Hardware interrupts - asynchronous interrupts

Software interrupts - due to exceptions or special instructions(sysenter - to shift the processor to privileged mode) - synchronous interrupts.

Interrupt Service Routine - ISR

To locate the appropriate ISR that corresponds to an interrupt event, interrupt vector tables are used. An interrupt vector is an address in memory that contains a reference to a software-defined interrupt service to be executed in response to an interrupt.

for most processor architectures, all supported vectors are set up in memory as a list called an interrupt vector table, whose address is programmed into a processor register by the platform software.

How interrupts are executed in x86?

Part 1 - Finding the correct handler.

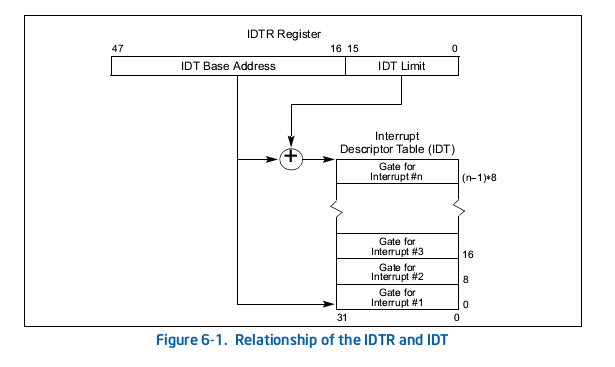
The x86 family of processors supports a total of 256 interrupt vectors, of which the first 32 are reserved for processor exceptions and the rest used for software and hardware interrupts.

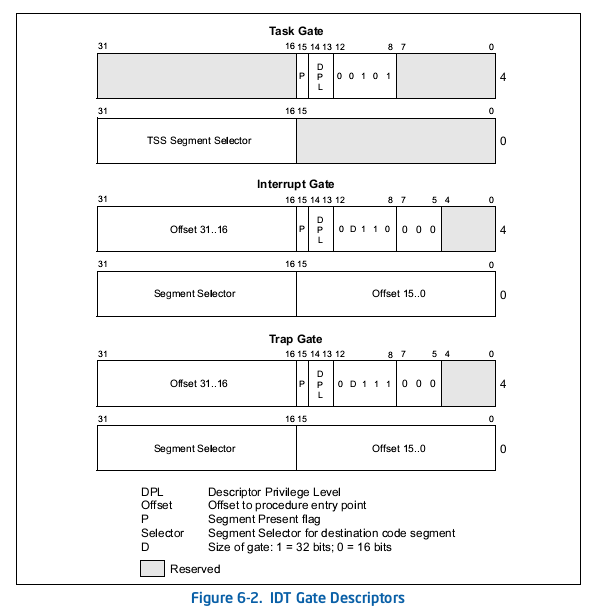
Implementation of a vector table by x86 is referred to as an interrupt descriptor table (IDT), which is an array of descriptors of either 8 byte (for 32-bit machines) or 16 byte (for 64-bit x86 machines) sizes. During early boot, the architecture-specific branch of the kernel code sets up the IDT in memory and programs the IDTR register (special x86 register)of the processor with the physical start address and length of the IDT.

When an interrupt occurs, the processor locates relevant vector descriptors by multiplying the reported vector number by the size of the vector descriptor (vector number x 8 on x86\_32 machines, and vector no x 16 on x86\_64 machines) and adding the result to the base address of the IDT. Once a valid vector descriptor is reached, the processor continues with the execution of actions specified within the descriptor.

On x86 platforms, each vector descriptor implements a gate (interrupt, task, or trap), which is used to transfer control of execution across segments. Vector descriptors representing hardware interrupts implement an interrupt gate, which refers to the base address and offset of the segment containing interrupt handler code. An interrupt gate disables all maskable interrupts before passing control to a specified interrupt handler. Vector descriptors representing exceptions and software interrupts implement a trap gate, which also refers to the location of code designated as a handler for the event. Unlike an interrupt gate, a trap gate does not disable maskable interrupts, which makes it suitable for execution of soft interrupt handlers.

