Linux Kernel Hacking Free Course, 3rd edition

E. Betti
University of Rome “Tor Vergata”

An introduction to I/O drivers

February 15, 2006

Index

The goal of this lecture is to explain:

- what is a driver

Index

The goal of this lecture is to explain:

- what is a driver
- which are the driver's tasks

Index

The goal of this lecture is to explain:

- what is a driver
- which are the driver's tasks

In order to do this, we need to introduce some general concepts:

- common programming model

Index

The goal of this lecture is to explain:

- what is a driver
- which are the driver's tasks

In order to do this, we need to introduce some general concepts:

- common programming model
- device file

Index

The goal of this lecture is to explain:

- what is a driver
- which are the driver's tasks

In order to do this, we need to introduce some general concepts:

- common programming model
- device file
- file operations

Index

The goal of this lecture is to explain:

- what is a driver
- which are the driver's tasks

In order to do this, we need to introduce some general concepts:

- common programming model
- device file
- file operations

Once this is done, we'll start discussing how to implement an example driver for a PCI device. We'll continue discussing the implementation in the next lecture of February 22.

Linux Filesystem

In Linux there are several kinds of files:

Linux Filesystem

In Linux there are several kinds of files: regular files

Linux Filesystem

In Linux there are several kinds of files: regular files, directories

Linux Filesystem

In Linux there are several kinds of files: regular files, directories, symbolic links

Linux Filesystem

In Linux there are several kinds of files: regular files, directories, symbolic links, FIFOs

Linux Filesystem

In Linux there are several kinds of files: regular files, directories, symbolic links, FIFOs, sockets

Linux Filesystem

In Linux there are several kinds of files: regular files, directories, symbolic links, FIFOs, sockets, *block and char device files*.

Linux Filesystem

In Linux there are several kinds of files: regular files, directories, symbolic links, FIFOs, sockets, *block and char device files*.

The device files represent the I/O devices, so each supported I/O device has one or more corrisponding device files.

Linux Filesystem

In Linux there are several kinds of files: regular files, directories, symbolic links, FIFOs, sockets, *block and char device files*.

The device files represent the I/O devices, so each supported I/O device has one or more corrisponding device files.

The device files are generally located in the `/dev` directory and created by using the `mknod` command, or dinamically by the `udev` daemon.

Linux Filesystem

In Linux there are several kinds of files: regular files, directories, symbolic links, FIFOs, sockets, *block and char device files*.

The device files represent the I/O devices, so each supported I/O device has one or more corrisponding device files.

The device files are generally located in the `/dev` directory and created by using the `mknod` command, or dinamically by the `udev` daemon.

A common programming model is used for both regular files and device files.

Common programming model

All Unix kernels, including Linux, have a software layer that handles all system calls related to a file, such as `open()`, `close()`, `read()`, `write()`, and so on . . .

Common programming model

All Unix kernels, including Linux, have a software layer that handles all system calls related to a file, such as `open()`, `close()`, `read()`, `write()`, and so on . . .

This model hides the differences between device files and regular files.

Common programming model

All Unix kernels, including Linux, have a software layer that handles all system calls related to a file, such as `open()`, `close()`, `read()`, `write()`, and so on . . .

This model hides the differences between device files and regular files. In fact:

- when a process accesses a regular file, it is accessing data blocks on a disk partition through a filesystem

Common programming model

All Unix kernels, including Linux, have a software layer that handles all system calls related to a file, such as `open()`, `close()`, `read()`, `write()`, and so on . . .

This model hides the differences between device files and regular files. In fact:

- when a process accesses a regular file, it is accessing data blocks on a disk partition through a filesystem
- when a process accesses a device file, it is just driving a hardware device

Common programming model

All Unix kernels, including Linux, have a software layer that handles all system calls related to a file, such as `open()`, `close()`, `read()`, `write()`, and so on . . .

This model hides the differences between device files and regular files. In fact:

- when a process accesses a regular file, it is accessing data blocks on a disk partition through a filesystem
- when a process accesses a device file, it is just driving a hardware device

. . . but the system calls used are the same!

