Figure 1.1 - basic interrupt handling flow

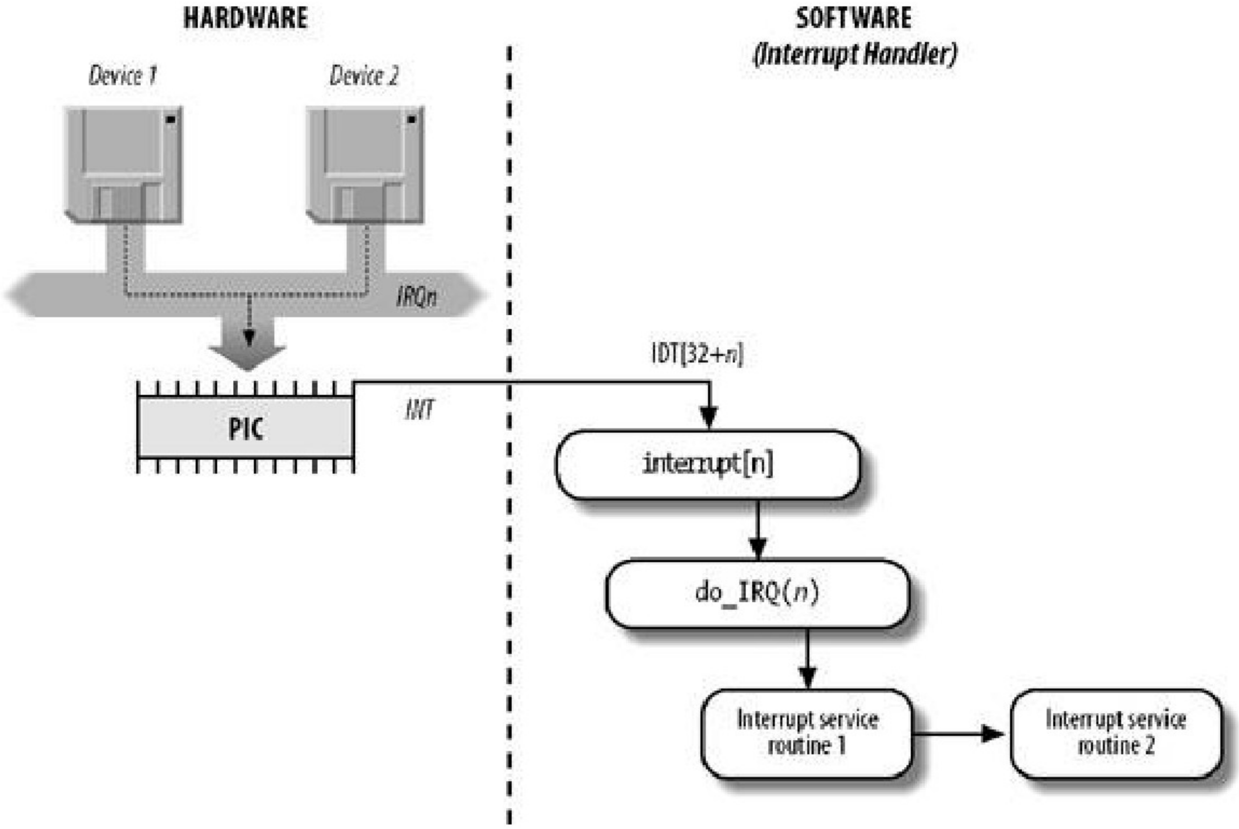


Figure 1.2 - low-level handler sample

```
common_interrupt:
    SAVE_ALL
    movl %esp,%eax
    call do_IRQ
    jmp ret_from_intr
```

The SAVE_ALL macro expands to the following fragment:

```
 cld
 push %es
 push %ds
 pushl %eax
 pushl %ebp
 pushl %edi
 pushl %esi
 pushl %edx
 pushl %ecx
 pushl %ebx
 movl $ _ _USER_DS,%edx
 movl %edx,%ds
 movl %edx,%es
```

