```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
	"http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" []>

<book id="Generic-IRQ-Guide">
 <bookinfo>
  <title>Linux generic IRQ handling</title>

  <authorgroup>
   <author>
    <firstname>Thomas</firstname>
    <surname>Gleixner</surname>
    <affiliation>
     <address>
      <email>tglx@linutronix.de</email>
     </address>
    </affiliation>
   </author>
   <author>
    <firstname>Ingo</firstname>
    <surname>Molnar</surname>
    <affiliation>
     <address>
      <email>mingo@elte.hu</email>
     </address>
    </affiliation>
   </author>
  </authorgroup>

  <copyright>
   <year>2005-2006</year>
   <holder>Thomas Gleixner</holder>
  </copyright>
  <copyright>
   <year>2005-2006</year>
   <holder>Ingo Molnar</holder>
  </copyright>

  <legalnotice>
   <para>
     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.
   </para>

   <para>
     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.
   </para>

   <para>
     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
    </para>

    <para>
      For more details see the file COPYING in the source
      distribution of Linux.
    </para>
  </legalnotice>
 </bookinfo>

<toc></toc>

  <chapter id="intro">
    <title>Introduction</title>
    <para>
        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.
    </para>
    <para>
        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.
    </para>
  </chapter>

  <chapter id="rationale">
    <title>Rationale</title>
        <para>
        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.
        </para>
        <para>
        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:
        <itemizedlist>
          <listitem><para>Level type</para></listitem>
          <listitem><para>Edge type</para></listitem>
          <listitem><para>Simple type</para></listitem>
        </itemizedlist>
        In the SMP world of the __do_IRQ() super-handler another type
        was identified:
        <itemizedlist>
          <listitem><para>Per CPU type</para></listitem>
        </itemizedlist>
        </para>
        <para>
        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.
     </para>
     <para>
     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.
     </para>
     <para>
     A more natural abstraction is the clean separation of the
     'irq flow' and the 'chip details'.
     </para>
     <para>
     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.
     </para>
     <para>
     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.)
     </para>
     <para>
     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.
     </para>
     <para>
     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.
     </para>
</chapter>
<chapter id="bugs">
  <title>Known Bugs And Assumptions</title>
  <para>
     None (knock on wood).
  </para>
</chapter>

<chapter id="Abstraction">
  <title>Abstraction layers</title>
  <para>
     There are three main levels of abstraction in the interrupt code:
```

```
    <orderedlist>
      <listitem><para>Highlevel driver API</para></listitem>
      <listitem><para>Highlevel IRQ flow handlers</para></listitem>
      <listitem><para>Chiplevel hardware encapsulation</para></listitem>
    </orderedlist>
</para>
<sect1 id="Interrupt_control_flow">
    <title>Interrupt control flow</title>
    <para>
    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 decription 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.
    </para>
    <para>
    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.
    </para>
</sect1>
<sect1 id="Highlevel_Driver_API">
    <title>Highlevel Driver API</title>
    <para>
      The highlevel Driver API consists of following functions:
      <itemizedlist>
      <listitem><para>request_irq()</para></listitem>
      <listitem><para>free_irq()</para></listitem>
      <listitem><para>disable_irq()</para></listitem>
      <listitem><para>enable_irq()</para></listitem>
      <listitem><para>disable_irq_nosync() (SMP only)</para></listitem>
      <listitem><para>synchronize_irq() (SMP only)</para></listitem>
      <listitem><para>set_irq_type()</para></listitem>
      <listitem><para>set_irq_wake()</para></listitem>
      <listitem><para>set_irq_data()</para></listitem>
      <listitem><para>set_irq_chip()</para></listitem>
      <listitem><para>set_irq_chip_data()</para></listitem>
      </itemizedlist>
      See the autogenerated function documentation for details.
    </para>
</sect1>
<sect1 id="Highlevel_IRQ_flow_handlers">
    <title>Highlevel IRQ flow handlers</title>
    <para>
      The generic layer provides a set of pre-defined irq-flow methods:
      <itemizedlist>
      <listitem><para>handle_level_irq</para></listitem>
      <listitem><para>handle_edge_irq</para></listitem>
      <listitem><para>handle_simple_irq</para></listitem>
      <listitem><para>handle_percpu_irq</para></listitem>
      </itemizedlist>
      The interrupt flow handlers (either predefined or architecture
      specific) are assigned to specific interrupts by the architecture
```

```
          either during bootup or during device initialization.
        </para>
        <sect2 id="Default_flow_implementations">
        <title>Default flow implementations</title>
            <sect3 id="Helper_functions">
                <title>Helper functions</title>
                <para>
                The helper functions call the chip primitives and
                are used by the default flow implementations.
                The following helper functions are implemented (simplified
excerpt):
                <programlisting>
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)
{
}


                </programlisting>
                </para>
            </sect3>
        </sect2>
        <sect2 id="Default_flow_handler_implementations">
        <title>Default flow handler implementations</title>
            <sect3 id="Default_Level_IRQ_flow_handler">
                <title>Default Level IRQ flow handler</title>
                <para>
                handle_level_irq provides a generic implementation
                for level-triggered interrupts.
