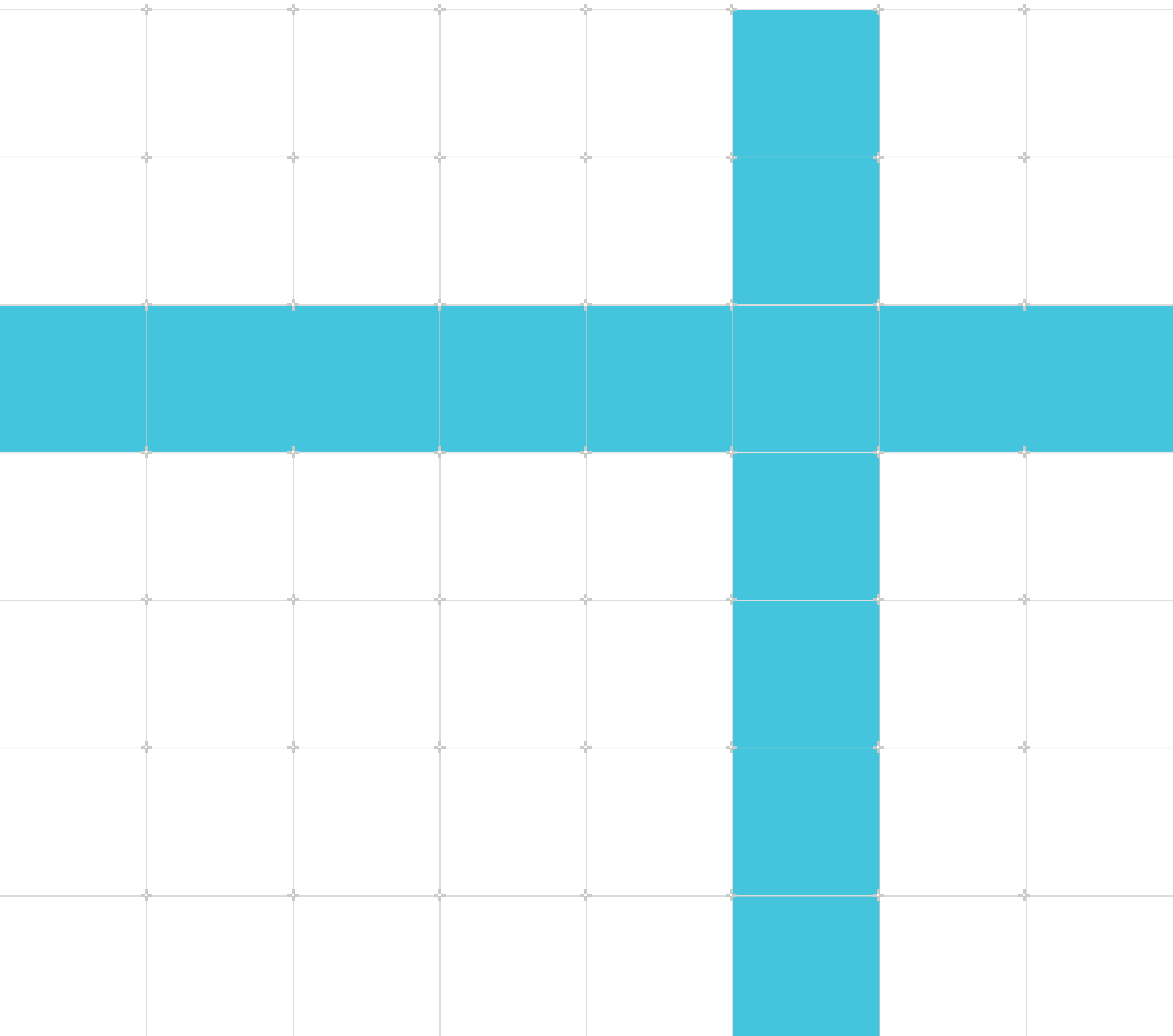
arm

Learn the architecture - Generic Interrupt Controller v3 and v4, Virtualization

Version 1.1

Issue 02
107627_0101_02_en



Learn the architecture - Generic Interrupt Controller v3 and v4, Virtualization

Release information

Document history

Issue	Date	Confidentiality	Change
0100-02	18 July 2022	Non-Confidential	First release
0101-01	27 September 2022	Non-Confidential	Technical error fix
0101-02	13 October 2022	Non-Confidential	Update images

Proprietary Notice

Arm Limited. Company 02557590 registered in England.

110 Fulbourn Road, Cambridge, England CB1 9NJ.

(LES-PRE-20349|version 21.0)

Confidentiality Status

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

Unrestricted Access is an Arm internal classification.

Product Status

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

Feedback

Arm welcomes feedback on this product and its documentation. To provide feedback on the product, create a ticket on https://support.developer.arm.com

To provide feedback on the document, fill the following survey: https://developer.arm.com/documentation-feedback-survey.

Inclusive language commitment

Arm values inclusive communities. Arm recognizes that we and our industry have used language that can be offensive. Arm strives to lead the industry and create change.

We believe that this document contains no offensive language. To report offensive language in this document, email terms@arm.com.

Contents

1. Introduction **6**

2. Virtualization **7**

3. GICv3 - Virtualization **9**
3.1 Interfaces 9
3.2 Managing virtual interrupts 11
3.3 Example of a physical interrupt being forwarded to a vPE 12
3.4 Maintenance interrupts 13
3.5 Context switching 13

4. GICv3.1 - Secure virtualization **15**

5. GICv4.1 - Direct injection of virtual interrupts **16**
5.1 Overview 16
5.2 Redistributor 18
5.3 Doorbells 20
5.4 ITS 24
5.5 Removing mappings 27
5.6 Changing vLPI configuration 27

6. GICv4.1 - Direct injection of vSGIs **28**

7. Example **31**

8. Check your knowledge **35**

1. Introduction

This guide describes the support for virtualization in the GICv3 and GICv4 architecture. It covers the controls available to a hypervisor for generating and managing virtual interrupts. The guide is for anyone who needs to understand the capabilities of the interrupt controller or who needs to write software to manage virtual interrupts.

This guide describes the features present in GICv3.x and GICv4.1. It does not cover GICv4.0 other than as an introduction.

This document complements the Arm® Generic Interrupt Controller Architecture Specification GIC architecture version 3 and 4. It is not a replacement or alternative. Refer to the Arm® Generic Interrupt Controller Architecture Specification GIC architecture version 3 and 4 for detailed descriptions of registers and behaviors.

At the end of this guide you will be able to:

- List the different ways that virtual interrupts can be generated.
- Name the registers used by software to manage GIC virtualization within the CPU interface.
- Describe how GICv4.1 allows virtual interrupts to be directly injected.

Before you begin

This guide assumes familiarity with the GIC's support for physical interrupts. If you have not already done so, you should read the Learn the architecture: Arm Generic Interrupt Controller v3 and v4 guide.

This guide also assumes familiarity with the support for virtualization in the Armv8-A architecture. For background information on virtualization in the Arm architecture, see the Learn the architecture - AArch64 Virtualization guide.

2. Virtualization

Armv8-A includes optional support for virtualization. To complement this functionality, GICv3 also supports virtualization. Support for virtualization in GICv3 adds the following functionality:

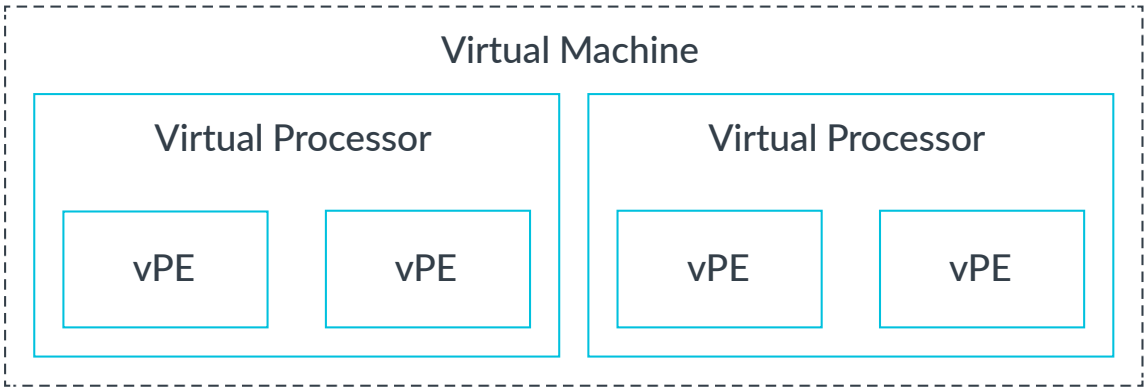
- Hardware virtualization of the CPU interface registers.
- The ability to generate and signal virtual interrupts.
- Maintenance interrupts, to inform supervising software (such as a hypervisor) about specific events within a virtual machine.

