MPS

Lesson 06

Interrupts

Todays Lesson

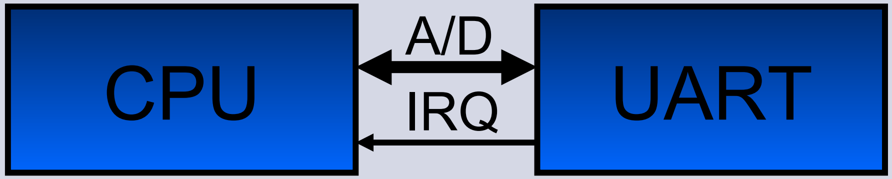
AARHUS SCHOOL OF ENGINEERING

8:00	Last Lesson
	Presentation of last exercise
	Hardware side of Interrupts
8:35	Break (10 min)
	Sleeping in Linux
	Interrupts in Linux
9:10	Break (10 min)
	Top & Bottom Halves
9:35	Exercise: GPIO Interrupt Linux Device Driver
12:00	Ahh! Lunch!

Interrupt Types

- Software Interrupts
  - Event driven code
    - Software driven interrupts Ex. GUI
  - Exceptions
    - Special SW Interrupts generated by the CPU itself on critical errors. Ex. page fault (MMU)
- Hardware Interrupts
  - Internal
    - Interrupts generated by internal devices to notify the core of an event. Ex Internal UART notifies that RX buffer is full. Could also be an exception in case of internal error.
  - External
    - Interrupts from external devices given to dedicated IRQ pins on the CPU

Level-Triggered Interrupts



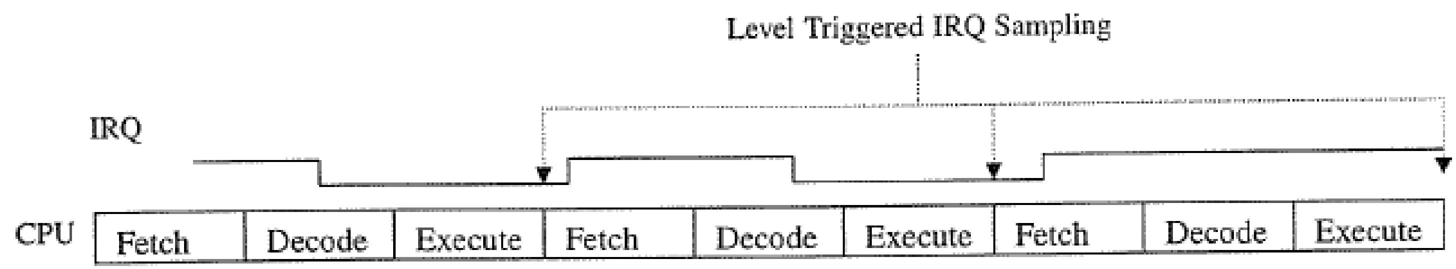
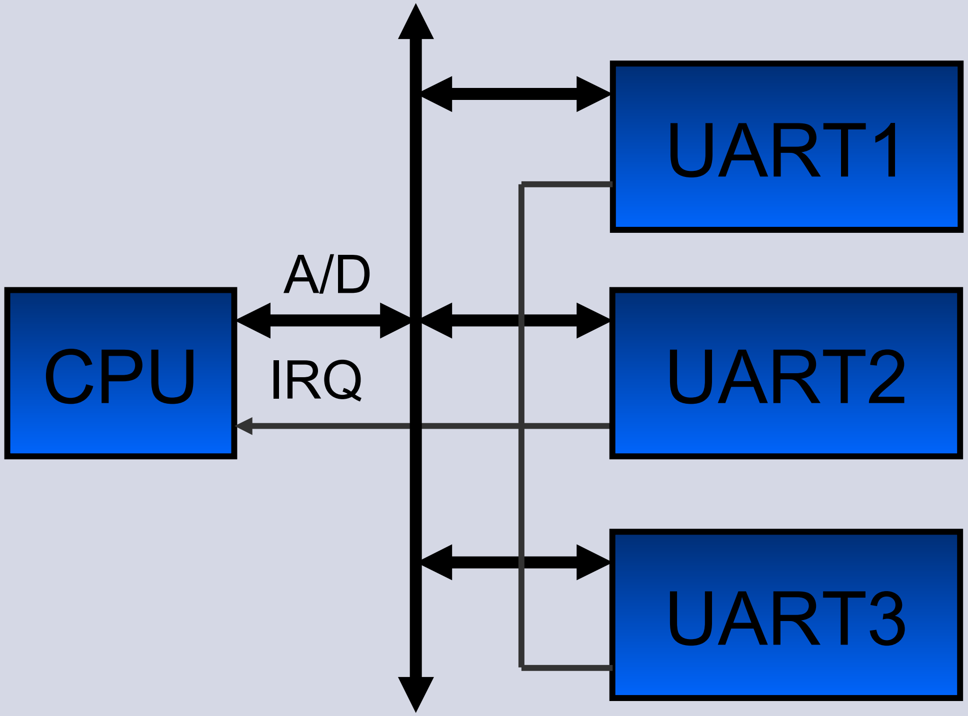


