

# **Linux generic IRQ handling**

**Thomas Gleixner**

**`tglx@linutronix.de`**

**Ingo Molnar**

**`mingo@elte.hu`**

## **Linux generic IRQ handling**

by Thomas Gleixner and Ingo Molnar

Copyright © 2005-2006 Thomas Gleixner

Copyright © 2005-2006 Ingo Molnar

This documentation is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License version 2 as published by the Free Software Foundation.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

For more details see the file COPYING in the source distribution of Linux.

# Table of Contents

<b>1. Introduction.....</b>	<b>1</b>
<b>2. Rationale .....</b>	<b>3</b>
<b>3. Known Bugs And Assumptions .....</b>	<b>5</b>
<b>4. Abstraction layers .....</b>	<b>7</b>
4.1. Interrupt control flow .....	7
4.2. Highlevel Driver API .....	7
4.3. Highlevel IRQ flow handlers .....	8
4.3.1. Default flow implementations .....	8
4.3.1.1. Helper functions.....	8
4.3.2. Default flow handler implementations.....	9
4.3.2.1. Default Level IRQ flow handler.....	9
4.3.2.2. Default Edge IRQ flow handler .....	9
4.3.2.3. Default simple IRQ flow handler.....	10
4.3.2.4. Default per CPU flow handler.....	10
4.3.3. Quirks and optimizations .....	10
4.3.4. Delayed interrupt disable .....	11
4.4. Chiplevel hardware encapsulation .....	11
<b>5. __do_IRQ entry point.....</b>	<b>13</b>
<b>6. Locking on SMP.....</b>	<b>15</b>
<b>7. Structures.....</b>	<b>17</b>
struct irq_chip .....	17
struct irq_desc .....	19
<b>8. Public Functions Provided .....</b>	<b>23</b>
synchronize_irq.....	23
disable_irq_nosync .....	23
disable_irq.....	24
enable_irq.....	25
set_irq_wake .....	26
free_irq.....	27
request_irq.....	28
set_irq_chip.....	29
set_irq_type.....	30
set_irq_data.....	31
set_irq_chip_data .....	32
<b>9. Internal Functions Provided .....</b>	<b>35</b>
handle_bad_irq.....	35
handle_IRQ_event .....	36
__do_IRQ.....	36
dynamic_irq_init.....	37

dynamic_irq_cleanup.....	38
set_irq_msi.....	39
handle_simple_irq.....	39
handle_level_irq.....	40
handle_fasteoi_irq.....	41
handle_edge_irq.....	42
handle_percpu_irq.....	43
<b>10. Credits.....</b>	<b>45</b>

# Chapter 1. Introduction

The generic interrupt handling layer is designed to provide a complete abstraction of interrupt handling for device drivers. It is able to handle all the different types of interrupt controller hardware. Device drivers use generic API functions to request, enable, disable and free interrupts. The drivers do not have to know anything about interrupt hardware details, so they can be used on different platforms without code changes.

This documentation is provided to developers who want to implement an interrupt subsystem based for their architecture, with the help of the generic IRQ handling layer.



# Chapter 2. Rationale

The original implementation of interrupt handling in Linux is using the `__do_IRQ()` super-handler, which is able to deal with every type of interrupt logic.

Originally, Russell King identified different types of handlers to build a quite universal set for the ARM interrupt handler implementation in Linux 2.5/2.6. He distinguished between:

- Level type
- Edge type
- Simple type

In the SMP world of the `__do_IRQ()` super-handler another type was identified:

- Per CPU type

This split implementation of highlevel IRQ handlers allows us to optimize the flow of the interrupt handling for each specific interrupt type. This reduces complexity in that particular codepath and allows the optimized handling of a given type.

The original general IRQ implementation used `hw_interrupt_type` structures and their `->ack()`, `->end()` [etc.] callbacks to differentiate the flow control in the super-handler. This leads to a mix of flow logic and lowlevel hardware logic, and it also leads to unnecessary code duplication: for example in i386, there is a `ioapic_level_irq` and a `ioapic_edge_irq` irq-type which share many of the lowlevel details but have different flow handling.

A more natural abstraction is the clean separation of the 'irq flow' and the 'chip details'.

