open wide INGENIERIE

JDEV2013

Pilotes PCI pour Linux

Pierre Ficheux (pierre.ficheux@openwide.fr)

Septembre 2013

Objectifs de la présentation

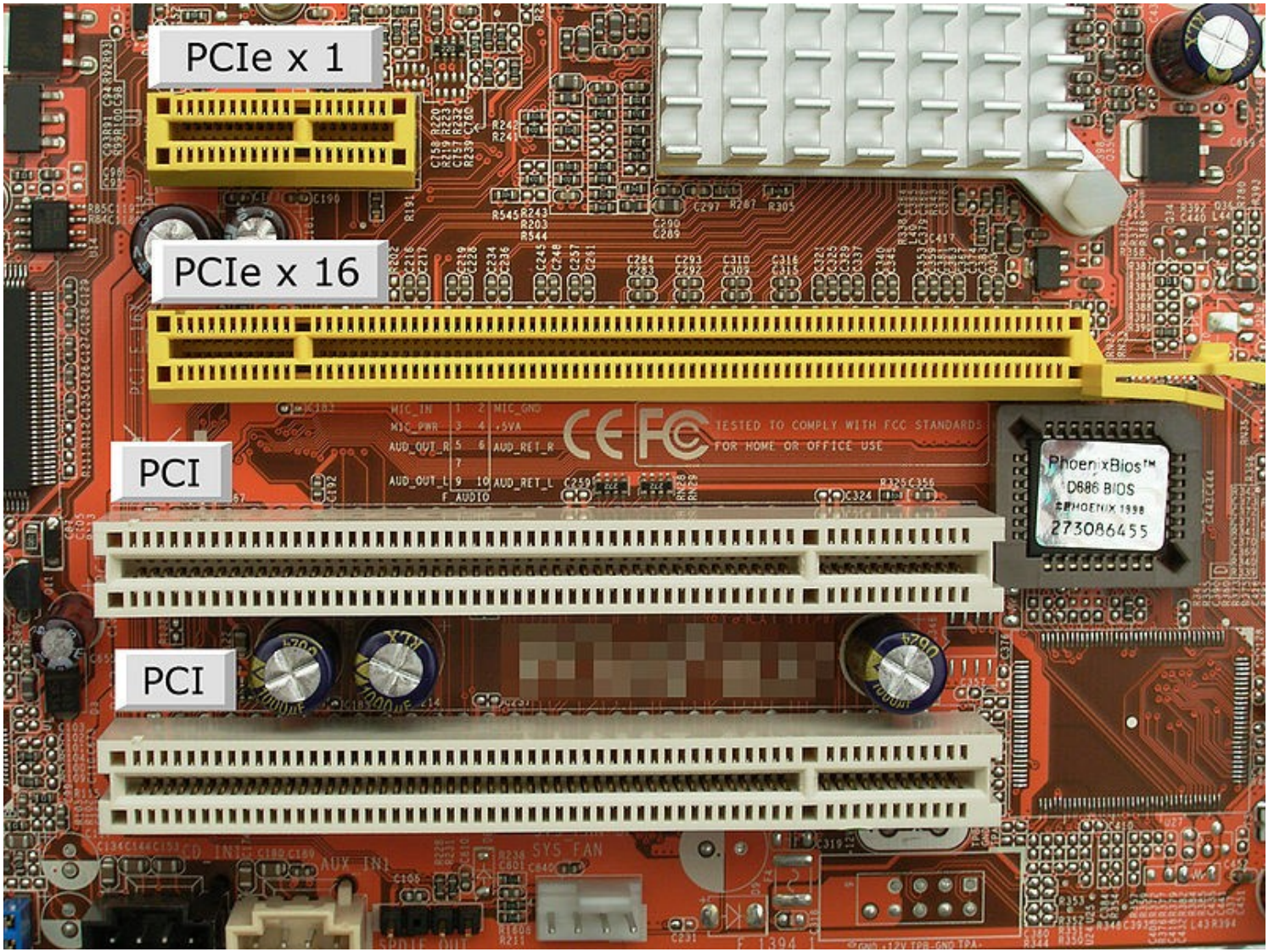
- Introduction générale au bus PCI
- Ressources d'un périphérique PCI
- Bus PCI sous Linux
- Un pilote PCI générique en quelques étapes

REMARQUE: il est nécessaire de connaître l'API des modules Linux et celle des pilotes en mode caractère

Le bus PCI (Peripheral Component Interconnect)