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

Terminology

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

Figure 2-1: Virtual machine, virtual processor and virtual PE



The virtualization support in GICv3.x and GICv4.1 works at the level of vPEs. For example, when creating a virtual interrupt, it is targeted at a specific vPE, not a VM. In general, the GIC does not know how different vPEs relate to the virtual machines. This is important to remember when thinking about some of the controls that are introduced later.

This guide uses the term hypervisor to mean any software running at EL2 which is responsible for managing vPEs. In this guide, we ignore the differences that can exist between virtualization

software because we are concentrating on the features in the GIC. However, remember that not all virtualization solutions use all the features available within the GIC.

A given vPE can be described as scheduled or not-scheduled. A scheduled vPE is one that has been scheduled by the hypervisor to a physical PE (pPE) and is running. A system might contain more vPEs than pPEs. A vPE that is not scheduled by the hypervisor is not running and therefore cannot currently receive interrupts.

3. GICv3 - Virtualization

This section gives an overview of the support for virtualization in GICv3. GICv3 virtualization is similar to the support first introduced in GICv2 and is mainly within the CPU interface. It allows virtual interrupts to be signaled to the currently scheduled vPE on a pPE.

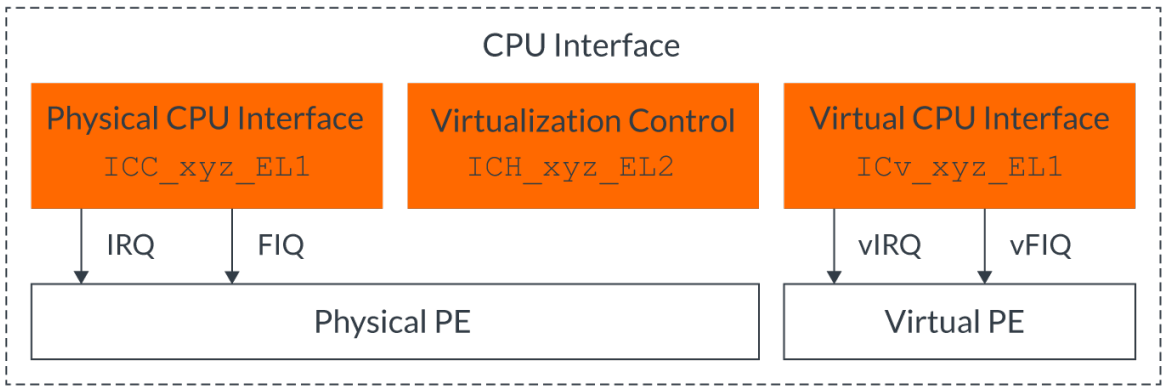
3.1 Interfaces

The CPU interface registers are split into three groups:

- ICC: Physical CPU interface registers
- ICH: Virtualization control registers
- ICV: Virtual CPU interface registers

The following image shows the three groups of CPU interface regsters:

Figure 3-1: CPU interface registers with virtualization



Physical CPU interface registers

These registers have names with the format ICC_*_ELx.

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

Virtualization control registers

These registers have names with the format ICH_*_EL2.

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

- Enabling and disabling the virtual CPU interface.

- Accessing virtual register state to enable context switching.
- Configuring maintenance interrupts.
- Controlling virtual interrupts for the currently scheduled vPE.

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

Virtual CPU interface registers

These registers have names with the format ICV_*_EL1.

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

The ICV and ICC registers have the same instruction encodings. At EL2 and EL3, the ICC registers are always accessed. At EL1, the routing bits in HCR_EL2 determine whether the ICC or the ICV registers are accessed.

The ICV registers are split into three groups:

Group 0

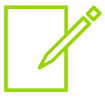
Registers used for handling Group 0 interrupts, for example ICC_IAR0_EL1 and ICV_IAR0_EL1. When HCR_EL2.FMO==1, ICV registers rather than ICC registers are accessed at EL1.

Group 1

Registers used for handling Group 1 interrupts, for example ICC_IAR1_EL1and ICV_IAR1_EL1. When HCR_EL2.IMO==1, ICV registers rather than ICC registers are accessed at EL1.

Common

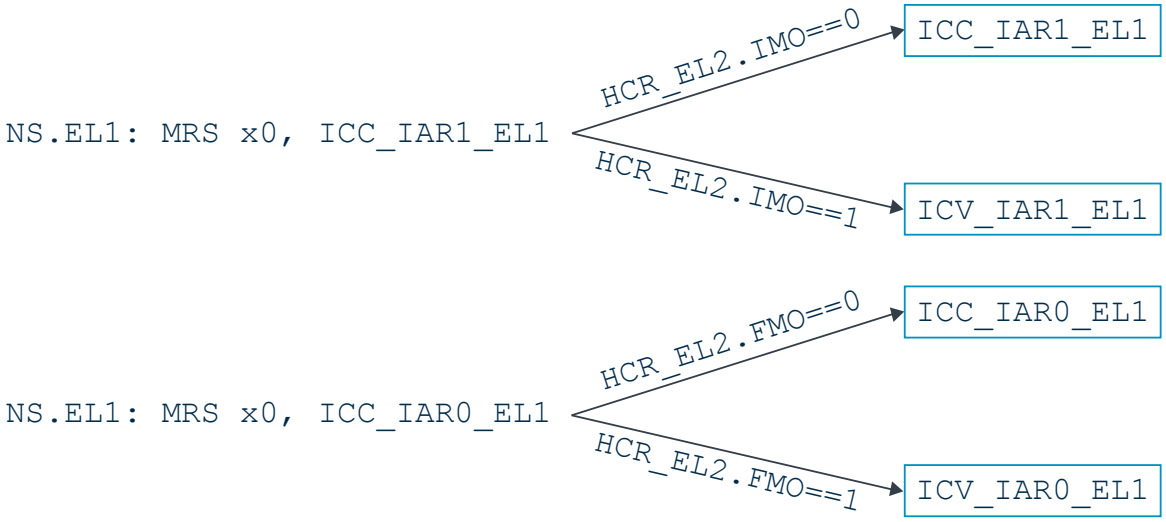
Registers used for handling both Group 0 and 1 interrupt, for example ICC_DIR_EL1 and ICV_DIR_EL1. When either HCR_EL2.IMO==1 or HCR_EL2.FMO==1, ICV registers rather than ICC registers are accessed at EL1.


Note

Whether the ICV registers are used in Secure EL1 depends on whether Secure virtualization is enabled. More on this later.

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

Figure 3-2: Example of ICC or ICV register selection



3.2 Managing virtual interrupts

A hypervisor can generate virtual interrupts for the currently scheduled vPE using the List registers, ICH_LR<n>_EL2. Each register represents one virtual interrupt, and records the following information:

vINTID (virtual INTID)

The INTID reported in the virtual environment.

State

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

Group

In Non-secure state, the virtual environment always behaves as if GICD_CTLR.DS==1. In Secure state, the virtual environment behaves as if GICD_CTLR.DS==0 with FIQs routed to EL1. Therefore, in both cases virtual interrupts can be Group 0 or Group 1. Group 0 interrupts are delivered as vFIQs. Group 1 interrupts are delivered as vIRQs.

pINTID (physical INTID)

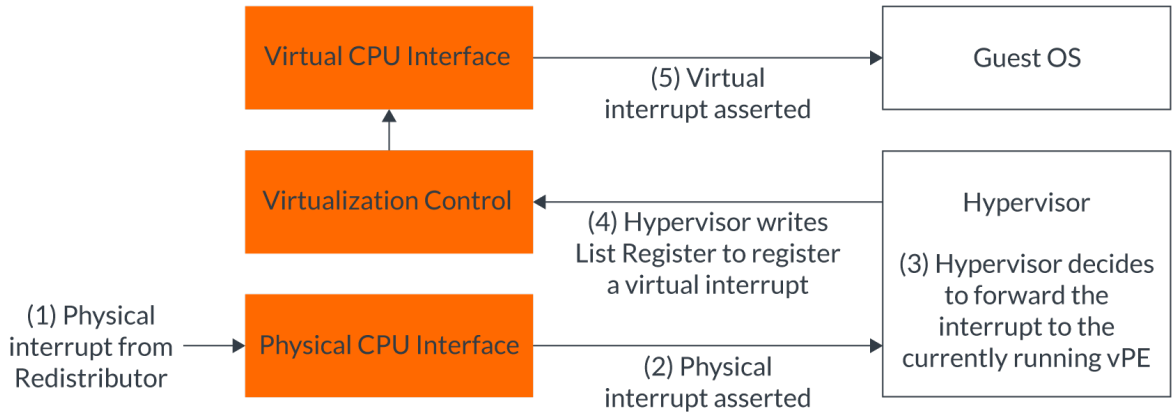
A virtual interrupt can be optionally tagged with the INTID of a physical interrupt. When the state machine of the vINTID is updated, so is the state machine of the pINTID.

