



UNIVERSITÉ
CÔTE D'AZUR

Pilote matériel et Protocole USB



Présentation: Stéphane Lavirotte
Auteurs: ... et al*

(*) Cours réalisé grâce aux documents de :
Bootlin

Mail: Stephane.Lavirotte@unice.fr
Web: <http://stephane.lavirotte.com/>
Université Côte d'Azur



Rappel sur les Pilotes Noyau / Espace Utilisateur



Pilotes dans l'espace noyau

- ✓ **Une fois que les sources sont acceptées dans la branche principale de développement:**
 - Elles sont maintenues par ceux qui font des changements
 - Maintenance des sources à coût zéro
 - Bénéficie des améliorations du noyau
 - Bénéficie des patches de sécurité
 - Accès facilité aux sources par les utilisateurs
 - De nombreux développeurs peuvent faire des améliorations à votre code



Pilotes dans l'espace utilisateur

- ✓ Dans certains cas, il est possible d'implémenter un pilote dans l'espace utilisateur
- ✓ Peut être utilisé quand:
 - Le noyau fournit un mécanisme qui permet aux applications d'avoir un accès direct au matériel
 - Il n'est pas nécessaire d'exploiter un sous-système du noyau tel que la pile réseau ou les systèmes de fichiers
 - Il n'est pas nécessaire que le noyau serve de multiplexeur pour le dispositif: une seule application accède au dispositif
- ✓ Possibilités pour les pilotes de périphériques de l'espace utilisateur:
 - USB avec libusb: <https://libusb.info/>
 - SPI avec spidev: Documentation/spi/spidev
 - I2C avec i2cdev: Documentation/i2c/dev-interface
 - Dispositifs mappés en mémoire avec UIO, y compris la gestion des interruptions: Documentation/DocBook/uio-howto/
- ✓ Certaines classes de dispositifs (imprimantes, scanners, accélération graphique 2D/3D) sont typiquement gérés à la fois par l'espace noyau et l'espace utilisateur



Pilotes dans l'espace utilisateur

✓ Avantages:

- Pas besoin de compétences de développement noyau
- Facile de partager du code pour la gestion des dispositifs
- Les pilotes peuvent être écrits en n'importe quel langage
- Les pilotes peuvent rester propriétaires
- Les pilotes peuvent être tués et débogués plus facilement, sans risque de plantage du noyau
- Les pilotes peuvent être swappés (par le noyau, pas les modules)
- Les pilotes peuvent faire du calcul flottant
- Moins de code et de complexité dans le noyau
- Certains fois de meilleures performances (pas d'appels systèmes)

✓ Inconvénients:

- Moins simple pour gérer les interruptions
- Augmentation de la latence d'interruption par rapport au code du noyau



Exemple du Bus USB

Ou comment écrire un pilote pour un
périphérique USB

USB et Commandes de l'espace utilisateur

✓ Lister les périphériques USB disponibles sur votre système

```
$ lsusb
```

```
Bus 001 Device 001: ID 1d6b:0002 Linux Foundation 2.0 root hub
```

```
Bus 002 Device 003: ID 06c2:0045 Phidgets Inc. (formerly GLAB)  
PhidgetInterface Kit 8-8-8
```

```
Bus 002 Device 002: ID 80ee:0021 VirtualBox USB Tablet
```

```
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

✓ Lister des caractéristiques du périphérique USB

```
$ lsusb -v
```

- Voir résultat et détail des informations plus loin



Descripteur USB

✓ Indépendant du système d'exploitation (norme USB)

✓ Device

– le dispositif connecté au bus

✓ Configurations

– l'état du dispositif

✓ Interfaces

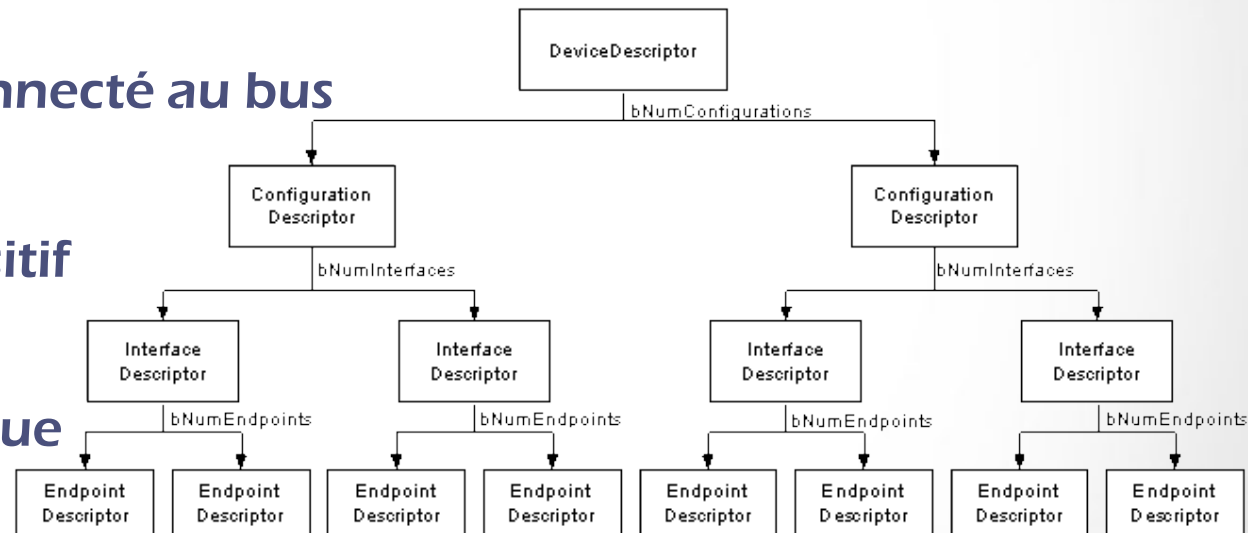
– Dispositif logique

✓ Endpoint

– Tube de communication unidirectionnel

– 2 types

- IN: dispositif vers ordinateur
- OUT: ordinateur vers dispositif



Cf: <http://www.beyondlogic.org/usbnutshell/usb5.shtml>



Exemple lsusb -v

