ELE-784
Ordinateurs et programmation système

Cours #5
Le pilote USB

Bruno De Kelper

Site internet :
http://www.ele.etsmtl.ca/academique/ele784/

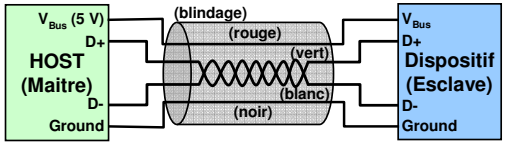
Plan d'aujourd'hui

1. Pilote USB (chap. 13)
   1. Les rudiments d'un pilote USB
   2. Les blocs de requête USB (Urb)
      1. Créer et initialiser un Urb
      2. Soumettre et compléter un Urb
      3. Éliminer ou annuler un Urb
   3. Détails d'un pilote USB
      1. Inscription du pilote USB
      2. Détails de "Probe" et "Disconnect"
      3. Soumettre et contrôler un Urb
   4. Les transferts sans Urbs

Réf. : Linux Device Drivers, 3ième éd., J. Corbet, A. Rubin, G. Kroah-Hartman, O'Reilly Media, chap. 13.

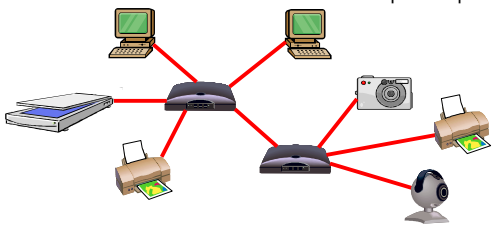
Pilote USB

- À la base, le Bus USB (Universal Serial Bus) est un Bus de type série-asynchrone bidirectionnel à deux fils, œuvrant selon le principe Maitre-Esclave :



Vitesse théorique
480 MBps

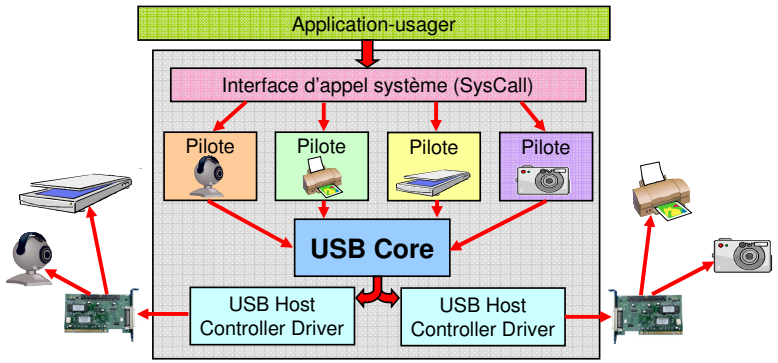
- Néanmoins, c'est un Bus complexe qui interconnecte de nombreux dispositifs, sous la forme d'un arbre de connections point-à-point :



  - Les unités-matériel ont la possibilité de demander une largeur de bande fixe pour leur transferts.
  - Le canal de communication USB ne requière pas que les données aient une signification ou structure particulière.

Pilote USB

- La complexité structurelle du Bus USB impose une complexité logicielle au niveau du pilote.
- En fait, le support logicielle du Bus USB reproduit sa structure matérielle, jusqu'à un certain niveau.



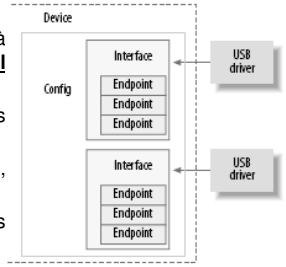
Pilote USB

- Donc, en fait, le **pilote USB** communique avec l'unité-matériel **au travers** d'un autre pilote, le **Noyau USB** (**USB Core**), qui lui communique avec les pilotes des **passerelles-matériel** (**USB Host Controller**).
- Les **standards USB** définissent une série de **classes d'unités-matériel** qui, lorsque l'appareil spécifique satisfait les spécifications de sa classe, ne requièrent pas le développement d'un pilote spécifique :

  Classes : - Unités de stockage - Souris - Unités-réseau
  - Claviers - Joystick - Modems

