

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 © 2005-2010 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

1. Introduction.....	1
2. Rationale	3
3. Known Bugs And Assumptions	5
4. Abstraction layers	7
4.1. Interrupt control flow	7
4.2. High-level Driver API.....	7
4.3. High-level 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 Fast EOI IRQ flow handler	9
4.3.2.3. Default Edge IRQ flow handler	10
4.3.2.4. Default simple IRQ flow handler	10
4.3.2.5. Default per CPU flow handler.....	10
4.3.2.6. EOI Edge IRQ flow handler.....	11
4.3.2.7. Bad IRQ flow handler	11
4.3.3. Quirks and optimizations	11
4.3.4. Delayed interrupt disable	11
4.4. Chip-level hardware encapsulation	12
5. __do_IRQ entry point.....	13
6. Locking on SMP.....	15
7. Generic interrupt chip.....	17
irq_gc_mask_set_bit	17
irq_gc_mask_clr_bit	17
irq_gc_ack_set_bit	18
irq_alloc_generic_chip.....	19
irq_alloc_domain_generic_chips	20
irq_get_domain_generic_chip.....	21
irq_setup_generic_chip	22
irq_setup_alt_chip.....	23
irq_remove_generic_chip.....	24
8. Structures.....	27
struct irq_data.....	27
struct irq_chip	28
struct irq_chip_regs.....	32
struct irq_chip_type	33
struct irq_chip_generic.....	34
enum irq_gc_flags.....	36

struct irqaction	37
struct irq_affinity_notify	39
irq_set_affinity	40
irq_force_affinity	41
9. Public Functions Provided	43
synchronize_hardirq.....	43
synchronize_irq.....	44
irq_set_affinity_notifier.....	44
disable_irq_nosync	45
disable_irq.....	46
enable_irq.....	47
irq_set_irq_wake.....	48
irq_wake_thread.....	49
setup_irq.....	50
remove_irq	51
free_irq	51
request_threaded_irq.....	52
request_any_context_irq	54
irq_set_chip.....	55
irq_set_irq_type	56
irq_set_handler_data.....	57
irq_set_chip_data	58
handle_simple_irq.....	59
handle_level_irq.....	60
handle_edge_irq.....	61
10. Internal Functions Provided	63
irq_get_next_irq	63
handle_bad_irq.....	63
irq_set_msi_desc_off	64
irq_set_msi_desc.....	65
irq_disable.....	66
handle_fasteoi_irq.....	67
handle_edge_eoi_irq	68
handle_percpu_irq.....	69
handle_percpu_devid_irq.....	70
irq_cpu_online	71
irq_cpu_offline.....	71
11. Credits.....	73

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 uses 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

During the implementation we identified another type:

- Fast EOI type

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

- Per CPU type

This split implementation of high-level IRQ handlers allows us to optimize the flow of the interrupt handling for each specific interrupt type. This reduces complexity in that particular code path 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 low-level hardware logic, and it also leads to unnecessary code duplication: for example in i386, there is an `ioapic_level_irq` and an `ioapic_edge_irq` IRQ-type which share many of the low-level 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 high-level flow handler, which is normally one of the generic implementations. (This high-level flow handler implementation also makes it simple to provide demultiplexing handlers which can be found in embedded platforms on various architectures.)

Chapter 2. Rationale

The separation makes the generic interrupt handling layer more flexible and extensible. For example, an (sub)architecture can use a generic IRQ-flow 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. It's deprecated for three years now and about to be removed.

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. High-level driver API
2. High-level IRQ flow handlers
3. Chip-level 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 low-level architecture code calls into the generic interrupt code by calling `desc->handle_irq()`. This high-level IRQ handling function only uses `desc->irq_data.chip` primitives referenced by the assigned chip descriptor structure.

