# LECCIÓN 3: INTERRUPCIONES HARDWARE

# LECCIÓN 3: INTERRUPCIONES HARDWARE

## 3.1 Introducción

Las interrupciones hardware son producidas por varias fuentes, por ejemplo del teclado, cada vez que se presiona una tecla y se suelta se genera una interrupción. Otras interrupciones son originadas por el reloj, la impresora, el puerto serie, el disco, etcétera.

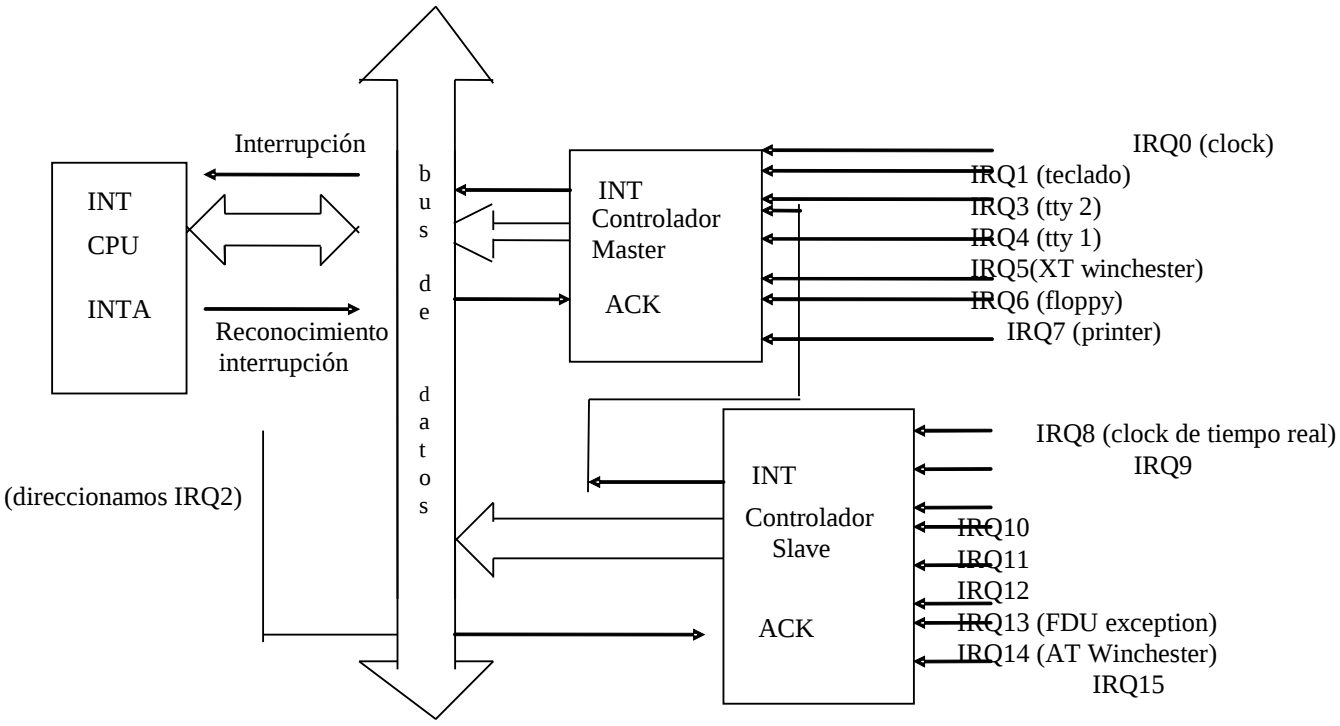
Una interrupción de tipo hardware es una señal eléctrica producida por un dispositivo físico del ordenador. Esta señal informa a la CPU que el dispositivo requiere su atención. La CPU parará el proceso que está ejecutando para atender la interrupción. Cuando la interrupción termina, la CPU reanuda la ejecución en donde fue interrumpida, pudiendo ejecutar el proceso parado originalmente o bien otro proceso.

Existe un hardware específico, para que los dispositivos puedan interrumpir lo que está haciendo la CPU. La propia CPU, tiene entradas específicas para ser interrumpida INT, cuando se activa esta entrada INT, la CPU para lo que está haciendo y activa la salida para reconocer la interrupción INTA, y comienza a ejecutar el código especial que maneja la interrupción. Algunas CPU´s disponen de un conjunto especial de registros, que solo son utilizados en el modo de ejecución de interrupciones, lo que facilita el trabajo de tratar las interrupciones.

La placa base del computador utiliza un controlador para decodificar las interrupciones que no son mas que señales eléctricas producidas por los dispositivos, coloca en el bus de datos información de que dispositivo interrumpió y activa la entrada INT de interrupción de la CPU. Este chip controlador protege a la CPU y la aísla de los dispositivos que interrumpen, además de proporcionar flexibilidad al diseño del sistema. El controlador de interrupciones tiene un registro de estado para permitir o inhibir las interrupciones en el sistema.

## 3.2 Sistema de Interrupciones basados en Controladores de Interrupciones Programables (arquitectura i386).

Existe diverso hardware para implementar un controlador de interrupciones, los computadores IBM PC o compatibles, utilizan el controlador de interrupciones programable de Intel 82C59A-2 Cmos o sus chips compatibles. Este controlador ha sido utilizado desde los comienzos del IBM PC, y es bien conocido el espacio de direccionamiento de sus registros en la arquitectura ISA. Incluso en chips más modernos se ha mantenido la misma localización.




En la figura, se muestra dos controladores de 8 entradas, cada uno de ellos tiene una máscara y un registro de estatus de interrupción, un PIC1 y un PIC2. Los registros de máscara están en los direccionamientos *0x21* y *0xA1* y los registros del estatus están *en 0x20 y 0xA0*.

Al escribir en un bit determinado del registro de máscara permite una interrupción, escribiendo un cero se invalida esta interrupción. Así pues, escribir un uno en la entrada 3 permite la interrupción 3, escribiendo cero se invalida. Los registros de máscara de interrupción son solamente de escritura, por lo tanto Linux debe guardar una copia local de lo que se ha escrito en los registros de máscara.

Cuando se produce una señal de interrupción, el código de manejo de la interrupción lee dos registros de estatus de interrupción (ISRs). Trata el ISR en *0x20* como los ocho bits inferiores, y el ISR en *0xA0* como los ocho bits superiores. Así pues, una interrupción en el dígito binario 1 del ISR en *0xA0* será tratada como la interrupción 9 del sistema. El segundo bit de PIC1 no es utilizado ya que sirve para encadenar las interrupciones del controlador PIC2, por lo tanto cualquier interrupción del controlador PIC2 se pasa al bit 2 del controlador PIC1.

El controlador de interrupción programable 8259 (PIC en la placa base) maneja todas las interrupciones hardware. Estos controladores toman las señales de los dispositivos y los convierten a las interrupciones específicas en el procesador.

Los IRQ o **i**nterrupt **r**e**q**uest (Pedido de Interrupción), son las notificaciones de las interrupciones enviadas desde los dispositivos hardware a la CPU, en respuesta a la IRQ, la CPU salta a una dirección – una **r**utina de **s**ervicio de **i**nterrupción (ISR), comúnmente llamada Interrupt handler (Manejador de interrupciones) - Que se encuentra como una función dentro del software manejador de ese dispositivo formando parte del núcleo. Así, una función manejadora de interrupciones es una función del núcleo que ejecuta el servicio de esa interrupción.

Los IRQ se encuentran numerados, y cada dispositivo hardware se encuentra asociado a un número IRQ. En la arquitectura IBM PC y compatibles, por ejemplo, IRQ 0 se encuentra asociado al reloj o temporizador, el cual genera 100 interrupciones por segundo, disquete el 6, los discos IDE la 14 y 15. Se puede compartir un IRQ entre varios dispositivos.

