

Lecture 21: Interrupt Handling

Spring 2016
Jason Tang

Topics

- Hardware Interrupts
- Linux Interrupt Handling
- Writing Interrupt Handlers
- Top-Half / Bottom-Half Design

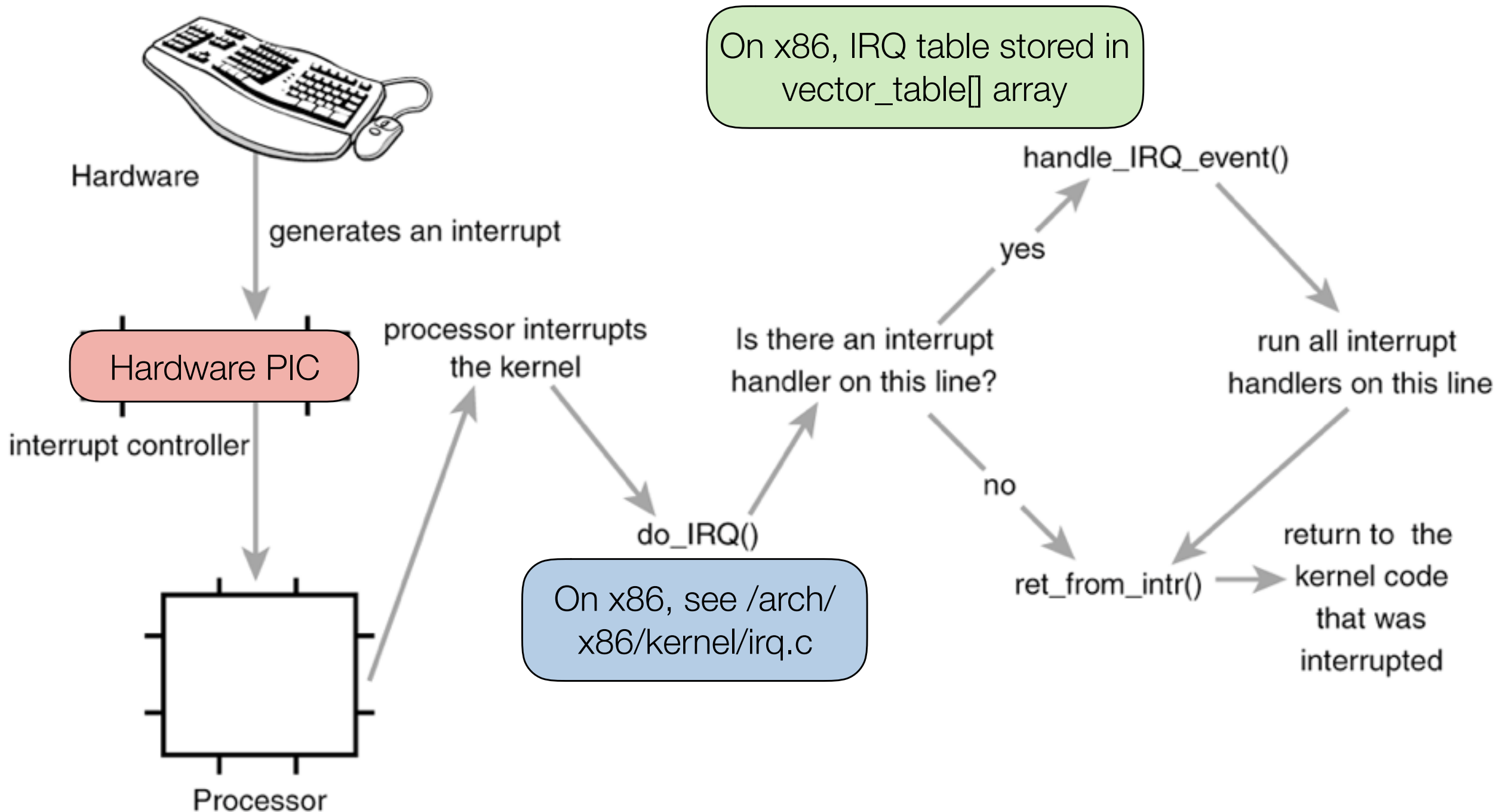
Hardware Interrupts

- Real device drivers have to deal with real hardware
- Device drivers usually written using event-based design:
 - Driver installs callbacks, and then enables hardware
 - Hardware generates interrupts **asynchronously**
 - Callbacks invoked to **handle** (or **service**) that interrupt