4.2. High-level Driver API

The high-level Driver API consists of following functions:

- `request_irq()`
- `free_irq()`
- `disable_irq()`
- `enable_irq()`
- `disable_irq_nosync()` (SMP only)
- `synchronize_irq()` (SMP only)
- `irq_set_irq_type()`
- `irq_set_irq_wake()`

- `irq_set_handler_data()`
- `irq_set_chip()`
- `irq_set_chip_data()`

See the autogenerated function documentation for details.

4.3. High-level IRQ flow handlers

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

- `handle_level_irq`
- `handle_edge_irq`
- `handle_fasteoi_irq`
- `handle_simple_irq`
- `handle_percpu_irq`
- `handle_edge_eoi_irq`
- `handle_bad_irq`

The interrupt flow handlers (either pre-defined 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(struct irq_data *data)
{
    desc->irq_data.chip->irq_unmask(data);
}

default_disable(struct irq_data *data)
{
    if (!delay_disable(data))
        desc->irq_data.chip->irq_mask(data);
}
```

```

}

default_ack(struct irq_data *data)
{
    chip->irq_ack(data);
}

default_mask_ack(struct irq_data *data)
{
    if (chip->irq_mask_ack) {
        chip->irq_mask_ack(data);
    } else {
        chip->irq_mask(data);
        chip->irq_ack(data);
    }
}

noop(struct irq_data *data)
{
}

```

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->irq_data.chip->irq_mask_ack();
handle_irq_event(desc->action);
desc->irq_data.chip->irq_unmask();

```

4.3.2.2. Default Fast EOI IRQ flow handler

`handle_fasteoi_irq` provides a generic implementation for interrupts, which only need an EOI at the end of the handler.

The following control flow is implemented (simplified excerpt):

```
handle_irq_event(desc->action);
desc->irq_data.chip->irq_eoi();
```

4.3.2.3. 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->irq_data.chip->irq_mask_ack();
    desc->status |= pending | masked;
    return;
}
desc->irq_data.chip->irq_ack();
desc->status |= running;
do {
    if (desc->status & masked)
        desc->irq_data.chip->irq_unmask();
    desc->status &= ~pending;
    handle_irq_event(desc->action);
} while (status & pending);
desc->status &= ~running;
```

4.3.2.4. 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);
```

4.3.2.5. 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):

```
if (desc->irq_data.chip->irq_ack)
    desc->irq_data.chip->irq_ack();
handle_irq_event(desc->action);
if (desc->irq_data.chip->irq_eoi)
    desc->irq_data.chip->irq_eoi();
```

4.3.2.6. EOI Edge IRQ flow handler

`handle_edge_eoi_irq` provides an abomination of the edge handler which is solely used to tame a badly wreckaged irq controller on powerpc/cell.

4.3.2.7. Bad IRQ flow handler

`handle_bad_irq` is used for spurious interrupts which have no real handler assigned..

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 high-level 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 is not configurable.

4.4. Chip-level 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.

- `irq_ack()`
- `irq_mask_ack()` - Optional, recommended for performance
- `irq_mask()`
- `irq_unmask()`
- `irq_eoi()` - Optional, required for EOI flow handlers
- `irq_retrigger()` - Optional
- `irq_set_type()` - Optional
- `irq_set_wake()` - Optional

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 low-level functionality.

Chapter 5. `__do_IRQ` entry point

The original implementation `__do_IRQ()` was an alternative entry point for all types of interrupts. It no longer exists.

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

Chapter 6. Locking on SMP

The locking of chip registers is up to the architecture that defines the chip primitives. The per-irq structure is protected via desc->lock, by the generic layer.