La siguiente figura, muestra las interrupciones hardware y su correspondiente puerto en el Controlador Programable de Interrupciones (PIC). No se deben confundir los números IRQ entradas al controlador con los números de la interrupción que son las entradas en la tabla de interrupciones. Los PIC se pueden programar para generar diversos números de interrupción para cada IRQ.

Los Controladores también controlan la prioridad de las interrupciones. Por ejemplo, el reloj (en IRQ 0) tiene una prioridad más alta que el teclado (IRQ 1). Si el procesador está atendiendo una interrupción del reloj, el PIC no generará una interrupción para el teclado hasta que ISR del reloj reajusta el PIC. Por otra parte, el reloj puede interrumpir ISR del teclado. El PICs se puede programar para utilizar una variedad de esquemas de la prioridad, pero no se suele hacer esto.

Se debe de tener en cuenta que el IRQ 2 del primer PIC, valida o invalida las entradas del Segundo PIC (8 a 15).

Algunas interrupciones son fijadas por convenio en la configuración del PC, así es que los manejadores de los dispositivos solicitan simplemente la interrupción cuando se inicializan. Por ejemplo esto es lo que lo hace el manejador de disquete, solicita siempre la IRQ 6.

| Interrupción | IRQ | Descripción |
|---|---|---|
| 00H | - | división por cero o desbordamiento |
| 02H | - | NMI (interrupción no-enmascarable) |
| 04H | - | desbordamiento (EN) |
| 08H | 0 | Temporizador del sistema |
| 09H | 1 | Teclado |
| 0AH | 2 | Interrupción del segundo PIC |
| 0BH | 3 | COM2 |
| 0CH | 4 | COM1 |
| 0DH | 5 | LPT2 |
| 0EH | 6 | disquete |
| 0FH | 7 | LPT1 |
| 70H | 8 | Reloj |
| 71H | 9 | I/o general |
| 72H | 10 | I/o general |

| | | |
|---|---|---|
| 73H | 11 | I/o general |
| 74H | 12 | I/o general |
| 75H | 13 | Coprocesador |
| 76H | 14 | Disco duro |
| 77H | 15 | I/o general |

## 3.3 Interrupciones hardware en Linux.

Una de las principales tareas del sistema de manejo de interrupciones es llevar las diferentes interrupciones a los códigos de manejo de esas interrupciones.
Cuando se activa el contacto 6 del controlador de interrupciones, se debe reconocer cual es la interrupción asociada a ese contacto, por ejemplo el controlador del dispositivo disquete, por lo tanto el sistema de manejo de interrupciones debe encaminar a la rutina que trata esta interrupción, para ello Linux proporciona un conjunto de estructuras de datos y tablas, y un conjunto de funciones que las inicializan y las manejan.
El sistema de interrupciones es muy dependiente de la arquitectura, Linux en la medida de lo posible, tratará de que sea independiente de la máquina en la que reside el sistema, para ello el sistema de interrupciones se va a implementar mediante una serie de estructuras de datos y funciones en lenguaje C que facilitarán la portabilidad. Veamos una presentación de las estructuras de datos implicadas, resaltando las más importantes y describiendo los campos de cada una. Posterior a eso se seguirá con la inicialización de estas estructuras que soportan las interrupciones, esta parte es fuertemente dependiente de la arquitectura, nos basaremos en i386. Finalmente se expondrá el flujo dentro del núcleo que sigue una interrupción hardware, desde que se origina en el dispositivo hardware hasta que se atiende por el manejador de la interrupción.

### 3.3.1 Estructuras de datos para soportar el sistema de interrupciones hardware

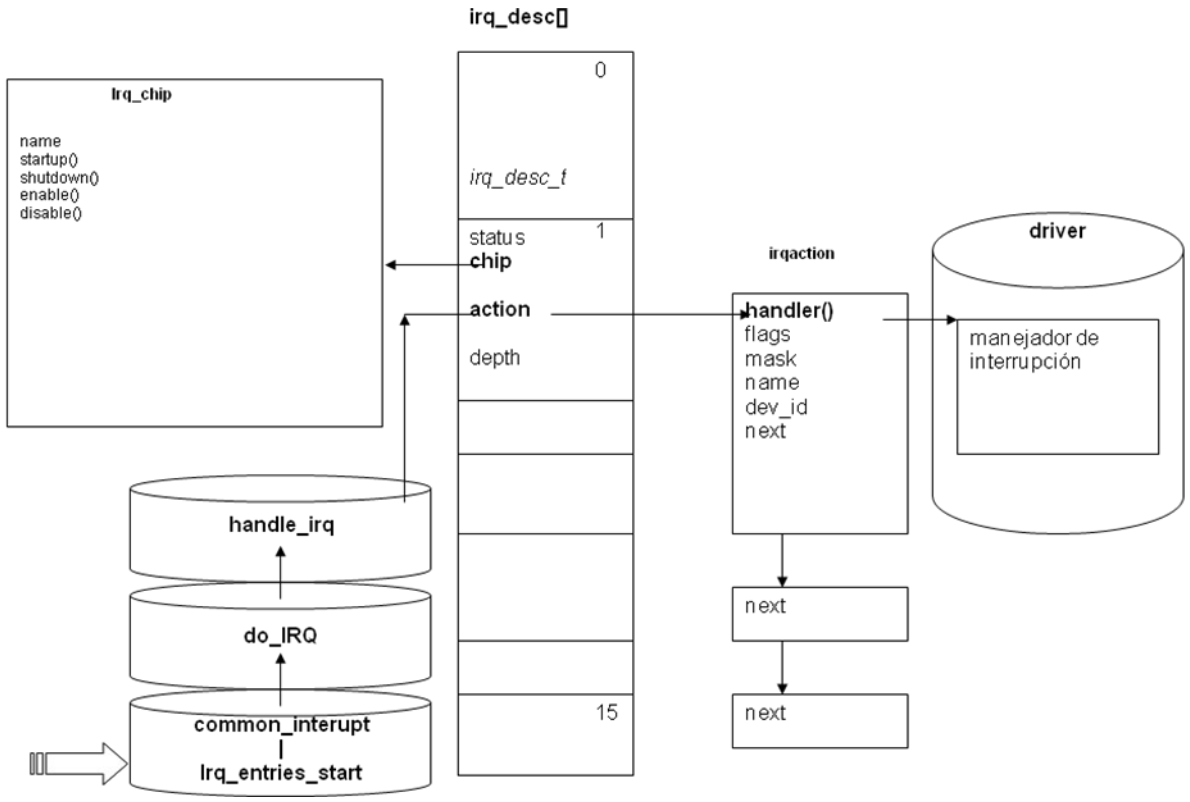
Estudiaremos las estructuras de datos del sistema de interrupciones:

**irqaction** almacena la dirección de la función de manejo de interrupciones.

**irq_chip** contiene las funciones que manejan un controlador de interrupciones particular, es dependiente de la arquitectura.

**irq_desc** vector con una entrada para cada una de las interrupciones que pueden ser atendidas.

Estructuras de datos para el sistema de interrupciones




**irqaction**

En el fichero **include/linux/interrupt.h** se define la estructura **struct irqaction** que almacena un puntero a la dirección de la función que hay que llevar a cabo cada vez que una interrupción se produce. Esta estructura es independiente del hardware. El código es:

```
struct irqaction {
        irq_handler_t handler;
        unsigned long flags;
        cpumask_t mask;
        const char *name;
        void *dev_id;
        struct irqaction *next;
        int irq;
        struct proc_dir_entry *dir;
};
```

Veamos el significado de los campos

**handler** Este campo contiene un puntero a la función que atiende a la interrupción, esta función se encuentra dentro del manejador del dispositivo que interrumpe, este campo es inicializado por el manejador.