Figure 1.3 - checking for preemption counter by kernel at end of int handling

```
resume_kernel:

  cli        ; these three instructions are
  cmpl $0, 0x14(%ebp) ;  //checking for preemption counter

  jz need_resched ;

restore_all:
  popl %ebx
  popl %ecx
  popl %edx
  popl %esi
  popl %edi
  popl %ebp
  popl %eax
  popl %ds
  popl %es
  addl $4, %esp
  iret
```

Figure 1.4 - checking for the rescheduling flag during interrupt handling

```
need_resched:
  movl 0x8(%ebp), %ecx
  testb $(1<<TIF_NEED_RESCHED), %cl
  jz restore_all
  testl $0x00000200,0x30(%esp)
  jz restore_all
  call preempt_schedule_irq
  jmp need_resched
```

Figure 1.5 - overall interrupt handling flow

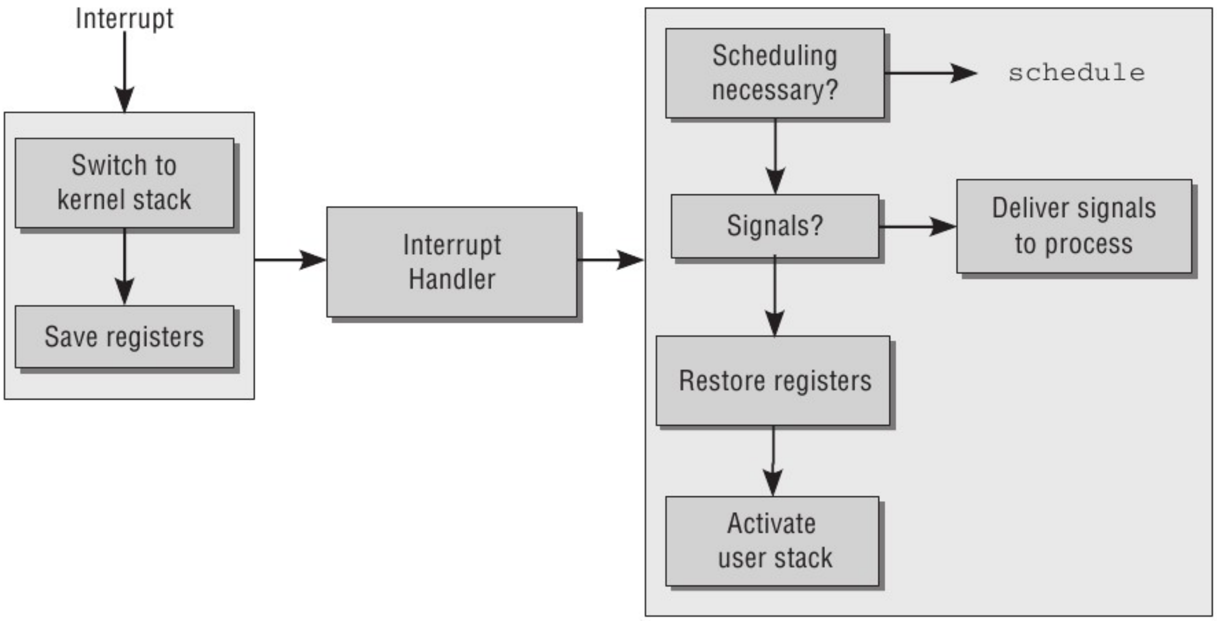
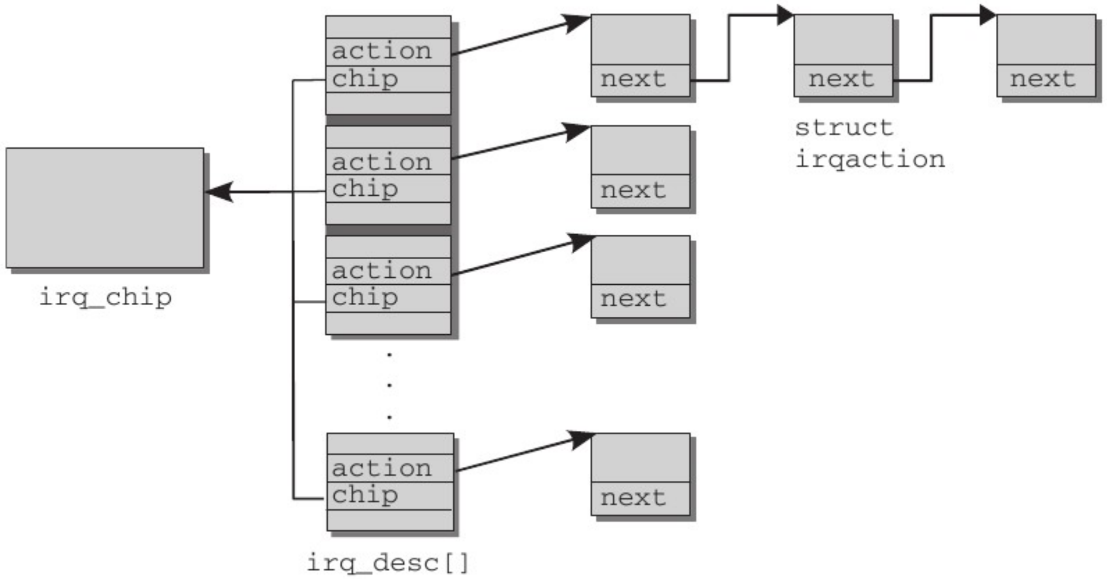


