The GIC (Generic Interrupt Controller) is a key component in ARM-based systems responsible for managing interrupts from various devices to the CPU cores. It is critical for efficient interrupt processing in multi-core systems.

The GIC (Generic Interrupt Controller) is designed to:

* Manage interrupts from multiple devices.
* Route them to one or more CPU cores.
* Prioritize and mask interrupts.
* Support virtualization (in GICv2 and GICv3).

The GICv3 (Generic Interrupt Controller version 3) is widely used in modern ARMv8 systems and supports advanced interrupt management, including virtualization and power management.

|  |  |
| --- | --- |
| Component | Description |
| Distributor (GICD) | Manages interrupt distribution to CPU interfaces. |
| Redistributor (GICR) | Manages interrupts for individual CPUs and power management. |
| CPU Interface (GIC-CPU) | Delivers interrupts to CPU cores. |
| ITS (Interrupt Translation Service) | Handles Message Signalled Interrupts (MSI) from PCIe devices. |

Interrupt Types in GICv3

1. SPIs (Shared Peripheral Interrupts): For devices shared across multiple CPUs (like PCIe devices).
2. PPIs (Private Peripheral Interrupts): Private to each CPU core (like local timers).
3. SGIs (Software Generated Interrupts): Raised by software for inter-core communication.
4. LPIs (Locality-specific Peripheral Interrupts): Delivered using ITS for high-performance devices.

Interrupt Flow from Hardware to Kernel

Step 1: Device Generates Interrupt

* A hardware device (like PCIe NIC) raises an interrupt to GICv3 via an IRQ line.
* If it's an MSI, the ITS translates it into an LPI and forwards it.

Step 2: GICv3 Distributor Receives Interrupt

* The Distributor (GICD) identifies the target CPU(s) based on interrupt affinity.
* It forwards the interrupt to the appropriate Redistributor (GICR).

Step 3: Redistributor Wakes Up the CPU

* The GICR triggers the corresponding CPU interface.
* If the CPU is in low power mode, it wakes it up.

Step 4: CPU Interface Signals the Kernel

* The GIC-CPU Interface asserts the IRQ line to the CPU.
* CPU jumps to the vector table and executes the kernel’s interrupt handler.

Step 5: Kernel Handles the Interrupt

* The kernel calls do\_IRQ() → handle\_IRQ() → device-specific driver.
* The driver processes the interrupt and performs necessary actions.

3. Detailed Internal Registers and Their Role

Here are the critical GICv3 registers:

|  |  |
| --- | --- |
| Register | Purpose |
| GICD\_CTLR | Enables/disables GICD. |
| GICD\_ISENABLERn | Enables specific interrupts. |
| GICD\_ISPENDRn | Shows pending interrupts. |
| GICD\_IPRIORITYRn | Configures interrupt priority. |
| GICR\_WAKER | Manages CPU wake-up from low power mode. |
| ICC\_IGRPEN1\_EL1 | Enables group 1 interrupts at CPU level. |
| ICC\_IAR1\_EL1 | Acknowledges active interrupts. |
| ICC\_EOIR1\_EL1 | Ends interrupt processing. |

4. Linux Kernel Interaction with GICv3

Step 1: GICv3 Initialization (During Boot)

* In the Linux Kernel, the GICv3 driver is typically found at:

drivers/irqchip/irq-gic-v3.c

* The kernel reads Device Tree (DT) to get GICv3 base addresses.
* gic\_of\_init() → gic\_v3\_probe() initializes GICv3.

Step 2: Mapping IRQ to CPU

* During device probe, the kernel calls:

irq\_of\_parse\_and\_map(dev, irq\_index);

* This maps hardware IRQ to Linux virtual IRQ and attaches a handler.

Step 3: Interrupt Handling Path

* When GICv3 signals an IRQ, the kernel jumps to:

do\_IRQ()

* This calls:

handle\_IRQ()

* Which ultimately triggers the device driver’s IRQ handler.

Step 4: EOI (End of Interrupt)

* Once the driver handles the interrupt, the kernel writes:

write\_icc\_eoir1\_el1(irq);

* This clears the interrupt from GICv3.