```
Bus 002 Device 004: ID 06c2:0045 Phidgets Inc. (formerly GLAB)
PhidgetInterface Kit 8-8-8
```

Device Descriptor :

bLength	18	
bDescriptorType	1	
bcdUSB	1.10	
bDeviceClass	0	(Defined at Interface level)
bDeviceSubClass	0	
bDeviceProtocol	0	
bMaxPacketSize0	8	
idVendor	0x06c2	Phidgets Inc. (formerly GLAB)
idProduct	0x0045	PhidgetInterface Kit 8-8-8
bcdDevice	9.04	
iManufacturer	1	Phidgets Inc.
iProduct	2	PhidgetInterfaceKit
iSerial	3	273756
bNumConfigurations	1	

Configuration Descriptor : ...



Exemple lsusb -v (suite)

Configuration Descriptor :

bLength	9
bDescriptorType	2
wTotalLength	34
bNumInterfaces	1
bConfigurationValue	1
iConfiguration	0
bmAttributes	0x80
(Bus Powered)	
MaxPower	500mA

Interface Descriptor :

bLength	9
bDescriptorType	4
bInterfaceNumber	0
bAlternateSetting	0
bNumEndpoints	1
bInterfaceClass	3 Human Interface Device
bInterfaceSubClass	0 No Subclass
bInterfaceProtocol	0 None
iInterface	0

HID Device Descriptor: ...



Exemple lsusb -v (fin)

Endpoint Descriptor :

bLength	7	
bDescriptorType	5	
bEndpointAddress	0x81	EP 1 IN
bmAttributes	3	
Transfer Type		Interrupt
Synch Type		None
Usage Type		Data
wMaxPacketSize	0x0040	1x 64 bytes
bInterval	8	

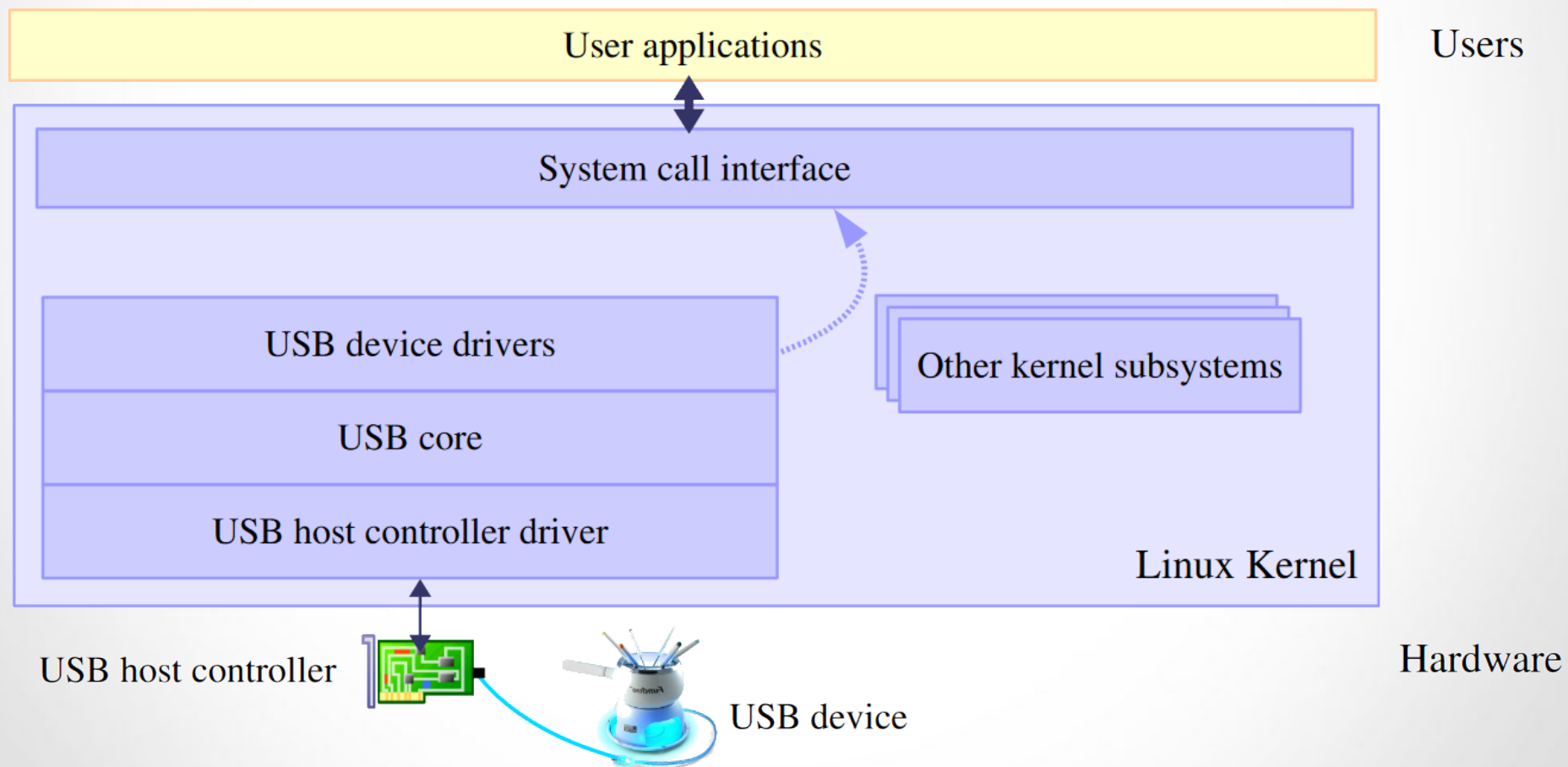
Device Status: 0x0000
(Bus Powered)



Identifiants de dispositif USB

- ✓ **Chaque dispositif USB dispose d'identifiants**
 - **Identifiant vendeur (`idVendor`)**
 - Fournit par USB.org: <http://www.usb.org/>
 - Ex: « Phidget.com »: `idVendor=0x06c2`
 - **Identifiant produit (`idProduct`):**
 - Définit par le fabricant
 - Ex: « Phidget Interface Kit 8/8/8 »: `idProduct=0x0045`
 - Attention, ce n'est pas un code unique par dispositif, mais un code par type de dispositif
 - **Il peut y avoir en plus de cela:**
 - **Un numéro de version du produit (`bcdDevice`)**
 - Device Release Number (défini par le fabricant)
 - **Un numéro de série du produit (`iSerial`) unique par produit**

Vue d'ensemble du fonctionnement USB pour le noyau Linux



Source Bootlin: <https://bootlin.com/>



Infrastructure Linux pour USB

- ✓ **Une infrastructure principale (pilote du bus)**
 - Disponible via le module `usb-core`
 - `struct bus_type` définie dans `drivers/usb/core`
- ✓ **Pilote d'interface (HCD: Host Control Device)**
 - Pour OHCI, UHCI, EHCI, XHCI
 - OHCI = USB 1.0 et 1.1
 - UHCI = USB 1.0 et 1.1
 - EHCI = USB 2.0 (compatible 1.0 et 1.1)
 - xHCI = USB 3.0 (compatible 1.0, 1.1 et 2.0)
 - Disponible via les modules `ehci_*`, `ohci_*`, ...
 - `drivers/usb/host`
- ✓ **Pilotes de périphériques**
 - Partout dans le noyau avec classification par leur type

USB 1.0
LowSpeed: jusqu'à 1.5 Mbps
USB 1.1
FullSpeed: jusqu'à 12 Mbps
USB 2.0
HiSpeed: jusqu'à 480 Mbps
USB 3.0
SuperSpeed: jusqu'à 5Gbps
USB 3.1
jusqu'à 10Gbps

Déclaration des périphériques supportés par le pilote

- ✓ `USB_DEVICE (vendor, product)`
 - **Crée une structure `usb_device_id` qui peut être utilisée pour identifier seulement les identifiants `vendor` et `product`.**
 - **Utilisé par la plupart des pilotes non standards**
- ✓ `USB_DEVICE_VER (vendor, product, lo, hi)`
 - **Identique, mais uniquement pour un intervalle de version**
 - **Seulement utilisé 11 fois depuis Linux 2.6.18**
- ✓ **Les deux macros suivantes sont seulement utilisées pour l'implémentation de classe de périphériques et interfaces**
 - `USB_DEVICE_INFO (class, subclass, protocol)`
 - **Identifie une classe spécifique de périphériques USB**
 - `USB_INTERFACE_INFO (class, subclass, protocol)`
 - **Identifie une sous-classe d'interfaces USB**

Déclaration des identifiants gérés par le pilote

- ✓ **USB Core doit savoir quels périphériques sont gérés par votre module**
 - Utilisation d'une table qui utilise les identifiants `vendor`, `product`
- ✓ **La macro `MODULE_DEVICE_TABLE()`**
 - Permet à `depmod` d'extraire à la compilation les relations entre identifiant de dispositif et pilote
 - Ainsi peut être automatiquement chargé par `udev`

