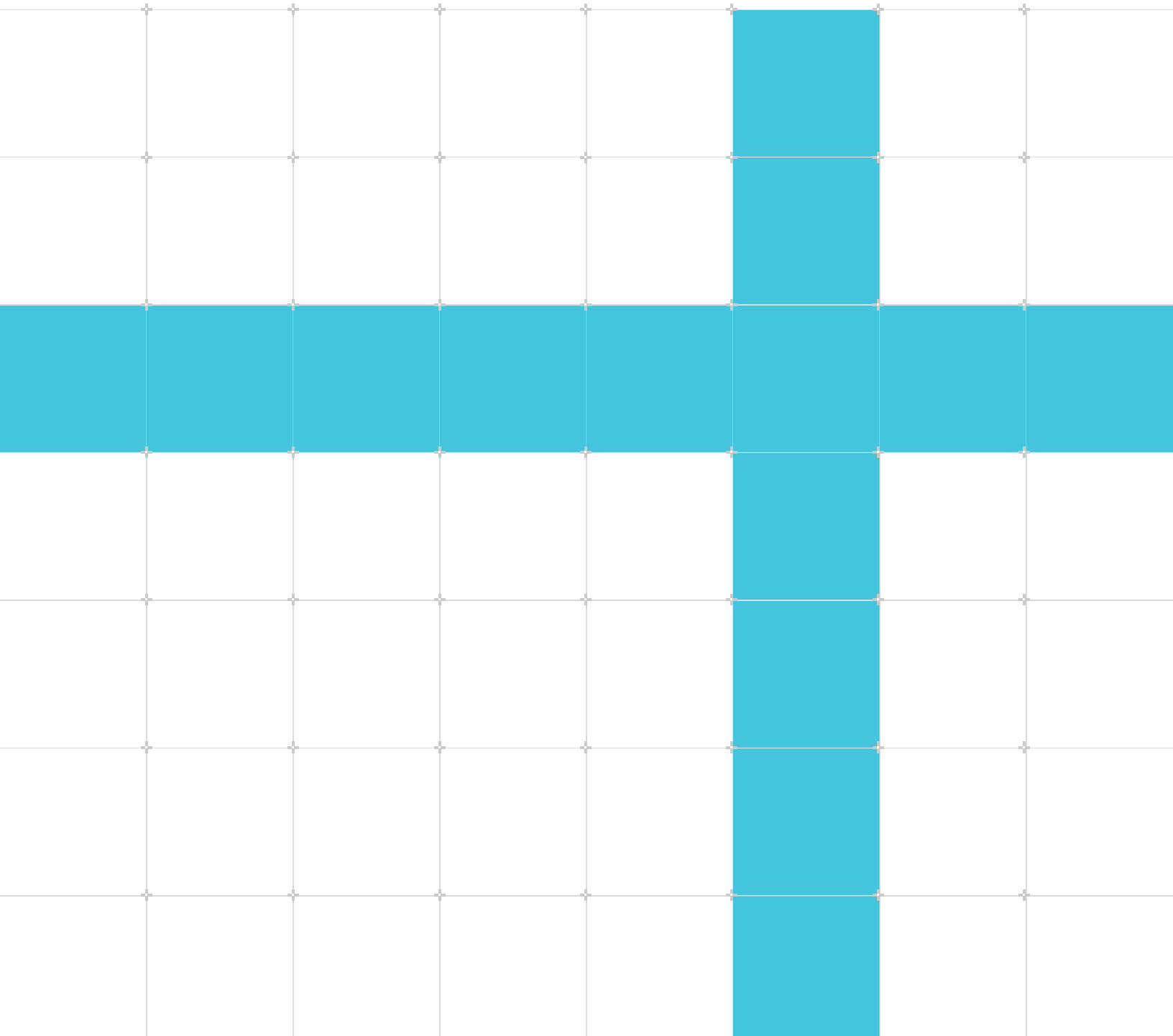
Learn the architecture - Generic Interrupt Controller v3 and v4, LPIs

Version 1.0

Issue 01
102923_0100_01_en



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

Release information

Document history

Issue	Date	Confidentiality	Change
0100-01	19 May 2022	Non-Confidential	First release

Proprietary Notice

or indirectly, in violation of such export laws. Use of the word “partner” in reference to Arm’s customers is not intended to create or refer to any partnership relationship with any other company. Arm may make changes to this document at any time and without notice.