Chapter 7. Generic interrupt chip

To avoid copies of identical implementations of IRQ chips the core provides a configurable generic interrupt chip implementation. Developers should check carefully whether the generic chip fits their needs before implementing the same functionality slightly differently themselves.

irq_gc_mask_set_bit

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_gc_mask_set_bit` — Mask chip via setting bit in mask register

Synopsis

```
void irq_gc_mask_set_bit (struct irq_data * d);
```

Arguments

d

irq_data

Description

Chip has a single mask register. Values of this register are cached and protected by `gc->lock`

irq_gc_mask_clr_bit

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_gc_mask_clr_bit` — Mask chip via clearing bit in mask register

Synopsis

```
void irq_gc_mask_clr_bit (struct irq_data * d);
```

Arguments

d

irq_data

Description

Chip has a single mask register. Values of this register are cached and protected by `gc->lock`

irq_gc_ack_set_bit

LINUX

Name

`irq_gc_ack_set_bit` — Ack pending interrupt via setting bit

Synopsis

```
void irq_gc_ack_set_bit (struct irq_data * d);
```

Arguments

d

`irq_data`

irq_alloc_generic_chip

LINUX

Name

`irq_alloc_generic_chip` — Allocate a generic chip and initialize it

Synopsis

```
struct irq_chip_generic * irq_alloc_generic_chip (const char *  
name, int num_ct, unsigned int irq_base, void __iomem *  
reg_base, irq_flow_handler_t handler);
```

Arguments

name

Name of the irq chip

num_ct

Number of irq_chip_type instances associated with this

irq_base

Interrupt base nr for this chip

reg_base

Register base address (virtual)

handler

Default flow handler associated with this chip

Description

Returns an initialized irq_chip_generic structure. The chip defaults to the primary (index 0) irq_chip_type and *handler*

irq_alloc_domain_generic_chips

LINUX

Kernel Hackers ManualSeptember 2014

Name

irq_alloc_domain_generic_chips — Allocate generic chips for an irq domain

Synopsis

```
int irq_alloc_domain_generic_chips (struct irq_domain * d, int
    irqs_per_chip, int num_ct, const char * name,
    irq_flow_handler_t handler, unsigned int clr, unsigned int
    set, enum irq_gc_flags gcflags);
```

Arguments

d

irq domain for which to allocate chips

irqs_per_chip

Number of interrupts each chip handles

num_ct

Number of `irq_chip_type` instances associated with this

name

Name of the irq chip

handler

Default flow handler associated with these chips

clr

IRQ_* bits to clear in the mapping function

set

IRQ_* bits to set in the mapping function

gcflags

Generic chip specific setup flags

irq_get_domain_generic_chip

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_get_domain_generic_chip` — Get a pointer to the generic chip of a `hw_irq`

Synopsis

```
struct irq_chip_generic * irq_get_domain_generic_chip (struct  
irq_domain * d, unsigned int hw_irq);
```

Arguments

d

irq domain pointer

hw_irq

Hardware interrupt number

irq_setup_generic_chip

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_setup_generic_chip` — Setup a range of interrupts with a generic chip

Synopsis

```
void irq_setup_generic_chip (struct irq_chip_generic * gc, u32
msk, enum irq_gc_flags flags, unsigned int clr, unsigned int
set);
```

Arguments

gc

Generic irq chip holding all data

msk

Bitmask holding the irqs to initialize relative to *gc->irq_base*

flags

Flags for initialization

clr

IRQ_* bits to clear

set

IRQ_* bits to set

Description

Set up max. 32 interrupts starting from *gc->irq_base*. Note, this initializes all interrupts to the primary *irq_chip_type* and its associated handler.

irq_setup_alt_chip

LINUX

Name

`irq_setup_alt_chip` — Switch to alternative chip

Synopsis

```
int irq_setup_alt_chip (struct irq_data * d, unsigned int
type);
```

Arguments

d

irq_data for this interrupt

type

Flow type to be initialized

Description

Only to be called from `chip->irq_set_type` callbacks.

irq_remove_generic_chip

LINUX

Name

`irq_remove_generic_chip` — Remove a chip

Synopsis

```
void irq_remove_generic_chip (struct irq_chip_generic * gc,  
u32 msk, unsigned int clr, unsigned int set);
```

Arguments

gc

Generic irq chip holding all data

msk

Bitmask holding the irqs to initialize relative to *gc->irq_base*

clr

IRQ_* bits to clear

set

IRQ_* bits to set

Description

Remove up to 32 interrupts starting from *gc->irq_base*.