Class of devices

There are three classes of devices: block device, char device, network interface.

Class of devices

There are three classes of devices: block device, char device, network interface. Block and char device files allow to access, respectively:

- block device: disk-like devices, in which data can be accessed by block number. In most Unix systems, a block device can transfer one or more blocks at a time (usually 512 bytes or another power of two).

Class of devices

There are three classes of devices: block device, char device, network interface. Block and char device files allow to access, respectively:

- block device: disk-like devices, in which data can be accessed by block number. In most Unix systems, a block device can transfer one or more blocks at a time (usually 512 bytes or another power of two).
- char device: almost all other devices, in which data can be read and written as byte streams; random accesses are usually not feasible on char devices

Class of devices

There are three classes of devices: block device, char device, network interface. Block and char device files allow to access, respectively:

- block device: disk-like devices, in which data can be accessed by block number. In most Unix systems, a block device can transfer one or more blocks at a time (usually 512 bytes or another power of two).
- char device: almost all other devices, in which data can be read and written as byte streams; random accesses are usually not feasible on char devices

Network cards are special devices that do not have a device file, but are managed by a network interface identified through a unique name (such as `eth0`)

Major and minor number (1)

All information needed by the filesystem to handle a file is included in a file's descriptor called the inode. A device file is identified by a triplet (boolean, integer, integer) stored in the file's inode.

Major and minor number (1)

All information needed by the filesystem to handle a file is included in a file's descriptor called the inode. A device file is identified by a triplet (boolean, integer, integer) stored in the file's inode.

The boolean determines whether the file is a character device file or a block device file. The two integers are the major and minor device numbers.

Major and minor number (1)

All information needed by the filesystem to handle a file is included in a file's descriptor called the inode. A device file is identified by a triplet (boolean, integer, integer) stored in the file's inode.

The boolean determines whether the file is a character device file or a block device file. The two integers are the major and minor device numbers.

Traditionally, the major number identifies the driver associated with the device, even if modern Linux kernels allow multiple drivers to share major numbers.

Major and minor number (1)

All information needed by the filesystem to handle a file is included in a file's descriptor called the inode. A device file is identified by a triplet (boolean, integer, integer) stored in the file's inode.

The boolean determines whether the file is a character device file or a block device file. The two integers are the major and minor device numbers.

Traditionally, the major number identifies the driver associated with the device, even if modern Linux kernels allow multiple drivers to share major numbers.

The minor number identifies which device is being referred to, even if a single can be referred by more than one device file.

Major and minor number (1)

All information needed by the filesystem to handle a file is included in a file's descriptor called the inode. A device file is identified by a triplet (boolean, integer, integer) stored in the file's inode.

The boolean determines whether the file is a character device file or a block device file. The two integers are the major and minor device numbers.

Traditionally, the major number identifies the driver associated with the device, even if modern Linux kernels allow multiple drivers to share major numbers.

The minor number identifies which device is being referred to, even if a single can be referred by more than one device file.

The same major number is used with different meanings for char and block devices.

Major and minor number (2)

With the `ls -l` command we can see the three values in the device file's inode.

Major and minor number (2)

With the `ls -l` command we can see the three values in the device file's inode.

On my PC with `ls -l /dev/hd* /dev/lp*` command, I get:

```
brw-rw----  1 root disk   3, 0 2006-02-09 17:50 /dev/hda
brw-rw----  1 root disk   3, 1 2006-02-09 17:50 /dev/hda1
brw-rw----  1 root disk   3, 2 2006-02-09 17:50 /dev/hda2
brw-rw----  1 root cdrom 22, 0 2006-02-09 17:50 /dev/hdc
crw-rw----  1 root lp     6, 0 2006-02-09 17:50 /dev/lp0
```

Major and minor number (2)

With the `ls -l` command we can see the three values in the device file's inode.

