Two important types are LPI (Locality-specific Peripheral Interrupts) and SPI (Shared Peripheral Interrupts).

1. LPI (Locality-specific Peripheral Interrupt)

* LPIs are designed for power efficiency and high-performance multi-core systems in large systems.
* They are message-based interrupts, meaning they do not require dedicated hardware lines.
* LPIs are handled by the GIC (Generic Interrupt Controller) and use an interrupt translation service (ITS) to map them. They are not stored in GIC distributor registers, reducing power consumption when not in use.

2. SPI (Shared Peripheral Interrupt)

* SPIs are used for peripherals that are shared across multiple CPU cores. They are managed by the GIC distributor and can be routed to one or more cores.
* SPIs are numbered from 32 onwards in the interrupt ID space. Used for devices like timers, GPUs, and communication peripherals.

Apart from LPI and SPI, the Arm architecture defines several other interrupt types:

1. PPI (Private Peripheral Interrupts)
   * Used for per-core peripherals (e.g., private timers). Interrupt IDs 16–31. Routed only to the core where they originate.
2. SGI (Software Generated Interrupts)
   * Used for inter-core communication. Interrupt IDs 0–15. Can be triggered by software.
3. FIQ (Fast Interrupt Requests)
   * Higher priority than normal IRQs. Used for time-critical applications. Available in Armv7-A and earlier architectures.
4. IRQ (Interrupt Request)
   * General interrupts for devices. Lower priority than FIQ. Used in Armv7 and older architectures.

In modern Armv8-A and Armv9, the primary interrupt controller is GICv3/GICv4, which primarily handles SPI, PPI, LPI, and SGI.

GIC Architecture Overview

The GIC consists of the following components:

1. Distributor (GICD)
   * Manages global interrupts (SPI, LPI, PPI). Routes interrupts to appropriate CPU interfaces. Controls enable/disable and priority settings.
2. Redistributor (GICR) [Introduced in GICv3]
   * Manages per-core interrupts (PPI, LPI). Each core has a separate redistributor. Essential for handling power-efficient LPIs.
3. CPU Interface (GICC for GICv2, ICC for GICv3/4)
   * Delivers the interrupt to the CPU. Supports priority-based interrupt handling. Uses registers to acknowledge and complete interrupts.
4. Interrupt Translation Service (ITS) [For LPI]
   * Required for handling LPI translation and mapping. Stores interrupt mappings in memory instead of registers.

Interrupt Types and Handling in GIC

|  |  |  |  |
| --- | --- | --- | --- |
| Interrupt Type | Handled by | Routing Mechanism | Use Case |
| SGI (Software Generated Interrupts) | Redistributor (GICR) | Sent by software to specific cores | Inter-core communication |
| PPI (Private Peripheral Interrupts) | Redistributor (GICR) | Always local to the core | Timers, watchdogs |
| SPI (Shared Peripheral Interrupts) | Distributor (GICD) | Routed to multiple cores | Shared peripherals like GPU, PCIe |
| LPI (Locality-specific Peripheral Interrupts) | Redistributor (GICR) + ITS | Message-based (memory-mapped) | Power-efficient interrupts |
| FIQ (Fast Interrupt Request) [Legacy] | CPU Interface | High-priority, non-maskable | Time-critical tasks |

|  |  |  |  |
| --- | --- | --- | --- |
| Feature | GICv2 | GICv3 | GICv4 |
| Interrupt Addressing | 8-bit | 10-bit (1020 interrupts) | 10-bit |
| Register-based LPIs | ❌ No support | ✅ Supported via GICR | ✅ Supported |
| Message-based LPIs | ❌ No | ✅ Yes (via ITS) | ✅ Yes |
| Redistributors | ❌ No | ✅ Yes | ✅ Yes |
| Multicore Affinity | Limited | ✅ Advanced | ✅ Advanced |
| Virtual Interrupts (for hypervisors) | Limited | ✅ Yes | ✅ Yes |

Interrupt Handling Flow in GICv3

1. Interrupt Generation
   * A device generates an SPI, PPI, or LPI. An SGI may be triggered via software.
2. Interrupt Routing
   * SPI: Routed through GIC Distributor to a core.
   * PPI: Always local to a specific CPU core.
   * LPI: Uses ITS to find the target CPU and route accordingly.
3. Interrupt Acknowledgment
   * The CPU interface (ICC) receives the interrupt. The core reads the Interrupt Acknowledge Register (IAR) to get the interrupt ID.
4. Interrupt Handling
   * The CPU executes the ISR (Interrupt Service Routine).
5. Completion Notification
   * After servicing, the CPU writes to the End of Interrupt Register (EOIR) to clear it.