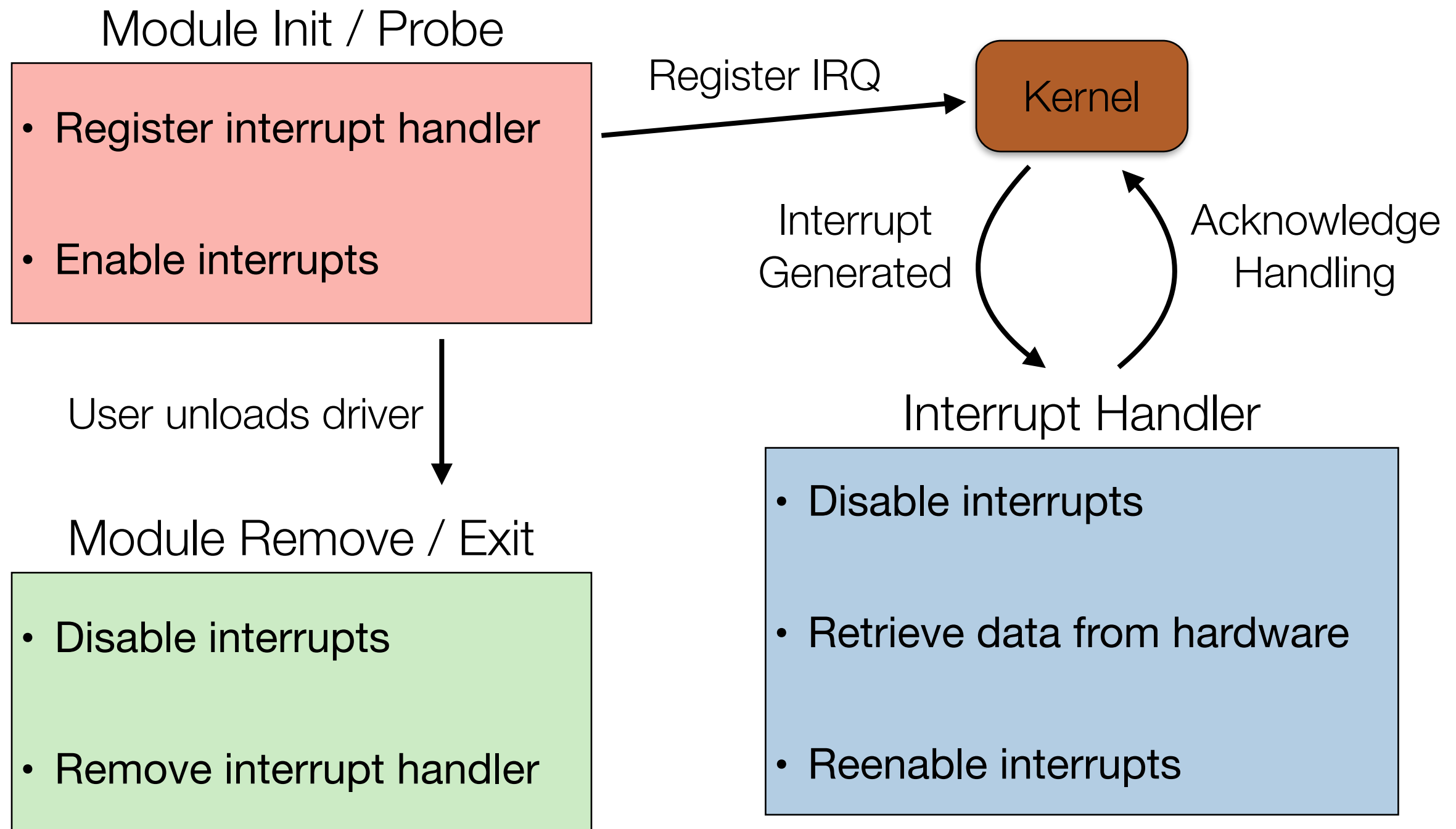
IRQ Handling

- Hardware sends an electrical signal on a physical **interrupt line**
- Processor detects that signal and translates it into an **interrupt request (IRQ)** number
- Processor then jumps to kernel's interrupt handling code
- Kernel searches through its **interrupt request table** (stored in RAM) for an entry that matches the IRQ
- If found, kernel jumps to the **interrupt service routine (ISR)** that was registered
- If not found, kernel ignores IRQ