Analysing a couple of architecture's IRQ subsystem implementations reveals that most of them can use a generic set of 'irq flow' methods and only need to add the chip level specific code. The separation is also valuable for (sub)architectures which need specific quirks in the irq flow itself but not in the chip-details - and thus provides a more transparent IRQ subsystem design.

Each interrupt descriptor is assigned its own highlevel flow handler, which is normally one of the generic implementations. (This highlevel flow handler implementation also makes it simple to provide demultiplexing handlers which can be found in embedded platforms on various architectures.)

The separation makes the generic interrupt handling layer more flexible and extensible. For example, an (sub)architecture can use a generic irq-flow

## *Chapter 2. Rationale*

implementation for 'level type' interrupts and add a (sub)architecture specific 'edge type' implementation.

To make the transition to the new model easier and prevent the breakage of existing implementations, the `__do_IRQ()` super-handler is still available. This leads to a kind of duality for the time being. Over time the new model should be used in more and more architectures, as it enables smaller and cleaner IRQ subsystems.



# Chapter 3. Known Bugs And Assumptions

None (knock on wood).



# Chapter 4. Abstraction layers

There are three main levels of abstraction in the interrupt code:

1. Highlevel driver API
2. Highlevel IRQ flow handlers
3. Chiplevel hardware encapsulation

## 4.1. Interrupt control flow

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 description 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.

Whenever an interrupt triggers, the lowlevel arch code calls into the generic interrupt code by calling `desc->handle_irq()`. This highlevel IRQ handling function only uses `desc->chip` primitives referenced by the assigned chip descriptor structure.

## 4.2. Highlevel Driver API

The highlevel Driver API consists of following functions:

- `request_irq()`
- `free_irq()`
- `disable_irq()`
- `enable_irq()`
- `disable_irq_nosync()` (SMP only)
- `synchronize_irq()` (SMP only)
- `set_irq_type()`
- `set_irq_wake()`
- `set_irq_data()`

- `set_irq_chip()`
- `set_irq_chip_data()`

See the autogenerated function documentation for details.

## 4.3. Highlevel IRQ flow handlers

The generic layer provides a set of pre-defined irq-flow methods:

- `handle_level_irq`
- `handle_edge_irq`
- `handle_simple_irq`
- `handle_percpu_irq`

The interrupt flow handlers (either predefined or architecture specific) are assigned to specific interrupts by the architecture either during bootup or during device initialization.

### 4.3.1. Default flow implementations

#### 4.3.1.1. Helper functions

The helper functions call the chip primitives and are used by the default flow implementations. The following helper functions are implemented (simplified excerpt):

```
default_enable(irq)
{
    desc->chip->unmask(irq);
}

default_disable(irq)
{
    if (!delay_disable(irq))
        desc->chip->mask(irq);
}

default_ack(irq)
{
    chip->ack(irq);
}
```

```
default_mask_ack(irq)
{
    if (chip->mask_ack) {
        chip->mask_ack(irq);
    } else {
        chip->mask(irq);
        chip->ack(irq);
    }
}

noop(irq)
{
}
```

## 4.3.2. Default flow handler implementations

### 4.3.2.1. Default Level IRQ flow handler

`handle_level_irq` provides a generic implementation for level-triggered interrupts.

The following control flow is implemented (simplified excerpt):

```
desc->chip->start();
handle_IRQ_event(desc->action);
desc->chip->end();
```

### 4.3.2.2. Default Edge IRQ flow handler

`handle_edge_irq` provides a generic implementation for edge-triggered interrupts.

The following control flow is implemented (simplified excerpt):

```
if (desc->status & running) {
    desc->chip->hold();
    desc->status |= pending | masked;
    return;
}
```

```
desc->chip->start();
desc->status |= running;
do {
    if (desc->status & masked)
        desc->chip->enable();
    desc->status &= ~pending;
    handle_IRQ_event(desc->action);
} while (status & pending);
desc->status &= ~running;
desc->chip->end();
```

### **4.3.2.3. Default simple IRQ flow handler**

`handle_simple_irq` provides a generic implementation for simple interrupts.

Note: The simple flow handler does not call any handler/chip primitives.

The following control flow is implemented (simplified excerpt):

```
handle_IRQ_event(desc->action);
```

### **4.3.2.4. Default per CPU flow handler**

`handle_percpu_irq` provides a generic implementation for per CPU interrupts.