```
#define VENDOR_ID 0x06c2
#define PRODUCT_ID 0x0045
static struct usb_device_id skel_table[] = {
    { USB_DEVICE(VENDOR_ID, PRODUCT_ID) },
    [...]
    { } // Termine la structure
};
MODULE_DEVICE_TABLE(usb, skel_table);
```


Instanciation d'une structure `usb_driver`

- ✓ **Structure `usb_driver` définie dans USB core**
 - **Hérite de `device_driver` qui définit un modèle de dispositif**
- ```
static struct usb_driver skel_driver = {
 .name = "skel_driver",
 .id_table = skel_table, // définit l'association id - driver
 .probe = skel_probe, // fonction appelée à la connexion
 .disconnect = skel_disconnect, // à la déconnexion
 .suspend = skel_suspend, // fonction appelée à la mise en veille
 .resume = skel_resume // au retour de veille
}
```
- ✓ **Chaque pilote usb doit créer une structure `usb_driver` et l'enregistrer auprès de USB core:**
    - `usb_register(struct usb_driver *)`
    - `usb_deregister(struct usb_driver *)`
  - ✓ **Mais la macro `module_usb_driver(struct usb_driver)`**
    - **Remplace module `module_init` et `module_exit`**
    - **Appel automatique de `usb_register` et `usb_deregister`**



# Méthode probe

- ✓ **Appelée lors de la connexion du dispositif au bus**