**flags** Contiene información sobre cómo tratar en ciertas situaciones la interrupción. Los valores que puede tomar están declarados en el archivo **/include/linux/interrupt.h** y son los siguientes:

**SA_INTERRUPT** Indica que esta interrupción puede ser interrumpida por otra.

**SA_SAMPLE_RANDOM** Esta interrupción puede ser considerada de naturaleza aleatoria, esto puede ser útil cuando se necesita una semilla de aleatoriedad, etc.
**SA_SHIRQ** Esta IRQ puede ser compartida por diferentes **struct irqaction**

**mask** Este campo no se usa en la arquitectura i386

**name** Un nombre asociado con el dispositivo que genera la interrupción. Como una misma IRQ puede ser compartida por diferentes dispositivos, esto nos puede ayudar a distinguir los dispositivos. Se puede leer en el fichero /proc/interrupts.

**dev_id** Todos y cada uno de los dispositivos implementados en Linux tiene un número identificador único grabado por el fabricante. De esta forma, este campo define qué dispositivo genera la interrupción. Estos identificadores se encuentran definidos en ficheros cabeceras.

#define PCI_DEVICE_ID_S3_868 0x8880
#define PCI_DEVICE_ID_S3_928 0x88b0

**next** Este campo contiene un puntero que apunta a la próxima **struct irqaction** en la cola, esto sólo tiene sentido cuando la IRQ se encuentra compartida, así pues, en la mayoría de los casos estará a NULL.

**irq** Numero de la línea IRQ.

**dir** Indica un puntero al descriptor del directorio /proc/irq/n asociado con la irq n.

### irq_chip

Es una estructuras dependiente de la arquitectura, para Intel está definida en el fichero **include/linux/irq.h**

su código es:

/* Describe el decodificador de interrupciones a bajo nivel y las funciones que lo manejan */

```
/**
  98struct irq_chip {
  99        const char      *name;
 100        unsigned int    (*startup)(unsigned int irq);
 101        void            (*shutdown)(unsigned int irq);
 102        void            (*enable)(unsigned int irq);
 103        void            (*disable)(unsigned int irq);
 104
 105        void            (*ack)(unsigned int irq);
 106        void            (*mask)(unsigned int irq);
 107        void            (*mask_ack)(unsigned int irq);
 108        void            (*unmask)(unsigned int irq);
 109        void            (*eoi)(unsigned int irq);
 110
 111        void            (*end)(unsigned int irq);
 112        void            (*set_affinity)(unsigned int irq, cpumask_t dest);
 113        int             (*retrigger)(unsigned int irq);
```

```
 114        int             (*set_type)(unsigned int irq, unsigned int flow_type);
 115        int             (*set_wake)(unsigned int irq, unsigned int on);
 116
 117        /* Currently used only by UML, might disappear one day.*/
 118#ifdef CONFIG_IRQ_RELEASE_METHOD
 119        void            (*release)(unsigned int irq, void *dev_id);
 120#endif
 121        /*
 122         * For compatibility, ->typename is copied into ->name.
 123         * Will disappear.
 124         */
 125        const char      *typename;
 126};
```

Esta estructura contiene todas las operaciones específicas de un determinado decodificador de interrupciones (8259) tales como el activado o desactivado de irqs, etc. Los campos se detallan a continuación:

**name** Un nombre para /proc/interrupts
**startup** Activa una entrada irq para que pueda interrumpir.
**shutdown** Deshabilita una entrada irq.
**enable y disable** Igual que **startup/shutdown** respectivamente.
**ack** Comienzo de una nueva interrupción.
**mask umask** Pone quita una máscara para una interrupción.
**eoi end** Final de una interrupción.
**set-affinity** Para trabajar con varias CPUs (smp).
**retrigger** Vuelve a enviar una petición irq a la cpu.
**set_type** Define el modo de disparo (nivel, flanco) de la Irq.
**set_wake** Permite despertar a la unidad de alimentación al recibir una irq.

### irq_desc

Es un vector de estructuras de tipo **irq_desc_t**, descriptor de interrupciones dependiente de la arquitectura, para Intel está definido en el fichero **incluye/linux/irq.h**. Por cada interrupción hardware del sistema habrá un elemento en el **array irq_desc[ ]**.

Su código es el siguiente:

```
struct irq_desc {
 156        unsigned int            irq;
 157        irq_flow_handler_t      handle_irq;
 158        struct irq_chip         *chip;
 159        struct msi_desc         *msi_desc;
 160        void                    *handler_data;
 161        void                    *chip_data;
 162        struct irqaction        *action;        /* IRQ action list */
 163        unsigned int            status;         /* IRQ status */
 164
 165        unsigned int            depth;          /* nested irq disables
*/
 166        unsigned int            wake_depth;     /* nested wake enables
*/
 167        unsigned int            irq_count;      /* For detecting broken
IRQs */
 168        unsigned int            irqs_unhandled;
```

```
 169        unsigned long           last_unhandled; /* Aging timer for
unhandled count */
 170        spinlock_t              lock;
 171#ifdef CONFIG_SMP
 172        cpumask_t               affinity;
 173        unsigned int            cpu;
 174#endif
 175#ifdef CONFIG_GENERIC_PENDING_IRQ
 176        cpumask_t               pending_mask;
 177#endif
 178#ifdef CONFIG_PROC_FS
 179        struct proc_dir_entry   *dir;
 180#endif
 181        const char              *name;
 182} ____cacheline_internodealigned_in_smp;

 184
 185extern struct irq_desc irq_desc[NR_IRQS];
 186
```

Sus campos son:
**Irq**: número de la interrupción para este descriptor
**handle_irq**: función manejadora de la interrupción [si es NULL, __do_IRQ()]
**chip**: **funciones de bajo nivel para el decodificador hardware**
**msi_desc**: MSI descriptor
**handler_data**: datos usados por el decodificador para una irq
**chip_data**: datos usados por el decodificador para una plataforma
**action: identifica a la rutina que maneja la interrupción, señala la cabeza de la lista de irqactions. Una irq puede ser compartida por varios equipos, existe un nodo de la lista para cada equipo.**
**status**: información del estado de la interrupción
**depth**: cantidad de interrupciones anidadas a la misma posición del vector
**wake_depth**: permite anidamiento para múltiples set_irq_wake()
**irq_count**: contador de ocurrencias de interrupciones por esa línea, para detectar fallos
**irqs_unhandled**: contador de interrupciones deshabilitadas, solo para estadísticas
**last_unhandled**: cronómetro para interrupciones espúreas
**lock**: bloqueo para SMP
**affinity**: IRQ afinidad para trabajar con SMP
**cpu**: indice para balanceo de carga en cpu
**pending_mask**: interrupciones pendientes de rebalanceo
**dir**: entrada en el sistema de ficheros /proc/irq/
**affinity_entry**: entrada en /proc/irq/smp_affinity
**name**: nombres para /proc/interrupts

**status** Este campo representa el estado en el que se encuentra actualmente la línea de interrupción IRQ, estos estados están definidos en el mismo archivo. Sus estados son los siguientes:

**#define** IRQ_INPROGRESS 1 /* IRQ en uso, no utilizar */
**#define** IRQ_DISABLED 2 /*IRQ desactivada, no utilizar */

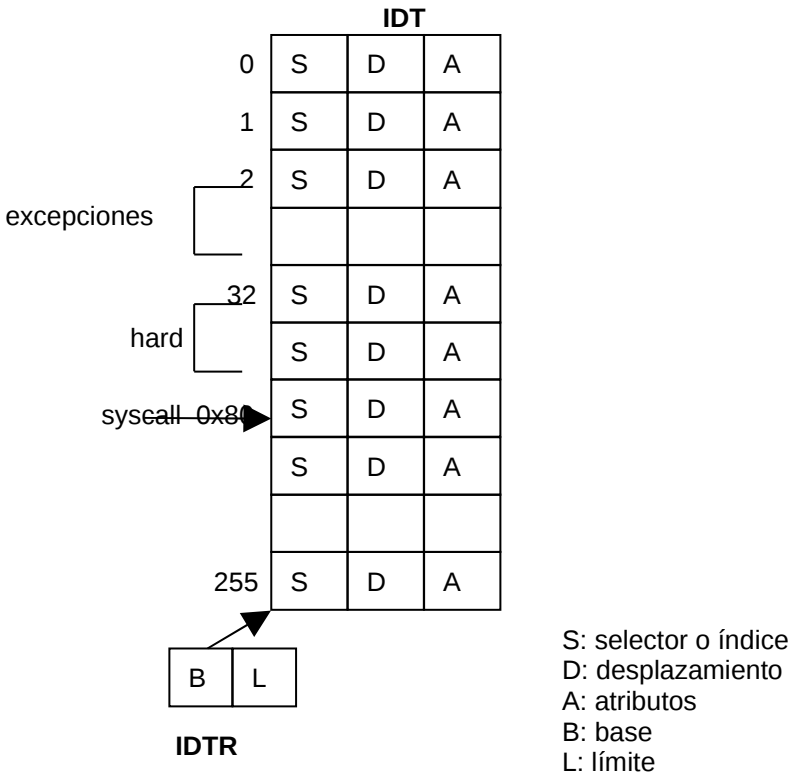
**#define** IRQ_PENDING 4 /* IRQ pendiente, ha sido reconocida por el PIC pero no ha sido atendida por el núcleo */
**#define** IRQ_REPLAY 8 /* IRQ reintenta pero todavía no atendida */
**#define** IRQ_AUTODETECT 16 /* IRQ esta siendo auto detectada */
**#define** IRQ_WAITING 32 /* IRQ no lista para auto detección */

**Tabla descriptora de interrupciones idt_table**

La tabla descriptora de las interrupciones IDT, es una tabla de 256 entradas que junto con la tabla global de descriptores GDT, nos va a llevar dentro del núcleo a las rutinas de manejo de interrupciones que se encuentran en el manejador del dispositivo. Definida en arch/x86/kernel/traps.c, como gate_desc idt_table[256]. Y cada entrada de la tabla es una estructura definida en,
arch/x86/include/asm/desc_defs.h

Como:

```
/* 16byte gate */
  45struct gate_struct64 {
  46        u16 offset_low;
  47        u16 segment;
  48        unsigned ist : 3, zero0 : 5, type : 5, dpl : 2, p : 1;
  49        u16 offset_middle;
  50        u32 offset_high;
  51        u32 zero1;
  52} __attribute__((packed));
```



Las primeras entradas de la tabla están ocupadas por las excepciones.

Las interrupciones hardware, comienzan en la entrada 32 de la tabla, ocupan 16 entradas van desde 0x20 y hasta 0x2f.

La interrupción software ocupa la entrada 128, (0x80 en exadecimal).

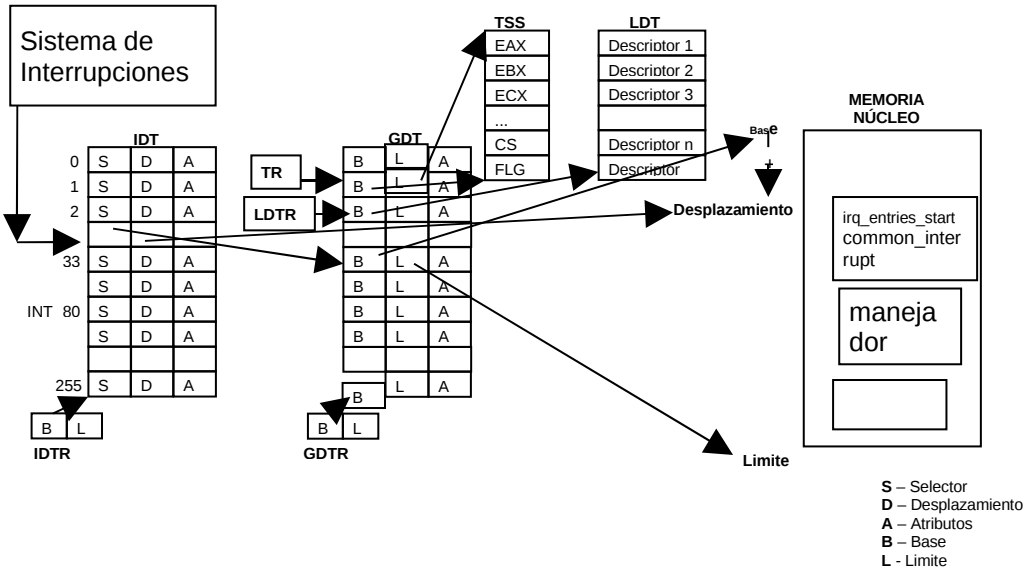
**IDTR** es un registro dentro de la CPU que contiene la base y el límite de esta tabla en memoria.

**S** con este campo seleccionamos una entrada dentro de la tabla GDT, que nos dará la base y el límite del núcleo en memoria.

**D** es el desplazamiento que sumado a la base del núcleo no lleva a la función de entrada para una determinada interrupción.

### Tabla global de descriptores GDT

Es una tabla única para el sistema con 32 entradas de descriptores de segmentos, las entradas 12 y 13 de esta tabla contienen la base y el límite del código y datos del núcleo, esta definida en include/asm-i386/segment.h.



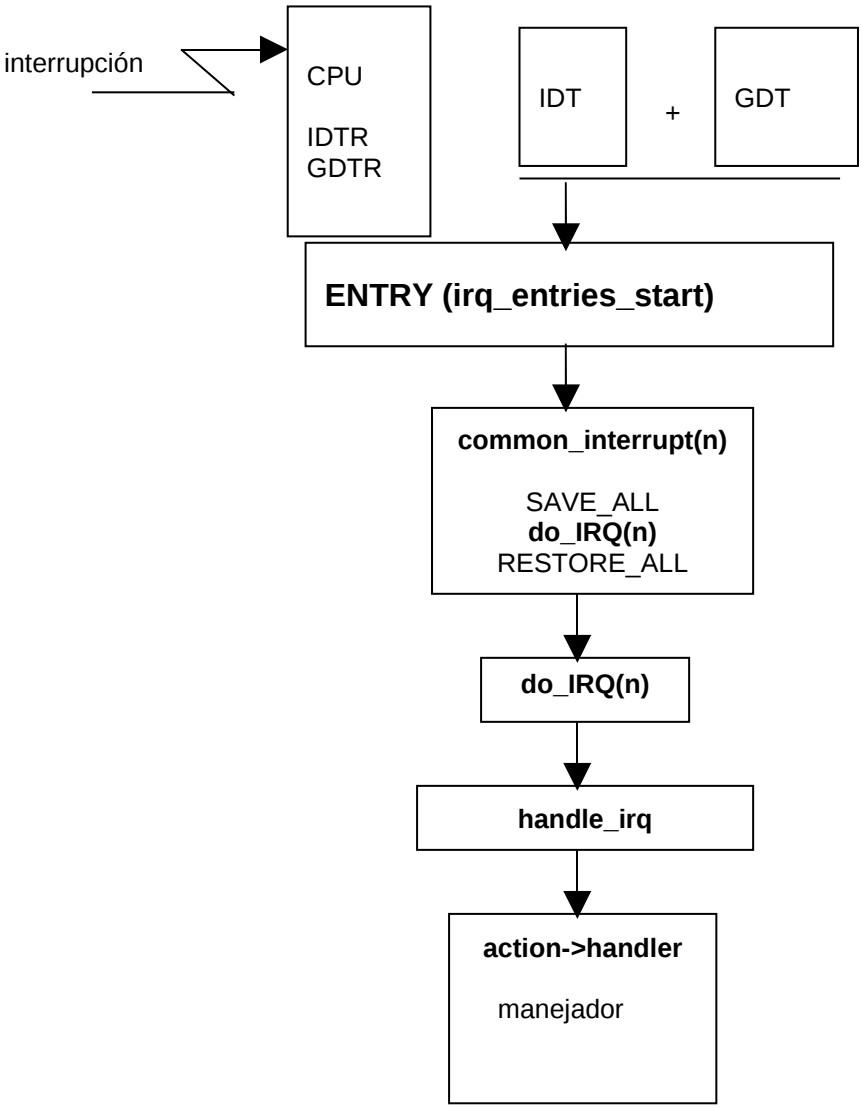

## 3.4 Procesado de interrupciones

Una vez que las estructuras de datos mencionadas en la sección anterior (la tabla de interrupciones (IDT) y demás estructuras de datos que conforman el sistema de interrupciones de Linux) están debidamente inicializadas y todos los manejadores de dispositivos cargados en el sistema e inicializados, el sistema está en disposición de atender cualquier tipo de interrupción que se genere por los dispositivos que usen este mecanismo para comunicarse con la CPU. De esta forma, la IDT estará rellena con el desplazamiento que nos lleva a la función **common_interrupt** pasándole como parámetro el número de la interrupción que se activó.