On my PC with `ls -l /dev/hd* /dev/lp*` command, I get:

```
brw-rw----  1 root disk   3, 0 2006-02-09 17:50 /dev/hda
brw-rw----  1 root disk   3, 1 2006-02-09 17:50 /dev/hda1
brw-rw----  1 root disk   3, 2 2006-02-09 17:50 /dev/hda2
brw-rw----  1 root cdrom 22, 0 2006-02-09 17:50 /dev/hdc
crw-rw----  1 root lp     6, 0 2006-02-09 17:50 /dev/lp0
```

↑

Block or Char Device

Major and minor number (2)

With the `ls -l` command we can see the three values in the device file's inode.

On my PC with `ls -l /dev/hd* /dev/lp*` command, I get:

```
brw-rw----  1 root disk   3, 0 2006-02-09 17:50 /dev/hda
brw-rw----  1 root disk   3, 1 2006-02-09 17:50 /dev/hda1
brw-rw----  1 root disk   3, 2 2006-02-09 17:50 /dev/hda2
brw-rw----  1 root cdrom 22, 0 2006-02-09 17:50 /dev/hdc
crw-rw----  1 root lp     6, 0 2006-02-09 17:50 /dev/lp0
↑                         ↑
```

Block or Char Device ↑

Major number

Major and minor number (2)

With the `ls -l` command we can see the three values in the device file's inode.

On my PC with `ls -l /dev/hd* /dev/lp*` command, I get:

```
brw-rw----  1 root disk   3, 0 2006-02-09 17:50 /dev/hda
brw-rw----  1 root disk   3, 1 2006-02-09 17:50 /dev/hda1
brw-rw----  1 root disk   3, 2 2006-02-09 17:50 /dev/hda2
brw-rw----  1 root cdrom 22, 0 2006-02-09 17:50 /dev/hdc
crw-rw----  1 root lp     6, 0 2006-02-09 17:50 /dev/lp0
↑                         ↑  ↑
Block or Char Device      ↑   Minor Number
                          Major number
```

Major and minor number (3)

The kernel uses the `dev_t` type to hold device numbers.

Major and minor number (3)

The kernel uses the `dev_t` type to hold device numbers. It is a 32 bit variable in which:

- the first 12 bits are used for the major number (so max $2^{12} = 4\,096$ major numbers);

Major and minor number (3)

The kernel uses the `dev_t` type to hold device numbers. It is a 32 bit variable in which:

- the first 12 bits are used for the major number (so max $2^{12} = 4\,096$ major numbers);
- the last 20 bits are used for the minor number (so max $2^{20} = 1\,048\,576$ minor numbers).

Major and minor number (3)

The kernel uses the `dev_t` type to hold device numbers. It is a 32 bit variable in which:

- the first 12 bits are used for the major number (so max $2^{12} = 4\,096$ major numbers);
- the last 20 bits are used for the minor number (so max $2^{20} = 1\,048\,576$ minor numbers).

The `dev_t` type must not be handled directly; rather, the programmer must use some macros as:

```
MAJOR(dev_t dev)
MINOR(dev_t dev)
MKDEV(int major, int minor)
```

File operations (1)

Linux manages inodes through an `inode` structure; one of its field is a pointer to another important structure: the `struct file_operations`.

File operations (1)

Linux manages inodes through an `inode` structure; one of its field is a pointer to another important structure: the `struct file_operations`.

This structure contains pointers to low level functions that implement the hardware (or filesystem) dependent operations of each of the file's system calls.

```
struct file_operations {
   loff_t (*llseek) (struct file *, loff_t, int);
   ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
   int (*open) (struct inode *, struct file *);
   int (*release) (struct inode *, struct file *);
   /* ...and many other fields... */
};
```

File operations (2)

Each driver must implement the proper file operations for its device file and store the pointers to these functions into a `struct file_operations`.

File operations (2)

Each driver must implement the proper file operations for its device file and store the pointers to these functions into a `struct file_operations`.

Linux ensures that the inode of a device file includes a pointer to the `struct file_operations` filled by the device driver.

