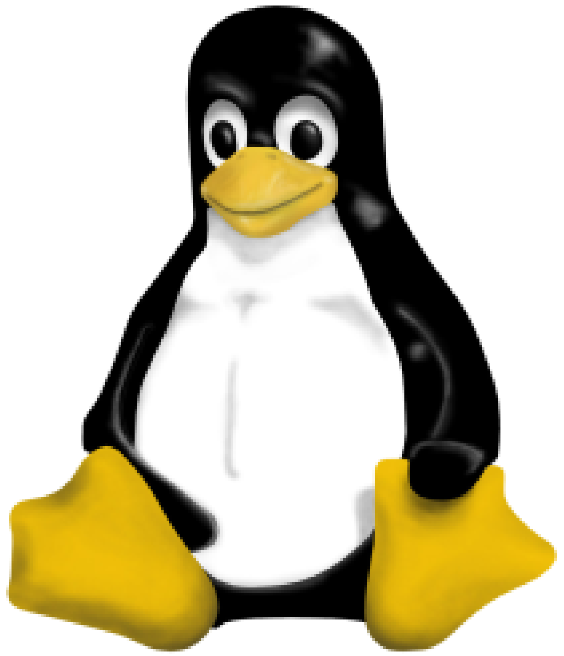
Linux Kernel Hacking Free Course, 3rd edition

E. Betti
University of Rome “Tor Vergata”

Drivers for character devices



February 22, 2006

Summary of the previous lecture

In the previous lecture we have seen:

- what is a driver
- which are the driver's tasks
- how to implement some of this tasks on a PCI device (Galil DMC 1800 controller):
  1. how to register a driver on the software layer of the hosting bus
  2. how to probe for compatible devices
  3. how to list the resources used by each device found
  9. how to release the resources obtained by a driver
  10. how to deregister a driver from the bus software layer

Outline

In this lecture we first introduce the functions used to comunicate with an I/O device, we'll then continue working on the example introduced in the previous lecture and we'll show how to:

4. initialize the device (in our example, the Galil DMC 1800 controller)
5. register a driver for this char device, thus assigning to it a major and minor number. To make this we must:
    6. implement the file operations of the device file
    7. manage the operations of the device
8. deregister the device file

Comunicating with an I/O device

Tipically, a device is controlled by writing and reading its registers. Those registers are accessed either in the memory address space (I/O memory) or in the I/O address space (I/O ports).

In the previous lecture we have seen how to obtain the starting address of the I/O memory and the address of the first I/O port; we have also seen how to tell the kernel that the driver will make use of those resources.

Now, we'll discuss the basic functions needed to comunicate with a device through I/O memory or I/O ports.

Comunicating with a device: I/O port

An I/O port can have different sizes, thus different C functions must be used to access different port sizes:

- 8 bit wide port:
  `unsigned inb(unsigned port)` to read,
  `void outb(unsigned char byte, unsigned port)` to write.

- 16 bit wide port:
  `unsigned inw(unsigned port)` to read,
  `void outw(unsigned short word, unsigned port)` to write.

- 32 bit wide port:
  `unsigned inl(unsigned port)` to read,
  `void outl(unsigned longword, unsigned port)` to write.

Comunicating with a device: I/O memory

Although `ioremap()` yields virtual address, direct use of pointers to I/O memory is discouraged. The safe way to access I/O memory is through the following functions:

```
unsigned int ioread8(void *addr)
unsigned int ioread16(void *addr)
unsigned int ioread32(void *addr)
void iowrite8(u8 value, void *addr)
void iowrite16(u16 value, void *addr)
void iowrite32(u32 value, void *addr)
```

To operate on a block of I/O memory:

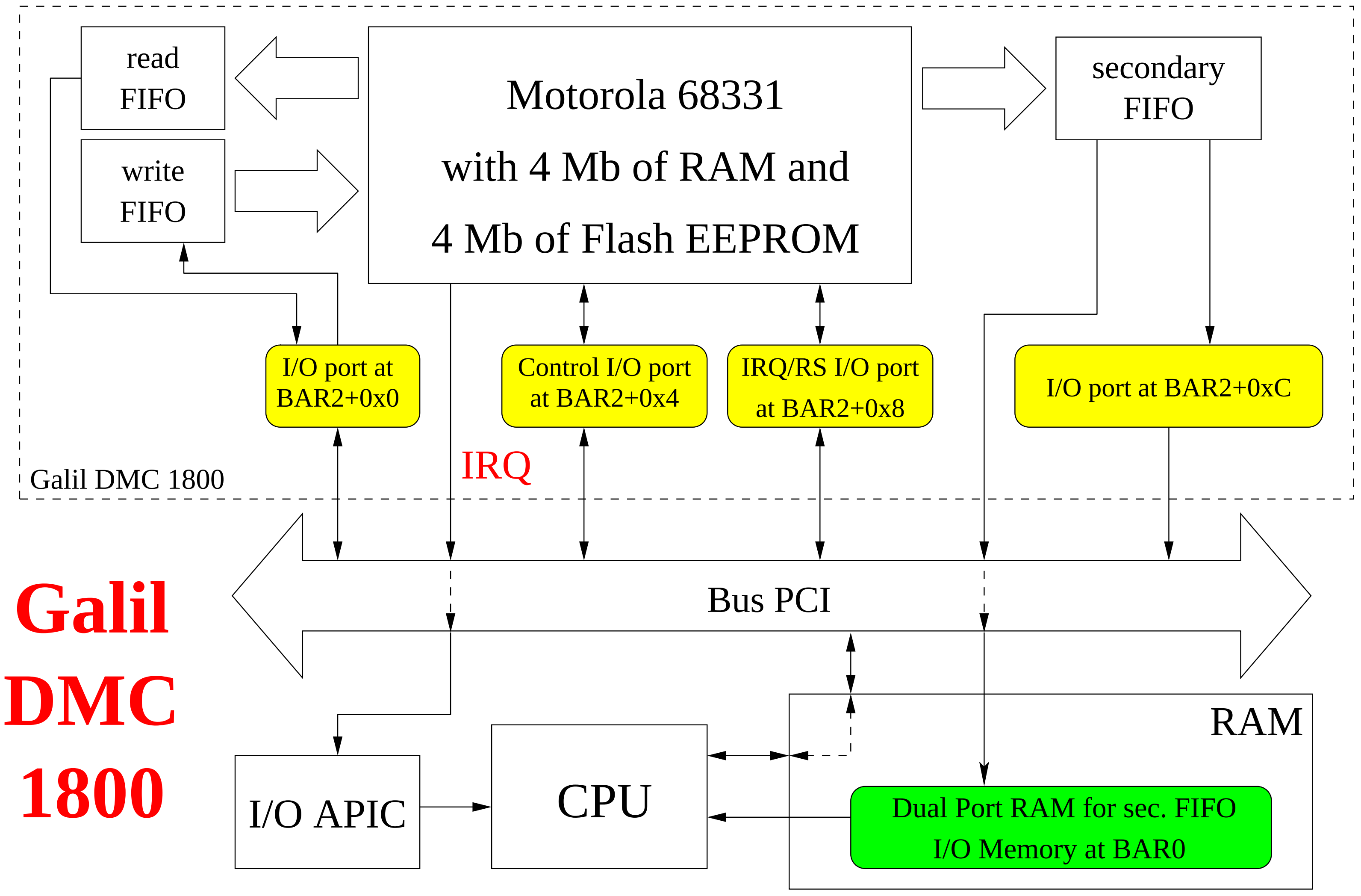
```
memset_io(void *addr, u8 value, unsigned int count)
memcpy_fromio(void *dest, void *src, unsigned int count)
memcpy_toio(void *dest, void *src, unsigned int count)
```

Comunicating with the Galil DMC 1800 device

To develop a driver for a device, we need to read the device's datasheet to know how to comunicate with that device.

So, to continue with the example of the previous lecture, we first need to explain somethings about how to comunicate with our test device, the Galil DMC 1800. In the next slide, I'll introduce a schematic rapresentation of that device. You should refer to that slide to get a better understanding of the code for some basic low level operations of the Galil DMC 1800.