Why is LPI More Efficient?

* Does not use GICD registers, reducing power consumption. Stored in memory, allowing better scalability for large systems.
* ITS manages mappings dynamically, making it flexible. Better suited for PCIe MSI (Message Signaled Interrupts).

Conclusion

* SPIs are traditional hardware interrupts, routed via GICD.
* LPIs are memory-mapped, power-efficient interrupts for large systems.
* GICv3 and GICv4 improve scalability, multi-core efficiency, and virtualization support.

Steps Involved in Interrupt Handling

1. Initialize GIC
2. Configure the Interrupt in GICD (Distributor)
3. Enable the Interrupt in GICR (Redistributor)
4. Enable Interrupt Handling in CPU Interface (ICC)
5. Write an Interrupt Service Routine (ISR)
6. Acknowledge and Complete the Interrupt

1. GIC Initialization

#define GICD\_BASE 0x08000000 // Example GIC Distributor base address

#define GICR\_BASE 0x080A0000 // Example Redistributor base address

#define ICC\_BASE 0x080B0000 // Example CPU Interface base address

void gic\_init() {

// Enable the Distributor

\*(volatile uint32\_t \*)(GICD\_BASE + 0x000) = 0x1; // GICD\_CTLR: Enable GIC

}

Enables the GIC Distributor (GICD). The GIC Distributor manages global interrupt routing.

2. Configure SPI in GIC Distributor

#define IRQ\_ID 50 // Example SPI interrupt ID

void gic\_configure\_interrupt(uint32\_t irq\_id) {

// Set interrupt priority (lower value = higher priority)

\*(volatile uint32\_t \*)(GICD\_BASE + 0x400 + (irq\_id \* 4 / 4)) = 0x80;

// Set interrupt as level-triggered and edge-sensitive (Interrupt Config Register)

uint32\_t reg\_offset = (irq\_id / 16) \* 4;

uint32\_t bit\_shift = (irq\_id % 16) \* 2;

uint32\_t reg\_val = \*(volatile uint32\_t \*)(GICD\_BASE + 0xC00 + reg\_offset);

reg\_val &= ~(0x3 << bit\_shift); // Clear existing bits

reg\_val |= (0x2 << bit\_shift); // Set as level-triggered

\*(volatile uint32\_t \*)(GICD\_BASE + 0xC00 + reg\_offset) = reg\_val;

// Enable the interrupt

\*(volatile uint32\_t \*)(GICD\_BASE + 0x100 + (irq\_id / 32) \* 4) = (1 << (irq\_id % 32));

}

Sets interrupt priority (higher values mean lower priority). Configures trigger mode (level-sensitive or edge-sensitive). Enables the interrupt in the GIC Distributor.

3. Enable Interrupt in Redistributor

void gic\_enable\_redistributor(uint32\_t irq\_id) {

\*(volatile uint32\_t \*)(GICR\_BASE + 0x100 + (irq\_id / 32) \* 4) = (1 << (irq\_id % 32));

}

* GICR manages LPIs and PPIs per CPU core.
* SPI is global, so it's managed in GICD, but some interrupts may need Redistributor settings.

4. Enable CPU Interface for Interrupt Handling

void gic\_enable\_cpu\_interface() {

\*(volatile uint32\_t \*)(ICC\_BASE + 0x000) = 1; // Enable CPU Interface (ICC\_CTLR)

\*(volatile uint32\_t \*)(ICC\_BASE + 0x004) = 0xFF; // Set priority mask (ICC\_PMR)

\*(volatile uint32\_t \*)(ICC\_BASE + 0x010) = 1; // Enable interrupt reception (ICC\_IAR)

}

* Enables CPU interface to receive interrupts.
* Sets priority masking (so it does not filter out high-priority interrupts).

5. Interrupt Service Routine (ISR)

void irq\_handler() {

// Read interrupt ID from the Interrupt Acknowledge Register

uint32\_t irq\_id = \*(volatile uint32\_t \*)(ICC\_BASE + 0x00C);

if (irq\_id == IRQ\_ID) {

// Handle the interrupt (e.g., clear peripheral interrupt flag)

printf("Handling interrupt %d\n", irq\_id);

}

// Write to End of Interrupt Register to complete the interrupt

\*(volatile uint32\_t \*)(ICC\_BASE + 0x010) = irq\_id;

}

* Reads IRQ ID from the Interrupt Acknowledge Register (ICC\_IAR).
* Handles the interrupt (e.g., clears the hardware flag).
* Writes to End of Interrupt Register (ICC\_EOIR) to complete processing.

