Linux Device Driver

(The PCI Interface)

Amir Hossein Payberah

payberah@yahoo.com

Contents

Introduction

- The PCI architecture was designed as a replacement for the ISA standard, with three main goals:
  - To get better performance when transferring data between the computer and its peripherals.
  - To be as platform independent as possible.
  - To simplify adding and removing peripherals to the system.

Contents

PCI addressing

- Each PCI peripheral is identified by a bus number, a device number, and a function number.
  - The PCI specification permits a system to host up to 256 buses.
  - Each bus hosts up to 32 devices.
  - Each device can be a multifunction board with a maximum of eight functions.
- Each function can thus be identified at hardware level by a 16-bit address, or key.

PCI addressing

- The 16-bit hardware addresses associated with PCI peripherals, are still visible occasionally:
  - lspci
  - /proc/pci
  - /proc/bus/pci

PCI addressing

- When the hardware address is displayed, it can either be shown as a 16-bit value, as two values:
  - An 8-bit bus number
  - An 8-bit device and function number
- As three values
  - Bus
  - Device
  - Functions

Contents

- Introduction
- PCI addressing
- Boot time
- Configuration registers and initialization
- Accessing the configuration space
- Accessing the I/O and memory space

Boot time

- When power is applied to a PCI device, the hardware remains inactive.
- The device will respond only to configuration transactions.
- At power on, the device has no memory and no I/O ports mapped in the computer’s address space.
- Every other device-specific feature, such as interrupt reporting, is disabled as well.

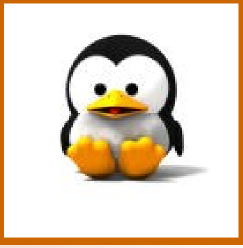
Boot time

- Every PCI motherboard is equipped with PCI-aware firmware:
  - BIOS
  - NVRAM
  - PROM
- At system boot, the firmware (or the Linux kernel, if so configured) performs configuration transactions with every PCI peripheral in order to allocate a safe place for any address region it offers.

Boot time

- PCI device list
  - /proc/bus/pci/devices
- The devices' configuration
  - /proc/bus/pci/*/*