- Standard universel pour le bus d'E/S
- Introduit en 1992 (1.0) par Intel → remplacement du VLB (VESA Local Bus)
- Utilisé sur toutes les architectures: x86, PowerPC, SPARC, …
- Plusieurs versions :
  - 2.2 : 32 bits / 33 MHz / 133 Mo/s
  - 2.2 : 64 bits / 66 MHz / 528 Mo/s
  - PCI-X : 64 bits / 133 MHz / 1066 Mo/s
  - PCI-X 2.0 : 64 bits / 266 MHz / 2133 Mo/s
  - PCI-Express : série / x1 à x32 / 250 Mo/s à 8 Go/s
  - Mini-PCI (2.2)

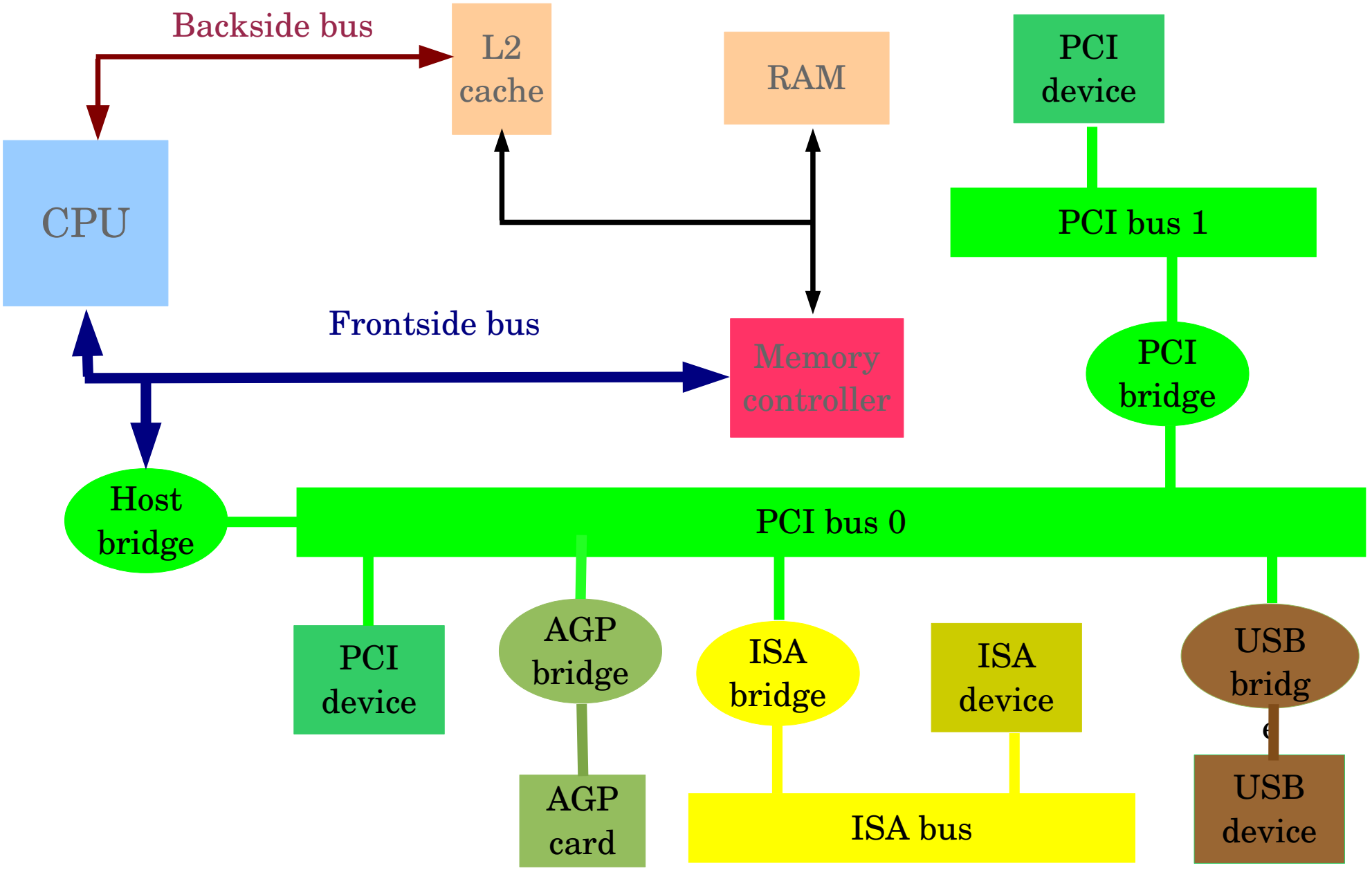
PCI et PCI-Express



Le bus PCI, suite.

