Développez vos pilotes de périphériques USB

Les périphériques USB se sont largement répandus : scanner, appareil photo, disques durs externes… Cette connectique offre en effet de nombreux avantages (configuration "à chaud", débit de 480 Mbits/s pour la version USB2.0) et semble à présent incontournable.
Cet article est une introduction à la gestion de l'USB sous Linux et à l'écriture de pilotes USB. Il s'inscrit dans la lignée des articles déjà parus dans ce même journal en mai 2000 (introduction à l'écriture driver) et septembre 2002 (écriture de pilotes PCI).

Après quelques précisions sur les spécifications USB (version 1.1), nous décrirons l'interface USB présentée par le noyau et détaillerons l'écriture d'un mini-driver nous permettant de lire directement les donnés d'une souris USB.
Connaissances et matériel requis :

- Langage C
- Une souris USB
- Noyau 2.4.x

Aperçu de l'architecture USB et concepts clefs

Un ordinateur ne comporte qu'un seul contrôleur USB implanté sous forme matérielle et logicielle. Cet unique contrôleur intègre un hub racine fournissant un ou plusieurs points d'attachements.
Les périphériques USB sont de deux sortes :

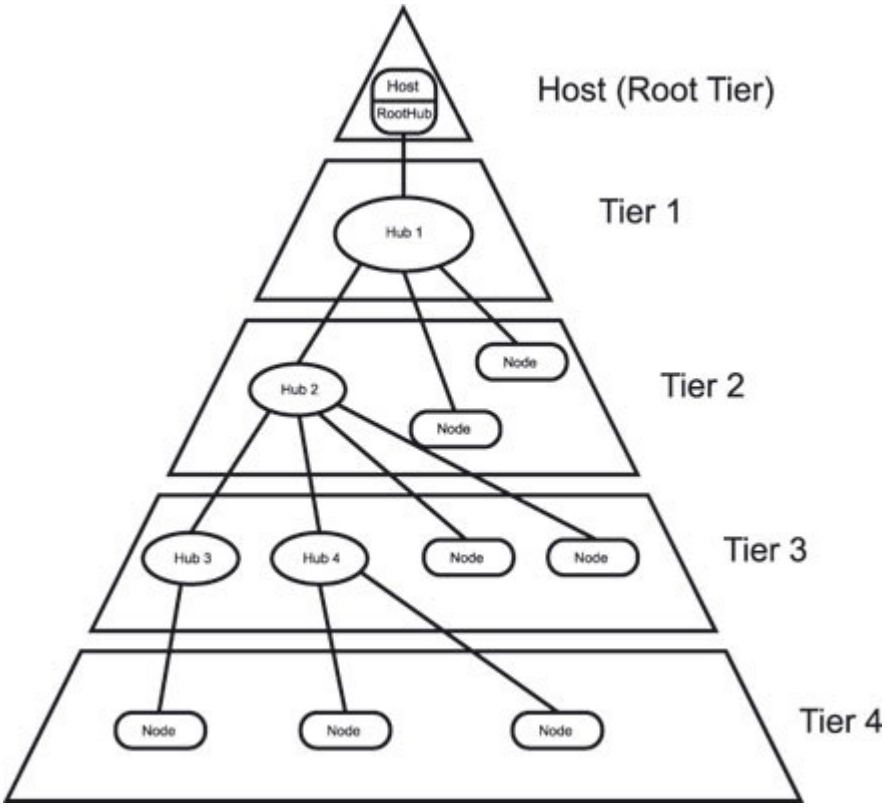
- Les hubs offrant de nouveaux points d'attachements ;
- Les "fonctions" offrant de nouvelles fonctionnalités : scanner, souris...

Tous présentent la même interface USB au travers de :

- Leur compréhension du protocole USB ;
- Leur réponse aux opérations USB standards : configuration, reset... ;
- Leur manière de se décrire.