Contents

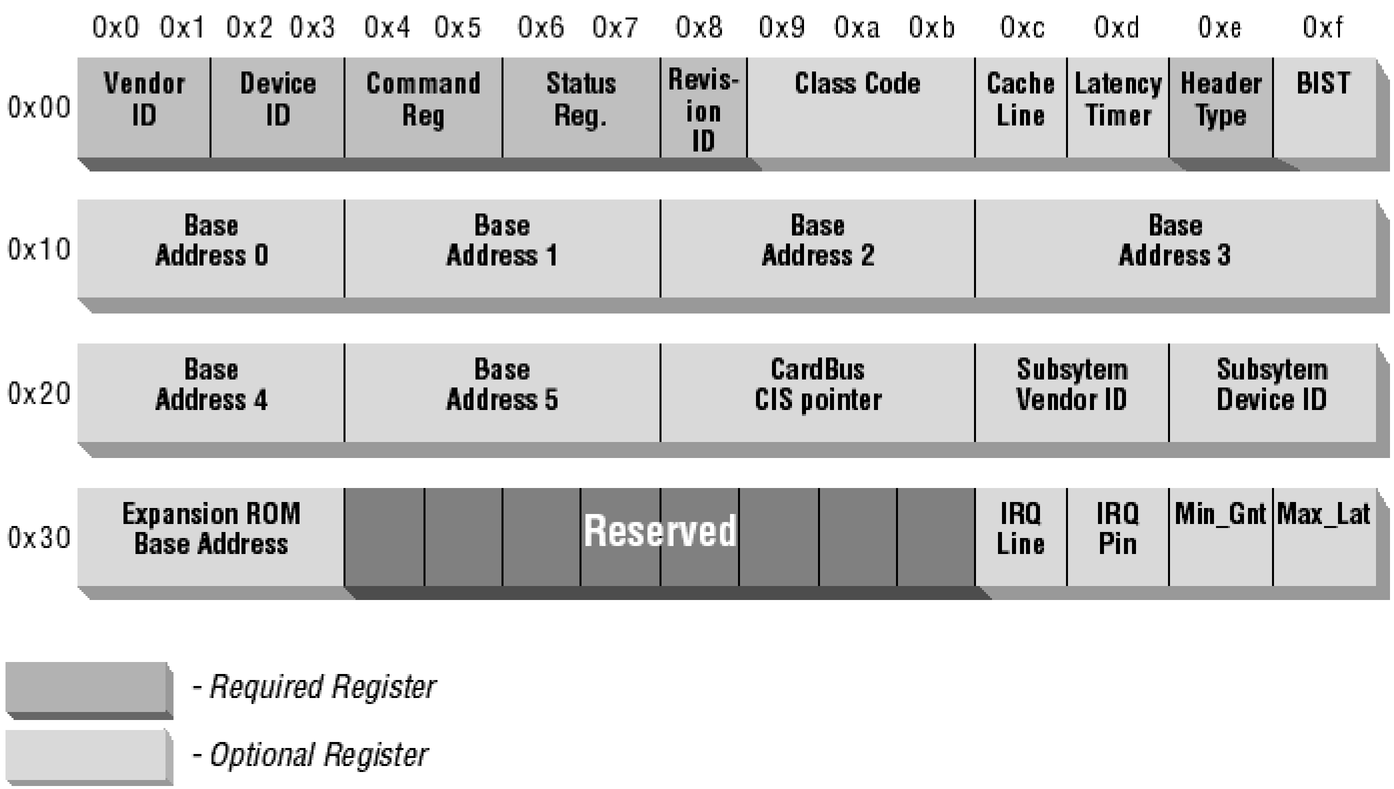


- Introduction
- PCI addressing
- Boot time
- Configuration registers and initialization
- Accessing the configuration space
- Accessing the I/O and memory space

Configuration registers

- The layout of the configuration space is device independent.
- PCI devices feature a 256-byte address space.
- The first 64 bytes are standardized, while the rest are device dependent.

Configuration registers



Configuration registers

- vendorID
  - This 16-bit register identifies a hardware manufacturer.
  - For instance, every Intel device is marked with the same vendor number, 0x8086.
- deviceID
  - This is another 16-bit register, selected by the manufacturer.
  - This ID is usually paired with the vendor ID to make a unique 32-bit identifier for a hardware device.
- Class
  - Every peripheral device belongs to a class.
  - The class register is a 16-bit value whose top 8 bits identify the ‘‘base class’’ (or group).
  - For example, ‘‘ethernet’’ and ‘‘token ring’’ are two classes belonging to the ‘‘network’’ group,

PCI necessary fields

- #include <linux/config.h>
  - By including this header, the driver gains access to the CONFIG_ macros.
- CONFIG_PCI
  - This macro is defined if the kernel includes support for PCI calls.

PCI necessary fields

- #include <linux/pci.h>
  - This header declares all the prototypes as well as the symbolic names associated with PCI registers and bits.
- int pci_present(void);
  - The pci_present function allows one to check if PCI functionality is available or not.
- struct pci_dev;
  - It is at the core of every PCI operation in the system.

PCI functions

- struct pci_dev *pci_find_device (unsigned int vendor, unsigned int device, const struct pci_dev *from);
  - This function is used to scan the list of installed devices looking for a device featuring a specific signature.
- struct pci_dev *pci_find_class (unsigned int class, const struct pci_dev *from);
  - This function is similar to the previous one, but it looks for devices belonging to a specific class.
- int pci_enable_device (struct pci_dev *dev);
  - This function actually enables the device.

Sample

```
#ifndef CONFIG_PCI
# error "This driver needs PCI support to be available"
#endif

int jail_find_all_devices(void)
{
	struct pci_dev *dev = NULL;
	int found;
	if (!pci_present())
		return -ENODEV;
	for (found=0; found < JAIL_MAX_DEV;)
	{
		dev = pci_find_device(JAIL_VENDOR, JAIL_ID, dev);
		if (!dev)
			break;
		found += jail_init_one(dev);
	}
	return (index == 0) ? -ENODEV : 0;
}
```

Contents

- Introduction
- PCI addressing
- Boot time
- Configuration registers and initialization
- Accessing the configuration space
- Accessing the I/O and memory space

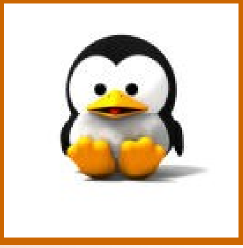
Accessing the Configuration Space

- After the driver has detected the device, it usually needs to read from or write to the three address spaces:
  - Memory
  - Port
  - Configuration
- Accessing the configuration space is vital to the driver because it is the only way it can find out where the device is mapped in memory and in the I/O space.

Configuration access

- int pci_read_config_byte (struct pci_dev *dev, int where, u8 *ptr);
- int pci_read_config_word (struct pci_dev *dev, int where, u16 *ptr);
- int pci_read_config_dword (struct pci_dev *dev, int where, u32 *ptr);
- int pci_write_config_byte (struct pci_dev *dev, int where, u8 val);
- int pci_write_config_word (struct pci_dev *dev, int where, u16 val);
- int pci_write_config_dword (struct pci_dev *dev, int where, u32 val);

Contents



- Introduction
- PCI addressing
- Boot time
- Configuration registers and initialization
- Accessing the configuration space
- Accessing the I/O and memory space

Accessing the I/O and mem Space

- A PCI device implements up to six I/O address regions.
- Each region consists of either memory or I/O locations.

I/O and mem access method

- unsigned long pci_resource_start (struct pci_dev *dev, int bar);
  - The function returns the first address (memory address or I/O port number) associated with one of the six PCI I/O regions.
- unsigned long pci_resource_end (struct pci_dev *dev, int bar);
  - The function returns the last address that is part of the I/O region number bar.
- unsigned long pci_resource_flags (struct pci_dev *dev, int bar);
  - This function returns the flags associated with this resource.

Question?