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

**Figure 3-5 Relationship between Secure and Non-secure views of interrupt priority fields**

Figure 3-6 on page 3-21 shows how the software views of the interrupt priorities, from Secure and Non-secure accesses, relate to the priority values held in the Distributor, and the interrupt value that are visible to Secure and Non-secure accesses. This is for a GIC that implements the maximum range of priority values.

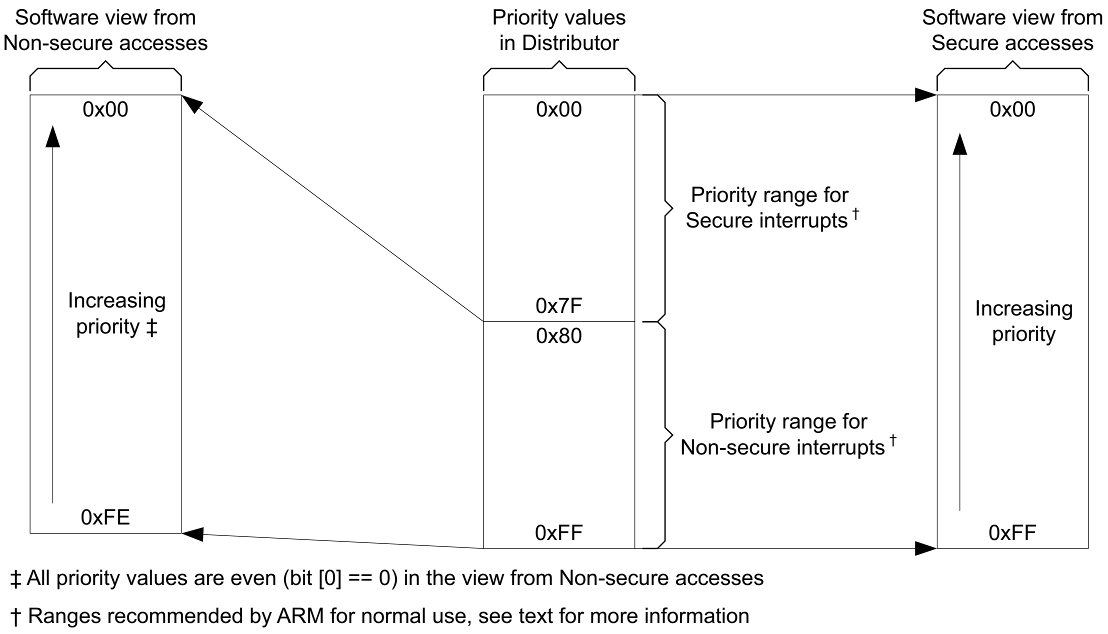

Software view from Non-secure accesses
Priority values in Distributor
Software view from Secure accesses
0x00
0x00
0x00
Priority range for Secure interrupts †
Increasing priority ‡
0x7F
0x80
Increasing priority
Priority range for Non-secure interrupts †
0xFE
0xFF
0xFF
‡ All priority values are even (bit [0] == 0) in the view from Non-secure accesses
† Ranges recommended by ARM for normal use, see text for more information

**Figure 3-6 Software views of the priorities of Non-secure and Secure interrupts**

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

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

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

This model for the presentation of priority values ensures software written to operate with an implementation of this GIC architecture functions as intended regardless of whether the GIC implements the Security Extensions. However, programmers must ensure secure software assigns appropriate priority levels to the Secure and Non-secure interrupts. See *Priority management and the Security Extensions* on page 3-24 for more information.

For more information about priority-related register access restrictions associated with the Security Extensions, see the pseudocode in *Interrupt generation when the GIC implements the Security Extensions* on page 3-23.

### Recommendations for managing priority values

ARM strongly recommends that:

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

This ensures that all Secure interrupts have lower priority values, and therefore higher priorities, than all Non-secure interrupts. However, a system might have requirements that cannot be met with this scheme, see *Priority management and the Security Extensions* on page 3-24.

**Note**

Software might not have any awareness of the Security Extensions, and therefore might not know whether it is making Secure or Non-secure accesses to GIC registers. However, for any implemented interrupt, software can write `0xFF` to the corresponding ICDIPR priority value field, and then read back the value stored in the field to determine the supported interrupt priority range. ARM recommends that, before checking the priority range in this way:

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

## 3.5.2 Control of preemption by Non-secure interrupts

See *Preemption* on page 3-13 and *Priority grouping* on page 3-14 for more information about preemption.

When the GIC implements the Security Extensions, it always uses the Secure ICCBPR to determine whether it signals a Secure interrupt to the processor, for possible preemption, see *Binary Point Register (ICCBPR)* on page 4-54. By default, it uses the Non-secure ICCBPR to determine whether to signal a Non-secure interrupt for possible preemption. However, Secure software can configure the CPU interface to always use the Secure ICCBPR for determining possible preemption, for both Secure and Non-Secure interrupts. To do this, it sets the SBPR bit in the Secure ICCICR to 1, see *CPU Interface Control Register (ICCICR)* on page 4-47.

## 3.5.3 The effect of the Security Extensions on priority masking

This section describes how the Security Extensions change the information given in *Priority masking* on page 3-13.

If the GIC implements the Security Extensions, the ICCPMR is RAZ/WI to Non-secure accesses if it holds a value with bit [7] == 0. In normal operation, Non-secure software does not access the ICCPMR when it is programmed with such a value. For more information see *Interrupt Priority Mask Register (ICCPMR)* on page 4-52 and *Non-secure access to register fields for Secure interrupt priorities* on page 4-8.

### 3.5.4 The effect of the Security Extensions on priority grouping

For Secure interrupts, the priority grouping behavior is as described in *Priority grouping* on page 3-14. For Non-secure interrupts, priority grouping is modified so that, for each binary point value, one bit moves from the subpriority field to the group priority field.

**Note**

Priority grouping always operates on the priority value held in the Distributor, not the value visible to a Non-secure read of the priority value corresponding to a Non-secure interrupt. See Figure 3-5 on page 3-20 and Figure 3-6 on page 3-21.

Table 3-2 on page 3-14 shows the priority grouping for Non-secure interrupts.

**Table 3-4 Priority grouping for Non-secure interrupts**

| Binary point value | Interrupt priority field [7:0] | | |
|---|---|---|---|
| | Group priority field | Subpriority field | Field with binary point |
| 0 | [7:0][a] | - | gggggggg. |
| 1 | [7:1][a] | [0] | ggggggg.s |
| 2 | [7:2][a] | [1:0] | gggggg.ss |
| 3 | [7:3][a] | [2:0] | ggggg.sss |
| 4 | [7:4][a] | [3:0] | gggg.ssss |
| 5 | [7:5][a] | [4:0] | ggg.sssss |
| 6 | [7:6][a] | [5:0] | gg.ssssss |
| 7 | [7][a] | [6:0] | g.sssssss |

[a] If a Non-secure write sets the priority value field for a Non-secure interrupt then bit [7] is 1.

The right shift of the binary point for Non-secure interrupts, and the views of interrupt priority described in *Software views of interrupt priority* on page 3-18, mean that software that has no awareness of the Security Extensions sees the same priority grouping mechanism regardless of whether it is running in Secure state or in Non-secure state.

### 3.5.5 Interrupt generation when the GIC implements the Security Extensions

The pseudocode in *Exception generation pseudocode, with the Security Extensions* on page 3-28 describes the generation of interrupts by the GIC when the GIC implements the Security Extensions.

### 3.5.6 Priority management and the Security Extensions

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

**Non-cooperative** All Secure interrupts have higher priority than any Non-secure interrupt, and can always preempt any Non-secure interrupt.

**Co-operative** Secure and Non-secure software interact to program some Secure interrupts with lower priority than some Non-secure interrupts.

Secure software is software that can make Secure accesses to the GIC, and might be able to make Non-secure accesses. Non-secure software can make only Non-secure accesses.

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

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

**Note**

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

## 3.6 Pseudocode details of interrupt handling and prioritization

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

- *General helper functions and definitions*
- *Exception generation pseudocode, without the Security Extensions* on page 3-27
- *Exception generation pseudocode, with the Security Extensions* on page 3-28
- *The effect of the Security Extensions on accesses to prioritization registers* on page 3-29.

### 3.6.1 General helper functions and definitions

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

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

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

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

boolean IsSecureInt(integer InterruptID) // Returns TRUE if the bitfield in the ICDISR associated
                                         // with the argument InterruptID is set to 1, indicating
                                         // that the interrupt is configured as a Secure
                                         // interrupt.

boolean IsPending(integer InterruptID)   // Returns TRUE if the interrupt specified by the
                                         // argument InterruptID is pending.

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

bits(8) ReadICDIPR(integer InterruptID)  // Returns the 8-bit priority field from the ICDIPR
                                         // associated with the argument InterruptID.

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

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


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

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

```
// bits(8) GIC_PriorityMask()
// ==========================

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

bits(8) GIC_PriorityMask(integer n, bit NS_mask) // Calculate the Binary Point (group) mask.
    assert n >= 0 && n <= 7;                     // Range check for the priority mask.
    if n < MINIMUM_BINARY_POINT then             // Saturate n on the minimum value supported.
        n = MINIMUM_BINARY_POINT;
    if NS_mask == ‘0’ then                       // Mask generation for a secure GIC access.
        n = n + 1;
    mask = ‘1111111100000000’<15-n:8-n>;        // Generate the 8-bit group priority mask.
    return mask;


// Global variables
// ================

integer CPU_INTERFACE_ID   // An identifier for a specific CPU Interface. The value of this
                           // variable has implicit effects on which CPU interface register,
                           // CPU interface signal or banked version of a Distributor
                           // register is accessed.

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

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


// global constants
// ================

integer    MINIMUM_BINARY_POINT   // A minimum binary point value of 0,1,2 or 3,
                                  // this is an IMPLEMENTATION DEFINED value.


bits(8)    P_MASK       // IMPLEMENTATION DEFINED mask of valid priority bits:
                        //   For systems without the Security Extensions, supported
                        //   values are 0xF0, 0xF8, 0xFC, 0xFE and 0xFF.
                        //   For systems with the Security Extensions, supported
                        //   values are 0xF8, 0xFC, 0xFE and 0xFF.



// registers associated with prioritization control and configuration
// ==================================================================

enumeration RegName {RegName_ICDIPR, RegName_ICCPMR, RegName_ICCRPR}
```

### 3.6.2 Exception generation pseudocode, without the Security Extensions

The following pseudocode describes how exceptions are generated by a CPU interface that does not implement the Security Extensions. It shows all the GIC prioritization.

```
//
// GIC_GenerateException()
// =======================
//
void GIC_GenerateException()
    while TRUE do                            // Loop continuously.
        if ICDDCR<0> == ‘1’ then             // GIC Distributor enabled.

            cpu_count = UInt(ICDICTR<7:5>);   // Determine the number of CPU interfaces.

            for CPU_INTERFACE_ID = 0 to cpu_count
                                              // The iterations of this loop are permitted to
                                              // occur in parallel.
                next_irq = FALSE;
                intID = ICCHPIR<9:0>;         // Establish the ID of the highest pending
                                              // interrupt on the appropriate CPU interface.


                if PriorityIsHigher(ReadICDIPR(intID), ICCPMR<7:0>) && IsPending(intID) &&
                   ICCICR.Enable == ‘1’) then
                    mask = GIC_PriorityMask(ICCBPR<2:0>, ‘0’);
                    if !AnyActiveInterrupts() then
                                              // No active interrupts.
                        next_irq = TRUE;
                    else                      // Currently active interrupt(s).
                        if PriorityIsHigher(ReadICDIPR(intID), ICCRPR<7:0> AND mask) then
                            next_irq = TRUE;

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

### 3.6.3 Exception generation pseudocode, with the Security Extensions

The following pseudocode describes how exceptions are generated by a CPU interface that implements the Security Extensions. It shows all the GIC prioritization and the security state information.

```
//
// GenerateException()
// ==================
//
void GIC_GenerateException()
    while TRUE do                           // Loop continuously.
        if ICDDCR<0> == '1' then            // GIC Distributor enabled.
            cpu_count = UInt(ICDICTR<7:5>);   // Determine the number of CPU interfaces.

            for CPU_INTERFACE_ID = 0 to cpu_count
                                            // The iterations of this loop are permitted to
                                            // occur in parallel.
                sbp   = UInt(ICCBPR<2:0>);    // Secure version of this register.
                nsbp  = UInt(ICCABPR<2:0>);
                next_fiq = FALSE;
                next_irq = FALSE;
                intID = ICCHPIR<9:0>;         // Establish the ID of the highest pending
                                            // interrupt on the appropriate CPU interface.

                if PriorityIsHigher(ReadICDIPR(intID), ICCPMR<7:0>) && IsPending(intID)

                    smsk = GIC_PriorityMask(sbp, '0');
                    if ICCICR.SBPR == '1' then
                        nsmsk = smsk;
                    else
                        nsmsk = GIC_PriorityMask(nsbp, '1');

                    if !AnyActiveInterrupts() then       // No active interrupt.
                        if IsSecureInt(intID) && (ICCICR.EnableS == '1') then
                            if ICCICR.FIQEn == '1' then
                                next_fiq = TRUE;
                            else
                                next_irq = TRUE;         // Secure interrupt signaled on IRQ.
                        if !IsSecureInt(intID) && (ICCICR.EnableNS == '1') then
                            next_irq = TRUE
                    else                      // Currently active interrupt(s).
                        if IsSecureInt(intID) && (ICCICR.EnableS == '1') then
                            if PriorityIsHigher(ReadICDIPR(intID), ICCRPR<7:0> AND smsk) then
                                if ICCICR.FIQEn == '1' then
                                    next_fiq = TRUE;
                                else
                                    next_irq = TRUE;
                        else                  // Highest pending interrupt is non-secure.
                            if (!IsSecureInt(intID)) && (ICCICR.EnableNS == '1') then
                               if PriorityIsHigher(ReadICDIPR(intID), ICCRPR<7:0> AND nsmsk) then
                                    next_irq = TRUE;

                SignalFIQ(next_fiq);          // Update driven status of FIQ.
                SignalIRQ(next_irq);          // Update driven status of IRQ.
```

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

The Security Extensions change some of the behavior of accesses to the ICDIPR, ICCPMR, and ICCRPR see *Non-secure access to register fields for Secure interrupt priorities* on page 4-8. The following pseudocode functions define the behavior:

- `PriorityRegWrite(integer InterruptID, RegName Register, bits(8) value)`
- `bits(8) PriorityRegRead(integer InterruptID, RegName Register).`

```
//
// PriorityRegWrite()
// ==================
//

PriorityRegWrite(integer InterruptID, RegName Register, bits(8) value)

    when RegName_ICDIPR
        if NS_access then                    // A non-secure GIC access.
            if !IsSecureInt(InterruptID) then
                mod_write_val = ('10000000' OR LSR(value,1)) AND P_MASK;
                WriteICDIPR(InterruptID, mod_write_val);
            else
                IgnoreWriteRequest();
        else                                 // A secure GIC access.
            mod_write_val = value AND P_MASK;
            WriteICDIPR(InterruptID, mod_write_val);

    when RegName_ICCPMR
        if NS_access then                    // A non-secure GIC access.
            mod_write_val = ('10000000' OR LSR(value,1)) AND P_MASK;
            if ICCPMR<7> == '1' then         // Non-secure execution can only update the
                ICCPMR<7:0> = mod_write_val; // Priority Mask Register if the current
                                             // value is in the range 0x80 to 0xFF.
            else
                IgnoreWriteRequest();
        else                                 // A secure GIC access.
            ICCPMR<7:0> = value AND P_MASK;




//
// PriorityRegRead()
// =================
//

// P_MASK used here to emphasize that the number of valid bits is IMPLEMENTATION DEFINED

bits(8) PriorityRegRead(integer InterruptID, RegName Register)

    when RegName_ICDIPR
        read_value = ReadICDIPR(InterruptID);
        if NS_access then                    // A non-secure GIC access.
            read_value<7:0> = LSL((read_value AND P_MASK),1);
            if IsSecureInt(InterruptID) then
            read_value = ‘00000000’;     // Can’t read a secure priority value.
    return(read_value);

when RegName_ICCPMR
    read_value = ICCPMR<7:0>;
    if NS_access then                    // A non-secure GIC access.
        if read_value <7> == ‘0’ then
            read_value = ‘00000000’;     // A secure priority value, RAZ.
        else
            read_value<7:0> = LSL((read_value AND P_MASK),1);
    return(read_value);

