Linux generic IRQ handling

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.

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.)

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.

Known Bugs And Assumptions

None (knock on wood).

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

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.

High-level Driver API

The high-level Driver API consists of following functions:

- request_irq()
- request_threaded_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.

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.

Default flow implementations

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))
{
}
```

Default flow handler implementations

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();
```

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();
```

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;
```

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);
```

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();
```

EOI Edge IRQ flow handler

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

Bad IRQ flow handler

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

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.

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.

Chip-level hardware encapsulation

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

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]genericirq.rst, line 347); *backlink***
>
> Unknown interpreted text role "c:type".

- `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.

__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.

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.

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.

> **System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]genericirq.rst, line 398)**
>
> Unknown directive type "kernel-doc".
>
> ```
> .. kernel-doc:: kernel/irq/generic-chip.c
>    :export:
> ```

Structures

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]genericirq.rst, line 407)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/irq.h
   :internal:
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]genericirq.rst, line 410)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/interrupt.h
   :internal:
```

Public Functions Provided

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

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]genericirq.rst, line 419)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: kernel/irq/manage.c
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]genericirq.rst, line 421)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: kernel/irq/chip.c
   :export:
```

Internal Functions Provided

This chapter contains the autogenerated documentation of the internal functions.

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]genericirq.rst, line 430)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: kernel/irq/irqdesc.c
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]genericirq.rst, line 432)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: kernel/irq/handle.c
```

System Message: ERROR/3 (D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\core-api\[linux-master][Documentation][core-api]genericirq.rst, line 434)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: kernel/irq/chip.c
   :internal:
```

Credits

The following people have contributed to this document:

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