open wide INGENIERIE

JDEV2013

Pilotes USB pour Linux

Pierre Ficheux (pierre.ficheux@openwide.fr)

Septembre 2013

Objectifs de la présentation

- Introduction générale au bus USB
- Bus USB sous Linux
- API des pilotes USB (espace noyau)
- Développement en espace utilisateur (libusb)

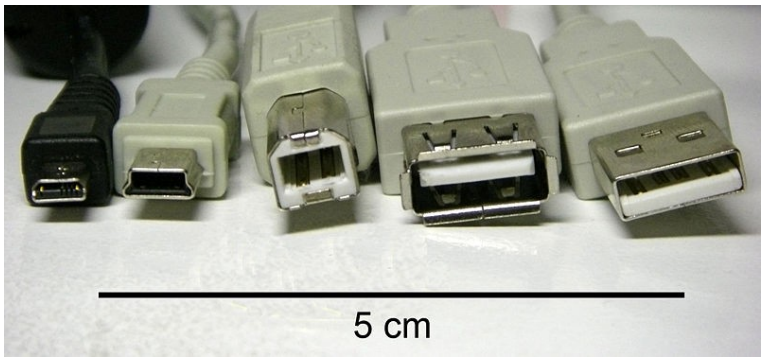
REMARQUE: il est nécessaire de connaître l'API des modules Linux et celle des pilotes en mode caractère

Bus USB: généralités

- Universal Serial Bus (http://www.usb.org)
- Destiné à remplacer les différentes connectiques (RS-232, parallèle, PS/2, joystick, ...)
- Multi-plateforme
- Versions :
  - 1996: 1.0 (low-speed, 1.5 Mbits/s)
  - 1998: 1.1 (full-speed, 12 Mbits/s)
  - 2000: 2.0 (high-speed, 480 Mbits/s)
  - 2008: 3.0 (super-speed, 4.8 Gbits/s)
- Attention, 2.0 n'est pas forcément high-speed !

Matériel 1/2

- 4 fils, transmission différentielle en « paire torsadée » (8 en USB 3.0, mais compatibilité)
  - Alimentation 5V, 500 mA maxi
  - D+
  - D-
  - Masse
- Différentes connectiques



Matériel 2/2 (contrôleurs)

- En USB 1.x, deux types de contrôleurs :
  - OHCI: *Open Host Controller Interface*, développé par COMPAQ
  - UHCI: *Universal Host Controller Interface*, développé par Intel
- En USB 2.0, une seule norme :)
  - EHCI: Extended Host Controller Interface

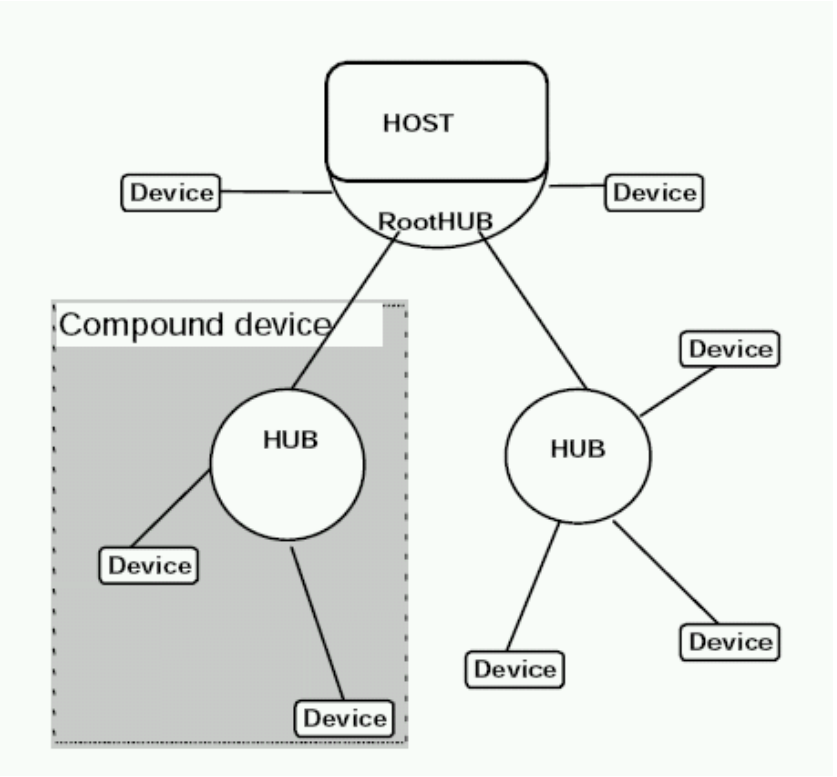
USB is cheap and fun :)

Principe de fonctionnement

- Bus « maître / esclave »
  - Le maître est appelé *hôte* (host) → PC
  - L'esclave est le *périphérique* (device)
- Le maître a *toujours* l'initiative de la communication, pas de véritable mode *interruption*
- Un périphérique peut avoir une interface *host* et une interface *device* (2 connecteurs)
- Il existe aussi une interface unique pouvant jouer les 2 rôles: OTG pour *On-The-Go*

Topologie du bus USB

- Structure arborescente
- Connexion de plusieurs *devices* grâce à un *hub*
- Le *hub* peut être alimenté ou non