when RegName_ICCRPR
    read_value = ICCRPR<7:0>;
    if NS_access then                    // A non-secure GIC access.
        if read_value <7> == ‘0’ then
            read_value = ‘00000000’;     // A secure priority value, RAZ.
        else
            read_value<7:0> = LSL((read_value AND P_MASK),1);
    return(read_value);
```

# Chapter 4
# Programmers Model

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

- *About the programmers model* on page 4-2
- *Effect of the Security Extensions on the programmers model* on page 4-7
- *Distributor register descriptions* on page 4-11
- *CPU interface register descriptions* on page 4-46.

## 4.1 About the programmers model

The programmers model provides the software interface to the GIC. This chapter describes the programmers model, that operates using a memory-mapped register interface.

The following sections describe the programmers model:

- *GIC register short names*
- *Distributor register map*
- *CPU interface register map* on page 4-4
- *GIC register access* on page 4-5
- *Reset behavior* on page 4-6
- *Effect of the Security Extensions on the programmers model* on page 4-7.

Table 4-1 and Table 4-2 on page 4-4 describe the register access type as follows:

| | |
|---|---|
| **RW** | Read and write. |
| **RO** | Read only. |
| **WO** | Write only. |

### 4.1.1 GIC register short names

All of the GIC registers have short names. In these names:

- the first two letters are IC, indicating a GIC register
- the third letter is one of:
  - — D, indicating a distributor register
  - — C, indicating a CPU interface register
- the remaining letters are a mnemonic for the register, for example ABR for Active Bit Register.

### 4.1.2 Distributor register map

Table 4-1 shows the Distributor register map. Address offsets are relative to the *Distributor base address* defined by the GIC system memory map.

All GIC registers are 32-bits wide. Reserved register addresses are RAZ/WI.

**Table 4-1 Distributor register map**

| Offset | Name[a] | Type | Reset[b] | Description |
|---|---|---|---|---|
| `0x000` | ICDDCR | RW | `0x00000000` | *Distributor Control Register (ICDDCR)* on page 4-12 |
| `0x004` | ICDICTR | RO | IMPLEMENTATION DEFINED | *Interrupt Controller Type Register (ICDICTR)* on page 4-14 |
| `0x008` | ICDIIDR | RO | IMPLEMENTATION DEFINED | *Distributor Implementer Identification Register (ICDIIDR)* on page 4-16 |

**Table 4-1 Distributor register map (continued)**

| Offset | Name[a] | Type | Reset[b] | Description |
|---|---|---|---|---|
| `0x00C-0x07C` | - | - | - | Reserved |
| `0x080` | ICDISR | RW | IMPLEMENTATION DEFINED[c] | *Interrupt Security Registers (ICDISRn)* on page 4-17[d] |
| `0x084-0x0FC` | | | `0x00000000` | |
| `0x100-0x17C` | ICDISER | RW | IMPLEMENTATION DEFINED | *Interrupt Set-Enable Registers (ICDISERn)* on page 4-19 |
| `0x180-0x1FC` | ICDICER | RW | IMPLEMENTATION DEFINED | *Interrupt Clear-Enable Registers (ICDICERn)* on page 4-21 |
| `0x200-0x27C` | ICDISPR | RW | `0x00000000` | *Interrupt Set-Pending Registers (ICDISPRn)* on page 4-23 |
| `0x280-0x2FC` | ICDICPR | RW | `0x00000000` | *Interrupt Clear-Pending Registers (ICDICPRn)* on page 4-26 |
| `0x300-0x37C` | ICDABR | RO | `0x00000000` | *Active Bit Registers (ICDABRn)* on page 4-29 |
| `0x380-0x3FC` | - | - | - | Reserved |
| `0x400-0x7F8` | ICDIPR | RW | `0x00000000` | *Interrupt Priority Registers (ICDIPRn)* on page 4-31 |
| `0x7FC` | - | - | - | Reserved |
| `0x800-0x81C` | ICDIPTR | RO[e] | IMPLEMENTATION DEFINED | *Interrupt Processor Targets Registers (ICDIPTRn)* on page 4-33 |
| `0x820-0xBF8` | | RW[e] | `0x00000000` | |
| `0xBFC` | - | - | - | Reserved |
| `0xC00-0xCFC` | ICDICFR | RW | IMPLEMENTATION DEFINED | *Interrupt Configuration Registers (ICDICFRn)* on page 4-36 |
| `0xD00-0xDFC` | - | - | - | IMPLEMENTATION DEFINED registers |
| `0xE00-0xEFC` | - | - | - | Reserved |
| `0xF00` | ICDSGIR | WO | - | *Software Generated Interrupt Register (ICDSGIR)* on page 4-39 |

**Table 4-1 Distributor register map (continued)**

| Offset | Name[a] | Type | Reset[b] | Description |
|---|---|---|---|---|
| 0xF04-0xFCC | - | - | - | Reserved |
| 0xFD0-0xFFC | - | RO | IMPLEMENTATION DEFINED | *Identification registers* on page 4-42 |

a. For legacy shortform register names see Appendix C *Register Shortform Names*.

b. For details of any restrictions that apply to the reset values of IMPLEMENTATION DEFINED cases, for example architecturally-required bit values, see the appropriate register description.

c. For more information see *ICDISR0 reset value* on page 4-18.

d. Present only if the GIC implements the Security Extensions, otherwise RAZ/WI.

e. In a uniprocessor implementation, these registers are RAZ/WI.

### 4.1.3 CPU interface register map

Table 4-2 shows the CPU interface register map. Address offsets are relative to the *CPU interface base address* defined by the system memory map.

All GIC registers are 32-bits wide. Reserved register addresses are RAZ/WI.

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

**Table 4-2 CPU interface register map**

| Offset | Name[a] | Type | Reset | Description |
|---|---|---|---|---|
| 0x00 | ICCICR | RW | 0x00000000 | *CPU Interface Control Register (ICCICR)* on page 4-47 |
| 0x04 | ICCPMR | RW | 0x00000000 | *Interrupt Priority Mask Register (ICCPMR)* on page 4-52 |
| 0x08 | ICCBPR | RW | 0x00000000-0x00000003[b] | *Binary Point Register (ICCBPR)* on page 4-54 |
| 0x0C | ICCIAR | RO | 0x000003FF | *Interrupt Acknowledge Register (ICCIAR)* on page 4-56 |
| 0x10 | ICCEOIR | WO | - | *End of Interrupt Register (ICCEOIR)* on page 4-59 |
| 0x14 | ICCRPR | RO | 0x000000FF | *Running Priority Register (ICCRPR)* on page 4-61 |
| 0x18 | ICCHPIR | RO | 0x000003FF | *Highest Pending Interrupt Register (ICCHPIR)* on page 4-63 |
| 0x1C | ICCABPR | RW | 0x00000000 | *Aliased Binary Point Register (ICCABPR)* on page 4-62 |

**Table 4-2 CPU interface register map (continued)**

| Offset | Name[a] | Type | Reset | Description |
|---|---|---|---|---|
| `0x20-0x3C` | - | - | - | Reserved |
| `0x40-0xCF` | - | - | - | IMPLEMENTATION DEFINED registers |
| `0xD0-0xF8` | - | - | - | Reserved |
| `0xFC` | ICCIIDR | RO | IMPLEMENTATION DEFINED | *CPU Interface Identification Register (ICCIIDR)* on page 4-65 |

[a] For legacy shortform register names see Appendix C *Register Shortform Names*.

[b] See the register description for more information.

### 4.1.4 GIC register access

All registers support 32-bit word accesses with the access type defined in Table 4-1 on page 4-2 and Table 4-2 on page 4-4.

In addition, the following registers support byte accesses:

- ICDIPR, see *Interrupt Priority Registers (ICDIPRn)* on page 4-31
- ICDIPTR, see *Interrupt Processor Targets Registers (ICDIPTRn)* on page 4-33.

Halfword register accesses are IMPLEMENTATION DEFINED.

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

- some registers are banked, see *Register banking*
- some registers are accessible only using Secure accesses
- optionally, the GIC supports lockdown of the values of some registers.

For more information see *Effect of the Security Extensions on the programmers model* on page 4-7.

#### Register banking

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

- If the GIC implements the Security Extensions, some registers are banked to provide separate Secure and Non-secure copies of the registers. The register bit assignments can differ in the Secure and Non-secure copies of a register. A secure access to the register address accesses the Secure copy of the register, and a Non-secure access accesses the Non-secure copy. See *Effect of the Security Extensions on the programmers model* on page 4-7 for more information.

- If the GIC is implemented as part of a multiprocessor system, some Distributor registers are banked to provide a separate copy for each connected processor.

  **Note**

  The GIC implements the CPU interface registers independently for each CPU interface, and each connected processor accesses these registers for the interface it connects to.

### 4.1.5 Reset behavior

On exit from reset, the GIC clears the enable bits in the ICDDCR and ICCICR registers to 0. This means that software can program the Distributor and CPU interface registers before enabling the GIC.

When ICDDCR.Enable is 0:

- software can read or write the Distributor registers
- any pending interrupts are not forwarded to the CPU interfaces.

When ICCICR.Enable is 0:

- software can read or write the CPU interface registers
- any pending interrupts are not forwarded to the processor
- any read of the ICCIAR or ICCHPIR returns a spurious interrupt ID.

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

**Note**

For an overview of the GIC implementation of the ARM Security Extensions, see *Security Extensions support* on page 1-3.

If the GIC implements the Security Extensions, the ICDICTR.SecurityExtn bit is RAO, see *Interrupt Controller Type Register (ICDICTR)* on page 4-14.

A GIC implementation of the Security Extensions provides the following features:

- Each supported interrupt is either Secure or Non-secure:
  - a GIC might implement some interrupts as always Secure, or as always Non-secure
  - otherwise, software configures each interrupt as Secure or Non-secure
  - some aspects of interrupt handling depend on whether interrupts are Secure or Non-secure.
- Accesses to the GIC registers are either Secure or Non-secure, see *Processor security state and Secure and Non-secure GIC accesses* on page 1-5.

In normal operation, Secure software accesses the GIC using only Secure accesses.

Table 4-3 shows the registers that are implemented differently as part of the Security Extensions. All registers not listed in Table 4-3 are Common registers.

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

| Register | Type | See: | Effect |
|---|---|---|---|
| Distributor registers | | | |
| ICDDCR | Banked | *Distributor Control Register (ICDDCR)* on page 4-12 | Register is banked[a] |
| ICDICTR | Common | *Interrupt Controller Type Register (ICDICTR)* on page 4-14 | Adds the LSPI field |
| ICDISR | Secure | *Interrupt Security Registers (ICDISRn)* on page 4-17 | Register is Secure |
| ICDSGIR | Common | *Software Generated Interrupt Register (ICDSGIR)* on page 4-39 | Adds the SATT bit |
| CPU interface registers | | | |
| ICCICR | Banked | *CPU Interface Control Register (ICCICR)* on page 4-47 | Register is banked[a] |
| ICCBPR | Banked | *Binary Point Register (ICCBPR)* on page 4-54 | Register is banked[a] |
| ICCABPR | Secure | *Aliased Binary Point Register (ICCABPR)* on page 4-62 | Register is Secure |

[a] Banked to provide Secure and Non-secure copies of the register, see *Register banking* on page 4-5.

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

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

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

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

In addition, in a GIC that implements the Security Extensions, the priority range available for Non-secure interrupts is half the range available for Secure interrupts, see *The effect of the Security Extensions on interrupt prioritization* on page 3-18.

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

- *Non-secure access to register fields for Secure interrupt priorities*
- *Configuration lockdown* on page 4-9.

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

Most register fields associated with a Secure interrupts are RAZ/WI to Non-secure accesses. The following Non-secure register accesses are exceptions to this rule:

**Non-secure access to an access priority field in the ICDIPRs**

- If the priority field corresponds to a Non-secure interrupt the access operates as defined by the Non-secure view of interrupt priority, see *Software views of interrupt priority* on page 3-18.
- If the priority field corresponds to a Secure interrupt, the field is RAZ/WI.

**Non-secure access to the ICCPMR and ICCRPR**

- If the current priority mask value is in the range `0x00-0x7F`:
  - — a read access returns the value `0x00`
  - — the GIC ignores a write access to the ICCPMR.
- If the current priority mask value is in the range `0x80-0xFF`:
  - — A read access returns the Non-secure view of the current value.
  - — A write access to the ICCPMR succeeds, based on the Non-secure view of the priority mask value written to the register. This means a Non-secure write cannot set a priority mask value in the rage `0x00-0x7F`.

The pseudocode in *The effect of the Security Extensions on accesses to prioritization registers* on page 3-29 describes accesses to the ICDIPRs, ICCPMR, and ICCRPR when the GIC implements the Security Extensions.

### 4.2.2 Configuration lockdown

A GIC that implements the Security Extensions can also implement configuration lockdown. This provides a control signal that the system can assert to prevent write access to the register fields controlling a configured range of SPIs, when those SPIs are configured as Secure interrupts, and to some configuration registers. When the control signal is asserted, the Secure SPIs and configuration registers are described as being *locked down*.

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

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

- If the GIC supports any LSPIs then the first possible LSPI has Interrupt ID 32
- The ICDICTR.LSPI field defines the maximum number of LSPIs, see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. If ICDICTR.LSPI is greater than 0 then the possible LSPIs have interrupt IDs 32 to (31+(ICDICTR.LSPI)).

  **Note**

  ICDICTR.LSPI only defines the range of possible LSPIs. The GIC might not support all the interrupts in this range.

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

When the SPIs and configuration registers are locked down, the GIC prevents write accesses to:

- The secure copy of the ICDDCR, see *Distributor Control Register (ICDDCR)* on page 4-12.
- All bits of the secure copy of the ICCICR, except for the EnableNS bit, see *CPU Interface Control Register (ICCICR)* on page 4-47. You can still write to the ICCICR.EnableNS bit.
- Fields in the following registers that correspond to Lockable SPIs that are configured as Secure:
  - ICDISERs, see *Interrupt Set-Enable Registers (ICDISERn)* on page 4-19
  - ICDICERs, see *Interrupt Clear-Enable Registers (ICDICERn)* on page 4-21
  - ICDISPRs, see *Interrupt Set-Pending Registers (ICDISPRn)* on page 4-23
  - ICDICPRs, see *Interrupt Clear-Pending Registers (ICDICPRn)* on page 4-26
  - ICDIPRs, see *Interrupt Priority Registers (ICDIPRn)* on page 4-31
  - ICDIPTRs, see *Interrupt Processor Targets Registers (ICDIPTRn)* on page 4-33
  - ICDICFRs, see *Interrupt Configuration Registers (ICDICFRn)* on page 4-36.
- Fields in the ICDISRs that correspond to lockable SPIs that you have configured as Secure, see *Interrupt Security Registers (ICDISRn)* on page 4-17. If you reconfigure a lockable SPI from Non-secure to Secure while **CFGSDISABLE** remains HIGH, the GIC prevents any further writes to ICDISR fields that correspond to that SPI.

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

**Note**

- ARM recommends that, during the system boot process, the system reads the ICDICTR.LSPI field to find the number of lockable SPIs, programs the registers and register fields that can be locked down, and then asserts **CFGSDISABLE** HIGH. Normally, this means that the Secure boot sequence that follows a full system reset must run appropriate Secure configuration code.
- ARM strongly recommends that once **CFGSDISABLE** is first asserted HIGH during the system boot process, the system ensures **CFGSDISABLE** cannot be deasserted except by a processor reset.

## 4.3 Distributor register descriptions

The following sections describe the Distributor registers:

- *Distributor Control Register (ICDDCR)* on page 4-12
- *Interrupt Controller Type Register (ICDICTR)* on page 4-14
- *Distributor Implementer Identification Register (ICDIIDR)* on page 4-16
- *Interrupt Security Registers (ICDISRn)* on page 4-17
- *Interrupt Set-Enable Registers (ICDISERn)* on page 4-19
- *Interrupt Clear-Enable Registers (ICDICERn)* on page 4-21
- *Interrupt Set-Pending Registers (ICDISPRn)* on page 4-23
- *Interrupt Clear-Pending Registers (ICDICPRn)* on page 4-26
- *Active Bit Registers (ICDABRn)* on page 4-29
- *Interrupt Priority Registers (ICDIPRn)* on page 4-31
- *Interrupt Processor Targets Registers (ICDIPTRn)* on page 4-33
- *Interrupt Configuration Registers (ICDICFRn)* on page 4-36
- *Software Generated Interrupt Register (ICDSGIR)* on page 4-39
- *Identification registers* on page 4-42.

### 4.3.1 Distributor Control Register (ICDDCR)

The ICDDCR characteristics are:

**Purpose** Enables the forwarding of pending interrupts to the CPU interfaces.

**Usage constraints** If the GIC implements the Security Extensions with configuration lockdown, the system can lock down the Secure ICDDCR, see *Configuration lockdown* on page 4-9.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions, this register is banked to provide Secure and Non-secure copies, see *Register banking* on page 4-5.

**Attributes** See the register summary in Table 4-1 on page 4-2.

Figure 4-1 shows the ICDDCR bit assignments.

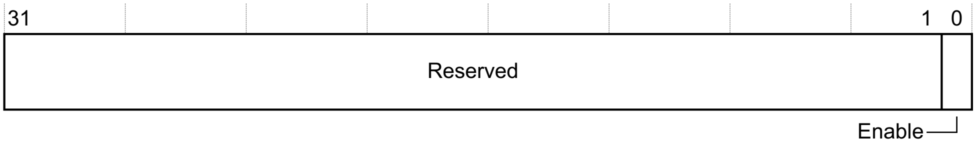


**Figure 4-1 ICDDCR bit assignments**

Table 4-4 shows the ICDDCR bit assignments.

**Table 4-4 ICDDCR bit assignments**

| Bits | Name | Function |
|---|---|---|
| [31:1] | - | Reserved. |
| [0] | Enable | Global enable for monitoring peripheral interrupt signals and forwarding pending interrupts to the CPU interfaces.<br>**0** The GIC ignores all peripheral interrupt signals, and does not forward pending interrupts to the CPU interfaces.<br>**1** The GIC monitors the peripheral interrupt signals, and forwards pending interrupts to the CPU interfaces. |

When ICDDCR.Enable is set to 0, disabling the distributor functions, other GIC register read and writes still operate normally. This means software can change the state of PPIs and SPIs before re-enabling the distributor. For example, software can:

- Make an interrupt pending by writing to the corresponding ICDISPR or the ICDSGIR, see:
  — *Interrupt Set-Pending Registers (ICDISPRn)* on page 4-23
  — *Software Generated Interrupt Register (ICDSGIR)* on page 4-39.
- Remove the pending state of an interrupt by writing to the corresponding ICDICPR, see *Interrupt Clear-Pending Registers (ICDICPRn)* on page 4-26. If the interrupt is level sensitive and the corresponding interrupt signal is asserted, the interrupt becomes pending again as soon as the software re-enables the distributor.
- Remove the active state from an interrupt by writing to the corresponding ICCEOIR, see *End of Interrupt Register (ICCEOIR)* on page 4-59.

**Note**

Setting ICDDCR.Enable to 0 disables forwarding of interrupts to the CPU interfaces, and therefore software cannot add the active state to any interrupt. The GIC clears the pending state of an SGI only when the SGI becomes active, and therefore software cannot clear the pending state of an SGI.