1.1 - Les rudiments d'un pilote USB

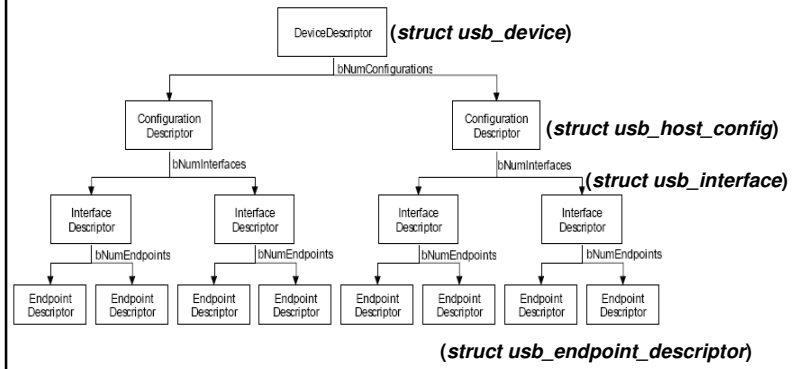
- Un **pilote USB** communique avec l'unité-matériel à l'aide de liens de communication **unidirectionnel** (**Endpoints**).
- L'**interface** avec l'unité-matériel **regroupe** tous les **liens de communication** reliés à celle-ci.
- Un **pilote USB** peut contenir **plusieurs interfaces**, selon le matériel qu'il contrôle (ex. : caméra+micro).
- Aussi, le pilote USB peut contenir et gérer plusieurs configurations pour une même unité-matériel.



Pilote USB

1.1 - Les rudiments d'un pilote USB

- Le tout définit la structure hiérarchique utilisée par le pilote USB :



Pilote USB

1.1 - Les rudiments d'un pilote USB

- Le Noyau de Linux fournit des structures de données standardisées pour gérer ses différents niveaux d'abstraction, qui sont définies dans "***usb.h***".
- La structure de base du pilote USB est **(*struct usb_device*),** tandis que pour les configurations d'unités-matériel, c'est **(*struct usb_host_config*)**.
- La structure **(struct usb_interface)** représente une unité-USB complète et se rattache directement à une des unités-matériel contrôlées par le pilote :

```
struct usb_interface {
    struct usb_host_interface *altsetting;
    struct usb_host_interface *cur_altsetting;
    unsigned num_altsetting;
    int minor;
    enum usb_interface_condition condition;
    struct device dev;
    struct class_device *class_dev;
};
```

Tableau de réglages alternatifs

Réglages courants

Nombre de réglages alternatifs

Numéro **minor** attribué à cette unité-matériel

Structure du **pilote** associé à l'unité-matériel

Attributs de la classe de cette unité-matériel

Pilote USB

1.1 - Les rudiments d'un pilote USB

Les Endpoints :

- C'est un lien de communication unidirectionnel, définit comme une entrée (**IN**) ou une sortie (**OUT**) au moment de son utilisation.
- Le Endpoint est soit **synchrone** ou **asynchrone** et est d'une de 4 **catégories** :

Endpoints asynchrones :

- Sont utilisés par le pilote pour des transfères à n'importe quel moment, selon le besoin.

CONTROL

- Utilisé pour configurer l'unité-matériel, récupérer de l'information ou des états, transmettre des commandes.
- Les transferts sont de petite taille et sont garantit d'avoir suffisamment de bande passante pour toujours arriver à destination.

BULK

- Utilisé pour transférer de grandes quantités de données, sans pertes.
- Ils ne sont pas garantit d'être transmit à l'intérieur d'un temps spécifique et si la quantité de données est large, elles peuvent être divisées dans plusieurs transferts.

Pilote USB

1.1 - Les rudiments d'un pilote USB

Les Endpoints :

Endpoints synchrones :

- Sont utilisés par le pilote pour des transfères **périodiques continus**, à intervalles réguliers (leur bande passante est réservée par le Noyau USB).

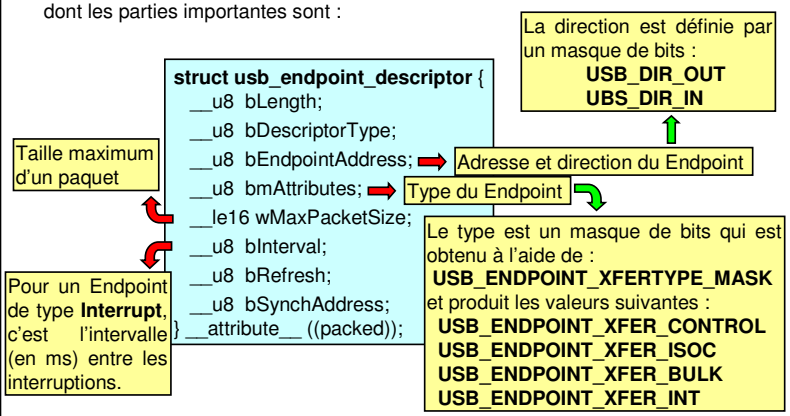
INTERRUPT	ISOCHRONOUS
- Utilisé habituellement pour recevoir de petites quantités de données à taux fixe ou transmettre des données de contrôle. - Les transferts sont de petite taille et sont garantit d'avoir suffisamment de bande passante pour toujours arriver à destination.	- Comme pour BULK, ils sont aussi utilisés pour de grandes quantités de données, mais celles-ci ne sont pas garanties d'arriver à destination. - Utilisés avec les unités-matériel qui peuvent supporter la perte de donnée. - Utiles pour assurer un flux constant de données, tel que dans les applications en temps réel (ex. : vidéo, audio).