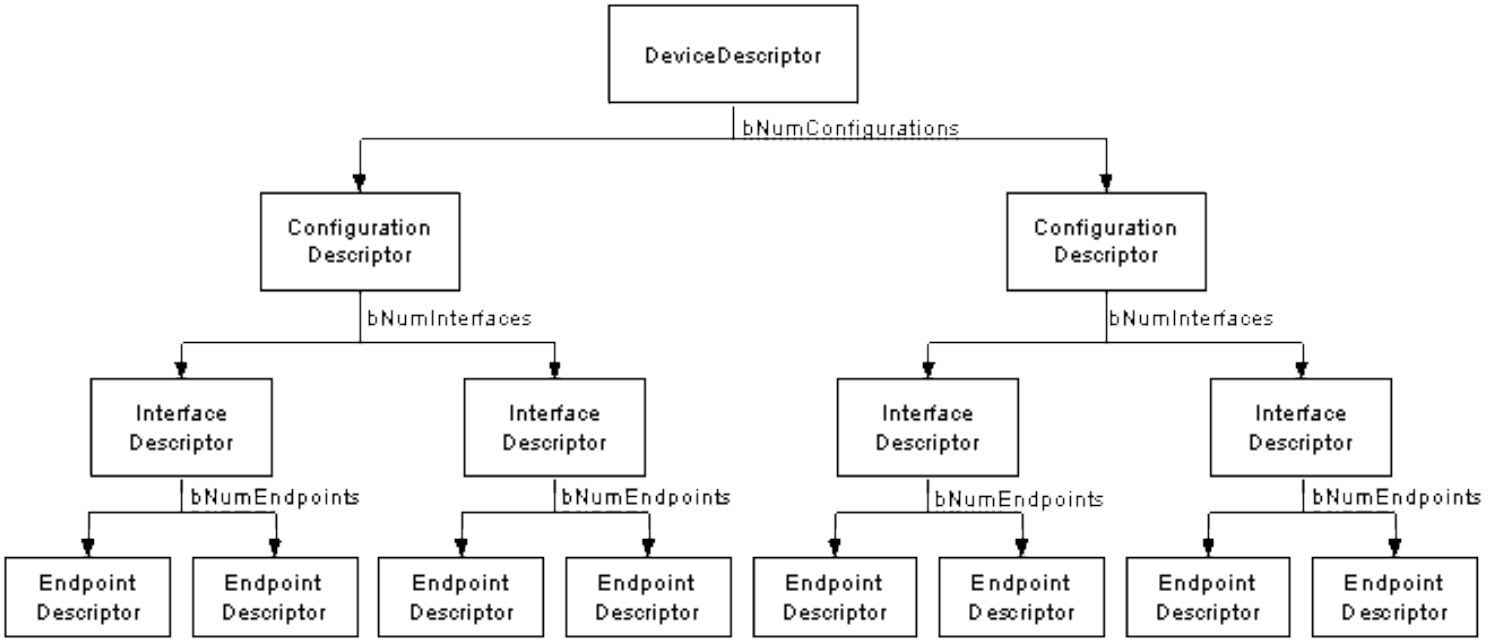


Structure du périphérique 1/3

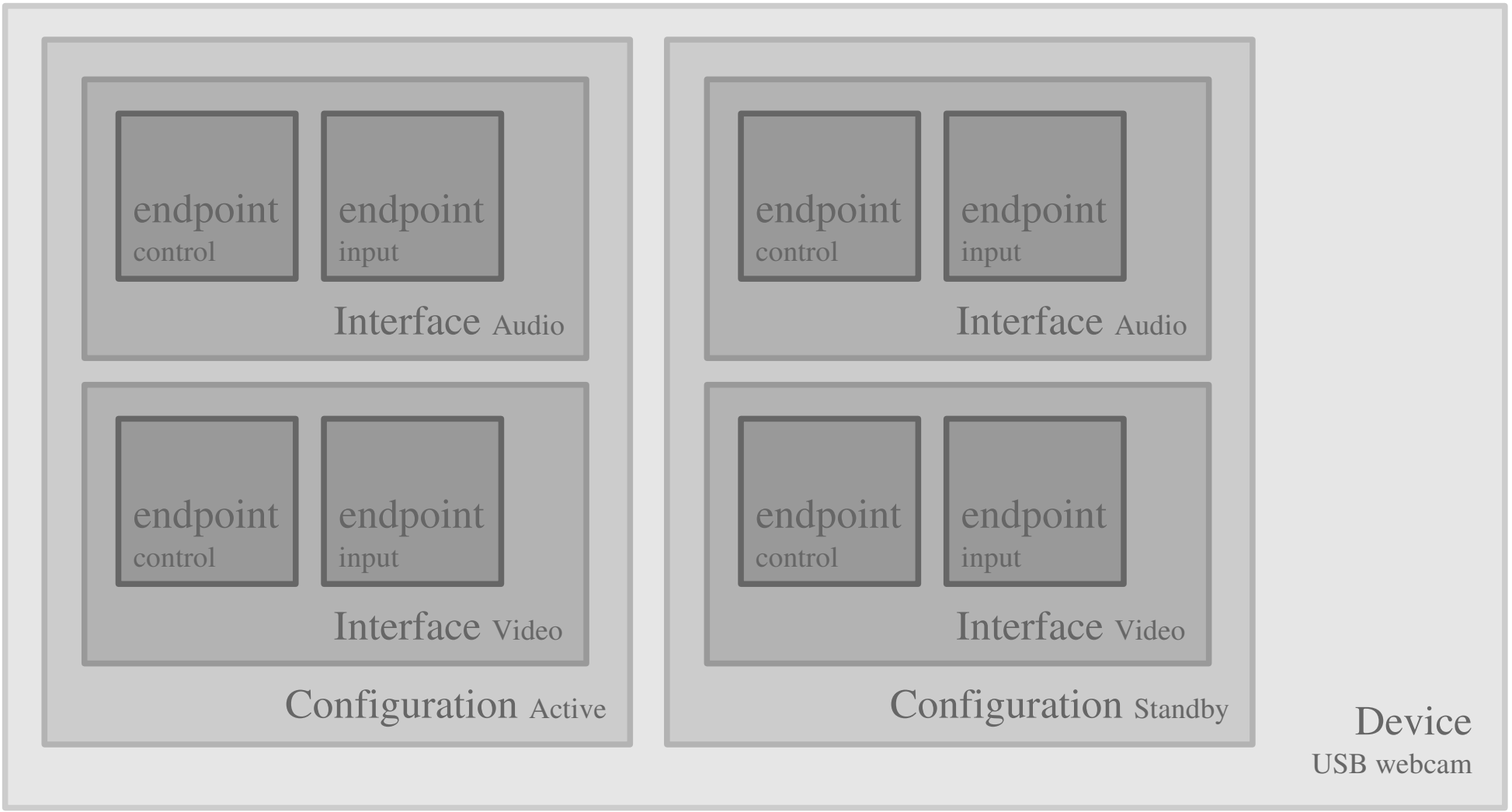
- La structure est également arborescente
- 4 éléments, indépendants du système d'exploitation :
  - *Device*: représentation (unique) du périphérique matériel
  - *Configuration(s)*: état (actif, en veille, initialisation, ...)
  - *Interface(s)*: représentation logique des fonctions (ex: vidéo, audio) => un pilote par interface
  - *Endpoint(s)*: Canal de communication unidirectionnel bas niveau avec le host (IN ou OUT)

Structure du périphérique 2/3

- Les noms sont définis par la norme USB (`bNumConfigurations`, `bNumInterfaces`, ...)