### 4.3.2 Interrupt Controller Type Register (ICDICTR)

The ICDICTR characteristics are:

**Purpose** Provides information about the configuration of the GIC. It indicates:

- whether the GIC implements the Security Extensions
- the maximum number of interrupt IDs that the GIC supports
- the number of CPU interfaces implemented
- if the GIC implements the Security Extensions, the maximum number of implemented *Lockable Shared Peripheral Interrupts* (LSPIs).

**Usage constraints** No usage constraints.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.

**Attributes** See the register summary in Table 4-1 on page 4-2.

Figure 4-2 shows the ICDICTR Register bit assignments.

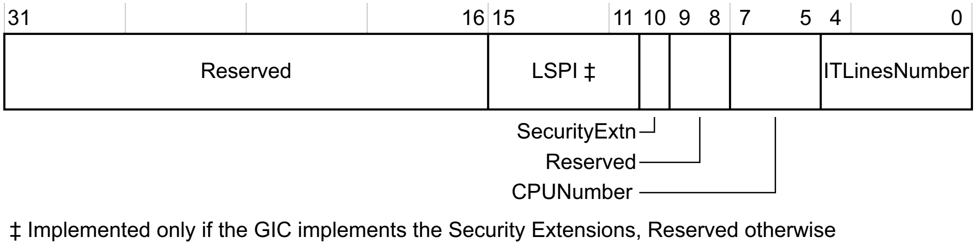

31
16
15
11
10
9
8
7
5
4
0
Reserved
LSPI ‡
ITLinesNumber
SecurityExtn
Reserved
CPUNumber
‡ Implemented only if the GIC implements the Security Extensions, Reserved otherwise

**Figure 4-2 ICDICTR Register bit assignments**

Table 4-5 shows the ICDICTR Register bit assignments.

**Table 4-5 ICDICTR Register bit assignments**

| Bits | Name | Function |
|---|---|---|
| [31:16] | - | Reserved. |
| [15:11] | - | If the GIC does not implement the Security Extensions, this field is Reserved. |
| [15:11] | LSPI | If the GIC implements the Security Extensions, the value of this field is the maximum number of implemented lockable SPIs, from 0 (0b00000) to 31 (0b11111), see *Configuration lockdown* on page 4-9. If this field is 0b00000 then the GIC does not implement configuration lockdown. |

Table 4-5 ICDICTR Register bit assignments (continued)

| Bits | Name | Function |
|---|---|---|
| [10] | SecurityExtn | Indicates whether the GIC implements the Security Extensions.<br>**0** Security Extensions not implemented.<br>**1** Security Extensions implemented. |
| [9:8] | - | Reserved. |
| [7:5] | CPUNumber | Indicates the number of implemented CPU interfaces. The number of implemented CPU interfaces is one more than the value of this field, for example if this field is 0b011, there are four CPU interfaces. |
| [4:0] | ITLinesNumber | Indicates the maximum number of interrupts that the GIC supports[a]. If the value of this field is N, the maximum number of interrupts is 32(N+1). The interrupt ID range is from 0 to one less than the number of IDs. For example:<br>**0b00011** Up to 128 interrupt lines, interrupt IDs 0-127.<br>The maximum number of interrupts is 1020 (0b11111).<br>See the text in this section for more information. |

[a] Regardless of the range of interrupt IDs defined by this field, interrupt IDs 1020-1023 are reserved for special purposes, see *Special interrupt numbers* on page 3-11. For more information about interrupt IDs, see *Interrupt IDs* on page 2-4.

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

- *Interrupt Security Registers (ICDISRn)* on page 4-17
- *Interrupt Set-Enable Registers (ICDISERn)* on page 4-19
- *Interrupt Clear-Enable Registers (ICDICERn)* on page 4-21
- *Interrupt Set-Pending Registers (ICDISPRn)* on page 4-23
- *Interrupt Clear-Pending Registers (ICDICPRn)* on page 4-26
- *Active Bit Registers (ICDABRn)* on page 4-29
- *Interrupt Priority Registers (ICDIPRn)* on page 4-31
- *Interrupt Processor Targets Registers (ICDIPTRn)* on page 4-33
- *Interrupt Configuration Registers (ICDICFRn)* on page 4-36.

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

### 4.3.3 Distributor Implementer Identification Register (ICDIIDR)

The ICDIIDR characteristics are:

**Purpose** Provides information about the implementer and revision of the Distributor.

**Usage constraints** No usage constraints.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.

**Attributes** See the register summary in Table 4-1 on page 4-2.

Figure 4-3 shows the ICDIIDR bit assignments.

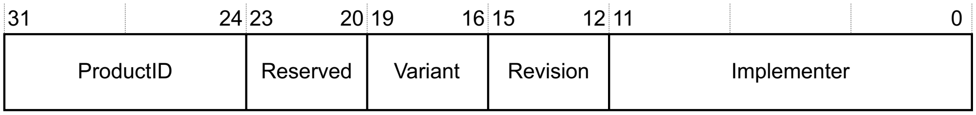


**Figure 4-3 ICDIIDR bit assignments**

Table 4-6 shows the ICDIIDR bit assignments.

**Table 4-6 ICDIIDR bit assignments**

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

[a] For an ARM implementation, the value of this field is `0x43B`.

### 4.3.4 Interrupt Security Registers (ICDISRn)

The ICDISR characteristics are:

**Purpose** The ICDISRs provide a Security status bit for each interupt supported by the GIC. Each bit controls the security status of the corresponding interrupt.

**Usage constraints** Accessible by Secure accesses only. The register addresses are RAZ/WI to Non-secure accesses.

A register bit corresponding to an unimplemented interrupt is RAZ/WI.

If the GIC implements configuration lockdown, the system can lockdown the Security status bits for the lockable SPIs that are configured as Secure, see *Configuration lockdown* on page 4-9.

**Configurations** Secure registers, only implemented if the GIC implements the Security Extensions. If the GIC does not implement the Security Extensions the ICDISR addresses are RAZ/WI.

The number of implemented ICDISRs is (ICDICTR.ITLinesNumber + 1), see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. The implemented ICDISRs number upwards from ICDISR0.

In a multiprocessor implementation, ICDISR0 is banked for each connected processor, see *Register banking* on page 4-5. This register holds the security status bits for interrupts 0-31.

**Attributes** See the register summary in Table 4-1 on page 4-2, and *ICDISR0 reset value* on page 4-18.

Figure 4-4 shows the ICDISR bit assignments.

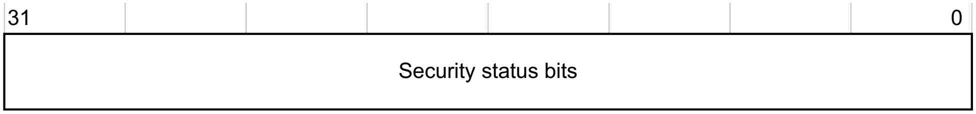


**Figure 4-4 ICDISR bit assignments**

Table 4-7 shows the ICDISR bit assignments.

**Table 4-7 ICDISR bit assignments**

| Bits | Name | Function | |
|---|---|---|---|
| [31:0] | Security status bits | For each bit: | |
| | | **0** | The corresponding interrupt is Secure. |
| | | **1** | The corresponding interrupt is Non-secure. |

**Note**

On start-up or reset, each interrupt with ID32 or higher resets as Secure and therefore all SPIs are Secure unless the system reprograms the appropriate ICDISR bit. See *ICDISR0 reset value* for information about the reset security configuration of interrupts with IDs 0-31.

ARM recommends that you statically allocate each implemented interrupt as either Secure or Non-secure. To change the security status of an interrupt you must ensure that all the status information for that interrupt is drained before you update the appropriate interrupt Security status bit.

For interrupt ID *N*, when DIV and MOD are the integer division and modulo operations:

- the corresponding ICDISR number, *M*, is given by *M* = *N* DIV 32
- the offset of the required ICDISR is (`0x080` + (4**M*))
- the bit number of the required Security status bit in this register is *N* MOD 32.

## ICDISR0 reset value

Normally, the reset value of all ICDISRs is zero, so that all interrupts are Secure unless reprogrammed as Non-secure by Secure accesses to the appropriate ICDISRs.

A multiprocessor implementation that supports the Security Extensions might include one or more *Non-secure processors*, meaning processors that cannot make Secure accesses to the GIC. In this situation only, a GIC can implement a Secure IMPLEMENTATION DEFINED mechanism that resets to 1 the ICDISR0 bits for the SGIs and PPIs of any Non-secure processor. This mechanism must apply only to:

- a banked ICDISR0 that corresponds to a Non-secure processor
- bits in that banked ICDISR0 that correspond to implemented interrupts.

### 4.3.5 Interrupt Set-Enable Registers (ICDISERn)

The ICDISER characteristics are:

**Purpose** The ICDISERs provide a Set-enable bit for each interupt supported by the GIC. Writing 1 to a Set-enable bit enables forwarding of the corresponding interrupt to the CPU interfaces. Reading a bit identifies whether the interrupt is enabled.

**Usage constraints** A register bit corresponding to an unimplemented interrupt is RAZ/WI.

If the GIC implements the Security Extensions:

- a register bit that corresponds to a Secure interrupt is RAZ/WI to Non-secure accesses
- if the GIC implements configuration lockdown, the system can lock down the Set-enable bits for the lockable SPIs that are configured as Secure, see *Configuration lockdown* on page 4-9.

Support of Set-enable bits for SGIs is IMPLEMENTATION DEFINED.

**Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.

The number of implemented ICDISERs is (ICDICTR.ITLinesNumber + 1), see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. The implemented ICDISERs number upwards from ICDISER0.

In a multiprocessor implementation, ICDISER0 is banked for each connected processor, see *Register banking* on page 4-5. This register holds the Set-enable bits for interrupts 0-31.

**Attributes** See the register summary in Table 4-1 on page 4-2.

Figure 4-5 shows the ICDISER bit assignments.

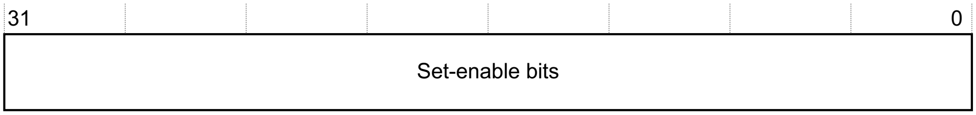


**Figure 4-5 ICDISER bit assignments**

Table 4-8 shows the ICDISER bit assignments.

**Table 4-8 ICDISER bit assignments**

| Bits | Name | Function | | |
|---|---|---|---|---|
| [31:0] | Set-enable bits | For SPIs and PPIs, for each bit: | | |
| | | **Reads** | **0** | The corresponding interrupt is disabled. |
| | | | **1** | The corresponding interrupt is enabled. |
| | | **Writes** | **0** | Has no effect. |
| | | | **1** | Enables the corresponding interrupt. A subsequent read of this bit returns the value 1. |
| | | For SGIs the behavior of the bit on reads and writes is IMPLEMENTATION DEFINED. | | |

For interrupt ID *N*, when DIV and MOD are the integer division and modulo operations:

- the corresponding ICDISER number, *M*, is given by *M* = *N* DIV 32
- the offset of the required ICDISER is (`0x100` + (4**M*))
- the bit number of the required Set-enable bit in this register is *N* MOD 32.

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

**Note**

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

### 4.3.6 Interrupt Clear-Enable Registers (ICDICERn)

The ICDICER characteristics are:

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

**Usage constraints** A register bit corresponding to an unimplemented interrupt is RAZ/WI.

If the GIC implements the Security Extensions:

- a register bit that corresponds to a Secure interrupt is RAZ/WI to Non-secure accesses
- if the GIC implements configuration lockdown, the system can lock down the Clear-enable bits for the lockable SPIs that are configured as Secure, see *Configuration lockdown* on page 4-9.

Support of Clear-enable bits for SGIs is IMPLEMENTATION DEFINED.

**Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.

The number of implemented ICDICERs is (ICDICTR.ITLinesNumber + 1), see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. The implemented ICDICERs number upwards from ICDICER0.

In a multiprocessor implementation, ICDICER0 is banked for each connected processor, see *Register banking* on page 4-5. This register holds the Clear-enable bits for interrupts 0-31.

**Attributes** See the register summary in Table 4-1 on page 4-2.

Figure 4-6 shows the ICDICER bit assignments.

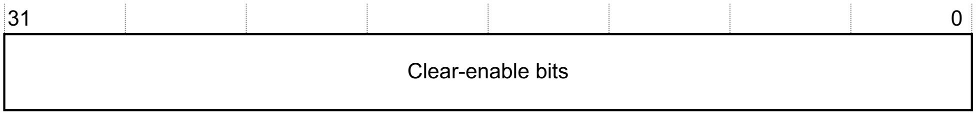


**Figure 4-6 ICDICER bit assignments**

Table 4-9 shows the ICDICER bit assignments.

**Table 4-9 ICDICER bit assignments**

| Bits | Name | Function |
|---|---|---|
| [31:0] | Clear-enable bits | For SPIs and PPIs, for each bit:<br>**Reads** **0** The corresponding interrupt is disabled.<br>**1** The corresponding interrupt is enabled.<br>**Writes** **0** Has no effect.<br>**1** Disables the corresponding interrupt. A subsequent read of this bit returns the value 0.<br>For SGIs the behavior of the bit on reads and writes is IMPLEMENTATION DEFINED. |

For interrupt ID *N*, when DIV and MOD are the integer division and modulo operations:

- the corresponding ICDICER number, *M*, is given by *M* = *N* DIV 32
- the offset of the required ICDICER is (`0x180` + (4**M*))
- the bit number of the required Clear-enable bit in this register is *N* MOD 32.

**Note**

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

### 4.3.7 Interrupt Set-Pending Registers (ICDISPRn)

The ICDISPR characteristics are:

**Purpose** The ICDISPRs provide a Set-pending bit for each interrupt supported by the GIC. Writing 1 to a Set-pending bit sets the status of the corresponding peripheral interrupt to pending. Reading a bit identifies whether the interrupt is pending.

**Usage constraints** A register bit corresponding to an unimplemented interrupt is RAZ/WI.

If the GIC implements the Security Extensions:

- a register bit that corresponds to a Secure interrupt is RAZ/WI to Non-secure accesses
- if the GIC implements configuration lockdown, the system can lock down the Set-pending bits for the lockable SPIs that are configured as Secure, see *Configuration lockdown* on page 4-9.

Set-pending bits for SGIs are read-only and ignore writes.

**Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.

The number of implemented ICDISPRs is (ICDICTR.ITLinesNumber + 1), see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. The implemented ICDISPRs number upwards from ICDISPR0.

In a multiprocessor implementation, ICDISPR0 is banked for each connected processor, see *Register banking* on page 4-5. This register holds the Set-pending bits for interrupts 0-31.

**Attributes** See the register summary in Table 4-1 on page 4-2.

Figure 4-7 shows the ICDISPR bit assignments.

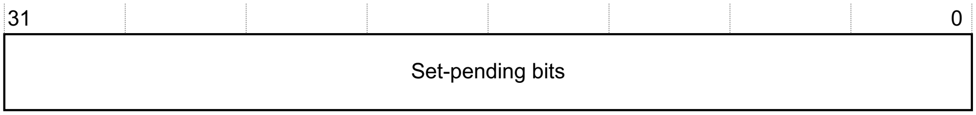


**Figure 4-7 ICDISPR bit assignments**

Table 4-10 shows the ICDISPR bit assignments.

**Table 4-10 ICDISPR bit assignments**

| Bits | Name | Function |
|---|---|---|
| [31:0] | Set-pending bits | For each bit:<br>**Reads** **0** The corresponding interrupt is not pending on any processor.<br>**1** • For SGIs and PPIs, the corresponding interrupt is pending[a] on this processor.<br>• For SPIs, the corresponding interrupt is pending[a] on at least one processor.<br>**Writes** For SPIs and PPIs:<br>**0** Has no effect.<br>**1** The effect depends on whether the interrupt is edge-triggered or level-sensitive:<br>**Edge-triggered**<br>Changes the status of the corresponding interrupt to:<br>• pending if it was previously inactive<br>• active and pending if it was previously active.<br>Has no effect if the interrupt is already pending[a].<br>**Level sensitive**<br>If the corresponding interrupt is not pending[a], changes the status of the corresponding interrupt to:<br>• pending if it was previously inactive<br>• active and pending if it was previously active.<br>If the interrupt is already pending[a]:<br>• because of a write to the ICDISPR, the write has no effect<br>• because the corresponding interrupt signal is asserted, the write has no effect on the status of the interrupt, but the interrupt remains pending[a] if the interrupt signal is deasserted.<br>For more information see *Control of the pending status of level-sensitive interrupts* on page 4-28.<br>For SGIs, the write is ignored. |

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

For interrupt ID *N*, when DIV and MOD are the integer division and modulo operations:

- the corresponding ICDISPR number, *M*, is given by *M* = *N* DIV 32
- the offset of the required ICDISPR is (`0x200` + (4**M*))
- the bit number of the required Set-pending bit in this register is *N* MOD 32.

### 4.3.8 Interrupt Clear-Pending Registers (ICDICPRn)

The ICDICPR characteristics are:

**Purpose** The ICDICPRs provide a Clear-pending bit for each interupt supported by the GIC. Writing 1 to a Clear-pending bit clears the pending status of the corresponding peripheral interrupt. Reading a bit identifies whether the interrupt is pending.

**Usage constraints** A register bit corresponding to an unimplemented interrupt is RAZ/WI.

If the GIC implements the Security Extensions:

- a register bit that corresponds to a Secure interrupt is RAZ/WI to Non-secure accesses
- if the GIC implements configuration lockdown, the system can lock down the Clear-pending bits for the lockable SPIs that are configured as Secure, see *Configuration lockdown* on page 4-9.

Clear-pending bits for SGIs are read-only and ignore writes.

**Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.

The number of implemented ICDICPRs is (ICDICTR.ITLinesNumber + 1), see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. The implemented ICDICPRs number upwards from ICDICPR0.

In a multiprocessor implementation, ICDICPR0 is banked for each connected processor, see *Register banking* on page 4-5. This register holds the Clear-pending bits for interrupts 0-31.

**Attributes** See the register summary in Table 4-1 on page 4-2.

Figure 4-8 shows the ICDICPR bit assignments.

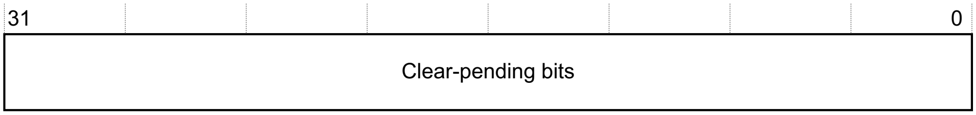


**Figure 4-8 ICDICPR bit assignments**

Table 4-11 shows the ICDICPR bit assignments.

**Table 4-11 ICDICPR bit assignments**

| Bits | Name | Function | | |
|---|---|---|---|---|
| [31:0] | Clear-pending bits | For each bit: | | |
| | | **Reads** | **0** | The corresponding interrupt is not pending on any processor. |
| | | | **1** | • For SGIs and PPIs, the corresponding interrupt is pending[a] on this processor.<br>• For SPIs, the corresponding interrupt is pending[a] on at least one processor. |
| | | **Writes** | For SPIs and PPIs: | |
| | | | **0** | Has no effect. |
| | | | **1** | The effect depends on whether the interrupt is edge-triggered or level-sensitive:<br>**Edge-triggered**<br>Changes the status of the corresponding interrupt to:<br>• inactive if it was previously pending<br>• active if it was previously active and pending.<br>Has no effect if the interrupt is not pending.<br>**Level-sensitive**<br>If the corresponding interrupt is pending[a] only because of a write to the ICDISPR, the write changes the status of the interrupt to:<br>• inactive if it was previously pending<br>• active if it was previously active and pending.<br>Otherwise the interrupt remains pending if the interrupt signal remains asserted.<br>For more information see *Control of the pending status of level-sensitive interrupts* on page 4-28 |
| | | | For SGIs, the write is ignored. | |

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

For interrupt ID *N*, when DIV and MOD are the integer division and modulo operations:

- the corresponding ICDICPR number, *M*, is given by *M* = *N* DIV 32
- the offset of the required ICDICPR is (`0x280` + (4**M*))
- the bit number of the required Set-pending bit in this register is *N* MOD 32.

### Control of the pending status of level-sensitive interrupts

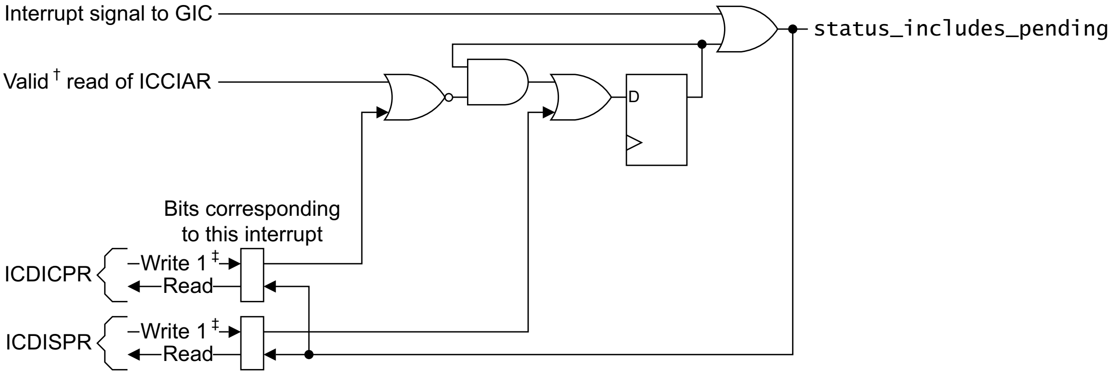
This subsection describes the status of an interrupt as *includes pending* if the interrupt status is one of:

- pending
- active and pending.

For an edge-triggered interrupt, the includes pending status is latched on either a write to the ICDISPR or the assertion of the interrupt signal to the GIC. However, for a level-sensitive interrupt, the includes pending status either:

- is latched on a write to the ICDISPR
- follows the state of the interrupt signal to the GIC, without any latching.