```
static int skel_probe(struct usb_interface *iface,
 const struct usb_device_id *id) {
 ...
}
```

- ✓ **Cette fonction est responsable :**
  - De l'initialisation des structures nécessaires pour le dispositif
  - De récupérer les informations de description de celui-ci
  - D'enregistrer le dispositif auprès de la plateforme pour accès via le fichier dans `/dev`
- ✓ **Cette fonction retourne:**
  - 0 si tout s'est bien passé
  - Une valeur négative sinon



# Méthode disconnect

## ✓ Appelée lors du retrait du dispositif du bus

```
static void skel_disconnect(struct usb_interface *interface)
{
 // libérer toutes les données allouées
}
```

## ✓ Cette fonction est responsable

- De désabonner le dispositif de la plateforme
- De libérer les données allouées dynamiquement

# La structure `usb_interface`

✓ C'est la structure passée par USB core aux pilotes

✓ Structure composée de:


- `struct usb_host_interface *altsetting;`
  - **Liste des paramètres alternatifs qui peuvent être sélectionnés**
  - **Permet d'accéder au** `usb_endpoint_descriptor`
    - `interface->altsetting[i]->endpoint[j]->desc`
- `unsigned int num_altsetting;`
  - **Nombre de paramètres alternatifs**
- `struct usb_host_interface *cur_altsetting;`
  - **Le paramètre actuellement actif**
- `int minor;`
  - **Le numéro mineur auquel l'interface est connectée**
  - **Utile pour les pilote utilisant** `usb_register_dev()`



# Association à une entrée de /dev

## ✓ Structure définissant le fichier associé dans /dev

```
static struct usb_class_driver skel_class = {
 .name = "skel%d",
 .fops = &skel_fops,
 .minor_base = USB_SKEL_MINOR_BASE
};
```

 c'est un file\_operations !

## ✓ Dans la méthode probe

```
result = usb_register_dev(iface, &skel_class);
```

## ✓ Dans la méthode disconnect

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



# usb\_set\_intfdata() usb\_get\_intfdata()

- ✓ `static inline void usb_set_intfdata(struct usb_interface *intf, void *data);`
  - **Fonction utilisée dans `probe()` pour attacher des données sur le périphérique à une interface**
  - **N'importe quel type de pointeur peut être utilisé**
  - **Utile pour stocker n'importe quelle information pour chaque dispositif supporté par le pilote, sans avoir à conserver un tableau statique**
- ✓ `void * usb_get_intfdata(struct usb_interface *intf)`
  - **Fonction typiquement utilisée dans la fonction `open()` pour récupérer les données associées au périphérique**
  - **Les données stockées doivent être libérées au `disconnect()` et en plus appeler `usb_set_intfdata(interface, NULL);`**
- ✓ **Nombreux exemples disponibles dans les sources du noyau**



# Association à une entrée de /dev

## ✓ Dans la méthode probe

```
struct skel_device *dev; // données de votre
device
```

→ c'est un file\_operations !

