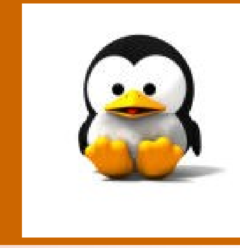
Linux Device Driver

(Hardware Management)

Amir Hossein Payberah

payberah@yahoo.com

Contents



- I/O Ports and I/O Memory
- Using I/O Ports
- Using I/O Memory
- Optimization

I/O Ports and I/O Memory

- Every peripheral device is controlled by writing and reading its registers.
- Most of the time a device has several registers.
- They are accessed at consecutive addresses, either in the memory address space or in the I/O address space.

I/O Ports and I/O Memory

- At the hardware level, there is no conceptual difference between memory regions and I/O regions.
  - Both of them are accessed by asserting electrical signals on the address bus and control bus and by reading from or writing to the data bus.

I/O Ports and I/O Memory

- Some CPU manufacturers implement a single address space in their chips.
- Some others decided that peripheral devices are different from memory.
    - Therefore deserve a separate address space.
    - Some processors have separate read and write electrical lines for I/O ports, and special CPU instructions to access ports.

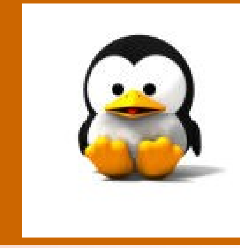
I/O Ports and I/O Memory

- Because peripheral devices are built to fit a peripheral bus, Linux implements the concept of I/O ports on all computer platforms it runs on, even on platforms where the CPU implements a single address space.

I/O Ports and I/O Memory

- Even if the peripheral bus has a separate address space for I/O ports, not all devices map their registers to I/O ports.
- Use of I/O ports is common for ISA peripheral boards.
- Most PCI devices map registers into a memory address region.

Contents



- I/O Ports and I/O Memory
- Using I/O Ports
- Using I/O Memory
- Optimization

I/O Ports

- I/O ports are the means by which drivers communicate with many devices out there.
- Information about registered resources is available in /proc/ioports.

Allocating I/O ports

- int check_region(unsigned long start, unsigned long len);
- struct resource *request_region(unsigned long start, unsigned long len, char *name);
- void release_region(unsigned long start, unsigned long len);
- They are defined in <linux/ioport.h>

Sample

```
static int skull_detect(unsigned int port, unsigned int
    range)
{
    int err;
    if ((err = check_region(port,range)) < 0)
        return err; /* busy */
    request_region(port,range,"skull"); /* "Can’t fail" */
    return 0;
}

static void skull_release(unsigned int port, unsigned int
    range)
{
    release_region(port,range);
}
```

Read and write I/O ports

- unsigned inb(unsigned port);
- void outb(unsigned char byte, unsigned port);
- unsigned inw(unsigned port);
- void outw(unsigned short word, unsigned port);
- unsigned inl(unsigned port);
- void outl(unsigned longword, unsigned port);
- They are defined in <asm/io.h>

User space I/O ports

- They can be used from user space.
- The GNU C library defines them in <sys/io.h>.
- User space conditions:
  - The program must be compiled with the *-O* option.
  - The ioperm or iopl system calls must be used to get permission to perform I/O operations on ports.

String operations

- void insb(unsigned port, void *addr, unsigned long count);
- void outsb(unsigned port, void *addr, unsigned long count);
- void insw(unsigned port, void *addr, unsigned long count);
- void outsw(unsigned port, void *addr, unsigned long count);
- void insl(unsigned port, void *addr, unsigned long count);
- void outsl(unsigned port, void *addr, unsigned long count);

Pausing I/O

- Some platforms can have problems when the processor tries to transfer data too quickly to or from the bus.
- The pausing functions are exactly like those listed previously, but their names end in _p.
  - They are called inb_p, outb_p, and so on.

Contents

- I/O Ports and I/O Memory
- Using I/O Ports
- Using I/O Memory
- Optimization

I/O memory

- Despite the popularity of I/O ports in the x86 world, the main mechanism used to communicate with devices is through memory-mapped registers and device memory.
- Both are called I/O memory because the difference between registers and memory is transparent to software.