This document may be translated into other languages for convenience, and you agree that if there is any conflict between the English version of this document and any translation, the terms of the English version of the Agreement shall prevail.

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

This guide introduces Locality-specific Peripheral Interrupts (LPIs), a type of interrupt introduced in GICv3/v4.

An interrupt is a signal to the processor that an event has occurred which needs to be dealt with. Interrupts are typically generated by peripherals. LPIs are typically used for peripherals that produce Message-Signaled Interrupts (MSIs).

The configuration and management of LPIs is different to the other interrupt types because their state is held in memory rather than registers. LPIs are Message-Signaled Interrupts (MSIs), with translation provided by an Interrupt Translation Service (ITS).

This guide is aimed at anyone who needs an understanding of how MSIs are translated by the GIC and how the resulting LPIs are managed. In particular, the guide is aimed at anyone who needs to configure the GIC in a bare-metal environment.

This guide complements the Arm Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and 4.0. It is not a replacement or alternative. Refer to the Arm Generic Interrupt Controller Architecture Specification GIC architecture version 3.0 and 4.0 for detailed descriptions of registers and behaviors.

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

- Name and describe the function of the memory structures used by Redistributors to handle LPIs.
- Explain how an ITS translates an incoming MSI into an interrupt.
- Write bare-metal code to enable LPI handling in a GIC.

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 Learn the architecture: Arm Generic Interrupt Controller v3 and v4.

2. LPIs

The configuration of LPIs is very different to the other interrupt types, and involves the following:

- Redistributors
- ITSs (Interrupt Translation Service)

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

Support for LPIs is optional and is indicated by GICD_TYPER.LPIS.

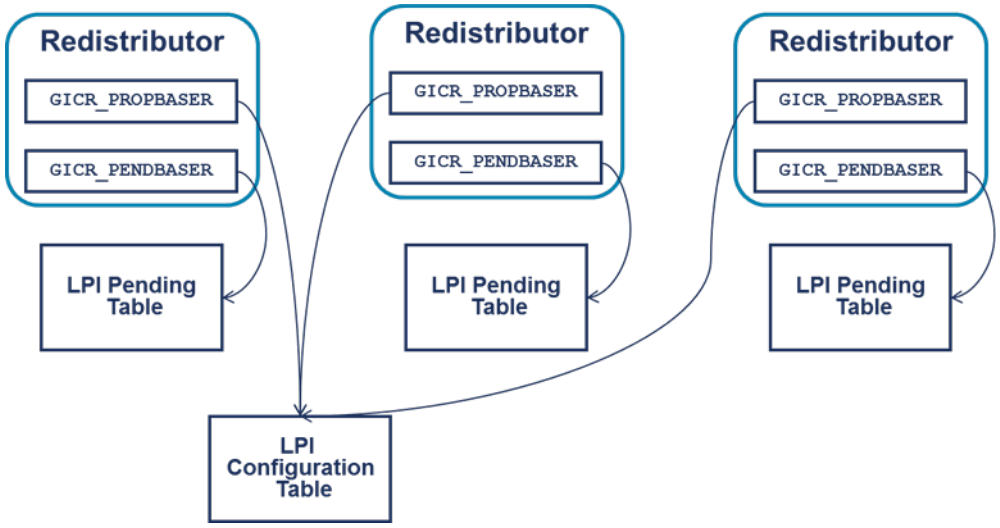
3. Redistributors

The Redistributors use tables held in memory for both LPI configuration information and the state of each physical LPI. Configuration information for LPIs is stored in the LPI Configuration table, which is pointed to by GICR_PROPBASER. LPI configuration is global, that is, all Redistributors must see the same configuration. Typically, a system has a single LPI Configuration table that is shared by all Redistributors.