5. How GICv3 Supports Virtualization

GICv3 has a built-in virtualization extension that supports:

|  |  |
| --- | --- |
| Feature | Purpose |
| Virtual CPU Interface (vGIC) | Allows Guest VMs to receive interrupts. |
| List Registers (LR) | Holds virtual interrupts for Guest VMs. |
| VM Exit on IRQ | Exits Guest mode when IRQ occurs. |
| Direct Injection | Allows hypervisor to inject virtual interrupts. |

6. Key Interview Questions for Senior Engineer

Here’s an exhaustive list of GICv3 questions:

1. What is the difference between GICv2 and GICv3?
2. Explain the role of Redistributor in GICv3.
3. How does GICv3 support LPIs (Locality-specific Peripheral Interrupts)?
4. Describe the internal interrupt flow from device to kernel in GICv3.
5. What happens when an IRQ occurs in the kernel for GICv3?
6. What is the role of ITS (Interrupt Translation Service)?
7. How does the Linux kernel handle MSI interrupts via GICv3?
8. What happens if two CPUs receive the same interrupt in GICv3?
9. How does the kernel acknowledge and clear interrupts in GICv3?
10. Explain GICv3’s power management capabilities.

The complete Linux kernel call flow for GICv3, including all function calls, context switches, and register accesses.

Step 1: Device Raises Interrupt

* A hardware device (e.g., PCIe, timer) triggers an interrupt line (SPI/PPI).
* The interrupt line is routed to the GICv3 Distributor (GICD).

Step 2: GICv3 Distributor Receives Interrupt

* GICD checks the interrupt ID (e.g., 32 for PCIe device).
* It routes the interrupt to the corresponding Redistributor (GICR) based on:
  + Affinity (which CPU should handle it).
  + Priority.

GICv3 Register Access

GICD\_ISPENDR[n] - Sets interrupt pending state

GICD\_ISENABLER[n] - Enables interrupt

GICR\_WAKER - Wakes up the target CPU if in low power

Step 3: GICv3 Triggers CPU Interface

* The GICR (Redistributor) signals the CPU interface (GIC-CPU).
* IRQ Line is asserted on the CPU (like IRQ 32).
* CPU hardware invokes Exception Vector Table to handle the interrupt.

Step 4: Kernel Entry Point (do\_IRQ())

* The CPU jumps to the kernel's vector table located at:

arch/arm64/kernel/entry.S

* The function do\_IRQ() is invoked.

Function Call Flow

entry.S

--> do\_IRQ()

--> generic\_handle\_domain\_irq()

--> handle\_irq\_desc()

--> action->handler(dev\_id)

Step 5: Kernel Handles the IRQ

The handle\_IRQ() function is the core part of interrupt handling.

Function Flow

1. do\_IRQ() receives the IRQ number.
2. Calls:

generic\_handle\_domain\_irq(domain, irq);

1. This maps the hardware IRQ to a virtual IRQ in the kernel.
2. Calls:

irq\_desc->handle\_irq(irq\_desc);

1. This reaches the device driver interrupt handler.

Step 6: Device Driver Interrupt Handler

The kernel now jumps to the device driver’s ISR (Interrupt Service Routine).

Function Flow

irqreturn\_t device\_irq\_handler(int irq, void \*dev\_id)

{

// Clear hardware interrupt status

clear\_interrupt\_status();

// Process device event

process\_device\_event();

return IRQ\_HANDLED;

}

Who Registers This Handler? During driver probe, the driver registers this handler like:

request\_irq(irq\_number, device\_irq\_handler, IRQF\_SHARED, "my\_device", dev\_id);

Step 7: End of Interrupt (EOI) in Kernel

After the device driver finishes processing, the kernel sends EOI (End of Interrupt) to GICv3.

Function Flow

1. Kernel calls:

write\_icc\_eoir1\_el1(irq);

1. This writes the EOI register:

ICC\_EOIR1\_EL1 = IRQ Number

1. GICv3 now unmasks the IRQ and allows new interrupts.

Step 8: Return from Interrupt

* The kernel returns from interrupt (IRET).
* CPU resumes execution from where it was interrupted.