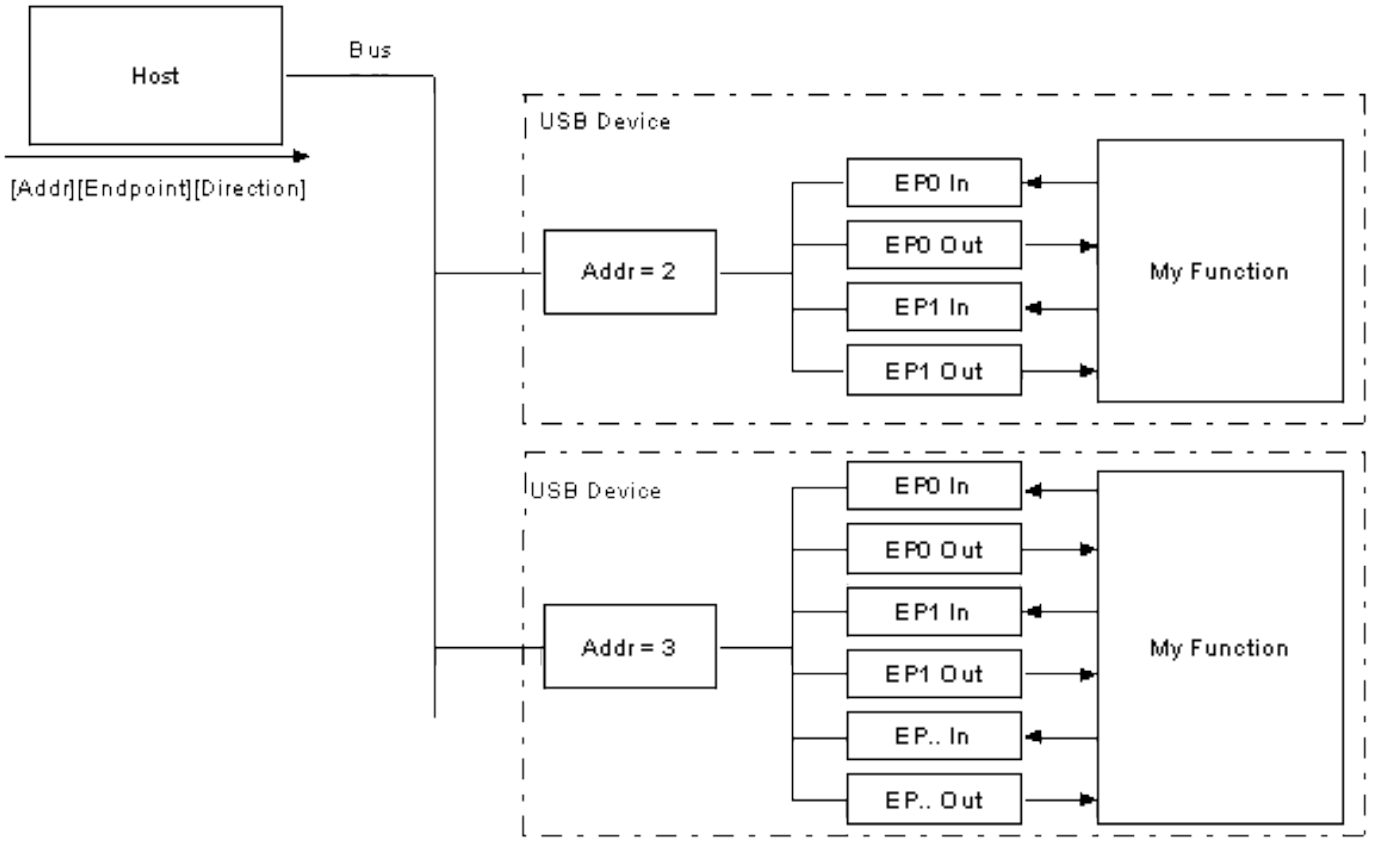


Structure du périphérique 3/3



Communication avec le *host*

- Chaque fonction correspond à un ensemble de *endpoints*
- Communication par « messages »



Les différents types de messages

- Correspondent aux 4 types de *endpoints*
  - CONTROL: configuration du périphérique. Il existe toujours un *endpoint 0* pour ce type de messages. Faible volume.
  - INTERRUPT: transfert à intervalle régulier (ex: souris) sur requête du *host.* Faible volume.
  - BULK: Transfert de gros volumes avec garantie d'intégrité mais pas de débit ni de temps réel
  - ISOCHRONOUS: Gros volumes, garantie de débit mais pas d'intégrité (ex: vidéo)

Support USB sous Linux

- Présent dans la version 2.4
- Largement amélioré dans la version 2.6
- Vu de l'utilisateur, les périphériques sont décrits dans `/proc` et `/sys`
- Insertion d'un périphérique :

```
$ dmesg
usb 6-2: new low speed USB device using uhci_hcd and address 2
usb 6-2: New USB device found, idVendor=1130, idProduct=0202
usb 6-2: New USB device strings: Mfr=0, Product=2, SerialNumber=0
usb 6-2: Product: Panic Button
input: Panic Button as /devices/pci0000:00/0000:00:1d.0/usb6/6-2/6-2:1.0/input/input5
generic-usb 0003:1130:0202.0003: input,hidraw2: USB HID v1.10 Device [Panic Button] on
usb-0000:00:1d.0-2/input0
input: Panic Button as /devices/pci0000:00/0000:00:1d.0/usb6/6-2/6-2:1.1/input/input6
generic-usb 0003:1130:0202.0004: input,hidraw3: USB HID v1.10 Device [Panic Button] on
usb-0000:00:1d.0-2/input1
```

Description du périphérique

- Dans `/proc/bus/usb/devices`

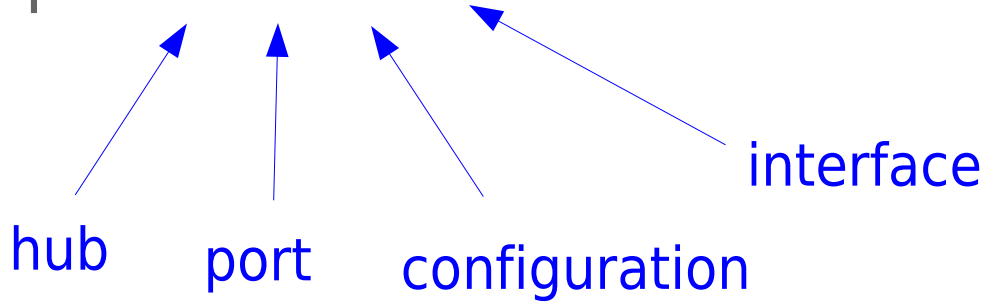
```
T:  Bus=06 Lev=01 Prnt=01 Port=01 Cnt=01 Dev#=  2 Spd=1.5 MxCh= 0
D:  Ver= 1.10 Cls=00(>ifc ) Sub=00 Prot=00 MxPS= 8 #Cfgs=  1
P:  Vendor=1130 ProdID=0202 Rev= 1.00
S:  Product=Panic Button
C:* #Ifs= 2 Cfg#= 1 Atr=80 MxPwr=100mA
I:* If#= 0 Alt= 0 #EPs= 1 Cls=03(HID  ) Sub=00 Prot=00 Driver=usbhid
E:  Ad=81(I) Atr=03(Int.) MxPS=   8 Ivl=10ms
I:* If#= 1 Alt= 0 #EPs= 1 Cls=03(HID  ) Sub=00 Prot=00 Driver=usbhid
E:  Ad=82(I) Atr=03(Int.) MxPS=   8 Ivl=10ms
```