6. Acknowledge and Complete the Interrupt

The GIC expects software to acknowledge and mark the interrupt as completed.

* ICC\_IAR (Interrupt Acknowledge Register) tells the CPU which interrupt to handle.
* ICC\_EOIR (End of Interrupt Register) informs GIC that the interrupt is serviced.

Summary of Steps

|  |  |  |
| --- | --- | --- |
| Step | Component | Register Used |
| Enable GIC | Distributor (GICD) | GICD\_CTLR |
| Configure SPI | Distributor (GICD) | GICD\_IPRIORITYR, GICD\_ICFGR |
| Enable Interrupt | Distributor (GICD) | GICD\_ISENABLER |
| Enable CPU Interface | CPU Interface (ICC) | ICC\_CTLR, ICC\_PMR |
| Read IRQ ID | CPU Interface (ICC) | ICC\_IAR |
| Handle Interrupt | Software ISR | - |
| Complete Interrupt | CPU Interface (ICC) | ICC\_EOIR |

Conclusion

* GIC handles different types of interrupts (SGI, PPI, SPI, LPI) efficiently.
* SPIs are routed through the Distributor, PPIs are per-core, LPIs are memory-mapped.
* ITS (Interrupt Translation Service) is needed for LPIs.
* Software must acknowledge and complete interrupts using GIC registers.

In this example, we'll configure and handle LPIs (Locality-specific Peripheral Interrupts) and multi-core SPI routing using GICv3.

1. Understanding LPI Handling in GICv3

Unlike SPIs, LPIs are:

* Message-based (no dedicated hardware IRQ line).
* Stored in memory (ITS - Interrupt Translation Service), not GICD registers.
* Used for scalable, power-efficient systems (e.g., PCIe MSI-X).

GICv3 uses Redistributors (GICR) and ITS to manage LPIs.

* Each core has a Redistributor (GICR).
* The Interrupt Translation Service (ITS) maps device interrupts to CPUs.

2. Configuring an LPI with ITS

Steps to set up an LPI:

1. Enable Redistributor (GICR) for each core
2. Allocate LPI configuration table in memory
3. Configure ITS to translate device interrupts to LPIs
4. Enable LPIs in GIC

Step 1: Enable Redistributor for Each CPU

#define GICR\_WAKER 0x14 // WAKER register offset

void enable\_redistributor(uint32\_t gicr\_base) {

// Wake up the Redistributor (GICR\_WAKER)

\*(volatile uint32\_t \*)(gicr\_base + GICR\_WAKER) &= ~(1 << 1);

while (\*(volatile uint32\_t \*)(gicr\_base + GICR\_WAKER) & (1 << 2));

}

* Each CPU core has its Redistributor (GICR).
* WAKER register brings Redistributor out of sleep mode.

Step 2: Allocate LPI Configuration Table in Memory

#define LPI\_CONFIG\_TABLE\_BASE 0x40000000 // Example base address

void configure\_lpi\_table() {

uint32\_t \*lpi\_table = (uint32\_t \*)LPI\_CONFIG\_TABLE\_BASE;

// Configure LPI priority (must be >= 0x80)

for (int i = 0; i < 1024; i++) {

lpi\_table[i] = 0x80; // Priority = 128 (lower = higher priority)

}

}

* LPIs do not use GICD priority registers.
* The priority is stored in a memory-mapped table.

Step 3: Configure ITS for Device Interrupts

#define ITS\_BASE 0x08100000 // Example ITS base address

#define DEVICE\_ID 0x01 // Example PCIe device ID

#define EVENT\_ID 45 // Example MSI event ID

#define LPI\_ID 200 // Example mapped LPI ID

void configure\_its() {

// Map Device Event (MSI) to an LPI

\*(volatile uint32\_t \*)(ITS\_BASE + 0x000) = DEVICE\_ID; // Target Device

\*(volatile uint32\_t \*)(ITS\_BASE + 0x100) = EVENT\_ID; // Event ID

\*(volatile uint32\_t \*)(ITS\_BASE + 0x200) = LPI\_ID; // Mapped LPI ID

}

* ITS translates PCIe MSI events to LPIs.
* Each event triggers a specific LPI.