Chapter 8. Structures

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

struct irq_data

LINUX

Kernel Hackers ManualSeptember 2014

Name

struct irq_data — per irq and irq chip data passed down to chip functions

Synopsis

```
struct irq_data {
    u32 mask;
    unsigned int irq;
    unsigned long hwirq;
    unsigned int node;
    unsigned int state_use_accessors;
    struct irq_chip * chip;
    struct irq_domain * domain;
    void * handler_data;
    void * chip_data;
    struct msi_desc * msi_desc;
    cpumask_var_t affinity;
};
```

Members

mask

precomputed bitmask for accessing the chip registers

irq

interrupt number

`hwirq`

hardware interrupt number, local to the interrupt domain

`node`

node index useful for balancing

`state_use_accessors`

status information for irq chip functions. Use accessor functions to deal with it

`chip`

low level interrupt hardware access

`domain`

Interrupt translation domain; responsible for mapping between hwirq number and linux irq number.

`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

`msi_desc`

MSI descriptor

`affinity`

IRQ affinity on SMP

Description

The fields here need to overlay the ones in `irq_desc` until we cleaned up the direct references and switched everything over to `irq_data`.

struct irq_chip

LINUX

Kernel Hackers Manual September 2014

Name

struct irq_chip — hardware interrupt chip descriptor

Synopsis

```
struct irq_chip {
    const char * name;
    unsigned int (* irq_startup) (struct irq_data *data);
    void (* irq_shutdown) (struct irq_data *data);
    void (* irq_enable) (struct irq_data *data);
    void (* irq_disable) (struct irq_data *data);
    void (* irq_ack) (struct irq_data *data);
    void (* irq_mask) (struct irq_data *data);
    void (* irq_mask_ack) (struct irq_data *data);
    void (* irq_unmask) (struct irq_data *data);
    void (* irq_eoi) (struct irq_data *data);
    int (* irq_set_affinity) (struct irq_data *data, const struct cpumask *cpumask);
    int (* irq_retrigger) (struct irq_data *data);
    int (* irq_set_type) (struct irq_data *data, unsigned int flow_type);
    int (* irq_set_wake) (struct irq_data *data, unsigned int on);
    void (* irq_bus_lock) (struct irq_data *data);
    void (* irq_bus_sync_unlock) (struct irq_data *data);
    void (* irq_cpu_online) (struct irq_data *data);
    void (* irq_cpu_offline) (struct irq_data *data);
    void (* irq_suspend) (struct irq_data *data);
    void (* irq_resume) (struct irq_data *data);
    void (* irq_pm_shutdown) (struct irq_data *data);
    void (* irq_calc_mask) (struct irq_data *data);
    void (* irq_print_chip) (struct irq_data *data, struct seq_file *p);
    int (* irq_request_resources) (struct irq_data *data);
    void (* irq_release_resources) (struct irq_data *data);
    unsigned long flags;
};
```

