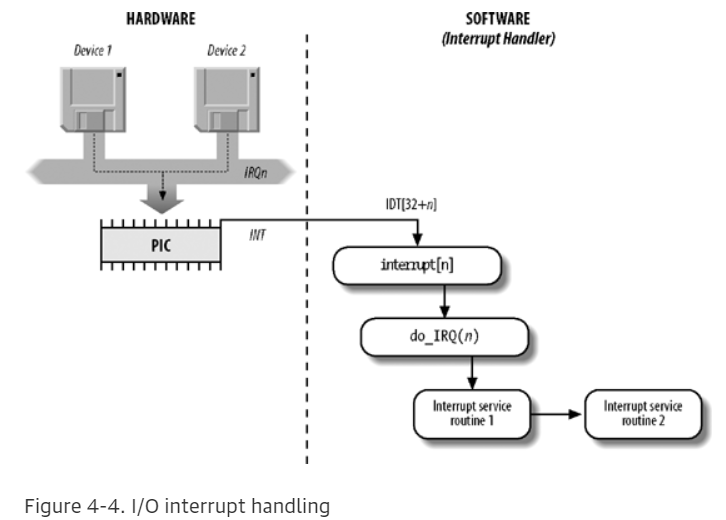
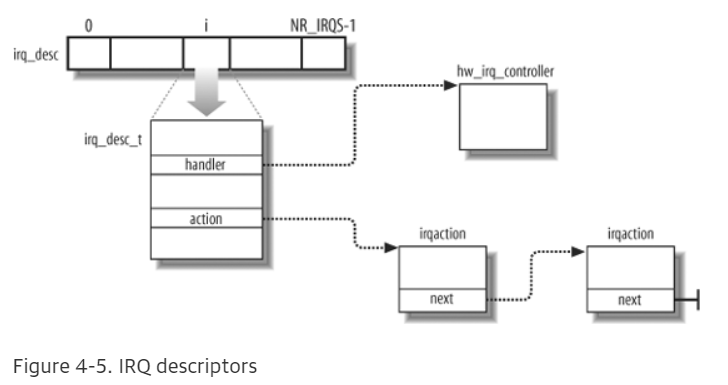
1. Physical IRQs may be assigned any vector in the range 32-238. However, Linux uses vector 128(0x80) to implement system calls.



1. IRQ Data Structure: -

The kernel must discover which I/O device corresponds to the IRQ number before enabling interrupts. Otherwise, for example, how could the kernel handle a signal from a SCSI disk without knowing which vector corresponds to the device? The correspondence is established while initializing each device driver.

Each interrupt is described by an interrupt descriptor structure irq\_desc. The interrupt is referenced by an ’unsigned int’ numeric value which selects the corresponding interrupt decription structure in the descriptor structures array. The descriptor structure contains status information and pointers to the interrupt flow method and the interrupt chip structure which are assigned to this interrupt.



# Linux IRQ Handling process: -

This document contains some notes relating to the code execution flow in the Linux kernel when an interrupt request (IRQ) is being addressed by the system. According to the [documentation](https://www.kernel.org/doc/html/latest/core-api/genericirq.html), the IRQ subsystem provides three main layers of abstraction:

1. High-level driver API
2. High-level IRQ flow handlers
3. Chip-level hardware encapsulation

The IRQ handling process deals mostly with flow handlers and chip-level abstraction, so the IRQ driver API will not be explored here.

When some significant hardware change occurs, low-level architecture code either calls the handle\_irq function stored in the IRQ descriptor (irq\_desc) directly or call a more generic generic\_handle\_irq\_desc function.

Because hardware can provide different types of interrupt sources (the rising/falling edge of an electric signal, the voltage level (low/high) of a circuit line) different actions may need to be taken in each case. The high-level IRQ flow handlers provide pre-defined approaches to deal with hardware interrupts. These flow handlers are assigned to the interrupt descriptors at boot time or during device initialization.

It is also possible for architecture code to implement specific flow handlers. Whatever function best suits the interrupt flow handling, irq\_desc holds a pointer to it which is then used to give appropriate continuation to the IRQ handling.

Assuming the flow handler is one of the high-level IRQ flow handlers, it may do a few things before proceeding with the IRQ handling. For instance, the hardware interrupt controller may somehow need an ack from the CPU, signaling that the interrupt was properly received.

