

Utilisation du bus USB

Christophe BLAESS

christophe.blaess@logilin.fr

<https://www.blaess.fr/christophe/>



Ingénierie et formations sur Linux et les logiciels libres
<https://www.logilin.fr>

Le sous-système USB de Linux.....	3
Principe des périphériques USB.....	3
Interface USB sous Linux.....	5
Implémentation d'un driver.....	6
Enregistrement du driver.....	6
Travaux pratiques : communication avec un bus USB.....	7
Endpoints et types de dialogues.....	8
Travaux pratiques : énumération des endpoints USB.....	10
Ajout d'une classe de driver.....	11
Communication avec les URB.....	12
Travaux pratiques : écriture sur un device USB.....	14
Travaux pratiques : lecture d'un device USB.....	16
Exemples de drivers pour autres modes.....	17
Endpoint Bulk et Control.....	17

Ce support de formation est distribué sous licence **Creative Commons 4.0**



(Attribution - Partage dans les mêmes conditions).

Vous êtes libres de copier et partager ce document, en mentionnant son origine. Si vous l'intégrez dans un contenu plus vaste, ce dernier devra être distribué avec les mêmes droits.

Ce cours a été rédigé en utilisant des logiciels libres sur système d'exploitation Linux :

- *LibreOffice Writer* pour le support et la mise en page
- *LibreOffice Draw* pour les dessins vectoriels
- *Gimp* pour les images bitmap

v. 9.12

Le sous-système USB de Linux

Principe des périphériques USB

La norme USB (*Universal Serial Bus*) développée dans le courant des années 1990 était destinée à remplacer les ports série RS-232, parallèle Centronic, et clavier-souris PS/2.

La norme USB 1.0 (puis 1.1) proposait deux vitesses de fonctionnement :

- 1,5 Mbits/s (basse vitesse - *Low Speed*) ;
- 12 Mbits/s (haute vitesse - *Full Speed*).

La norme USB 2.0 permet d'obtenir un débit jusqu'à 480 Mbits/s (*Hi-Speed*).

Le standard USB 3.0 (*SuperSpeed*) puis 3.1 affiche un débit allant jusqu'à 4,8 Gbits/s.

Le standard USB 3.1 Gen 2 ajoute un nouveau type de prises réversibles (USB-C) et un débit allant jusqu'à 10 Gbits/s.

La norme USB 3.2 propose un débit de 20 Gbits/s et la version USB4 annonce un débit théorique maximum de 40 Gbits/s.

La norme USB définit le dialogue entre :

- un *hôte* (ex. ordinateur),
- et un *périphérique* (ex. disque externe).

Avec la norme USB 2.0, une extension USB *on-the-go* (OTG) est apparue permettant à un périphérique de devenir à son tour maître (exemple : tablette tactile contrôlant un disque dur).

Les connecteurs USB contiennent une alimentation 5V pour les périphériques.

Les données sont transmises par codage différentiel pour assurer une bonne robustesse face au bruit électromagnétique ("0" codé par $D-=D+=0\text{ V}$; "1" codé par $D-= -3.3\text{ V}$ et $D+= +3.3\text{ V}$).

Pour en savoir plus...

Les spécifications USB sont disponibles sur le site de l'*USB Implementers Forum* (<http://www.usb.org/developers/docs/>).

Les connexions USB s'établissent sous forme de réseau en étoile, avec un unique *hôte* et plusieurs *périphériques*. L'utilisation de *hub* permet d'augmenter le nombre de périphériques (jusqu'à 127 au maximum).

Les périphériques USB dialoguent directement avec l'ordinateur hôte et n'envoient des données que lorsque celui-ci les interroge.