Figure 8-5a: Level-triggered interrupts [8-3]

- The Interrupt is active as long as IRQ is active
- IRQ is sampled at ex. Command Fetch time
- Interrupt Service Routing (ISR) must acknowledge IRQ source, to let it deactivate the IRQ line
- Good for shared IRQ lines

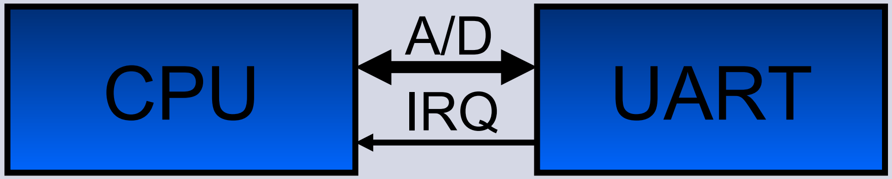
Shared Interrupts

```
void ISR (){
  ...
  [who did it?]
  [do something]
  [ack IRQ source]
 IRET;
}
```



- Interrupt lines are sparse and should be preserved
- ISR will continue to be called until all sources has been serviced
- Only possible with Level-Triggered Interrupts
- Clients typically have an IRQ register that can be read by the CPU

Edge-Triggered Interrupts



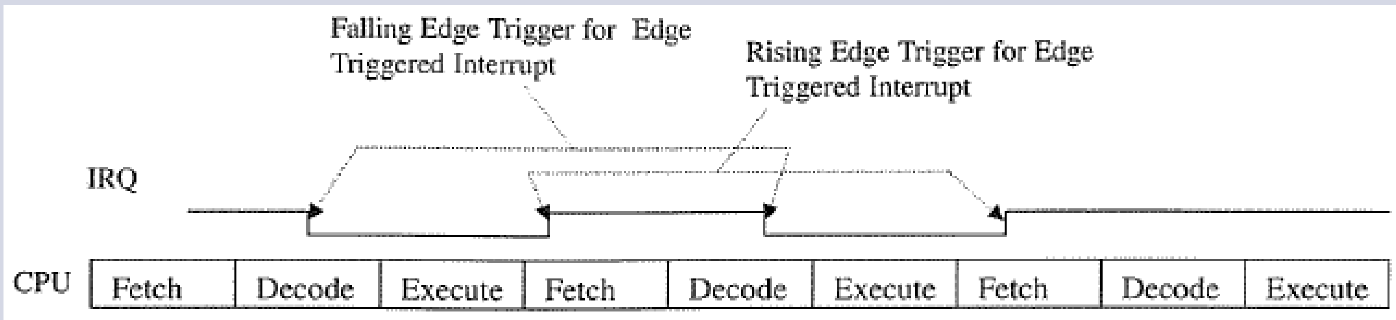
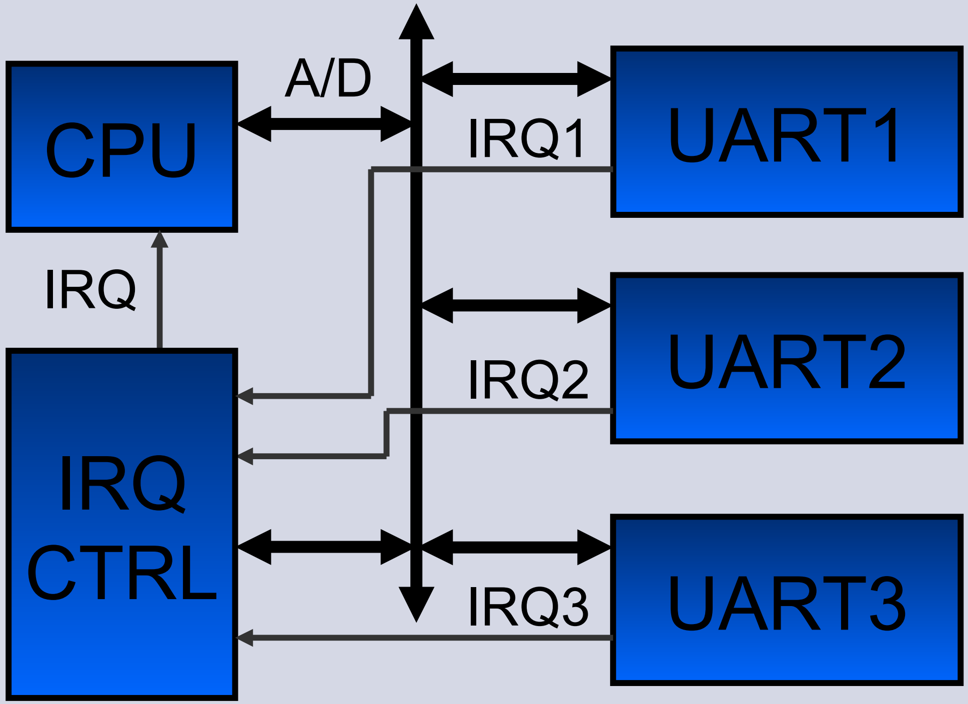


Figure 8-5b: Edge-triggered interrupts [8-3]

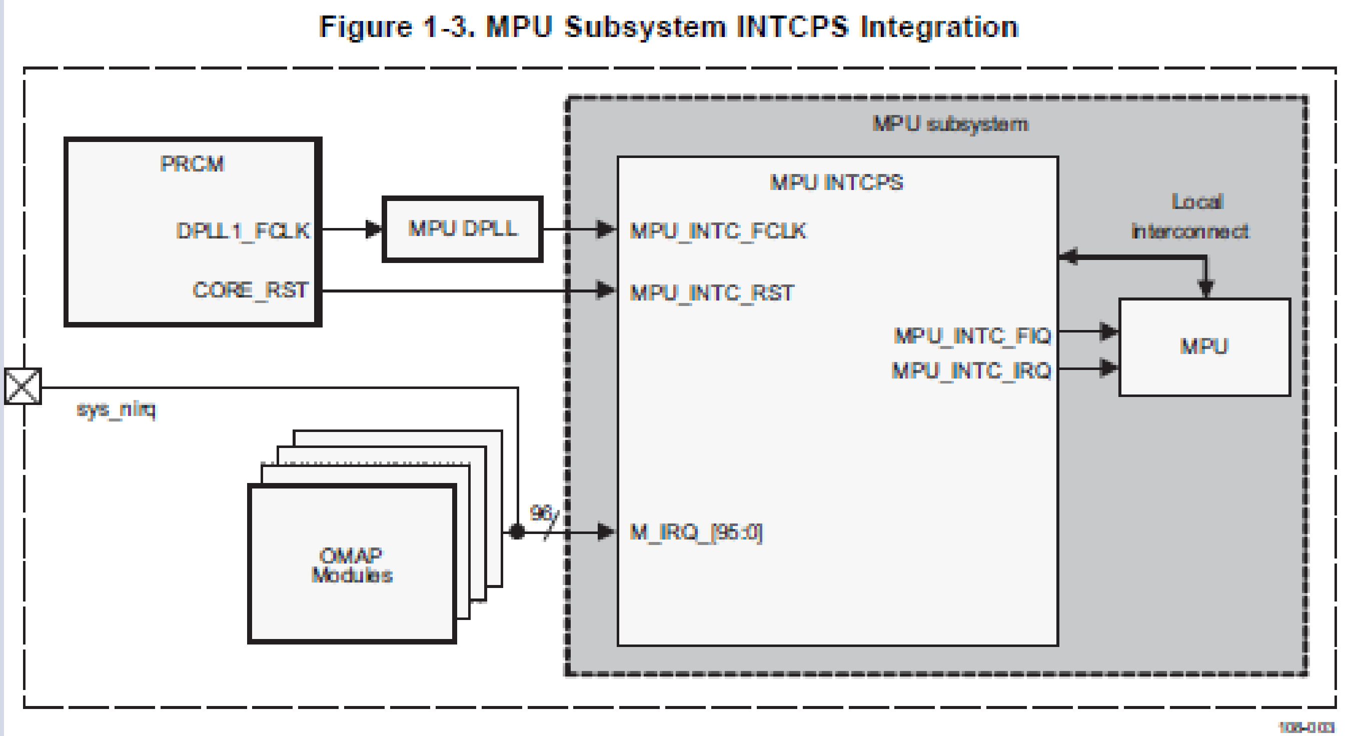
- The Interrupt is activated by an IRQ signal transition
- ISR is called only one time, hence it does not block
- If shared, the CPU can choose which IRQ to service when.
- Depending on architecture, IRQs may not be seen while in ISR
- Good for short (or very long) IRQ signals that doesn't need ACK