- Dans /sys/bus/usb/devices/usb6/6.2

```
$ tree /sys/bus/usb/devices/usb6/6-2
/sys/bus/usb/devices/usb6/6-2
|-- 6-2:1.0
|   |-- 0003:1130:0202.0003
|   |   |-- driver -> ../../../../../../../bus/hid/drivers/generic-usb
...
$ cat /sys/bus/usb/devices/usb6/6-2/idVendor
1130
$ cat /sys/bus/usb/devices/usb6/6-2/idProduct
0202
```

Numérotation du périphérique

- « Adresse » sur le bus USB
- Définie par :
  - Root hub
  - Port
  - Configuration
  - Interface
- Exemple : `6-2:1.0`



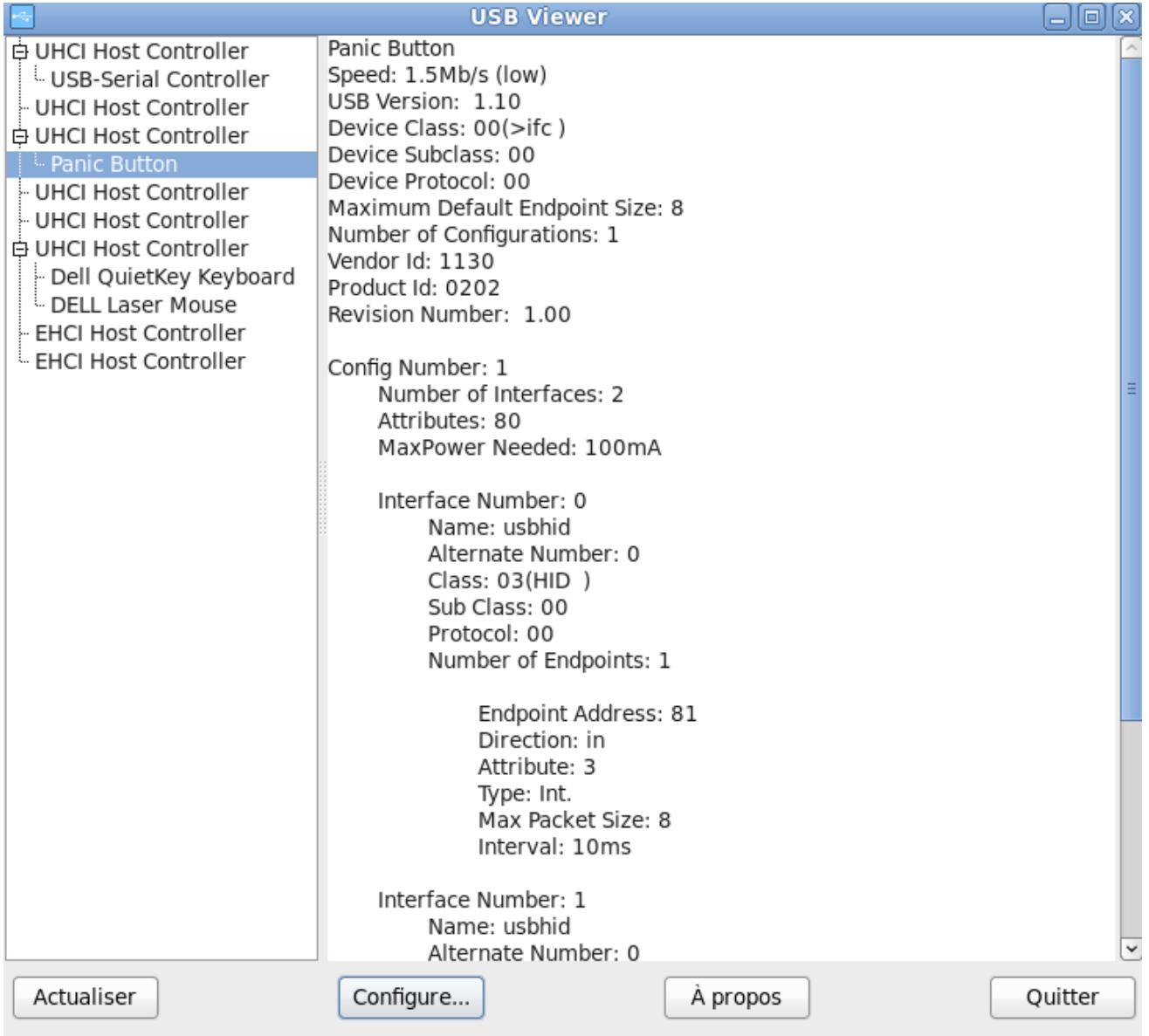
open wide INGENIERIE

Liste avec `lsusb`

- Affichage des données à partir de `/proc` et `/sys`
- Par défaut, affiche tous les périphériques

```
$ lsusb
Bus 008 Device 002: ID 067b:2303 Prolific Technology, Inc. PL2303 Serial Port
Bus 008 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 007 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 006 Device 002: ID 1130:0202 Tenx Technology, Inc.
Bus 006 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 005 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 004 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 003 Device 003: ID 0461:4d51 Primax Electronics, Ltd
Bus 003 Device 002: ID 413c:2106 Dell Computer Corp. Dell QuietKey Keyboard
Bus 003 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
Bus 002 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

Outil graphique `usbview`



Support USB sous Linux 1/2

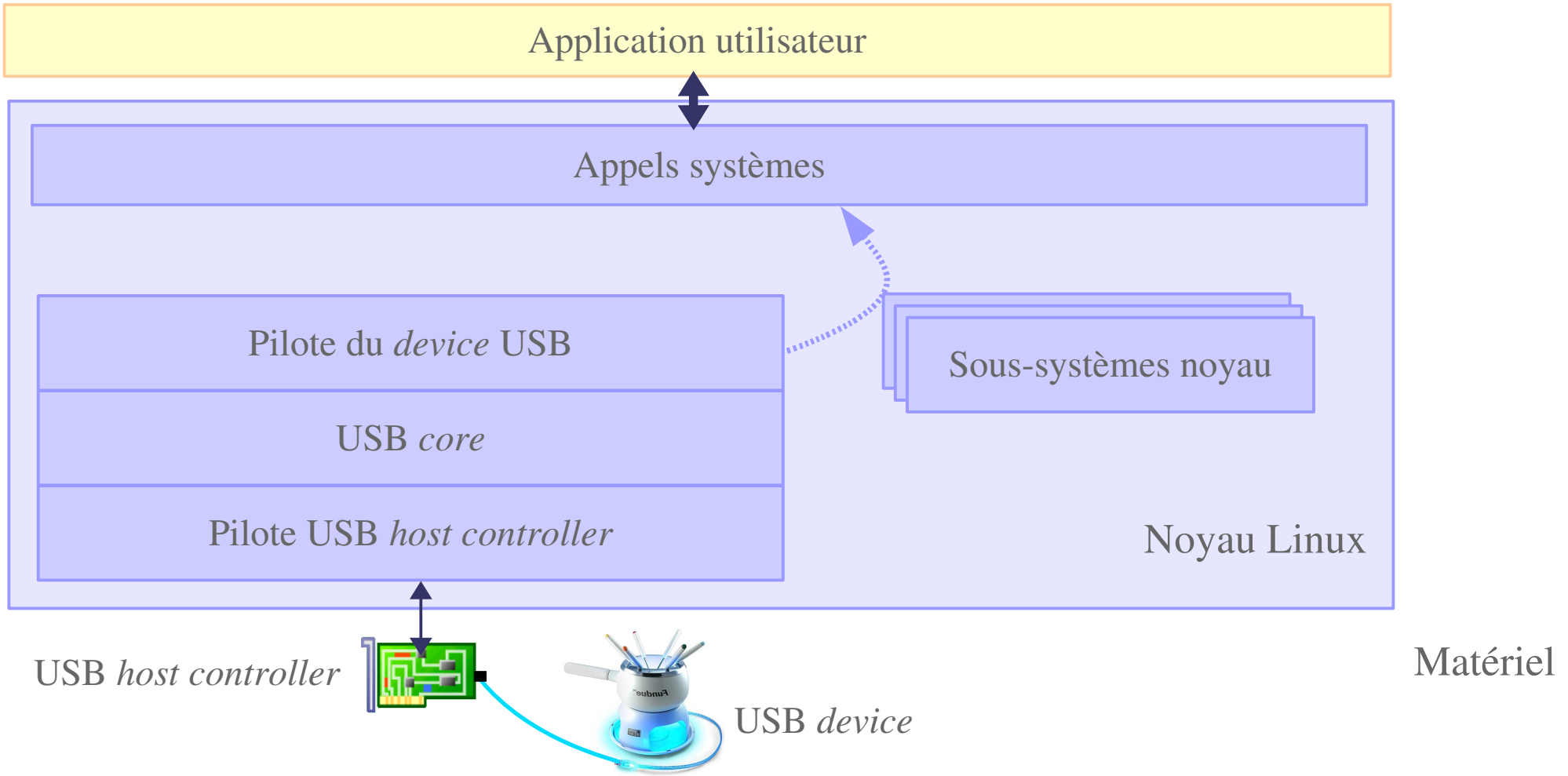


Schéma de M. Opdenacker

Support USB sous Linux 2/2

- Similitude avec le sous-système PCI
- idVendor / idProduct <=> VendorID / DeviceID
- Un pilote de périphérique USB est défini par :
  - Une liste de périphériques définie par des idVendor / idProduct