You can find the complete device's datasheet at www.galilmc.com.


read FIFO
write FIFO
Motorola 68331 with 4 Mb of RAM and 4 Mb of Flash EEPROM
secondary FIFO
I/O port at BAR2+0x0
Control I/O port at BAR2+0x4
IRQ/RS I/O port at BAR2+0x8
I/O port at BAR2+0xC
Galil DMC 1800
IRQ
Bus PCI
RAM
I/O APIC
CPU
Dual Port RAM for sec. FIFO I/O Memory at BAR0

Galil DMC 1800

Step 4. Initializing the device

This step is really device-dependent; also in this case, the operations that must be performed can be found in the device’s datasheet.

To simplify the example we only perform two simple actions:

- reset the read and write buffers through the control I/O port;
- perform a global reset of the controller by putting a specific command in the write FIFO.

Step 5. Registering the driver for a char device (1)

If a driver wants register itself for one or more character devices, first of all, it needs to obtain a range of device numbers.

If you know exactly which device numbers you want, you can use:

```
int register_chrdev_region(dev_t first,
           unsigned int count, char *name)
```

Tipically, however, you don't know which major device number you device can use, so you can let the kernel choose for you by calling:

```
int alloc_chrdev_region(dev_t *dev,
  unsigned int firstminor, unsigned int count, char *name)
```

The `/proc/devices` file lists the current associations between major numbers and device drivers.

Step 5. Registering the driver for a char device (2)

The kernel describe a character device through a `struct cdev` structure.

This data structure has a pointer to the `struct file_operations` of the device, so to complete the device's registration, we need to implement the file operations and to initialize a `struct file_operations` for the device. We'll do this later in this lecture.

The `struct cdev` structure can be initialized by calling:

```
void cdev_init(struct cdev *cdev,
                  struct file_operations *fops)
```

Finally, to inform the kernel about the new char device we can use:

```
int cdev_add(struct cdev *cdev, dev_t num,
                              unsigned int count)
```

Step 8. Deregistering the device file

When the driver module is unloaded or when the device is not more available the device file must be deregistered. At this scope we must use:

```
void cdev_del(struct cdev *dev)
```

To release also the device numbers obtained with `alloc_chrdev_region(...)` or with `register_chrdev_region(...)`, we must use:

```
unregister_chrdev_region(dev_t first,
                         unsigned int count)
```

Step 6. Implementing the file operations for a device file (1)

To any function that operates on a file is passed a pointer to a data structure, called `file`, that describes an opened file. This structure is created by the kernel when a file is opened and it is removed when a file is closed.
Some interesting fields are:

- `mode_t f_mode`: read/write permission flags, set by `FMODE_READ` and `FMODE_WRITE` macro
- `unsigned int f_flags`: other flags, such as `O_RDWR`, `O_NONBLOCK`,...
- `struct file_operations *f_op`: a pointer to the `struct file_operations` of the file
- `void *private_data`: this field is initilized to `NULL` and can be used to specific driver data

Step 6. Implementing the file operations for a device file (2)

Some file operations, such as *open* and *release*, accept as argument a pointer to the `struct inode` that describe the device file. In general, writing a driver, two fields of `inode` structure are interesting:

- `dev_t i_rdev`: contain the device number
- `struct cdev *i_cdev`: pointer to `struct cdev` passed to `cdev_add(...)` function.

Generally, other details on how to implement the file operations are device-dependent. We will see an example for the Galil DMC 1800 device.

Step 7. Managing the operations of an I/O device

Besides implementing the file operations, device drivers must control the termination of each I/O operation.

If your device issues interrupts on an interrupt line, its driver must include an interrupt handler to handle each interrupt raised.

Handling an interrupt means associate with an IRQ (Interrupt ReQuest) number a function that will be called for each interrupt raised on the registred channel.

This function, which typically performs some device-dependent operations, must respect some restrictions: because it runs in atomic context, it can't call any functions that could block and can't access the User Mode address space.

Interrupt

To request an IRQ you must know the IRQ number associated with the device. For a PCI device we can get it from the `struct pci_dev` structure that describes the device: the field's name is `irq`. Then we can request it to the kernel:

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

To release the IRQ:

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

The `/proc/interrupts` file yields the number of interrupts raised for each IRQ and the corresponding device's name.