The List Registers do not record the target vPE. The List Registers implicitly target the currently scheduled vPE, it is the responsibility of the software to context switch the List Registers when changing the scheduled vPE.

3.3 Example of a physical interrupt being forwarded to a vPE

The following diagram shows an example sequence of a physical interrupt that is forwarded to a vPE:

Figure 3-3: Example of forwarding a physical interrupt to a vPE



The sequence proceeds as follows:

1. A physical interrupt is forwarded to the physical CPU interface from the Redistributor.
2. The physical CPU interface checks whether the physical interrupt can be forwarded to the PE. In this instance, the checks pass, and a physical exception is asserted.
3. The interrupt is taken to EL2. The hypervisor reads the IAR, which returns the pINTID. The pINTID is now in the Active state. The hypervisor determines that the interrupt is to be forwarded to the currently running vPE. The hypervisor writes the pINTID to ICC_EOIR1_EL1. With ICC_CTLR_EL1.EOImode==1, this only performs priority drop without deactivating the physical interrupt.
4. The hypervisor writes one of the List registers to register a virtual interrupt as pending. The List register entry specifies the vINTID that is to be sent and the original pINTID. The hypervisor then performs an exception return, returning execution to the vPE.
5. The virtual CPU interface checks whether the virtual interrupt can be forwarded to the vPE. These checks are the same as for physical interrupts, other than that they use the ICV registers. In this instance, the checks pass, and a virtual exception is asserted.

6. The virtual exception is taken to EL1. When software reads the IAR, the vINTID is returned and the virtual interrupt is now in the Active state.
7. The Guest OS handles the interrupt. When it has finished handling the interrupt, it writes the EOIR to perform a priority drop and deactivation. As the List register recorded the pINTID, this deactivates both the vINTID and pINTID.

This example shows a physical interrupt being forwarded to a vPE as a virtual interrupt. This could, for example, be from a peripheral assigned to the VM by the hypervisor. Not all virtual interrupts need be due to a physical interrupt. Virtualization software can create virtual interrupts within the List Registers at any time.

3.4 Maintenance interrupts

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

These interrupts are reported as a PPI, with INTID 25. This interrupt is typically configured as Non-secure Group 1 and handled by the hypervisor software at EL2.

The generation of maintenance interrupts is controlled by ICH_HCR_EL2, and the interrupts that are currently asserted are reported in ICH_MISR_EL2.

Maintenance interrupt example

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

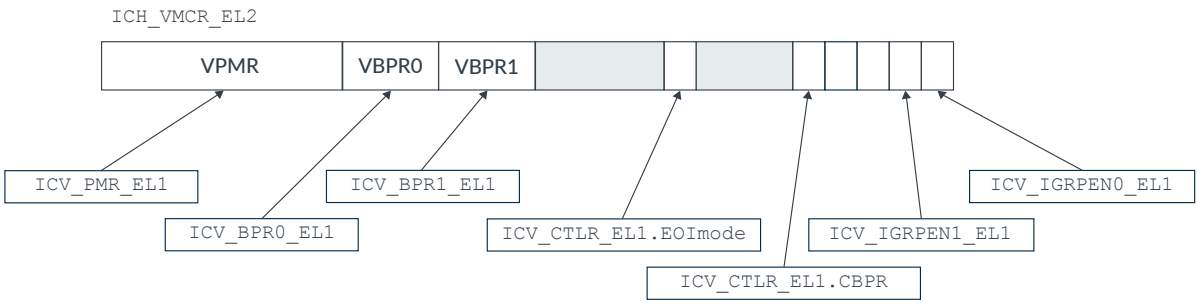
3.5 Context switching

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

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

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

Figure 3-4: Accessing ICV state from EL2



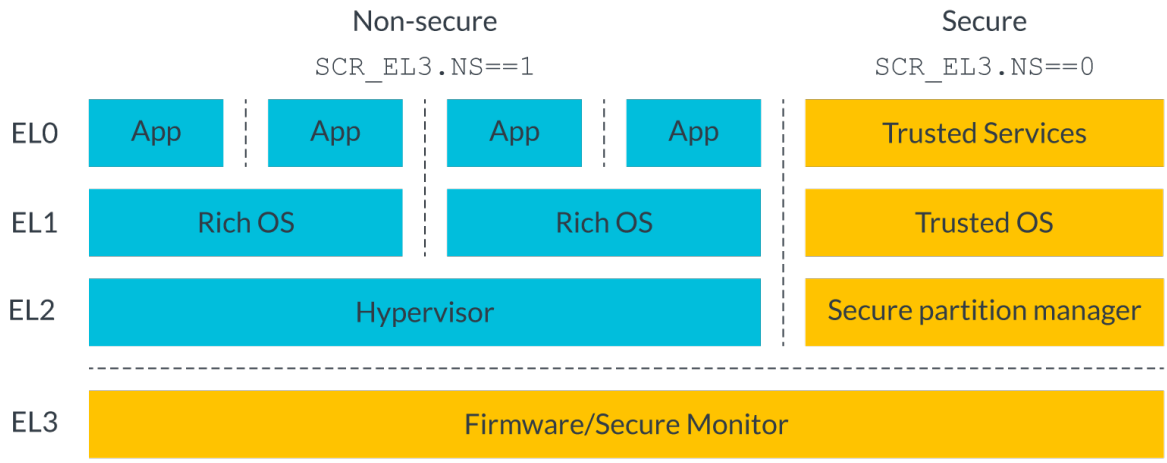
The active virtual priorities must be saved and restored when switching vPEs. The active priorities for the current vPE can be accessed using the ICH_APxRn_EL2 registers.

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

4. GICv3.1 - Secure virtualization

Armv8.4-A introduced support for virtualization in Secure state, as shown in the following diagram:

Figure 4-1: Exception levels with Secure virtualization



When supported by a PE, support for Secure virtualization can be enabled or disabled using SCR_EL3.EEL2.

GICv3.1 extends the GICv3.0 support for virtualization to Secure state, to align with Armv8.4-A. When SCR_EL3.EEL2==1, all the features described in the previous section also apply in Secure state.

There are some minor differences between Secure and Non-secure state virtualization. In Non-secure state, the virtual environment always behaves as if GICD_CTLR.DS==1. In Secure state, the virtual environment behaves as if GICD_CTLR.DS==0 with FIQs routed to EL1. For most register accesses, this distinction makes not practical difference. However, when writing ICV_BPR1_EL1 it changes what the minimum permitted value is.

Sharing the maintenance interrupt

There is only one GIC maintenance interrupt, shared by the different virtualization software in the Secure and Non-secure states. One approach to dealing with this would be to save and restore the configuration of this interrupt on changing Security state.

5. GICv4.1 - Direct injection of virtual interrupts

GICv4 inherits all the support for virtualization introduced in the previous sections. It adds support for the direct injection of virtual interrupts. This feature allows software to describe to the ITS how physical events map to virtual interrupts in advance. If the vPE targeted by a virtual interrupt is running, the virtual interrupt can be forwarded without the need to first enter the hypervisor. This can reduce the overhead associated with virtualized interrupts, by reducing the number of times the hypervisor is entered.

GICv4.0 supports directly injecting virtual LPIs (vLPIs). GICv4.1 extends support to also cover virtual SGIs (vSGIs). There are several changes between GICv4.0 and GICv4.1 which make them incompatible with each other. This guide covers the GICv4.1 programming model.

Direct injection, in both GICv4.0 and GICv4.1, is limited to the Non-secure state. Direct injection is not supported in Secure state.

