# About the interrupt :

The core responsibility of the Linux kernel is managing the hardware{External memory, GPIO,wireless radios,}connectivity on the machine .To meet this responsibility, the kernel needs to communicate with the machine’s individual devices. The processor on given machine is faster than the hardware need to talk.So it's not ideal for the kernel to issue a request and wait for a response from the significantly slower hardware.

Instead,the hardware is comparatively respond is slow,So kernel must be free to go and handle other work,after finising task on current hardware the kenrel touch the other task based on the hardware requrest. So the using the polling the processor work with hardware without impacting the machine’s overall performance. The kernel periodically check the status of the hardware in the system and respond accordingly.During the polling the CPU check the hardware status repeatedly a fixed time duration.So during the polling cpu wast more cpu cycle than expeced,so this is main drawapack exist in polling mechanism.

To overcome with this drawback we need better solution is to provide a mechanism for the hardware to signal to the kernel when attention is needed.This mechanism is called an interrupt.

# Interrupts :

The external device triggered an hardware interrupt,to signal to the processor.The processor receives the interrupt(from the data bus and decode it opcode to find IRQ number and based on the IRQ number it hints to kernel about IRS) through the irq line and signals the operating system to enable the operating system to respond to the new data.The Hardware devices generate interrupts asynchronously with respect to the processor clock. Consequently,the kernel can be interrupted at any time to process interrupts.

An interrupt is the electronic signals physically origined from external hardware devices and directed into input pins on an interrupt controller.The interrupt controller is a simple chip that multiplexes multiple interrupt lines into a single line(irq line) to the processor.Upon receiving an interrupt, the interrupt controller sends a signal to the processor interrupt pins.The processor detects this signal and decode it to find and interrupts its current execution to handle the interrupt.

Different devices can be associated with different interrupts by means of a unique value associated with each interrupt. This way,interrupts from the keyboard are distinct from interrupts from the hard drive. This enables the operating system to differentiate between interrupts and to know which hardware device caused which interrupt. In turn, the operating system can service each interrupt with its corresponding handler.

These “interrupt values” are often called interrupt request (IRQ) lines. Each IRQ line is

assigned a numeric value for example IRQ zero is the timer interrupt and IRQ one is the keyboard interrupt.The important notion is that a specific interrupt is associated with a

specific device, and the kernel knows this.The hardware then issues interrupts to get the

kernel’s attention: “Hey, I have new key presses waiting! Read and process these bad boys!…………………….”

# Exceptions:

In the OS exceptions are often discussed at the same time as interrupts.The exceptions occur synchronously with respect to the processor clock.Indeed, they are often called synchronous interrupts.Exceptions are produced by the processor while executing instructions either in response to a programming error or abnormal conditions that must be handled by the kernel  like this.For example, divide by zero and a page fault).

# Interrupt Handlers :

The function that run by kernel in response to a specific interrupt is called an interrupt handler or interrupt service routine (ISR).The vector address of the ISR stored in Vector address table of Random access memory.So when interrupt occurs the Programe Counter load the vector address of ISR from the Vector address table and run the device specific ISR. Each device that generates interrupts has an associated interrupt handler. For example, osystem timer has different interrupt handler from interrupts handler of keyboard. The interrupt handler for a device is part of the device’s driver.

The interrupt handlers are normal C functions. They match a **specific prototype**,which enables the kernel to pass the handler information in a standard way to the cpu.The kernel invokes the ISR in response to interrupts and that they run in a special context called **interrupt context**. This special context is occasionally called **atomic context .** The code executing in the atomic context is unable to block.

The interrupt can occur at any time, So an interrupt handler can, in turn, be executed at any time.It is imperative that the handler runs quickly, to resume execution of the interrupted code as soon as possible. The kernel need to serve the interrupt handler of the device without any and delay and ISR need execute the code in as short a period as possible.