Pilote USB

1.1 - Les rudiments d'un pilote USB

Les Endpoints :

- Les Endpoints sont gérés par le pilote et le Noyau grâce à la structure suivante, dont les parties importantes sont :



Pilote USB

1.2 - Les blocs de requête USB (Urb)

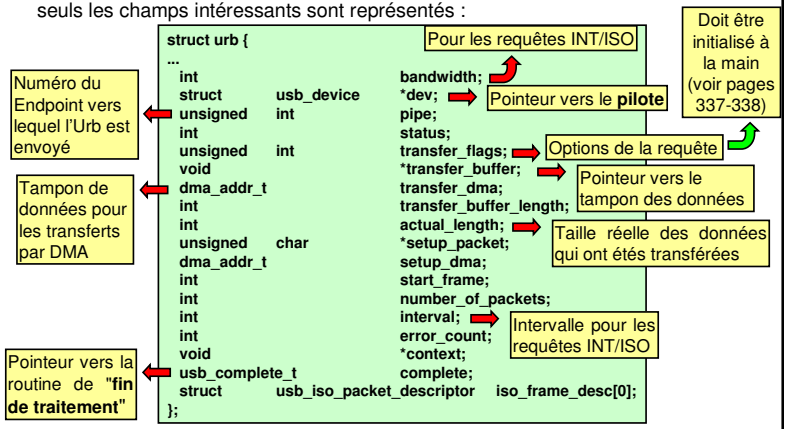
- Les **Urbs** sont utilisés pour **transmettre** ou **recevoir** des données d'une unité-USB, au travers d'un **Endpoint**, de façon asynchrone.
- Plusieurs **Urbs** peuvent être créés et envoyés à un même **Endpoint** (ils seront placés dans une **file d'attente**).
- Un même **Urb** peut être **réutilisé** plusieurs fois avec le même **Endpoint** ou avec différents **Endpoints**.
- Un **Urb** peut être **annulé** n'importe quand par le **pilote** qui l'a soumit ou par le **Noyau USB**, si l'unité-USB est **déconnectée** du système.
- Les **Urbs** sont créés **dynamiquement** et contiennent un **compte** des usagers qui le tiennent.
- Le **cycle de vie** d'un **Urb** est le suivant :

1 – Créé par le pilote	4 – Transféré à la passerelle-matériel
2 – Assigné à un Endpoint	5 – Traité à la passerelle-matériel
3 – Soumit au Noyau USB	6 – La passerelle-matériel informe le pilote lorsque le traitement est achevé

Pilote USB

1.2 - Les blocs de requête USB (Urb)

- Les Urbs sont représentés dans le Noyau USB par la structure suivante, dont seuls les champs intéressants sont représentés :



Pilote USB

1.2 - Les blocs de requête USB (Urb)

1.2.1 - Créer et initialiser un Urb :

- Les Urbs doivent absolument être créés dynamiquement à l'aide de :

```
struct urb *usb_alloc_urb (int iso_packets, int mem_flags);
```

- Retourne un pointeur vers le Urb qui a été créé ou NULL
- Nombre de paquets isochrones, si le Urb sera de type ISO.
- Type d'allocation dynamique (voir pages 213 à 215)

Création du tampon de données d'un Urb :

- Le tampon de données de l'Urb doit être créé dynamiquement.