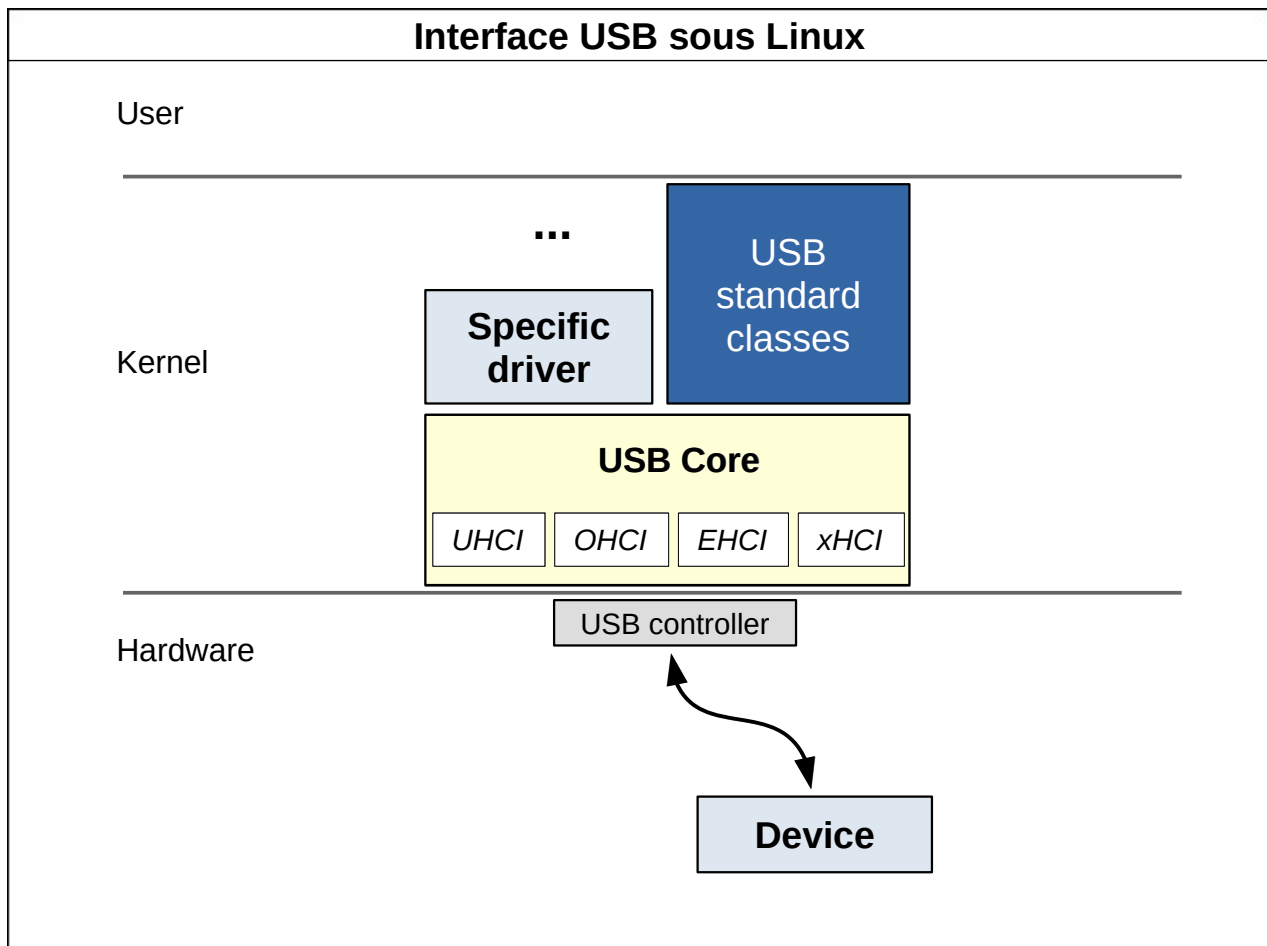
Il n'y a pas de possibilité de liaison directe entre deux hôtes ou d'utiliser plusieurs hôtes par réseau.

Avantages des connexions USB :

- Standard, portabilité
- Branchement et extraction à chaud
- Classes de périphériques pré-définies

Inconvénients :

- Débit relativement faible (du moins avant USB 3.0)
- Pas de mécanisme d'interruption, *polling* à la milliseconde
- Protocole de dialogue et interface physique assez complexes



L'implémentation du protocole USB côté hôte fut défini par plusieurs API successives :

- *Open Host Controller Interface (OHCI)* : USB 1.0 et 1.1 par Compaq, Microsoft, etc. ;
- *Universal Host Controller Interface (UHCI)* : USB 1.0 et 1.1, développée par Intel ;
- *Enhanced Host Controller Interface (EHCI)* : USB 2.0 ;
- *Extensible Host Controller Interface (xHCI)* : USB 3.0.