File operations (2)

Each driver must implement the proper file operations for its device file and store the pointers to these functions into a `struct file_operations`.

Linux ensures that the inode of a device file includes a pointer to the `struct file_operations` filled by the device driver.

Each system call acting on that device file triggers the execution of a file operation provided by the device driver.

What is a driver?

A driver is a set of programs that makes a hardware device respond to the programming interface defined by the file operations.

What is a driver?

A driver is a set of programs that makes a hardware device respond to the programming interface defined by the file operations.

Each hardware device has:

- a standard interface (such as a PCI interface)

What is a driver?

A driver is a set of programs that makes a hardware device respond to the programming interface defined by the file operations.

Each hardware device has:

- a standard interface (such as a PCI interface)
- a device-specific interface

What is a driver?

A driver is a set of programs that makes a hardware device respond to the programming interface defined by the file operations.

Each hardware device has:

- a standard interface (such as a PCI interface)
- a device-specific interface

In general, the standard interface is controlled directly by kernel core functions.

What is a driver?

A driver is a set of programs that makes a hardware device respond to the programming interface defined by the file operations.

Each hardware device has:

- a standard interface (such as a PCI interface)
- a device-specific interface

In general, the standard interface is controlled directly by kernel core functions.

The device-specific interface must be controlled by a dedicated software that knows each device features

What is a driver?

A driver is a set of programs that makes a hardware device respond to the programming interface defined by the file operations.

Each hardware device has:

- a standard interface (such as a PCI interface)
- a device-specific interface

In general, the standard interface is controlled directly by kernel core functions.

The device-specific interface must be controlled by a dedicated software that knows each device features, that is, by the *device driver*.

Driver’s tasks

1. register itself on the software layer of the hosting bus

Driver's tasks

1. register itself on the software layer of the hosting bus

2. probe for compatible devices

Driver’s tasks

1. register itself on the software layer of the hosting bus
2. probe for compatible devices
3. for each device found, obtain its resources

Driver’s tasks

1. register itself on the software layer of the hosting bus
2. probe for compatible devices
3. for each device found, obtain its resources
4. initialize the device

Driver's tasks

1. register itself on the software layer of the hosting bus
2. probe for compatible devices
3. for each device found, obtain its resources
4. initialize the device
5. register itself as a driver for block or char device, thus assigning to the device a major and minor number

Driver's tasks

1. register itself on the software layer of the hosting bus
2. probe for compatible devices
3. for each device found, obtain its resources
4. initialize the device
5. register itself as a driver for block or char device, thus assigning to the device a major and minor number. To make this a driver must:
    6. implement the file operations of the device file

Driver’s tasks

1. register itself on the software layer of the hosting bus
2. probe for compatible devices
3. for each device found, obtain its resources
4. initialize the device
5. register itself as a driver for block or char device, thus assigning to the device a major and minor number. To make this a driver must:
    6. implement the file operations of the device file
    7. manage the operations of the device

Driver’s tasks (2)

Moreover, if the driver is a module or if the device supports hotplugging:

8. deregister the device file

Driver’s tasks (2)

Moreover, if the driver is a module or if the device supports hotplugging:

8. deregister the device file
9. release the resources

Driver's tasks (2)

Moreover, if the driver is a module or if the device supports hotplugging:

8. deregister the device file
9. release the resources
10. deregister itself from the bus software layer

Driver’s tasks (2)

Moreover, if the driver is a module or if the device supports hotplugging:

8. deregister the device file
9. release the resources
10. deregister itself from the bus software layer

In the remaining part of this lecture we’ll show how to implement steps 1, 2, 3, 9, and 10 for a generic driver of a PCI device.

An example driver

For the sake of concreteness, we refer to a Galil DMC 1800 motion controller.

An example driver

For the sake of concreteness, we refer to a Galil DMC 1800 motion controller.

In particular we'll see:

- how to register a PCI driver

An example driver

For the sake of concreteness, we refer to a Galil DMC 1800 motion controller.