Topologie du bus

Topologie physique : l'USB connecte des périphériques USB à un hôte USB. L'interconnexion physique décrit une topologie en étoile sur plusieurs niveaux.



Un hub est au centre de chaque étoile.
Topologie logique : L'hôte communique avec chaque périphérique comme s'il était directement connecté au hub racine.

Support physique

L'USB transfère signal et alimentation dans un câble à quatre fils. Il autorise deux taux de transfert : full-speed (12 Mb/s) et low-speed (1,5 Mb/s) pour les périphériques ne nécessitant pas une large bande passante, comme les souris par exemple.

Flux de communication

Les communications USB se déroulent au travers de deux interfaces logicielles :

- Host Controller Driver (HCD), couche logicielle permettant de faire abstraction du contrôleur USB matériel. Elle fournit une SPI (System Programming Interface) permettant d'interagir avec le contrôleur USB, et de cacher les spécificités de l'implantation matérielle.
- USB Driver (USBD), le pilote USB qui fournit les fonctionnalités propres au périphérique.

Périphérique logique

Un périphérique USB apparaît au système comme une collection de "endpoints". Ce sont d'uniques parties adressables du périphérique et sont sources ou cibles du flux de communication entre le périphérique et l'hôte. Chaque "endpoint" est caractérisé par un identifiant unique (fixé par le constructeur) appelé endpoint number et constitue une connexion supportant un flux de données du périphérique vers l'hôte ou inversement.
Chaque endpoint possède une direction de flux prédéterminée (IN/OUT) et différents paramètres décrivant les caractéristiques des transferts qu'il est susceptible de supporter :

- Temps de latence et fréquence d'accès au bus ;
- Bande passante ;
- Endpoint address ;
- Gestion d'erreur ;
- Taille maximale des paquets qu'il peut recevoir et envoyer ;
- Type de transfert ;
- Direction des transferts.

Les "pipes" sont l'abstraction logique représentant l'association entre un "endpoint" et l' USBD. Les endpoints sont regroupés en "endpoints sets" pour former des interfaces. Une interface peut être assimilée à une vue du périphérique.
Chaque périphérique USB implémente de manière obligatoire une méthode de contrôle par défaut utilisant un endpoint particulier possédant le numéro 0. A cet endpoint est associé un tube de contrôle par défaut (Default Control Pipe), qui fournit un accès aux informations de configuration du périphérique. Ce tube supporte des transferts de contrôle permettant de configurer le périphérique. Les endpoints de numéro 0 sont accessibles dès que le périphérique est attaché au bus. Sous Linux, ces informations sont lisibles soit directement via /proc/bus/usb/devices, soit par des outils tels que lsusb (mode texte) ou usbview (mode graphique).

Types de transfert

L'USB définit quatre types de transfert, optimisés en termes de temps de latence, contraintes de taille… suivant le type de service demandé.

- Transfert de contrôle : ponctuel, non périodique, à l'initiative de l'hôte, utilisé typiquement pour les opérations de configuration et de contrôle de statut ;
- Transfert synchrone : périodique, continu et pouvant nécessiter une certaine bande passante (utilisé pour une camera USB par exemple) ;
- Transfert d'interruption : données de petite taille, faible temps de latence, faible fréquence (utilisé pour une souris USB par exemple) ;
- Bulk transfert : large quantité de données, non périodique, ne nécessitant pas une bande passante prédéterminée (utilisés pour un scanner USB par exemple).