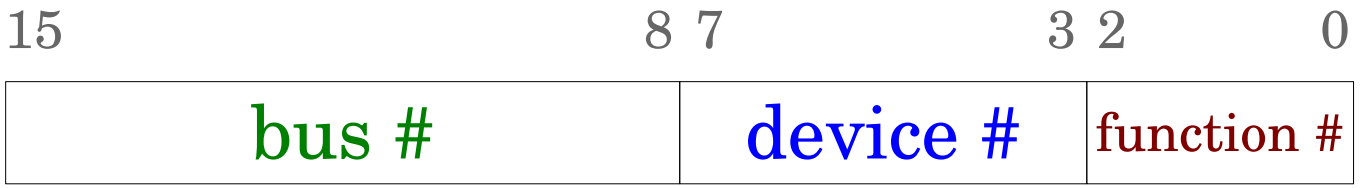
- Transfert en mode *burst* → cadence ininterrompu avec le périphérique
- *Bus mastering* → communication avec un autre périphérique sans passer par le CPU
- Configuration des périphériques en « plug and play » → allocation automatique des adresses, niveaux d'IRQ, ... par le BIOS PCI
- Possibilité de « hot-plug » (exemple: CardBus)

Schéma de l'architecture PCI



Bus, device et fonction

- Chaque périphérique PCI est identifié par une valeur sur 16 bits
  - Numéro de bus (bus #) de 0 à 255
  - Numéro de device (device #) de 0 à 31
  - Numéro de fonction (function #) de 0 à 7
- Chaque carte peut avoir plusieurs fonctions logiques (cf: USB)



Visualisation avec lspci

```
$ lspci
00:00.0 Host bridge: Intel Corporation 4 Series
Chipset DRAM Controller (rev 03)
00:01.0 PCI bridge: Intel Corporation 4 Series
Chipset PCI Express Root Port (rev 03)
00:02.0 VGA compatible controller: Intel Corporation
4 Series Chipset Integrated Graphics Controller (rev
03)
00:02.1 Display controller: Intel Corporation 4
Series Chipset Integrated Graphics Controller (rev
03)
00:03.0 Communication controller: Intel Corporation 4
Series  Chipset HECI Controller (rev 03)
...
```

Format: bus:device.fonction

Ressources d'un périphérique PCI

- Mémoire locale (rien à voir avec la RAM du système !)
- Registres matériels (contrôle, état, données) « mappés » soit par des ports d'E/S soit sur de la mémoire locale
- Lignes d'IRQ, 4 maximum par périphérique
- Régions (6 maximum) pouvant donner accès à des zones mémoire de la carte depuis le bus PCI → BAR = *Base Address Register*, BAR0 → BAR5

Interruptions

- Chaque périphérique PCI peut disposer de 4 lignes d'interruption: #A, #B, #C, #D
- Plusieurs lignes car plusieurs fonctions possibles sur la carte
- Le partage d'interruption est *natif* en PCI → en tenir compte dans les pilotes
- Visible avec `/proc/interrupts`

```
$ cat /proc/interrupts
           CPU0       CPU1
  0:      48373          0   IO-APIC-edge      timer
  ...
 18:          1       3525   IO-APIC-fasteoi   uhci_hcd:usb4, wifi0
```