Par exemple :

(allocation standard)

```
void *buf = NULL;
…
buf = kmalloc (size, GFP_KERNEL);
```

(allocation pour accès par DMA)

```
void *buf = NULL;
…
urb->transfer_flags |= URB_NO_TRANSFER_DMA_MAP;
buf = usb_buffer_alloc (dev, size, GFP_KERNEL, &urb->transfer_dma);
```


Pilote USB

1.2 - Les blocs de requête USB (Urb)

1.2.1 - Créer et initialiser un Urb :

- Une fois créé, l'Urb doit être attaché à son Endpoint, son tampon de données et à sa fonction de "**fin de traitement**".
- L'Urb doit être initialisé selon le type de requête qu'il effectuera :

Urb BULK :

```
void usb_fill_bulk_urb (struct urb *urb, struct usb_device *dev,
                        unsigned int pipe, void *transfer_buffer,
                        int buffer_length, usb_complete_t complete,
                        void *context);
```

- Pointeur vers le **Urb**
- Pointeur vers le **pilote**
- **Endpoint** qui sera utilisé par le **Urb**
- Pointeur vers le **tampon** des données
- Pointeur vers le paramètre qui sera fournit à la routine de "**fin de traitement**"
- Pointeur vers la routine de "**fin de traitement**"

- L'initialisation des **Urbs** des types **Interrupt** et **Control** est identique, sauf pour un ou deux détails.
- Pour les **Urbs** de type **Isochronous**, l'initialisation doit être entièrement faite à la main.

Pilote USB

1.2 - Les blocs de requête USB (Urb)

1.2.1 - Créer et initialiser un Urb :

Urb INTERRUPT : (identique au Urb BLUK)

```
void usb_fill_int_urb (struct urb *urb, struct usb_device *dev,
                       unsigned int pipe, void *transfer_buffer,
                       int buffer_length, usb_complete_t complete,
                       void *context, int interval);
```

(sauf) → **Intervalle** d'exécution du **Urb**

Unités : Pour **low-speed** et **full-speed** : Cadres (ms)
Pour **High-speed** : microCadres (1/8 ms)

Urb CONTROL : (identique au Urb BLUK)

(sauf) Pointeur vers le paquet de préparation du transfère

```
void usb_fill_control_urb (struct urb *urb, struct usb_device *dev,
                           unsigned int pipe, unsigned char *setup_packet,
                           void *transfer_buffer, int buffer_length,
                           usb_complete_t complete, void *context, int interval);
```


Pilote USB

1.2 - Les blocs de requête USB (Urb)

1.2.1 - Créer et initialiser un Urb :

Urb ISOCHRONOUS :

- Les Urbs isochrones doivent être initialisés à la main.

Par exemple :

```
urb->dev = dev;
urb->context = dev;
urb->pipe = usb_rcvisocpipe(dev, endpointDesc.bEndpointAddress);
urb->transfer_flags = URB_ISO_ASAP | URB_NO_TRANSFER_DMA_MAP;
urb->interval = endpointDesc.bInterval;
urb->complete = complete_callback;
urb->number_of_packets = nbPackets;
urb->transfer_buffer_length = size;
for (j = 0; j < nbPackets; ++j) {
   urb->iso_frame_desc[j].offset = j * packetSize;
   urb->iso_frame_desc[j].length = packetSize;
}
```

(Ce qui fait essentiellement la même chose que les fonctions précédentes)

Pilote USB

1.2 - Les blocs de requête USB (Urb)

1.2.1 - Créer et initialiser un Urb :

- Lors de l'**initialisation** d'un **Urb**, le type du **Endpoint** qui sera utilisé doit être caractérisé de la façon suivante :

Pour un **Endpoint** de **transmission** :

```
unsigned int usb_sndctrlpipe  (struct usb_device *dev, unsigned int endpoint);
unsigned int usb_sndbulkpipe (struct usb_device *dev, unsigned int endpoint);
unsigned int usb_sndintpipe   (struct usb_device *dev, unsigned int endpoint);
unsigned int usb_sndisocpipe (struct usb_device *dev, unsigned int endpoint);
```

La valeur retournée est utilisée pour l'initialisation du **Urb**

Pointeur vers la structure du **pilote**

Numéro du **Endpoint**

Pour un **Endpoint** de **reception** :

```
unsigned int usb_rcvctrlpipe  (struct usb_device *dev, unsigned int endpoint);
unsigned int usb_rcvbulkpipe (struct usb_device *dev, unsigned int endpoint);
unsigned int usb_rcvintpipe   (struct usb_device *dev, unsigned int endpoint);
unsigned int usb_rcvisocpipe (struct usb_device *dev, unsigned int endpoint);
```


Pilote USB

1.2 - Les blocs de requête USB (Urb)

1.2.2 - Soumettre et compléter un Urb :

- Lorsque le Urb a été créé et initialisé correctement, il peut être soumit au Noyau USB avec :

```
int usb_submit_urb (struct urb *urb, int mem_flags);
```

Retourne 0 (succès) ou un code d'erreur

Pointeur vers le **Urb**

Type d'allocation de mémoire qui sera permise au Noyau USB

Valeurs habituelles :
GFP_ATOMIC
GFP_NOIO
GFP_KERNEL
(voir pages 213-215)

Compléter un Urb :

- Lorsque le Noyau USB a fini avec le Urb, il retourne le contrôle de l'Urb au pilote en appelant la routine de "**fin de traitement**" (***usb_complete_t***).