5.1 Overview

This section starts with a brief overview of how direct injection works in GICv4.1. The following sections provide more detail.

GICv4.1 allows software to define several virtual PEs (vPE), and map physical interrupts to those vPEs. A vPE is identified by a vPEID (virtual Processing Element ID). The vPEID is a global identifier, shared by all the Redistributors and ITSs in the system.

The configuration and state of vPEs is stored in memory-based tables. This is similar to how the configuration and state of physical LPIs are managed. There are three types of memory-based table used by the Redistributors for managing virtual interrupts:

Virtual LPI Pending Table

There is one Virtual LPI Pending table per-vPE. It stores the pending state of virtual interrupts targeting that vPE.

Virtual LPI Configuration Table

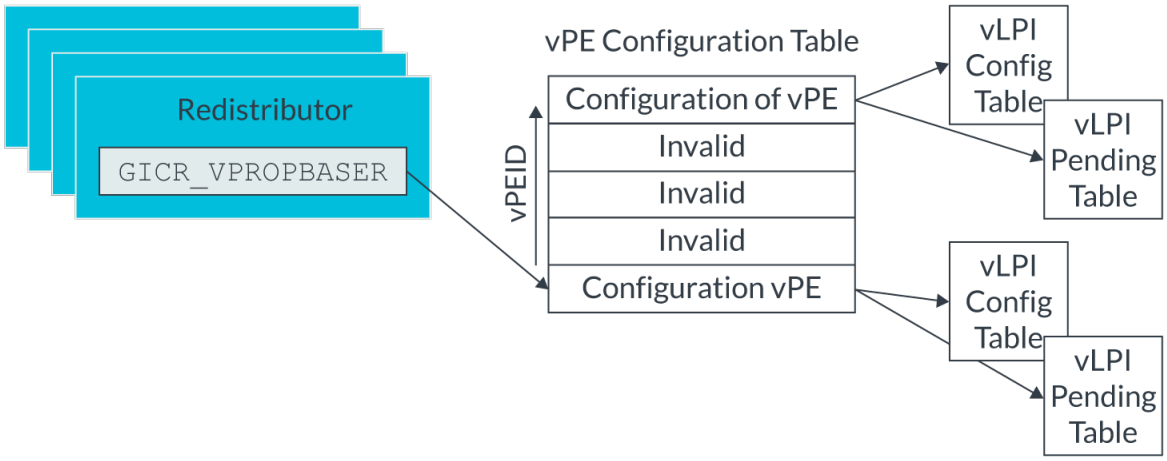
The Virtual LPI Configuration Table stores the configuration (enable and priority) of vLPIs. A virtual Configuration table may be shared by multiple vPEs. For example, all the vPEs in one VM might share a Virtual LPI Configuration table.

vPE Configuration Table

The vPE Configuration stores the settings for the all vPEs. There is one entry in the table per-vPE, storing pointers to that vPE's virtual Pending and Configuration tables. A vPE Configuration Table entry also stores other information about the vPE, such as how big the vINTID namespace is. A vPE Configuration Table is shared by multiple Redistributors, typically there is one copy of the table per-SoC.

The following diagram shows the relationship between these tables:

Figure 5-1: Redistributor memory structures for vPEs



The location and size of the vPE Configuration Table is specified by software using the GICR_VPROPBASER register. The entries in that table are populated as a side effect of issuing ITS commands. This will be covered later.

The GIC needs to know which, if any, vPE is currently scheduled on a physical PE (pPE). For GICv4.1, the vPE currently scheduled is specified in GICR_VPENDBASER.

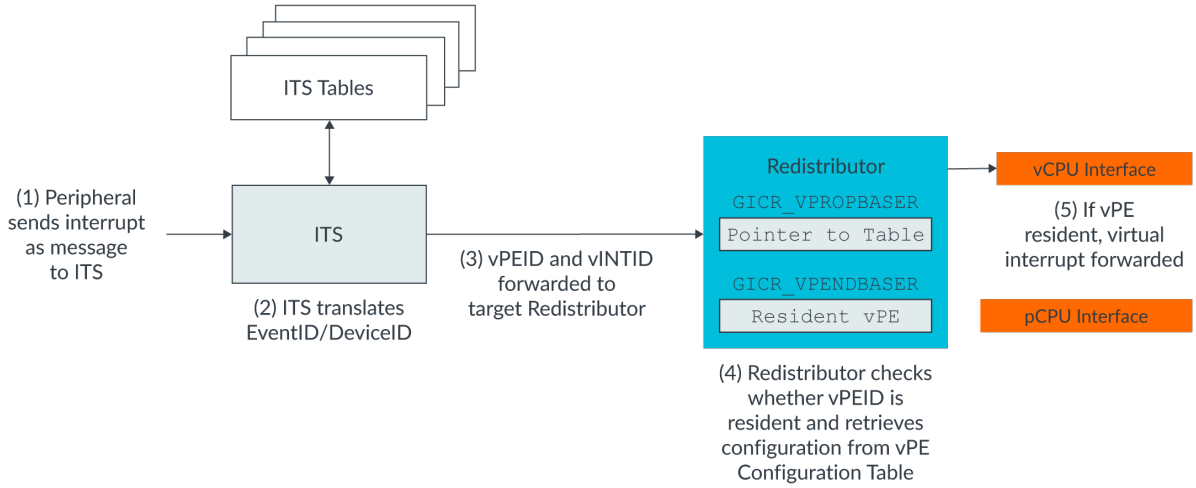
Note

This is a significant difference between GICv3 and GICv4. In both GICv3 and GICv4 a virtual interrupt can only be delivered to the currently scheduled vPE. In GICv3, the hardware does not know the ID of the scheduled vPE, rather it is software's responsibility to manage this on context switching. In GICv4, the hardware needs to know which vPE is currently scheduled as interrupts can arrive at any time.

Software uses ITS commands to create and manage vPEs. `VMAPP` defines a new vPE, specifying its configuration and the location of the virtual Pending and Configuration tables. This information is stored in the vPE Configuration Table. `VMAPI` and `VMAPTI` map physical interrupts to virtual interrupts targeting a specific vPE.

The following diagram shows what happens when an interrupt targeting a vPE arrives:

Figure 5-2: Overview of the direct injection mechanism



1. The peripheral sends an MSI to the ITS
2. The ITS translates the EventID/DeviceID in the MSI. The returned mapping indicates that the interrupt is mapped to a vPE, rather than a physical LPI.
3. The ITS forwards the interrupt to the target Redistributor, sending the vINTID and vPEID of the interrupt.
4. The Redistributor retrieves the configuration for the vPE and vINTID from the vPE Configuration Table. It also checks whether the vPE is scheduled, using GICR_VPENDBASER.
5. If the vPE is scheduled, the interrupt is forwarded to the Virtual CPU interface. Otherwise, the interrupt is recorded as pending, and is delivered the next time the vPE is scheduled.

The ITS and Redistributors can cache information from the different tables. Therefore, in practice not all interrupts require memory accesses to retrieve table contents.

5.2 Redistributor

The Redistributor retrieves the configuration for the vPE and vINTID from the vPE Configuration Table.

CommonLPIAff groups

Redistributors are grouped together, with the groups defined by GICR_TYPER.CommonLPIAff and GICR_TYPER.Affinity. CommonLPIAff acts as a mask on the Affinity value, Redistributors with the same affinity value after the mask is applied are part of the same group.

For example, if CommonLPIAff==2 then all Redistributors with the same Aff3.Aff2 value are in the same group.

Consider a system with four Redistributors, with the following affinities:

- `0.0.0.0`
- `0.0.0.1`
- `0.1.0.0`
- `0.1.0.1`

After the mask is applied, this gives us:

- `0.0.x.x`
- `0.0.x.x`
- `0.1.x.x`
- `0.1.x.x`

That is, we have two groups `0.1.x.x` and `0.0.x.x`.

CommonLPIAff groups are expected to be Redistributors which are physically close to each other. For example, in a multi-chip design there might be one group for each chip.

The CommonLPIAff value is important as it determines how many memory structures software must allocate and what Redistributors a vPE can be scheduled on. We will discuss this in the following sections.

The vPE Configuration Table

The vPE Configuration Table stores details of all the vPEs. The size of the table dictates how many vPEs can be created by software.

The vPE Configuration Table is populated and maintained by the GIC, as a side effect of ITS commands. Software is never expected to read or write the table after the memory has been given to the GIC. Doing so can cause the GIC to behave incorrectly.

