Interrupt Management Under Linux

Understanding the Linux Interrupt Controller API

Bill Gatliff <bgat@billgatliff.com>

Overview

Interrupt handling is a fundamental part of the Linux kernel. Most of the kernel's functionality, in particular the parts of interest to embedded developers, in some way involve interrupt handling.

This paper describes the most important concepts related to the Linux kernel's interrupt handling mechanisms. These concepts include the relevant code and data structures. Sample code from Linux kernel version 2.6.12 is also provided.

`struct irqdesc` and `do_IRQ`

Each interrupt source available to the system has allocated to it a single `struct irqdesc` structure. This structure stores important information for the interrupt controller, handler and others:

```
struct irqdesc {
        irq_handler_t   handle;
        struct irqchip  *chip;
        struct irqaction *action;
        struct list_head pend;
        void            *chipdata;
        void            *data;
        unsigned int    disable_depth;
...
};

extern struct irqdesc irq_desc[];
```

The `handle` field points to a high-level "handler" for the interrupt line asserting the interrupt request. In ARM architectures, the handler is called `do_level_IRQ`:

```
void do_level_IRQ (unsigned int irq,
            struct irqdesc *desc, struct pt_regs *regs)
    {
            struct irqaction *action;
            const unsigned int cpu = smp_processor_id();

            desc->triggered = 1;

            /*
             * Acknowledge, clear _AND_ disable the interrupt.
             */
            desc->chip->ack(irq);

            if (likely(!desc->disable_depth)) {
                    kstat_cpu(cpu).irqs[irq]++;

                    smp_set_running(desc);

                    /*
                     * Return with this interrupt masked
                     * if no action
                     */
                    action = desc->action;
                    if (action) {
                            int ret = __do_irq(irq, desc->action, regs);

                            if (ret != IRQ_HANDLED)
                                    report_bad_irq(irq, regs, desc, ret);

                            if (likely(!desc->disable_depth &&
                                    !check_irq_lock(desc, irq, regs))
                                    desc->chip->unmask(irq);
                        }

                    smp_clear_running(desc);
            }
    }
```

`Do_level_IRQ` acknowledges the interrupt request, then invokes the list of device interrupt handlers registered with that interrupt source by calling `__do_irq`. We will come back to this function in a moment.

The `chip` field refers to the functions that manage the interrupt controller hardware. It is not uncommon for hardware that supports Linux to contain several different interrupt controllers, each having its own procedure for enabling, disabling and acknowledging interrupts. Most PCs have two interrupt controllers; some microcontrollers used in embedded applications have one or two built-in controllers, with one or more “external” interrupt controllers implemented using programmable logic devices.

The `struct irqchip` structure looks like this:

```
struct irqchip {
      void (*ack)(unsigned int);
      void (*mask)(unsigned int);
      void (*unmask)(unsigned int);
      int (*retrigger)(unsigned int);
      int (*type)(unsigned int, unsigned int);
      int (*wake)(unsigned int, unsigned int);
};
```

The `action` field of `struct irqdesc` maintains a list of `struct irqaction` structures, each of which represents a device interrupt handler that has registered with the interrupt request line using `request_irq`. The `__do_irq` function “walks” this list each time the interrupt request line is serviced:

```
static int
__do_irq(unsigned int irq, struct irqaction *action,
      struct pt_regs *regs)
{
      unsigned int status;
      int ret, retval = 0;

      spin_unlock(&irq_controller_lock);

#ifdef CONFIG_NO_IDLE_HZ
      if (!(action->flags & SA_TIMER)
            && system_timer->dyn_tick != NULL) {
            write_seqlock(&xtime_lock);
            if (system_timer->dyn_tick->state
               & DYN_TICK_ENABLED)
               system_timer->dyn_tick->handler(irq, 0, regs);
            write_sequnlock(&xtime_lock);
      }
#endif

      if (!(action->flags & SA_INTERRUPT))
            local_irq_enable();

      status = 0;
      do {
```

```
                ret = action->handler(irq, action->dev_id, regs);
                if (ret == IRQ_HANDLED)
                        status |= action->flags;
                retval |= ret;
                action = action->next;
        } while (action);

        if (status & SA_SAMPLE_RANDOM)
                add_interrupt_randomness(irq);

        spin_lock_irq(&irq_controller_lock);

        return retval;
}
```

The pend field is an ARM-specific field, used to make sure that pending interrupt requests are fully serviced during interrupt handling.

The `chipdata` field is another ARM-specific field, used by interrupt controller handlers as a private data pointer. Some ARM implementations like SA1111 use this field to store the physical address of the interrupt controller that manages the request line. Others, like AT91RM920, use this pointer to store the physical address of the GPIO controller that manages the line.

The `data` field is another ARM-specific field, used by device interrupt handlers as a private data pointer. Many device drivers uses this field to store a pointer to per-device data structures.

The `disable_depth` field keeps interrupt enable and disable requests balanced. An interrupt request line is not truly disabled until the number of disable requests matches the number of enable requests.