Prototype :

```
static void complete_callback (struct urb *urb, struct pt_regs *regs);
```

- La routine de "**fin de traitement**" est appelée dans trois situations :
  - L'Urb a été traité sans problème (urb->status = 0)
  - L'Urb a été traité avec des erreurs (urb->status = code d'erreur)
  - L'Urb a été annulé (urb->status = code d'erreur)

Pilote USB

1.2 - Les blocs de requête USB (Urb)

1.2.3 - Éliminer ou annuler un Urb :

Élimination d'un Urb :

- Lorsque le Urb n'est plus utilisé par le pilote, il doit être éliminé du système avec :

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

Pointeur vers le Urb qui doit être éliminé

Annulation d'un Urb :

- Un Urb peut être annulé n'importe quand par le pilote ou par le Noyau USB

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

Utilisée par le Noyau USB ou dans la fonction de rappel "***disconnect***" lorsque l'unité-USB est déconnectée du système.

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

Cette fonction est **non-bloquante**, mais requière que le drapeau "***flag***" soit initialisé à **URB_ASYNC_UNLINK**.

Pilote USB

1.3 - Détails d'un pilote USB

1.3.1 - Inscription du pilote USB :

- Dans le cas d'un **pilote USB**, le numéro **MAJOR** est attribué au **Noyau-USB** et l'enregistrement du pilote est faite à l'aide de ***usb_register ()***, après avoir initialisé la structure (***struct usb_driver***).
- La structure (***struct usb_driver***) ressemble à la structure (struct file_operations), plus particulièrement, elle contient des pointeurs vers les fonctions qui sont supportées par le pilote :

```
struct usb_driver {
   const char *name;
   int      (*probe)      (struct usb_interface *intf, const struct usb_device_id *id);
   void     (*disconnect) (struct usb_interface *intf);
   int      (*ioctl)      (struct usb_interface *intf, unsigned int code, void *buf);
   int      (*suspend)    (struct usb_interface *intf, pm_message_t message);
   int      (*resume)     (struct usb_interface *intf);
   const struct usb_device_id *id_table;
   ...
};
```

Nom du Pilote

Pointeurs vers les fonctions supportées

Structure qui identifie la liste des unités-USB supportées par le pilote.

Pilote USB

1.3 - Détails d'un pilote USB

1.3.1 - Inscription du pilote USB :

- En plus de la structure (***struct usb_driver***) qui inscrit le pilote dans le Noyau-USB, la structure (***struct usb_device_id***) identifie les types d'unités-USB qui sont supportées par le pilote.
- Cette information sert au Noyau-USB, entre autre, pour identifier le bon pilote pour un appareil-USB qui vient d'être connecté au système.
- Pour effectuer cette identification, le **Noyau-USB** appelle la fonction ***probe ()*** de chaque pilote pour lui demander si la nouvelle unité-USB lui appartient.
- En fait, le pilote déclare un tableau de cette structure.

```
struct usb_device_id {
    __u16           match_flags;
    __u16           idVendor;
    __u16           idProduct;
    __u16           bcdDevice_lo;
    __u16           bcdDevice_hi;
    __u8            bDeviceClass;
    __u8            bDeviceSubClass;
    __u8            bDeviceProtocol;
    __u8            bInterfaceClass;
    __u8            bInterfaceSubClass;
    __u8            bInterfaceProtocol;
    kernel_ulong_t  driver_info;
};
```

Spécifique au produit

Spécifique à la classe

Spécifique à l'interface

Pilote USB

1.3 - Détails d'un pilote USB

1.3.1 - Inscription du pilote USB :

- Avant d'inscrire le pilote-USB dans le Noyau-USB, ces deux structures doivent être déclarées et pré-initialisées :

Par exemple :

```
static struct usb_device_id skel_table [ ] = {
{ USB_DEVICE(VENDOR_ID, PROD_ID) },
{ }
};
MODULE_DEVICE_TABLE (usb, skel_table);
```

1ière entrée dans la table

2ième entrée dans la table

Macro utilisée pour l'initialisation

```
USB_DEVICE(vendor, product)
USB_DEVICE_VER(vendor, product, lo, hi)
USB_DEVICE_INFO(class, subclass, protocol)
USB_INTERFACE_INFO(class, subclass, protocol)
```

Et

```
static struct usb_driver skel_driver = {
.name = "skeleton",
.id_table = skel_table,
.probe = skel_probe,
.disconnect = skel_disconnect,
};
```