  - Une fonction `probe()` : appelée lors de l'insertion du périphérique
  - Une fonction `disconnect()` : appelée lors de la déconnexion du périphérique
  - Si nécessaire, une interface en mode caractère, bloc ou réseau
- REMARQUE: en PCI les fonction équivalentes sont `probe()` et `remove()`

Développement d'un pilote USB

- Périphériques répertoriés sur : http://www.linux-usb.org
- Problème de la disponibilité du protocole de communication (host / device)
- Principe: On enregistre le protocole (initialisation, pilotage) et on le reproduit sous Linux
  - Utilisation d'un espion logiciel sous Windows
    - USBTrace (http://www.sysnucleus.com)
    - USBlyzer (http://www.usblyzer.com)
  - Exécution du pilote Windows dans une machine virtuelle + utilisation *usbmon* sous Linux
  - Espion de protocole matériel (cher !)

Exemple du *panic button*

- Périphérique simple, en entrée seulement
- Utilise le *endpoint 0* de type CONTROL pour renvoyer l'état du bouton (nombre de clics)
- Par défaut, détecté par Linux comme un HID car déclaré comme *USB hid class* (03)
- Principe:
  - Détacher le périphérique du pilote HID → utilisation d'un script ou d'un programme en espace utilisateur
  - Le nouveau pilote peut alors le détecter
  - Cette procédure peut être automatisée par une règle UDEV

Pilote panicb: version 0

- Premier squelette du pilote (*panicb*)
  - La liste des devices de type `usb_device_id`
  - Structure `usb_driver`
  - Fonctions `probe()` et disconnect() « vides »
  - Enregistrer le pilote par `usb_register()` dans `module_init()`
  - Supprimer le pilote par `usb_deregister()` dans `module_exit()`
- REMARQUE: pas d'allocation de majeur/mineur pour l'instant

Définition de la liste usb_device_id

```
#include <linux/usb.h>
...