Software must allocate a copy of the table per CommonLPIAff group. That is, if there are two CommonLPIAff groups, software must allocate enough memory for two copies of the table. This is a performance optimization, as it allows the Redistributors to use memory which is close to them.

Software must allocate the required number of tables and populate GICR_VPROPBASER of each active Redistributor before creating mapping vPEs.

Redistributors belonging to different CommonLPIAff groups must not share the same copy of the vPE Configuration Table.

Controlling which vPE is scheduled

Which vPE is currently running on a PE is defined by GICR_VPENDBASER. To change the scheduled vPE, software must:

Clear GICR_VPENDBASER.Valid

Clearing the Valid bit informs the Redistributor that a context switch is taking place. The Redistributor retrieves any pending virtual interrupts from the virtual CPU interface and ensures that the Virtual LPI Pending Table in memory is correct.

Poll GICR_VPENDBASER.Dirty until it reads 0

The Dirty bit reports when the Redistributor has finished updating its internal state. This includes retrieving any pending virtual interrupts for the old vPE from the vCPU interface. A new vPE cannot be scheduled until this bit reads 0. Arm recommends that virtualization software does not context switch the ICH registers until Dirty has been observed to be 0.

Update GICR_VPENDBASER, setting Valid==1 in the process

Setting the Valid bit to 1 informs the Redistributor that the new vPEID is now valid, and that virtual interrupts for that vPE can be forwarded. GICR_VPENDBASER also contains the virtual Distributor Group enables, which controls which virtual interrupt groups can be forwarded to the CPU interface.

Optional: Poll GICR_VPENDBASER.Dirty until it reads 0

On Valid being written to 1, the GIC searches for interrupts for the newly scheduled vPE. Dirty reads as 1 until either the GIC has found an interrupt it can deliver, or it has completed walking the pending table and found no pending interrupts.

In the ITS, a vPE is mapped to a specific Redistributor. That mapping can change over time, but at any given point there is a single target Redistributor for a vPE. However, a vPE may be scheduled on any Redistributor that is a member of the same CommonLPIAff group as the target Redistributor. Scheduling on a Redistributor that is part of a different group can cause the GIC to misbehave.

In a single-chip design it is possible that all the Redistributors are part of the same CommonLPIAff group. In this case you would be able to schedule the vPE on any Redistributor.

Software must not:

- Set a vPEID as scheduled on any Redistributor before mapping that vPE in the ITS.
- Mark the same vPEID as scheduled on multiple Redistributors.

5.3 Doorbells

Hypervisors typically divide vPEs into three categories:

Running

The vPE is currently scheduled by the hypervisor and to a physical PE. For the GIC, this means the vPE can receive directly injected virtual interrupts.

Runnable or to-be-scheduled

The vPE is not scheduled on any physical PE. The hypervisor knows that there is work for the vPE to do, so schedules it at some point in the future. Virtual interrupts cannot currently be delivered to this vPE by the GIC.

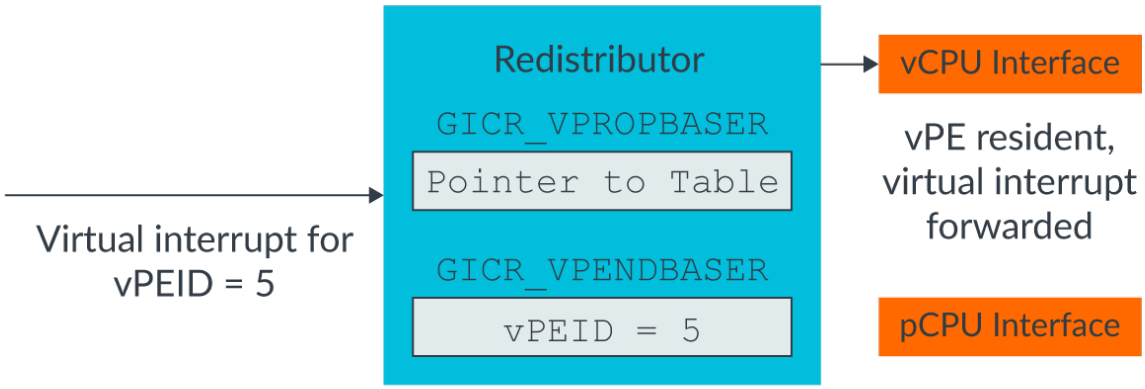
Idle

The vPE is not currently scheduled on any physical PE. The hypervisor believes there is no work for the vPE to do, and therefore does not schedule it in the future.

A vPE moves from Idle to Runnable when work for the vPE becomes available. One way this could happen is an interrupt arriving targeting the vPE. But this requires the hypervisor to be aware that the interrupt has arrived. The mechanism for this is called a doorbell interrupt.

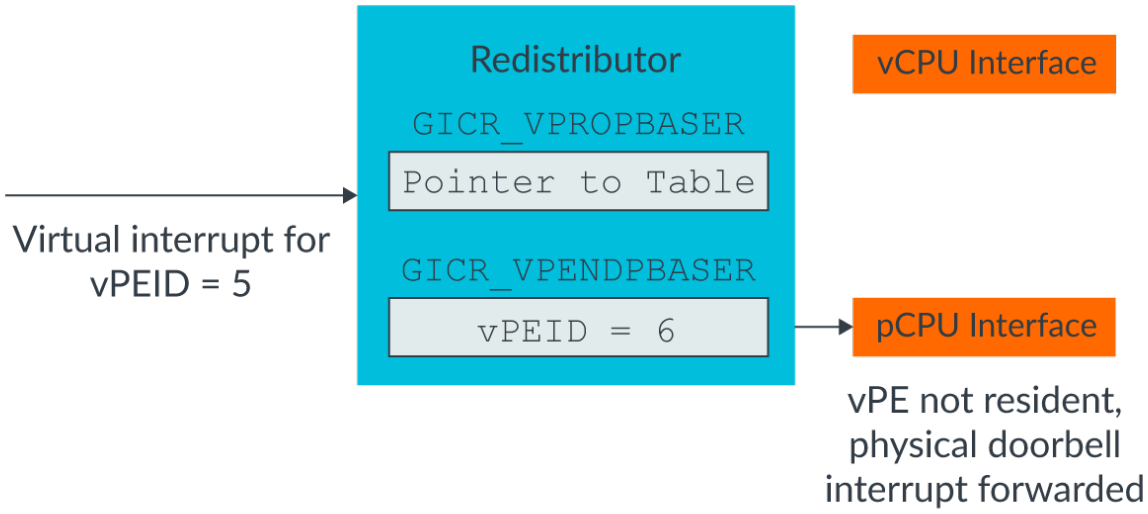
When a virtual interrupt arrives, if the target vPE is scheduled the interrupt can be forwarded to the CPU interface:

Figure 5-3: Virtual interrupt for scheduled vPE



When the vPE is not scheduled, a doorbell interrupt can optionally be generated instead:

Figure 5-4: Virtual interrupt for non-scheduled vPE causing a physical doorbell interrupt



This doorbell is a physical interrupt and would typically be taken to EL2 and handled by the hypervisor. It signals to the hypervisor that there is a pending interrupt for the non-scheduled vPE, meaning that it should be moved to the Runnable queue for future scheduling.

GICv4.1 supports two types of doorbell:

- Default doorbells
- Individual doorbells (support is optional in GICv4.1)

Default doorbells

Each vPE can be assigned a default doorbell. A default doorbell is generated when any interrupt targeting that vPE becomes pending and the vPE is not scheduled.

The architecture makes several guarantees for default doorbells:

- The default doorbell for a vPE is set to pending no more than once between residencies.

  Once the vPE is moved from Idle to Runnable, software does not need further doorbells for that vPE. The vPE is already going to be scheduled.
- A default doorbell is only generated if it was requested when the vPE was made non-scheduled.

  If on being made non-scheduled the hypervisor is already going to mark the vPE as runnable, it does not need a doorbell. Receiving one would be inefficient.
- A default doorbell is only generated if the pending interrupt is enabled.

  Software only wants to know of pending interrupts that would have been forwarded to the vPE. If the interrupt is disabled, then we do not need to make the vPE runnable.

- A pending default doorbell is cleared by making the vPE scheduled.

  If there is still an outstanding default doorbell for a vPE when it is scheduled, that interrupt is cleared. As the hypervisor no longer needs to know there is working for the vPE to perform.