This means that the operation of the Set-pending and Clear-pending registers is more complicated for level-sensitive interrupts. Figure 4-9 shows the logic of the pending status of a level-sensitive interrupt. The logical output `status_includes_pending` is TRUE when the interrupt status includes pending, and FALSE otherwise.



† A read that acknowledges this interrupt

‡ The register ignores a write of 0

ICCIAR: Interrupt Acknowledge Register

ICDICPR: Interrupt Clear-Pending Register

ICDISPR: Interrupt Set-Pending Register

**Figure 4-9 Logic of the pending status of a level-sensitive interrupt**

### 4.3.9 Active Bit Registers (ICDABRn)

The ICDABR characteristics are:

**Purpose** The ICDABRs provide an Active bit for each interupt supported by the GIC. Reading an Active bit identifies whether the corresponding interrupt is active.

**Note**

The bit reads as one if the status of the interrupt is active or active and pending. Read the ICDSPR or ICDCPR to find the pending status of the interrupt.

**Usage constraints** A register bit corresponding to an unimplemented interrupt is RAZ.

If the GIC implements the Security Extensions a register bit that corresponds to a Secure interrupt is RAZ to Non-secure accesses.

**Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.

The number of implemented ICDABRs is (ICDICTR.ITLinesNumber + 1), see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. The implemented ICDABRs number upwards from ICDABR0.

In a multiprocessor implementation, ICDABR0 is banked for each connected processor, see *Register banking* on page 4-5. This register holds the Active bits for interrupts 0-31.

**Attributes** See the register summary in Table 4-1 on page 4-2.

Figure 4-10 shows the ICDABR bit assignments.

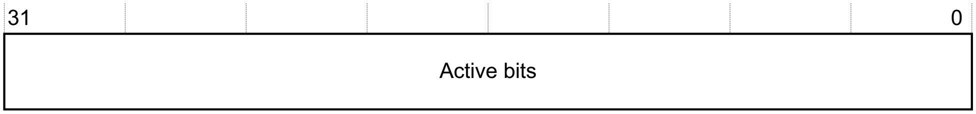


Figure 4-10 ICDABR bit assignments

Table 4-12 shows the ICDABR bit assignments.

**Table 4-12 ICDABR bit assignments**

| Bits | Name | Function | |
|---|---|---|---|
| [31:0] | Active bits | For each bit: | |
| | | **0** | Corresponding interrupt is not active[a]. |
| | | **1** | Corresponding interrupt is active[a]. |

[a] Active interrupts include interrupts that are active and pending.

For interrupt ID *N*, when DIV and MOD are the integer division and modulo operations:

- the corresponding ICDABR number, *M*, is given by *M* = *N* DIV 32
- the offset of the required ICDABR is (`0x300` + (4**M*))
- the bit number of the required Active bit in this register is *N* MOD 32.

### 4.3.10 Interrupt Priority Registers (ICDIPRn)

The ICDIPR characteristics are:

**Purpose** The ICDIPRs provide an 8-bit Priority field for each interupt supported by the GIC. This field stores the priority of the corresponding interrupt.

**Usage constraints** These registers are byte accessible.

A register field corresponding to an unimplemented interrupt is RAZ/WI.

A GIC might implement fewer than eight priority bits, but must implement at least bits [7:4] of each field. In each field, unimplemented bits are RAZ/WI.

If the GIC implements the Security Extensions:

- a register field that corresponds to a Secure interrupt is RAZ/WI to Non-secure accesses
- a Non-secure access to a field that corresponds to a Non-secure interrupt behaves as described in *Software views of interrupt priority* on page 3-18
- if the GIC implements configuration lockdown, the system can lock down the Priority fields for the lockable SPIs that are configured as Secure, see *Configuration lockdown* on page 4-9

It is IMPLEMENTATION DEFINED whether changing the value of a priority field changes the priority of an active interrupt.

**Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.

The number of implemented ICDIPRs is (8*(ICDICTR.ITLinesNumber+1)), see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. The implemented ICDIPRs number upwards from ICDIPR0.

In a multiprocessor implementation, ICDIPR0 to ICDIPR7 are banked for each connected processor, see *Register banking* on page 4-5. These registers hold the Priority fields for interrupts 0-31.

**Attributes** See the register summary in Table 4-1 on page 4-2.

Figure 4-11 shows the ICDIPR bit assignments.

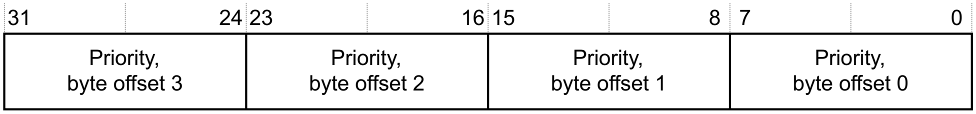


**Figure 4-11 ICDIPR bit assignments**

Table 4-13 shows the ICDIPR bit assignments.

**Table 4-13 ICDIPR bit assignments**

| Bits | Name[a] | Function |
|---|---|---|
| [31:24] | Priority, byte offset 3 | Each priority field holds a priority value, from an IMPLEMENTATION DEFINED range. The lower the value, the greater the priority of the corresponding interrupt. For more information see *Interrupt prioritization* on page 3-12 and, if appropriate, *The effect of the Security Extensions on interrupt prioritization* on page 3-18. |
| [23:16] | Priority, byte offset 2 | |
| [15:8] | Priority, byte offset 1 | |
| [7:0] | Priority, byte offset 0 | |

[a] Each field holds the priority value for a single interrupt. This section describes how the interrupt ID value determines the ICDIPR register number and the byte offset of the priority field in that register.

For interrupt ID *N*, when DIV and MOD are the integer division and modulo operations:

- the corresponding ICDIPR number, *M*, is given by *M* = *N* DIV 4
- the offset of the required ICDIPR is (`0x400` + (4**M*))
- the byte offset of the required Priority field in this register is *N* MOD 4, where:
  - — byte offset 0 refers to register bits [7:0]
  - — byte offset 1 refers to register bits [15:8]
  - — byte offset 2 refers to register bits [23:16]
  - — byte offset 3 refers to register bits [31:24].

### 4.3.11 Interrupt Processor Targets Registers (ICDIPTRn)

The ICDIPTR characteristics are:

**Purpose** The ICDIPTRs provide an 8-bit CPU targets field for each interrupt supported by the GIC. This field stores the list of processors that the interrupt is sent to if it is asserted.

**Usage constraints** For a multiprocessor implementation:

- These registers are byte accessible.
- A register field corresponding to an unimplemented interrupt is RAZ/WI.
- ICDIPTR0 to ICDIPTR7 are read-only, and each field returns a value corresponding only to the processor reading the register.
- It is IMPLEMENTATION DEFINED which, if any, SPIs are statically configured in hardware. The CPU targets field for such an SPI is read-only, and returns a value that indicates the CPU targets for the interrupt.
- if the GIC implements the Security Extensions:
  - — a register field that corresponds to a Secure interrupt is RAZ/WI to Non-secure accesses
  - — if the GIC implements configuration lockdown, the system can lock down the CPU targets fields for the lockable SPIs that are configured as Secure, see *Configuration lockdown* on page 4-9.

See also *The effect of changes to an ICDIPTR* on page 4-35.

**Note**

In a uniprocessor implementation, all interrupts target the one processor, and the ICDIPTRs are RAZ/WI.

**Configurations** These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.

The number of implemented ICDIPTRs is (8*(ICDICTR.ITLinesNumber+1)), see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. The implemented ICDIPTRs number upwards from ICDIPTR0.

In a multiprocessor implementation, ICDIPTR0 to ICDIPTR7 are banked for each connected processor, see *Register banking* on page 4-5. These registers hold the CPU targets fields for interrupts 0-31.

**Attributes** See the register summary in Table 4-1 on page 4-2.

Figure 4-12 shows the ICDIPTR bit assignments, for a multiprocessor implementation.

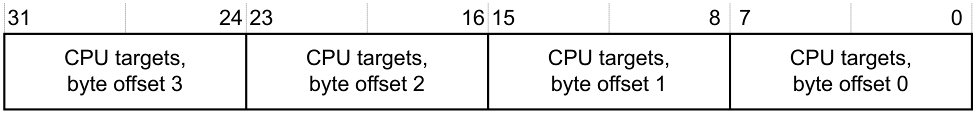

31
24
23
16
15
8
7
0
CPU targets, byte offset 3
CPU targets, byte offset 2
CPU targets, byte offset 1
CPU targets, byte offset 0

**Figure 4-12 ICDIPTR bit assignments**

Table 4-14 shows the ICDIPTR bit assignments, for a multiprocessor implementation.

**Table 4-14 ICDIPTR bit assignments**

| Bits | Name[a] | Function |
|---|---|---|
| [31:24] | CPU targets, byte offset 3 | Processors in the system number from 0, and each bit in a CPU targets field refers to the corresponding processor, see Table 4-15. For example, a value of `0x3` means that the Pending interrupt is sent to processors 0 and 1.<br>For ICDIPTR0 to ICDIPTR7, a read of any CPU targets field returns the number of the processor performing the read. |
| [23:16] | CPU targets, byte offset 2 | |
| [15:8] | CPU targets, byte offset 1 | |
| [7:0] | CPU targets, byte offset 0 | |

[a] Each field holds the CPU targets list for a single interrupt. This section describes how the interrupt ID value determines the ICDIPTR register number and the byte offset of the CPU targets field in that register.

Table 4-15 shows how each bit of a CPU targets field targets the interrupt at one of the CPU interfaces.

**Table 4-15 Meaning of CPU targets field bit values**

| CPU targets field value | Interrupt targets |
|---|---|
| 0bxxxxxxx1 | CPU interface 0 |
| 0bxxxxxx1x | CPU interface 1 |
| 0bxxxxx1xx | CPU interface 2 |
| 0bxxxx1xxx | CPU interface 3 |
| 0bxxx1xxxx | CPU interface 4 |
| 0bxx1xxxxx | CPU interface 5 |
| 0bx1xxxxxx | CPU interface 6 |
| 0b1xxxxxxx | CPU interface 7 |

A CPU targets field bit that corresponds to an unimplemented CPU interface is RAZ/WI.

For interrupt ID *N*, when DIV and MOD are the integer division and modulo operations:

- the corresponding ICDIPTR number, *M*, is given by *M* = *N* DIV 4
- the offset of the required ICDIPTR is (`0x800` + (4**M*))
- the byte offset of the required Priority field in this register is *N* MOD 4, where:
  - byte offset 0 refers to register bits [7:0]
  - byte offset 1 refers to register bits [15:8]
  - byte offset 2 refers to register bits [23:16]
  - byte offset 3 refers to register bits [31:24].

### The effect of changes to an ICDIPTR

Software can write to an ICDIPTR at any time. Any change to a CPU targets field value:

- Has no effect on any active interrupt. This means that removing a CPU interface from a targets list does not cancel an active state for that interrupt on that CPU interface.
- Has an immediate effect on any pending interrupts. This means:
  - adding a CPU interface to the target list of a pending interrupt makes that interrupt pending on that CPU interface
  - removing a CPU interface from the target list of a pending interrupt removes the pending state of that interrupt on that CPU interface.
- If applied to an interrupt that is active and pending, will not change the interrupt targets until the active status is cleared.

### 4.3.12 Interrupt Configuration Registers (ICDICFRn)

The ICDICFR characteristics are:

**Purpose**

The ICDICFRs provide a 2-bit Int_config field for each interupt supported by the GIC. This field identifies whether the corresponding interrupt is:

- edge-triggered or level-sensitive, see *Interrupt types* on page 1-4
- handled using the 1-N model or using the N-N model, see *Models for handling interrupts* on page 1-5.

**Usage constraints**

For each supported PPI, it is IMPLEMENTATION DEFINED whether software can program the corresponding Int_config field.

For SGIs, Int_config fields are read-only, meaning that ICDICFR0 is read-only. For PPIs, it is IMPLEMENTATION DEFINED whether the most significant bit of the Int_config field is programmable. See Table 4-16 on page 4-37 for more information.

A register field corresponding to an unimplemented interrupt is RAZ/WI.

If the GIC implements the Security Extensions:

- a register field that corresponds to a Secure interrupt is RAZ/WI to Non-secure accesses
- if the GIC implements configuration lockdown, the system can lock down the Int_config fields for the lockable SPIs that are configured as Secure, see *Configuration lockdown* on page 4-9.

Before changing the value of a programmable Int_config field, software must disable the corresponding interrupt, otherwise GIC behavior is UNPREDICTABLE.

**Configurations**

These registers are available in all configurations of the GIC. If the GIC implements the Security Extensions these registers are Common.

In a multiprocessor implementation, if the most significant bit of the Int_config field for any PPI is programmable then ICDICFR1 is banked for each connected processor, see *Register banking* on page 4-5. This registers holds the Int_config fields for the PPIs, interrupts 16-31.

The number of implemented ICDICFRs is (2*(ICDICTR.ITLinesNumber+1)), see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. The implemented ICDICFRs number upwards from ICDICFR0.

**Attributes**

See the register summary in Table 4-1 on page 4-2.

Figure 4-13 on page 4-37 shows the ICDICFR bit assignments.

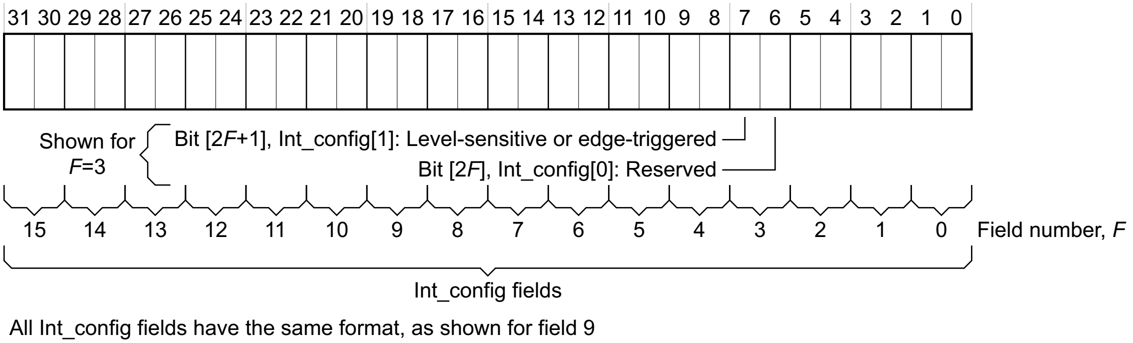


**Figure 4-13 ICDICFR bit assignments**

Table 4-16 shows the ICDICFR bit assignments, for version 1 of the GIC Architecture Specification.

**Table 4-16 ICDICFR bit assignments**

| Bits | Name | Function |
|---|---|---|
| [2*F*+1:2*F*] | Int_config, field *F* | For Int_config[1], the most significant bit, bit [2*F*+1], the encoding is:<br>**0** Corresponding interrupt is level-sensitive.<br>**1** Corresponding interupt is edge-triggered.<br>Int_config[0], the least significant bit, bit [2*F*], is Reserved, but see Table 4-17 on page 4-38 for the encoding of this bit on some early implementations of this GIC architecture.<br>For SGIs:<br>**Int_config[1]** Not programmable, RAO/WI[a].<br>For PPIs and SPIs:<br>**Int_config[1]** For SPIs, this bit is programmable. For PPIs it is IMPLEMENTATION DEFINED whether this bit is programmable. A read of this bit always returns the correct value to indicate whether the corresponding interrupt is level-sensitive or edge-triggered. |

[a] If the GIC implements the Security Extensions and the bit corresponds to a Secure interrupt, it is RAZ/WI to Non-secure accesses. This is the usual behavior of bits that correspond to Secure interrupts.

In some implementations of this GIC architecture before the publication of version 1 of the GIC Architecture Specification, the model for handling each peripheral interrupt can be configured using bit [0] of the corresponding Int_config field. Table 4-17 on page 4-38 shows the encoding of Int_config[0] on these implementations.

**Table 4-17 ICDICFR Int_config[0] encoding in some early GIC implementations**

| Bits | Name | Function |
|---|---|---|
| [2*F*] | Int_config[0], field *F* | On a GIC where the handling mode of peripheral interrupts is configurable, the encoding of Int_config[0] for PPIs and SPIs, is:<br>**0** Corresponding interrupt is handled using the N-N model.<br>**1** Corresponding interrupt is handled using the 1-N model. |

For interrupt ID *N*, when DIV and MOD are the integer division and modulo operations:

- the corresponding ICDICFR number, *M*, is given by *M* = *N* DIV 16
- the offset of the required ICDIPTR is (`0xC00` + (4**M*))
- the required Priority field in this register, *F*, is given by *F* = *N* MOD 16, where field 0 refers to register bits [1:0], field 1 refers to bits [3:2], and so on, up to field 15 refers to bits [31:30], see Figure 4-13 on page 4-37.

### 4.3.13 Software Generated Interrupt Register (ICDSGIR)

The ICDSGIR characteristics are:

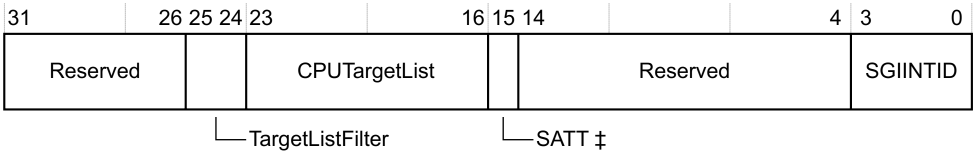
**Purpose** Controls the generation of SGIs.

**Usage constraints** The SATT field, bit [15], is implemented only if the GIC implements the Security Extensions.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.

**Attributes** See the register summary in Table 4-1 on page 4-2.

Figure 4-14 shows the ICDSGIR bit assignments.



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

**Figure 4-14 ICDSGIR bit assignments**

Table 4-18 shows the ICDSGIR bit assignments.

**Table 4-18 ICDSGIR bit assignments**

| Bits | Name | Function | |
|---|---|---|---|
| [31:26] | - | Reserved. | |
| [25:24] | TargetListFilter | **0b00** | Send the interrupt to the CPU interfaces specified in the CPUTargetList field[a]. |
| | | **0b01** | Send the interrupt to all CPU interfaces except the CPU interface that requested the interrupt. |
| | | **0b10** | Send the interrupt only to the CPU interface that requested the interrupt. |
| | | **0b11** | Reserved. |
| [23:16] | CPUTargetList | When TargetList Filter = 0b00, defines the CPU interfaces the Distributor must send the interrupt to.<br>Each bit of CPUTargetList[7:0] refers to the corresponding CPU interface, for example CPUTargetList[0] corresponds to CPU interface 0. Setting a bit to 1 sends the interrupt to the corresponding interface. | |
| [15] | - | If GIC does not implement the Security Extensions, this field is Reserved. | |

**Table 4-18 ICDSGIR bit assignments (continued)**

| Bits | Name | Function |
|---|---|---|
| [15] | SATT | If the GIC implements the Security Extensions, this field is writable only using a Secure access. Any Non-secure write to the ICDSGIR issues an SGI only if the specified SGI is programmed as Non-secure, regardless of the value of bit [15] of the write.<br>Specifies the required security value of the SGI:<br>**0** Send the SGI specified in the SGIINTID field to a specified CPU interface only if the SGI is configured as Secure on that interface.<br>**1** Send the SGI specified in the SGIINTID field to a specified CPU interfaces only if the SGI is configured as Non-secure on that interface.<br>See *SGI generation when the GIC implements the Security Extensions* for more information. |
| [14:4] | - | Reserved, SBZ. |
| [3:0] | SGIINTID | The Interrupt ID of the SGI to send to the specified CPU interfaces. The value of this field is the Interrupt ID, in the range 0-15, for example a value of 0b0011 specifies Interrupt ID 3. |

a. When TargetListFilter is 0b00, if the CPUTargetList field is `0x00` the Distributor does not send the interrupt to any CPU interface.

### SGI generation when the GIC implements the Security Extensions

If the GIC implements the Security Extensions, whether an SGI is sent to a processor specified in the write to the ICDSGIR depends on:

- the security status of the write to the ICDSGIR
- for a Secure write to the ICDSGIR, the value of the ICDSGIR.SATT bit
- whether the specified SGI is configured as Secure or Non-secure on the targeted processor.

ICDISR0 holds the security states of the SGIs, see *Interrupt Security Registers (ICDISRn)* on page 4-17. In a multiprocessor system, ICDISR0 is banked for each connected processor, so the system configures the security of each SGI independently for each processor. A single write to the ICDSGIR can target more than one processor. For each targeted processor, the Distributor determines whether to send the SGI to the processor.

Table 4-19 on page 4-41 shows the truth table for whether the Distributor sends a particular SGI to a specific target CPU interface.

**Table 4-19 Truth table for sending an SGI to a target processor**

| Status of ICDSGIR write | SATT value | Status of SGI on target processor | Send SGI?[a] |
|---|---|---|---|
| Secure | 0 | Secure | Yes |
| | | Non-secure | No |
| | 1 | Secure | No |
| | | Non-secure | Yes |
| Non-secure | x | Secure | No |
| | | Non-secure | Yes |

[a] Whether the SGI is sent to the CPU interface for the target processor.