Autres Macros pouvant servir à l'initialisation (voir page 347)

Fonctions qui seront implémentées

Pilote USB

1.3 - Détails d'un pilote USB

1.3.1 - Inscription du pilote USB :

- Lorsque ces deux structures ont été créées correctement, le **pilote-USB** est inscrit dans le **Noyau-USB** avec la fonction ***usb_register ()*** :

Par exemple :

```
static int __init usb_skel_init (void) {
    int result;
    result = usb_register(&skel_driver);
    if (result)
        err("usb_register failed. Error number %d", result);
    return result;
}

static void __exit usb_skel_exit (void) {
    usb_deregister(&skel_driver);
}

module_init (usb_skel_init);
module_exit (usb_skel_exit);
```

La structure du Pilote

Inscription du Pilote

Élimination du Pilote

La structure du Pilote

Déclarations des fonctions d'initialisation et d'élimination du Pilote

Pilote USB

1.3 - Détails d'un pilote USB

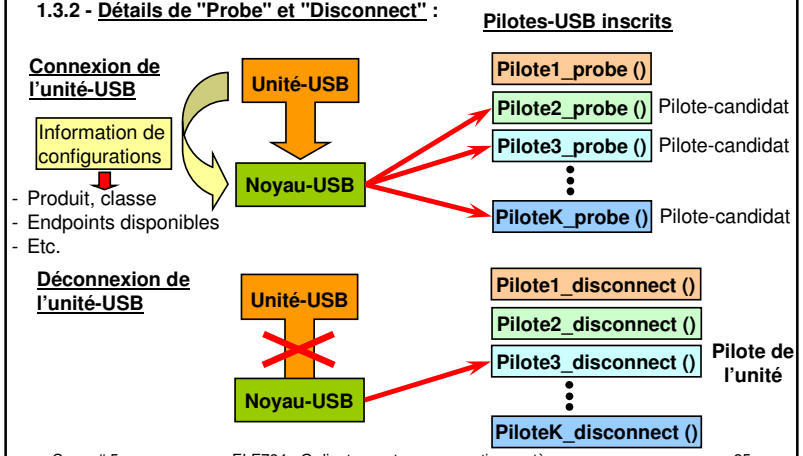
1.3.2 - Détails de "Probe" et "Disconnect" :

- Une particularité du **Pilote-USB** est qu'il se superpose par-dessus un autre Pilote, le **Noyau-USB**.
- Une autre particularité est qu'une **unité-USB** peut **apparaître** ou **disparaître** n'importe quand, lorsqu'elle est connectée ou déconnectée du système.
- C'est le **Noyau-USB** qui est chargé de gérer l'apparition ou la disparition d'une unité-USB en rattachant l'**unité-USB** au **Pilote-USB** qui lui correspond.
- Lors de l'**apparition** d'une unité-USB, le **Noyau-USB** appelle la fonction ***probe ()*** des pilotes-USB **inscrits** qui **pourraient gérer** cette nouvelle unité-USB.
- Le Noyau-USB se base sur l'information fournie par (***struct usb_device_id***) des **Pilotes-USB** inscrits et l'information fournie par l'**unité-USB** elle-même.
- Lors de la **disparition** de l'unité-USB, le Noyau-USB appelle la fonction ***disconnect ()*** du pilote-USB qui a accepté l'unité-USB.
- Donc, les fonctions ***probe ()*** et ***disconnect ()*** doivent obligatoirement être implémentées dans le **Pilote-USB**.

Pilote USB

1.3 - Détails d'un pilote USB

1.3.2 - Détails de "Probe" et "Disconnect" :



Pilote USB

1.3 - Détails d'un pilote USB

1.3.2 - Détails de "Probe" et "Disconnect" :

Fonction *probe ()* :

Identité de l'unité-USB

Interface de l'unité-USB

```
int my_probe (struct usb_interface *intf,  const struct usb_device_id *id);
```

- Afin d'améliorer les **performances** du **Noyau-USB** lorsqu'il tente de détecter le bon **pilote-USB**, la fonction ***probe ()*** doit **limiter** son action à la **vérification** et à l'obtention des **informations** dont le pilote a besoin.
- Par exemple, la fonction ***probe ()*** peut se limiter à :
  - Choisir la bonne configuration des **réglages** de l'**interface** dans la liste des réglages alternatifs (voir **struct usb_interface**).
  - Déterminer si les **Endpoints** requis sont **disponibles** dans les **réglages** de l'**interface**.
  - **Conserver** l'information sur les **Endpoints** trouvés dans une structure locale.
  - Attacher la structure locale du pilote à l'**interface** de l'**unité-USB**.
  - Inscrire l'**unité-USB** dans le **Noyau-USB** afin de permettre à une application-usager d'en utiliser ses **services**.