Default doorbell generation is controlled by two bits in GICR_VPENDBASER:

GICR_VPENDBASER.PendingLast
: When a vPE is made non-scheduled, this bit reports whether there are outstanding pending interrupts for the vPE.

GICR_VPENDBASER.Doorbell
: Software sets this bit to indicate whether it wants a doorbell generated.

When PendingLast reports that there are pending interrupts for the vPE, Doorbell is treated as being 0 (no doorbell). Otherwise it would imply that a doorbell must be generated immediately.

The INTID used for a vPE's default doorbell is set using ITS commands, we will cover this later.

Software does not need to register a default doorbell when creating a vPE. Setting the default doorbell INTID as 1023 means no doorbell.

Changing the configuration of default doorbells

Default doorbells are physical LPIs, meaning that their configuration is stored in memory. Specifically, in the physical LPI configuration table. As covered in the Locality-Specific Peripheral Interrupts guide, if software wants to change the configuration of a physical LPI it writes to the table and then invalidates any caching of the old configuration using an INV command.

Although default doorbells are physical LPIs, they behave differently to other LPIs in terms of caching. Software must issue an `INVDB` command for INTIDs used as default doorbells.

Individual doorbells

An individual doorbell can optionally be set per-virtual interrupt, rather than per-vPE. This means that a hypervisor could potentially take different actions depending on which interrupt targeting the vPE had become pending. For example, most interrupts could use the default doorbell and just cause the vPE to be marked as runnable. A high priority interrupt could be assigned an individual doorbell and cause immediate rescheduling.

Individual doorbells do not have all the same guarantees that the default doorbells do. In particular:

- There is no guarantee that an individual doorbell is set pending once between residencies.
- Software cannot register whether it wants an individual doorbell on making a vPE non-scheduled. If one has been supplied for the virtual interrupt, it is generated while the vPE is non-scheduled.

Software can allocate the same physical INTID for multiple virtual interrupts, as long as all those interrupts belong to the same vPE.

Support for individual doorbells is optional, with support reported by `GITS_TYPER.nID`.

Software does not need to register an individual doorbell when mapping a virtual interrupt. Setting the Doorbell INTID as 1023 means no doorbell.

5.4 ITS

The ITS is responsible for translating incoming MSIs and forwarding them as virtual interrupts. In a previous guide, we covered the basic translation mechanism. If you are not already familiar with this process, read the Locality-Specific Peripheral Interrupts guide.

vPE Table

For virtual interrupts, the Interrupt Translation Tables (ITTs) record which vPE an interrupt targets and the virtual INTID. The ITS also needs to record which group of Redistributors a vPE is mapped to. There are two ways an ITS can do this, GITS_TYPER.SVEPT indicates which model is supported:

SVEPT==0

The ITS uses a private table to record the vPE mappings. Software must allocate memory for this table and set GITS_BASER2 to point at the allocated memory.

SVEPT==1

The ITS reuses the Redistributors' vPE Configuration Table. Software must set GITS_BASER2 to point at the vPE Configuration Table allocated for the Redistributors.

As with the other ITS tables, these structures must be allocated before the ITS is enabled. This applies to both models.

Mapping a vPE

A vPE is created using the `VMAPP` command:

```
VMAPP <vPEID>, <RDADDR>, <VPT size>, <VPT address>, <VCT address>, <doorbell>
```

In this command:

- `vPEID` is the ID of the vPE.
- `RDADDR` is the target Redistributor.
- `VPT address` and `VCT address` are the addresses of the virtual Pending and Configuration tables.
- `VPT size` specifies the width in bits of the vINTID used for the vPE. From this, the sizes of the Pending and Configuration tables are determined.

- `doorbell` is the physical INTID of the vPE's default doorbell. Specifying 1023 (spurious) means that the vPE has no doorbell interrupt.

Software must not create a vPE using VMAPP before the vPE Configuration Table is allocated and set up in the target Redistributor.

Mapping MSI to vINTID

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

```
VMAPI <DeviceID>, <EventID>, <pINTID>, <vPEID>
```

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

```
VMAPTI <DeviceID>, <EventID>, <vINTID>, <pINTID>, <vPEID>
```

In these commands:

- `DeviceID` and `EventID` together identify the interrupt that is being mapped.
- `vPEID` is the ID of the vPE.
- `vINTID` is the INTID of the virtual LPI. For VMAPI, EventID and vINTID have the same value.
- `pINTID` is the individual doorbell interrupt that is generated if the vPE is not scheduled. Specifying 1023 means that there is no individual doorbell interrupt.

Software must not map interrupts to a vPE before creating the vPE with a `VMAPP` command.

Example

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

- EventID 0 – vINTID 8725, no individual doorbell interrupt
- EventID 1 – vINTID 9000, no individual doorbell interrupt

vPE 6 is mapped to the Redistributor number 7 and uses 8192 as its default doorbell.

The command sequence for this is as follows:

```
VMAPP 6, 7, 14, <Pending Table Addr>, <Config Table Addr>, 8192
VMAPTI 5, 0, 8725, 1023, 6
VMAPTI 5, 1, 9000, 1023, 6
VSYNC 6
```

The example assumes that GITS_TYPER.PTA==0, and that a `MAPD` command has previously been issued to map the ITT.

Remapping a vPE to a different Redistributor

If a hypervisor migrates a vPE to a Redistributor that is part of a different CommonLPIAff group, the ITS mappings must be updated so that virtual interrupts are sent to the correct location. The ITS mappings are updated using the `VMOVP` command, followed by `VSYNC` to synchronize the context.


Note

Doorbell interrupts are always delivered to the mapped Redistributor, but vPEs can be scheduled to any Redistributor within the same CommonLPIAff group. If software wants the doorbell interrupts of a vPE delivered to a different PE, it must issue a `VMOVP` command.

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

GITS_TYPER.VMOVP==1

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

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

Arm expects this to be model implemented by most GICs.

GITS_TYPER.VMOVP==0

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

```
VMOVP <vPEID>, <RDADDR>, <ITS List>, <Sequence Number>
```

In this command:

- `vPEID` is the ID of the vPE.
- `RDADDR` is the Redistributor that the vPE is being remapped to.
- `ITS List` is a list of all the ITSs with mappings for the vPE. This field is encoded as one bit per-ITS, where bit 0 maps to ITS 0. The number of an ITS is reported by GITS_CTLR.ITS_Number.
- `Sequence Number` is the synchronization point. Software must use the same value when issuing the `VMOVP` command to the different ITSs and must not reuse the same value until the commands have completed on all ITSs.

Remapping vINTIDs

The `VMOVI` command remaps an EventID and DeviceID combination to a different vINTID and vPE.

```
VMOVI <DeviceID>, <EventID>, <vPEID>, <vINTID>, <pINTID>
```

In this command:

- `DeviceID` and `EventID` together identify the interrupt that is being remapped.

- `vPEID` is the ID of the vPE that the interrupt is being moved to.
- `vINTID` is the virtual INTID that the interrupt should now use.
- `pINTID` is the individual doorbell interrupt that is generated if the vPE is not scheduled. Specifying 1023 means that there is no individual doorbell interrupt.

5.5 Removing mappings

The mappings commands, `VMAPP` and `VMAPTI`, have a `V` field. When `V==1`, they are treated as mapping commands. When `V==0`, they are treated as unmapping commands.

When removing the mapping for a vPE, software must first remove all the interrupt mappings for that vPE.

5.6 Changing vLPI configuration

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

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

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

In GICv4.1, invalidation can also be carried out using the GICR_INVLPIR register in each Redistributor. Arm expects that software uses either the commands or the registers but does not regularly mix use of both approaches.