### 4.3.14 Identification registers

This architecture specification defines offsets `0xFD0-0xFFC` in the Distributor register map as a read-only identification register space. This space includes three architecturally defined registers. as Table 4-20 shows:

Table 4-20 The GIC identification register space

| Offset | Name | Type | Reset[a] | Description |
|---|---|---|---|---|
| `0xFD0-0xFE4` | - | RO | - | IMPLEMENTATION DEFINED registers |
| `0xFE8` | ICPIDR2 | RO | -[b] | *Peripheral ID2 Register (ICPIDR2)* |
| `0xFEC-0xFFC` | - | RO | - | IMPLEMENTATION DEFINED registers |

[a] The reset value of an IMPLEMENTATION DEFINED register is IMPLEMENTATION DEFINED.

[b] See the register description for information about the architecturally defined bits in this register.

The assignment of this register space, and naming of registers in this space, is consistent with the ARM identification scheme for PrimeCells and CoreSight components. ARM implementations of this GIC architecture implement that identification scheme, and ARM strongly recommends that other implementers also use this scheme, to provide a consistent software discovery model, see *The ARM implementation of the GIC Identification Registers* on page 4-43 and *Use of identification registers* on page B-2.

#### Peripheral ID2 Register (ICPIDR2)

The ICPIDR2 characteristics are:

**Purpose** Provides a four-bit architecturally-defined architecture revision field. The remaining bits of the register are IMPLEMENTATION DEFINED.

**Usage constraints** There are no usage constraints.

**Configurations** This register is available in all configurations of the GIC.

**Attributes** See the register summary in Table 4-20.

Figure 4-15 shows the ICPIDR2 bit assignments.

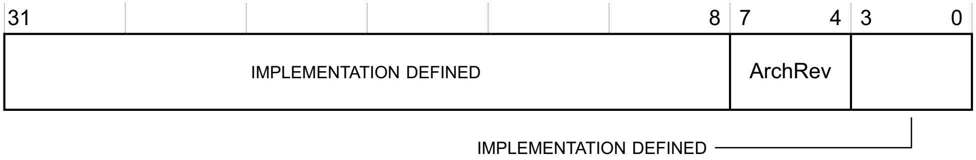


Figure 4-15 ICPIDR2 bit assignments

Table 4-21 shows the ICPIDR2 bit assignments.

**Table 4-21 ICPIDR2 bit assignments**

| Bits | Name | Function |
|---|---|---|
| [31:8] | - | IMPLEMENTATION DEFINED. |
| [7:4] | ArchRev | Revision field for the GIC architecture. The value of this field is `0x1`. This value might increase in future versions of the architecture. |
| [3:0] | - | IMPLEMENTATION DEFINED. |

### The ARM implementation of the GIC Identification Registers

**Note**

- The ARM implementation of these registers is consistent with the identification scheme for PrimeCells and CoreSight components. This implementation identifies the device as a GIC that implements this architecture. It does not identify the designer or manufacturer of the GIC implementation. For information about the designer and manufacturer of a GIC implementation see:
  - *Distributor Implementer Identification Register (ICDIIDR)* on page 4-16
  - *CPU Interface Identification Register (ICCIIDR)* on page 4-65
- In other contexts, this identification scheme identifies a component in a system. The GIC use of the scheme is different. It identifies only that the device is an implementation of a version of this GIC architecture. Software must read the ICDIIDR and ICCIIDR to discover, for example, the implementer and version of the GIC hardware. For more information see *Use of identification registers* on page B-2.

Table 4-22 shows the Identification Registers for an ARM implementation of this version of the GIC architecture. ARM recommends other implementers to include the registers described here.

**Table 4-22 Identification Registers for a GIC, with ARM implementation values**

| Register[a] | Offset | Bits | ARM implementation | |
|---|---|---|---|---|
| | | | Value | Description |
| Component ID0 (ICCIDR0) | `0xFF0` | [7:0] | `0x0D` | ARM-defined fixed values for the preamble for component discovery. |
| Component ID1 (ICCIDR1) | `0xFF4` | [7:0] | `0xF0` | |
| Component ID2 (ICCIDR2) | `0xFF8` | [7:0] | `0x05` | |
| Component ID3 (ICCIDR3) | `0xFFC` | [7:0] | `0xB1` | |

**Table 4-22 Identification Registers for a GIC, with ARM implementation values (continued)**

| Register[a] | Offset | Bits | ARM implementation | |
|---|---|---|---|---|
| | | | Value | Description |
| Peripheral ID0 (ICPIDR0) | `0xFE0` | [7:0] | `0x90` | Bits [7:0] of the ARM-defined DevID field. |
| Peripheral ID1 (ICPIDR1) | `0xFE4` | [7:4] | `0xB` | Bits [3:0] of the ARM-defined ArchID field. |
| | | [3:0] | `0x3` | Bits [11:8] of the ARM-defined DevID field. |
| Peripheral ID2 (ICPIDR2) | `0xFE8` | [7:4] | `0x1` | Architecturally-defined ArchRev field. |
| | | [3] | 1 | ARM-defined UsesJEPcode field. |
| | | [2:0] | 0b011 | Bits [6:4] of the ARM-defined ArchID field. |
| Peripheral ID3 (ICPIDR3) | `0xFEC` | [3:0] | `0x0` | Reserved by ARM. |
| | | [7:4] | `0x0` | ARM-defined Revision field. |
| Peripheral ID4 (ICPIDR4) | `0xFD0` | [3:0] | `0x4` | ARM-defined ContinuationCode field. |
| | | [7:4] | `0x0` | Reserved by ARM. |
| Peripheral ID5 (ICPIDR5) | `0xFD4` | [7:0] | `0x00` | Reserved by ARM. |
| Peripheral ID6 (ICPIDR6) | `0xFD8` | [7:0] | `0x00` | Reserved by ARM. |
| Peripheral ID7 (ICPIDR7) | `0xFDC` | [7:0] | `0x00` | Reserved by ARM. |

[a] In the ARM implementation, bits [31:8] of each register are reserved. Bits [7:0] of the four Component ID registers together define a conceptual 32-bit Component ID, and bits [7:0] of the eight Peripheral ID registers together define a conceptual 64-bit Peripheral ID.
In the GIC implementation, despite their names, Component ID and Peripheral ID refer only to the architecture of the implementation, see the Note at the start of this section for more information.

**Note**

Some previous ARM implementations of the GIC did not implement Peripheral ID registers 4-7. Software can use the value of bit [3] of the ICPIDR2 to identify these implementations:

| | |
|---|---|
| **0** | Legacy format. |
| **1** | ARM GIC architecture v1.0 format. |

### The ARM peripheral ID for a GIC

Together, the Peripheral ID registers ICPIDR0 to ICPIDR7 define an 64-bit peripheral ID. In current ARM implementations, bits [63:36] of that ID are reserved, RAZ. Figure 4-16 shows bits [35:0] of the Peripheral ID for a GIC, and Table 4-23 shows all the fields in the 64-bit Peripheral ID.

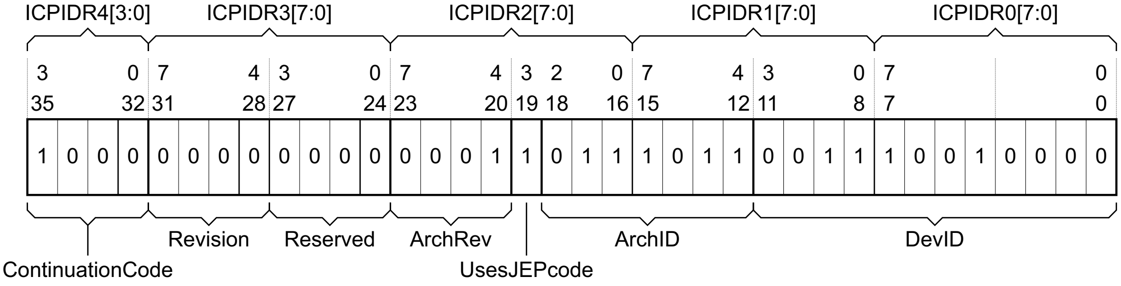


**Figure 4-16 ARM Peripheral ID fields for a GIC**

**Table 4-23 Fields in the GIC Peripheral ID, for an ARM implementation**

| Name | Bits | Source | Value | Function |
|---|---|---|---|---|
| - | [63:40] | ICPIDR7[7:0],<br>ICPIDR6[7:0],<br>ICPIDR5[7:0] | RAZ | |
| - | [39:36] | ICPIDR4[7:4] | `0x0` | Reserved[a] |
| ContinuationCode | [35:32] | ICPIDR4[3:0] | `0x4` | JEP106 continuation code for ARM[a] |
| Revision | [31:28] | ICPIDR3[7:4] | `0x0` | Revision field[a] |
| - | [27:24] | ICPIDR3[3:0] | `0x0` | Reserved[a] |
| ArchRev | [23:20] | ICPIDR2[7:4] | `0x1` | Architecturally-defined revision number for the ARM GIC architecture, see *Peripheral ID2 Register (ICPIDR2)* on page 4-42 |
| UsesJEPcode | [19] | ICPIDR2[3] | 1 | Indicate that the identifier string uses JEP106 codes to identify ARM as the designer of the architecture[a] |
| ArchID | [18:12] | ICPIDR2[2:0],<br>ICPIDR1[7:4] | `0x3B` | Identifies ARM as the designer of the GIC architecture[a] |
| DevID | [11:0] | ICPIDR1[3:0],<br>ICPIDR0[7:0] | `0x390` | Identifies the device as a GIC[a] |

[a] ARM-defined field.

## 4.4 CPU interface register descriptions

The following sections describe the CPU interface registers:

- *CPU Interface Control Register (ICCICR)* on page 4-47
- *Interrupt Priority Mask Register (ICCPMR)* on page 4-52
- *Binary Point Register (ICCBPR)* on page 4-54
- *Interrupt Acknowledge Register (ICCIAR)* on page 4-56
- *End of Interrupt Register (ICCEOIR)* on page 4-59
- *Running Priority Register (ICCRPR)* on page 4-61
- *Aliased Binary Point Register (ICCABPR)* on page 4-62
- *Highest Pending Interrupt Register (ICCHPIR)* on page 4-63
- *CPU Interface Identification Register (ICCIIDR)* on page 4-65.

### 4.4.1 CPU Interface Control Register (ICCICR)

The ICCICR characteristics are:

**Purpose** Enables the signalling of interrupts to the target processors. In a GIC that implements the Security Extensions, provides additional global controls for handling Secure interrupts.

**Usage constraints** If the GIC implements the Security Extensions with configuration lockdown, the system can lock down the Secure ICCICR, see *Configuration lockdown* on page 4-9.

**Configurations** If the GIC implements the Security Extensions, this register is banked to provide Secure and Non-secure copies, see *Register banking* on page 4-5, and the bit assignments are different in the Secure and Non-secure copies of the register.

**Attributes** See the register summary in Table 4-2 on page 4-4.

Figure 4-17 shows the Non-secure ICCICR bit assignments for:

- a GIC that does not implement the Security Extensions
- the Non-secure copy of the ICCICR in a GIC that implement the Security Extensions.

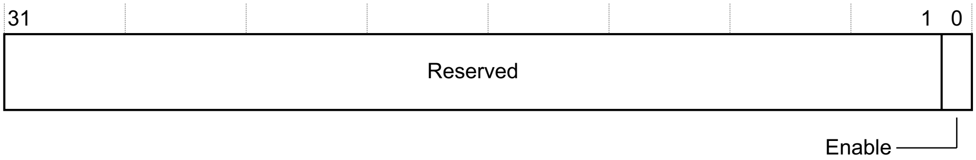


**Figure 4-17 ICCICR bit assignments, GIC without Security Extensions and Non-secure ICCICR**

Table 4-24 shows the ICCICR bit assignments for a GIC that does not implement the Security Extensions, and for the Non-secure ICCICR.

**Table 4-24 ICCICR bit assignments, GIC without Security Extensions and Non-secure ICCICR**

| Bits | Name | Function |
|---|---|---|
| [31:1] | - | Reserved. |
| [0] | Enable | Global enable for the signalling of interrupts by the CPU interfaces to the connected processors.<br>**0** Disable signalling of interrupts.<br>**1** Enable signalling of interrupts.<br>On a GIC that implements the Security Extensions, this bit controls only the signalling of Non-secure interrupts. |

For a GIC that implements the Security Extensions, Figure 4-18 on page 4-48 shows bit assignments for the the Secure copy of the ICCICR.

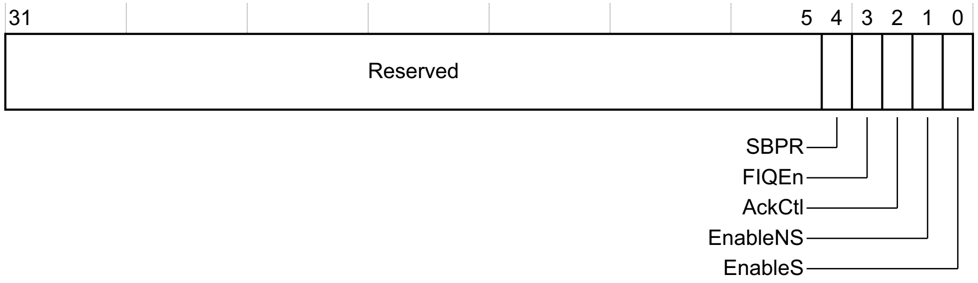


**Figure 4-18 Secure ICCICR bit assignments**

Table 4-25 shows the Secure ICCICR bit assignments.

**Table 4-25 Secure ICCICR bit assignments**

| Bit | Name | Function |
|---|---|---|
| [31:5] | - | Reserved. |
| [4] | SBPR | Controls whether the CPU interface uses the Secure or Non-secure Binary Point Register for preemption.<br>**0** To determine any preemption, use:<br>• the Secure Binary Point Register for Secure interrupts<br>• the Non-secure Binary Point Register for Non-secure interrupts.<br>**1** To determine any preemption use the Secure Binary Point Register for both Secure and Non-secure interrupts. |
| [3] | FIQEn | Controls whether the GIC signals Secure interrupts to a target processor using the FIQ or the IRQ signal.<br>**0** Signal Secure interrupts using the IRQ signal.<br>**1** Signal Secure interrupts using the FIQ signal.<br>The GIC always signals Non-secure interrupts using the IRQ signal. |
| [2] | AckCtl | Controls whether a Secure read of the ICCIAR, when the highest priority pending interrupt is Non-secure, causes the CPU interface to acknowledge the interrupt.<br>**0** If the highest priority pending interrupt is Non-secure, a Secure read of the ICCIAR returns an Interrupt ID of 1022. The read does not acknowledge the interrupt, and the pending status of the interrupt is unchanged.<br>**1** If the highest priority pending interrupt is Non-secure, a Secure read of the ICCIAR returns the Interrupt ID of the Non-secure interrupt. The read acknowledges the interrupt, and the status of the interrupt becomes active, or active and pending.<br>For more information see *The effect of the Security Extensions on interrupt prioritization* on page 3-18. |

**Table 4-25 Secure ICCICR bit assignments (continued)**

| Bit | Name | Function |
|---|---|---|
| [1] | EnableNS | An alias of the Enable bit in the Non-secure ICCICR. This alias bit means Secure software can enable the signalling of Non-secure interrupts.<br>**0** Disable signalling of Non-secure interrupts.<br>**1** Enable signalling of Non-secure interrupts. |
| [0] | EnableS | Global enable for the signalling of Secure interrupts by the CPU interfaces to the connected processors.<br>**0** Disable signalling of Secure interrupts.<br>**1** Enable signalling of Secure interrupts. |

## Optional support for interrupt signal pass-through

Optionally, a GIC can multiplex the interrupt request from a CPU interface with a legacy interrupt signal so that:

- When the system enables interrupt signalling by the GIC, the CPU interface drives the GIC interrupt request output.
- When the system disables interrupt signalling by the GIC, the legacy interrupt signal drives the GIC interrupt request output. Operating in this way is called interrupt signal pass-through.

Figure 4-19 shows a simple implementation of interrupt signal pass-through.

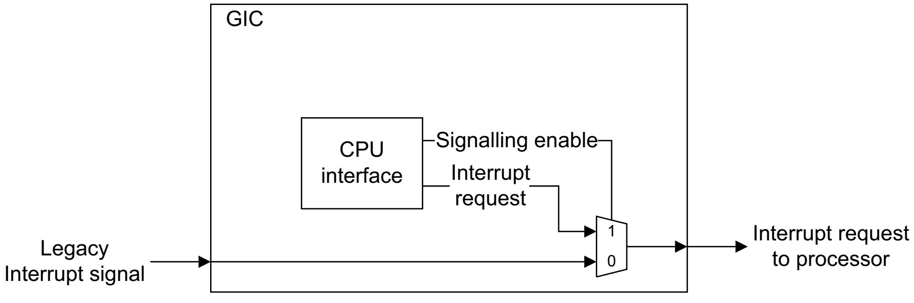


**Figure 4-19 Interrupt signal pass-through**

GIC support for interrupt signal pass-through is IMPLEMENTATION DEFINED.

**Note**

In a multiprocessor system support might be different on different CPU interfaces, and some or all interfaces might not support any pass-through.

### interrupt signal pass-through with Security Extensions support

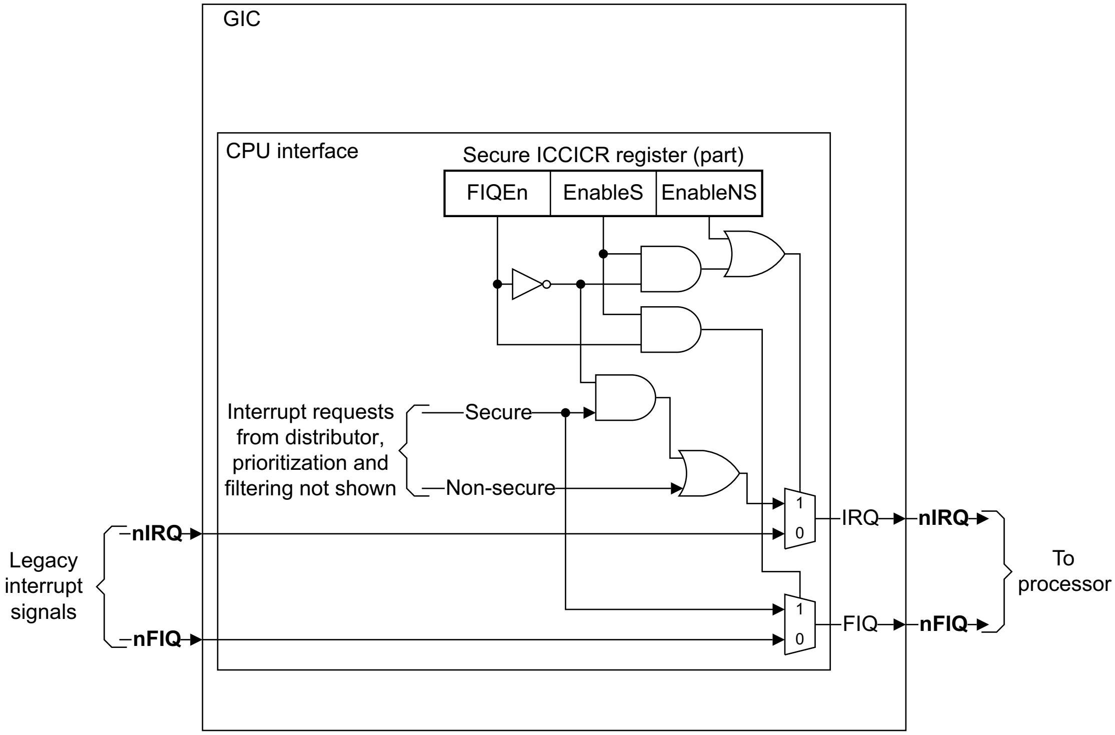
When a GIC implements the Security Extensions, it has two interrupt exception request outputs, described as IRQ and FIQ. It always uses the IRQ output to signal Non-secure interrupts, but can use the FIQ output to signal Secure interrupts. In many implementations these GIC signals correspond to the **nIRQ** and **nFIQ** input ports on an ARM processor, and on a processor that implements the ARMv7-A or ARMv7-R architecture profile these GIC outputs normally correspond to requests for the IRQ and FIQ interrupt exceptions. In such an implementation, the appropriate GIC CPU interface might include pass-through of both interrupt signals. Table 4-26 shows how the Secure ICCICR controls the GIC interrupt outputs.

**Table 4-26 Interrupt pass through behavior**

| Secure ICCICR register bits | | | GIC interrupt outputs | |
|---|---|---|---|---|
| **FIQEn** | **EnableS** | **EnableNS**[a] | **IRQ request behavior** | **FIQ request behavior** |
| 0 | 0 | 0 | Pass through | Pass through |
| | | 1 | Driven by GIC | Pass through |
| | 1 | 0 | Driven by GIC | Pass through |
| | | 1 | Driven by GIC | Pass through |
| 1 | 0 | 0 | Pass through | Pass through |
| | | 1 | Driven by GIC | Pass through |
| | 1 | 0 | Pass through | Driven by GIC |
| | | 1 | Driven by GIC | Driven by GIC |

[a] The EnableNS bit in the Secure ICCICR register. This is an alias of the Enable bit in the Non-secure ICCICR register.

For such an implementation, Figure 4-20 on page 4-51 shows how the Secure and Non-secure interrupts might be signaled, for a CPU interface that implements active-LOW interrupt signaling.



GIC implemented with active-LOW interrupt signals

**Figure 4-20 Secure and Non-secure interrupt signaling, with pass-through**

### 4.4.2 Interrupt Priority Mask Register (ICCPMR)

The ICCPMR characteristics are:

**Purpose** Provides an interrupt priority filter. Only interrupts with higher priority than the value in this register can be signalled to the processor.

**Note**

Higher priority corresponds to a lower Priority field value.

**Usage constraints** If the GIC implements the Security Extensions then:

- a Non-secure access to this register can only read or write a value that corresponds to the lower half of the priority range, see *The effect of the Security Extensions on interrupt prioritization* on page 3-18.
- if a Secure write has programmed the ICCPMR with a value that corresponds to a value in the upper half of the priority range then:
  - any Non-secure read of the ICCPMR returns 0x00, regardless of the value held in the register
  - any Non-secure write to the ICCPMR is ignored.

For more information see *Non-secure access to register fields for Secure interrupt priorities* on page 4-8.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions, this register is Common.

**Attributes** See the register summary in Table 4-2 on page 4-4.

Figure 4-21 shows the ICCPMR bit assignments.

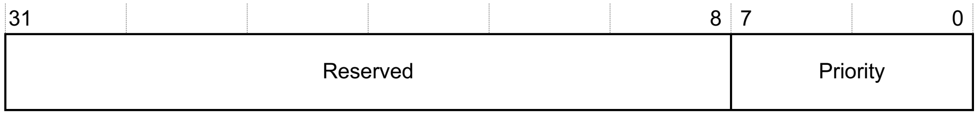


**Figure 4-21 ICCPMR bit assignments**

Table 4-27 shows the ICCPMR Register bit assignments.

**Table 4-27 ICCPMR Register bit assignments**

| Bits | Name | Function |
|---|---|---|
| [31:8] | - | Reserved. |
| [7:0] | Priority | The priority mask level for the CPU interface. If the priority of an interrupt is higher than the value indicated by this field, the interface signals the interrupt to the processor.<br>If the GIC supports fewer than 256 priority levels then some bits are RAZ/WI, as follows:<br>**128 supported levels** Bit [0] = 0.<br>**64 supported levels** Bit [1:0] = 0b00.<br>**32 supported levels** Bit [2:0] = 0b000.<br>**16 supported levels** Bit [3:0] = 0b0000.<br>For more information see *Interrupt prioritization* on page 3-12. |

### 4.4.3 Binary Point Register (ICCBPR)

The ICCBPR characteristics are:

**Purpose** The register defines the point at which the priority value fields split into two parts, the *group priority* field and the *subpriority* field. The group priority field is used to determine interrupt preemption. For more information see *Preemption* on page 3-13 and *Priority grouping* on page 3-14.

**Usage constraints** The minimum binary point value is IMPLEMENTATION DEFINED in the range 0-3. An attempt to program the binary point field to a value less than the minimum value sets the field to the minimum value. On a reset, the binary point field is set to the minimum supported value.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions:

- This register is banked to provide Secure and Non-secure copies, see *Register banking* on page 4-5.
- The Non-secure copy of the ICCBPR is aliased as the ICCABPR, see *Aliased Binary Point Register (ICCABPR)* on page 4-62, so that a processor can use Secure accesses to access the Non-secure ICCBR.
- The ICCICR.SBPR bit controls whether the Secure or Non-secure copy of the ICCBPR is used to determine the preemption of Non-secure interrupts, see *CPU Interface Control Register (ICCICR)* on page 4-47.

**Attributes** See the register summary in Table 4-2 on page 4-4.

Figure 4-22 shows the ICCBPR bit assignments.

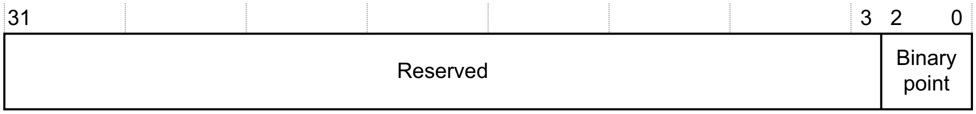


**Figure 4-22 ICCBPR bit assignments**

Table 4-28 shows the ICCBPR bit assignments.

**Table 4-28 ICCBPR bit assignments**

| Bits | Name | Function |
|---|---|---|
| [31:3] | - | Reserved. |
| [2:0] | Binary point | The value of this field controls how the 8-bit interrupt priority field is split into a group priority field, used to determine interrupt preemption, and a subpriority field. For how this field determines the interrupt priority bits assigned to the group priority field see:<br>• Table 3-2 on page 3-14, for a GIC that does not implement the Security Extensions, and for a GIC that implements the Security Extensions when processing a Secure interrupt<br>• Table 3-4 on page 3-23, for a GIC that implements the Security Extensions when processing a Non-secure interrupt.<br>See *Priority grouping* on page 3-14 for more information. |

**Note**

Aliasing the Non-secure ICCBPR as the ICCABPR means that, in a multiprocessor system, a processor that can make only Secure accesses to the GIC can access the Non-secure ICCBPR, to configure the preemption setting for Non-secure interrupts.