```
dev = kzalloc(sizeof(struct skel_device),
GFP_KERNEL);
usb_set_intfdata(iface, dev);
```

## ✓ Dans la méthode disconnect

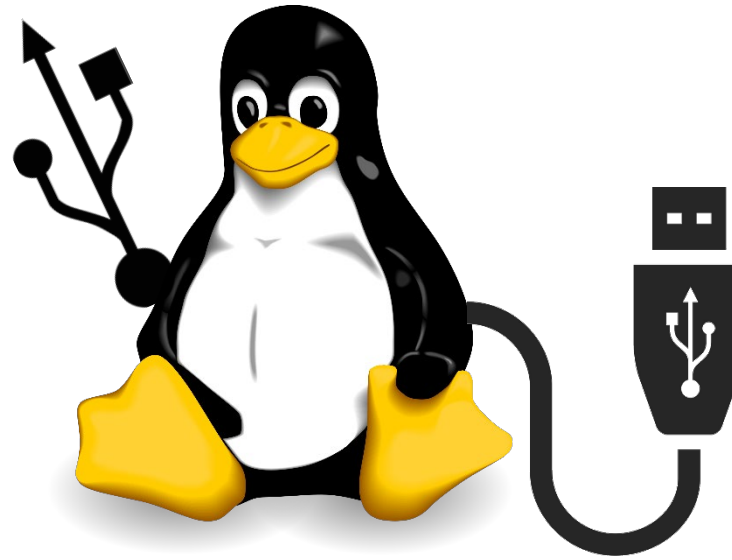
```
struct skel_device *dev;