6. GICv4.1 - Direct injection of vSGIs

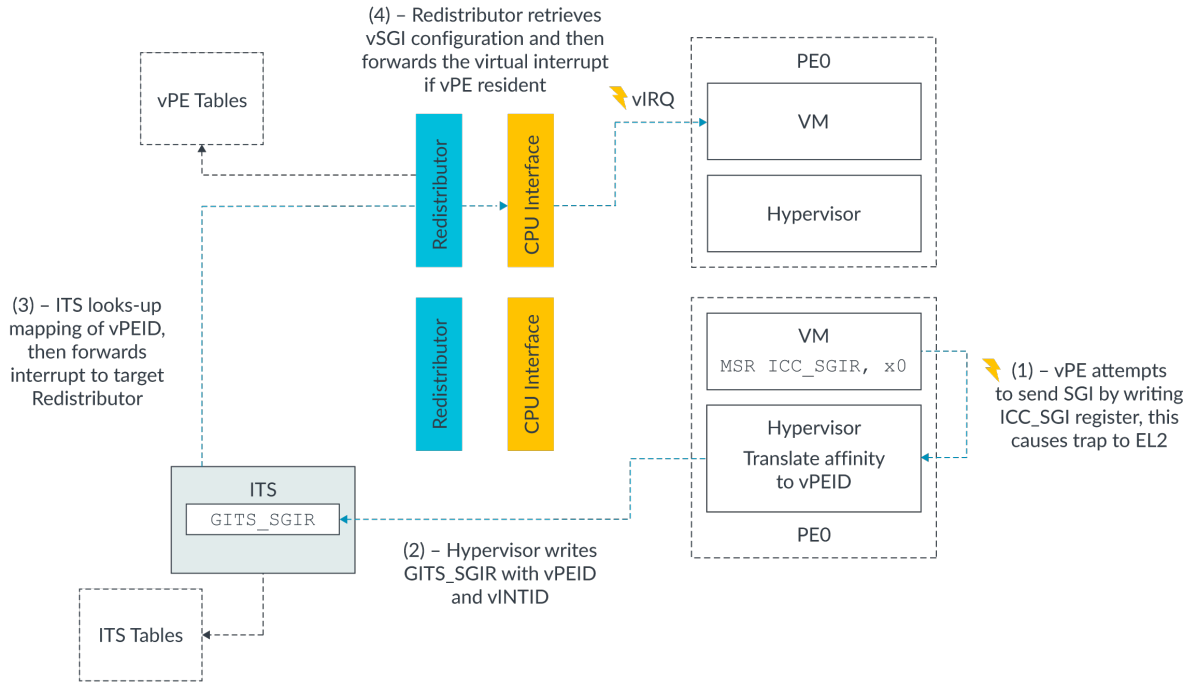
GICv4.1 introduces a new feature, the ability to directly inject virtual SGIs (vSGI). This feature further reduces overhead by removing some of the cases where the hypervisor needs to be entered.

Sending a virtual SGI

Software generates a vSGI by writing to the GITS_SGIR register in the ITS. Software writes the vPEID of the target vPE and the INTID of the SGI being sent. The ITS looks up the mapping for the specified vPE and then forwards the vSGI to the target Redistributor.

The following diagram shows the flow for sending a vSGI:

Figure 6-1: Process of sending a vSGI



1. Software running in the VM writes to one of the ICC_SGI registers to generate an SGI. This write contains the INTID being sent and the affinity values of the target vPE(s). This register write triggers a trap exception to EL2.
2. The hypervisor translates the affinity value written by the VM to a vPEID, then writes GITS_SGIR to generate a virtual interrupt.
3. The ITS looks up the target Redistributor for the specified vPEID, then forwards the vSGI to that Redistributor.

4. If the target vPE is scheduled, the Redistributor retrieves the interrupt's configuration and forwards the interrupt to the CPU interface. The CPU interface then signals the vSGI as a virtual interrupt exception.

If the vPE is not scheduled, the interrupt is recorded as pending. Optionally, a default doorbell might be generated.

This process still requires some degree of hypervisor interaction to translate the virtual affinity value written to the ICC_SGIR register into a vPEID. But after this, the GIC's direct injection mechanism can handle the rest of the process.

SGI configuration

In order to directly inject vSGIs, the Redistributor needs to know the configuration (enable, priority, and group) of the SGIs. This information is recorded in the virtual LPI Pending Table. In the Locality-Specific Peripheral Interrupts guide we introduced the Pending table and described how the first 1K of that table is used to record implementation-specific information. GICv4.1 uses a small portion of that space to store the vSGI configuration.

Unlike vLPIs, software cannot directly write the table to update the configuration of vSGIs. Instead, the configuration is set using a new ITS command:

```
VSGI <vPEID>, <vINTID>, <Enable>, <Group>, <Priority>, <Clear>
```

Where:

- `vPEID` identifies the target vPE.
- `vINTID` is the vSGI being updated.
- `Enable`, `Group` and `Priority` are that vSGIs configuration.
- `Clear` can be used to clear the pending state of the interrupt.

Sending Group versus receiving Group

When software generates an SGI, the written register specifies the Group being sent:

- ICC_SGI0R_EL1: Send Group 0 interrupt
- ICC_SGI1R_EL1: Send Group 1 interrupt

The receiver also specifies a Group for each SGI INTID. For physical interrupts this is specified using the GICR_IGROUPR0 and GICR_IGRPMODR0 registers. For virtual SGIs, as we have just seen, the Group is set using the `VSGI` command.

For physical SGIs, the GIC checks the sent Group against the Group configured on the receiver. Only if they match is the interrupt set to pending.

For virtual SGIs, there is no Group field in GITS_SGIR. The GIC will always use the Group configured for the receiver. Therefore, it is up to software, typically the hypervisor, to check the sent Group against that configured by the receiver.

Differences between vSGIs and pSGIs

In most respects virtual and physical SGIs behave the same as each other. There is one area where they are different - vSGIs use the same state machine as LPIs, as shown in the following diagram:

Figure 6-2: State machine used for LPIs and vSGIs

This reduces complexity for the GIC hardware. In most cases this difference would be invisible to software. To see a difference, software within the VM would have to use EOImode==1. This is where priority-drop and deactivation are performed using two separate register writes. With a pSGI, the same INTID could not be seen until after deactivation. With a vSGI, the same INTID could be seen after priority drop, before deactivation.

Querying SGI state

Sometimes a hypervisor needs to check whether a vSGI is pending. For example, to handle reads of the GICR_ISPENDR0 register by the VM.

To enable this, two new registers are added to the Redistributor to query the current pending state of the vSGIs for a vPE. Software does the following:

1. Writes the vPEID to GICR_VSGIR.
2. Polls GICR_VSGIPENDR.Busy until it reads 0, at which point GICR_VSGIPENDR.Pending reports the pending state of the vPE's SGIs.

There are no Redistributor registers to emulate writes to the GICR_ISPENDR0 and GICR_ICPENDR0 registers. These can be emulated using the ITS:

- Writing GICR_ICPENDR0 can be emulated using the `vSGI` command with `Clear==1`.
- Writing GICR_ISPENDR0 can be emulated by writing the GITS_SGIR register.

7. Example

There is a short example to accompany this guide. The example demonstrates initializing the GIC, configuring a vPE and vLPI using the ITS generating a vLPI.

The example is downloadable as a zip file.

The example requires Arm Development Studio. If you do not already have a copy, an evaluation copy is available from here.

The example includes a `ReadMe.txt` file which provides instructions for building and running the example.

Most of the important code for this example is in the following files:

- `gicv3_basic.c` contains functions for interacting with the GIC
- `gicv4_virt.c` contains functions specifically related to LPIs
- `main_vlpi.c` is a short test program that runs on PE 0.0.0.0, it demonstrates direct injection of vLPIs
- `secondary_virt.s` is the test program that runs on PE 0.0.0.1.

Looking at the file `main_vlpi.c`:

```
int main(void)
{
  uint32_t type, entry_size;
  uint32_t rd0, rd1, target_rd0, target_rd1;

  //
  // Configure the interrupt controller
  //
  rd0 = initGIC();

  // The example sends the vLPI to 0.0.0.1, so we also need its RD number
  rd1 = getRedistID(0x00000001);
```

The function `initGIC()` performs the basic initialization of the GIC, this is discussed in the Locality-Specific Peripheral Interrupts guide and the Arm CoreLink Generic Interrupt Controller v3 and v4 guide. For this example, most of the code runs on the physical PE `0.0.0.0`. But, the vPE is configured to run on the physical PE `0.0.0.1`. Therefore, we need to get a handle for the Redistributors for both PEs.

```
//
  // Set up Redistributor structures used for LPIs
  //

  setLPIConfigTableAddr(rd0, CONFIG_TABLE, GICV3_LPI_DEVICE_nGnRnE, 15);
  setLPIPendingTableAddr(rd0, PENDING0_TABLE, GICV3_LPI_DEVICE_nGnRnE, 15);
  enableLPIs(rd0);

  setLPIConfigTableAddr(rd1, CONFIG_TABLE, GICV3_LPI_DEVICE_nGnRnE, 15);
  setLPIPendingTableAddr(rd1, PENDING1_TABLE, GICV3_LPI_DEVICE_nGnRnE, 15);
  enableLPIs(rd1);
```

```
  setVPEConfTableAddr(rd0, VPE_TABLE, 0 /*attributes*/, 1 /*num_pages*/);
  setVPEConfTableAddr(rd1, VPE_TABLE, 0 /*attributes*/, 1 /*num_pages*/);
```

Next, the code installs the physical LPI configuration and pending tables on the two Redistributors. Remember that the LPI Configuration Table is shared by the Redistributors, while each Redistributor has its own LPI Pending table.