Esto es así porque en la IDT solo se generan saltos a **common_interrupt** que es una función que, haciendo uso del manejador de interrupciones asociado a cada irq en **irq_desc** ejecuta la acción necesaria mediante la estructura **irqaction**. Otro enfoque sería rellenar directamente la IDT con el salto a la función manejadora, pero esto generaría un código altamente dependiente de la arquitectura en capas altas del sistema.

### Flujo general

A continuación se expondrá el flujo general sin detalles y posteriormente se expondrá más detenidamente cada una de las etapas.




1. El decodificador de interrupciones recibe una interrupción (la patilla IRQ a nivel alto), este interrumpe a la CPU, que automáticamente mediante la entrada que corresponde a esta interrupción y con la ayuda de la IDT y la GDT nos lleva a **ENTRY (irq_entries_start)** dentro de arch/i386/kernel/entry.S, que salta a **common_interrupt**.

2. **common_interrupt** despues de guardar el estado del proceso **SAVE_ALL** llama a **do_irq** pasándole el número de interrupción para ejecutar la función manejadora

de la interrupción y posteriormente recupera el estado del proceso interrumpido **RESTART_ALL** saltando a **ret_from_intr**.

3. **do_IRQ** llama a **handle_IRQ**.

4. **handle_IRQ** llama a las funciones necesarias para atender la interrupción **action->handler** dentro del manejador.

5. **action->handler** se ejecuta la función manejadora de la interrupción dentro del manejador (driver) del dispositivo que interrumpió.

6. **ret_from_intr** restaura el estado del proceso interrumpido RESTORE_ALL y continúa con la ejecución normal.

**Flujo proceso a proceso**

La función set_intr_gate inicializa la IDT, colocando el desplazamiento que nos lleva al punto de entrada **ENTRY (irq_entries_start)**, dentro del fichero Linux/arch/x86/kernel/entry_32.S.

**irq_entries_start** nos lleva a **common_interrupt** pasándole como parámetro el número de interrupción en el stack cambiado de signo.

```
ENTRY(irq_entries_start)
 587         RING0_INT_FRAME
 588vector=0
 589.rept NR_IRQS
 590         ALIGN
 591 .if vector
 592         CFI_ADJUST_CFA_OFFSET -4
 593 .endif
 5941:       pushl $~(vector)
 595         CFI_ADJUST_CFA_OFFSET 4
 596         jmp common_interrupt
 597 .previous
 598         .long 1b
 599 .text
 600vector=vector+1
 601.endr
 602END(irq_entries_start)
```

Guarda el número de la interrupción en el stack, forma de pasar parámetros a una función, y en negativo para que no exista conflicto con el número de señales
pushl **$~(vector)**

y salta a la función jmp **common_interrupt**

**common_interrupt,** también se encuentra en el fichero entry_32.S, es una función en ensamblador que realiza los tres pasos fundamentales para atender una interrupción: almacenar el estado del proceso que se interrumpe (**SAVE_ALL**); llamar a la función que atiende la interrupción (mediante la llamada a **do_IRQ** pasándole en eax el número de la interrupción); y restablece el proceso interrumpido (**RESTORE_ALL** saltando a **ret_from_intr)**.

```
    common_interrupt:
 614        SAVE_ALL
 615        TRACE_IRQS_OFF
 616        movl %esp,%eax
/ * Llama a do_IRQ pasandole en eax el numero de la interrupción */
 617        call do_IRQ
 618        jmp ret_from_intr
 619ENDPROC(common_interrupt)


 244ret_from_intr:
 245        GET_THREAD_INFO(%ebp)
 246check_userspace:
 247        movl PT_EFLAGS(%esp), %eax      # # mezcla EFLAGS con CS
 248        movb PT_CS(%esp), %al
 249        andl $(VM_MASK | SEGMENT_RPL_MASK), %eax
 250        cmpl $USER_RPL, %eax
 251        jb resume_kernel        # # pregunta si vuelve al modo núcleo virtual
 252
 253ENTRY(resume_userspace)    # punto de entrada para volver a modo usuario
 254        DISABLE_INTERRUPTS(CLBR_ANY)     # # deshabilita las interrupciones
 255
 256
 # Comprueba si tiene trabajo pendiente por tener  pendiente otra interrupción
 # o tiene señales pendientes y necesita volver a asignar la CPU

 257        movl TI_flags(%ebp), %ecx
 258        andl $_TIF_WORK_MASK, %ecx      # is there any work to be done
on
 259                                        # int/exception return?
 260        jne work_pending
 261        jmp restore_all   # recupera el estado del proceso interrumpido


restore_all:
 442        movl PT_EFLAGS(%esp), %eax      # mezcla EFLAGS, SS and CS
 443        # Warning: PT_OLDSS(%esp) contains the wrong/random values if
we
 444        # are returning to the kernel.
 445        # See comments in process.c:copy_thread() for details.
 446        movb PT_OLDSS(%esp), %ah
 447        movb PT_CS(%esp), %al
 448        andl $(X86_EFLAGS_VM | (SEGMENT_TI_MASK << 8) |
SEGMENT_RPL_MASK), %eax
 449        cmpl $((SEGMENT_LDT << 8) | USER_RPL), %eax
 450        CFI_REMEMBER_STATE
 451        je ldt_ss                       # returning to user-space with
LDT SS
 452restore_nocheck:
```

```
 453        TRACE_IRQS_IRET
 454restore_nocheck_notrace:
 455        RESTORE_REGS  # recupera el estado de la máquina
 456        addl $4, %esp                   # skip orig_eax/error_code
 457        CFI_ADJUST_CFA_OFFSET -4
```

restore_all:

**RESTORE_REGS**

**do_IRQ** , se encuentra en el fichero arch/x86/kernel/irq.c, maneja todas las interrupciones normales de los dispositivos IRQ's (las interrupciones especiales producidas por tener varias CPU's (SMP), tienen un tratamiento especifico. Recibe el número de irq a través del registro eax y un puntero a una estructura pt_regs donde se han guardado los registros. Llama a la función **handle_irq(irq, desc)**, que contiene el manejador de esa interrupción.

```
 199unsigned int do_IRQ(struct pt_regs *regs)
 200{
 201        struct pt_regs *old_regs;
 202        /* high bit used in ret_from_ code */
 203        int overflow;
```

/* el numero de interrupción se encuentra en los bits más bajos de eax */

```
 204        unsigned vector = ~regs->orig_ax;
 205        struct irq_desc *desc;
 206        unsigned irq;
 207
 208
 209        old_regs = set_irq_regs(regs);
 210        irq_enter();
 211        irq = __get_cpu_var(vector_irq)[vector];
 212
 213        overflow = check_stack_overflow();
 214
```

/* accede a la posición del vector irq_desc para esta interrupción */

```
 215        desc = irq_to_desc(irq);
 216        if (unlikely(!desc)) {
 217                printk(KERN_EMERG "%s: cannot handle IRQ %d vector %#x
cpu %d\n",
 218                                        __func__, irq, vector,
smp_processor_id());
 219                BUG();
 220        }
 221
 222        if (!execute_on_irq_stack(overflow, desc, irq)) {
 223                if (unlikely(overflow))
 224                        print_stack_overflow();
```

/* llamada a la función

Llama a la función de alto nivel manejadora de la interrupción, con el número de irq y un puntero a la entrada del vector de interrupciones **irq_desc** , la estructura irq_desc mediante el campo **action** apunta al campo **irqaction->handler** que tiene grabada la dirección de la función. Este campo ha sido previamente inicializado por el driver del dispositivo.

```
 225                desc->handle_irq(irq, desc);
```

```
 226         }
 227
 228         irq_exit();
 229         set_irq_regs(old_regs);
 230         return 1;
 231}
```

Por último la función que entra en juego es el manejador de la interrupción que es una función que se encuentra dentro del código driver del dispositivo.