dev = usb_get_intfdata(interface);
usb_set_intfdata(interface, NULL);
```

# Pour aller plus loin dans la réalisation du pilote USB

- ✓ **Maintenant que nous pouvons identifier un dispositif et appeler un driver correspondant**
  - On peut recevoir et envoyer des informations au dispositif
  
- ✓ **Mais le faire fonctionner, c'est une autre histoire...**
  - Il faut connaître la spécification des données reçues et à envoyer
  - La mise en œuvre de la communication est asynchrone
    - Utilisation de URB (USB Request Blocks)





# Communication USB et Linux

Endpoint et URB  
(USB Request Blocks)



# USB endpoints

## ✓ Control endpoints

- Contrôler le dispositif
- Avoir de l'information sur le dispositif
- Envoyer des commandes
- Tout dispositif à un control endpoint (endpoint 0)

## ✓ Interrupt endpoints

- Transfert peu de données à intervalle fixe
- Bande passante réservée
- Nécessite un polling constant de l'hôte

## ✓ Bulk endpoints

- Gros transferts sporadiques
- Utilisation de la bande passante restante (pas de garantie sur le délai et la bande passante, mais pas de perte de données)

## ✓ Isochronous endpoints

- Utilisé pour les gros transferts
- Vitesse garantie (pas de garantie que toutes les données le permette)

# La structure `usb_endpoint_descriptor`

- ✓ Contient toutes les données spécifiques au dispositif USB
- ✓ Informations utiles pour écrire un driver:
  - `bEndpointAddress`:
    - Adresse USB du endpoint
    - Encode aussi le sens (masque `USB_ENDPOINT_DIR_MASK` pour dire si c'est un `USB_DIR_IN` ou `USB_DIR_OUT`)
  - `bmAttributes`:
    - Type de endpoint (Control, Interrupt, Bulk, Isochronous)
    - Encodé avec un masque `USB_ENDPOINT_XFERTYPE_MASK` pour dire si c'est un `USB_ENDPOINT_XFER_*`)
  - `wMaxPacketSize`:
    - Taille maximum du paquet que le endpoint peut gérer
  - `bInterval`:
    - Pour les endpoints Interrupt, intervalle de polling (en ms pour USB 1.x et 2.x ou 1/8 de ms pour USB Hi-speed)



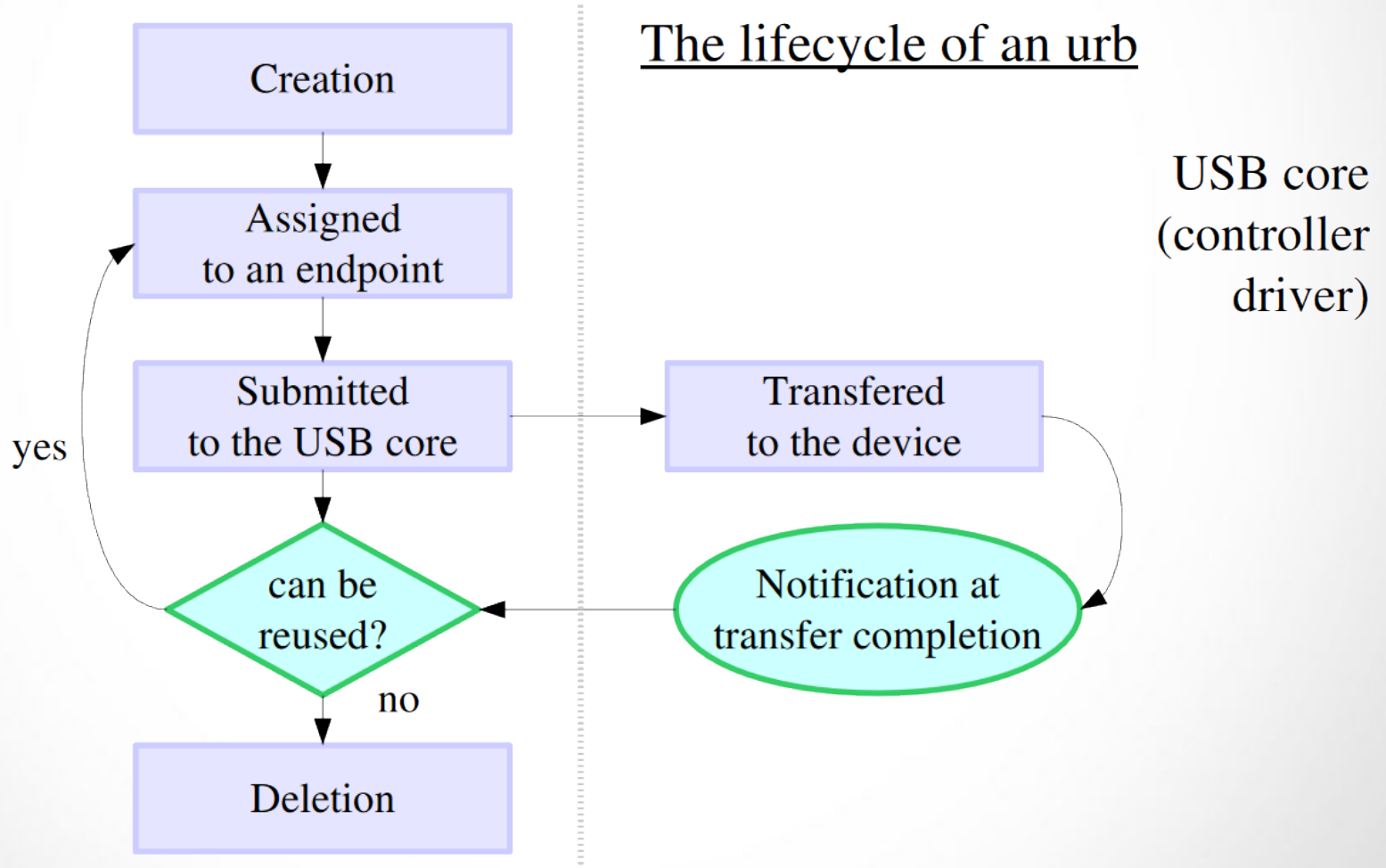
# URB: USB Request Blocks

- ✓ **Toute communication entre *Host* et *Device* est réalisée de manière asynchrone**
  - Similaire à la communication par paquets sur un réseau
  - Chaque endpoint peut accueillir une file de URBs
  - Chaque URB dispose d'un gestionnaire (*handler*)
  - Un pilote peut allouer plusieurs URBs pour un seul endpoint, ou réutiliser un même URB pour différents endpoints
  - **Voir la documentation :** `Documentation/usb/URB.txt`



# Cycle de Vie d'un URB

Device  
driver



Source Bootlin: <https://bootlin.com/>



## ✓ Champs importants pour les pilotes USB:

- `struct usb_device *dev;`
  - **Dispositif vers lequel le URB est envoyé.**
- `unsigned int pipe;`
  - **Information sur le endpoint du dispositif cible**
- `int status;`
  - **Statut du transfert**
- `unsigned int transfer_flags;`
  - **Instruction pour gérer le URB**
- `void * transfer_buffer;`
  - **Buffer pour transférer les donnée (doit être alloué par `kmalloc()` !)**
- `int transfer_buffer_length;`
  - **Taille du buffer de transfert**
- `int actual_length;`
  - **Longueur actuelle des données transférées ou reçues via le `urb`**
- `usb_complete_t complete;`
  - **Gestionnaire de complétion appelé quand le transfert est complet**



# struct urb (suite)

## ✓ Champs importants pour les pilotes USB (suite):

- void \* context;
  - **Blob de données** qui peut être utilisé dans le gestionnaire de complétion
- unsigned char \* setup\_packet; **(URB control)**
  - **Paquet de configuration** transféré avant les données dans le buffer de transfert
- dma\_addr\_t setup\_dma; **(urb control)**
  - **Idem, mais quand le paquet de configuration est transféré en mode DMA**
- int interval; **(URBs Isochronous et Interrupt)**
  - **Intervalle de polling du urb**
- int error\_count; **(URB Isochronous)**
  - **Nombre de transfert isochronous en erreur**
- int start\_frame; **(URB Isochronous)**
  - **Affecte ou retourne le numéro initial de frame à utiliser**
- int number\_of\_packets; **(URB Isochronous)**
  - **Nombre de buffer de transferr isochonous à utiliser**
- struct usb\_iso\_packet\_descriptor **(URB isochronous)**  
iso\_frame\_desc[0];
  - **Permettre à un seul URB de définir de multiples transfert isochonous en une fois**



# Création et destruction de URBs

- ✓ **Les structures `urb` doivent toujours être allouées par la fonction `usb_alloc_urbs()`**

- Ceci est nécessaire pour compter les références par USB core