Shared Interrupts (edge)



- An interrupt controller will latch any incoming IRQ
- The controller will activate a level-triggered IRQ to the CPU
- The CPU will read the interrupt controllers IRQ source register and service the proper IRQ source
- The interrupt controller can be build into the processor

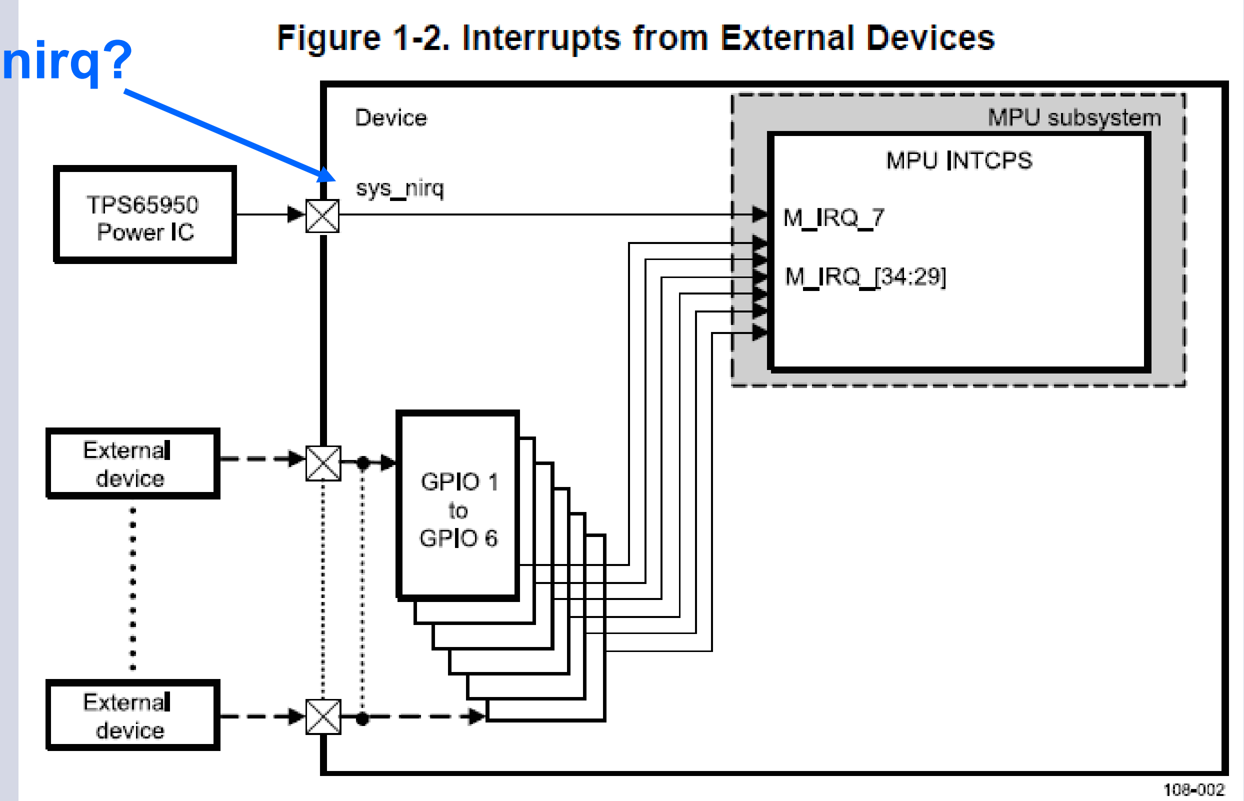
OMAP Interrupts (1)