Members

name

name for /proc/interrupts

irq_startup

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

irq_shutdown

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

irq_enable

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

irq_disable

disable the interrupt

irq_ack

start of a new interrupt

irq_mask

mask an interrupt source

irq_mask_ack

ack and mask an interrupt source

irq_unmask

unmask an interrupt source

irq_eoi

end of interrupt

irq_set_affinity

set the CPU affinity on SMP machines

irq_retrigger

resend an IRQ to the CPU

irq_set_type

set the flow type (IRQ_TYPE_LEVEL/etc.) of an IRQ

`irq_set_wake`

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

`irq_bus_lock`

function to lock access to slow bus (i2c) chips

`irq_bus_sync_unlock`

function to sync and unlock slow bus (i2c) chips

`irq_cpu_online`

configure an interrupt source for a secondary CPU

`irq_cpu_offline`

un-configure an interrupt source for a secondary CPU

`irq_suspend`

function called from core code on suspend once per chip

`irq_resume`

function called from core code on resume once per chip

`irq_pm_shutdown`

function called from core code on shutdown once per chip

`irq_calc_mask`

Optional function to set `irq_data.mask` for special cases

`irq_print_chip`

optional to print special chip info in `show_interrupts`

`irq_request_resources`

optional to request resources before calling any other callback related to this
`irq`

`irq_release_resources`

optional to release resources acquired with `irq_request_resources`

`flags`

chip specific flags

struct irq_chip_regs

LINUX

Kernel Hackers Manual September 2014

Name

struct irq_chip_regs — register offsets for struct irq_gci

Synopsis

```
struct irq_chip_regs {
    unsigned long enable;
    unsigned long disable;
    unsigned long mask;
    unsigned long ack;
    unsigned long eoi;
    unsigned long type;
    unsigned long polarity;
};
```

Members

enable

Enable register offset to reg_base

disable

Disable register offset to reg_base

mask

Mask register offset to reg_base

ack

Ack register offset to reg_base

eoi

Eoi register offset to reg_base

type

Type configuration register offset to reg_base

polarity

Polarity configuration register offset to reg_base

struct irq_chip_type

LINUX

Kernel Hackers Manual September 2014

Name

struct irq_chip_type — Generic interrupt chip instance for a flow type

Synopsis

```
struct irq_chip_type {
    struct irq_chip chip;
    struct irq_chip_regs regs;
    irq_flow_handler_t handler;
    u32 type;
    u32 mask_cache_priv;
    u32 * mask_cache;
};
```

Members

chip

The real interrupt chip which provides the callbacks

regs

Register offsets for this chip

handler

Flow handler associated with this chip

type

Chip can handle these flow types

mask_cache_priv

Cached mask register private to the chip type

mask_cache

Pointer to cached mask register

Description

A `irq_generic_chip` can have several instances of `irq_chip_type` when it requires different functions and register offsets for different flow types.

struct irq_chip_generic

LINUX

Kernel Hackers ManualSeptember 2014

Name

`struct irq_chip_generic` — Generic irq chip data structure

Synopsis

```
struct irq_chip_generic {
    raw_spinlock_t lock;
    void __iomem * reg_base;
    unsigned int irq_base;
    unsigned int irq_cnt;
    u32 mask_cache;
    u32 type_cache;
    u32 polarity_cache;
```

```
u32 wake_enabled;  
u32 wake_active;  
unsigned int num_ct;  
void * private;  
unsigned long installed;  
unsigned long unused;  
struct irq_domain * domain;  
struct list_head list;  
struct irq_chip_type chip_types[0];  
};
```

Members

lock

Lock to protect register and cache data access

reg_base

Register base address (virtual)

irq_base

Interrupt base nr for this chip

irq_cnt

Number of interrupts handled by this chip

mask_cache

Cached mask register shared between all chip types

type_cache

Cached type register

polarity_cache

Cached polarity register

wake_enabled

Interrupt can wakeup from suspend

wake_active

Interrupt is marked as an wakeup from suspend source

`num_ct`

Number of available `irq_chip_type` instances (usually 1)

`private`

Private data for non generic chip callbacks

`installed`

bitfield to denote installed interrupts

`unused`

bitfield to denote unused interrupts

`domain`

irq domain pointer

`list`

List head for keeping track of instances

`chip_types[0]`

Array of interrupt `irq_chip_types`

Description

Note, that `irq_chip_generic` can have multiple `irq_chip_type` implementations which can be associated to a particular irq line of an `irq_chip_generic` instance. That allows to share and protect state in an `irq_chip_generic` instance when we need to implement different flow mechanisms (level/edge) for it.