Call Flow in Kernel

do\_IRQ()

--> handle\_irq()

--> irq\_exit()

--> preempt\_enable()

--> return\_to\_user()

GICv3 Complete Register Interaction

Here’s the complete list of register interactions in the call flow:

|  |  |  |
| --- | --- | --- |
| Register | Purpose | Accessed By |
| GICD\_ISENABLERn | Enables interrupt | Device Tree Init |
| GICD\_ISPENDRn | Marks IRQ pending | Hardware |
| ICC\_IAR1\_EL1 | Acknowledge IRQ | Kernel Entry |
| ICC\_EOIR1\_EL1 | End of Interrupt | Kernel Exit |
| GICR\_WAKER | Wake up CPU | Redistributor |

Here’s the complete call stack from hardware IRQ to driver:

Hardware Device

↓

GICv3 Distributor

↓

GICv3 Redistributor

↓

CPU Interface (IRQ asserted)

↓

entry.S (vector table)

↓

do\_IRQ()

↓

handle\_IRQ()

↓

generic\_handle\_domain\_irq()

↓

irq\_desc->handle\_irq()

↓

device\_irq\_handler()

↓

write\_icc\_eoir1\_el1()

↓

return\_from\_irq()

Handling MSI with ITS (Interrupt Translation Service): If your device supports Message Signaled Interrupts (MSI) like PCIe, the flow changes:

1. Device sends MSI → GICv3 ITS receives MSI.
2. ITS translates MSI to LPI (Locality Specific Interrupt).
3. GICv3 Redistributor receives LPI.
4. Kernel flow remains the same.

GICv3 ITS (Interrupt Translation Service) with MSI and GICv3 Virtualization Flow with KVM in two detailed sections:

What is ITS (Interrupt Translation Service)? In GICv3, ITS (Interrupt Translation Service) handles MSI (Message Signaled Interrupts) from PCIe devices.

Instead of traditional IRQ lines, modern PCIe devices use MSI/MSI-X to signal interrupts.

* MSI (Message Signaled Interrupt): Interrupts sent as memory writes.
* ITS (Interrupt Translation Service): Translates MSI → LPI.

Why is ITS Important in GICv3?

* It allows PCIe devices to generate interrupts without physical IRQ lines.
* It converts MSI (Message) to LPI (Locality-specific Peripheral Interrupt).
* It can distribute interrupts dynamically across CPUs.

Step-by-Step Flow for MSI → LPI with ITS

Step 1: PCIe Device Triggers MSI

* PCIe device writes to MSI Address (in BAR).
* Example:

Address: 0xFEEDBEEF

Data: 0x00000001

* This memory write represents the MSI interrupt.

Step 2: PCIe Root Complex Sends Message

* The Root Complex (RC) forwards the MSI to GICv3's ITS.
* ITS receives the message and checks the device BDF (Bus/Device/Function).

Step 3: ITS Translates MSI to LPI

* ITS translates the MSI Message to LPI (Locality-specific Peripheral Interrupt).
* LPI is a lightweight interrupt for high-speed devices.
* ITS looks up Device ID → Collection Table → Interrupt ID.

Step 4: GICv3 Redistributor Delivers LPI

* ITS now tells GICR to deliver LPI to target CPU.
* GICR asserts IRQ line to CPU core.

Step 5: Kernel Handles LPI

* Kernel entry:

arch/arm64/kernel/entry.S

* Function flow:

do\_IRQ()

→ handle\_IRQ()

→ irq\_desc->handle\_irq()

→ PCIe device driver

GICv3 Registers Used in ITS

|  |  |
| --- | --- |
| Register | Purpose |
| GITS\_TRANSLATER | Translates MSI to LPI |
| GITS\_CREADR | Command queue read pointer |
| GITS\_CWRITER | Command queue write pointer |
| GITS\_BASER | ITS memory base address |
| GICR\_WAKER | Wakes up CPU when LPI occurs |

PCIe Device → MSI → PCIe RC → ITS → LPI → GICv3 → Kernel IRQ → Device Driver

Linux Kernel Call Flow for MSI

1. PCIe MSI triggered → Root Complex sends MSI.
2. ITS translates MSI → LPI.
3. Kernel Entry:

do\_IRQ() → handle\_IRQ()

1. Kernel calls device driver:

request\_irq(irq, handler);

1. End of Interrupt (EOI):

write\_icc\_eoir1\_el1(irq);

Why is ITS Critical?

* It allows thousands of interrupts with LPI without consuming physical IRQ lines.
* Ideal for high-performance PCIe devices like NIC, GPU, etc.

Part 2: GICv3 Virtualization Flow with KVM (Kernel-based Virtual Machine)

In virtualization, Guest VMs should receive interrupts without interfering with the host.

* GICv3 introduces vGIC (Virtual GIC).
* Each Guest VM gets a Virtual Interrupt Controller (vGIC).
* Hypervisor (KVM) manages interrupt injection.