At the very least, an interrupt handler’s job is to acknowledge the interrupt’s receipt to the hardware.The interrupt handlers have a large amount of work to perform that can divied in two part: Top Halves and Bottom halves. The top halves executed for critical operation and Bottom halves run for operating that can be peform in future. For example, consider the interrupt handler for a network device. On top halves mechanism following is done :

* Copy networking packets from the hardware(netword device memory) into memory.
* Processing the packet .
* Push the packets down to the appropriate protocol stack.
* Acknowlegde the Hardware about the packet recipt in system memory.
* The Resting the hardware like register ,flag etc.
* The scheduling which part of ISR execute in bottom halves.

# Top Halves Versus Bottom Halves:

These two mechanism an interrupt handler execute quickly and perform a large amount of work. The interrupts processing is split into two parts,top halves and bottom halves.The interrupt handler execute in the top half.The top half is run immediately upon receipt of the interrupt and performs only the work that is **time-critical**, such as acknowledging receipt of the interrupt or resetting the hardware.the work that can be performed later is deferred to the bottom half.The bottom half runs in the future, at a more convenient time, with all interrupts enabled .Linux provides various mechanisms for implementing bottom halves.Let’s look at an example of the top-half/bottom-half :

When network cards receive packets from the network, they need to alert the kernel of their availability.They want and need to do this immediately, to optimize network throughput and latency and avoid timeouts.Thus, they immediately issue an interrupt: Hey, kernel, I have some fresh packets here! The kernel responds by executing the network card’s registered interrupt.So on the ISR following operation is executed :acknowledges the hardware,copies the new networking packets into main memory, and readies the network card for more packets.These jobs are the important, time-critical, and hardware-specific work.

# Registering an Interrupt Handler:

The Interrupt handlers are the responsibility of the linux driver that managing the hardware. Each device has one associated driver and, if that device uses interrupts then that driver must register one interrupt handler.The drivers can register an interrupt handler and enable a given interrupt line for handling with the function **request\_irq()** , which is declared in **<linux/interrupt.h>** .The kernel allocate an interrupt line and using do\_irq() keep monitoring them.When interrupt raised on this given line cpu interrupt the the to monitor the interrupt line:

int request\_irq(unsigned int irq,

irq\_handler\_t handler,

unsigned long flags,

const char \*name,

void \*dev)

The first parameter, irq , specifies the interrupt number to allocate. This can be hard-coded. For most other devices, it is probed or otherwise determined programmatically and dynamically.

The second parameter, handler , is a function pointer to the actual interrupt handler

that services this interrupt.This function is invoked whenever the operating system

receives the interrupt.

typedef irqreturn\_t (\*irq\_handler\_t)(int, void \*);

**Interrupt Handler Flags:**

The third parameter, flags , can be either zero or a bit mask of one or more of the flags

defined in <linux/interrupt.h> .Among these flags, the most important are given bellow:

**IRQF\_DISABLED** —When set, this flag instructs the kernel to disable all interruptswhen executing thiinterrupt handler.When unset, interrupt handlers run with allinterrupts except their own enabled. Most interrupt handlers do not set this flag, as disabling all interrupts is bad form.

**IRQF\_SAMPLE\_RANDOM** —This flag specifies that interrupts generated by this device should contribute to the kernel entropy pool.The kernel entropy pool provides truly random numbers derived from various random events. If this flag is specified, the timing of interrupts from this device are fed to the pool as entropy. Do not set this if your device issues interrupts at a predictable rate (for example, the systemtimer) or can be influenced by external attackers (for example, a networking device).

**IRQF\_TIMER** —This flag specifies that this handler processes interrupts for the sys-

tem timer.

**IRQF\_SHARED** —This flag specifies that the interrupt line can be shared among multiple interrupt handlers. Each handler registered on a given line must specify this flag. More information on shared handlers is provided in a following section.

The fourth parameter, **name** , is an ASCII text representation of the device associated

with the interrupt.These text names are used by /proc/irq and /proc/interrupts for communication with the user,

The fifth parameter, dev , is used for shared interrupt lines. When an interrupt handler

is freed,dev provides a unique cookie to enable the removal of only the desired interrupt handler from the interrupt line.Without this parameter, it would beimpossible for the kernel to know which handler to remove on a given interrupt line.