Step 4: Enable LPIs in GIC

#define GICR\_PROPBASER 0x70 // Redistributor Property Base Register

#define GICR\_PENDBASER 0x78 // Pending Table Base Register

void enable\_lpi(uint32\_t gicr\_base) {

// Set LPI Configuration Table

\*(volatile uint64\_t \*)(gicr\_base + GICR\_PROPBASER) = LPI\_CONFIG\_TABLE\_BASE | 0x1;

// Set LPI Pending Table (for edge-triggered interrupts)

\*(volatile uint64\_t \*)(gicr\_base + GICR\_PENDBASER) = (LPI\_CONFIG\_TABLE\_BASE + 0x1000) | 0x1;

}

* PROPBASER stores LPI priority & properties.
* PENDBASER tracks pending LPIs (like a status register).

3. Multi-Core SPI Routing

Distribute an SPI Across Multiple CPUs

#define SPI\_ID 50 // Example SPI interrupt ID

void configure\_spi\_multicore() {

// Set SPI target CPUs (Route to CPU0 & CPU1)

uint32\_t target\_mask = (1 << 0) | (1 << 1); // Target CPU0 & CPU1

\*(volatile uint32\_t \*)(GICD\_BASE + 0x800 + (SPI\_ID / 4)) = target\_mask;

}

* This routes SPI ID 50 to CPU0 and CPU1.
* Useful for load balancing interrupts.

4. Handling an LPI in an ISR

void irq\_handler() {

uint32\_t irq\_id = \*(volatile uint32\_t \*)(ICC\_BASE + 0x00C); // Read interrupt ID

if (irq\_id >= 8192) { // LPI IDs start from 8192

printf("Handling LPI %d\n", irq\_id);

}

// End of interrupt (EOIR)

\*(volatile uint32\_t \*)(ICC\_BASE + 0x010) = irq\_id;

}

* LPIs start from interrupt ID 8192.
* EOIR must be written to signal completion.

Summary

|  |  |  |
| --- | --- | --- |
| Step | Action | Component |
| Enable Redistributor | Wake up each CPU's Redistributor | GICR |
| Set LPI Configuration | Define priorities in memory | PROPBASER |
| Configure ITS | Map device events to LPIs | ITS |
| Enable LPIs | Activate LPIs in Redistributor | GICR |
| Route SPI to Multiple CPUs | Target multiple CPUs | GICD |
| Handle LPI in ISR | Read LPI ID and clear | CPU Interface |

Key Takeaways

* LPIs are efficient because they don’t consume GICD register space.
* ITS dynamically translates PCIe MSI to LPIs.
* SPIs can be routed to multiple cores for load balancing.
* Redistributors (GICR) are per-core and manage LPIs & PPIs.

Interrupt addressing bits refer to the number of bits used to identify and address interrupts in an interrupt controller, such as the Generic Interrupt Controller (GIC) in ARM architectures.

Interrupt ID Width in ARM GIC

The number of addressing bits determines how many unique interrupt sources a system can handle.

1. GICv2 (ARMv7-A & Early ARMv8-A)

* Supports up to 1020 interrupts.
* Interrupt IDs range:
  + 0 - 15 → Software Generated Interrupts (SGIs)
  + 16 - 31 → Private Peripheral Interrupts (PPIs)
  + 32 - 1019 → Shared Peripheral Interrupts (SPIs)
  + 1020 - 1023 → Reserved

Thus, Interrupt ID is represented using 10 bits (since 1024 = 2¹⁰).

2. GICv3 & GICv4 (ARMv8-A)

* Supports up to 1 million (2²⁰) interrupts using LPIs (Locality-specific Peripheral Interrupts).
* Interrupt IDs range:
  + 0 - 15 → SGIs (10-bit encoding)
  + 16 - 31 → PPIs (10-bit encoding)
  + 32 - 1019 → SPIs (10-bit encoding)
  + 8192 - 2²⁰ (1M) → LPIs (20-bit encoding)

For LPIs, 20-bit addressing is used, allowing for a much larger number of interrupts.

Summary: Addressing Bit Requirements

|  |  |  |  |
| --- | --- | --- | --- |
| Interrupt Type | GIC Version | Interrupt ID Range | Bits Required |
| SGI (Software Interrupts) | GICv2/v3/v4 | 0 - 15 | 4 bits |
| PPI (Per-Core Interrupts) | GICv2/v3/v4 | 16 - 31 | 5 bits |
| SPI (Global Interrupts) | GICv2/v3/v4 | 32 - 1019 | 10 bits |
| LPI (Message-based Interrupts) | GICv3/v4 | 8192 - ~1M | 20 bits |

Key Takeaways

* GICv2 uses 10-bit interrupt addressing (max 1020 interrupts).
* GICv3/4 extends LPI addressing to 20 bits (supports up to 1 million interrupts).
* LPI uses memory-based addressing rather than traditional GICD registers.