Linux gère l'interface hôte bas-niveau dans un sous-système nommé *USB-core* et implémente les principales *classes* USB dans des drivers prédéfinis :

- *Mass Storage* : clés USB, disques externes, etc.
- *Human Interface Device (HID)* : claviers, souris, dalles tactiles, joysticks...
- *Modem* : standard *Communication Device Class Abstract Control Modem (CDC ACM)*.
- *Printer* : imprimantes...

Implémentation d'un driver

Enregistrement du driver

```
<linux/usb.h>

int  usb_register  (struct usb_driver *driver);
void usb_deregister (struct usb_driver *driver);

struct usb_driver {
    const char *name;

    int  (*probe) (struct usb_interface      *intf,
                  const struct usb_device_id *id);
    void (*disconnect)(struct usb_interface *intf);
    [...]
    const struct usb_device_id *id_table;
};

(struct usb_device_id *)USB_DEVICE(ID_VENDOR, ID_PRODUCT)

(void) MODULE_DEVICE_TABLE(usb, id_table)
```

La structure `usb_driver` est la représentation du driver pour le sous-système *USB Core*.

La méthode `probe()` est invoquée lorsqu'un périphérique correspondant aux paramètres de la table `id_table` est détecté. Elle doit renvoyer zéro si elle peut le piloter et une erreur négative sinon. La méthode `disconnect()` est appelée à l'extraction du périphérique ou au déchargement du module.

Le nom fournit dans le champ `name` de la structure `usb_driver` doit être unique sur le système.

La structure `usb_driver` contient également des méthodes facultatives – et rarement implémentées – comme `ioctl()` (invoquée via *usbfs*), `suspend()` et `resume()` (appelées lors de la mise en veille de l'ordinateur).

La macro `USB_DEVICE()` construit une structure `usb_device_id` à partir de l'identifiant du vendeur du matériel (sur 16 bits) et de l'identifiant du produit chez le vendeur (16 bits également). On l'utilise pour initialiser la table `id_table` de la structure `usb_driver`.

La macro `MODULE_DEVICE_TABLE()` permet d'exporter la table des identifiants pour qu'elle soit accessible au mécanisme *hotplug* et permettre le chargement automatique du module.

Travaux pratiques : communication avec un bus USB

Branchez une carte d'acquisition Velleman K8055 sur votre poste. Dans le message `dmesg`, retrouvez l'identifiant du vendeur et celui du produit.

Reportez-les dans le code source du module `examp le-VII-01.c` puis compilez le module.

Chargez le module. Qu'observez-vous ?

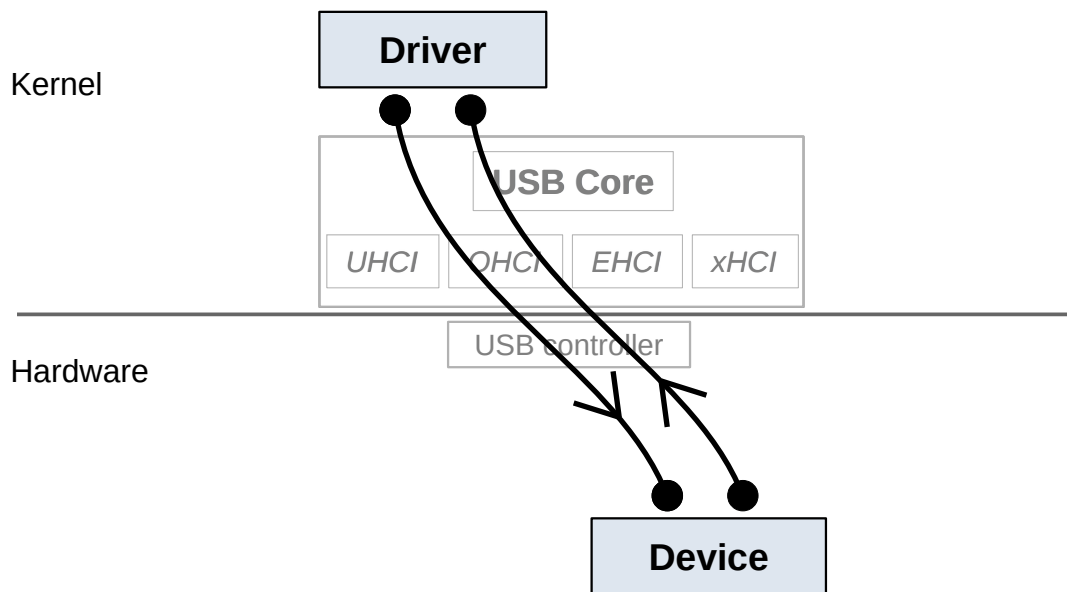
Si le module USBHID prend le contrôle du périphérique dès qu'on le branche, il faut le désactiver, avant de charger notre module, en écrivant dans `/sys/bus/usb/drivers/usbhid/unbind` le numéro USB de la carte :

```
# grep 10cf /sys/bus/usb/devices/*/idVendor
/sys/bus/usb/devices/4-2/idVendor:10cf
# ls /sys/bus/usb/devices/4-2
4-2:1.0          bMaxPacketSize0    devpath            quirks
authorized      bMaxPower          driver             remove
[...]
```

```
# echo -n 4-2:1.0 > /sys/bus/usb/drivers/usbhid/unbind
#
```