Figure 1.6 - irq desc table maintaining high-level handlers of drivers and other activities



```
struct irq_chip {
        const char   *name;
        unsigned int (*startup)(unsigned int irq);
        void        (*shutdown)(unsigned int irq);
        void        (*enable)(unsigned int irq);
        void        (*disable)(unsigned int irq);
        void        (*ack)(unsigned int irq);
        void        (*mask)(unsigned int irq);
        void        (*mask_ack)(unsigned int irq);
        void        (*unmask)(unsigned int irq);
        void        (*eoi)(unsigned int irq);
        ....

}
struct irqaction {
        irq_handler_t handler;
        unsigned long flags;
        const char *name;
        void *dev_id;
        struct irqaction *next;
}
```

Figure 1.7 - high-level handlers and flow

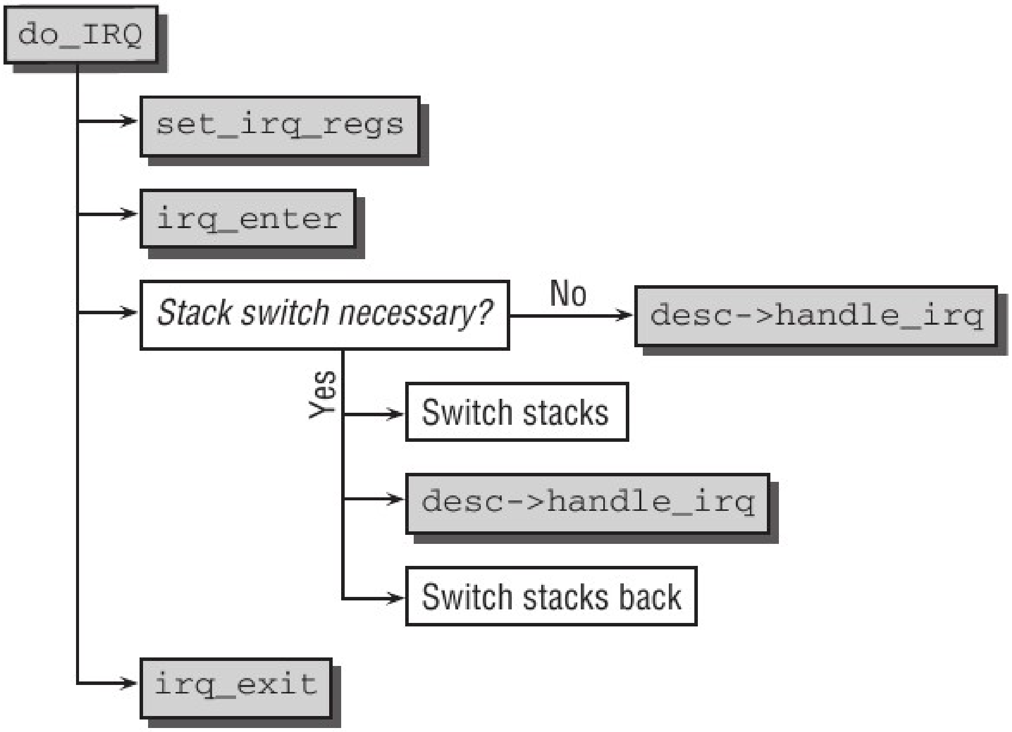


Figure 1.8 - high level handlers and flow continued

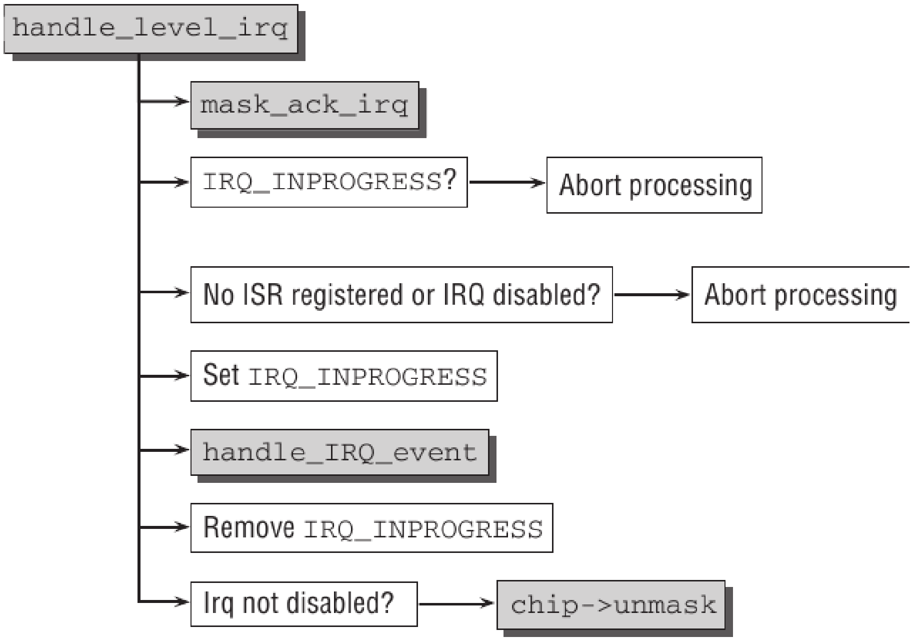
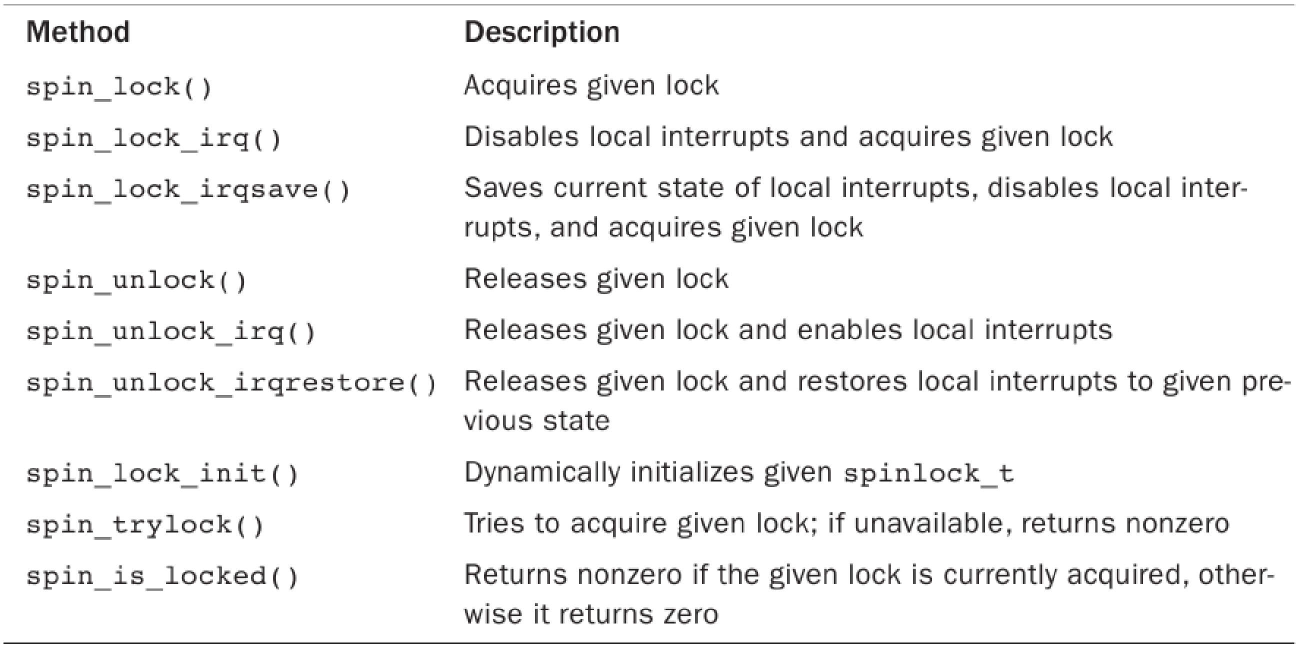