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

The diagram below shows three Redistributors and the associated LPI tables:

Figure 3-1: LPI Configuration and LPI Pending tables



3.1 Initial configuration of a Redistributor

The steps to initialize the Redistributors in a system are as follows:

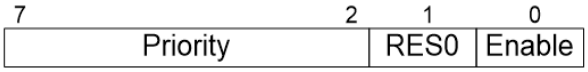
1. Allocate memory for the LPI Configuration table and initialize the table with the appropriate configurations for each LPI.
2. Set GICR_PROPBASER in each Redistributor to point to the LPI Configuration table.
3. Allocate memory for the LPI Pending table of each Redistributor and initialize the content of each table. At system start-up, this typically means zeroing the memory, meaning that all LPI INTIDs are in the inactive state.
4. Set GICR_PENDBASER in each Redistributor to point to the LPI Pending table associated with that Redistributor.

5. Set GICR_CTLR.EnableLPIs to 1 in each Redistributor to enable LPIs.

LPI Configuration table

The LPI Configuration table contains one byte for each LPI INTID. The following diagram shows the format of each entry:

Figure 3-2: Format of an entry in the LPI Configuration table



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

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

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

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

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

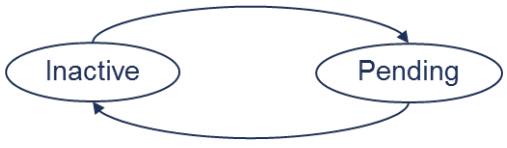
Size in bytes = $2^{\text{GICR\_PROPBASER.IDbits+1}}$ – 8192

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

LPI Pending tables

State information for LPIs is stored in memory. LPIs have two states: inactive or pending.

Figure 3-3: State machine for LPIs



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

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

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

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

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

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

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

While LPIs are enabled in the owning Redistributor, software is never expected to directly access the LPI Pending table.

3.2 Reconfiguring LPIs

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

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

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

If an ITS is not implemented, software must write to the GICR_INVLPIR or GICR_INVALLR registers instead to cause the Redistributors to reload the interrupt configuration.

4. ITS

A peripheral generates an LPI by writing a message to GITS_TRANSLATER in the ITS. This provides the ITS with the following information:

EventID

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

DeviceID

The DeviceID identifies the peripheral. The mechanism by which a DeviceID is generated is **IMPLEMENTATION DEFINED**.

The ITS handles the translation of the message into an INTID that can be delivered to a connected core.

Physical LPIs are grouped together in collections. All LPIs in a collection are routed to the same Redistributor. Software allocates LPIs to collections, allowing it to efficiently move interrupts from one processing element to another.

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

Device table

There is one Device table per ITS. Device tables map DeviceIDs to Interrupt Translation Tables.

Interrupt Translation Tables

There is one Interrupt Translation Table (ITT) per DeviceID or peripheral. The ITT contains the peripheral-specific mappings between EventID and INTID. They also contain the collection which the INTID is a member of.

Collection table

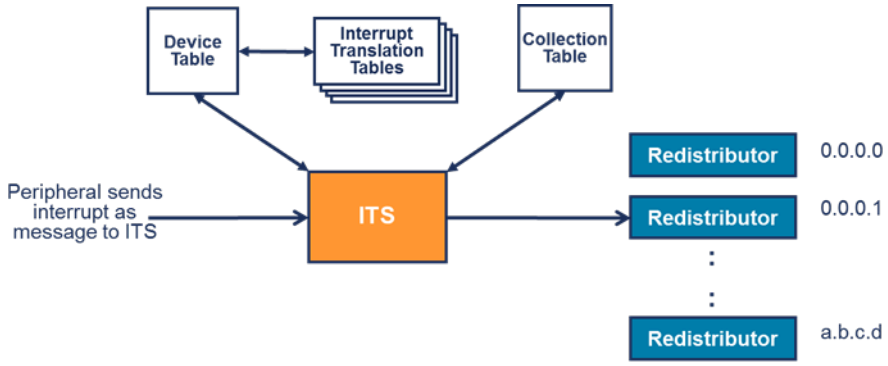
There is one Collection table per ITS. The Collection table maps collections to Redistributors.

When a peripheral writes a message to GITS_TRANSLATER, the ITS does the following:

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

The following diagram illustrates this process:

Figure 4-1: An ITS forwarding an LPI to a Redistributor



Note

An ITS can optionally support a number of hardware collections. Hardware collections are where the ITS holds the configuration internally, rather than storing it in memory. GITS_TYPER.HCC reports the number of hardware collections that are supported. You can still think of this as being part of the Collection table, it is just not stored in memory.