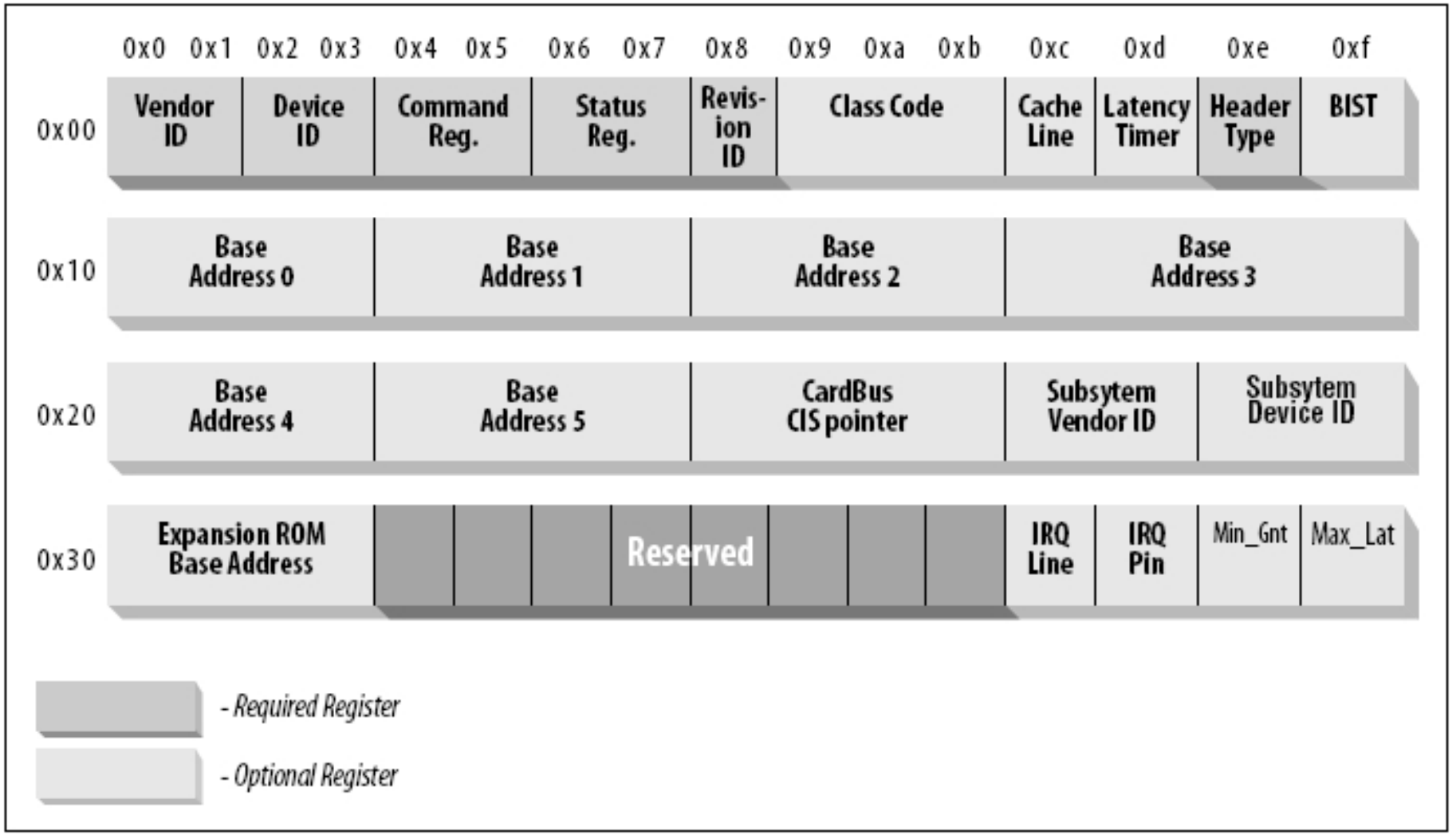
nom du module

Partage du niveau 18

Registres de configuration 1/2

- Chaque périphérique PCI (fonction) définit une plage de 256 octets mini contenant des registres de configuration :
  - Vendor-ID
  - Device-ID (dépend de la fonction)
  - Adresses des BAR
  - IRQ #
  - ...
- L'affectation de certaines valeurs est effectuée dynamiquement au démarrage par le BIOS PCI
- Un pilote PCI *doit* utiliser cette zone de configuration dynamique

Registres de configuration 2/2



Support « plug and play »

- Pas de cavalier sur une carte PCI !
- Pas de nécessité de « découvrir » les ressources de la carte
- La fonction `probe()` du pilote est appelée si le périphérique est détecté (cf USB) en utilisant Vendor-ID / Device-ID
- Parfois on peut utiliser en plus :
  - Subsystem Vendor-ID
  - Subsystem Device-ID

Linux et PCI

- Support très stable, existe depuis « toujours »
- Peu d'évolution de l'API entre 2.4 et 2.6
- Description des périphériques avec `/proc` et `/sys` + `lspci`
- Options de `lspci`
  - nn → affiche le nom + Device-ID / Vendor-ID
  - v → verbeux
  - tv → arborescence
  - x → affiche les registres de configuration
- Entrées dans `/sys/bus/pci/devices`