Débranchez la carte. Qu'observez-vous ?

Endpoints et types de dialogues



Pour dialoguer avec le périphérique, le driver utilise un (ou plusieurs) *endpoint*, sorte de descripteur géré par le sous-système *USB Core*.

Il existe quatre types d'*endpoint* :

- *bulk* : transfert de données volumineuses sans pertes, la bande passante n'est pas garantie, les données peuvent être scindées en plusieurs paquets (ex : disques externes, etc.) ;
- *control* : utilisé pour paramétrer le périphérique, le transfert des données est garanti mais la bande passante est faible ;
- *interrupt* : pour le transfert périodique de données, la bande passante est réservée (généralement assez faible, ex. clavier, souris, etc.) ;
- *isochronous* : transfert de données volumineuses mais sans garantie de résultat, les données sont perdues si la bande passante n'est pas suffisante (ex : webcam, carte son, etc.).

```
struct usb_interface {
    struct usb_host_interface *cur_altsetting;
    [...]
};
```

```
struct usb_host_interface {
    struct usb_interface_descriptor desc;
    struct usb_host_endpoint *endpoint;
    [...] };
```

```
struct usb_interface_descriptor {
    __u8 bNumEndpoints;
    [...] };

struct usb_host_endpoint { // [...]
    struct usb_endpoint_descriptor desc;
    [...] };
```

```
struct usb_endpoint_descriptor {
    __u8 bEndpointAddress;
    __u8 bmAttributes;
    __le16 wMaxPacketSize;
    __u8 bInterval;
};
```

Dans la méthode `probe()` du driver, on reçoit en argument une structure `usb_interface` qui contient, entre autres, un champ pointant vers une structure `usb_host_interface`. Certains périphériques peuvent proposer plusieurs interfaces différentes.

La structure `usb_host_interface` regroupe une table de `usb_host_endpoint` et une structure `usb_interface_descriptor` donnant accès au nombre de *endpoints* dans la table.

Les champs importants de la structure `usb_endpoint_descriptor` sont :

- `bEndpointAddress` : adresse USB du *endpoint* ainsi que la direction de la communication (masque `USB_DIR_IN` ou `USB_DIR_OUT`);
- `bmAttributes` : type d'*endpoint* : le masque `USB_ENDPOINT_XFERTYPE_MASK` permet d'extraire `USB_ENDPOINT_XFER_BULK`, `USB_ENDPOINT_XFER_CONTROL`, `USB_ENDPOINT_XFER_INT`, ou `USB_ENDPOINT_XFER_ISOC` ;
- `wMaxPacketSize` : taille maximale d'un bloc transmis en une seule fois ;
- `bInterval` : délai en millisecondes entre deux requêtes en mode *interrupt* ;

Travaux pratiques : énumération des endpoints USB

Observez le contenu du fichier `example-VII-02.c`, il énumère les *endpoints* disponibles sur l'interface reçue dans sa fonction `probe()`, ainsi que leur type, et leur sens de fonctionnement.

Chargez le module. Combien voyez-vous de *endpoints* ?