#define VENDOR_ID   0x1130
#define PRODUCT_ID  0x0202

static struct usb_device_id id_table [] = {
  { USB_DEVICE(VENDOR_ID, PRODUCT_ID) },
  { },
};

MODULE_DEVICE_TABLE (usb, id_table);
```

Structure usb_driver

```
static struct usb_driver panicb_driver = {
   .name        =  "panicb",
   .probe       =  panicb_probe,
   .disconnect  =  panicb_disconnect,
   .id_table    =  id_table,
};
```

Insertion: `panicb_probe()`

- Fonction appelée si le périphérique est détecté

```
static int panicb_probe (struct usb_interface
*interface, const struct usb_device_id *id)
{
  dev_info(&interface, "USB Panic Button device now
attached\n");
  return 0;
}
```

Déconnexion: `panicb_disconnect()`

- Appelée quand le périphérique est *retiré*
- On doit défaire ce qui est fait dans `probe()`

```
static void panicb_disconnect(struct usb_interface
*interface)
{
  dev_info(&interface->dev, "USB Panic Button now
disconnected\n");
}
```

module_init() et module_exit()

- Appel aux fonctions d'enregistrement et suppression dans la liste des pilotes USB

```
static int __init usb_panicb_init(void)
{
  return usb_register(&panicb_driver);
}

static void __exit usb_panicb_exit(void)
{
  usb_deregister(&panicb_driver);
}

module_init (usb_panicb_init);
module_exit (usb_panicb_exit);
```

Test du pilote version 0

- A l'insertion, le périphérique est « capté » par le pilote HID (classe = 03)
- Visible avec `/proc/bus/usb/devices`
- Avant insertion du module, il faut « détacher » le périphérique pour pouvoir exploiter le nouveau pilote en utilisant une entrée dans `/sys`

```
# echo -n 6-2:1.0 > /sys/bus/usb/drivers/usbhid/unbind
```

- La valeur de l'identifiant (ex: 6-2:1.0) dépend de la prise USB utilisée → on général, on utilise une règle UDEV pour automatiser (voir plus loin)

Pilote panicb: version 1

- On ajoute la structure définissant le périphérique

représentation du *device* dans le noyau

```
struct usb_panicb {
  struct usb_device * udev;
  unsigned int     button;   // button state
};
```

paramètre(s) « privé(s) »

Insertion: panicb_probe()

- On doit allouer les données privée (structure)

```
static int panicb_probe (struct usb_interface *interface, const
struct usb_device_id *id)
{
  struct usb_device *udev = interface_to_usbdev (interface);
  struct usb_panicb *panicb_dev;

  printk (KERN_INFO "panicb_probe: starting\n");
  ...
  panicb_dev = kmalloc (sizeof(struct usb_panicb), GFP_KERNEL);
  panicb_dev->udev = usb_get_dev(udev); // get device
  usb_set_intfdata (interface, panicb_dev); // Save private data

  dev_info(&interface->dev, "USB Panic Button device now
attached\n");
  return 0;
}
```

Déconnexion: panicb_disconnect()

- Libération des différentes structures

```
static void panicb_disconnect(struct usb_interface
*interface)
{
  struct usb_panicb *dev;

  dev = usb_get_intfdata (interface); // Get private data
  ...
  usb_put_dev(dev->udev); // release device structure
  kfree(dev);

  dev_info(&interface->dev, "USB Panic Button now
disconnected\n");
}
```

Pilote panicb: version 2

- Le pilote actuel ne fait pas grand chose...
- Importance des fonctions `usb_get/set_intfdata()`
  → pas de variable globale
- Étape suivante: gestion de l'entrée du pilote dans `/sys`, soit `/sys/bus/usb/drivers/panicb`
- On ajoute une entrée `button` au répertoire
- On peut alors piloter le périphérique par des E/S standards :

```
$ cat /sys/bus/usb/drivers/panicb/6-2:1.0/button
```

- Utilisation des fonctions `device_create_file()` et `device_remove_file()`

Ajout de l'entrée 1/2

- On utilise des macros `show()` et `set()`

```
static ssize_t show_button(struct device *dev, struct device_attribute
*attr, char *buf) {
  struct usb_interface *intf = to_usb_interface(dev);
  struct usb_panicb *panicb_dev = usb_get_intfdata(intf);

  get_panicb_button_status (panicb_dev); // to be done with endpoint 0

  return sprintf(buf, "%d\n", panicb_dev->button);
}

// Not used for panic button (read only !)
static ssize_t set_button(struct device *dev, struct device_attribute
*attr, const char *buf, size_t count) {
  return count;
}

static DEVICE_ATTR(button, S_IWUGO | S_IRUGO, show_button, set_button);
```

Ajout de l'entrée 2/2

- Dans `panicb_probe()` on ajoute

```
device_create_file(&interface->dev, &dev_attr_button);
```

- Dans `panicb_disconnect()` on ajoute :

```
device_remove_file(&interface->dev, &dev_attr_button);
```

- Il reste a écrire la communication bas-niveau (*endpoint*) avec le périphérique :
  - Méthode standard basée sur les URB (USB Request Block) → asynchrone
  - Méthode simplifiée sans utiliser d'URB → synchrone

Communication en mode synchrone

- Cas fréquent
- Mode synchrone → on attend la réponse du périphérique :
  - Ne pas appeler dans un contexte d'IT !
  - Annulation du message impossible
- Fonctions:
  - `usb_control_msg()`
  - `usb_bulk_msg()`
  - `...`
- Syntaxe très proche des versions en espace utilisateur avec *libusb*

Exemple: usb_control_msg()

```
int usb_control_msg (struct usb_device *dev, unsigned int
pipe, __u8 request, __u8 requesttype, __u16 value, __u16
index, void *data, __u16 size, int timeout);
```

dev: pointer to the usb device to send the message to

pipe: endpoint "pipe" to send the message to (IN/OUT)

request: USB message request value

requesttype: USB message request type value

value: USB message value

index: USB message index value

data: pointer to the data to send

size: length in bytes of the data to send

timeout: time to wait for the message to complete before timing out (if 0 the wait is forever)

Cas du Panic Button: get_panicb_button_status()

- Les paramètres (0x01, 0xA1, 0x300, 0x00) sont obtenus par espionnage du dialogue

```
  if (!(buf = kmalloc(8, GFP_KERNEL))) {
    printk(KERN_WARNING "panicb: can't alloc buf\n");
    return -1;
  }


  memset (buf, 0, 8);
  usb_control_msg (panicb_dev->udev, usb_rcvctrlpipe
(panicb_dev->udev, 0), 0x01, 0xA1, 0x300, 0x00, buf, 8, 2 *
HZ);
  panicb_dev->button = *buf;
  kfree (buf);
```

message sur 8 caractères

device vers *host*

endpoint 0

buffer de réception

nombre de clics = 1er octet

Pilote panicb: version 3

- L'étape suivante est d'ajouter une interface en mode caractère
- Par forcément utile pour ce périphérique mais « pédagogique »
  - Ajout d'une `struct file_operations`
  - Ajout des fonctions `open()`, `release()`, `ioctl()`, [`read()`, `write()`]
  - L 'entrée dans `/dev` est créée automatiquement (majeur = 180)
  - On utilise le type `struct usb_class_driver`

Déclaration des structures

- Structure `file_operations` (pilote en mode *char*)