Per CPU interrupts are only available on SMP and the handler provides a simplified version without locking.

The following control flow is implemented (simplified excerpt):

```
desc->chip->start();
handle_IRQ_event(desc->action);
desc->chip->end();
```

### 4.3.3. Quirks and optimizations

The generic functions are intended for 'clean' architectures and chips, which have no platform-specific IRQ handling quirks. If an architecture needs to implement quirks on the 'flow' level then it can do so by overriding the highlevel irq-flow handler.

### 4.3.4. Delayed interrupt disable

This per interrupt selectable feature, which was introduced by Russell King in the ARM interrupt implementation, does not mask an interrupt at the hardware level when `disable_irq()` is called. The interrupt is kept enabled and is masked in the flow handler when an interrupt event happens. This prevents losing edge interrupts on hardware which does not store an edge interrupt event while the interrupt is disabled at the hardware level. When an interrupt arrives while the `IRQ_DISABLED` flag is set, then the interrupt is masked at the hardware level and the `IRQ_PENDING` bit is set. When the interrupt is re-enabled by `enable_irq()` the pending bit is checked and if it is set, the interrupt is resent either via hardware or by a software resend mechanism. (It's necessary to enable `CONFIG_HARDIRQS_SW_RESEND` when you want to use the delayed interrupt disable feature and your hardware is not capable of retriggering an interrupt.) The delayed interrupt disable can be runtime enabled, per interrupt, by setting the `IRQ_DELAYED_DISABLE` flag in the `irq_desc` status field.

## 4.4. Chiplevel hardware encapsulation

The chip level hardware descriptor structure `irq_chip` contains all the direct chip relevant functions, which can be utilized by the irq flow implementations.

- `ack()`
- `mask_ack()` - Optional, recommended for performance
- `mask()`
- `unmask()`
- `retrigger()` - Optional
- `set_type()` - Optional
- `set_wake()` - Optional

#### *Chapter 4. Abstraction layers*

These primitives are strictly intended to mean what they say: ack means ACK, masking means masking of an IRQ line, etc. It is up to the flow handler(s) to use these basic units of lowlevel functionality.



# Chapter 5. `__do_IRQ` entry point

The original implementation `__do_IRQ()` is an alternative entry point for all types of interrupts.

This handler turned out to be not suitable for all interrupt hardware and was therefore reimplemented with split functionality for egde/level/simple/percpu interrupts. This is not only a functional optimization. It also shortens code paths for interrupts.

To make use of the split implementation, replace the call to `__do_IRQ` by a call to `desc->chip->handle_irq()` and associate the appropriate handler function to `desc->chip->handle_irq()`. In most cases the generic handler implementations should be sufficient.



# Chapter 6. Locking on SMP

The locking of chip registers is up to the architecture that defines the chip primitives. There is a `chip->lock` field that can be used for serialization, but the generic layer does not touch it. The `per-irq` structure is protected via `desc->lock`, by the generic layer.



# Chapter 7. Structures

This chapter contains the autogenerated documentation of the structures which are used in the generic IRQ layer.

## struct irq\_chip

**LINUX**

Kernel Hackers Manual July 2010

### Name

struct irq\_chip — hardware interrupt chip descriptor

### Synopsis

```
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);
    void (* end) (unsigned int irq);
    void (* set_affinity) (unsigned int irq, cpumask_t dest);
    int (* retrigger) (unsigned int irq);
    int (* set_type) (unsigned int irq, unsigned int flow_type);
    int (* set_wake) (unsigned int irq, unsigned int on);
#ifdef CONFIG_IRQ_RELEASE_METHOD
    void (* release) (unsigned int irq, void *dev_id);
#endif
    const char * typename;
};
```