Ajout d'une classe de driver

Le driver doit proposer une interface côté utilisateur en fonction du type de périphérique qu'il gère (caractère, bloc, réseau...). Pour une interface caractère, on peut se rattacher à celle USB générale qui présente l'intérêt d'entrer dans le modèle des drivers du noyau 2.6.

```
int  usb_register_dev  (struct usb_interface  *intf,
                       struct usb_class_driver *drv);

void usb_deregister_dev (struct usb_interface  *intf,
                       struct usb_class_driver *drv);

struct usb_class_driver {
    char * name; // peut contenir %d pour numero d'interface
    const struct file_operations *fops;
    int minor_base;
};
```

```
# ls /sys/class/usbmisc/
hiddev0 velleman0
# ls -l /sys/class/usb/velleman0/
total 0
-r--r--r-- 1 root root 4096 Sep  9 10:55 dev
lrwxrwxrwx 1 root root    0 Sep  9 10:55 device -> ../../../../4-2:1.0
[...]
# ls -l /dev/vell*
crw-rw---- 1 root root 180, 0 Sep  9 10:55 /dev/velleman0
#
```

La structure `usb_class_driver` identifie un driver qui sera accessible par un fichier spécial en mode caractère avec le numéro majeur du sous-système USB (180) et un numéro mineur personnalisé.

Pour en savoir plus...

Voir les commentaires détaillés dans le fichier `linux/include/usb.h`.

Communication avec les URB

On communique avec le périphérique en soumettant des URB (*USB Request Block*) au sous-système USB Core via un *endpoint*.

```
struct urb * usb_alloc_urb (int iso_pkt, gfp_t flags);  
void usb_free_urb (struct urb *urb);
```

```
void usb_fill_bulk_urb (struct urb *urb,  
                        struct usb_device *dev, unsigned int pipe,  
                        void *buffer, int buffer_length,  
                        usb_complete_t complete_cb, void *private);
```

```
void usb_fill_control_urb (struct urb *urb,  
                           struct usb_device *dev, unsigned int pipe,  
                           unsigned char *setup_data_packet,  
                           void *buffer, int buffer_length,  
                           usb_complete_t complete_cb, void *private);
```

```
void usb_fill_int_urb (struct urb *urb,  
                       struct usb_device *dev, unsigned int pipe,  
                       void *buffer, int buffer_length,  
                       usb_complete_t complete_cb, void *private,  
                       int period);
```

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

Le paramètre *iso_pkt* de `usb_alloc_urb()` indique le nombre de paquets isochrones à inclure dans l'URB. Le paramètre *flags* est identique à celui de `kmalloc()` (`GFP_KERNEL` ou `GFP_ATOMIC`).

On utilise les fonctions `usb_fill_xxx_urb()` pour initialiser les champs de la structure *urb* en fonction du type de dialogue :

- *urb* : structure à initialiser ;
- *dev* : périphérique associé ;
- *pipe* : numéro de canal associé à un *endpoint* (décrit plus loin).
- *setup_data_packet* : paquet de données d'initialisation (seulement URB *Control*) ;
- *buffer* : emplacement des données à transférer vers/depuis le périphérique ;
- *buffer_length* : longueur des données à transférer ;
- *complete_cb* : fonction *callback* invoquée lorsque l'opération sera terminée ;
- *private* : pointeur sur des données privées, récupérées dans la fonction *callback* ;
- *periode* : période en ms (1/8 de ms en *High-Speed*) de soumission d'un URB *Interrupt*.

L'initialisation des URB *Isochronous* est plus compliquée, il faut remplir directement les champs de la structure.

Le *pipe* que l'on fournit à `usb_fill_xxx_urb()` regroupe le numéro du *endpoint*, son type et son sens d'utilisation. On obtient le *pipe* à l'aide de l'une des fonctions suivantes :

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

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

Une fois l'URB prêt, on le soumet au sous-système USB Core :

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

Le paramètre `flags` de `usb_submit_urb()` doit contenir `GFP_KERNEL` ou `GFP_ATOMIC`.