There are registers on the 80386 and higher processors that are not well documented by Intel. These are divided in control registers, debug registers, test registers and protected mode segmentation registers. As far as I know, the control registers, along with the segmentation registers, are used in protected mode programming, all of these registers are available on 80386 and higher processors except the test registers that have been removed on the pentium. Control registers are CR0 to CR4, Debug registers are DR0 to DR7, test registers are TR3 to TR7 and the protected mode segmentation registers are GDTR (Global Descriptor Table Register), IDTR (Interrupt Descriptor Table Register), LDTR (Local DTR), and TR





The base address(the address where the IDT starts) of the IDT (Interrupt Description Table) is given by the bits 16 to 47, 16 and 47 inclusive. To get the address of the vector #3 the size of the first 2 entries has to be added to the base address.

 the processor locates the IDT using the IDTR register. This register holds both a 32-bit base address and 16-bit limit for the IDT.

The base addresses of the IDT should be aligned on an 8-byte boundary to maximize performance of cache line fills. The limit value is expressed in bytes and is added to the base address to get the address of the last valid byte. A limit value of 0 results in exactly 1 valid byte. Because IDT entries are always eight bytes long, the limit should always be one less than an integral multiple of eight (that is, 8N – 1).

BaseAddress=IDTR/(2^16) -> Integer division

Limit=IDTR%(2^16)

VectorPhysicalAddress=BaseAddress+(VectorNumber-1)\*8

Physical addresses of the segments selector bytes are: VectorPhysicalAddress+2 and VectorPhysicalAddress+3

Physical addresses of the offset bytes are: VectorPhysicalAddress, VectorPhyscialAddress+1, VectorPhysicalAddress+6, VectorPhysicalAddress+7

Programmable Interrupt Controller (PIC)

 All computing platforms use a hardware circuit called a programmable interrupt controller (PIC) to multiplex the CPU's interrupt pin across various interrupt request lines. All of the existing IRQ lines originating from on-board device controllers are routed to input pins of the interrupt controller, which monitors each IRQ line for an interrupt signal, and upon arrival of an interrupt, converts the request into a cpu-understandable vector number and relays the interrupt signal on to the CPU's interrupt pin.

 Intel x86 multiprocessor platforms use Advanced Programmable Interrupt Controller (APIC). The APIC design splits interrupt controller functionality into two distinct chipsets: the first component is an I/O APIC that resides on the system bus. All shared peripheral hardware IRQ lines are routed to the I/O APIC; this chip translates an interrupt request into vector code. The second is a per-CPU controller called Local APIC (usually integrated into the processor core) which delivers hardware interrupts to specific CPU cores. I/O APIC routes the interrupt events to a Local APIC of the chosen CPU core. It is programmed with a redirection table, which is used for making interrupt routing decisions. CPU Local APICs manage all external interrupts for a specific CPU core; additionally, they deliver events from CPU local hardware such as timers and can also receive and generate inter-processor interrupts(IPIs) that can occur on an SMP platform.

Multicore ARM platforms split the generic interrupt controller (GIC) implementation into two. The first component is called a distributor, which is global to the system and has several peripheral hardware interrupt sources physically routed to it. The second component is replicated per-CPU and is called the cpu interface. The distributor component is programmed with distribution logic of shared peripheral interrupts(SPI) to known CPU interfaces.

struct irq\_chip

Architecture independent structure that declares a set of function pointers to account for all peculiarities of IRQ chips found across various architectures.

The kernel's generic interrupt-management layer maps each hardware IRQ to a unique identifier called Linux IRQ; these numbers abstract hardware IRQs, thereby ensuring portability of kernel code. All of the peripheral device drivers are programmed to use the Linux IRQ number to bind or register their interrupt handlers.

Linux IRQs are represented by IRQ descriptor structure, which is defined by struct irq\_desc; for each IRQ source, an instance of this structure is enumerated during early kernel boot. A list of IRQ descriptors is maintained in an array indexed by the IRQ number, called the IRQ descriptor table:

struct irq\_desc - Linux IRQ descriptor structure.