I/O memory

- I/O memory is simply a region of RAM-like locations that the device makes available to the processor over the bus.
- Information about I/O memory registered resources is available in /proc/iomem.

Advantage of I/O memory

- It doesn’t require use of special-purpose processor instructions.
- CPU cores access memory much more efficiently, and the compiler has much more freedom in register allocation and addressing-mode selection when accessing memory.

Accessing I/O memory

- According to the computer platform and bus being used, I/O memory may or may not be accessed through page tables.
- When access passes though page tables, the kernel must first arrange for the physical address to be visible from your driver.
  - ioremap
- If no page tables are needed, then I/O memory locations look pretty much like I/O ports, and you can just read and write to them using proper wrapper functions.

Allocating I/O memory

- int check_mem_region(unsigned long start, unsigned long len);
- void request_mem_region(unsigned long start, unsigned long len, char *name);
- void release_mem_region(unsigned long start, unsigned long len);

Allocating I/O memory

- The start argument to pass to the functions is the physical address of the memory region, before any remapping takes place.

Sample

```
if (check_mem_region(mem_addr, mem_size))
 {
   printk("drivername: memory already in use\n");
   return -EBUSY;
}

request_mem_region(mem_addr, mem_size,
   "drivername");

release_mem_region(mem_addr, mem_size);
```

Read and write I/O memory

- unsigned readb(address);
- void writeb(unsigned value, address);
- unsigned readw(address);
- void writew(unsigned value, address);
- unsigned readl(address);
- void writel(unsigned value, address);

Software mapped I/O memory

- Devices live at well-known physical addresses, but the CPU has no predefined virtual address to access them.
- The well-known physical address can be either hardwired in the device (ISA) or assigned by system firmware at boot time (PCI).

Software mapped I/O memory

- For software to access I/O memory, there must be a way to assign a virtual address to the device.
  - This is the role of the ioremap function.

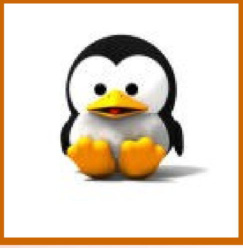
Ioremap

- void *ioremap(unsigned long phys_addr, unsigned long size);
- void iounmap(void * addr);
- They are defined in <asm/io.h>.

Sample

```
check_mem_region(reset, 84);
request_mem_region(reset, 84,
  "mydev");
virtual_reset = ioremap(reset, 84);
writeb(0x40, virtual_reset + 83);
iounmap(virtual_reset);
release_mem_region(reset, 84);
```

Contents



- I/O Ports and I/O Memory
- Using I/O Ports
- Using I/O Memory
- Optimization

Optimization

- Despite the strong similarity between hardware registers and memory, a programmer must be careful to avoid being tricked by CPU (or compiler) optimizations that can modify the expected I/O behavior.

I/O Ports and I/O Memory optimization

- I/O operations have side effects.
- Memory operations have none.
- Because memory access speed is so critical to CPU performance, the values are cached and read/write instructions are reordered.

I/O Ports and I/O Memory optimization

- These optimizations are transparent and benign when applied to memory
- But they can be fatal to correct I/O operations.

Driver optimization view

- A driver must therefore ensure that no caching is performed and no read or write reordering takes place when accessing registers.

Optimization solution

- The solution to compiler optimization and hardware reordering is to place a memory barrier between operations that must be visible to the hardware in a particular order.

Memory barrier

- void barrier(void);
- void rmb(void);
- void wmb(void);
- void mb(void);

Barrier

- Compiled code will store to memory all values that are currently modified and resident in CPU registers, and will reread them later when they are needed.
- It is defined in <linux/kernel.h>

rmb, wmb, mb

- The rmb (read memory barrier) guarantees that any reads appearing before the barrier are completed prior to the execution of any subsequent read.
- The wmb (write memory barrier) guarantees ordering in write operations,
- The mb (memory barrier) instruction guarantees both.
- They are defined in <asm/system.h>

Sample

```
writel(dev->registers.addr,
   io_destination_address);
writel(dev->registers.size, io_size);
writel(dev->registers.operation, DEV_READ);
wmb();
writel(dev->registers.control, DEV_GO);
```

Question?