Un pilote *générique* en quelques étapes

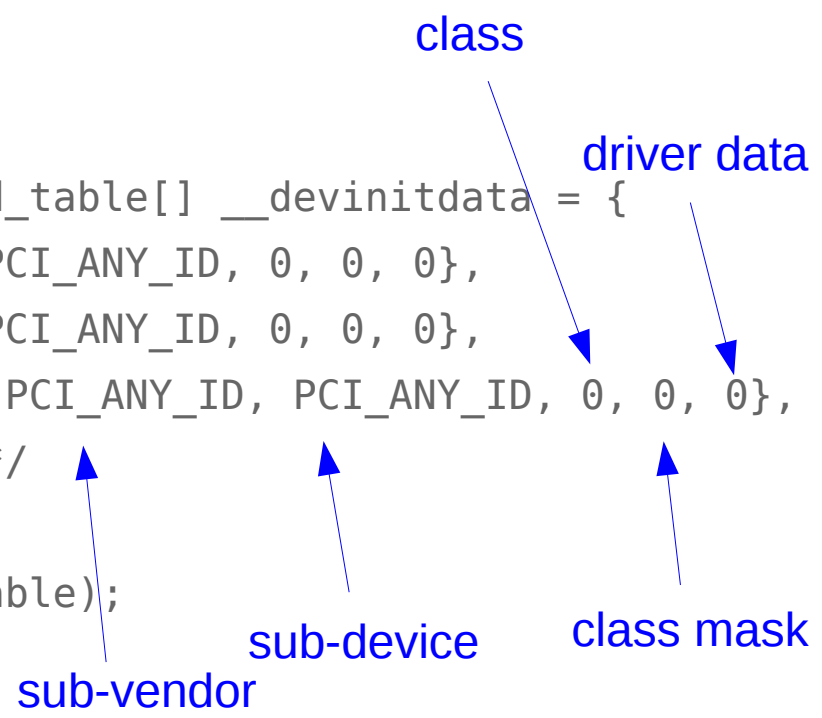
- Initialisation de la table des identifiants (`pci_device_id`)
- Création d'un descripteur de pilote PCI (`pci_driver`)
- Initialisation du périphérique (BAR, IRQ, ...) → fonction `probe()`
- Libération du périphérique → fonction `remove()`
- Chargement/déchargement du module → `module_init()`/`module_exit()` → enregistrement PCI
- Ajout d'une interface utilisateur, exemple en mode caractère

La table des identifiants

- Allouer une table de type `pci_device_id`
- Cette table contient la liste des identifiants pouvant être exploités par ce pilote

```
#include <linux/init.h>
#include <linux/pci.h>

static struct pci_device_id pcidemo_id_table[] __devinitdata = {
        {0x10ec, 0x8136, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0},
        {0x168c, 0x001c, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0},
        {PCI_VENDOR_ID_INTEL, 0x1050, PCI_ANY_ID, PCI_ANY_ID, 0, 0, 0},
        {0,}    /* 0 terminated list */
};
MODULE_DEVICE_TABLE(pci, pcidemo_id_table);
```



Descripteur de pilote PCI 1/2

- La structure `pci_driver` décrit le pilote de périphérique
  - Le nom du pilote (chaîne de caractères)
  - L'adresse de la table `pci_device_id`
  - L'adresse de la fonction `probe()`
  - L'adresse de la fonction `remove()`
  - L'adresse de 2 autres fonctions optionnelles concernant le *power-management* : `suspend()`, `resume()`

Descripteur de pilote PCI 2/2

```
static struct pci_driver pcidemo_driver = {
        .name =         "pcidemo",
        .id_table =     pcidemo_id_table,
        .probe =        pcidemo_probe,
        .remove =       pcidemo_remove,
#ifdef CONFIG_PM
        .resume =       pcidemo_resume,
        .suspend =      pcidemo_suspend
#endif
};
```

Initialisation du périphérique: fonction probe()

- La fonction `probe()` est appelée par la couche PCI du noyau pour chaque évènement concernant les périphériques de la liste `pci_device_id`
  - Au chargement du pilote (si module)
  - Au chargement du noyau (si statique)
  - Lorsque le périphérique est « inséré » (détecté)
- La fonction doit retourner 0 si le pilote désire contrôler ce périphérique

Prototype de la fonction probe()

```
static int __devinit pcidemo_probe(struct
pci_dev *dev, const struct pci_device_id *ent)
```

- La structure `pci_dev` décrit un périphérique dans la couche PCI du noyau
- L'identifiant `ent` correspond à l'entrée du périphérique dans la liste `pci_device_id`

```
ent->vendor
ent->device
...
```