An array of IRQ descriptors indexed by IRQ number - IRQ descriptor table.

struct irq\_data - contains Linux IRQ number, hardware IRQ number, pointer to struct irq\_chip.

the mapping between Linux irq numbers and HW irq numbers is provided by the irq\_domain library.  
You can check the <linux src>/Documentation/IRQ-domain.txt for complete explanation of this.  
  
Basicaly to get Linux irq number from hw irq number you use irq\_find\_mapping() function.  
To find HW irq number from linux irq number you can use something like:  
  
unsigned int hw\_irq, linux\_irq;  
struct irq\_data \*irq\_data = irq\_get\_irq\_data(linux\_irq);  
hw\_irq = irqd\_to\_hwirq(irqd); /\* it just returns irq\_data->hwirq \*/

The handle\_irq element of the irq\_desc structure is a function pointer of type irq\_flow\_handler\_t, which refers to a high-level function that deals with flow management on the line. The generic irq layer provides as set of predefined irq flow functions; an appropriate routine is assigned to each interrupt line based on its type.

* handle\_level\_irq(): Generic implementation for level-triggered interrupts
* handle\_edge\_irq(): Generic implementation for edge-triggered interrupts
* handle\_fasteoi\_irq(): Generic implementation for interrupts that only need an EOI at the end of the handler
* handle\_simple\_irq(): Generic implementation for simple interrupts
* handle\_percpu\_irq(): Generic implementation for per-CPU interrupts
* handle\_bad\_irq(): Used for spurious interrupts

The \*action element of the irq\_desc structure is a pointer to one or a chain of action descriptors, which contain driver-specific interrupt handlers among other important elements. Each action descriptor is an instance of struct irqaction defined in the kernel header <linux/interrupt.h>

/\*\*

\* struct irqaction - per interrupt action descriptor

\* @handler: interrupt handler function

\* @name: name of the device

\* @dev\_id: cookie to identify the device

\* @percpu\_dev\_id: cookie to identify the device

\* @next: pointer to the next irqaction for shared interrupts

\* @irq: interrupt number

\* @flags: flags

\* @thread\_fn: interrupt handler function for threaded interrupts

\* @thread: thread pointer for threaded interrupts

\* @secondary: pointer to secondary irqaction (force threading)

\* @thread\_flags: flags related to @thread

\* @thread\_mask: bitmask for keeping track of @thread activity

\* @dir: pointer to the proc/irq/NN/name entry

\*/

/\*\*

\* request\_irq - allocate an interrupt line

\* @irq: Interrupt line to allocate

\* @handler: Function to be called when the IRQ occurs.

\* @irqflags: Interrupt type flags

\* @devname: An ascii name for the claiming device

\* @dev\_id: A cookie passed back to the handler function

\*/

int request\_irq(unsigned int irq, irq\_handler\_t handler, unsigned long flags,

const char \*name, void \*dev);

request\_irq() instantiates an irqaction object with values passed as parameters and binds it to the irq\_desc specified as the first (irq) parameter. This call allocates interrupt resources and enables the interrupt line and IRQ handling. Handler is a function pointer of type irq\_handler\_t, which takes the address of a driver-specific interrupt handler routine.flags is a bitmask of options related to interrupt management.

Flag bits are defined in the kernel header <linux/interrupt.h>:

* IRQF\_SHARED: Used while binding an interrupt handler to a shared IRQ line.
* IRQF\_PROBE\_SHARED: Set by callers when they expect sharing mismatches to occur.
* IRQF\_TIMER: Flag to mark this interrupt as a timer interrupt.
* IRQF\_PERCPU: Interrupt is per CPU.
* IRQF\_NOBALANCING: Flag to exclude this interrupt from IRQ balancing.
* IRQF\_IRQPOLL: Interrupt is used for polling (only the interrupt that is registered first in a shared interrupt is considered for performance reasons).
* IRQF\_NO\_SUSPEND: Do not disable this IRQ during suspend. Does not guarantee that this interrupt will wake the system from a suspended state.
* IRQF\_FORCE\_RESUME: Force-enable it on resume even if IRQF\_NO\_SUSPEND is set.
* IRQF\_EARLY\_RESUME: Resume IRQ early during syscore instead of at device resume time.
* IRQF\_COND\_SUSPEND: If the IRQ is shared with a NO\_SUSPEND user, execute this interrupt handler after suspending interrupts. For system wakeup devices, users need to implement wakeup detection in their interrupt handlers.

void \*dev - used to identify corresponding device when registering shared IRQs.