In particular we'll see:

- how to register a PCI driver
- how to probe a PCI device

An example driver

For the sake of concreteness, we refer to a Galil DMC 1800 motion controller.

In particular we'll see:

- how to register a PCI driver
- how to probe a PCI device
- how to get resources for a PCI device

An example driver

For the sake of concreteness, we refer to a Galil DMC 1800 motion controller.

In particular we'll see:

- how to register a PCI driver
- how to probe a PCI device
- how to get resources for a PCI device
- how to release the resources

An example driver

For the sake of concreteness, we refer to a Galil DMC 1800 motion controller.

In particular we'll see:

- how to register a PCI driver
- how to probe a PCI device
- how to get resources for a PCI device
- how to release the resources
- how to unregister a PCI driver

Step 1 - Register PCI driver (1)

In order to register a PCI device, the developer must allocate and initialize two structures: an array of `struct pci_device_id` (the last element must be zeroed) and a `struct pci_driver`.

Step 1 - Register PCI driver (1)

In order to register a PCI device, the developer must allocate and initialize two structures: an array of `struct pci_device_id` (the last element must be zeroed) and a `struct pci_driver`.

`struct pci_device_id` is used to identify each PCI-compatible device by matching some fields of the PCI configuration space.

```
struct pci_device_id {
   __u32 vendor, device;  /* Vendor and device ID or PCI_ANY_ID */
   __u32 subvendor, subdevice;  /* Subsystem ID's or PCI_ANY_ID */
   __u32 class, class_mask; /* (class,subclass,prog-if) triplet */
   kernel_ulong_t driver_data;    /* Data private to the driver */
};
```

PCI configuration space for a generic device

<table>
<tr><th></th><th>0x0</th><th>0x1</th><th>0x2</th><th>0x3</th><th>0x4</th><th>0x5</th><th>0x6</th><th>0x7</th></tr>
<tr><td>0x00</td><td colspan="2">Vendor ID</td><td colspan="2">Device ID</td><td colspan="2">Command reg.</td><td colspan="2">Status reg.</td></tr>
<tr><td>0x08</td><td>RI</td><td colspan="3">Class Code</td><td>CL</td><td>LT</td><td>HT (=0)</td><td>BIST</td></tr>
<tr><td>0x10</td><td colspan="4">Base address 0</td><td colspan="4">Base address 1</td></tr>
<tr><td>0x18</td><td colspan="4">Base address 2</td><td colspan="4">Base address 3</td></tr>
<tr><td>0x20</td><td colspan="4">Base address 4</td><td colspan="4">Base address 5</td></tr>
<tr><td>0x28</td><td colspan="4">Card Bus CIS Pointer</td><td colspan="2">Subsystem vendor ID</td><td colspan="2">Subsystem device ID</td></tr>
<tr><td>0x30</td><td colspan="4">Expansion ROM base address</td><td>CP</td><td colspan="3">Reserved</td></tr>
<tr><td>0x38</td><td colspan="4">Reserved</td><td>IL</td><td>IP</td><td>MG</td><td>ML</td></tr>
</table>

RI=Revision ID, CL=Cache Line, LT=Latency Timer, HT=Header Type, BIST=Built-In Self Test, CP=Capabilities Pointer, IL=IRQ Line, IP=IRQ Pin, MG=MIN_GNT, ML=MAX_LAT

PCI Configuration space for Galil DMC 1800

With the `lspci -vvx` command you get the list of PCI devices recognized by the kernel and a dump of their PCI configuration spaces.