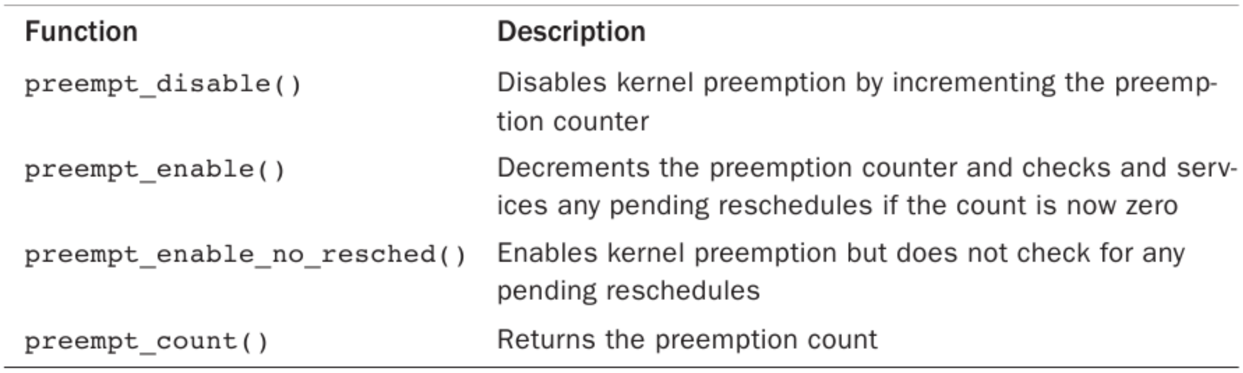
Figure 1.9 - interrupt handling functions