IRQ Handling



Interrupt Handling Overview



Linux Interrupt Handling

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

- Declared in include/linux/interrupt.h
- First parameter is which IRQ number to register
- Second parameter is ISR to invoke upon interrupt
- Third parameter is flag(s) when registering IRQ (often set to 0)
- Fourth parameter is a free-form name for ISR
- Fifth parameter is the ISR [cookie](#)
- Returns 0 on success, negative on error

Example Interrupt Registration

```
if (request_irq(HP680_TS_IRQ, hp680_ts_interrupt,  
               0, MODNAME, NULL) < 0) {  
    printk(KERN_ERR "hp680_touchscreen.c: Can't allocate irq %d\n",  
           HP680_TS_IRQ);  
    err = -EBUSY;  
    goto fail1;  
}
```

From drivers/input/
touchscreen/hp680_ts_input.c

- String given as fourth parameter is the one shown in `/proc/interrupts`:

```
$ cat /proc/interrupts
```

	CPU0	CPU1	CPU2	CPU3		
0:	47	0	0	0	IO-APIC-edge	timer
1:	1818	0	0	0	IO-APIC-edge	i8042
6:	38	27	27	18	IO-APIC-edge	
8:	0	0	0	0	IO-APIC-edge	rtc0
9:	0	0	0	0	IO-APIC-fastEOI	acpi
12:	733	0	0	0	IO-APIC-edge	i8042

IRQ number

Number of times ISR was invoked

Name of ISR

Example ISR

```
static irqreturn_t hp680_ts_interrupt(int irq, void *dev)
{
    disable_irq_nosync(irq);
    schedule_delayed_work(&work, HZ / 20);

    return IRQ_HANDLED;
}
```

- First parameter to ISR is IRQ number that raised interrupt
- Second parameter is the cookie (same as last parameter to `request_irq()`)
- Return value is of type `irqreturn_t` (declared in `include/linux/irqreturn.h`)
 - `IRQ_NONE`: interrupt was not from this device (used when [sharing](#) IRQs)
 - `IRQ_HANDLED`: interrupt was handled by this device

Example Interrupt Freeing

```
static void __exit hp680_ts_exit(void)
{
    free_irq(HP680_TS_IRQ, NULL);
    cancel_delayed_work_sync(&work);
    input_unregister_device(hp680_ts_dev);
}
```

- First parameter to `free_irq()` is registered IRQ number (first parameter to `request_irq()`)
- Second parameter is the cookie (same as last parameter to `request_irq()`)
- If module does not unregister ISR, kernel will **panic** when interrupt is raised
 - Make sure ISRs are removed in module init/probe error paths

ISR Cookies

- Cookie usually points to some device-private data
- If driver is handling multiple instances of hardware, each hardware will be assigned different IRQ numbers
 - Pass a pointer to the `kmalloc()` device data within `request_irq()`

```
static int ili210x_i2c_probe(struct i2c_client *client,
                             const struct i2c_device_id *id)
{
    /* ... */
    struct ili210x *priv;
    /* ... */
    priv = kzalloc(sizeof(*priv), GFP_KERNEL);
    /* ... */
    error = request_irq(client->irq, ili210x_irq, pdata->irq_flags,
                        client->name, priv);
}
```

From drivers/input/
touchscreen/ili210x.c

Writing Interrupt Handlers

- While kernel is servicing interrupt, no other useful work is occurring
- Also while servicing interrupt, no other interrupts being serviced by that CPU
 - May miss interrupts, especially when running a real-time system
 - ISRs must finish quickly
- CPU disables preemption just before it calls into interrupt handling code
 - ISRs run in **interrupt context**, as compared to **process context**

Interrupt Context

- While running in interrupt context, ISR will not be preempted (on that CPU)
- ISR may not call functions that can sleep:
 - `kmalloc()` with `GFP_KERNEL` (**must** instead use `GFP_ATOMIC`)
 - `mutex_lock()`
 - `schedule_timeout()`
- **Calling any sleeping function will usually cause a kernel deadlock**