PCI Configuration space for Galil DMC 1800

With the `lspci -vvx` command you get the list of PCI devices recognized by the kernel and a dump of their PCI configuration spaces.

```
00:0a.0 Class ff00: PLX Technology, Inc. PCI <-> IOBus Bridge (rev 02)
    Subsystem: I-Bus: Unknown device 1800
    /* ...other information... */
    Interrupt: pin A routed to IRQ 18
    Region 0: Memory at ee000000 (32-bit, non-prefetchable) [size=128]
    Region 2: I/O ports at e800 [size=16]
    00: b5 10 50 90 03 00 80 02 02 00 00 ff 08 00 00 00
    10: 00 00 00 ee 00 00 00 00 01 e8 00 00 00 00 00 00
    20: 00 00 00 00 00 00 00 00 00 00 00 00 79 10 00 18
    30: 00 00 00 00 00 00 00 00 00 00 00 00 09 01 00 00
```

Step 1 - Register PCI driver (2)

Filling the `struct pci_device_id` for the Galil DMC 1800:

```
struct pci_device_id galil1800_idtable[] = {
   {  .vendor = 0x10B5,
      .device = 0x9050,
      .subvendor = 0x1079,
      .subdevice = 0x1800,
      .class = 0,
      .class_mask = 0,
      .driver_data = 0 },
   { 0, }
};
```

If the driver supports more than one device, this array contains one `struct pci_device_id` for each supported device.

Step 1 - Register PCI driver (3)

`struct pci_driver` defines some functions to handle some events and a pointer to the `pci_device_id` table.

```
struct pci_driver {
   char *name;
   const struct pci_device_id *id_table;
   int  (*probe)  (struct pci_dev *dev, const struct pci_device_id *id);
   void (*remove) (struct pci_dev *dev);
   int  (*suspend) (struct pci_dev *dev, pm_message_t state);
   int  (*resume) (struct pci_dev *dev);
   /* ...and other fields... */
};
```

Step 1 - Register PCI driver (4)

`struct pci_driver` for this first version of Galil DMC 1800 driver:

```
struct pci_driver galil1800_driver  = {
    .name = "galil1800",
    .id_table = galil1800_idtable,
    .probe = NULL,
    .remove = NULL,
    .suspend = NULL,
    .resume = NULL,
    .enable_wake = NULL,
    .shutdown = NULL
};
```

Step 1 - Register PCI driver (5)

A minimal `init` function for initializing the module of our driver could be:

```
int __init galil1800_init(void)
{
   return pci_register_driver(&galil1800_driver);
}
```

...where `galil1800_driver` is the instance of the `struct pci_driver` properly initialized, and `pci_register_driver()` is a function exported by the PCI sofware layer of the Linux kernel.

Step 10 - Unregister PCI driver

A minimal `exit` function for terminating the module of our driver could be:

```
void __exit galil1800_cleanup(void)
{
   pci_unregister_driver(&galil1800_driver);
}
```

…where `galil1800_driver` is the instance of the `struct pci_driver` properly initialized, and `pci_unregister_driver()` is a function exported by the PCI software layer of the Linux kernel.

Step 2 - Probe for PCI devices

Once PCI registration is done, the kernel knows from the `pci_device_id` table the devices that can be controlled by the driver.

Step 2 - Probe for PCI devices

Once PCI registration is done, the kernel knows from the `pci_device_id` table the devices that can be controlled by the driver.

For each compatible device found in the system, the kernel calls the function pointed by the `probe` field of `struct pci_driver`.

Step 2 - Probe for PCI devices

Once PCI registration is done, the kernel knows from the `pci_device_id` table the devices that can be controlled by the driver.

For each compatible device found in the system, the kernel calls the function pointed by the `probe` field of `struct pci_driver`.

If a device is already present when the module is loaded, the kernel calls the `probe` function immediately.

Step 2 - Probe for PCI devices

Once PCI registration is done, the kernel knows from the `pci_device_id` table the devices that can be controlled by the driver.

For each compatible device found in the system, the kernel calls the function pointed by the `probe` field of `struct pci_driver`.

If a device is already present when the module is loaded, the kernel calls the `probe` function immediately.

When the module is unloaded or a device is disconnected, the kernel calls the function pointed by the `remove` field of `struct pci_driver`.

Step 3 - Obtain resources for PCI devices (1)

To learn about the resources used by the PCI device, we can read the device's datasheet or we can issue the `lspci -vvx` command.