- The main processor has only two Interrupt inputs
  - FIQ: Fast Interrupt
  - IRQ: Slow Interrupt
- The INTCPS interrupt controller prioritizes the 96 inputs

OMAP Interrupts (2)

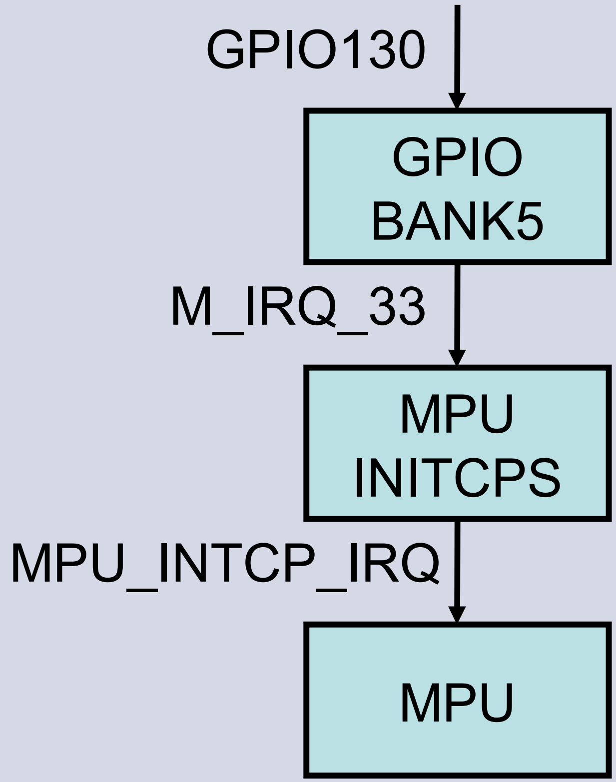
What's nirq?



- The GPIO sub-system occupies IRQs [34:29]
- GPIO pins can be configured to generate interrupts on input change

GPIO Interrupt Flow

```
void MPU_ISR (){
  ...
  [GetInitCpsSource]
  [GetGpioIrqSource]
  [DoSomething]
  [AckGpioIrqSource]
  [AckInitCpsSource]
  IRET;
}
```