### 4.4.4 Interrupt Acknowledge Register (ICCIAR)

The ICCIAR characteristics are:

**Purpose** The processor reads this register to obtain the interrupt ID of the signaled interrupt. This read acts as an acknowledge for the interrupt.

**Usage constraints** There are no usage constraints.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.

**Attributes** See the register summary in Table 4-2 on page 4-4.

Figure 4-23 shows the IAR bit assignments.

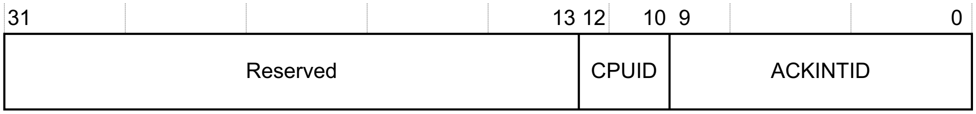


**Figure 4-23 ICCIAR bit assignments**

Table 4-29 shows the IAR bit assignments.

**Table 4-29 ICCIAR bit assignments**

| Bit | Name | Function |
|---|---|---|
| [31:13] | - | Reserved. |
| [12:10] | CPUID | For SGIs in a multiprocessor implementation, this field identifies the processor that requested the interrupt. It returns the number of the CPU interface that made the request, for example a value of 3 (0b011) means the request was generated by a write to the IDCSFGIR on CPU interface 3.<br>For all other interrupts this field is RAZ. |
| [9:0] | ACKINTID | The interrupt ID. |

A read of the ICCIAR returns the interrupt ID of the highest priority pending interrupt for the CPU interface, The read returns a spurious interrupt ID of 1023 if any of the following apply:

- Signalling of interrupts to the CPU interface is disabled
- There is no pending interrupt on this CPU interface with sufficient priority for the interface to signal it to the processor.

**Note**

The following sequence of events is an example of when the GIC returns an interrupt ID of 1023, and shows how reads of the ICCIAR can be timing critical:

1. A peripheral asserts a level-sensitive interrupt.
2. The interrupt has sufficient priority and therefore the GIC signals it to a targeted processor.
3. The peripheral deasserts the interrupt. Because there is no other pending interrupt of sufficient priority, the GIC deasserts the interrupt request to the processor.
4. Before it has recognized the deassertion of the interrupt request from stage 3, the targeted processor reads the ICCIAR. Because there is no interrupt with sufficient priority to signal to the processor, the GIC returns the spurious ID value of 1023.

---

The determination of the returned interrupt ID is more complex if the CPU interface implements the Security Extensions, see *Effect of the Security Extensions on reads of the ICCIAR* on page 4-58.

A non-spurious interrupt ID returned by a read of the ICCIAR is called a valid interrupt ID.

When the GIC returns a valid interrupt ID to a read of the ICCIAR it treats the read as an acknowledge of that interrupt and, as a side-effect of the read, changes the interrupt status from pending to active, or to active and pending if the pending state of the interrupt persists. Normally, the pending state of an interrupt persists only if the interrupt is level-sensitive and remains asserted.

For every read of a valid Interrupt ID from the ICCIAR, the interrupt service routine on the connected processor must perform a matching write to the ICCEOIR, see *End of Interrupt Register (ICCEOIR)* on page 4-59.

**Note**

- For compatibility with possible extensions to the GIC architecture specification, ARM recommends that software preserves the entire register value read from the ICCIAR, and writes that value back to the ICCEOIR when it has completed its processing of the interrupt.
- ARM recommends that software uses spin-locks to arbitrate which processor handles an SPI. When the GIC generates an SPI that is targeted at more than one processor, it cannot guarantee either of the following:
  - — only one processor receives the Interrupt ID of the SPI from a read of the ICCIAR
  - — other processors receive a spurious interrupt response.

  For more information see *Implications of the 1-N model* on page 3-8.

---

### Effect of the Security Extensions on reads of the ICCIAR

If a CPU interface implements the Security Extensions, whether a read of the ICCIAR returns a valid interrupt ID depends on:

- whether there is a pending interrupt of sufficient priority for it to be signaled to the processor, and if so, whether the highest priority pending interrupt is a Secure or a Non-secure interrupt
- whether the ICCIAR read access is Secure or Non-secure
- the value of the ICCICR.AckCtl bit, see *CPU Interface Control Register (ICCICR)* on page 4-47.

Reads of the ICCIAR that do not return a valid interrupt ID returns a spurious interrupt ID, ID 1022 or 1023, see *Special interrupt numbers when the Security Extensions are implemented* on page 3-16. Table 4-30 shows all possible ICCIAR reads for a CPU interface that implements the Security Extensions.

**Table 4-30 Effect of the Security Extensions on ICCIAR reads**

| Security of highest priority pending interrupt[a] | ICCIAR read | ICCICR.AckCtl | Returned interrupt ID |
|---|---|---|---|
| Non-secure | Non-secure | x | ID of Non-secure interrupt |
| | Secure | 1 | ID of Non-secure interrupt |
| | | 0 | Interrupt ID 1022 |
| Secure | Non-secure | x | Interrupt ID 1023 |
| | Secure | x | ID of Secure interrupt |
| No pending interrupts[a] or signalling of interrupts by CPU interface disabled | x | x | Interrupt ID 1023 |

[a] Of sufficient priority to be signaled to the processor.

### 4.4.5 End of Interrupt Register (ICCEOIR)

The ICCEOIR characteristics are:

**Purpose** A processor writes to this register to inform the CPU interface that it has completed its interrupt service routine for the specified interrupt.

**Usage constraints** There are no usage constraints.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.

**Attributes** See Table 4-2 on page 4-4.

Figure 4-24 shows the ICCEOIR bit assignments.

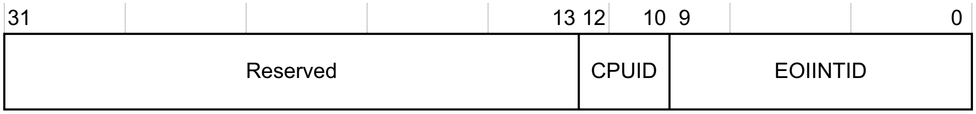


**Figure 4-24 ICCEOIR bit assignments**

Table 4-31 shows the ICCEOIR bit assignments.

**Table 4-31 ICCEOIR bit assignments**

| Bits | Name | Function |
|---|---|---|
| [31:13] | - | Reserved. |
| [12:10] | CPUID | On a multiprocessor implementation, on completion of the processing of an SGI, this field contains the CPUID value from the corresponding ICCIAR access.<br>In all other cases this field SBZ. |
| [9:0] | EOIINTID | The ACKINTID value from the corresponding ICCIAR access. |

Writing to this register causes the GIC to change the status of the identified interrupt:

- to inactive, if it was active
- to pending, if it was active and pending.

The interrupt service routine on the connected processor must write to the ICCEOIR for every read of a valid Interrupt ID from the ICCIAR, see *Interrupt Acknowledge Register (ICCIAR)* on page 4-56. The value written to the ICCEOIR must be the interrupt ID read from the ICCIAR.

——— **Note** ———

- For compatibility with possible extensions to the GIC architecture specification, ARM recommends that software preserves the entire register value read from the ICCIAR when it acknowledges the interrupt, and writes that value back to the ICCEOIR when it has completed its processing of the interrupt.

- If a read of the ICCIAR returns the ID of a spurious interrupt, software does not have to make a corresponding write to the ICCEOIR. If software writes the ID of a spurious interrupt to the ICCEOIR, the GIC ignores that write.

For nested interrupts, the order of interrupt completion must be the reverse of the order of interrupt acknowledgement. That is, the order of writes to the ICCEOIR must be the reverse of the order of reads from the ICCIAR.

If the values in the write to the ICCEOIR do not match a currently active interrupt, the effect of the write is UNPREDICTABLE.

If the CPU interface implements the Security Extensions, the possible effects of the write to the ICCEOIR are more complex, see *Effect of the Security Extensions on writes to the ICCEOIR*.

### Effect of the Security Extensions on writes to the ICCEOIR

If a CPU interface implements the Security Extensions, whether a write to the ICCEOIR removes the active status of the identified interrupt depends on:

- whether the ICCIAR write is Secure or Non-secure
- the value of the ICCICR.AckCtl bit, see *CPU Interface Control Register (ICCICR)* on page 4-47.

Table 4-32 shows all possible results of a write to the ICCEOIR.

**Table 4-32 Effect of the Security Extensions on ICCEOIR writes**

| Active interrupt is | ICCEOIR write | ICCICR.AckCtl | Active status removed |
|---|---|---|---|
| Non-secure | Non-secure | x | Yes |
| | Secure | 1 | Yes |
| | Secure | 0 | No |
| Secure | Non-secure | x | No |
| | Secure | x | Yes |

### 4.4.6 Running Priority Register (ICCRPR)

The ICCRPR characteristics are:

**Purpose** Indicates the priority of the highest priority interrupt that is active on the CPU interface.

**Usage constraints** If there is no active interrupt on the CPU interface, and the GIC implements 8-bit priority fields, a read of this register returns the value `0xFF`, corresponding to the lowest possible interrupt priority. If the GIC implements priority fields of less than 8 bits, the read might return the register reset value of `0xFF`, or might return a value corresponding to the lowest possible interrupt priority. Software cannot determine the number of implemented priority bits from a read of this register.

If the GIC implements the Security Extensions, the value returned by a Non-secure read of the Priority field is:

- `0x00` if the field value is less than `0x80`
- the Non-secure view of the Priority value if the field value is `0x80` or more.

For more information see *Non-secure access to register fields for Secure interrupt priorities* on page 4-8.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.

**Attributes** See the register summary in Table 4-2 on page 4-4.

Figure 4-25 shows the ICCRPR bit assignments.

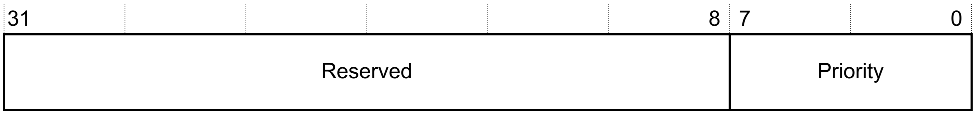


**Figure 4-25 ICCRPR bit assignments**

Table 4-33 shows the RPR bit assignments.

**Table 4-33 ICCRPR bit assignments**

| Bit | Name | Description |
|---|---|---|
| [31:8] | - | Reserved. |
| [7:0] | Priority | The priority value of the highest priority interrupt that is active on the CPU interface.<br>See the Usage constraints at the start of this section for more information about the value returned when software reads this field. |

### 4.4.7 Aliased Binary Point Register (ICCABPR)

The ICCABPR characteristics are:

**Purpose** Provides an alias of the Non-secure ICCBPR, see *Binary Point Register (ICCBPR)* on page 4-54.

**Usage constraints** Accessible by Secure accesses only.

**Configurations** Secure register, only implemented if the GIC implements the Security extensions. If the GIC does not implement the Security Extensions the ICCABPR address is RAZ/WI.

**Attributes** See the register summary in Table 4-2 on page 4-4.

For the register bit assignments, see Figure 4-22 on page 4-54 and Table 4-28 on page 4-55.

### 4.4.8 Highest Pending Interrupt Register (ICCHPIR)

The ICCHPIR characteristics are:

**Purpose** Indicates the Interrupt ID, and processor ID if appropriate, of the pending interrupt with the highest priority on the CPU interface.

**Usage constraints** Never returns the Interrupt ID of an interrupt that is Active and Pending. Returns a processor ID only for an SGI in a multiprocessor implementation.

If the GIC implements the Security Extensions, the value returned to a Non-secure read depends on whether the highest priority pending interrupt is Secure or Non-secure, see *Effect of the Security Extensions on reads of the ICCHPIR* on page 4-64.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.

**Attributes** See the register summary in Table 4-2 on page 4-4.

Figure 4-26 shows the ICCHPIR bit assignments.

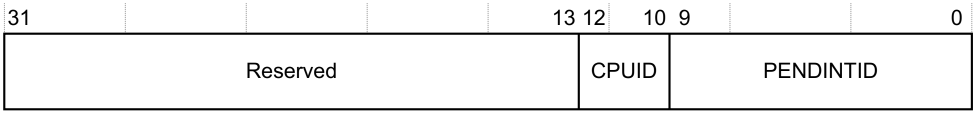


**Figure 4-26 ICCHPIR bit assignments**

Table 4-34 shows the ICCHPIR bit assignments.

**Table 4-34 ICCHPIR bit assignments**

| Bit | Name | Description |
|---|---|---|
| [31:13] | - | Reserved. |
| [12:10] | CPUID | On a multiprocessor implementation, if the PENDINTID field returns the ID of an SGI, this field contains the CPUID value for that interrupt. This identifies the processor that generated the interrupt.<br>In all other cases this field is RAZ. |
| [9:0] | PENDINTID | The interrupt ID of the highest priority pending interrupt. See Table 4-35 on page 4-64 for more information about the result of Non-secure reads of the ICCHPIR when the GIC implements the Security Extensions. |

### Effect of the Security Extensions on reads of the ICCHPIR

If a CPU interface implements the Security Extensions, whether a read of the ICCHPIR returns a valid interrupt ID depends on:

- whether there is a pending interrupt of sufficient priority for it to be signaled to the processor, and if so, whether the highest priority pending interrupt is a Secure or a Non-secure interrupt
- whether the ICCHPIR read access is Secure or Non-secure
- the value of the ICCICR.AckCtl bit, see *CPU Interface Control Register (ICCICR)* on page 4-47.

Reads of the ICCHPIR that do not return a valid interrupt ID returns a spurious interrupt ID, ID 1022 or 1023, see *Special interrupt numbers when the Security Extensions are implemented* on page 3-16.
Table 4-35 shows all possible ICCHPIR reads for a CPU interface that implements the Security Extensions.

**Table 4-35 Effect of the Security Extensions on ICCHPIR reads**

| Security of highest priority pending interrupt[a] | ICCHPIR read | ICCICR.AckCtl | Returned interrupt ID |
|---|---|---|---|
| Non-secure | Non-secure | x | ID of Non-secure interrupt |
| | Secure | 0 | Interrupt ID 1022 |
| | | 1 | ID of Non-secure interrupt |
| Secure | Non-secure | x | Interrupt ID 1023 |
| | Secure | x | ID of Secure interrupt |
| No pending interrupts[a] or forwarding of interrupts to CPU interface is disabled | x | | Interrupt ID 1023 |

[a] Of sufficient priority to be signaled to the processor.

### 4.4.9 CPU Interface Identification Register (ICCIIDR)

The ICCIIDR characteristics are:

**Purpose** Provides information about the implementer and revision of the CPU interface.

**Usage constraints** No usage constraints.

**Configurations** This register is available in all configurations of the GIC. If the GIC implements the Security Extensions this register is Common.

**Attributes** See the register summary in Table 4-2 on page 4-4.

Figure 4-27 shows the ICCIIDR bit assignments.

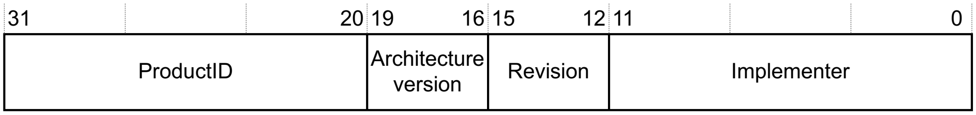


**Figure 4-27 ICCIIDR bit assignments**

Table 4-36 shows the ICCIIDR bit assignments.

**Table 4-36 ICCIIDR bit assignments**

| Bit | Name | Description |
|---|---|---|
| [31:20] | ProductID | An IMPLEMENTATION DEFINED product identifier.[a] |
| [19:16] | Architecture version | For an implementation that complies with this specification, the value is `0x1`. |
| [15:12] | Revision | An IMPLEMENTATION DEFINED revision number for the CPU interface. |
| [11:0] | Implementer | Contains the JEP106 code of the company that implemented the GIC CPU interface:[b]<br>**Bits [11:8]** The JEP106 continuation code of the implementer.<br>Bit **[7]** Always 0.<br>Bits **[6:0]** The JEP106 identity code of the implementer. |

[a] For an ARM implementation, the value of this field is `0x390`.

[b] For an ARM implementation, the value of this field is `0x43B`.

# Appendix A
# Pseudocode Index

This appendix gives an index of the pseudocode functions defined in this specification. It contains the following section:

- *Index of pseudocode functions* on page A-2.

The pseudocode in this document follows the ARM architecture pseudocode conventions. For more information, see *ARM Architecture Reference Manual ARMv7-A and ARMv7-R edition*.

## A.1 Index of pseudocode functions

Table A-1 is an index of the pseudocode functions defined in this specification. Where different forms of the function are used to support the architecture with and without the Security Extensions, the index refers to both forms.

**Table A-1 Pseudocode functions and procedures**

| Function | Meaning | See |
|---|---|---|
| `AnyActiveInterrupts()` | Return TRUE if any interrupt is in the active state. | *General helper functions and definitions* on page 3-25 |
| `BinaryMask()` | Return the priority mask to be used for priority grouping as part of interrupt prioritization | *General helper functions and definitions* on page 3-25 |
| `GenerateException()` | Exception generation by the CPU interface using the GIC prioritization scheme. | *Exception generation pseudocode, without the Security Extensions* on page 3-27<br>*Exception generation pseudocode, with the Security Extensions* on page 3-28 |
| `IgnoreWriteRequest()` | No operation. Indicates cases where the GIC ignores a write to a register. | *General helper functions and definitions* on page 3-25 |
| `IsPending()` | Return TRUE if the interrupt identified by the function argument is pending. | *General helper functions and definitions* on page 3-25 |
| `IsSecure()` | Return TRUE if the interrupt identified by the function argument is configured as a Secure interrupt. | *General helper functions and definitions* on page 3-25 |
| `PriorityIsHigher()` | Return TRUE if the first argument of the function has a higher priority than the second argument. | *General helper functions and definitions* on page 3-25 |
| `PriorityRegRead()` | Read behavior of accesses to the ICDIPR, ICCPMR and ICCRPR when the Security Extensions are implemented. | *The effect of the Security Extensions on accesses to prioritization registers* on page 3-29 |
| `PriorityRegWrite()` | Write behavior of accesses to the ICDIPR and ICCPMR when the Security Extensions are implemented. | *The effect of the Security Extensions on accesses to prioritization registers* on page 3-29 |
| `ReadICDIPR()` | Return the priority value of the interrupt identified by the function argument, by reading the appropriate ICDIPR. | *General helper functions and definitions* on page 3-25 |

**Table A-1 Pseudocode functions and procedures (continued)**

| Function | Meaning | See |
|---|---|---|
| `SignalFIQ()` | If the input parameter is TRUE, signal the target processor to request an FIQ exception. | *General helper functions and definitions* on page 3-25 |
| `SignalIRQ()` | If the input parameter is TRUE, signal the target processor to request an IRQ exception. | *General helper functions and definitions* on page 3-25 |
| `WriteICDIPR()` | Set the priority value of the interrupt identified by the function argument, by writing to the appropriate ICDIPR. | *General helper functions and definitions* on page 3-25 |

# Appendix B
# **Software Examples for the GIC**

This appendix gives examples of how software uses the GIC. It contains the following sections:

## B.1 Use of identification registers

The Distributor and each CPU interface include an identification register that software can read to discover information about the implementer and the device version, see:

- *Distributor Implementer Identification Register (ICDIIDR)* on page 4-16