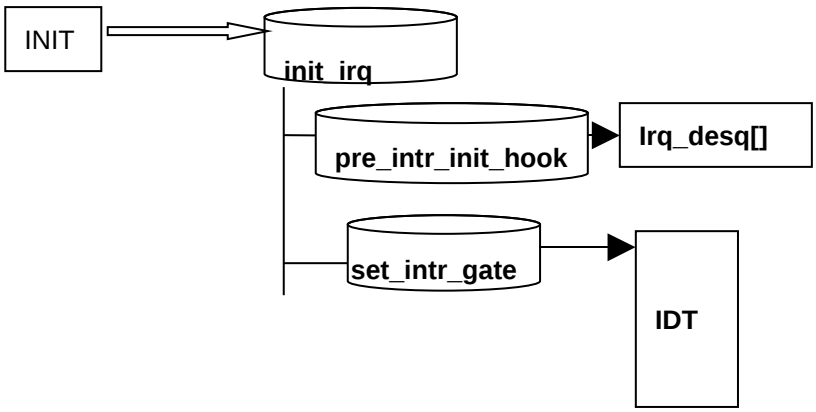
## 3.5 Procedimientos de Inicialización

Para que los dispositivos puedan utilizar el sistema de interrupciones, el sistema al arrancar debe inicializar mediante unos procedimientos, que se encuentran en **/arch/x86/kernel/irqinit.c,** las estructuras de datos relacionas con las interrupciones.

**init_IRQ().**

Esta función es llamada por el programa de inicio del sistema **init start_kernel ()**, en **init/main.c.** Esta función pone a punto la tabla de descriptores de interrupciones (IDT). Se encuentra definida en el fichero arch/i386/kernel/i8259.c

Primeramente inicializa el vector de interrupciones hardware **irq_desc[0..15].**
Seguidamente inicializa la tabla IDT, llamando a la función **set_intr_gate()**.



A continuación se muestra el código de esta función, básicamente se ha mantenido la estructura pero quitando el soporte a otras plataformas y el soporte para varios procesadores.

```
 385void init_IRQ(void) __attribute__((weak, alias("native_init_IRQ")));
 386
 387void __init native_init_IRQ(void)
 388{
 389        int i;
 390
 391        /* se inicializa el vector irq_desc[0..15]  */
 392        pre_intr_init_hook();
 393
 394        /*
 395         * Inicializa la tabla IDT completa dando valores por defecto,
 396         * algunas entradas se reescribirán posteriormente
 397         * como el caso de interrupciones con varios procesadores SMP
 398         */
 399        for (i = 0; i < (NR_VECTORS - FIRST_EXTERNAL_VECTOR); i++) {
 400                int vector = FIRST_EXTERNAL_VECTOR + i;
 401                if (i >= NR_IRQS)
 402                        break;
 403                if (vector != SYSCALL_VECTOR)
 404                        set_intr_gate(vector, interrupt[i]);
 405        }
 406      /* interrupt[i] desplazamiento a la función donde salta cuando
          * llega la interrupción, IRQ 0xNN_interrupt */

 407        /* realiza alguna inicialización especial dependiente
 408         * de la arquitectura
 409         */
 410        intr_init_hook();
 411
 412        /*
 413         * Si existe coprocesador matemático
 414         * activa la entrada irq13 correspondiente
 415         */
 416        if (boot_cpu_data.hard_math && !cpu_has_fpu)
 417                setup_irq(FPU_IRQ, &fpu_irq);
 418
 419        irq_ctx_init(smp_processor_id());
 420}
```

### set_intr_gate

Por medio de la función **set_intr_gate** se rellena la tabla de descriptores de interrupciones (**IDT**). Básicamente esta es la parte en la que se realiza el "cableado" de las interrupciones dentro del sistema, a partir de este momento, cualquier interrupción será atendida. Se encuentra en el fichero arch/x86/include/asm/desc.h.

```
static inline void set_intr_gate(unsigned int n, void *addr)
```

```
 318{
 319        BUG_ON((unsigned)n > 0xFF);
 320        _set_gate(n, GATE_INTERRUPT, addr, 0, 0, __KERNEL_CS);
 321}
_
```

**n** va apuntando a las distintas entradas de la IDT.
**DESCTYPE_INT** tipo de descriptor interrupcion
**addr** es el desplazamiento que le va a llevar a la función **irq_entries_start**.
**KERNEL_CS** selector que apunta a la entrada de la GDT donde está la base del código del núcleo.

llama a la macro **_set_gate** en Linux/arch/x86/asm/desc.h

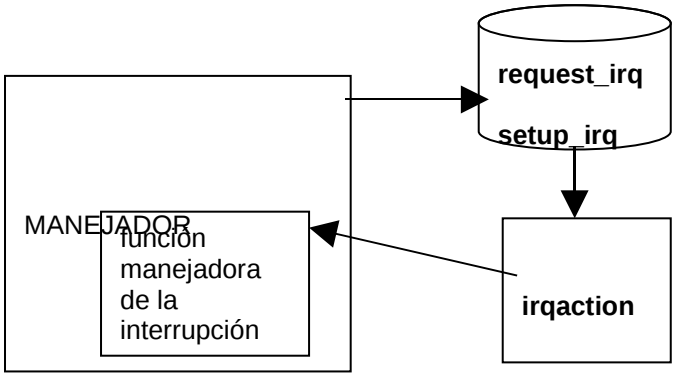
```
static inline void _set_gate(int gate, unsigned type, void *addr,
 300                             unsigned dpl, unsigned ist, unsigned seg)
 301{
 302        gate_desc s;
 303        pack_gate(&s, type, (unsigned long)addr, dpl, ist, seg);
 304        /*
 305         * does not need to be atomic because it is only done once at
 306         * setup time
 307         */
 308        write_idt_entry(idt_table, gate, &s);
 309}


static inline void pack_gate(gate_desc *gate, unsigned char type,
  66                             unsigned long base, unsigned dpl, unsigned
flags,
  67                             unsigned short seg)
  68{
  69        gate->a = (seg << 16) | (base & 0xffff);
  70        gate->b = (base & 0xffff0000) |
  71                  (((0x80 | type | (dpl << 5)) & 0xff) << 8);
  72}


static inline void native_write_idt_entry(gate_desc *idt, int entry,
 116                                          const gate_desc *gate)
 117{
 118        memcpy(&idt[entry], gate, sizeof(*gate));
 119}
```

**request_irq**

Una vez que el sistema de interrupciones está inicializado, cuando se vayan cargando los diferentes manejadores de dispositivos éstos harán uso de las funciónes **request_irq** y **setup_x86_irq** para colocar la dirección de la función manejadora de interrupción propia y activar la interrupción.




**request_irq** es llamado por los manejadores, en su inicio, para colocar la dirección de la función manejadora de interrupción. Definida en kernel/irq/manage.c.