4.1 The command queue

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

GITS_CBASER

This register specifies the base address and size of the command queue. The command queue must be 64KB aligned and the size must be a multiple of 4KB. Each entry in the command queue is 32 bytes. GITS_CBASER also specifies the cacheability and shareability settings that the ITS uses when accessing the command queue.

GITS_CREADR

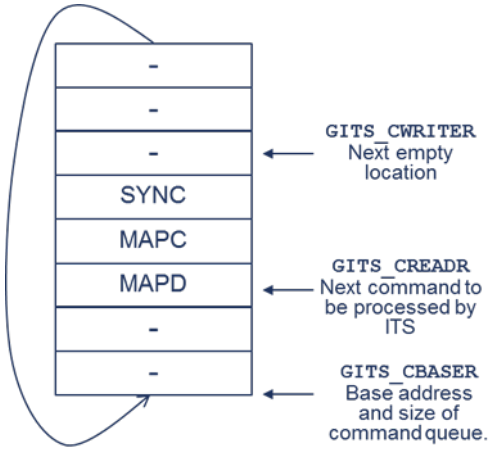
This register points to the next command that the ITS will process.

GITS_CWRITER

This register points to the entry in the queue where the next new command should be written.

The following diagram shows a simplified representation of a command queue:

Figure 4-2: ITS circular command queue



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

4.2 Initial configuration of an ITS

To configure an ITS at system startup, software must:

1. Allocate memory for the Device and Collection tables.

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

   Software must allocate the memory for the command queue and set GITS_CBASER and GITS_CWRITER to point to the start of this allocated memory.
3. Enable the ITS.

   When the tables and command queue have been allocated, the ITS can be enabled. This is done by setting the GITS_CTLR.Enable bit to 1.

   Once GITS_CTLR.Enable has been set, the GITS_BASERn and GITS_CBASER registers become read-only.

4.3 The sizes and layout of Collection and Device tables

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

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

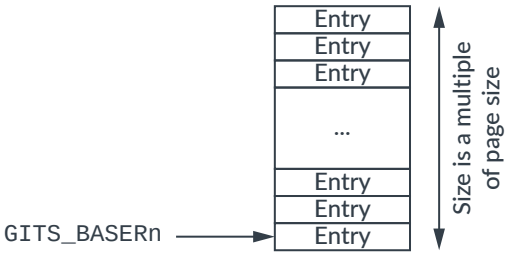
Support for two-level tables is optional. If the ITS only supports flat tables, GITS_BASERn.Indirect is **RAZ/WI**.

Flat level tables

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

The following diagram shows a flat table. The table is a single contiguous block of memory, with a GITS_BASERn register pointing to the base address of the table:

Figure 4-3: A flat Device or Collection table



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

Size in bytes = 2^{ID_width} * entry_size

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

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

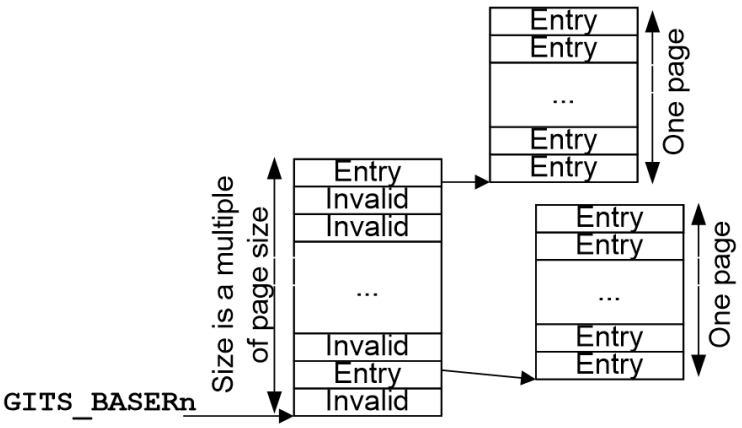
For example, if a system implements an 8-bit DeviceID, the bytes per table entry is 8 and a 4K page size is used:

28 * 8 = 2048 bytes = 4K (rounded up to the next full page)

Two-level tables

With two-level tables, software allocates a single first level table, and several second level tables as shown in the following diagram:

Figure 4-4: A two-level Device or Collection table



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

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

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

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

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

Size in bytes = $(2^{ID\_width}$ / (page_size / entry_size)) * 8

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

4.4 Adding a new command to the command queue

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

1. Write the new command to the queue.

   GITS_CWRITER points to the next entry that does not contain a valid command in the command queue. Software must write the command to this entry, and it must ensure global visibility.
2. Update GITS_CWRITER.