Top-Half / Bottom-Half

- As that ISR must run fast and is running in interrupt context, common pattern is to divide interrupt handling into two parts
- **Top-Half**: routine that responds to interrupt, running in interrupt context
 - Acknowledges interrupt
 - Wakes up a kthread to finish servicing interrupt
- **Bottom-Half**: routine that does actual work (delayed work)
 - As that it is a kthread, it runs in process context

Threaded Interrupt Handling

```
int request_threaded_irq(unsigned int irq,  
                        irq_handler_t handler,  
                        irq_handler_t thread_fn,  
                        unsigned long flags, const char *name,  
                        void *dev);
```

- Like `request_irq()`, but with additional parameter `thread_fn`
- If the `handler` function (top-half) returns `IRQ_WAKE_THREAD`, then kernel will automatically schedule a kthread to run `thread_fn` (bottom-half)
 - Top-half should disable interrupts on that device
 - Bottom-half should reenale interrupts after it has finished running

Example Threaded Interrupt Registration

```
err = request_threaded_irq(dev->dev->irq, b43_interrupt_handler,  
                           b43_interrupt_thread_handler,  
                           IRQF_SHARED, KBUILD_MODNAME, dev);  
  
if (err) {  
    b43err(dev->wl, "Cannot request IRQ-%d\n",  
            dev->dev->irq);  
    goto out;  
}
```

From drivers/net/wireless/
b43/main.c

- Register threaded interrupt handler similarly as `request_irq()`, with addition of parameter that specifies bottom-half
- Just like `request_irq()`, driver must call `free_irq()` at module exit / remove and also within error handling code in module init/probe

Example Top-Half

```
static irqreturn_t b43_interrupt_handler(int irq, void *dev_id)
{
    struct b43_wldev *dev = dev_id;
    irqreturn_t ret;

    if (unlikely(b43_status(dev) < B43_STAT_STARTED))
        return IRQ_NONE;

    spin_lock(&dev->wl->hardirq_lock);
    ret = b43_do_interrupt(dev);
    mmiowb();
    spin_unlock(&dev->wl->hardirq_lock);

    return ret;
}
```

This function returns `IRQ_NONE`
or `IRQ_WAKE_THREAD`

- A top-half returns `IRQ_NONE` if this driver is not handling interrupt, `IRQ_HANDLED` if it finished handling interrupt, or `IRQ_WAKE_THREAD` to continue processing within a bottom-half

Example Bottom-Half

```
static irqreturn_t b43_interrupt_thread_handler(int irq, void *dev_id)
{
    struct b43_wldev *dev = dev_id;

    mutex_lock(&dev->wl->mutex);
    b43_do_interrupt_thread(dev);
    mmiowb();
    mutex_unlock(&dev->wl->mutex);

    return IRQ_HANDLED;
}
```

- Parameters to bottom-half same as parameters to top-half
- Bottom-half returns `IRQ_HANDLED` when it has finished its work
- **Because bottom-half is running in process context, it may use mutexes and call other functions that can sleep**

Spinlocks and Interrupt Handlers

- Spinlocks are safe to use in top-halves as that they do not sleep
- **May be dangerous** for a kthread to hold a spinlock that a top-half also needs
 - Deadlock will occur if top-half is blocked on a single-processor system
- Solution is for kthread to ensure that top-half will not deadlock, even if spinlock already held
- Use special form `spin_lock_irqsave()`
 - **Must use this form whenever a spinlock is used in both interrupt and process context**

Spinlocks and Interrupt Handlers

```
void spin_lock_irqsave(spinlock_t *lock, unsigned long flags);  
void spin_unlock_irqrestore(spinlock_t *lock, unsigned long flags);
```

- `spin_lock_irqsave()` acquires spinlock, saves the current state in `flags`, and disables interrupts on that CPU
- `spin_unlock_irqrestore()` releases spinlock, then reenables interrupts

```
static inline unsigned char rtc_is Updating(void)  
{  
    unsigned long flags;  
    unsigned char uip;  
  
    spin_lock_irqsave(&rtc_lock, flags);  
    uip = (CMOS_READ(RTC_FREQ_SELECT) & RTC_UIP);  
    spin_unlock_irqrestore(&rtc_lock, flags);  
    return uip;  
}
```

From drivers/char/rtc.c

Threaded Interrupt Handling Summary