                </para>
                <para>
                The following control flow is implemented (simplified excerpt):
```

```
                <programlisting>
desc->chip->start();
handle_IRQ_event(desc->action);
desc->chip->end();
                </programlisting>
                </para>
            </sect3>
            <sect3 id="Default_Edge_IRQ_flow_handler">
                <title>Default Edge IRQ flow handler</title>
                <para>
                handle_edge_irq provides a generic implementation
                for edge-triggered interrupts.
                </para>
                <para>
                The following control flow is implemented (simplified excerpt):
                <programlisting>
if (desc->status &amp; running) {
        desc->chip->hold();
        desc->status |= pending | masked;
        return;
}
desc->chip->start();
desc->status |= running;
do {
        if (desc->status &amp; masked)
                desc->chip->enable();
        desc->status &amp;= ~pending;
        handle_IRQ_event(desc->action);
} while (status &amp; pending);
desc->status &amp;= ~running;
desc->chip->end();
                </programlisting>
                </para>
            </sect3>
            <sect3 id="Default_simple_IRQ_flow_handler">
                <title>Default simple IRQ flow handler</title>
                <para>
                handle_simple_irq provides a generic implementation
                for simple interrupts.
                </para>
                <para>
                Note: The simple flow handler does not call any
                handler/chip primitives.
                </para>
                <para>
                The following control flow is implemented (simplified excerpt):
                <programlisting>
handle_IRQ_event(desc->action);
                </programlisting>
                </para>
            </sect3>
            <sect3 id="Default_per_CPU_flow_handler">
                <title>Default per CPU flow handler</title>
                <para>
                handle_percpu_irq provides a generic implementation
                for per CPU interrupts.
```

```
                </para>
                <para>
                Per CPU interrupts are only available on SMP and
                the handler provides a simplified version without
                locking.
                </para>
                <para>
                The following control flow is implemented (simplified excerpt):
                <programlisting>
desc->chip->start();
handle_IRQ_event(desc->action);
desc->chip->end();
                </programlisting>
                </para>
            </sect3>
        </sect2>
        <sect2 id="Quirks_and_optimizations">
        <title>Quirks and optimizations</title>
        <para>
        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.
        </para>
        </sect2>
        <sect2 id="Delayed_interrupt_disable">
        <title>Delayed interrupt disable</title>
        <para>
        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.
        </para>
        </sect2>
    </sect1>
    <sect1 id="Chiplevel_hardware_encapsulation">
        <title>Chiplevel hardware encapsulation</title>
        <para>
        The chip level hardware descriptor structure irq_chip
        contains all the direct chip relevant functions, which
        can be utilized by the irq flow implementations.
          <itemizedlist>
          <listitem><para>ack()</para></listitem>
          <listitem><para>mask_ack() - Optional, recommended for
```

```
performance</para></listitem>
          <listitem><para>mask()</para></listitem>
          <listitem><para>unmask()</para></listitem>
          <listitem><para>retrigger() - Optional</para></listitem>
          <listitem><para>set_type() - Optional</para></listitem>
          <listitem><para>set_wake() - Optional</para></listitem>
          </itemizedlist>
        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.
        </para>
    </sect1>
  </chapter>

  <chapter id="doirq">
     <title>__do_IRQ entry point</title>
     <para>
        The original implementation __do_IRQ() is an alternative entry
        point for all types of interrupts.
     </para>
     <para>
        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.
      </para>
      <para>
        To make use of the split implementation, replace the call to
        __do_IRQ by a call to desc->handle_irq() and associate
        the appropriate handler function to desc->handle_irq().
        In most cases the generic handler implementations should
        be sufficient.
     </para>
  </chapter>

  <chapter id="locking">
     <title>Locking on SMP</title>
     <para>
        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.
     </para>
  </chapter>
  <chapter id="structs">
     <title>Structures</title>
     <para>
     This chapter contains the autogenerated documentation of the structures
which are
     used in the generic IRQ layer.
     </para>
!Iinclude/linux/irq.h
!Iinclude/linux/interrupt.h
  </chapter>
```

```
  <chapter id="pubfunctions">
     <title>Public Functions Provided</title>
     <para>
     This chapter contains the autogenerated documentation of the kernel API
functions
      which are exported.
     </para>
!Ekernel/irq/manage.c
!Ekernel/irq/chip.c
  </chapter>

  <chapter id="intfunctions">
     <title>Internal Functions Provided</title>
     <para>
     This chapter contains the autogenerated documentation of the internal
functions.
     </para>
!Ikernel/irq/handle.c
!Ikernel/irq/chip.c
  </chapter>

  <chapter id="credits">
     <title>Credits</title>
        <para>
                The following people have contributed to this document:
                <orderedlist>
                        <listitem><para>Thomas
Gleixner<email>tglx@linutronix.de</email></para></listitem>
                        <listitem><para>Ingo
Molnar<email>mingo@elte.hu</email></para></listitem>
                </orderedlist>
        </para>
  </chapter>
</book>
```