Pilote USB

Fonction *probe ()* :

Par exemple : En supposant que le pilote possède une structure locale appelée (***struct usb_skel***), dont le détail n'est pas donné ici.

```
static int my_probe (struct usb_interface *intf, const struct usb_device_id *devid) {
   const struct usb_host_interface        *interface;
   const struct usb_endpoint_descriptor *endpoint;
   struct usb_device *dev = interface_to_usbdev (intf);
   struct usb_skel    *skeldev = NULL;
   int                n, m, altSetNum, activeInterface = -1;

   skeldev = kmalloc (sizeof(struct usb_skel), GFP_KERNEL);
   skeldev->usbdev = usb_get_dev (dev);
   for (n = 0; n < intf->num_altsetting; n++) {
      interface = &intf->altsetting[n];
      altSetNum = interface->desc.bAlternateSetting;
      for (m = 0; m < interface->desc.bNumEndpoints; m++) {
         endpoint = &interface->endpoint[m].desc;
```

Récupère la structure du pilote-USB

Alloue la structure locale du pilote

Passe au travers des réglages alternatifs de l'interface

Passe au travers des Endpoints de l'interface

Pilote USB

Fonction *probe ()* :

(suite de l'exemple)

Est-ce que le Endpoint est de type BULK-IN ?

```
if (!skeldev->bulk_in_endpointAddr &&
    (endpoint->bEndpointAddress & USB_DIR_IN) &&
    ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
      == USB_ENDPOINT_XFER_BULK)) {
   skeldev->bulk_in_size = endpoint->wMaxPacketSize;
   skeldev->bulk_in_endpointAddr = endpoint->bEndpointAddress;
   skeldev->bulk_in_buffer = kmalloc(skeldev->bulk_in_size, GFP_KERNEL);
   activeInterface = altSetNum;
   break;
}
}
if (activeInterface != -1)
   break;
}
usb_set_intfdata (intf, skeldev);
usb_register_dev (intf, &skel_class);
usb_set_interface (dev, interface->desc.bInterfaceNumber, activeInterface);
return 0;
}
```

Conserve dans la structure locale du pilote les informations sur le Endpoint et sort des deux boucles car on a trouvé la bonne configuration de réglages d'interface.

Attache la structure locale à l'interface

Inscrit l'unité-USB dans le Noyau-USB

Attribue la configuration de réglages trouvée à l'interface.

Pilote USB

1.3 - Détails d'un pilote USB

1.3.2 - Détails de "Probe" et "Disconnect" :

Fonction *probe ()* :

- Lors de l'inscription de l'unité-USB dans le Noyau-USB avec la fonction ***usb_register_dev ()***, cette fonction reçoit la structure (***struct usb_class_driver***).
- Cette structure permet de connecter les services du pilote-USB à l'application-usager qui veut l'utiliser, au travers des appels-système, comme dans le cas de n'importe quel pilote.
- Elle est initialisée par le pilote-USB avec, entre autre, une (**struct file_operations**).

Par exemple :

```
static struct file_operations skel_fops = {
   .owner  =  THIS_MODULE,
   .read    =  skel_read,
   .write    =  skel_write,
   .open    =  skel_open,
   .release =  skel_release,
};
```

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


Pilote USB

1.3 - Détails d'un pilote USB

1.3.2 - Détails de "Probe" et "Disconnect" :

Fonction *disconnect ()* :

Interface de l'unité-USB

```
void my_disconnect (struct usb_interface *intf);
```

- La fonction ***disconnect ()*** est appelée par le **Noyau-USB** lorsque le pilote-USB doit arrêter d'utiliser l'**unité-USB**, par exemple lorsqu'elle est déconnectée du système.
- Elle sert essentiellement à faire du nettoyage.
- Lorsque la fonction ***disconnect ()*** est appelée, le **Noyau-USB** a déjà **annulé** tous les **Urbs** qui étaient en instance de traitement; il n'est donc pas nécessaire de le faire ici.
- La fonction peut se borner à :
  - Détacher la structure locale de l'interface (par sécurité)
  - Désinscrire l'unité-USB du Noyau-USB à l'aide de ***usb_deregister_dev ()***

Pilote USB

1.3 - Détails d'un pilote USB

1.3.2 - Détails de "Probe" et "Disconnect" :

Fonction *disconnect ()* :