```
#include <linux/usb.h>
struct urb * usb_alloc_urb(
 int iso_packets, // nb of isochonous packets the urb should contain.
 // 0 for other transfert types
 gfp_t mem_flags); // standard kmalloc() flags
```

- ✓ **Vérifier que la valeur de retour n'est pas `NULL`!**

- ✓ **Exemple typique:**

- `urb = usb_alloc_urb(0, GFP_KERNEL);`

- ✓ **De manière symétrique, une fonction permet de libérer un `urb`**

- `void usb_free_urb(struct urb * urb);`





# Initialisation du URB Interruption

```
void usb_fill_int_urb (
 struct urb *urb, // urb to be initialized
 struct usb_device *dev, // device to send the urb to
 unsigned int pipe, // pipe (endpoint and device
 specific)
 void *transfer_buffer, // transfer buffer
 int buffer_length, // transfer buffer size
 usb_complete_t complete, // completion handler
 void *context, // context (for handler)
 int interval // scheduling interval
);
```

- ✓ **Ceci ne nous empêche pas de faire des modifications dans le URB avant soumission**
- ✓ **Le champ `transfer_flags` doit être affecté par le pilote**

# Initialisation des autres types de URB

- ✓ `void usb_fill_bulk_urb(...);`
  - **Identique à** `usb_fill_int_urb`, **sans le paramètre** `interval`
- ✓ `void usb_fill_control_urb();`
  - **Identique à** `usb_fill_bulk_urb`, **avec un paramètre** `setup_packet` **supplémentaire**
  - **De nombreux pilotes utilisent plutôt** `usb_control_msg()`
- ✓ **Pour un URB Isochronous, il n'y a pas de fonction d'aide**
  - **Doit être réalisée manuellement**
  - **Voir un exemple dans le fichier:**
    - `drivers/media/video/usbvideo/usbvideo.c`



# Soumission d'un URB

## ✓ Après création et initialisation d'un URB

```
int usb_submit_urb(
 struct urb *urb // packet to submit
 int mem_flags // kmalloc flags !
);
```

## ✓ `mem_flags` est utilisé pour les allocations internes qui doivent être réalisées

- `GFP_ATOMIC`: appelé depuis du code ne pouvant faire de pause: un URB gestionnaire de complétion, une interruption hard ou soft. Ou appelé quand l'appelant contient un spinlock
- `GFP_NOIO`: dans certains cas quand le stockage block est utilisé
- `GFP_KERNEL`: pour les autres cas

# Les valeurs de retour de `usb_submit_urb()`

- ✓ `usb_submit_urb()` **retourne immédiatement:**
  - **0: le URB est ajouté à la file**
  - `-ENOMEM` : **Pas assez de mémoire**
  - `-ENODEV` : **Dispositif déconnecté**
  - `-EPIPE` : **Endpoint bloqué**
  - `-EAGAIN` : **trop de transferts ajoutés dans la queue**
  - `-EFBIG` : **trop de requêtes**
  - `-EINVAL` : **intervalle INT invalide. Plus d'un paquet pour INT**



# Handler de Completion

- ✓ Le handler completion dans le cas d'un URB Interrupt et seulement dans 3 situations
- ✓ Vérifier la valeur d'erreur dans le champ `urb->status`
  - Après que la donnée soit complètement transférée
    - `urb->status == 0`
  - Un (Des) erreur(s) est(sont) intervenue(s) durant le transfert
  - Le URB a été déconnecté par USB core
  - Pour ces deux derniers cas, voir les codes d'erreurs dans la **documentation**: `Documentation/usb/errorcodes.txt`
- ✓ `urb->status`
  - doit seulement être vérifié dans le handler de completion



# Completion handler

## ✓ **Prototype:**