It also allocates a vPE Configuration Table, to record details of vPEs. Again, this table is shared between the Redistributors.

The example configures two interrupts:

```
  //
  // Configure virtual interrupt
  //
  configureVLPI((uint8_t*)(VCONFIG_TABLE), 8192, GICV3_LPI_ENABLE, 0);


  //
  // Configure physical doorbell interrupt
  //
  configureLPI(rd0, 8192, GICV3_LPI_ENABLE, 0);   // We'll use this as a Default
Doorbell
```

This code configures a virtual interrupt, for the vPE we are going to create. Then it configures a physical interrupt, for the vPE's default doorbell.

Next, the example configures the ITS. These steps are the same as for physical LPIs, which is described in a previous guide. Once the ITS is enabled, the example creates the mapping for the vPE and virtual interrupt:

```
  // Set up a mapping
  itsMAPD(0 /*DeviceID*/, ITT /*addr of ITT*/, 2 /*bit width of EventID*/);
  itsVMAPP(0 /*vpeid*/, target_rd0, VCONFIG_TABLE, VPENDING_TABLE,
           1 /*alloc*/, 1 /*v*/, 8192 /*default doorbell*/, 14 /*size*/);
  itsINVDB(0 /*vpeid*/);
  itsVMAPTI(0 /*DeviceID*/, 0 /*EventID*/, 1023 /*individual doorbell*/,
            0 /*vpeid*/, 8192 /*vINTID*/);
  itsVSYNC(0 /*vpeid*/);
```

The example code does the following:

- Maps DeviceID 0 to an Interrupt Translation Table.
- Creates vPE, with vPEID 0.
- Specifies the location of its LPI Configuration and Pending table and allocates physical INTID 8192 as the default doorbell. The vPE is mapped to physical core 0.0.0.0, which is the core running this part of the example.
- Invalidates any cached configuration for the default doorbell.
- Maps EventID0 from DeviceID 0 to a virtual interrupt targeting the new vPE.
- Synchronizes the changes.

At this point we have configured the GIC, and we can generate an interrupt to test the configuration:

```
itsINT(0 /*DeviceID*/, 0 /*EventID*/);

// Wait for interrupt
while(flag < 1)
{}
```

The `INT` command generates an interrupt, which we previously mapped to a virtual interrupt target vPE0. The example has not yet made vPE scheduled, so the interrupt triggers the default doorbell.

In response to seeing the default doorbell, vPEID 0 is made scheduled on the second core (0.0.0.1). This is permitted because 0.0.0.0 and 0.0.0.1 are part of the same `CommonLPIAff` group. At this point, the virtual interrupt is delivered to 0.0.0.1.

Running the example with debug output enabled produces the following output:

```
C:\Program Files\Arm\FVP_ARM_Std_Library\FVP_Base>FVP_Base_AEMvA-AEMvA.exe -f
      C:\SVN\GICv3.x_GICv4.x_example\gicv4_all.params --application=C:\mytest
\image_vlpi.axf
terminal_0: Listening for serial connection on port 5000
terminal_3: Listening for serial connection on port 5001
terminal_1: Listening for serial connection on port 5002
terminal_2: Listening for serial connection on port 5003
setLPIConfigTableAddr:: Installing LPI Configuration Table on RD0
setLPIConfigTableAddr:: Tabble base 0x80020000, with 15 ID bits
setLPIPendingTableAddr:: Installing LPI Pending Table on RD0
setLPIPendingTableAddr:: Tabble base 0x80030000, with 15 ID bits
enableLPIs:: Enabling physical LPIs on RD0
setLPIConfigTableAddr:: Installing LPI Configuration Table on RD1
setLPIConfigTableAddr:: Tabble base 0x80020000, with 15 ID bits
setLPIPendingTableAddr:: Installing LPI Pending Table on RD1
setLPIPendingTableAddr:: Tabble base 0x80040000, with 15 ID bits
enableLPIs:: Enabling physical LPIs on RD1
setVPEConfTableAddr:: Setting up vPE Configuration Table on RD0 at 0x80080000, with
 1 pages
setVPEConfTableAddr:: Setting up vPE Configuration Table on RD1 at 0x80080000, with
 1 pages
configureVLPI:: Configuring vINITD 8192 as priority 0x0 and enable=1
configureLPI:: Configuring INTID 8192, with priority 0x0 and enable 0x1, on RD0
initITSCommandQueue:: Setting up Command Queue at 0x80050000, with 1 pages
setITSTableAddr:: Setting up ITS table 0 at 0x80060000, with 16 pages
setITSTableAddr:: Setting up ITS table 1 at 0x80070000, with 16 pages
setITSTableAddr:: Setting up ITS table 2 at 0x80080000, with 1 pages
itsAddCommand:: Wrote command with:
DW0: 00 00 00 00 00 00 00 08
DW1: 00 00 00 00 00 00 00 01
DW2: 80 00 00 00 80 0b 00 00
DW0: 00 00 00 00 00 00 00 00
itsAddCommand:: Wrote command with:
DW0: 00 00 00 00 80 09 03 29
DW1: 00 00 00 00 00 00 20 00
DW2: 80 00 00 00 00 00 00 00
DW0: 00 00 00 00 80 0a 00 0e
itsAddCommand:: Wrote command with:
DW0: 00 00 00 00 00 00 00 2e
DW1: 00 00 00 00 00 00 00 00
DW2: 00 00 00 00 00 00 00 00
DW0: 00 00 00 00 00 00 00 00
itsAddCommand:: Wrote command with:
DW0: 00 00 00 00 00 00 00 2a
DW1: 00 00 00 00 00 00 00 00
```

```
DW2: 00 00 03 ff 00 00 20 00
DW0: 00 00 00 00 00 00 00 00
itsAddCommand:: Wrote command with:
DW0: 00 00 00 00 00 00 00 25
DW1: 00 00 00 00 00 00 00 00
DW2: 00 00 00 00 00 00 00 00
DW0: 00 00 00 00 00 00 00 00
main(): Sending vLPI 8192 to vPEID 0
itsAddCommand:: Wrote command with:
DW0: 00 00 00 00 00 00 00 0c
DW1: 00 00 00 00 00 00 00 00
DW2: 00 00 00 00 00 00 00 00
DW0: 00 00 00 00 00 00 00 00
itsAddCommand:: Wrote command with:
DW0: 00 00 00 00 00 00 00 03
DW1: 00 00 00 00 00 00 00 00
DW2: 00 00 00 00 00 00 00 00
DW0: 00 00 00 00 00 00 00 00
FIQ: Received INTID 1021
FIQ: Received Non-secure interrupt from the ITS
FIQ: Read INTID 8192 from IAR1
makeResident:: Making vPEID 0x0 resident on RD1
main(): Test end
Secondary Core in IRQ handler


Info: /OSCI/SystemC: Simulation stopped by user.
```

8. Check your knowledge

The following questions will help you test you knowledge:

In GICv3.x, how are virtual interrupts generated?

Using the List Registers. Software at EL2 (or above) writes the vINTID, priority, and group information into the register to create a new virtual interrupt.

Do virtual interrupts target a VM or vPE?

A vPE.

Write an instruction to read ICV_IAR1_EL1

`MRS Xn, ICC_IAR1_EL1`. Software always uses the ICC register names in MRS and MSR instructions. The access is redirected to the equivalent ICV register if the corresponding HCR_EL2.xMO bit is set.

In GICv3.1, if Secure Virtualization is implemented and enabled, how many GIC maintenance interrupts are there?

There is always a single GIC maintenance interrupt, shared by both Security states.

In GICv4.1, name the data structures in memory used by a Redistributor to handle direct injection.

The vPE Configuration Table records all the created vPEs. Each vPE has its own virtual LPI Pending table. There is also the virtual LPI Configuration table, typically one of these per VM.

In GICv4.1, what is a default doorbell?

Each vPE can be assigned a default doorbell. This is a physical LPI which is set pending if an interrupt arrives for the vPE while it is not scheduled.