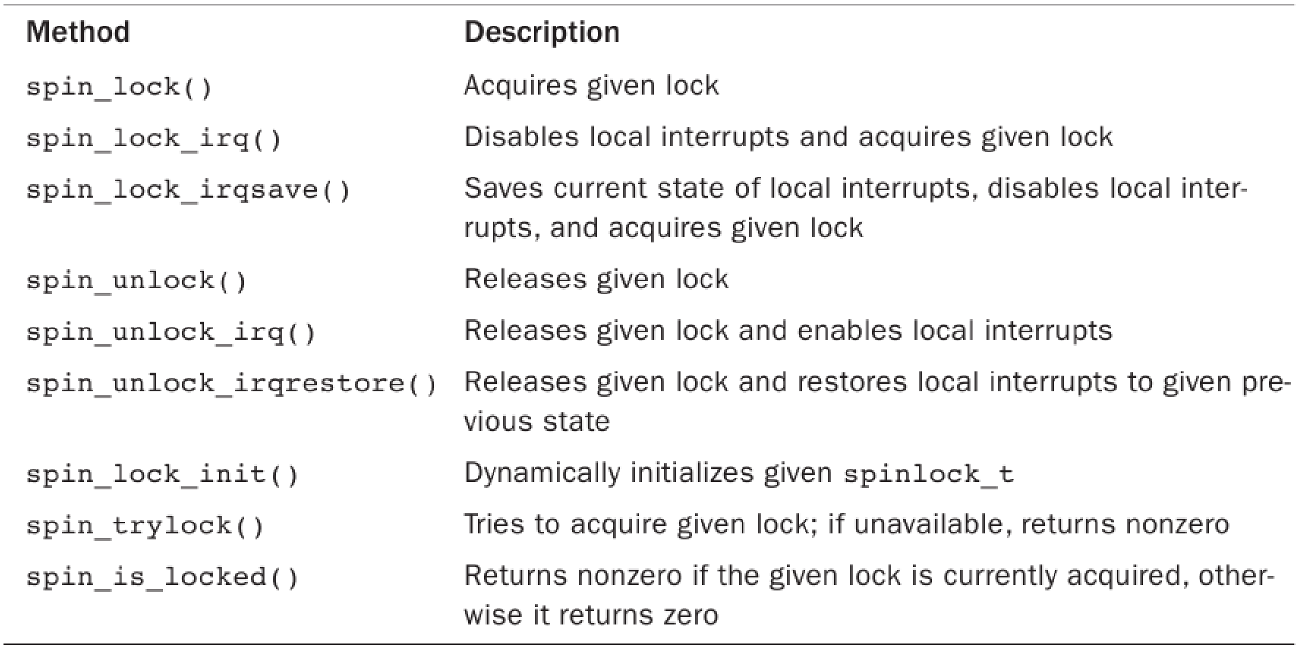
Method	Description
`spin_lock()`	Acquires given lock
`spin_lock_irq()`	Disables local interrupts and acquires given lock
`spin_lock_irqsave()`	Saves current state of local interrupts, disables local interrupts, and acquires given lock
`spin_unlock()`	Releases given lock
`spin_unlock_irq()`	Releases given lock and enables local interrupts
`spin_unlock_irqrestore()`	Releases given lock and restores local interrupts to given previous state
`spin_lock_init()`	Dynamically initializes given `spinlock_t`
`spin_trylock()`	Tries to acquire given lock; if unavailable, returns nonzero
`spin_is_locked()`	Returns nonzero if the given lock is currently acquired, otherwise it returns zero

Function	Description
`local_irq_disable()`	Disables local interrupt delivery
`local_irq_enable()`	Enables local interrupt delivery
`local_irq_save()`	Saves the current state of local interrupt delivery and then disables it
`local_irq_restore()`	Restores local interrupt delivery to the given state
`disable_irq()`	Disables the given interrupt line and ensures no handler on the line is executing before returning
`disable_irq_nosync()`	Disables the given interrupt line
`enable_irq()`	Enables the given interrupt line
`irqs_disabled()`	Returns nonzero if local interrupt delivery is disabled; otherwise returns zero
`in_interrupt()`	Returns nonzero if in interrupt context and zero if in process context
`in_irq()`	Returns nonzero if currently executing an interrupt handler and zero otherwise

Figure 1.10 - preemption related functions



Function	Description
`preempt_disable()`	Disables kernel preemption by incrementing the preemption counter
`preempt_enable()`	Decrements the preemption counter and checks and services any pending reschedules if the count is now zero
`preempt_enable_no_resched()`	Enables kernel preemption but does not check for any pending reschedules
`preempt_count()`	Returns the preemption count



Method	Description
`spin_lock()`	Acquires given lock
`spin_lock_irq()`	Disables local interrupts and acquires given lock
`spin_lock_irqsave()`	Saves current state of local interrupts, disables local interrupts, and acquires given lock
`spin_unlock()`	Releases given lock
`spin_unlock_irq()`	Releases given lock and enables local interrupts
`spin_unlock_irqrestore()`	Releases given lock and restores local interrupts to given previous state
`spin_lock_init()`	Dynamically initializes given `spinlock_t`
`spin_trylock()`	Tries to acquire given lock; if unavailable, returns nonzero
`spin_is_locked()`	Returns nonzero if the given lock is currently acquired, otherwise it returns zero