La SPI

Un pilote USB utilise uniquement l'interface de programmation fournie par le HCD pour gérer un périphérique, contrairement à un pilote PCI, PCMCIA…, qui utilise des adresses mémoires ou des ports d'E/S. Sous Linux, l'interface est fournie par les modules **usb_uhci** (ou **usb_ohci** suivant le type de contrôleur) et usbcore. Les structures et fonctions que nous allons utiliser sont décrites dans le fichiers d'en-tête **linux/usb.h**.
Au niveau du code en C, nous utiliserons les structures, fonctions et macros suivantes :

```
struct usb_device
```

Une telle structure est allouée par le HCD à chaque branchement d'un nouveau périphérique USB sur le bus. Elle contient toutes les informations concernant la structure logique du périphérique : identifiant vendeur, identifiant constructeur, endpoints… Toutes les informations fournit par **usbview** sont accessibles via cette structure, qui est transmise au pilote afin de lui permettre de

reconnaître dans un premier temps le périphérique qu'il va prendre en charge puis de communiquer avec.

```
static void *probe (struct usb_device *udev,unsigned ifnum, const struct
usb_device_id *prod)
```

Cette fonction est appelée à chaque nouvel attachement d'un périphérique au bus USB. L'appel de cette fonction est réalisé avec la structure **usb_device** décrite précédemment, correspondant au nouveau périphérique. Le rôle de cette fonction est de vérifier les informations de configuration du périphérique et de s'attacher le périphérique le cas échéant. L'attachement du pilote au périphérique est réalisé si la fonction retourne un pointeur non nul (généralement un pointeur sur une structure de données spécifique au pilote et allouée au sein de celui-ci).

```
static void disconnect(struct usb_device *udev, void *ptr)
```

Fonction appelée lors du débranchement du périphérique. Son rôle est la libération des ressources mobilisées par le driver. Ptr est un pointeur sur la structure de données retournée par probe.

```
struct usb_driver
```

Structure représentant le driver USB. Contient le nom du driver, un pointeur sur les fonctions probe, disconnect et sur une structure file_operations classique (stockant les différentes fonctions implémentées pour le périphérique : lecture, écriture…)

```
usb_register( * struct usb_driver ), usb_deregister (*struct usb_driver)
```

Permet d'enregistrer (et de dé-enregistrer) le pilote auprès du HDC.

```
struct urb ( pour USB Request Block)
```

Pour échanger des informations avec le périphérique, nous avons besoin de dire au sous-système USB comment communiquer. Cette tâche est accomplie en remplissant une structure urb et en la passant à la fonction **usb_submit_urb**.

```
struct urb * usb_alloc_urb(int iso_frames)
```

Pour allouer une structure URB.

```
usb_rcvintpipe(* struct usb_device udev ,int endpointadresse)
```

Pour créer un pipe supportant le transfert de type interruption entre le périphérique logique udev et le endpoint d'adresse endpointadresse.

```
FILL_INT_URB( struct urb *purb, struct usb_device *udev, int pipe,
              (void*) tampon, int size, (* fonction_de_rappel),
               void *context), int interval)
```

Cette macro permet de remplir une structure **urb** qui spécifiera au HCD que le type de transfert sera "interruption". Le périphérique concerné est pointé par **udev**, l'**urb** utilisé pour les transfert de données est pointé par purb. pipe représente le pipe que l'on aura créé par la macro précédente. Fonction_de rappel est un pointeur vers une fonction de prototype **void fonction_de_rappel (struct urb *purb)**. Cette fonction sera appelée après chaque interruption spécifiant l'achèvement d'un transfert de données du périphérique vers l'hôte. (Elle doit notamment respecter les contraintes spécifiques au code appelé dans un contexte d'interruption). **Interval** représente l'intervalle en ms entre les interruptions. **tampon** est pointeur sur une zone de mémoire non "swappable" qui contient les données transférées depuis le périphérique. (Une version de cette macro existe pour chaque type de transfert : **FILL_BULK_URB**, **FILL_ISO_URB**…).

```
usb_inc_dev(struct usb_driver *dev), usb_dev_dev(struct usb_driver *dev)
```

Incrémente et décrémente le compteur d'utilisation des modules du HCD.
Les autres fonctions et structures utilisées ne sont pas spécifiques à l'USB :

```
MOD_INC_USE_COUNT, MOD_DEC_USE_COUNT
```