enum irq_gc_flags

LINUX

Kernel Hackers Manual September 2014

Name

`enum irq_gc_flags` — Initialization flags for generic irq chips

Synopsis

```
enum irq_gc_flags {
    IRQ_GC_INIT_MASK_CACHE,
    IRQ_GC_INIT_NESTED_LOCK,
    IRQ_GC_MASK_CACHE_PER_TYPE,
    IRQ_GC_NO_MASK
};
```

Constants

`IRQ_GC_INIT_MASK_CACHE`

Initialize the mask_cache by reading mask reg

`IRQ_GC_INIT_NESTED_LOCK`

Set the lock class of the irqs to nested for irq chips which need to call `irq_set_wake` on the parent irq. Usually GPIO implementations

`IRQ_GC_MASK_CACHE_PER_TYPE`

Mask cache is chip type private

`IRQ_GC_NO_MASK`

Do not calculate `irq_data->mask`

struct irqaction

LINUX

Kernel Hackers Manual September 2014

Name

`struct irqaction` — per interrupt action descriptor

Synopsis

```
struct irqaction {
    irq_handler_t handler;
    void * dev_id;
    void __percpu * percpu_dev_id;
    struct irqaction * next;
    irq_handler_t thread_fn;
    struct task_struct * thread;
    unsigned int irq;
    unsigned int flags;
    unsigned long thread_flags;
    unsigned long thread_mask;
    const char * name;
    struct proc_dir_entry * dir;
};
```

Members

handler

interrupt handler function

dev_id

cookie to identify the device

percpu_dev_id

cookie to identify the device

next

pointer to the next irqaction for shared interrupts

thread_fn

interrupt handler function for threaded interrupts

thread

thread pointer for threaded interrupts

irq

interrupt number

flags

flags (see IRQF_* above)

thread_flags

flags related to *thread*

thread_mask

bitmask for keeping track of *thread* activity

name

name of the device

dir

pointer to the proc/irq/NN/name entry

struct irq_affinity_notify

LINUX

Kernel Hackers Manual September 2014

Name

struct irq_affinity_notify — context for notification of IRQ affinity changes

Synopsis

```
struct irq_affinity_notify {
    unsigned int irq;
    struct kref kref;
    struct work_struct work;
    void (* notify) (struct irq_affinity_notify *, const cpumask_t *mask);
    void (* release) (struct kref *ref);
};
```

Members

`irq`

Interrupt to which notification applies

`kref`

Reference count, for internal use

`work`

Work item, for internal use

`notify`

Function to be called on change. This will be called in process context.

`release`

Function to be called on release. This will be called in process context. Once registered, the structure must only be freed when this function is called or later.

`irq_set_affinity`

LINUX

Kernel Hackers Manual September 2014

Name

`irq_set_affinity` — Set the irq affinity of a given irq

Synopsis

```
int irq_set_affinity (unsigned int irq, const struct cpumask *  
cpumask);
```

Arguments

irq

Interrupt to set affinity

cpumask

cpumask

Description

Fails if cpumask does not contain an online CPU

irq_force_affinity

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_force_affinity` — Force the irq affinity of a given irq

Synopsis

```
int irq_force_affinity (unsigned int irq, const struct cpumask
* cpumask);
```

Arguments

irq

Interrupt to set affinity

cpumask

`cpumask`

Description

Same as `irq_set_affinity`, but without checking the mask against online cpus.

Solely for low level cpu hotplug code, where we need to make per cpu interrupts affine before the cpu becomes online.

Chapter 9. Public Functions Provided

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

synchronize_hardirq

LINUX

Kernel Hackers Manual September 2014

Name

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

Synopsis

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

Arguments

irq

interrupt number to wait for

Description

This function waits for any pending hard 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. It does not take associated threaded handlers into account.