```
void (*usb_complete_t) (
 struct urb *,
 // The completed urb
 struct pt_regs *
 // Register values at the time
 // of the corresponding interrupt (if any)
);
```

- ✓ **Rappel: vous êtes dans un contexte URB Interrupt:**
  - Ne pas faire d'appel qui pourrait provoquer une pause (utiliser GFP\_ATOMIC, , etc.).
- ✓ **Terminer aussi vite que possible**
- ✓ **Ordonnancer dans un *tasklet* si nécessaire**



# Annuler un URB

## ✓ Annulation d'un URB asynchrone (sans attente)

```
int usb_unlink_urb(struct urb *urb);
```

- **Succès: renvoie** `-EINPROGRESS`
- **Echec: n'importe quelle autre valeur**
  - Un URB jamais soumis
  - Quand le URB a déjà été annulé
  - Quand le URB a déjà été fait pas le matériel sans encore avoir eu l'appel de la fonction de completion.

## ✓ Annulation d'un urb synchrone (attente)

```
void usb_kill_urb(struct urb *urb);
```

- **Typiquement utilisé dans une fonction** `disconnect()` **ou** `close()`
- **Attention, cette fonction ne doit pas être utilisée dans un contexte qui ne peut pas attendre: un contexte d'interruption, dans un gestionnaire de completion, en attente de spinlock**
- **Voir les commentaires dans** `drivers/usb/core/urb.c`



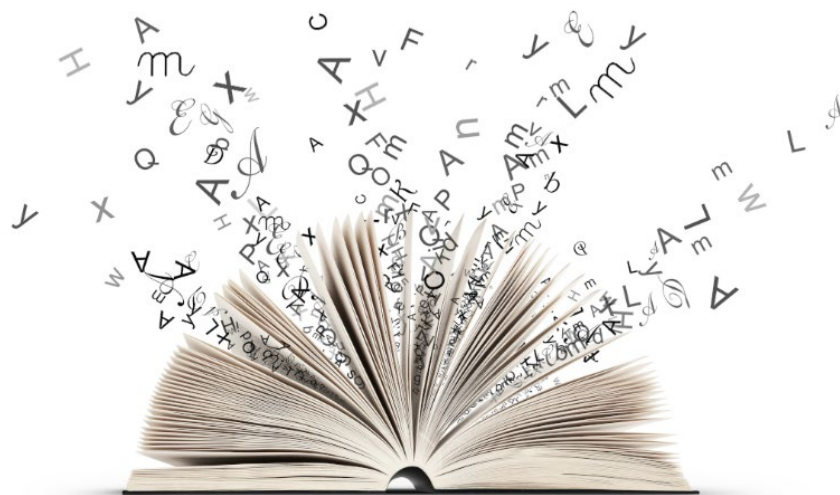
# Destruction d'un urb

- ✓ `void usb_poison_urb(struct urb *urb);`
  - **Tue de manière fiable un transfert et empêche l'utilisation ultérieure d'un URB**
- ✓ `void usb_free_urb(struct urb *urb);`
  - **libère la mémoire utilisée par un URB quand toutes les utilisations sont finies**



# Conseils pour les développements embarqués

- ✓ Si vous devez développer un pilote USB pour un systèmes Linux embarqué
  - Développer votre pilote sur votre environnement GNU/Linux hôte!
  - Votre pilote fonctionnera sans aucun changement sur votre systèmes cible (sous condition d'écrire du code portable, bien entendu!)
    - Tous les pilotes USB sont indépendants de la plateforme
  - Votre pilote sera plus facile à développer sur un système hôte grâce à plus de flexibilité et de confort de développements (outils et environnements de développement disponibles)



# Références

Quelques éléments pour aller plus loin



# Références bibliographiques

- ✓ **Descripteurs USB:**
  - <http://www.beyondlogic.org/usbnutshell/usb5.shtml>
- ✓ **Spécifications USB:**
  - <http://www.usb.org/developers/docs/>
- ✓ **Le chapitre « Pilotes USB » du « Linux Device Drivers book »:**
  - <http://static.lwn.net/images/pdf/LDD3/ch13.pdf> (Licence Libre!)
- ✓ **Linux USB project**
  - <http://www.linux-usb.org/>
- ✓ **Ecriture de pilote pour Linux:**
  - <http://opensourceforu.com/tag/linux-device-drivers-series/>
- ✓ **Cours Free-Electrons sur le développement de pilotes USB:**
  - <https://bootlin.com/doc/legacy/linux-usb/linux-usb.pdf>
- ✓ **Documentation du noyau Linux:**
  - `Documentation/usb/`
- ✓ **Sources du noyau Linux (des centaines d'exemples (« May the sources be with you! »))**