Crea un nuevo nodo en la lista **irqaction** y llena sus campos con los valores suministrados para una IRQ.

```
 request_irq – Asigna una línea de interrupción
 irq: Línea de interrupción a asignar
 handler: Función que se tiene que ejecutar cuando ocurre la IRQ
 irqflags: Flags del estado de la interrupción
 devname: Un nombre para el dispositivo asociado a esa interrupción
 dev_id: Identificador del dispositivo pasado a la función que trata la
interrupción

*       Esta función asigna recursos a la interrupción y habilita la
*       línea de interrupción y el tratamiento de la IRQ. Una vez
*       ejecutada esta función el manejador ya puede ser invocado.
*       Ya que la función manejadora puede borrar cualquier
*       interrupción, se tiene que inicializar el hardware
*       y el manejador de interrupciones en el orden adecuado.
*
*       Dev_id debe ser un identificador único. Normalmente se toma
*       como identificador la dirección de la estructura device. Ya que
*       el manejador recibe este valor tiene sentido usar este valor.
*
*       Si la interrupción es compartida, el dev_id debe ser no NULL
*       esto se require cuando se libera la interrupción.
*
*       Flags:
*
*       IRQF_SHARED             Interrupción compartida
*       IRQF_DISABLED   Desactivar interrupción cuando se está procesando
*       IRQF_SAMPLE_RANDOM      La interrupción se puede usar para random
*       IRQF_TRIGGER_*      Especifica si se activa por flancos o por nivel


int request_irq(unsigned int irq, irq_handler_t handler,
 670                unsigned long irqflags, const char *devname, void
*dev_id)
 671{
 672        struct irqaction *action;
```

```
 673         struct irq_desc *desc;
 674         int retval;
 675
 676#ifdef CONFIG_LOCKDEP
 677         /*
 678          * Lockdep wants atomic interrupt handlers:
 679          */
 680         irqflags |= IRQF_DISABLED;
 681#endif
 682         /*
 683          * Sanity-check: shared interrupts must pass in a real dev-ID,
 684          * otherwise we'll have trouble later trying to figure out
 685          * which interrupt is which (messes up the interrupt freeing
 686          * logic etc).
 687          */
/*
         * Comprueba que si es una interrupción compartida tiene,
         * que tener un dispositivo dev_id asociado.
         * Si se sobrepasa el nº de IRQ.
         * Comprueba que existe una función para tratar la interrupción
         */

 688         if ((irqflags & IRQF_SHARED) && !dev_id)
 689                 return -EINVAL;
 690
 691         desc = irq_to_desc(irq);
 692         if (!desc)
 693                 return -EINVAL;
 694
 695         if (desc->status & IRQ_NOREQUEST)
 696                 return -EINVAL;
 697         if (!handler)
 698                 return -EINVAL;
 699
/* con la función kmalloc asigna memoria dinámicamente para el nuevo nodo */
 700         action = kmalloc(sizeof(struct irqaction), GFP_ATOMIC);
 701         if (!action)
 702                 return -ENOMEM;
 703
/* llena los campos de action con el manejador, flags, nombre, identificador */
 704         action->handler = handler;
 705         action->flags = irqflags;
 706         cpus_clear(action->mask);
 707         action->name = devname;
 708         action->next = NULL;
 709         action->dev_id = dev_id;
 710
/* añade el nuevo nodo a la lista con setup_irq
 711         retval = __setup_irq(irq, desc, action);
/* si hay error libera el nodo */
 712         if (retval)
 713                 kfree(action);
 714
 715#ifdef CONFIG_DEBUG_SHIRQ
 716         if (irqflags & IRQF_SHARED) {
 717                 /*
 718                  * It's a shared IRQ -- the driver ought to be prepared
for it
 719                  * to happen immediately, so let's make sure....
```

```
 720                 * We disable the irq to make sure that a 'real' IRQ
doesn't
 721                 * run in parallel with our fake.
 722                 */
 723                unsigned long flags;
 724
 725                disable_irq(irq);
 726                local_irq_save(flags);
 727
 728                handler(irq, dev_id);
 729
 730                local_irq_restore(flags);
 731                enable_irq(irq);
 732        }
 733#endif
 734        return retval;
 735}
 736EXPORT_SYMBOL(request_irq);
```

**setup_irq**

Veamos el código de la función **setup_irq** que se encuentra definido en kernel/irq/manage.c.

La función tiene como parámetros el número de interrupción irq y un nodo de estructura **irqaction**, registra la interrupción insertando el nodo en la lista y rellenando la estructura **irq_desc** con la interrupción deseada.