```
static struct file_operations panicb_fops = {
  .open    = panicb_open,
  .release = panicb_release,
  .ioctl   = panicb_ioctl
};
```

- Classe du pilote USB

```
static struct usb_class_driver panicb_class_driver = {
  .name = "usb/panicb%d", // => /dev/panicb0
  .fops = &panicb_fops,
  .minor_base = 0
};
```

Fonctions d'enregistrement du pilote caractère

- Dans `panicb_probe()`

```
usb_register_dev(interface, &panicb_class_driver);
```

- Dans `panicb_disconnect()`

```
usb_deregister_dev (interface, &panicb_class_driver);
```

Fonction open( )

- On doit sauver les données de la structure privée du périphérique dans la structure `file`

```
minor = iminor(inode);

// Get interface for device
interface = usb_find_interface (&panicb_driver, minor);

// Get private data from interface
dev = usb_get_intfdata (interface);

// Save private data to file structure (VFS)
file->private_data = (void *)dev;
```

Fonction ioctl()

- On utilise une fonction `ioctl()` et non pas `read()` → mieux adapté
- Principe: lire l'état du bouton et le retourner à l'espace utilisateur → `copy_to_user()`

```
// get the dev object from VFS
dev = file->private_data;


switch (cmd) {
  case 0 :
    if (get_panicb_button_status (dev) == 0) {
      if (copy_to_user((void*)arg, &(dev->button),
sizeof(int))) {
        printk (KERN_WARNING "panicb: copy_to_user
error\n");
        return -EFAULT;
    }
```

Test du pilote « final »

- L'entrée dans `/sys` dépend de la prise USB utilisée → Utilisation d'un script pour extraire l'identifiant à partir du répertoire :

```
ID=$(ls /sys/bus/usb/drivers/panicb | cut -d" " -f1)
```

- Utilisation de l'entrée `button`

```
# cat /sys/bus/usb/drivers/panicb/$ID/button
2
```

- Utilisation d'un appel `ioctl()` dans un programme

```
# ./panicb_test /dev/panicb0
Button= 2
```

- Voir l'exemple `alert.sh :-)`

Utilisation de UDEV 1/3

- Gestion dynamique du contenu de `/dev` (remplace la tentative `devfs` de 2.4)
- Détection d'un périphérique → ajout d'une entrée dans `/dev`
- Démon `udevd` en espace utilisateur (basé sur `/sys`), utilise `hotplug` en espace noyau
- Configuration dans `/etc/udev` :
  - `udev.conf` : configuration générale
  - `rules.d` => répertoire contenant les fichiers de règles (`.rules`)
- Voir http://reactivated.net/writing_udev_rules.html

Utilisation de UDEV 2/3

- Le pilote seul ne peut gérer l'attachement/détachement du périphérique !
- Principes de la règle :
  - Détecter le périphérique par VendorID, ProductID
  - Détacher le périphérique du pilote HID (usbhid) → unbind
  - Attacher le périphérique au pilote dédié (panicb) → bind

Utilisation de UDEV 3/3

```
ATTRS{idVendor}=="1130",
ATTRS{idProduct}=="0202",
```

Vendor ID

Device ID

script exécuté lors de la détection

```
PROGRAM="/bin/sh -c 'echo -n $id:1.0
>/sys/bus/usb/drivers/usbhid/unbind;
```

contient root_hub-port

```
echo -n $id:1.0 >
/sys/bus/usb/drivers/panicb/bind'"
```

- Les champs sont séparés par des virgules !
- Mode trace udevd: `udev_log="info"` dans `/etc/udev/udev.conf`
- Pour relire les règles:

```
# udevadm control --reload-rules
```

USB Request Blocks

- Méthode générale
- La communication entre *host* et *device* est faite de manière asynchrone par des URBs
- Similaires aux paquets d'une communication réseau/socket
- Chaque *endpoint* gère une « file » d'URBs
- L'URB a un fonction *handler* de fin de traitement
- Un pilote peut allouer plusieurs URBs et les utiliser pour plusieurs *endpoints*
- Voir `Documentation/usb/URB.txt` dans les sources du noyau

URBan life

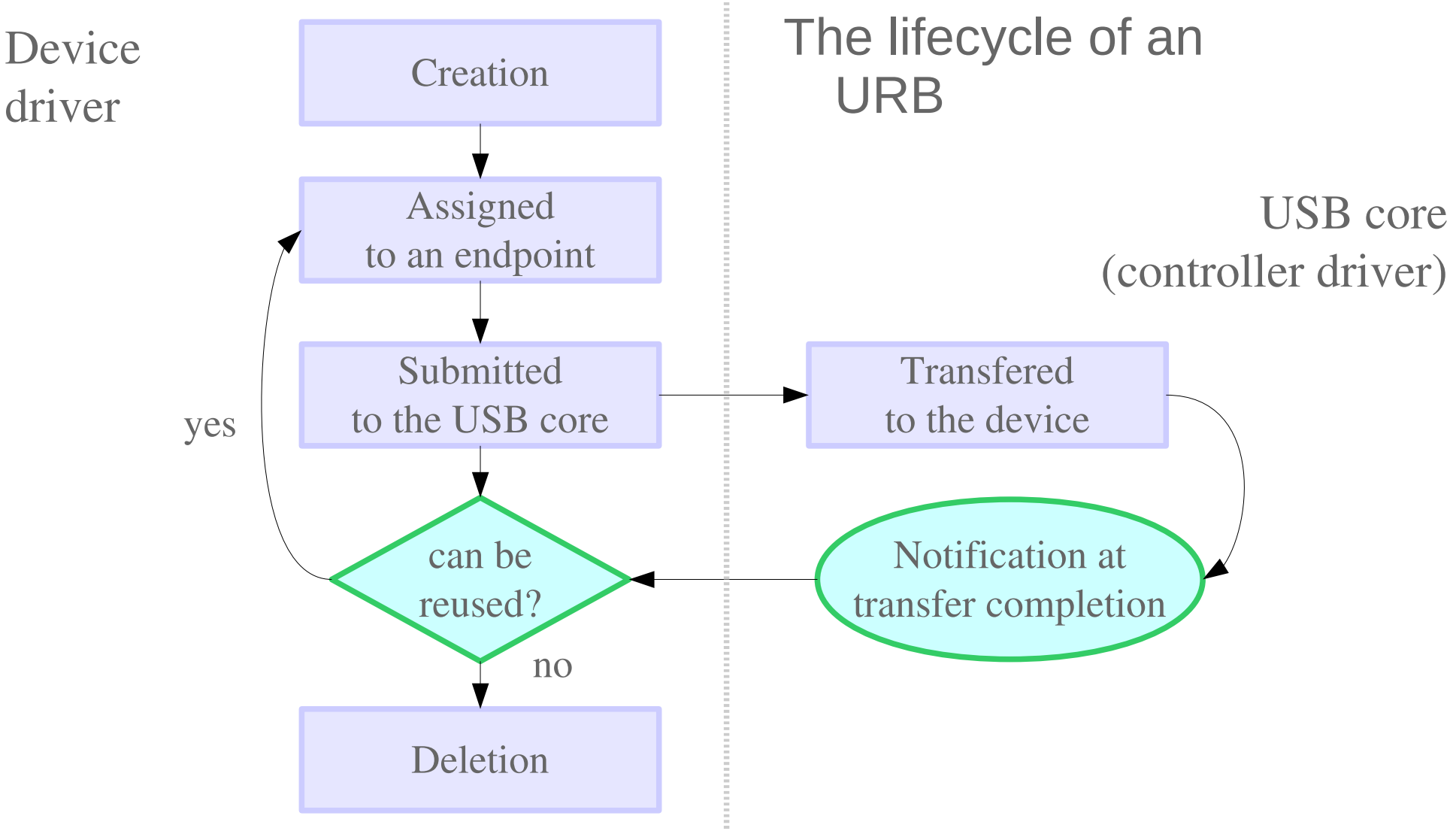


Schéma de M. Opdenacker

USB Request Blocks - utilisation

- Utilise le type `struct urb`
- Création avec la fonction `usb_alloc_urb()`
- Remplissage `usb_fill_control_urb()`
- Soumission par `usb_submit_urb()`
- Détruit avec `usb_free_urb()`
- Ne doit PAS être alloué statiquement, ni avec `kmalloc()`
- Utilisation complexe, à utiliser pour un périphérique évolué
- Voir exemple plus loin (version 4 du pilote panic button)

Pilote panicb, version 4 (URB)

- Plus complexe
- Pas de problème pour la méthode `write()`
- Asynchrone → utilisation d'une *workqueue* pour la synchronisation
- Plus complexe pour la méthode `read()` ou `ioctl()`
  - Construction + soumission de l'URB
  - Blocage du processus dans la méthode `read()` par `wait_event_interruptible()`
  - Déblocage par `wake_up_interruptible()` dans le handler de fin d'exécution de l'URB

open wide INGENIERIE

Mise au point avec usbmon

- Entrée dans debugfs pour tracer les échanges USB