Incrémentation et décrémentation du compteur d'usage du module. (colonne used de lsmod). Empêche le déchargement d'un driver lorsque celui-ci est en cours d'utilisation.

```
wait_queue_head_t, init_waitqueue_head( wait_queue_head_t *p),
                                  interruptible_sleep_on(wait_queue_head_t *p),
                                  wake_up_interruptible (wait_queue_head_t *p)
```

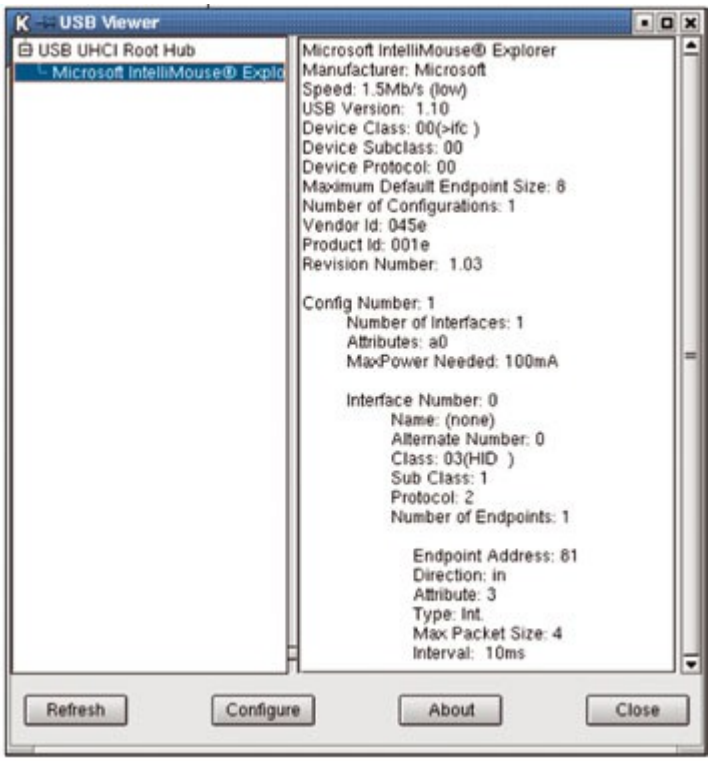
Structure et fonctions permettant d'endormir et de réveiller un processus sur une file d'attente.

```
module_init, module_exit
```

Fixe les fonctions d'initialisation et de destruction du driver.

Notre premier driver

Ces précisions étant faites, nous allons pouvoir passer à l'écriture proprement dite de notre pilote de souris USB.
Au niveau des fonctionnalités, notre pilote implémentera seulement une lecture en mode bloquant. La première tâche à effectuer est la découverte des informations de configuration de la souris USB dont on veut écrire le pilote. Je branche ma souris sur un port USB. usbview me donne alors les informations :



usbview nous indique l'identifiant vendeur et l'identifiant constructeur, ce qui permettra à la fonction probe de reconnaître le périphérique que nous voulons prendre en charge. On apprend aussi que l'on communiquera avec le endpoint d'adresse **0x81**, sur l'interface zéro avec un type de transfert **interruption**, une taille maximale de paquet de quatre octets, etc.
Pour stocker toutes les données nécessaires au pilote, on utilisera notre propre structure : Struct LMAG_priv. Cette structure sera dynamiquement allouée à la fonction d'initialisation du pilote.
L'implémentation de la lecture bloquante est réalisée comme suit :

- Un processus qui veut lire des données provenant du périphérique est mis en sommeil sur une file d'attente.
- Une interruption est déclenchée lorsque les données provenant du périphérique sont accessibles. Cette interruption est prise en charge par un gestionnaire d'interruption dont le rôle est d'ordonnancer une tâche permettant de réveiller, en dehors de ce contexte d'interruption, les processus endormis sur la file d'attente. Ceci est réalisé par l'intermédiaire d'une structure tasklet.
- Lorsqu'un processus lecteur est réveillé, cela signifie que des données provenant de la souris sont accessibles. Il ne reste alors qu'à les rendre accessibles dans le contexte de l'utilisateur.