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

   Software can add multiple commands to the queue at the same time, provided there are enough empty spaces in the command queue and that GITS_CWRITER is updated accordingly.
3. Wait for the command to be read by the ITS.

   Software can check whether the command has been read by the ITS by polling GITS_CREADR. All commands have been read by the ITS when GITS_CWRITER.Offset is equal to GITS_CREADR.Offset.

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

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


Note

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

4.5 Mapping an interrupt to a Redistributor

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

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

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

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

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

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

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

Collections are mapped to a Redistributor using the `MAPC` command:

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

Identification of the target Redistributor depends on GITS_TYPER.PTA:

- GITS_TYPER.PTA==0

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

  The Redistributor is specified by its physical address.

Example

A timer has DeviceID 5 and uses a 2-bit EventID. We want EventID 0 to be mapped to INTID 8725. The ITT allocated for the timer is at address `0x84500000`.

We decide to use collection number 3 and deliver the interrupt to the Redistributor with ID 7.

The command sequence for this is as follows:

```
MAPD 5, 0x84500000, 2   // Map DeviceID 5 to an ITT
MAPTI 5, 0, 8725, 3     // Map EventID 0 to INTID 8725 and collection 3
MAPC 3, 7               // Map collection 3 to Redistributor 7
SYNC 7
```

The example assumes that none of the mappings have previously been set up, and that GITS_TYPER.PTA==0.

4.6 Migrating interrupts between Redistributors

You can use several different techniques to move an interrupt from one Redistributor to another:

- Remap a collection.

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

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

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

  If there were multiple collections targeting `RDADDR1`, then we would need multiple `MAPC` commands, one for each collection. This sequence assumes that all the collections are being remapped to the same new target Redistributor.
- Map an interrupt to a different collection. Individual interrupts can be remapped to a different collection. This can be done using the following command sequence:

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

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

4.7 Removing interrupt mappings

To remap or remove the mapping of an interrupt, software must do the following:

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

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

4.8 Remapping or removing the mapping of devices

To change or remove the mapping for devices software must do the following:

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

5. Example

Download a short example to accompany this guide here. The example demonstrates initializing the GIC, configuring an LPI using the ITS, and handling a generated LPI interrupt.

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

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

The example contains the following files:

- `gicv3_basic.c` contains functions for interacting with the GIC
- `gicv3_lpis.c` contains functions specifically related to LPIs
- `main_lpi.c` is a short test program which uses LPIs

In this example, we look at the file `main_lpi.c` as follows:

```
int main(void)
{
  uint32_t type, entry_size;
  uint32_t rd, target_rd;

  //
  // Configure the interrupt controller
  //
  rd = initGIC();
```

The function `initGIC()` performs the basic initialization of the GIC. For more information, see Learn the architecture: Arm Generic Interrupt Controller v3 and v4 guide.

The following code allocates the memory for the LPI Configuration and Pending Tables:

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

  setLPIConfigTableAddr(rd, CONFIG_TABLE, GICV3_LPI_DEVICE_nGnRnE /*Attributes*/,
                        15 /* Number of ID bits */);
  setLPIPendingTableAddr(rd, PENDING_TABLE, GICV3_LPI_DEVICE_nGnRnE /*Attributes*/,
                         15 /* Number of ID bits */);
  enableLPIs(rd);
```

The Configuration Table is shared by multiple Redistributors but there is a single Pending Table for each Redistributor. This example uses a single core, so it only allocates a single LPI Pending table.

Next, the code configures the ITS:

```
  // Allocate memory for the ITS command queue
  initITSCommandQueue(CMD_QUEUE, GICV3_ITS_CQUEUE_VALID /*Attributes*/,
                        1 /*num_pages*/);
```