```
# modprobe usbmon
# mount -t debugfs debugfs /sys/kernel/debug
```

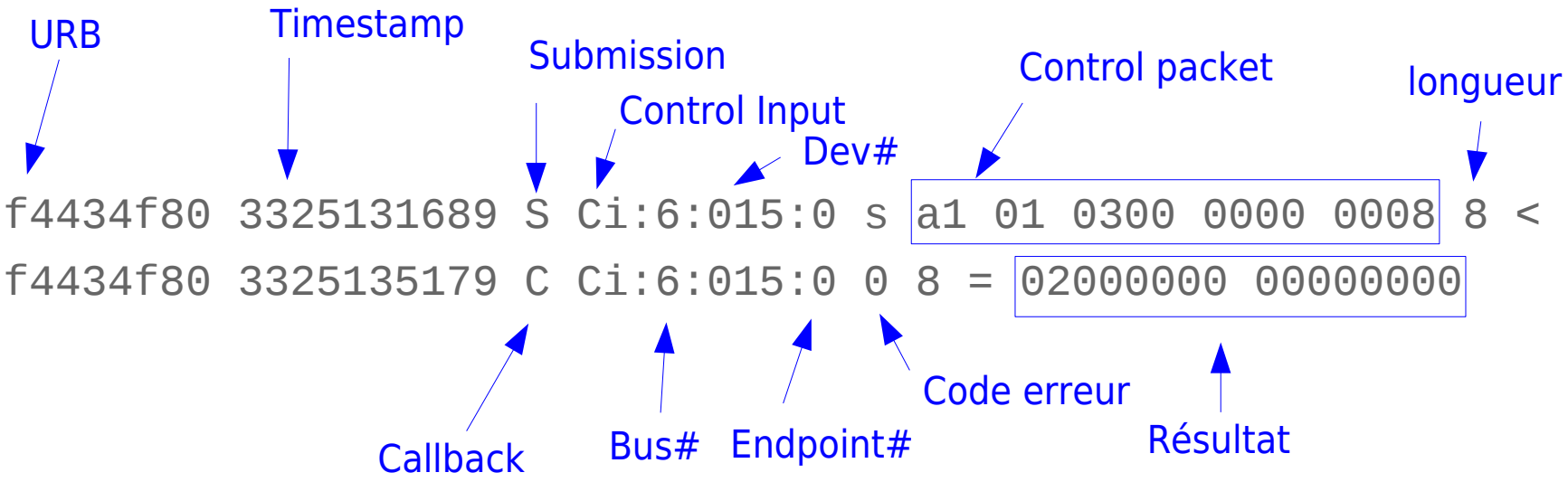
- Voir `Documentation/usb/usbmon.txt`
- L'entrée dépend du numéro de bus utilisé
- Exemple: Panic Button connecté au bus 6, voir `/proc/bus/usb/devices`

```
T:  Bus=06 Lev=01 Prnt=01 Port=00 Cnt=01 Dev#= 15 Spd=1.5 MxCh= 0
```

- Ouverture du fichier de trace

```
# cat /sys/kernel/debug/usb/usbmon/6u
f4434f80 3325131689 S Ci:6:015:0 s a1 01 0300 0000 0008 8 <
f4434f80 3325135179 C Ci:6:015:0 0 8 = 02000000 00000000
```

Trace usbmon pour URB « Control »



Utilisation de *libusb*: pilotes en espace utilisateur

- Écriture d'un programme en espace utilisateur et non plus un module noyau
- Plus simple d'accès, mais ne peut pas fournir de véritable entrée dans `/dev`
- Souvent utilisable en remplacement d'un pilote noyau
- Syntaxe très proche de celle l'espace noyau
- Deux versions :
  - 0.1 (legacy) : Uniquement synchrone
  - 1.0 : synchrone / asynchrone
  - Voir l'exemple du Panic Button
- Voir http://www.libusb.org

Bibliographie

- http://www.kernel.org/doc/htmldocs/usb.html
- http://www.usbmadesimple.co.uk/
- http://lwn.net/images/pdf/LDD3/ch13.pdf
- http://www.linuxjournal.com/article/7353