L'accès au périphérique sera effectué grâce à un fichier spécial de numéro majeur 183 (pour tous les périphériques USB) créé par :

```
mknod  LMAG_mouse c 183 251
```

par exemple (si le numéro mineur 251 n'est pas utilisé par un autre périphérique USB).

CODE SOURCE

```
#include<linux/module.h>
#include<linux/kernel.h>
```

```
#include<linux/init.h>
#include<linux/usb.h>
#include<linux/malloc.h>
#include<linux/sched.h>
#include<asm/uaccess.h>

struct LMAG_priv { /** structure de données privées **/
        char mess[20] ;/** pour stocker les données de la souris **/
        struct urb  *lmagurb;
        struct usb_device *udev;
        wait_queue_head_t readwait; /** file d'attente lecture **/
        int plugged; /** souris branchée **/
        int opened; /** souris ouverte **/
};

 static struct LMAG_priv *ppriv =NULL;

 void LMAG_interruption2( unsigned long param) {
        wake_up_interruptible(&(ppriv->readwait));}/** reveil des processus
lecteur **/

DECLARE_TASKLET(int_tasklet,LMAG_interruption2,0);

 void LMAG_interruption1 (struct urb *ptr) {/** fonction de gestion des
interruptions **/
        tasklet_schedule(&int_tasklet);}/** ordonnacement de la fonction de
gestion **/

static void *LMAG_probe (struct usb_device *udev,unsigned ifnum, const struct
usb_device_id *prod){
        printk(KERN_DEBUG" LMAG_driver: entrée dans probe\n");
        if(udev->descriptor.iManufacturer != 1 || udev->descriptor.idVendor !=
1118)
                return 0;
        usb_inc_dev_use(udev);
        if ((ppriv->lmagurb=usb_alloc_urb(0))==0) {
                printk(KERN_DEBUG" echeC ALLOCATION URB\n");}
        ppriv->udev=udev;
        (ppriv->plugged)++;
        return (ppriv);}

int LMAG_open(struct inode *inode,struct file *filp){
        int res;
        if(!ppriv->plugged==0) return -ENODEV;
        if (ppriv->opened==0 ){/** installation du mode de communication par
d'interruption     ******/
                FILL_INT_URB(ppriv->lmagurb,
                        ppriv->udev,
                  usb_rcvintpipe(ppriv->udev,0x81),
                        (void*) ppriv->mess,
                        4,
                        LMAG_interruption1,
                        ppriv,
                  0x10);
                if ((res=usb_submit_urb((ppriv->lmagurb)))!=0) {
                        printk(KERN_DEBUG" LMAG_driv:Unable to allocate INT
URB.erreur :%i",res);
                        return -1;}
                else printk(KERN_DEBUG" LMAG_driv: allocate INT URB.ok:\n");}
        MOD_INC_USE_COUNT ;
        (ppriv->opened)++;
        return 0;}
```

```
static void LMAG_disconnect(struct usb_device *udev, void *ptr){
        int res;
        printk(KERN_DEBUG" LMAG_driver: entrée dans disconnect\n");
        wake_up_interruptible(&(ppriv->readwait));
        (ppriv->plugged)--;
        if(ppriv->opened !=0 ) {
                        if((res=usb_unlink_urb(ppriv->lmagurb))!=0)
                                printk(KERN_DEBUG" LMAG_driv: echec unlink
urb:\n");}
        ppriv->opened=0;
        usb_free_urb(ppriv->lmagurb);
        usb_dec_dev_use(udev);}

int LMAG_release(struct inode *inode,struct file *filp){
        printk(KERN_DEBUG" LMAG_driver: entrée dans release\n");
        ppriv->opened--;
        if(ppriv->opened ==0)
                usb_unlink_urb(ppriv->lmagurb);
        MOD_DEC_USE_COUNT;
        return 0;}

ssize_t LMAG_read(struct file *fp,char * buf, size_t count,loff_t *pos){
        interruptible_sleep_on(&(ppriv->readwait));
        if(copy_to_user(buf,ppriv->mess,ppriv->lmagurb->actual_length))
                return -EFAULT;
        return (ppriv->lmagurb->actual_length);}

struct file_operations LMAG_fops= {
        open:LMAG_open,
        release:LMAG_release,
        read:LMAG_read};

static struct usb_driver LMAG_usb={
        name : "LMAG_usb",
        probe: LMAG_probe,
        disconnect:LMAG_disconnect,
        fops:&LMAG_fops,
        minor:251};

int LMAG_init (void) {
        if(!(ppriv=(struct LMAG_priv*)kmalloc(sizeof *ppriv,GFP_ATOMIC))) {
                printk(KERN_DEBUG" echeC ALLOCATION LMAG_priv\n");
                return -1;}
        if(usb_register(&LMAG_usb) <0){
                 return (-1);   }
        ppriv->opened=0;
        ppriv->plugged=0;
        init_waitqueue_head (&(ppriv->readwait));
        printk(KERN_DEBUG" LMAG_driver: enregistrement du driver OK\n");
        return 0;}

void LMAG_cleanup (void) {
        printk(KERN_DEBUG" LMAG_driver: liberation du pilote\n");
        kfree(ppriv);
        usb_deregister(&LMAG_usb);}

module_init(LMAG_init);
module_exit(LMAG_cleanup);
```

```
/***********fin du driver*******************/
```