Travaux pratiques : écriture sur un device USB

La carte Velleman 8055 utilisée dans ces exemples emploie un protocole non documenté pour dialoguer avec l'hôte. Toutefois un peu de *reverse engineering* sur le driver fourni (pour Windows) a permis d'extraire le format d'une trame de dialogue :

Pour écrire sur les sorties du périphérique on doit envoyer les huit octets suivants :

- La valeur 5 : probablement un numéro de commande ;
- Un octet correspondant à l'état désiré des huit sorties numériques ;
- Un octet contenant l'état désiré de la première sortie analogique ;
- Un octet contenant l'état désiré de la seconde sortie analogique ;
- Quatre octets à zéro.

Il existe probablement d'autres commandes, par exemple pour remettre à zéro les compteurs d'impulsion.

Étudiez le code source du module `example-VII-03.c`. Chargez le module. Voyez-vous le fichier spécial dans `/dev` ? Écrivez sur les sorties de la carte avec des commandes :

```
$ echo 5 85 00 255 0 0 0 0 > /dev/velleman0
$ echo 5 170 00 255 0 0 0 0 > /dev/velleman0
```

A vous...

Essayez des écritures rapides (avec des boucles du shell). Quel problème se pose ? Pourquoi ?

Dans le module `example-VII-04.c`, une variable entière indique si une requête de sortie est en cours, et une file d'attente est créée pour endormir les appels-système `write()` survenant avant la fin de la requête précédente.

Afin de traiter également les risques de déconnexions intempestives du périphérique, les attentes sont temporisées pour abandonner les opérations automatiquement au bout d'une seconde.

Vérifiez que la correction apportée dans `example-VII-04.c` fonctionne correctement.

Travaux pratiques : lecture d'un device USB

Lors d'une interrogation en lecture, la carte nous envoie (en mode *Interrupt*) les huit octets suivants :

- Un octet correspondant à la valeur des cinq entrées numériques ;
- Un octet représentant le numéro d'index de la carte (configurable entre 0 et 3 par un *jumper*) ;
- Un octet contenant l'état de la première entrée analogique ;
- Un octet contenant l'état de la seconde entrée analogique ;
- Deux octets (poids faible d'abord) représentant le nombre d'impulsions reçues sur l'entrée numérique 1 ;
- Deux octets comptant le nombre d'impulsions reçues sur l'entrée numérique 2 ;

La lecture des données est asynchrone, on soumet une requête *URB*, et la fonction *callback* associée sera invoquée quand les données seront prêtes.

Observez `exemple-VII-05.c`, puis chargez-le. Essayez de lire sur son fichier spécial ainsi :

```
# cat /dev/vl1e1an0
```

Exemples de drivers pour autres modes

Endpoint Bulk et Control

Pour les communications en mode *Bulk* ou *Control* il n'est pas toujours nécessaire d'utiliser un *URB*. Ces modes nécessitant obligatoirement un compte-rendu (réussite ou échec) de la communication, on utilise plutôt les routines synchrones suivantes :

```
int usb_bulk_msg (struct usb_device * dev,
                  unsigned int pipe,
                  void * data, int len,
                  int * len_return,
                  int timeout);
```

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

Les arguments de la fonction `usb_bulk_msg()` sont :

- `dev` et `pipe` : périphérique et canal à utiliser ;
- `data` et `len` : données à transférer et leur taille ;
- `len_return` : taille des données effectivement transférées ;
- `timeout` : durée maxi, en ticks.

Les arguments de `usb_control_msg()` sont les suivants :

- `dev` et `pipe` : périphérique et canal à utiliser ;
- `request` et `request_type` : le numéro et le type de requête ;
- `value` et `index` : argument et adresse précisant la requête ;
- `data` et `size` : emplacement et taille des données à transférer ;
- `timeout` : délai en ticks.

Les spécifications USB mentionnent des requêtes standards mais il est possible d'utiliser des requêtes personnalisées. Le paramètre *request_type* décrit le sens, le destinataire et le genre (standard ou personnalisé) de la requête.

A vous...

Vous pouvez observer des drivers USB dans les sources du noyau, notamment :

`linux/usb/serial/usb-serial.c` (driver de convertisseur USB / RS232)

qui utilise quatre *endpoints* :

- deux (*in* et *out*) de type *bulk* pour le transfert des données
- deux (*in* et *out*) de type *control* pour le paramétrage de la ligne (vitesse, parité...).

La partie bas-niveau de la communication se trouvera par exemple dans :

`linux/usb/serial/pl2303.c`.