La primera parte tiene que ver con la fuente de números aleatorios, no vamos a entrar en ella. Posteriormente se comprueba si la interrupción puede compartir irq en el caso de que haya ya alguna ocupando el espacio y seguidamente se inserta. La inserción se realizará al final de la cola (si la interrupción es compartida) o al principio (si es la primera que se inserta). Finalmente se rellenan los campos que faltan en el caso de que sea la primera vez que se inserta la interrupción.

```
/*
* Función para añadir un nodo a irqaction, contiene
* interrupciones que dependen de la arquitectura hardware.
*/

static int
 393 __setup_irq(unsigned int irq, struct irq_desc * desc, struct irqaction *
*new)
 394{
 395        struct irqaction *old, **p;
 396        const char *old_name = NULL;
 397        unsigned long flags;
 398        int shared = 0;
 399        int ret;
 400
 401        if (!desc)
 402                return -EINVAL;
 403
 404        if (desc->chip == &no_irq_chip)
 405                return -ENOSYS;
 406        /*
```

```
        /*
         * Algunos manejadores como serial.c usan
         * request_irq()frecuentemente, por lo que hay que
         * tener cuidado de no interferir con un sistema en
         */ ejecución.

 407          * Some drivers like serial.c use request_irq() heavily,
 408          * so we have to be careful not to interfere with a
 409          * running system.
 410          */
 411         if (new->flags & IRQF_SAMPLE_RANDOM) {
 412                 /*
 413                  * This function might sleep, we want to call it first,
 414                  * outside of the atomic block.
 415                  * Yes, this might clear the entropy pool if the wrong
 416                  * driver is attempted to be loaded, without actually
 417                  * installing a new handler, but is this really a
problem,
 418                  * only the sysadmin is able to do this.
 419                  */
/* Esta función inicializa el /dev/random  y /dev/urandom que
          * contienen caracteres aleatorios.  */
 420                 rand_initialize_irq(irq);
 421         }
 422
 423         /*
 424          * The following block of code has to be executed atomically
 425          */
 286          * El siguiente bloque de código tiene que ejecutarse indivisiblemente
 287          */
 426         spin_lock_irqsave(&desc->lock, flags);
 427         p = &desc->action;
        /* nodo de la lista no vacío, existe ya una interrupción */
 428         old = *p;
 429         if (old) {
 430                 /*
 431                  * Can't share interrupts unless both agree to and are
 432                  * the same type (level, edge, polarity). So both flag
 433                  * fields must have IRQF_SHARED set and the bits which
 434                  * set the trigger type must match.
 435                  */
/* verifica que irq puede compartirse. No se pueden compartir  interrupciones a
menos que estén de acuerdo  y sean del mismo tipo (level, edge, polarity). A si,
tiene que estar activo IRQF_SHARED y tener los  mismos flag que definen el tipo de
disparo.  */
 436                 if (!((old->flags & new->flags) & IRQF_SHARED) ||
 437                     ((old->flags ^ new->flags) & IRQF_TRIGGER_MASK)) {
 438                         old_name = old->name;
 439                         goto mismatch;
 440                 }
 441
 442#if defined(CONFIG_IRQ_PER_CPU)
 443                 /* All handlers must agree on per-cpuness */
 444                 if ((old->flags & IRQF_PERCPU) !=
 445                     (new->flags & IRQF_PERCPU))
 446                         goto mismatch;
 447#endif
 448
```

```
 449                /* add new interrupt at end of irq queue */
/* añade una nueva interrupción al final de la  cola  irq  y  coloca a uno el flag shared
 450                do {
 451                        p = &old->next;
 452                        old = *p;
 453                } while (old);
 454                shared = 1;
 455        }
 456
/* si no se habían establecido acciones en esta irq inicializar el resto de campos   */
 457        if (!shared) {
 458                irq_chip_set_defaults(desc->chip);
 459
 460                /* Setup the type (level, edge polarity) if configured:
/* Define el tipo de disparo (level, edge polarity) si nueva */
 461                if (new->flags & IRQF_TRIGGER_MASK) {
 462                        ret = __irq_set_trigger(desc, irq, new->flags);
 463
 464                        if (ret) {
 465                                spin_unlock_irqrestore(&desc->lock,
flags);
 466                                return ret;
 467                        }
 468                } else
 469                        compat_irq_chip_set_default_handler(desc);
 470#if defined(CONFIG_IRQ_PER_CPU)
 471                if (new->flags & IRQF_PERCPU)
 472                        desc->status |= IRQ_PER_CPU;
 473#endif
 474
 475                desc->status &= ~(IRQ_AUTODETECT | IRQ_WAITING |
 476                                  IRQ_INPROGRESS |
IRQ_SPURIOUS_DISABLED);
 477
 478                if (!(desc->status & IRQ_NOAUTOEN)) {
 479                        desc->depth = 0;
 480                        desc->status &= ~IRQ_DISABLED;
                              /* inicia y permite interrumpir por irq */
 481                        desc->chip->startup(irq);
 482                } else
 483                        /* Undo nested disables: */
/* si es la primera profundidad 1 */
 484                        desc->depth = 1;
 485
 486                /* Exclude IRQ from balancing if requested */
 487                if (new->flags & IRQF_NOBALANCING)
 488                        desc->status |= IRQ_NO_BALANCING;
 489
 490                /* Set default affinity mask once everything is setup
*/
 491                do_irq_select_affinity(irq, desc);
 492
 493        } else if ((new->flags & IRQF_TRIGGER_MASK)
 494                        && (new->flags & IRQF_TRIGGER_MASK)
 495                                != (desc->status &
IRQ_TYPE_SENSE_MASK)) {
 496                /* hope the handler works with the actual trigger
mode... */
```

```
 497                pr_warning("IRQ %d uses trigger mode %d; requested
%d\n",
 498                                   irq, (int)(desc->status &
IRQ_TYPE_SENSE_MASK),
 499                                   (int)(new->flags & IRQF_TRIGGER_MASK));
 500        }
 501
 502        *p = new;
 503
 504        /* Reset broken irq detection when installing new handler */
/* Resetea contadores de esta interrupción cuando instalamos nuevo manejador */
 505        desc->irq_count = 0;
 506        desc->irqs_unhandled = 0;
 507
 508        /*
 509         * Check whether we disabled the irq via the spurious handler
 510         * before. Reenable it and give it another chance.
 511         */
 512        if (shared && (desc->status & IRQ_SPURIOUS_DISABLED)) {
 513                desc->status &= ~IRQ_SPURIOUS_DISABLED;
 514                __enable_irq(desc, irq);
 515        }
 516
 517        spin_unlock_irqrestore(&desc->lock, flags);
 518
 /* crea un directorio en el sistema de ficheros /proc/irq  para registrar el manejador
 519        new->irq = irq;
 520        register_irq_proc(irq, desc);
 521        new->dir = NULL;
 522        register_handler_proc(irq, new);
 523
 524        return 0;
 525
 526mismatch:
 527#ifdef CONFIG_DEBUG_SHIRQ
 528        if (!(new->flags & IRQF_PROBE_SHARED)) {
 529                printk(KERN_ERR "IRQ handler type mismatch for IRQ
%d\n", irq);
 530                if (old_name)
 531                        printk(KERN_ERR "current handler: %s\n",
old_name);
 532                dump_stack();
 533        }
 534#endif
 535        spin_unlock_irqrestore(&desc->lock, flags);
 536        return -EBUSY;
 537}
```

**free_irq**

Llamada por los manejadores para quitar un nodo de la lista **irqaction**, es la acción inversa de request_irq, se encuentra definida en kernel/irq/manage.c.
Quita un manejador de interrupción y si es el último libera la interrupción. Tiene dos parámetros **irq**, la interrupción a liberar y **dev_id** el dispositivo que ya no interrumpe.

```
/**
*      free_irq – libera una interrupción
*      @irq: Línea de interrupción a liberarse
*      @dev_id: Identificación del dispositivo a liberarse
*
*      Quita un manejador de interrupción. Cuando el manejador se
*      quita y la línea de interrupción ya no se usa por ningún
dispositivo, se coloca como desactivada.
*      En una IRQ compartida, se debe comprobar que la interrupción
*      esta desactivada. La function no retorna hasta que todas las
*      interrupciones que se están ejecutando sobre esta IRQ
*      se han completado.
*
*      Esta function no debe llamarse dentro de un contexto de
interrupción.
*/


void free_irq(unsigned int irq, void *dev_id)
 568{
 569        struct irq_desc *desc = irq_to_desc(irq);
 570        struct irqaction **p;
 571        unsigned long flags;
 572
 573        WARN_ON(in_interrupt());
 574
 575        if (!desc)
 576                return;
 577
 578        spin_lock_irqsave(&desc->lock, flags);
/* Obtiene la entrada correspondiente en irq_desc y avanza por la lista hasta
encontrar el dispositivo para realizar las acciones */
 579        p = &desc->action;
 580        for (;;) {
 581                struct irqaction *action = *p;
 582
 583                if (action) {
 584                        struct irqaction **pp = p;
 585
 586                        p = &action->next;
                                /* es el ID del dispositivo */
 587                        if (action->dev_id != dev_id)
 588                                continue;
 589
 590                        /* Found it - now remove it from the list of
entries */
/* encuentra el dispositivo quita el nodo de la lista */
 591                        *pp = action->next;
 592
 593                        /* Currently used only by UML, might disappear
one day.*/
 594#ifdef CONFIG_IRQ_RELEASE_METHOD
 595                        if (desc->chip->release)
 596                                desc->chip->release(irq, dev_id);
 597#endif
 598
 /* si la cola tenía un solo elemento la entrada irq es deshabilitada */
 599                        if (!desc->action) {
 600                                desc->status |= IRQ_DISABLED;
```

```
 601                                    if (desc->chip->shutdown)
 602                                            desc->chip->shutdown(irq);
 603                                    else
 604                                            desc->chip->disable(irq);
 605                            }
 606                            spin_unlock_irqrestore(&desc->lock, flags);
```

/* borra el registro del manejador en el sistema de ficheros /proc */

```
 607                            unregister_handler_proc(irq, action);
 608
 609                            /* Make sure it's not being used on another CPU
```

*//* se asegura que no está siendo usada por otra CPU */

```
 610                            synchronize_irq(irq);
 611#ifdef CONFIG_DEBUG_SHIRQ
 612                            /*
 613                             * It's a shared IRQ -- the driver ought to be
 614                             * prepared for it to happen even now it's
 615                             * being freed, so let's make sure....  We do
 616                             * this after actually deregistering it, to
 617                             * make sure that a 'real' IRQ doesn't run in
 618                             * parallel with our fake
 619                             */
 620                            if (action->flags & IRQF_SHARED) {
 621                                    local_irq_save(flags);
 622                                    action->handler(irq, dev_id);
 623                                    local_irq_restore(flags);
 624                            }
 625#endif
```

/* Elimina el elemento de la lista y libera memoria */

```
 626                            kfree(action);
 627                            return;
 628                    }
```

/* si se ha alcanzado este punto es porque no se ha encontrado en la lista ningún
 * dev_id, y hay un error de liberar una irq realiza este printk */

```
 629                    printk(KERN_ERR "Trying to free already-free IRQ %d\n",
irq);
 630#ifdef CONFIG_DEBUG_SHIRQ
 631                    dump_stack();
 632#endif
 633                    spin_unlock_irqrestore(&desc->lock, flags);
 634                    return;
 635            }
 636}
 637EXPORT_SYMBOL(free_irq);
 638
```