Le makefile

```
KERNELDIR=/usr/src/linux

include ${KERNELDIR}/.config

CFLAGS=-D__KERNEL__ -DMODULE -I${KERNELDIR}/include -O -Wall

ifdef CONFIG_SMP
 CFLAGS+=-D__SMP__ -DSMP
endif

all:LMAG_driver.o

LMAG_driver.o: LMAG_driver.c
        gcc ${CFLAGS} -c LMAG_driver.c -o LMAG_driver.o

clean :
        rm -f LMAG_driver.o
```

Fonctionnement

Après compilation (utiliser le **Makefile**), nous obtenons le fichier LMAG_driver.o, code objet du pilote de notre souris. On crée le fichier spécial **LMAG_mouse** (ou tout autre nom) par la commande

```
mknod  LMAG_mouse c 183 251
```

Pour éviter qu'un driver pré-existant soit directement chargé par le système lors du branchement de la souris, il faut "killer" le démon usbd s'il est présent sur la machine (Killall usbd).
On charge le driver par **insmod LMAG_driver.o** et on le décharge par **rmmod LMAG_driver**. On peut indifféremment brancher la souris avant de charger le driver ou charger le driver puis brancher la souris. Si la souris USB est branchée, on peut lire les données qu'elle envoie en ouvrant le fichier spécial **LMAG_mouse**. Par exemple :

```
Od –x –v -w4 < LMAG_mouse
```

affiche par groupe de quatre octets en hexadécimal les données de la souris.
La commande dmesg permet de voir les différents messages de débogage envoyés par le pilote et de tracer le passage dans les différentes fonctions.

REMARQUES IMPORTANTES :

- Afin de le simplifier, le driver a été écrit spécifiquement pour la souris que j'ai utilisée (une Microsoft IntelliMouse Explorer) puisque ses données de configurations (**idvendor**,

endpointaddress…) ont été codées en dur dans le driver. Un véritable driver devrait lire les informations de configuration (comme usbview) et s'y adapter. Si vous en avez compris le fonctionnement, vous pouvez toutefois l'adapter sans peine à votre souris USB.

- Un véritable pilote de souris devrait travailler les données provenant de la souris pour les présenter selon un protocole prédéfini (**man mouse** pour plus de détails).