## Members

name

name for /proc/interrupts

startup

start up the interrupt (defaults to ->enable if NULL)

shutdown

shut down the interrupt (defaults to ->disable if NULL)

enable

enable the interrupt (defaults to chip->unmask if NULL)

disable

disable the interrupt (defaults to chip->mask if NULL)

ack

start of a new interrupt

mask

mask an interrupt source

mask\_ack

ack and mask an interrupt source

unmask

unmask an interrupt source

eoi

end of interrupt - chip level

end

end of interrupt - flow level

set\_affinity

set the CPU affinity on SMP machines

retrigger

resend an IRQ to the CPU

set\_type

set the flow type (IRQ\_TYPE\_LEVEL/etc.) of an IRQ

set\_wake

enable/disable power-management wake-on of an IRQ

release

release function solely used by UML

typename

obsoleted by name, kept as migration helper

## struct irq\_desc

### LINUX

Kernel Hackers Manual July 2010

### Name

struct irq\_desc — interrupt descriptor

### Synopsis

```
struct irq_desc {
    irq_flow_handler_t handle_irq;
    struct irq_chip * chip;
    struct msi_desc * msi_desc;
    void * handler_data;
    void * chip_data;
    struct irqaction * action;
    unsigned int status;
    unsigned int depth;
    unsigned int wake_depth;
    unsigned int irq_count;
    unsigned int irqs_unhandled;
    unsigned long last_unhandled;
    spinlock_t lock;
#ifdef CONFIG_SMP
```

```
    cpumask_t affinity;
    unsigned int cpu;
#endif
#if defined(CONFIG_GENERIC_PENDING_IRQ) || defined(CONFIG_IRQBALANCE)
    cpumask_t pending_mask;
#endif
#ifdef CONFIG_PROC_FS
    struct proc_dir_entry * dir;
#endif
    const char * name;
};
```

## Members

`handle_irq`

highlevel irq-events handler [if NULL, `__do_IRQ`]

`chip`

low level interrupt hardware access

`msi_desc`

MSI descriptor

`handler_data`

per-IRQ data for the `irq_chip` methods

`chip_data`

platform-specific per-chip private data for the chip methods, to allow shared chip implementations

`action`

the irq action chain

`status`

status information

`depth`

disable-depth, for nested `irq_disable` calls



wake\_depth

enable depth, for multiple `set_irq_wake` callers

irq\_count

stats field to detect stalled irqs

irqs\_unhandled

stats field for spurious unhandled interrupts

last\_unhandled

aging timer for unhandled count

lock

locking for SMP

affinity

IRQ affinity on SMP

cpu

cpu index useful for balancing

pending\_mask

pending rebalanced interrupts

dir

`/proc/irq/` procfs entry

name

flow handler name for `/proc/interrupts` output



# Chapter 8. Public Functions Provided

This chapter contains the autogenerated documentation of the kernel API functions which are exported.

## synchronize\_irq

### LINUX

Kernel Hackers Manual July 2010

### Name

`synchronize_irq` — wait for pending IRQ handlers (on other CPUs)

### Synopsis

```
void synchronize_irq (unsigned int irq);
```

### Arguments

*irq*

interrupt number to wait for

### Description

This function waits for any pending IRQ handlers for this interrupt to complete before returning. If you use this function while holding a resource the IRQ handler may need you will deadlock.

This function may be called - with care - from IRQ context.

# disable\_irq\_nosync

## LINUX

Kernel Hackers Manual July 2010

### Name

`disable_irq_nosync` — disable an irq without waiting

### Synopsis

```
void disable_irq_nosync (unsigned int irq);
```

### Arguments

*irq*

Interrupt to disable

### Description

Disable the selected interrupt line. Disables and Enables are nested. Unlike `disable_irq`, this function does not ensure existing instances of the IRQ handler have completed before returning.

This function may be called from IRQ context.

# disable\_irq

## LINUX

## Name

`disable_irq` — disable an irq and wait for completion

## Synopsis

```
void disable_irq (unsigned int irq);
```

## Arguments

*irq*

Interrupt to disable

## Description

Disable the selected interrupt line. Enables and Disables are nested. This function waits for any pending IRQ handlers for this interrupt to complete before returning. If you use this function while holding a resource the IRQ handler may need you will deadlock.

This function may be called - with care - from IRQ context.

## `enable_irq`

### LINUX

## Name

`enable_irq` — enable handling of an irq

## Synopsis

```
void enable_irq (unsigned int irq);
```

## Arguments

*irq*

Interrupt to enable

## Description

Undoes the effect of one call to `disable_irq`. If this matches the last disable, processing of interrupts on this IRQ line is re-enabled.

This function may be called from IRQ context.