Structure privée du périphérique

- Tout comme en USB, on ne doit pas utiliser de variable globale → utilisation d'une liste chaînée pour les périphériques
- La cellule doit contenir un pointeur `pci_dev *`

```
static LIST_HEAD(pcidemo_list);

struct pcidemo_struct {
 struct list_head     link; /* Double linked list */
 struct pci_dev       *dev; /* PCI device */
 int                  minor; /* Minor number */
 void   *mmio[DEVICE_COUNT_RESOURCE];    /* I/O ptr */
 u32    mmio_len[DEVICE_COUNT_RESOURCE]; /* I/O size */
};
```

Sauvegarde des pointeurs vers les BAR + tailles

Actions effectuées par `probe()` 1/3

- Allocation de la structure des données « privées » du périphérique

```
data = (struct pcidemo_struct *)kmalloc(sizeof(struct pcidemo_struct), GFP_KERNEL);
```

- Sauvegarde de ces données dans le descripteur de matériel de type `pci_dev*`

```
pci_set_drvdata(dev, data);
data->dev = dev;
data->minor = minor++;
```

Actions effectuées par `probe()` 2/3

- Activation du périphérique car il n'est *pas* activé par le BIOS

```
pci_enable_device(dev);
```

- Réservation des ressources E/S et mémoire

```
pci_request_regions(dev, "pcidemo");
```

- Recherche des zones de mémoire (BAR) accessibles → boucle + test du flag `IORESOURCE_MEM`

```
if (pci_resource_flags(dev, i) & IORESOURCE_MEM) {
}
```

- Transformation de l'adresse matérielle en adresse noyau par `ioremap()` + sauvegarde

```
data->mmio[i] = ioremap(pci_resource_start(dev, i), pci_resource_len(dev, i));
```

Actions effectuées par `probe()` 3/3

- Installation de la fonction de traitement d'interruption
  - En théorie, on doit tester `PCI_INTERRUPT_PIN`

```
pci_read_config_byte(dev, PCI_INTERRUPT_PIN, &mypin);
if (mypin) {
 ret = request_irq(dev->irq, pcidemo_irq_handler,
  IRQF_SHARED, "pcidemo", data);
}
```

- Activation du « bus mastering »

```
pci_set_master(dev);
```

- Ajout à la liste chaînée

```
list_add_tail(&data->link, &pcidemo_list);
```

Libération du périphérique, fonction `remove()` 1/2

- On doit libérer tout ce qui est alloué dans `probe()`
- Pointeur vers les données privées

```
struct pcidemo_struct *data = pci_get_drvdata(dev);
```

- Libération des BAR par `iounmap()`

```
iounmap(data->mmio[i]);
```

- Libération des ressources

```
pci_release_regions(dev);
```

- Désactivation du périphérique

```
pci_disable_device(dev);
```

- Libération de l'IRQ

```
free_irq(dev->irq, data);
```

Libération du périphérique, fonction `remove()` 2/2

- Suppression de l'entrée dans la liste chaînée

```
list_del(&data->link);
```

- Libération des données privées

```
kfree(data);
```

Enregistrement du pilote PCI

- L'enregistrement s'effectue dans la fonction `module_init()`

```
pci_register_driver(&pcidemo_driver);
```

- La suppression s'effectue dans `module_exit()`

```
pci_unregister_driver(&pcidemo_driver);
```

Ajout d'une interface en mode caractère 1/2

- Le pilote n'est pour l'instant qu'un module !
- Il suffit d'ajouter les fonctions classiques : `open()`, `release()`, `read()`, `write()`, ...
- Ajout d'une structure `file_operations`
- Enregistrement majeur/mineur

```
register_chrdev()
unregister_chrdev()
```

- Utilisation des classes, etc.
- `open()` → pointer sur le bon mineur car il y a plusieurs périphériques

```
data = list_entry(cur, struct pcidemo_struct, link);
if (data->minor == MINOR(inode->i_rdev)) {
file->private_data = data; return 0;  }
```

Ajout d'une interface en mode caractère 2/2

- Exemple pour la fonction `read()` → accès au contenu de la première zone BAR accessible

```
# hexdump -C /dev/pcidemo0 | more
```

ou bien

```
# od -x /dev/pcidemo0 | more
```

- Utiliser les données privées

```
data->mmio[i]
data->mmio_len[i]
```

Bibliographie et pointeurs

- http://pficheux.free.fr/articles/lmf/pci
- http://free-electrons.com/docs/pci-drivers
- http://lwn.net/images/pdf/LDD3/ch12.pdf