```
// Allocate Device table
setITSTableAddr(0 /*index*/,
                DEVICE_TABLE /* addr */,
                (GICV3_ITS_TABLE_PAGE_VALID | GICV3_ITS_TABLE_PAGE_DIRECT |
                 GICV3_ITS_TABLE_PAGE_DEVICE_nGnRnE),
                GICV3_ITS_TABLE_PAGE_SIZE_4K,
                16 /*num_pages*/);

//Allocate Collection table
setITSTableAddr(1 /*index*/,
                COLLECTION_TABLE /* addr */,
                (GICV3_ITS_TABLE_PAGE_VALID | GICV3_ITS_TABLE_PAGE_DIRECT |
                 GICV3_ITS_TABLE_PAGE_DEVICE_nGnRnE),
                GICV3_ITS_TABLE_PAGE_SIZE_4K,
                16 /*num_pages*/);

// Enable the ITS
enableITS();
```

This code allocates memory for the ITS's Command Queue, Device table, and Collection table. Once the tables are initialized, the ITS is enabled.

With the ITS enabled, an interrupt can be mapped by adding commands to the command queue, as shown in the following code:

```
// Set up a mapping
itsMAPD(0 /*DeviceID*/, ITT /*addr of ITT*/, 2 /*bit width of ID*/);
itsMAPTI(0 /*DeviceID*/, 0 /*EventID*/, 8193 /*intid*/, 0 /*collection*/);
itsMAPC(target_rd /* target Redistributor*/, 0 /*collection*/);
itsSYNC(target_rd /* target Redistributor*/);
```

This example does the following:

- Maps DeviceID 0 to an Interrupt Translation Table
- Maps EventID 0 from DeviceID 0 to the INTID 8193 and allocates it to Collection 0
- Maps Collection 0 to the current Redistributor (0.0.0.0)

At this point, we have the GIC enabled with an ITS mapping for a DeviceID/EventID. Next, we configure INTID 8193 as follows:

```
configureLPI(rd, 8193 /*INTID*/, GICV3_LPI_ENABLE, 0 /*Priority*/);
printf("main(): Sending LPI 8193\n");
itsINV(0 /*DeviceID*/, 0 /*EventID*/);
itsINT(0 /*DeviceID*/, 0 /*EventID*/);
```

This code enables INTID 8193 and sets its priority. It is possible the GIC has already cached the old configuration, so an `INV` command is needed to ensure the new configuration is used.

The FVP Base Platform model does not include a peripheral capable of generating this MSI, so instead we generate it using an `INT` command.

6. Check your knowledge

The following questions and answers let you check your understanding of this guide.

How is the state-machine for LPIs different to the other interrupt types?

The other interrupt types have four states: Inactive, Pending, Active, and Active and Pending. LPIs only have two states: Inactive and Pending.

Where is the configuration and state for LPIs stored?

In memory. The Redistributors share a common LPI Configuration table which stores the configuration. There is one LPI Pending table per Redistributor which stores pending state.

What information does an MSI contain which allows an ITS to translate it?

A DeviceID and an EventID. The DeviceID identifies which peripheral sent the interrupt. The EventID identifies which interrupt the peripheral is sending.

How does software create mappings in the ITS?

By issuing commands using the ITS command queue. A `MAPD` command creates a mapping for the device. `MAPI` and `MAPTI` commands map the EventIDs of a peripheral.

How does software change the priority of an LPI?

The software must do the following:

1. Update the appropriate entry in the LPI Configuration Table
2. Ensure global visibility of the change
3. Invalidate any configuration caching in the GIC by issuing an invalidate operation using the ITS or writing to one of the invalidation registers

7. Related information

Here are some resources related to material in this guide:

- Learn the architecture: Arm Generic Interrupt Controller v3 and v4
- GIC specifications
- GIC home page on developer.arm.com
- GIC Stream Protocol interface version B
- Arm Community
- Learn the architecture: Exception model guide

8. Next steps

This guide introduced the LPI interrupt type in GICv3/4. It explained how an ITS receives LPIs from peripherals and forwards them to the appropriate Redistributor, and provided information about the memory structures used by Redistributors to handle LPIs.

GICv4 allows for hardware managed virtualization LPIs, for more information see GIC Support for Virtualization.