Controlling a program

```
// child
close(STDIN_FILENO)
dup2(indes[0], STDIN_FILENO)
close(indes[0])
clsse(indes[1])             // outdes

execlp("./circle", "./circle", (char *)NULL)

// mother
fcntl(indes[1], F_SETFL, fcntl(indes[1], F_GETFL) | O_NONBLOCK) // outdes
do{
    nread = read(outdes[0], buffer, MAX_BUFFER);
while(nread <= 0);
write(STDOUT_FILENO, buffer, nread);
```

---

Initialize Device

```
if(!use_mem){
    if(! request_region(short_base, NR, "short"))
        ERROR
}else{
    if(! request_mem_region(...))
        ERROR
    short_base = (unsigned long) ioremap(short_base, NR)
}

result = register_chrdev(major, "short", &short_fops);
if(result < 0)
    ERROR - release_?_region
.........................................
if(scull_major)
    dev = MKDEV(scull_ajor, scull_minor)
    result = register_chrdev_region(dev, NR, "scull")
else
    result = alloc_chrdev_region(&dev, scull_minor, NR, "scull")
    scull_major = MAJOR(dev)

scull_devices = kmalloc(NR*sizeof(struct scull_dev), GFP_KERNEL)
memset(scull_devices, 0, size)
for(i -> NR)
    initialize
```

---

Self Probe Example

```
** mask = probe_irq_on()
short_irq = 0
outb_p(0x10, short_base+2)
outb_p(0x00, short_base)
outb_p(0xFF, short_base)
outb_p(0x00, short_base+2)
udelay(5)
** short_irq = probe_irq_off(mask)
```

---

Pipe içinde yapılanlar - bekletmeler

```
while(1){
    inpipe = pipe2[0]             // outpipe
    FD_ZERO(&fds_rd)
    FD_SET(inpipe, &fds_rd)       // outpipe

    select(n+1, &fds_rd, &fds_wr, NULL, NULL)

    if(FD_ISSET(inpipe, &fds_rd))
         i = read(inpipe, buf, 1024)
}
```

---

IRQ ne zaman register edilmeli -- First Opening > Registering Device

The interrupt handler can be installed either at driver initialization or when the device is first opened. Although installing the interrupt handler from within the module's initialization function might sound like a good idea, it often isn't, especially if your device does not share interrupts. Because the number of interrupt lines is limited, you don't want to waste them. You can easily end up with more devices in your computer than there are interrupts. If a module requests an IRQ at initialization, it prevents any other driver from using the interrupt, even if the device holding it is

never used. Requesting the interrupt at device open, on the other hand, allows some sharing of resources.

---

IRQ autodetect - initialize, probe handling, open, close

One of the most challenging problems for a driver at initialization time can be how to determine which IRQ line is going to be used by the device. Autodetect

```
if (short_irq < 0) /* not yet specified: force the default on */
  switch(short_base) {
    case 0x378: short_irq = 7; break;
    case 0x278: short_irq = 2; break;
    case 0x3bc: short_irq = 5; break;
  }
// interrupt handler
irqreturn_t short_interrupt(int irq, void *dev_id, struct pt_regs *regs){
     struct timeval tv
     int written
     do_gettimeofday(&tv)
     written = sprintf((char*)short_head, "%08u.%06u\n", (int)(tv.tv_sec % 100000000),
(int)(tv.tv_usec))
     short_incr_bp(&short_head, written)
     wake_up_interruptable(&short_queue)
     return IRQ_HANDLED
}

int short_open(struct inode *inode, struct file *filp)
    extern struct file_operations short_i_fops
    if(iminor(inode) & 0x809
         filp->f_op = &short_i_fops
    return 0

irqreturn_t short_probing(int irq, void *dev_id, struct pt_regs *regs){
     if(short_irq == 0)
          short_irq = irq;
     if(short_irq != irq)
          short_irq = -irq;
     return IRQ_HANDLED
}
```

Autodeteched irq İnterruptor yaz

```
if(short_irq >= 0)
    result = request_irq(short_irq, short_interrupt,
             SA_INTERRUPT, "short", NULL)
    outb(0x10, short_base + 2)
```

One of the main problems with interrupt handling is how to perform lengthy tasks within a handler. Often a substantial amount of workmust be done in response to a device interrupt, but interrupt handlers need to finish up quickly and not keep interrupts blocked for long.

Linux (along with many other systems) resolves this problem by splitting the interrupt handler into two halves. The so-called top half is the routine that actually responds to the interrupt—the one you register with request_irq. The bottom half is a routine that is scheduled by the top half to be executed later, at a safer time.

---

I/O System layerlerini açıkla

user processes
device-independent software
device drivers
interrupt handlers
hardware

---

I/O Request lifecycleyi açıkla

request I/O (user process) -> system call
can already satisfy request? -> no
send request to device driver -> process request
device controller -> interrupt -> I/O completed
interrupt handler -> receives interrupt
store data in buffer
determine which I/O completed -> transfer data
I/O completed

---

Advantages = Device Drivers as Modules
provides a standart interface
hides details of the hardware devices communicaiton protocols