On success, request\_irq() returns zero.A nonzero value indicates an error, in whichcase the specified interrupt handler was not registered A common error is -EBUSY , which

denotes that the given interrupt line is already in use .

Note that request\_irq() can sleep and therefore cannot be called from interrupt context or other situations where code cannot block.In a driver, requesting an interrupt line and installing a handler is done via :

if (request\_irq(irqn, my\_interrupt, IRQF\_SHARED, "my\_device", my\_dev))

{

printk(KERN\_ERR "my\_device: cannot register IRQ %d\n", irqn);

return -EIO;

}

# Freeing an Interrupt Handler:

When your driver unloads, you need to unregister your interrupt handler and potentially disable the interrupt line.To do this, need to call :

void free\_irq(unsigned int irq, void \*dev)

If the specified interrupt line is not shared, this function removes the handler and disables the line but If the interrupt line is shared, the handler identified via **dev** is removed, but the interrupt line is disabled only when the last handler is removed.With shared interrupt lines, a unique cookie is required to differentiate between the multiple handlers that can exist on a single line and enable free\_irq() to remove only the correct handler.

# Writing an Interrupt Handler:

The following is a declaration of an interrupt handler:

static irqreturn\_t intr\_handler(int irq, void \*dev);

This declaration(intr\_handler) matches the prototype of the handler argument given to the request\_irq(intr\_handler) function.The first parameter **irq** is the numeric value of the interrupt line the on that handler is servicing.This value is passed into the handler, but it is not used very often,except in printing log messages.

The second parameter, **dev** , is a generic pointer to the same dev that was given to request\_irq() when the interrupt handler was registered. If this value is unique or set it can act as a cookie to differentiate multiple devices that potentially using the same interrupt handler.dev might also point to a structure of use to the interrupt handler .

The return value of an interrupt handler is the special type **irqreturn\_t.** An interrupt handler can return two special values, IRQ\_NONE or IRQ\_HANDLED.The former(IRQ\_NONE) is returned when the interrupt handler detects an interrupt is’t generateed from original matched device .The latter is returned if the interrupt handler was correctly invoked, and its device did indeed cause the interrupt.

These special values are used to let the kernel know whether devices are issuing spurious interrupts.If all the interrupt handlers on a given interrupt line return IRQ\_NONE , then the kernel can detect the problem.The role of the interrupt handler depends entirely on the device and its reasons forissuing the interrupt.At a minimum, most interrupt handlers need to provide acknowledgment to the device that they received the interrupt.

How ISR can be expected :

* Interrupt handlers in Linux need not be reentrant.
* Interrupt handlers in Linux need not be prempbtle .
* The same interrupt handler is never invoked concurrently to service a nested interrupt. It me perfect locking mechanism not blocked or sleep
* ISR must be simple and highly optimized
* Can not used dynamic library function .

# Interrupt Context :

When executing an interrupt handler, the kernel is in **interrupt context**. When kernel is in while it is executing on behalf of a process then its called process context-for example, executing a system call or running a kernel thread.In process context, the current macro points to the associated task and a process is **process context** can sleep or otherwise invoke the scheduler.

Interrupt context, on the other hand, is not associated with a process.The currentmacro is not relevant (although it points to the interrupted process).Without a backing process, interrupt context cannot sleep Therefore, you cannot call certain functions from interrupt context. If a function sleeps, you cannot use it from your interrupt handler—this limits the functions that one can call from an interrupt handler.

Interrupt context is time-critical because the interrupt handler interrupts other code.Code should be quick and simple. Busy looping is possible, but discouraged.This is animportant point; always keep in mind that your interrupt handler has interrupted othercode (possibly even another interrupt handler on a different line!). Because of this asynchronous nature, it is imperative that all interrupt handlers be as quick and as simple as possible.As much as possible, work should be pushed out from the interrupt handler and performed in a bottom half, which runs at a more convenient time.Your interrupt handler should not care what stack setup is in use or what the size ofthe kernel stack is.Always use an absolute minimum amount of stack space.