Par exemple :

```
static void skel_disconnect (struct usb_interface *intf) {
   lock_kernel();                          // Bloque la préemption du Noyau-Linux
   usb_set_intfdata (intf, NULL);          // Détache la structure locale de l'interface
   usb_deregister_dev (intf, &skel_class); // Désinscrit l'unité-USB du Noyau-USB
   unlock_kernel();                        // Débloque la préemption du Noyau-Linux
}
```


Pilote USB

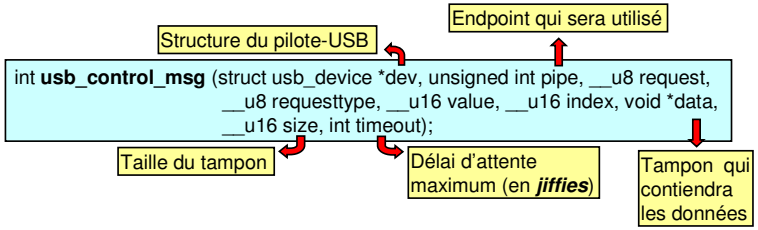
1.4 - Les transferts sans Urbs

- L'utilisation des **Urbs** est une méthode utile pour communiquer avec une **unité-USB**, mais est assez lourde lorsque le message à communiquer est petit.
- Le **Noyau-USB** fournit deux autres fonctions permettant de communiquer facilement des messages de petite dimension, tel que lors de la fonction ***ioctl ()*** du pilote : ***usb_bulk_msg ()*** et ***usb_control_msg ()***
- Ces deux fonctions permettent de **transmettre** ou de **recevoir** un message d'une **unité-USB** et d'**attendre** (**dormir**) que la communication se termine.
- Elles ne peuvent donc être utilisées à partir du contexte d'une **interruption** ou lorsqu'un **verrou tournant** (**spin lock**) a été capturé.
- De plus, elles ne peuvent être **interrompue** et la fonction ***disconnect ()*** du **pilote-USB** doit être avertit qu'il doit attendre qu'elles se terminent avant de **désinscrire** l'**unité-USB**.
- Chacune peuvent spécifier un **délai** d'attente maximum (**timeout**), en nombre de ***jiffies*** (lorsque 0, aucun délai maximum n'est utilisé).
- Aussi, les deux retournent **0** (**zéro**) lorsque la communication s'est bien déroulée ou un **code d'erreur** (nombre négatif) dans le cas contraire.

Pilote USB

1.4 - Les transferts sans Urbs

La fonction *usb_control_msg ()* :

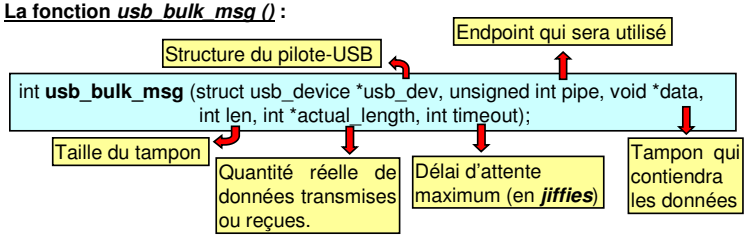


- Cette fonction sert à communiquer des messages de type Control (petit bloc de données, communication asynchrone).
- Le Endpoint qui lui est fournit doit être obtenu avec ***usb_rcvctrlpipe ()*** (réception) ou ***usb_sndctrlpipe ()*** (transmission).
- Les paramètres ***request***, ***requesttype***, ***value*** et ***index*** sont défini dans le chapitre 9 des spécifications USB.

Pilote USB

1.4 - Les transferts sans Urbs

La fonction *usb_bulk_msg ()* :



- Cette fonction sert à communiquer des messages de type **Bulk** (gros bloc de données, communication asynchrone).
- Le Endpoint qui lui est fournit doit être obtenu avec ***usb_rcvbulkpipe ()*** (réception) ou ***usb_sndbulkpipe ()*** (transmission).

Pilote USB

1.4 - Les transferts sans Urbs

La fonction *usb_bulk_msg ()* :

Exemple d'utilisation

```
retval = usb_bulk_msg (dev->udev, usb_rcvbulkpipe(dev->udev,
                       dev->bulk_in_endpointAddr), dev->bulk_in_buffer,
                       min(dev->bulk_in_size, count), &count, HZ*10);
if (!retval) {
   if (copy_to_user(buffer, dev->bulk_in_buffer, count))
      retval = -EFAULT;
   else
      retval = count;
}
```

Si tout s'est bien passé, transmet les données reçues à l'usager