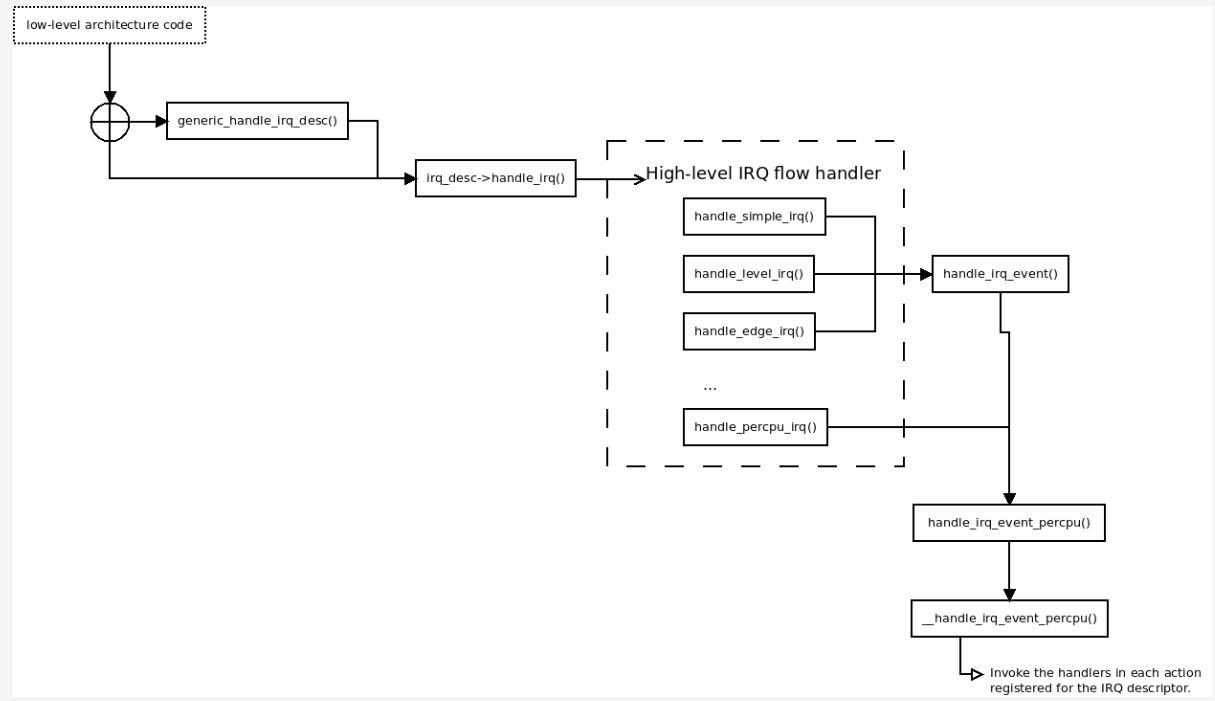
It may also be needed to mask (disable) and/or unmask (enable) interrupts for some chips. The beauty in it is that the flow handlers don’t need to know architecture-specific details to accomplish these actions, all of it can be done relying on the irq\_chip abstraction which encapsulates the hardware relevant functions.

Afterward, **handle\_irq\_event** is called to set the IRQ state as “in progress”, acquire the IRQ description lock, and then call **handle\_irq\_event\_percpu**. If **handle\_percpu\_irq** is the flow handler being used, the number of IRQs handled by the CPU is incremented and handle\_irq\_event\_percpu is called directly (per CPU IRQs are not serialized).

handle\_irq\_event\_percpu does nothing but calling \_\_handle\_irq\_event\_percpu and waiting for its return to add some randomness to the pool of interrupts handled by the CPU.

\_\_handle\_irq\_event\_percpu is where the IRQ subsystem finally let other stakeholders of the interrupt event take actions. The IRQ descriptor has a list of actions, each one having a handler. The handle irq event function calls the handler function for each action as can be seen in both the diagram and code below. This handler is known as the “top half” of IRQ handling.

The action may have an additional handler function (thread\_fn) which, if not NULL, is executed in a threaded interrupt context (this is the bottom half of IRQ handling). Both the handler and the threaded handler functions are assigned to the irq\_desc struct by calling request\_threaded\_irq, which is often done, direct or indirectly, by device drivers during device initialization. Therefore, the for\_each\_action\_of\_desc loop in \_\_handle\_irq\_event\_percpu is very important because it is where drivers get notified of hardware events and can work over them, for instance, fetching data from a device.



Function call diagram for the IRQ handling process



Figure: The \_\_handle\_irq\_event\_percpu function. The place where IRQ actions are invoked.

After all the actions have been executed or any of them have returned IRQ\_HANDLED, \_\_handle\_irq\_event\_percpu returns an oring with all the flags returned by each executed handler. These flags are used by handle\_irq\_event\_percpu to add randomness to IRQ handling as mentioned above. After that, the flags are returned back to handle\_irq\_event (or to handle\_percpu\_irq) which in turn returns the flags back to the flow handlers. Though none of the generic flow handlers use the returned flags, these will be available in case architecture-specific flow handlers want to use them.

Important source code files: -

* Documentation/core-api/genericirq.rst
* include/linux/interrupt.h
* include/linux/irqdesc.h
* kernel/irq/chip.c
* kernel/irq/handle.c
* kernel/irq/manage.c