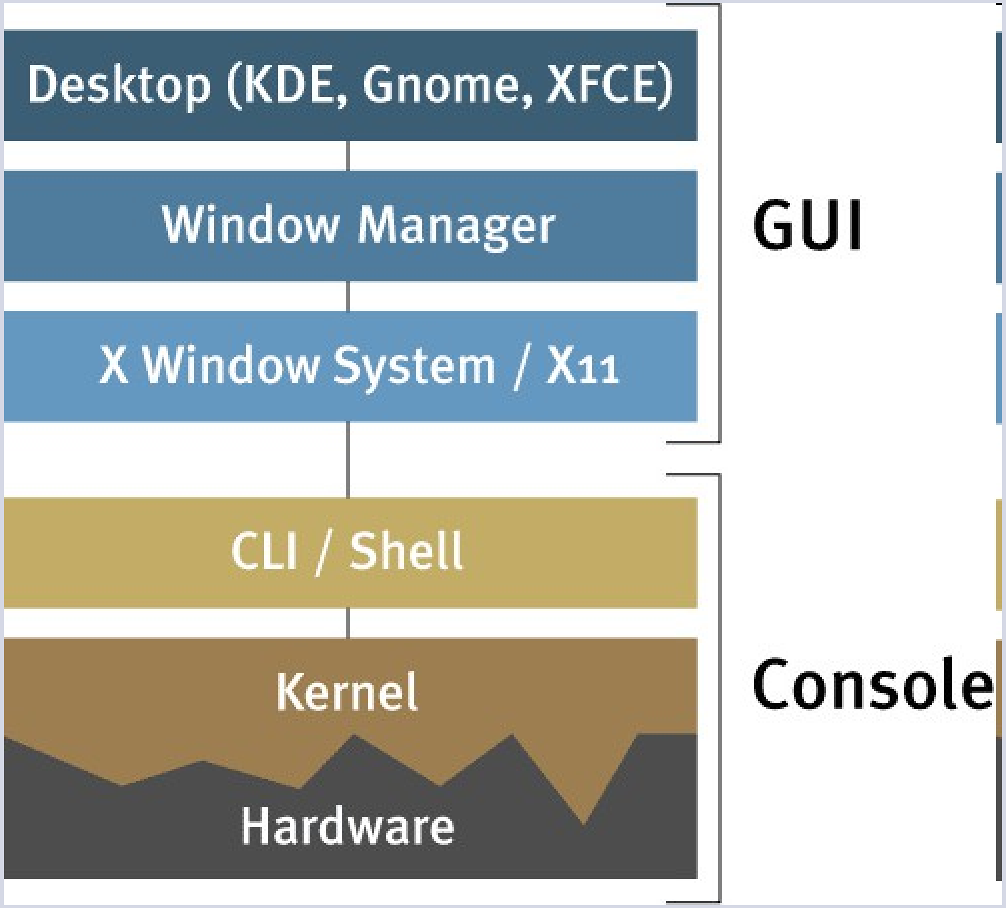
- ISR will have to go thru the nested list of interrupt sources to find the actual interrupt source

Hardware Summary

- Interrupt Types?
- Level Triggered Interrupts?
- Edge Triggered Interrupts
- Shared Interrupts?
- GPIO Interrupts?

Interrupts in Linux

Interrupts in Linux



- We would like an interrupt to cause an event in our application
- BUT we don’t want the hardware to know of our application, how?

Blocking I/O

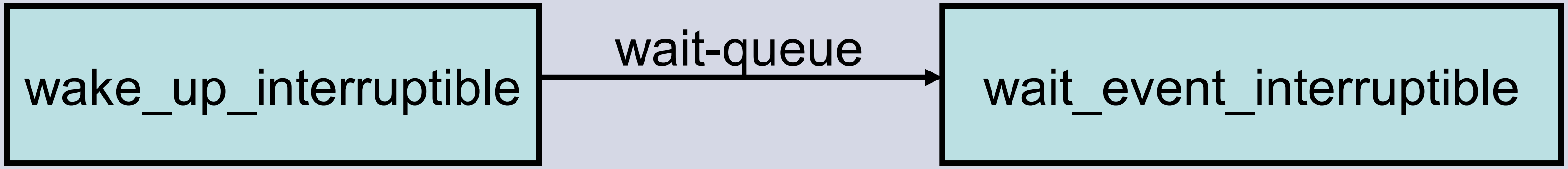
- The Linux scheme for separating driver and application is to use blocking I/O access

- The application requests data from a device, and is put to sleep until that data is available
  - Or you could say, it subscribes to an event
- In user space, this is done using blocking file access
- In kernel space, this is implemented in a driver
  - The driver's read/write methods sleeps until an interrupt subscribed for occurs
- Interrupts are ONLY available in kernel space!