Step-by-Step Virtualization Flow

Step 1: Guest VM Triggers Virtual IRQ

* Device in Guest VM generates an IRQ.
* GICv3 routes it to the Hypervisor first.

Step 2: Hypervisor Handles IRQ

* Host kernel (KVM) captures the IRQ:

kvm\_handle\_irq()

* It uses List Registers (LR) to inject IRQ to the Guest.

Step 3: vGIC Injects Virtual IRQ

* vGIC (Virtual GIC) is created for each Guest VM.
* vGIC populates List Register (LR) with Virtual IRQ.
* Guest OS sees the interrupt as if it was hardware.

Step 4: Guest VM Handles IRQ

* Guest kernel receives the interrupt normally via:

do\_IRQ()

* However, the IRQ is virtualized.

What is List Register (LR)?

* List Register (LR): Holds Virtual IRQs for Guest VM.
* KVM injects IRQ into LR.
* GICv3 then delivers the Virtual IRQ.

Key Registers in GICv3 Virtualization

|  |  |
| --- | --- |
| Register | Purpose |
| ICC\_SGI1R | Software interrupts from Host → Guest |
| ICC\_LR0\_EL1 | Holds virtual interrupts for Guest |
| ICC\_IAR1\_EL1 | Acknowledge virtual IRQ |
| ICC\_EOIR1\_EL1 | End of Virtual IRQ |

Call Flow for Virtual IRQ

Guest Device → Virtual IRQ → vGIC → List Register → Guest Kernel

Call flow in kernel:

kvm\_handle\_irq()

→ vgic\_inject\_irq()

→ write\_lr\_register()

Why Is This Important?

* Allows Guest VMs to use devices without host intervention.
* Provides near-bare-metal performance.
* KVM injects IRQ via List Registers (LR).

Key Differences in IRQ Flow (Host vs Guest)

|  |  |  |
| --- | --- | --- |
| Feature | Host IRQ | Guest IRQ |
| Trigger | GICv3 Distributor | vGIC (Virtual GIC) |
| Handling | Kernel IRQ handler | KVM hypervisor |
| Registers | ICC\_IAR1\_EL1 | ICC\_LR0\_EL1 |

If PCIe device triggers MSI in Guest:

1. PCIe Device → MSI → Host ITS → Host Kernel.
2. KVM Captures IRQ → Injects to Guest VM LR.
3. Guest handles the interrupt seamlessly.

GICv3 ITS Complete Command Flow and Memory Map. What Does ITS (Interrupt Translation Service) Do?

* ITS handles MSI (Message Signaled Interrupts) from PCIe devices.
* It translates MSI → LPI (Locality-specific Peripheral Interrupt).
* GICR then delivers the LPI to the target CPU.

ITS Command Flow (Detailed Breakdown)

Step 1: Device Sends MSI (Message Signaled Interrupt)

* PCIe device writes:

Address: 0xFEEDBEEF

Data: 0x00000001

* This represents MSI interrupt.

Step 2: ITS Receives MSI

* PCIe Root Complex forwards the MSI to ITS.
* ITS captures:
  + Device ID (BDF)
  + Event ID (Message)

Step 3: ITS Translates MSI → LPI

ITS executes these internal commands:

|  |  |
| --- | --- |
| Command | Purpose |
| MAPD | Map Device ID to Collection Table |
| MAPVI | Map Event ID to LPI |
| INV | Invalidate previous mappings |
| SYNC | Synchronize command queue |

Command Flow in Detail

1. MAPD (Map Device)

MAPD(DeviceID, Collection Table)

* + Maps PCIe device to collection table.

1. MAPVI (Map Virtual Interrupt)

MAPVI(DeviceID, EventID, LPI)

* + Converts MSI → LPI.

1. INV (Invalidate)

INV(DeviceID, EventID)

* + Clears stale interrupts.

1. SYNC (Synchronize)

SYNC(DeviceID)

* + Synchronizes the ITS memory.