Do not use this for shutdown scenarios where you must be sure that all parts (`hardirq` and threaded handler) have completed.

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

synchronize_irq

LINUX

Kernel Hackers ManualSeptember 2014

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.

irq_set_affinity_notifier

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_set_affinity_notifier` — control notification of IRQ affinity changes

Synopsis

```
int irq_set_affinity_notifier (unsigned int irq, struct  
irq_affinity_notify * notify);
```

Arguments

irq

Interrupt for which to enable/disable notification

notify

Context for notification, or `NULL` to disable notification. Function pointers must be initialised; the other fields will be initialised by this function.

Description

Must be called in process context. Notification may only be enabled after the IRQ is allocated and must be disabled before the IRQ is freed using `free_irq`.

disable_irq_nosync

LINUX

Kernel Hackers ManualSeptember 2014

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 only when `desc->irq_data.chip->bus_lock` and `desc->chip->bus_sync_unlock` are NULL !

irq_set_irq_wake

LINUX

Kernel Hackers Manual September 2014

Name

`irq_set_irq_wake` — control irq power management wakeup

Synopsis

```
int irq_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”.

irq_wake_thread

LINUX

Kernel Hackers Manual September 2014

Name

`irq_wake_thread` — wake the irq thread for the action identified by `dev_id`

Synopsis

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

Arguments

irq

Interrupt line

dev_id

Device identity for which the thread should be woken

setup_irq

LINUX

Kernel Hackers Manual September 2014

Name

`setup_irq` — setup an interrupt

Synopsis

```
int setup_irq (unsigned int irq, struct irqaction * act);
```

Arguments

irq

Interrupt line to setup

act

irqaction for the interrupt

Description

Used to statically setup interrupts in the early boot process.

remove_irq

LINUX

Kernel Hackers ManualSeptember 2014

Name

`remove_irq` — free an interrupt

Synopsis

```
void remove_irq (unsigned int irq, struct irqaction * act);
```

Arguments

irq

Interrupt line to free

act

irqaction for the interrupt

Description

Used to remove interrupts statically setup by the early boot process.

free_irq

LINUX

Kernel Hackers ManualSeptember 2014

Name

`free_irq` — free an interrupt allocated with `request_irq`

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_threaded_irq

LINUX

Kernel Hackers Manual September 2014

Name

`request_threaded_irq` — allocate an interrupt line

Synopsis

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

Arguments

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

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.

If you want to set up a threaded irq handler for your device then you need to supply *handler* and *thread_fn*. *handler* is still called in hard interrupt context and has to check whether the interrupt originates from the device. If yes it needs to disable the interrupt on the device and return `IRQ_WAKE_THREAD` which will wake up the handler thread and run *thread_fn*. This split handler design is necessary to support shared interrupts.

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_TRIGGER_*` Specify active edge(s) or level

request_any_context_irq

LINUX

Kernel Hackers Manual September 2014

Name

`request_any_context_irq` — allocate an interrupt line

Synopsis

```
int request_any_context_irq (unsigned int irq, irq_handler_t  
handler, unsigned long flags, const char * name, void *  
dev_id);
```

Arguments

irq

Interrupt line to allocate

handler

Function to be called when the IRQ occurs. Threaded handler for threaded interrupts.

flags

Interrupt type flags

name

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. It selects either a hardirq or threaded handling method depending on the context.

On failure, it returns a negative value. On success, it returns either `IRQC_IS_HARDIRQ` or `IRQC_IS_NESTED`.

irq_set_chip

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_set_chip` — set the irq chip for an irq

Synopsis

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

Arguments

irq

irq number

chip

pointer to irq chip description structure

irq_set_irq_type

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_set_irq_type` — set the irq trigger type for an irq

Synopsis

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

Arguments

irq

irq number

type

IRQ_TYPE_{LEVEL,EDGE}_* value - see include/linux/irq.h

irq_set_handler_data

LINUX

Kernel Hackers Manual September 2014

Name

`irq_set_handler_data` — set irq handler data for an irq

Synopsis