- *CPU Interface Identification Register (ICCIIDR)* on page 4-65.

In addition, the architecture reserves a region in the Distributor memory map for identification registers for the GIC architecture, see *Identification registers* on page 4-42. The only architectural requirement for this region is a 4-bit field that defines the architecture version, see *Peripheral ID2 Register (ICPIDR2)* on page 4-42.

An ARM implementation of this GIC architecture uses the identification registers region to implement a set of identification registers that are consistent with the identification scheme used for PrimeCell and CoreSight components. For consistency with the PrimeCell and CoreSight schemes, these registers are named the Component ID and Peripheral ID registers. However, in the GIC scheme they identify only that the device implements a particular version of the ARM GIC architecture. They do not identify the implementer of the GIC, and do not provide any version information for the GIC implementation. Software can read these registers to discover that it is operating with a GIC that implements this architecture.

Software that knows it is operating with a GIC that implements this architecture can read the ICDICTR to find more information about the GIC implementation, see *Interrupt Controller Type Register (ICDICTR)* on page 4-14. This identifies:

- the maximum number of interrupts the GIC supports
- the number of CPU interfaces on the GIC
- whether the GIC implements the Security Extensions
- if the GIC implements the Security Extensions, the maximum number of lockable SPIs (LSPIs) it supports.

For more information about software discovery of the features of a GIC see *Initialization after reset or power on* on page B-3.

## B.2 Initialization after reset or power on

After a reset, the Distributor and CPU interfaces are disabled, and software must initialize the Distributor and all CPU interfaces.

——— **Note** ———

- Before enabling the CPU interfaces, software might have to discover which interrupts are implemented, see *Identifying the supported interrupts* on page 3-3.
- If the GIC implements the Security Extensions, only software running in Secure mode can initialize the GIC.

———————

For example, software can initialize the Distributor and each CPU interface to signal interrupts to each connected processor as follows:

1. The SPIs are the interrupts with IDs in the range 32-1019. For these interrupts:
   a. If the GIC implements the Security Extensions, write to the ICDISRs, to specify which interrupts are Non-secure, see *Interrupt Security Registers (ICDISRn)* on page 4-17.

      ——— **Note** ———

      - A reset sets all bits in the ICDISRs corresponding to the SPIs to 0, configuring all of the interrupts as Secure.
      - Bits in ICDISR0 can reset to either 0 or 1, see *ICDISR0 reset value* on page 4-18.

      ———————

   b. Write to the ICDICFRs to specify whether each interrupt is level-sensitive or edge-triggered, see *Interrupt Configuration Registers (ICDICFRn)* on page 4-36.
   c. Write to the ICDIPRs to specify the priority value for each interrupt, see *Interrupt Priority Registers (ICDIPRn)* on page 4-31.
   d. If the GIC is part of a multiprocessor implementation, write to the ICDIPTRs to specify the target processor list for each interrupt, see *Interrupt Processor Targets Registers (ICDIPTRn)* on page 4-33.
   e. Write to the ICDISERs to enable the interrupts, see *Interrupt Set-Enable Registers (ICDISERn)* on page 4-19.

2. The PPIs are the interrupts with IDs in the range 16-31, and the SGIs are the interrupts with IDs in the range 0-15. If the GIC is part of a multiprocessor implementation, these interrupts and the corresponding registers are banked for each connected processor, and each processor must configure its interrupts as follows:

   a. If the GIC implements the Security Extensions, write to ICDISR0, to specify which interrupts are Non-secure.

   b. For the PPIs, interrupt IDs 16-31, write to ICDICFR1 to specify whether each interrupt is level-sensitive or edge-triggered.

      **Note**

      For PPIs, it is IMPLEMENTATION DEFINED whether each interrupt is configurable as level-sensitive or edge-triggered. For any interrupt that is not configurable, reading the ICDICFR1 identifies whether the interrupt is level-sensitive or edge-triggered.

   c. Write to the appropriate ICDIPRs to specify the priority value for each interrupt.

   d. Write to ICDISER0 to enable the PPIs.

      **Note**

      The behavior of ICDISER0[15:0], the Set-enable bits for the SGIs, is IMPLEMENTATION DEFINED. Processors might have to write to these bits to enable SGIs.

3. For each implemented CPU interface:

   a. Write to the ICCPMR, to set the priority mask for the interface, see *Interrupt Priority Mask Register (ICCPMR)* on page 4-52.

   b. Write to the ICCBPR, to set the binary point position, that determines preemption on the interface, see *Binary Point Register (ICCBPR)* on page 4-54. If the GIC implements the Security Extensions you must set the binary point position in both the Secure and Non-secure copies of the ICCBPR.

   c. Write to the ICCICR to enable signalling of interrupts by the interface, see *CPU Interface Control Register (ICCICR)* on page 4-47.

      If the interface implements the Security Extensions, software can write to the Secure ICCICR to enable the signalling of both Secure and Non-secure interrupts. This write must also configure how the interface handles Secure and Non-secure interrupts, see *Security Extensions support* on page 3-15.

4. Write to the ICDDCR to enable the Distributor, see *Distributor Control Register (ICDDCR)* on page 4-12.

## B.3 Processor response to an initial interrupt

After the system software has initialized the GIC, the process of sending a first interrupt to a connected processor might be as follows:

1. In the GIC, the Distributor receives an interrupt.
2. The distributor sends the interrupt to the CPU interfaces specified in the target list for the interrupt.
3. If the interrupt priority is higher than the value in the ICCPMR, see *Interrupt Priority Mask Register (ICCPMR)* on page 4-52, the CPU interface signals the interrupt to the connected processor.
4. The processor branches to the appropriate interrupt vector determined by whether the GIC caused the assertion of an IRQ or an FIQ exception on that processor.

   If the processor implements the Security Extensions then:

   - System software defines its vector base address independently for operation in Secure and Non-secure state, The system can also program the processor to handle interrupts in Monitor mode. This is defined independently for IRQ and FIQ exceptions, and the software must defines a separate vector base address for exceptions handled in Monitor mode.
   - System software can configure the GIC to signal Secure interrupts using the FIQ exception request. By also configuring the processor to handle FIQ exceptions in Monitor mode it provides a mechanism for the processor to take a Secure interrupt when operating in Non-secure state, and switch to operating in Secure state. The Monitor mode code at the Monitor mode FIQ exception vector stacks the current state of the processor, including its Non-secure security state, and then changes operation to Secure state.

     If the GIC is signalling Secure interrupts using the FIQ exception request and Non-secure interrupts using the IRQ exception request, a processor operating in Secure state might handle Non-secure interrupts in Monitor mode, so that the Monitor mode handler code can change operation to Non-secure state.

   **Note**

   Monitor mode code must ensure that, when the processor starts operating in the other security state, the processor state is consistent with the appropriate interrupt having been taken in that security state.

5. The processor stacks the workspace.

6. If the interrupt is preempting another interrupt, the processor also:
   - stacks the IRQ or FIQ mode Link Register (LR) and SPSR values
   - changes to System or Supervisor mode and stacks all the state required for processing the original, preempted, interrupt.
   - returns to IRQ or FIQ mode to process the preempting interrupt.

7. The processor reads the ICCIAR to obtain the interrupt ID, see *Interrupt Acknowledge Register (ICCIAR)* on page 4-56, and checks that the interrupt ID is valid.

   If the ICCIAR read returns a spurious interrupt, the processor jumps to step 14. In this case it does not write to the End of Interrupt register. If the system implements the Security Extensions and the ICCIAR read returns the spurious interrupt ID of 1022, before jumping to step 14 the software running on the processor might inform the Secure scheduler that a Non-secure interrupt is pending, see *Effect of the Security Extensions on handling an initial interrupt* on page B-8.

8. If the interrupt is an SPI that targets more than one processor, the processor tries to obtain a lock on the interrupt service code routine. If it cannot obtain a lock, it assumes that another processor is processing the interrupt, and jumps to step 13.

9. To permit preemption of the interrupt, the processor ensures that the appropriate bit of the CPSR is set to 0. If the system implements the Security Extensions and the GIC is using the FIQ signal for Secure interrupts this is:
   - the I bit for Non-secure interrupts
   - the F bit for Secure interrupts.

10. In a cooperative system, the system must ensure that the security state of the processor corresponds to the security of the interrupt, Secure or Non-secure. This might require the processor to issue an `SMC` instruction, a Secure Monitor Call.

    For more information about cooperative systems see *Priority management and the Security Extensions* on page 3-24 and *ARM recommendations for interrupt handling with the Security Extensions* on page B-10.

11. The processor jumps to the *Interrupt Service Routine* (ISR) to execute the routine.

12. If the interrupt is an SPI or PPI, the ISR must service the requirements of the device that asserted the interrupt, so that the device de-asserts the interrupt. This is particularly important for level-sensitive interrupts, where the ISR must ensure the interrupt is de-asserted at the GIC input before proceeding to the next step.

13. If the interrupt ID is valid, the processor writes the interrupt ID to the ICCEOIR, see *End of Interrupt Register (ICCEOIR)* on page 4-59. When the GIC recognizes this write it removes the active status from the interrupt, for this processor.

14. The processor restores the workspace it stacked at step 5.

15. The processor returns from the interrupt vector. If the processor implements the Security state this might require it to enter Monitor mode to change its security state.

**Note**

If an ARMv6 processor uses the SRS and RFE instructions, it does not have to use the CPSR I bit to disable interrupts as part of its interrupt return routine.

### B.3.1 Effect of the Security Extensions on handling an initial interrupt

If the GIC implements the Security Extensions:

- If software running on a processor in Non-secure state reads the ICCIAR when no Non-secure interrupt with sufficient priority to be signaled to the processor is pending, the read returns interrupt ID 1023, indicating no outstanding interrupt, regardless of whether there are pending Secure interrupts of sufficient priority to be signaled to the processor.

- If software running on a processor in Secure state makes a Secure read of the ICCIAR when no interrupt with sufficient priority to be signaled to the processor is pending, the read returns interrupt ID1023, indicating no outstanding interrupt. If a Non-secure interrupt with sufficient priority to be signaled to the processor is pending, the result of the read depends on the value of the AckCtl bit in the Secure ICCICR, see *CPU Interface Control Register (ICCICR)* on page 4-47. The ID returned is one of:
  - — ID 1022, if AckCtl is 0. The GIC does not treat this read of the ICCICR as acknowledging any interrupt.
  - — The ID of the Non-secure interrupt, if AckCtl is 1. The GIC treats this read of the ICCICR as acknowledging the interrupt.

## B.4 Preemptive processing

The following sections describe how the GIC signals interrupts for possible preemption, and how a processor that implements the Security Extensions might process preempting interrupts:

- *CPU interface signalling of interrupts for possible preemption*
- *ARM recommendations for interrupt handling with the Security Extensions* on page B-10.

### B.4.1 CPU interface signalling of interrupts for possible preemption

When an interrupt is active on a CPU interface, indicating that software on the processor is handling the interrupt, the Distributor can send another interrupt to the CPU interface. The CPU interface uses the priority value of this interrupt to determine whether or not to signal the interrupt to the processor. The CPU interface signals the new interrupt to the processor only if the priority of that new interrupt is higher than the priority indicated by the ICCRPR, considering only the bits in the Group priority field as defined by the appropriate Binary Point Register.

For more information see:

- *Running Priority Register (ICCRPR)* on page 4-61
- *Priority grouping* on page 3-14
- if appropriate, *The effect of the Security Extensions on priority grouping* on page 3-23.

**Note**

- A smaller register priority field value always corresponds to a higher priority interrupt.
- The Distributor can forward a number of interrupts to a CPU interface, and all of those interrupts then have a state of pending on that CPU interface. If there is at least one active interrupt on the interface, the CPU interface considers whether it must signal the highest priority pending interrupt to the processor, for possible preemption.

### B.4.2 ARM recommendations for interrupt handling with the Security Extensions

In a system that implements the Security Extensions, a processor that implements the Security Extensions follows either a non-cooperative or a co-operative scheme for managing prioritization between Secure and Non-secure interrupts, see *Priority management and the Security Extensions* on page 3-24. The following sections give software recommendations for both schemes.

#### Non-cooperative scheme

In a processor that implements a non-cooperative scheme, ARM recommends that:

- The processor makes a Secure write to set the ICCICR.SBPR bit to 0, see *CPU Interface Control Register (ICCICR)* on page 4-47. This means the GIC CPU interface:
  - — uses the Non-secure ICCBPR to determine the preemption of Non-secure interrupts
  - — uses the Secure ICCBPR to determine the preemption of Secure interrupts.
- Divides the implemented interrupt priority range so that:
  - — all Secure interrupts use only priorities in the higher priority half of the priority range
  - — all Non-secure interrupts use only priorities in the lower priority half of the priority range.

Using this configuration:

- Non-secure software, that can only make Non-secure accesses to the GIC, writes to the Non-secure Binary Point Register to manage the preemption of Non-secure interrupts
- Secure software writes to the Secure Binary Point Register to manage the preemption of Secure interrupts.

#### Co-operative scheme

In a processor that implements a co-operative scheme, ARM recommends that:

- The processor makes a Secure write to set the ICCICR.SBPR bit to 1. This means the GIC CPU interface uses the Secure ICCBPR to determine the preemption of both Secure and Non-secure interrupts.
- The processor programs the priorities of Non-secure and Secure interrupts so that there is some overlap of priorities. That is, so that at least one Non-secure interrupt has higher priority than at least one Secure interrupt.

- On the processor, the cooperative interrupt handling software ensures that a preempted interrupt restarts correctly after processing a preempting interrupt of the opposite security.

## B.5 Generating a software interrupt

To generate an SGI, software writes to the ICDSGIR, specifying:

- the processors targeted by the interrupt
- the SGI interrupt ID.

If the system implements the Security Extensions, and the processor makes a Secure write to the register, the ICDSGIR.SATT bit value specifies the interrupt security required for the interrupt.

For more information see *Software Generated Interrupt Register (ICDSGIR)* on page 4-39.

The write changes the state of the interrupt from inactive to pending or from active to active and pending. It does this for:

- if the GIC does not implement the Security Extensions, all target CPU interfaces
- if the GIC implements the Security Extensions, any target CPU interfaces for which the specified SGI is configured to the security specified by the ICDSGIR.SATT bit, see Table 4-19 on page 4-41.

The priority of the SGI is the same on all targeted CPU interfaces, as defined in ICDIPR0, see *Interrupt Priority Registers (ICDIPRn)* on page 4-31.

If an SGI is active and pending on a CPU interface, it means that when the ISR on the connected processor writes to the ICCEOIR to indicate that it has completed its processing of the interrupt, the interrupt status on that interface becomes pending. If the SGI has sufficient priority the interface signals it to the processor.,

Software might use an SGI to post messages to other processors asynchronously. For example, each processor maintains in shared memory a queue of messages it wants to send to other processors. Each time it writes a new entry to the queue, it then writes to the ICDSGIR to send the appropriate SGI to the target processors. When the SGI is signalled to it, a target processor inspects the message queue maintained by the requesting processor, and performs each requested action, until it has drained the queue and marked all the messages as acted on.

——— **Note** ———

Race conditions can cause a target processor to receive a spurious interrupt from an SGI associated with adding a new entry to the queue while the target processor is processing entries in the queue.

## B.6 Changing a CPU interface interrupt priority mask

Each CPU interface has an ICCPMR, see *Interrupt Priority Mask Register (ICCPMR)* on page 4-52. If an interrupt has the priority indicated by the ICCPMR, or a lower priority, then the interface never signals it to the processor.

This section describes managing the interrupt priority mask in a system where both the GIC and a connected processor implement the Security Extensions.

If the processor uses a Non-secure write to the ICCPMR to set the mask to the highest possible priority then the CPU interface never signals any Non-secure interrupts to the processor. The CPU interface continues to signal Secure interrupts to the processor that have a priority that is higher than the mask priority.

If the processor uses a Secure write to the ICCPMR to set the mask to the highest possible priority then the CPU interface never signals any Secure or Non-secure interrupts to the processor. This enables software to perform a task without any possibility of an interrupt occurring.