Memory Structure in ITS

|  |  |
| --- | --- |
| Region | Purpose |
| Command Queue | Holds MAPD/MAPVI commands. |
| Device Table | Maps DeviceID → Collection Table. |
| Collection Table | Maps LPI → Redistributor. |
| Interrupt Translation Table | Maps EventID → LPI. |

LPI Flow After Translation

PCIe Device → MSI → ITS → LPI → GICR → CPU

Kernel Flow for ITS

1. Kernel initializes ITS driver:

drivers/irqchip/irq-gic-v3-its.c

1. Maps MSI to LPI:

gic\_msi\_init(dev, irq);

1. Sends EOI:

write\_icc\_eoir1\_el1(irq);

ITS Register Flow

|  |  |
| --- | --- |
| Register | Purpose |
| GITS\_TRANSLATER | Translates MSI to LPI |
| GITS\_BASER | Base address of ITS |
| GITS\_CREADR | Read pointer for command queue |
| GITS\_CWRITER | Write pointer for command queue |

Why Is ITS Critical?

* Handles thousands of interrupts without physical IRQ lines.
* Allows high-performance devices (NIC, GPU) to work seamlessly.

2. GICv3 Nested Virtualization Flow (Hypervisor inside Hypervisor)

What Is Nested Virtualization?

* Level 0 (L0): Physical Hypervisor (KVM in Host).
* Level 1 (L1): Guest VM (Linux in VM).
* Level 2 (L2): Nested VM (Another Hypervisor in Guest).

GICv3 supports nested virtualization by managing:

* Physical IRQs (PPI, SPI) → Physical devices.
* Virtual IRQs (vIRQ) → Guest OS devices.
* Virtualized Virtual IRQs (vvIRQ) → Nested Guest OS.

Key Components in Nested Virtualization

|  |  |
| --- | --- |
| Component | Purpose |
| vGIC (Virtual GIC) | Injects virtual interrupts to Guest OS. |
| vITS (Virtual ITS) | Translates MSI to LPI in Guest. |
| List Register (LR) | Holds pending virtual interrupts. |

Flow of Interrupt in Nested Virtualization

Step 1: L2 Guest Generates Interrupt

* Nested VM (L2) triggers an IRQ.
* vGIC in L1 captures the IRQ.

Step 2: L1 Hypervisor Handles Interrupt

* L1 Hypervisor (KVM) uses:

kvm\_handle\_irq();

* It injects the interrupt to L2.

Step 3: L2 Handles Virtualized Interrupt

* L2 kernel receives IRQ:

do\_IRQ();

* However, this IRQ was already virtualized once.

Interrupt Conversion Table

|  |  |  |
| --- | --- | --- |
| Origin | Target | Handling Layer |
| Physical IRQ | Host OS | GICv3 |
| Virtual IRQ | Guest OS (L1) | vGIC |
| vvIRQ | Nested Guest (L2) | vITS + vGIC |

How List Registers (LR) Work in Nested Virtualization

* GICv3 maintains LR (List Register).
* It queues virtual IRQs for L1.
* When L1 traps, it queues vvIRQs for L2.

Flow of Nested Virtual IRQ

Physical Device → GICv3 → L0 Host

↓

KVM Handles IRQ → Injects vIRQ to L1

↓

L1 Handles IRQ → Injects vvIRQ to L2

KVM Code Handling Flow

1. L0 Receives IRQ:

kvm\_handle\_irq();

1. L1 Receives vIRQ:

vgic\_v2\_inject\_irq();

1. L2 Receives vvIRQ:

vgic\_v3\_inject\_irq();

Critical Registers in Nested Virtualization

|  |  |
| --- | --- |
| Register | Purpose |
| ICC\_IAR1\_EL1 | Acknowledge IRQ in Guest |
| ICC\_EOIR1\_EL1 | End IRQ in Guest |
| ICC\_LR0\_EL1 | Inject virtual IRQ |
| ICC\_SGI1R\_EL1 | Send IPI to Guest |

Edge Case: MSI in Nested Virtualization

If PCIe device triggers MSI in L2:

1. PCIe → MSI → ITS → GICv3 → Host IRQ.
2. KVM traps and injects vIRQ.
3. L1 sees vIRQ and injects vvIRQ to L2.
4. L2 handles interrupt natively.

Why Is Nested Virtualization Complex?

* It requires:
  + Double IRQ translation (Physical → Virtual → Virtualized).
  + Multi-level IRQ processing.
  + High latency management.

Bonus: Handling EOI in Nested Virtualization: In L2, when the driver completes IRQ:

write\_icc\_eoir1\_el1(irq);

GICv3 then:

* Clears LR for L2.
* Propagates EOI to L1.
* Finally clears IRQ in Host.