Interrupt handler routines have the following prototype:

irqreturn\_t handler(int irq, void \*dev\_id);

The interrupt handler should return IRQ\_NONE to indicate that the interrupt was not handled. It is also used to indicate that the source of the interrupt was not from its device in a shared IRQ case. When interrupt handling has completed normally, it must return IRQ\_HANDLED to indicate success. IRQ\_WAKE\_THREAD is a special flag, returned to wake up the threaded handler;

/\*\*

\* free\_irq - free an interrupt allocated with request\_irq

\* @irq: Interrupt line to free

\* @dev\_id: Device identity to free

\*

\* Remove an interrupt handler. The handler is removed and if the

\* interrupt line is no longer in use by any driver it is disabled.

\* On a shared IRQ the caller must ensure the interrupt is disabled

\* on the card it drives before calling this function. The function

\* does not return until any executing interrupts for this IRQ

\* have completed.

\* Returns the devname argument passed to request\_irq.

\*/

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

dev\_id is the unique cookie (assigned while registering the handler) to identify the handler to be deregistered in a shared IRQ case; this argument can be NULL for other cases. This function is a potential blocking call, and must not be invoked from an interrupt context: it blocks calling context until completion of any interrupt handler currently in execution, for the specified IRQ line.

**Threaded Interrupt Handlers**

Interrupt handler code path is asynchronous, and runs by suspending scheduler preemption and hardware interrupts on the local processor, and so is referred to as a hard IRQ context. So interrupt handlers must be:

- as short as possible

- non-blocking (atomic)

A split-handler design for the interrupt handler, called top half and bottom half. Top half routines are invoked in hard interrupt context, and these functions are programmed to execute interrupt critical operations, such as physical I/O on the hardware registers, and schedule the bottom half for deferred execution. Bottom half routines are usually programmed to deal with the rest of the interrupt non-critical and deferrable work, such as processing of data generated by the top half, interacting with process context, and accessing user address space.

the kernel supports setting up interrupt handlers that can execute in a thread context, called threaded interrupt handlers. Drivers can set up threaded interrupt handlers through an alternate interface routine called request\_threaded\_irq():

/\*\*

\* request\_threaded\_irq - allocate an interrupt line

\* @irq: Interrupt line to allocate

\* @handler: Function to be called when the IRQ occurs.

\* Primary handler for threaded interrupts

\* If NULL and thread\_fn != NULL the default

\* primary handler is installed

\* @thread\_fn: Function called from the irq handler thread

\* If NULL, no irq thread is created

\* @irqflags: Interrupt type flags

\* @devname: An ascii name for the claiming device

\* @dev\_id: A cookie passed back to the handler function

\*/

int request\_threaded\_irq(unsigned int irq ,irq\_handler\_handler , irq\_handler\_thread\_fn, unsigned long irqflags, const char\* devname,void\* dev\_id);

thread\_fn runs when handler function returns IRQ\_WAKE\_THREAD.

irqflags related to threaded IRQ:

* IRQF\_ONESHOT: The interrupt is not re-enabled after the hard IRQ handler is finished. This is used by threaded interrupts that need to keep the IRQ line disabled until the threaded handler has been run.
* IRQF\_NO\_THREAD: The interrupt cannot be threaded. This is used in shared IRQs to restrict the use of threaded interrupt handlers.

If handler is null, a default primary handler is run which simply returns IRQ\_WAKE\_THREAD.

**Deferred Work**

-Softirqs

-Tasklets

-Workqueues

Each of these frameworks constitute a set of data structures, and function interfaces, used for registering, scheduling, and queuing of the bottom half routines.

**Softirqs**