Sleeping in the Kernel (1)

- The Linux kernel has a run-queue with active processes
- Each process is devised a time-slice, based on its priority
- Active process in the run-queue is marked as TASK_RUNNING
- A sleeping process is taken out of the run-queue and marked as TASK_(UN)INTERRUPTIBLE
- When woken up, the process is put into the run-queue again

Sleeping in the Kernel (2)



- To implement sleeping/awakening we must use:
  - A wait queue – to pass events from wake_up to wait
  - A wake_up method, to be called when an event occur (ex in an ISR)
  - A wait method, that sleeps until woken up by an event passed thru the wait queue. This could ex be inside the read method

Sleeping in the Kernel (3)

```
static DECLARE_WAIT_QUEUE_HEAD(wq);
static int flag = 0;
ssize_t sleepy_read (struct file *filp, char __user *buf,
                     size_t count, loff_t *pos){
  printk(KERN_DEBUG "process %i (%s) going to sleep\n",
  current->pid, current->comm);
  wait_event_interruptible(wq, flag != 0);
  flag = 0;
  printk(KERN_DEBUG "awoken %i (%s)\n", current->pid,
                                        current->comm);
  return 0;} /* EOF */

ssize_t sleepy_write (struct file *filp, const char __user *buf,
                      size_t count, loff_t *pos){
  printk(KERN_DEBUG "process %i (%s) awakening the readers\n",
          current->pid, current->comm);
  flag = 1;
  wake_up_interruptible(&wq);
  return count;} /* succeed, to avoid retrial */
```

Interrupt Handling

- To use interrupts in a driver, you must request an IRQ line
- A module is expected to request an IRQ line before using it and to release it afterwards
- The IRQ line should be requested in the "**open**" method rather than in the "init" method
  - If the device is not opened, we won't listen to the IRQ anyway
  - If put in the "init" method, the line will be occupied as soon as the module is inserted in the kernel
- The IRQ line should be released in the "release" method.
- A driver with several minor numbers can share a line
  - Special care must be taken with the request/release methods

request_irq / free_irq

ISR

IRQ line

/proc/interrupts name

```
int request_irq(unsigned int irq,
      irqreturn_t (*handler)(int, void *,
      struct pt_regs *),
      unsigned long flags, const char *dev_name,
      void *dev_id);

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

void ptr used in ISR to tell which dev is interrupting

IRQ line

device ptr

/proc

```
root@DevKit8000:~# cat /proc/interrupts
             CPU0
 11:            0        INTC  prcm
 12:          577        INTC  DMA
 56:          385        INTC  i2c_omap
 74:          232        INTC  serial
 77:            0        INTC  ehci_hcd:usb1
 83:        13231        INTC  mmc0
 93:            0        INTC  musb_hdrc
 95:         7016        INTC  gp timer
186:            0        GPIO  user
187:            0        GPIO  ads7846
369:            0     twl4030  twl4030_keypad
378:            0     twl4030  twl4030_usb
Err:            0
```

IRQ Line | **# Events** | **IRQ Controller** | ***dev_name***

/proc

```
root@DevKit8000:~# cat /proc/stat
cpu  142 0 530 73528 1269 5 0 0 0
cpu0 142 0 530 73528 1269 5 0 0 0
intr 120154 0 0 0 0 0 0 0 0 0 0 0 0 577 0 0 0 0 0 ...
0 0 0 385 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 232 0 ...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 ...
ctxt 56055
btime 1233051572
processes 1343
procs_running 2
procs_blocked 0
```

 # Total events

IRQ0

IRQ12

Detecting IRQ Numbers

- At hardware level, devices are attached to IRQ lines on an interrupt controller
- In the driver, we need to retrieve the IRQ line number, to be able to install a handler.
- This can be done using several methods:
  - Interrupt controller may provide methods for retrieving the number
  - The number may be static and can be found in Interupt handler code (linux-2.6.x/arch/arm/omap2/irq.c a.o.)
  - Interrupt probing
    - Check for available IRQs
    - Generate an interrupt event
    - Check for a change in the available IRQs
    - Manually or using kernel methods
  - ***`irq = gpio_to_irq(gpio_no);`***

Interrupt Handler

- The interrupt handler is called upon an interrupt event
- The handler is plain C-code, but:
  - The handler must be FAST
  - It must not do anything that can sleep
  - It must not allocate memory
- The typical ISR must:
  - Acknowledge the "irq_pending" bit on the device
  - Wake-up the device
  - Acquire data from the device
  - Wake-up a sleeping function in the driver (ex read)
  - Exit with IRQ_HANDLED or IRQ_NONE

handle_my_interrupt

ISR

IRQ line that caused event