Step 3 - Obtain resources for PCI devices (1)

To learn about the resources used by the PCI device, we can read the device's datasheet or we can issue the `lspci -vvx` command.

The Galil DMC 1800 card has an I/O port described at Base Address Register 2 and an I/O memory described at Base Address Register 0.

Step 3 - Obtain resources for PCI devices (1)

To learn about the resources used by the PCI device, we can read the device's datasheet or we can issue the `lspci -vvx` command.

The Galil DMC 1800 card has an I/O port described at Base Address Register 2 and an I/O memory described at Base Address Register 0.

The driver usually obtains the resources of the devices found in the system by implementing the corresponding `probe` function.

Step 3 - Obtain resources for PCI devices (1)

To learn about the resources used by the PCI device, we can read the device's datasheet or we can issue the `lspci -vvx` command.

The Galil DMC 1800 card has an I/O port described at Base Address Register 2 and an I/O memory described at Base Address Register 0.

The driver usually obtains the resources of the devices found in the system by implementing the corresponding `probe` function.

The `probe` function accepts as argument a pointer to a `struct pci_dev`. This structure is allocated by kernel, one for each PCI device, and contains all information required to obtain device's resources. In order to read its fields, the driver must use some specific macros.

Step 3 - Obtain resources for PCI devices (2)

This function reads the address stored in the Base Address Register `bar`:

```
pci_resource_start(struct pci_dev *pcidev, int bar)
```

Step 3 - Obtain resources for PCI devices (2)

This function reads the address stored in the Base Address Register `bar`:

`pci_resource_start(struct pci_dev *pcidev, int bar)`

This function reads the resource size pointed by the Base Address Register `bar`:

`pci_resource_len(struct pci_dev *pcidev, int bar)`

Step 3 - Obtain resources for PCI devices (2)

This function reads the address stored in the Base Address Register `bar`:

```
pci_resource_start(struct pci_dev *pcidev, int bar)
```

This function reads the resource size pointed by the Base Address Register `bar`:

```
pci_resource_len(struct pci_dev *pcidev, int bar)
```

This function requests I/O ports from `address` to `address + size`. Tipically `name` is driver's name:

```
struct resource *request_region(unsigned long address,
                  unsigned long size, const char *name)
```

To see the I/O ports and the corresponding drivers: `cat /proc/ioports`

Step 3 - Obtain resources for PCI devices (3)

This function requests I/O memory from `address` to `address + size`. Tipically `name` is driver's name:

```
struct resource *request_mem_region(unsigned long address,
                    unsigned long size, const char *name)
```

To see the I/O memory regions and the corresponding drivers: `cat /proc/iomem`

Step 3 - Obtain resources for PCI devices (3)

This function requests I/O memory from `address` to `address + size`. Tipically `name` is driver's name:

```
struct resource *request_mem_region(unsigned long address,
                 unsigned long size, const char *name)
```

To see the I/O memory regions and the corresponding drivers: `cat /proc/iomem`

The I/O memory address read from PCI configuration space is a *phisical address*. To access this memory area we needs a *linear address*; to obtain a linear address, use this function:

```
void *ioremap(unsigned long phis_address,
                             unsigned long size)
```

Step 9 - Release resources

When the module is unloaded or the device is unplugged, the `remove` function (pointed by a field of `struct pci_driver`) is called.

Step 9 - Release resources

When the module is unloaded or the device is unplugged, the `remove` function (pointed by a field of `struct pci_driver`) is called.

This function must release the resources obtained by the driver. To do this:

- to release I/O ports:

```
release_region(unsigned long address,
               unsigned long size)
```

Step 9 - Release resources

When the module is unloaded or the device is unplugged, the `remove` function (pointed by a field of `struct pci_driver`) is called.

This function must release the resources obtained by the driver. To do this:

- to release I/O ports:
```
release_region(unsigned long address,
                                  unsigned long size)
```

- to release I/O memory:
```
iounmap(void *virtual_address)
release_mem_region(unsigned long phis_address,
                                  unsigned long size)
```