## set\_irq\_wake

### LINUX

Kernel Hackers Manual July 2010

## Name

`set_irq_wake` — control irq power management wakeup

## Synopsis

```
int set_irq_wake (unsigned int irq, unsigned int on);
```

## Arguments

*irq*

interrupt to control

*on*

enable/disable power management wakeup

## Description

Enable/disable power management wakeup mode, which is disabled by default. Enables and disables must match, just as they match for non-wakeup mode support.

Wakeup mode lets this IRQ wake the system from sleep states like “suspend to RAM”.

## free\_irq

### LINUX

Kernel Hackers Manual July 2010

## Name

`free_irq` — free an interrupt

## Synopsis

```
void free_irq (unsigned int irq, void * dev_id);
```

## Arguments

*irq*

Interrupt line to free

*dev\_id*

Device identity to free

## Description

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.

This function must not be called from interrupt context.

## request\_irq

### LINUX

Kernel Hackers Manual July 2010

## Name

`request_irq` — allocate an interrupt line

## Synopsis

```
int request_irq (unsigned int irq, irq_handler_t handler,  
unsigned long irqflags, const char * devname, void * dev_id);
```



## Arguments

*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

## Description

This call allocates interrupt resources and enables the interrupt line and IRQ handling. From the point this call is made your handler function may be invoked. Since your handler function must clear any interrupt the board raises, you must take care both to initialise your hardware and to set up the interrupt handler in the right order.

Dev\_id must be globally unique. Normally the address of the device data structure is used as the cookie. Since the handler receives this value it makes sense to use it.

If your interrupt is shared you must pass a non NULL dev\_id as this is required when freeing the interrupt.

## Flags

IRQF\_SHARED Interrupt is shared  
IRQF\_DISABLED Disable local interrupts while processing  
IRQF\_SAMPLE\_RANDOM The interrupt can be used for entropy

# set\_irq\_chip

## LINUX

Kernel Hackers Manual July 2010

### Name

`set_irq_chip` — set the irq chip for an irq

### Synopsis

```
int set_irq_chip (unsigned int irq, struct irq_chip * chip);
```

### Arguments

*irq*

irq number

*chip*

pointer to irq chip description structure

# set\_irq\_type

## LINUX

Kernel Hackers Manual July 2010

### Name

`set_irq_type` — set the irq type for an irq

## Synopsis

```
int set_irq_type (unsigned int irq, unsigned int type);
```

## Arguments

*irq*

irq number

*type*

interrupt type - see include/linux/interrupt.h

## set\_irq\_data

### LINUX

Kernel Hackers Manual July 2010

## Name

`set_irq_data` — set irq type data for an irq

## Synopsis

```
int set_irq_data (unsigned int irq, void * data);
```

## Arguments

*irq*

Interrupt number

*data*

Pointer to interrupt specific data

## Description

Set the hardware irq controller data for an irq

# set\_irq\_chip\_data

## LINUX

Kernel Hackers Manual July 2010

## Name

`set_irq_chip_data` — set irq chip data for an irq

## Synopsis

```
int set_irq_chip_data (unsigned int irq, void * data);
```

## Arguments

*irq*

Interrupt number

*data*

Pointer to chip specific data

## **Description**

Set the hardware irq chip data for an irq



# Chapter 9. Internal Functions Provided

This chapter contains the autogenerated documentation of the internal functions.

## handle\_bad\_irq

**LINUX**

Kernel Hackers Manual July 2010

### Name

`handle_bad_irq` — handle spurious and unhandled irqs

### Synopsis

```
void handle_bad_irq (unsigned int irq, struct irq_desc *  
desc);
```

### Arguments

*irq*

the interrupt number

*desc*

description of the interrupt

### Description

Handles spurious and unhandled IRQ's. It also prints a debugmessage.

# handle\_IRQ\_event

## LINUX

Kernel Hackers Manual July 2010

### Name

`handle_IRQ_event` — irq action chain handler

### Synopsis

```
irqreturn_t handle_IRQ_event (unsigned int irq, struct  
irqaction * action);
```

### Arguments

*irq*

the interrupt number

*action*

the interrupt action chain for this irq

### Description

Handles the action chain of an irq event

## \_\_do\_IRQ

## LINUX



## Name

`__do_IRQ` — original all in one highlevel IRQ handler

## Synopsis

```
unsigned int __do_IRQ (unsigned int irq);
```

## Arguments

*irq*

the interrupt number

## Description

`__do_IRQ` handles all normal device IRQ's (the special SMP cross-CPU interrupts have their own specific handlers).