Handling an interrupt request

When the host microcontroller responds to an interrupt request, control first goes to a bit of assembly language code that knows how to store register values and other information critical to restoring the machine state after the interrupt is serviced. In

ARM machines, the function is called `__irq_svc`:

```
      .align      5
__irq_svc:
      svc_entry irq
...
1:    get_irqnr_and_base r0, r6, r5, lr
      movne r1, sp
      @
      @ routine called with r0 = irq number,
      @ r1 = struct pt_regs *
      @
      adrne lr, 1b
      bne   asm_do_IRQ
      ldr   r0, [sp, #S_PSR]        @ irqs are already disabled
      msr   spsr_cxsf, r0
      ldmia sp, {r0 – pc}^          @ load r0 - pc, cpsr
```

With the current processor state saved away, ARM machines then invoke `asm_do_IRQ`:

```
asmlinkage void asm_do_IRQ(unsigned int irq,
            struct pt_regs *regs)
{
      struct irqdesc *desc = irq_desc + irq;

      irq_enter();
      spin_lock(&irq_controller_lock);
      desc->handle(irq, desc, regs);

      /* Now re-run any pending interrupts. */
      if (!list_empty(&irq_pending))
            do_pending_irqs(regs);

      irq_finish(irq);
      spin_unlock(&irq_controller_lock);
      irq_exit();
}
```

The `desc->handle()` invocation calls `do_level_IRQ` or `do_edge_IRQ`, depending on the type of interrupt request line being serviced.

A basic interrupt handler

The Cogent Computer Systems, Inc. CSB637 single board computer has a pushbutton connected to GPIO PB29. When pressed, this pushbutton sends an edge-triggered interrupt to the GPIO controller, which forwards the request to the interrupt controller[1]. A simple “device interrupt handler” for this pushbutton might look like the following:

```
static int gpio_pb29_irqs;
static irqreturn_t gpio_pb29_irq (int irq, void *_cf,
                                  struct pt_regs *r)
{
  printk(KERN_ERR "%s: %d\n", __FUNCTION__, gpio_pb29_irqs);
  if (++gpio_pb29_irqs > 10)
    {
      printk(KERN_ERR "%s: disabling\n", __FUNCTION__);
      disable_irq(AT91_PIN_PB29);
    }
  return IRQ_HANDLED;
}
```

This code would be registered with the Linux kernel's `request_irq` function:

```
...
   request_irq(AT91_PIN_PB29, gpio_pb29_irq,
               SA_SHIRQ, "gpio_pb29", (void*)4);
...
```

`Probe_irq_on` and `probe_irq_off`

The Linux kernel's interrupt management system can help you determine which physical interrupt request line is assigned to your device. This feature is also useful for confirming that the interrupt request line is actually functioning before your system commits to using it.

1 A bit of magic further demultiplexes the interrupt request to a unique irqdesc. This magic is necessary because the AT91RM9200's GPIO controllers each have only one line leading to the chip's interrupt controller. This magic is well hidden from device interrupt handlers.

This code shows how to “probe” for an interrupt request line:

```
int mychip_probe_irq (void)
{
   unsigned long mask;
   int irq, retry = 4;

   while(retry--) {
      mask = probe_irq_on();

      /* do something that causes the device
         to generate an interrupt request */
      ...
      irq = probe_irq_off(mask);
      if (irq >= 0) break;
   }
   if (irq < 0) printk(KERN_INFO __FUNCTION__
        “: cannot identify interrupt line.\n”);
   return irq; /* to request_irq(...) */
}
```

“Virtual” interrupt descriptors

A multifunction chip might use one interrupt request line to signal interrupt requests from all of its onboard functions. Many multichannel UART chips have only a single interrupt request line leading back to the host microcontroller, for example.

Device drivers for multifunction chips can often be made more reusable and flexible if they can focus on only one feature of the target chip. The SM501 graphics processor has built-in AC97, UART and USBH, but only one interrupt request line back to the host processor. Rather than writing a combined video-plus-audio-plus-USB that would only be useful for that chip, with some effort it is possible to re-use the Linux kernel's standard framebuffer, audio and USB device drivers instead.

To do so, you must “demultiplex” the interrupt request line for the multifunction chip so that the signaling sub-component can be serviced by the right driver. Under Linux, this is done by creating “virtual interrupt descriptors” that look like unique interrupt sources for the purposes of device drivers:

```
#if defined(CONFIG_MY_MULTIFUNCTION_CHIP)
```

```
#define FUNCTION_A_IRQ 128 /* vector for function A */
#define FUNCTION_B_IRQ 129 /* vector for function B */
#endif

struct irqdesc irqdesc[]= {
... /* 0-127 */
#if defined(CONFIG_MY_MULTIFUNCTION_CHIP)
... /* 128 and 129 */
#endif
};
```

The device interrupt handler for the physical interrupt request line gets the interrupt request first. It then reads the interrupt status from the chip, and redirects the interrupt request to the descriptor associated with the function requesting service:

```
void mychip_irq_handler (unsigned irq,
            struct irqdesc *desc, ...)
{
   int ftn;
   unsigned long isr;

   /* read the chip's interrupt status register */
   isr = __raw_readl(desc->chipdata);

   /* map isr bits to associated entries in irq_desc[] */
   /* (TODO: what if multiple interrupts are asserted?) */
   switch (isr) {
      case 1: ftn = FUNCTION_A_IRQ; break;
      case 2: ftn = FUNCTION_B_IRQ; break;
      case 4: ftn = FUNCTION_C_IRQ; break;
   }
...
   irq_desc[ftn].handle(ftn, irq_desc + ftn, regs);
...
```

Conclusion

A clear understanding of the Linux kernel's interrupt handling mechanism is essential if you are to write solid, reusable device interrupt handlers. It is also mandatory if you are to successfully port Linux to custom hardware.