```
int irq_set_handler_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

irq_set_chip_data

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_set_chip_data` — set irq chip data for an irq

Synopsis

```
int irq_set_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

handle_simple_irq

LINUX

Kernel Hackers ManualSeptember 2014

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

Kernel Hackers ManualSeptember 2014

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_edge_irq

LINUX

Kernel Hackers Manual September 2014

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

Chapter 9. Public Functions Provided

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.

Chapter 10. Internal Functions Provided

This chapter contains the autogenerated documentation of the internal functions.

irq_get_next_irq

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_get_next_irq` — get next allocated irq number

Synopsis

```
unsigned int irq_get_next_irq (unsigned int offset);
```

Arguments

offset

where to start the search

Description

Returns next irq number after offset or nr_irqs if none is found.

handle_bad_irq

LINUX

Kernel Hackers ManualSeptember 2014

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.

irq_set_msi_desc_off

LINUX

Name

`irq_set_msi_desc_off` — set MSI descriptor data for an irq at offset

Synopsis

```
int irq_set_msi_desc_off (unsigned int irq_base, unsigned int  
irq_offset, struct msi_desc * entry);
```

Arguments

irq_base

Interrupt number base

irq_offset

Interrupt number offset

entry

Pointer to MSI descriptor data

Description

Set the MSI descriptor entry for an irq at offset

irq_set_msi_desc

LINUX

Name

`irq_set_msi_desc` — set MSI descriptor data for an irq

Synopsis

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

Arguments

irq

Interrupt number

entry

Pointer to MSI descriptor data

Description

Set the MSI descriptor entry for an irq

irq_disable

LINUX

Name

`irq_disable` — Mark interrupt disabled

Synopsis

```
void irq_disable (struct irq_desc * desc);
```

Arguments

desc

irq descriptor which should be disabled

Description

If the chip does not implement the `irq_disable` callback, we use a lazy disable approach. That means we mark the interrupt disabled, but leave the hardware unmasked. That's an optimization because we avoid the hardware access for the common case where no interrupt happens after we marked it disabled. If an interrupt happens, then the interrupt flow handler masks the line at the hardware level and marks it pending.

handle_fasteoi_irq

LINUX

Kernel Hackers Manual September 2014

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_eoi_irq

LINUX

Kernel Hackers Manual September 2014

Name

`handle_edge_eoi_irq` — edge eoi type IRQ handler

Synopsis

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


Arguments

irq

the interrupt number

desc

the interrupt description structure for this irq

Description

Similar as the above `handle_edge_irq`, but using `eoi` and w/o the mask/unmask logic.

handle_percpu_irq

LINUX

Kernel Hackers ManualSeptember 2014

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

handle_percpu_devid_irq

LINUX

Kernel Hackers ManualSeptember 2014

Name

`handle_percpu_devid_irq` — Per CPU local irq handler with per cpu dev
ids

Synopsis

```
void handle_percpu_devid_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. Same as `handle_percpu_irq` above but with the following extras:

action->percpu_dev_id is a pointer to percpu variables which contain the real device id for the cpu on which this handler is called

irq_cpu_online

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_cpu_online` — Invoke all `irq_cpu_online` functions.

Synopsis

```
void irq_cpu_online ( void);
```

Arguments

void

no arguments

Description

Iterate through all irqs and invoke the `chip.irq_cpu_online` for each.

irq_cpu_offline

LINUX

Kernel Hackers ManualSeptember 2014

Name

`irq_cpu_offline` — Invoke all `irq_cpu_offline` functions.

Synopsis

```
void irq_cpu_offline ( void );
```

Arguments

void

no arguments

Description

Iterate through all irqs and invoke the `chip.irq_cpu_offline` for each.

Chapter 11. Credits

The following people have contributed to this document:

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