```
irqreturn_t handle_my_irq(int irq,
                void *dev_id, struct pt_regs *);
```

Must return: IRQ_HANDLED / IRQ_NONE

pointer to data structure entered in request_irq

cpu context at irq time Depricated in later linux versions DON'T USE

ISR Example

```
irqreturn_t handle_my_irq(int irq,
                void *dev_id, struct pt_regs *){
  [ack irq pending]
  [readbuf = read value from device]
  wake_up_interruptible(wait_queue);
  return IRQ_HANDLED;
}
```

```
ssize_t my_read(...){

  wait_event_interruptible(wait_queue, flag == 1);
  copy_to_user(buf, readbuf, sizeof(readbuf));
  return sizeof(readbuf);
}
```

ISR Example (LDD p.271)

```
ssize_t short_i_read (...)
{
int count0;
DEFINE_WAIT(wait);
while (short_head == short_tail) {
      prepare_to_wait(&short_queue, &wait,
                              TASK_INTERRUPTIBLE);
      if (short_head = = short_tail)
            schedule( );
      finish_wait(&short_queue, &wait);
      if (signal_pending (current))
            return -ERESTARTSYS;
      }
```

Set task to TASK_INTERRUPTIBLE

ISR Active

Set task back to TASK_RUNNING

Take the task out of the running queue

Top- & Bottom Halves

AARHUS SCHOOL OF ENGINEERING

- Top-Half:
  - ISR created with request_irq()
  - Ack device irq_pending
  - Read device data to buffer
  - Schedule tasklet/workqueue
  - IRQ can be disabled
- Bottom-Half
  - Scheduled by the kernel
  - Handles all heavier processing
  - IRQs enabled during processing
  - Implemented as tasklet or workqueue

Top Half
Interrupt Handler
Fast

Bottom Half
Tasklet / Workqueue
Slow
More Processing

Tasklet

- Tasklet operation is atomic. Not tasklet ever runs in parallel with itself!
- Runs in interrupt context. Cannot Sleep, but is fast

Workqueue

```
static struct work_struct short_wq;
/* this line is in short_init( ) */
INIT_WORK(&short_wq, (void (*)(void *)) do_wq, NULL);

irqreturn_t wq_interrupt(int irq, void *dev_id, struct
pt_regs *regs){
 [Do ack, get data from device]
 schedule_work(&short_wq);
 return IRQ_HANDLED; }

void do_wq (unsigned long unused)
{ ... }
```

- Works very much like a tasklet, but
  - Works in process context (can sleep, allocate memory etc)
  - Slower

Shared Interrupts

- Call irq_request(irq_line, isr, SA_INTERRUPT | SA_SHIRQ, “irq_name”, irq_dev_id)
- To minimize the number of IRQ lines used, they may be shared
- The ISR should check the IRQ source, and return IRQ_NONE if it wasn’t its device who was the source

```
irqreturn_t short_sh_interrupt(int irq, void *dev_id,
struct pt_regs *regs) {

  /* If it wasn't my device, return immediately */
  value = inb(short_base); // Read port
  if (!(value & 0x80))      // If not the source
    return IRQ_NONE;
  /* clear the interrupting bit */
  outb(value & 0x7F, short_base); // Else clr irq_pend
```

Linux Summary

- Linux Interrupt Model?
- Blocking access?
- Sleeping?
- Interrupt handling in Linux?
- Detecting IRQ numbers?
- ISR?
- Top- / Bottom Half?
- Shared Interrupts?