This is the original x86 implementation which is used for every interrupt type.

# dynamic\_irq\_init

## LINUX

## Name

`dynamic_irq_init` — initialize a dynamically allocated irq

## Synopsis

```
void dynamic_irq_init (unsigned int irq);
```

## Arguments

*irq*

irq number to initialize

## dynamic\_irq\_cleanup

### LINUX

Kernel Hackers Manual July 2010

## Name

`dynamic_irq_cleanup` — cleanup a dynamically allocated irq

## Synopsis

```
void dynamic_irq_cleanup (unsigned int irq);
```

## Arguments

*irq*

irq number to initialize

# set\_irq\_msi

## LINUX

Kernel Hackers Manual July 2010

### Name

`set_irq_msi` — set irq type data for an irq

### Synopsis

```
int set_irq_msi (unsigned int irq, struct msi_desc * entry);
```

### Arguments

*irq*

Interrupt number

*entry*

Pointer to MSI descriptor data

### Description

Set the hardware irq controller data for an irq

# handle\_simple\_irq

## LINUX

## Name

`handle_simple_irq` — Simple and software-decoded IRQs.

## Synopsis

```
void handle_simple_irq (unsigned int irq, struct irq_desc *  
desc);
```

## Arguments

*irq*

the interrupt number

*desc*

the interrupt description structure for this irq

## Description

Simple interrupts are either sent from a demultiplexing interrupt handler or come from hardware, where no interrupt hardware control is necessary.

## Note

The caller is expected to handle the ack, clear, mask and unmask issues if necessary.

## `handle_level_irq`

**LINUX**

## Name

`handle_level_irq` — Level type irq handler

## Synopsis

```
void handle_level_irq (unsigned int irq, struct irq_desc *  
desc);
```

## Arguments

*irq*

the interrupt number

*desc*

the interrupt description structure for this irq

## Description

Level type interrupts are active as long as the hardware line has the active level. This may require to mask the interrupt and unmask it after the associated handler has acknowledged the device, so the interrupt line is back to inactive.

## `handle_fasteoi_irq`

**LINUX**

## Name

`handle_fasteoi_irq` — irq handler for transparent controllers

## Synopsis

```
void handle_fasteoi_irq (unsigned int irq, struct irq_desc *  
desc);
```

## Arguments

*irq*

the interrupt number

*desc*

the interrupt description structure for this irq

## Only a single callback will be issued to the chip

an `->eoi` call when the interrupt has been serviced. This enables support for modern forms of interrupt handlers, which handle the flow details in hardware, transparently.

## `handle_edge_irq`

**LINUX**

## Name

`handle_edge_irq` — edge type IRQ handler

## Synopsis

```
void handle_edge_irq (unsigned int irq, struct irq_desc *  
desc);
```

## Arguments

*irq*

the interrupt number

*desc*

the interrupt description structure for this irq

## Description

Interrupt occurs on the falling and/or rising edge of a hardware signal. The occurrence is latched into the irq controller hardware and must be acked in order to be reenabled. After the ack another interrupt can happen on the same source even before the first one is handled by the associated event handler. If this happens it might be necessary to disable (mask) the interrupt depending on the controller hardware. This requires to reenale the interrupt inside of the loop which handles the interrupts which have arrived while the handler was running. If all pending interrupts are handled, the loop is left.

# handle\_percpu\_irq

## LINUX

Kernel Hackers Manual July 2010

### Name

`handle_percpu_irq` — Per CPU local irq handler

### Synopsis

```
void handle_percpu_irq (unsigned int irq, struct irq_desc *  
desc);
```

### Arguments

*irq*

the interrupt number

*desc*

the interrupt description structure for this irq

### Description

Per CPU interrupts on SMP machines without locking requirements



# Chapter 10. Credits

The following people have contributed to this document:

1. Thomas Gleixner<tglx@linutronix.de>
2. Ingo Molnar<mingo@elte.hu>