If the processor has used a Secure write to programme the ICCPMR to a value that corresponds to a priority in the upper half of the priority range then the GIC CPU interface ignores any Non-secure write to the ICCPMR.

In any GIC implementation, a processor can modify the priority mask for its own interface to the GIC. It is IMPLEMENTATION DEFINED whether it has any access to the ICCPMR for any other CPU interface.

## B.7 Changing the priority of an interrupt

To change the priority of an interrupt, software writes to the corresponding ICDIPR, see *Interrupt Priority Registers (ICDIPRn)* on page 4-31. The ICDIPRs are byte accessible, so software can change the priority for a single interrupt field in an ICDIPR.

It is IMPLEMENTATION DEFINED whether a change to the priority of an interrupt applies to an active interrupt.

A Non-secure write to an ICDIPR can change only priority fields that correspond to Non-secure interrupts. A Secure write can change priority fields corresponding to both Secure and Non-secure interrupts.

Changing the priority of a pending interrupt can change the interrupt ID that is sent to a CPU interface.

## B.8 Changing the processor targets of an interrupt

To change the processor targets of an interrupt, software writes to the corresponding ICDIPTR, see *Interrupt Processor Targets Registers (ICDIPTRn)* on page 4-33. The ICDIPTRs are byte accessible, so software can change the priority for a single interrupt field in an ICDIPTR.

See *The effect of changes to an ICDIPTR* on page 4-35 for information about when changes to the targets list are effective.

A Non-secure write to an ICDIPTR can change only CPU targets fields that correspond to Non-secure interrupts. A Secure write can change CPU targets fields corresponding to both Secure and Non-secure interrupts.

**Note**

Secure software cannot restrict the processors targeted by a Non-secure interrupt.

## B.9 Disabling a peripheral interrupt

In any implementation, some interrupt might be permanently enabled, see *Identifying the supported interrupts* on page 3-3.

To disable an interrupt, software writes a 1 to the corresponding Clear-enable bit, see *Interrupt Clear-Enable Registers (ICDICERn)* on page 4-21. Disabling an interrupt has no effect on the active state of an interrupt and software must complete its processing of an active interrupt, even if it has disabled the interrupt.

When software disables a peripheral interrupt, the Distributor does not forward the interrupt to the CPU interface. However, if a peripheral asserts a disabled peripheral interrupt, or software generates an SGI targeted at a CPU interface on which it is disabled, then the Distributor still changes the state of that interrupt to pending.

If the interrupt is pending when software disables the interrupt, and the processor acknowledges another interrupt, the GIC might signal the interrupt to the processor before it recognizes the write to the ICDICER that disables the interrupt. If this happens, the interrupt becomes active and the processor must handle the interrupt.

Software can remove the pending state of a peripheral interrupt as follows:

1. If the interrupt is level-sensitive, ensure that the relevant peripheral is not asserting the interrupt signal.
2. Write a 1 to the Clear-pending bit that corresponds to the interrupt, see *Interrupt Clear-Pending Registers (ICDICPRn)* on page 4-26.
3. Write `0xFF` to the corresponding ICDIPR, see *Interrupt Priority Registers (ICDIPRn)* on page 4-31.

   **Note**

   Setting the priority to `0xFF` is the only way to disable a permanently-enabled interrupt.

## B.10 Changing the security configuration of an interrupt

**Note**

This section applies only to a GIC that implements the Security Extensions

Normally, software changes the security configuration of an interrupt as part of the larger task of transferring control of a peripheral either:

- from a Non-secure device driver to a Secure device driver
- from a Secure device driver to a Non-secure device driver.

Only a Secure access to the GIC can change the security configuration of an interrupt. This means the processor must be in Secure state to make this change. Typically, the software operates as follows:

1. Determine that the peripheral and any *interrupt service routines* (ISRs) that service interrupts generated by the peripheral are in a suitable state to transfer control from one security state to the other. This means either:
   - the peripheral has completed any outstanding tasks, and is idle
   - software can force the peripheral to abandon any outstanding tasks, and force any active ISRs for the interrupt to complete.
2. Prevent the peripheral from issuing any interrupts, for example, by writing to an interrupt mask in the peripheral, or by disabling the peripheral.
3. Disable the interrupt in the Distributor by writing a 1 to the corresponding Clear-Enable bit, see *Interrupt Clear-Enable Registers (ICDICERn)* on page 4-21.
4. Ensure that the interrupt is not pending, by reading the corresponding Set-pending or Clear-pending bit, see *Interrupt Set-Pending Registers (ICDISPRn)* on page 4-23 and *Interrupt Clear-Pending Registers (ICDICPRn)* on page 4-26. If the interrupt is pending, either:
   - write to the corresponding Clear-pending bit to remove its pending state
   - wait for it to complete.
5. Ensure that the interrupt is not active, by reading the corresponding Active bit, see *Active Bit Registers (ICDABRn)* on page 4-29. If the interrupt is active then wait for it to complete.

   **Note**

   In a multiprocessor system, the interrupt might be active on another CPU interface.

6. Program the required priority level, and targets for the new use of the interrupt. Normally, the security of the register writes used to do this corresponds to the required security of the interrupt. That is, if the interrupt will be Secure the software makes Secure writes, and if it will be Non-secure the software makes Non-secure writes.

   **Note**

   When transferring an interrupt from Secure to Non-secure use, if the Secure software knows the priority model used by the Non-secure software, it can use Secure writes for the reprogramming, specifying priorities in the lower half of the priority range (`0x80-0xFF`).

   Write to the corresponding:

   - ICDIPTR to program the targets, see *Interrupt Processor Targets Registers (ICDIPTRn)* on page 4-33
   - ICDIPR to program the priority, see *Interrupt Priority Registers (ICDIPRn)* on page 4-31.

7. Make a Secure write to the corresponding ICDISR to configure the interrupt as Secure or Non-secure, see *Interrupt Security Registers (ICDISRn)* on page 4-17.

8. Enable the interrupt in the Distributor by writing a 1 to the corresponding Set-Enable bit, see *Interrupt Set-Enable Registers (ICDISERn)* on page 4-19.

9. Enable the peripheral to issue interrupts, for example, by writing to an interrupt mask in the peripheral, or by enabling the peripheral.

## B.11 Disabling a CPU interface on the GIC

**Note**

The process described here is more likely to be required in a multiprocessor implementation, and is described only for such an implementation.

If software disables a CPU interface, for example before powering down a processor, interrupts might be lost. To ensure no interrupts are lost, ARM recommends the following procedure, for software running on the processor connected to the interface that is being disabled:

1. For each implemented interrupt ID, use Secure accesses to remove your own ID from the processor targets list. This requires a read, modify, write sequence on each ICDIPTR, see *Interrupt Processor Targets Registers (ICDIPTRn)* on page 4-33.
2. Read the ICCIAR, see *Interrupt Acknowledge Register (ICCIAR)* on page 4-56. If this read returns:
   a. a Spurious Interrupt, then there is no Pending interrupt for this CPU
   b. a valid interrupt ID, take the interrupt, and repeat this step.

   If the system implements the Security Extensions software must perform this step twice, as follows:
   - using Secure reads of the ICCIAR, to ensure there are no Secure interrupts pending on this interface
   - using Non-secure reads of the ICCIAR, to ensure there are no Non-secure interrupts pending on this interface.
3. Disable the CPU interface by writing 0 to the ICCICR.Enable bit, see *CPU Interface Control Register (ICCICR)* on page 4-47.

## B.12 Message passing between processors

Message passing involves two tasks:

1. Storing the message in shared memory, performing any operations required to ensure the message is visible to each target processor
2. Issuing an SGI to inform the target processors that the message is available.

In a system that implements the Security Extensions, a processor normally issues a Secure SGI to inform other processors of a secure message, and a Non-secure interrupt to inform them of a non-secure message. However, for a system that contains processors that operate in a fixed security state to send messages across security states, processors in Secure state must be able to respond to Non-secure interrupts, and must be able to signal messages using Non-secure interrupts. This might require a co-operative interrupt priority scheme, see *Priority management and the Security Extensions* on page 3-24.

## B.13 Example of using the binary point

Table B-1 shows three interrupt sources, and their priority field values and relative priorities. It also shows how a binary point value of 3 splits the priority value field into the group priority and subpriority fields, indicated by the g and s bits if the field value is shown as 0bgggg.ssss.

**Table B-1 Preemption example with three interrupts**

| Source | Interrupt | Priority field value | Relative priority | Priority field with binary point |
|---|---|---|---|---|
| a | A | 0b00001100 | Highest | 0b0000.1100 |
| b | B | 0b00010000 | Medium | 0b0001.0000 |
| c | C | 0b00010100 | Lowest | 0b0001.0100 |

The CPU interface uses only the group priority field to determine whether an interrupt can preempt the active interrupt. This means, for this example:

- interrupt A can preempt interrupt B or C if either is active on the processor
- interrupt B cannot preempt interrupt C if it is active on the processor.

The GIC uses all bits of the priority field to determine interrupt priority. If interrupt A is active on a processor, and interrupts B and C are pending, when the processor indicates completion of interrupt A, the GIC signals interrupt B to the processor, in preference to interrupt C.

**Note**

The GIC only signals a pending interrupt to the processor if the priority of that interrupt is higher than the priority indicated by the ICCPMR, see *Interrupt Priority Mask Register (ICCPMR)* on page 4-52.

For more information see:

- *Preemption* on page 3-13
- *Priority grouping* on page 3-14
- *The effect of the Security Extensions on interrupt prioritization* on page 3-18.

# Appendix C
# Register Shortform Names

This appendix describes the relationship between the architectural shortform names of the registers described in this specification, and their legacy shortform aliases. It also provides an index of the architectural shortform names. It contains the following sections:

- *Register name aliases* on page C-2.
- *Index of architectural shortform names* on page C-4.

## C.1 Register name aliases

Some implementations of this GIC architecture, for historical reasons, do not use the architectural shortform names of the registers described in this specification. Developers must not rely on this distinction being maintained in future versions of the ARM GIC architecture. Table C-1 shows the relationship of the shortform names for the registers in the Distributor.

**Table C-1 Shortform names for the registers in the Distributor**

| Register | Architectural shortform | Aliased shortform |
|---|---|---|
| Distributor Control | ICDDCR | enable_s, enable_ns |
| Interrupt Controller Type | ICDICTR | ic_type_reg |
| Distributor Implementor Identification | ICDIIDR | dist_ident_reg |
| Interrupt Security | ICDISR | int_security |
| Interrupt Set-Enable | ICDISER | enable_set |
| Interrupt Clear-Enable | ICDICER | enable_clr |
| Interrupt Set-Pending | ICDISPR | pending_set |
| Interrupt Clear-Pending | ICDICPR | pending_clr |
| Active Bit | ICDABR | active_status |
| Interrupt Priority | ICDIPR | priority_level |
| Interrupt Processor Targets | ICDIPTR | target |
| Interrupt Configuration | ICDICR | int_config |
| Software Generated Interrupt | ICDSGIR | sti_control |
| Identification | - | - |

Table C-1 on page C-2 shows the relationship of the shortform names for the registers in the CPU interface.

**Table C-2 Shortform names for the registers in the CPU interface**

| Register | Architectural shortform | Aliased shortform |
|---|---|---|
| CPU Interface Control | ICCICR | control_s, control_ns |
| Priority Mask | ICCPMR | priority_mask |
| Binary Point Register | ICCBPR | bin_pt_s, bin_pt_ns |
| Interrupt Acknowledge | ICCIAR | int_ack |
| End of Interrupt | ICCEOIR | EOI |
| Running Priority | ICCRPR | run_priority |
| Aliased Binary Point | ICCABPR | alias_bin_pt_ns |
| Highest Pending Interrupt | ICCHPIR | hi_pending |
| CPU Implementor Identification | ICCIIDR | cpu_ident |

## C.2 Index of architectural shortform names

Table C-3 is an alphabetic index of the GIC register shortform names, indexing the description of each register. An n at the end of a register name, as in ICDABRn, shows that there are multiple instances of the register.

**Table C-3 Index of register shortform names**

| Shortform name | Description |
|---|---|
| Component IDn | *Identification registers* on page 4-42 |
| ICCABPR | *Aliased Binary Point Register (ICCABPR)* on page 4-62 |
| ICCBPR | *Binary Point Register (ICCBPR)* on page 4-54 |
| ICCEOIR | *End of Interrupt Register (ICCEOIR)* on page 4-59 |
| ICCHPIR | *Highest Pending Interrupt Register (ICCHPIR)* on page 4-63 |
| ICCIAR | *Interrupt Acknowledge Register (ICCIAR)* on page 4-56 |
| ICCICR | *CPU Interface Control Register (ICCICR)* on page 4-47 |
| ICCIIDR | *CPU Interface Identification Register (ICCIIDR)* on page 4-65 |
| ICCPMR | *Interrupt Priority Mask Register (ICCPMR)* on page 4-52 |
| ICCRPR | *Running Priority Register (ICCRPR)* on page 4-61 |
| ICDABRn | *Active Bit Registers (ICDABRn)* on page 4-29 |
| ICDDCR | *Distributor Control Register (ICDDCR)* on page 4-12 |
| ICDICERn | *Interrupt Clear-Enable Registers (ICDICERn)* on page 4-21 |
| ICDICFRn | *Interrupt Configuration Registers (ICDICFRn)* on page 4-36 |
| ICDICPRn | *Interrupt Clear-Pending Registers (ICDICPRn)* on page 4-26 |
| ICDICTR | *Interrupt Controller Type Register (ICDICTR)* on page 4-14 |
| ICDIIDR | *Distributor Implementer Identification Register (ICDIIDR)* on page 4-16 |
| ICDIPRn | *Interrupt Priority Registers (ICDIPRn)* on page 4-31 |
| ICDIPTRn | *Interrupt Processor Targets Registers (ICDIPTRn)* on page 4-33 |
| ICDISERn | *Interrupt Set-Enable Registers (ICDISERn)* on page 4-19 |
| ICDISPRn | *Interrupt Set-Pending Registers (ICDISPRn)* on page 4-23 |

**Table C-3 Index of register shortform names (continued)**

| Shortform name | Description |
|---|---|
| ICDISRn | *Interrupt Security Registers (ICDISRn)* on page 4-17 |
| ICDSGIR | *Software Generated Interrupt Register (ICDSGIR)* on page 4-39 |
| Peripheral IDn | *Identification registers* on page 4-42 |

# Glossary

**Banked register**

Is a register that has multiple instances, with the instance that is in use depending on the processor mode, security state, or other processor state.

**IMP** Is an abbreviation used in diagrams to indicate that the bit or bits concerned have IMPLEMENTATION DEFINED behavior.

**IMPLEMENTATION DEFINED**

Means that the behavior is not architecturally defined, but should be defined and documented by individual implementations.

**Observer**

A processor or mechanism within the system, such as peripheral device, that is capable of generating reads from or writes to memory.

**Read-As-One (RAO)**

In any implementation, the bit must read as 1, or all 1s for a bit field.

**Read-As-Zero (RAZ)**

In any implementation, the bit must read as 0, or all 0s for a bit field.

**RAO** *See* Read-As-One.

**RAO/WI** Read-As-One, Writes Ignored.

In any implementation, the bit must read as 1, or all 1s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 1, or all 1s for a bit field, and on writes being ignored.

**RAZ** *See* Read-As-Zero.

**RAZ/WI** Read-As-Zero, Writes Ignored.

In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the field must be ignored.

Software can rely on the bit reading as 0, or all 0s for a bit field, and on writes being ignored.

**Reserved**

Registers and instructions that are reserved are UNPREDICTABLE unless otherwise stated. Bit positions described as Reserved are UNK/SBZP.

**SBZ** *See* Should-Be-Zero.

**SBZP** *See* Should-Be-Zero-or-Preserved.

**Security hole**

Is a mechanism that bypasses system protection.

**Should-Be-Zero (SBZ)**

Should be written as 0 (or all 0s for a bit field) by software. Values other than 0 produce UNPREDICTABLE results.

**Should-Be-Zero-or-Preserved (SBZP)**

Must be written as 0, or all 0s for a bit field, by software if the value is being written without having been previously read, or if the register has not been initialized. Where the register was previously read on the same processor, since the processor was last reset, the value in the field should be preserved by writing the value that was previously read.

Hardware must ignore writes to these fields.

If a value is written to the field that is neither 0 (or all 0s for a bit field), nor a value previously read for the same field on the same processor, the result is UNPREDICTABLE.

**UNKNOWN**

An UNKNOWN value does not contain valid data, and can vary from moment to moment, instruction to instruction, and implementation to implementation. An UNKNOWN value must not be a security hole. UNKNOWN values must not be documented or promoted as having a defined value or effect.

**UNK/SBZP**

UNKNOWN on reads, Should-Be-Zero-or-Preserved on writes.

In any implementation, the bit must read as 0, or all 0s for a bit field, and writes to the field must be ignored.

Software must not rely on the field reading as 0, or all 0s for a bit field, and must use an SBZP policy to write to the field.

**UNK** Software must treat a field as containing an UNKNOWN value.

In any implementation, the bit must read as 0, or all 0s for a bit field. Software must not rely on the field reading as zero.

### UNPREDICTABLE

The behavior cannot be relied upon. UNPREDICTABLE behavior must not represent security holes. UNPREDICTABLE behavior must not halt or hang the processor, or any parts of the system. UNPREDICTABLE behavior must not be documented or promoted as having a defined effect.

### Unsigned data types

Represent a non-negative integer in the range 0 to $+2^N-1$, using normal binary format.