Programmation et développements autour d'un microcontrôleur USB

Simon Guinot, Jean-Michel Friedt

association Projet Aurore, 11 Juin 2004

Transparents disponibles à

`http://projetaurore.assos.univ-fcomte.fr/USB`

Aspects matériels

Constat :
le port série (RS232) est voué à disparaître
le port série est trop lent pour certaines applications

Nécessité de passer à USB par souci de portabilité (laptops, Apple, PDAs) et de recherche de débits plus élevés

$\rightarrow$ recherche d'un microcontrôleur capable de communiquer par USB, ne nécessitant pas de programmeur dédié, avec assembleur/compilateur opensource.

Le microcontrôleur Motorola 68HC908JB8

Noyau 6808 (assembleurs `asxxxx`/compilateurs disponibles)
`http://shop-pdp.kent.edu/ashtml/asxget.htm` (DOS/linux) et
`http://www.imagecraft.com/software/demos.html` (Windows)

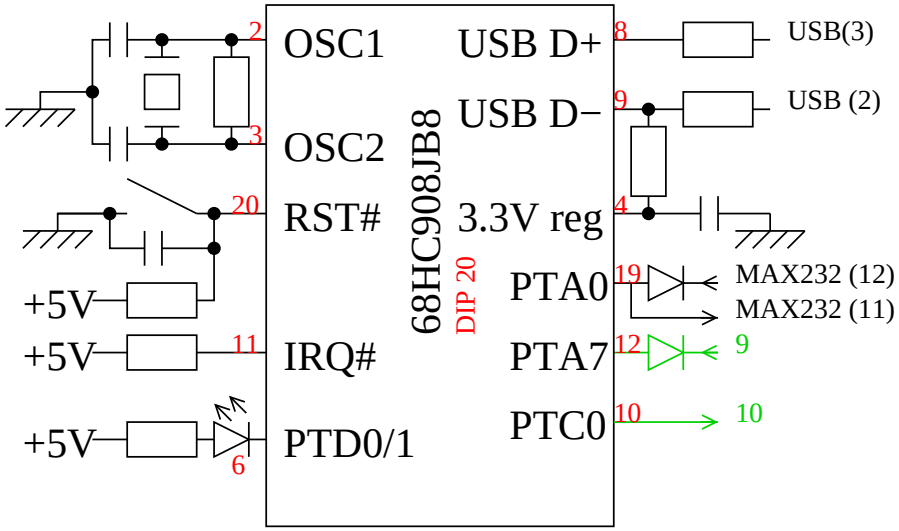
Mémoire FLASH (8 KB), RAM (256 B), package SOIC28 ou DIL20, quartz 6 MHz

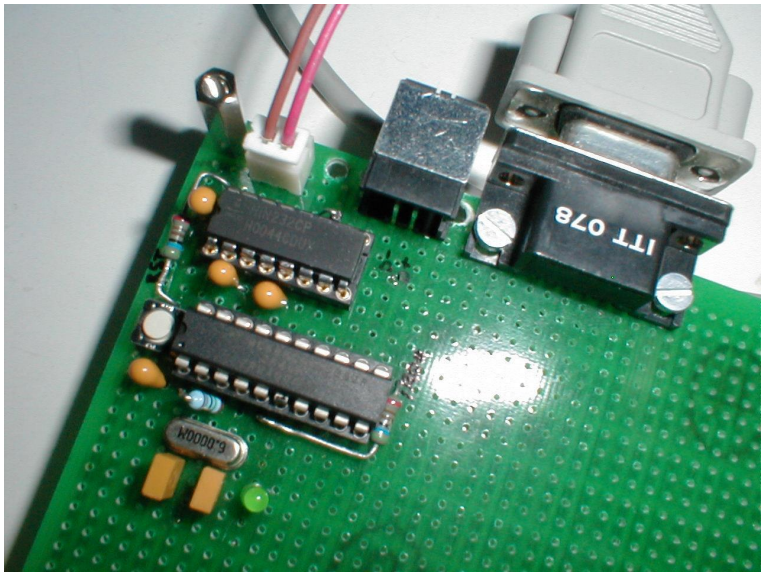
Pas de conversion analogique-numérique

L'USB apparaît à l'utilisateur comme un UART un peu plus compliqué

Circuit électronique

Simplicité de mise en œuvre





Deux ports RS232, un port USB, 1 diode, 8 pins pour clavier (IRQ), RESET, NMI, quartz (8 diodes pour le SOIC28)

La programmation du 68HC908

Moniteur communiquant par RS232

Commandes : write (0x49), execute (0x28), adresse de la pile (0x0C)
un programme pour bulk erase

Un exemple simple : la diode qui clignote ...

Sans même savoir ce qu'est un driver, on peut vérifier si le client USB fonctionne ... (identification du microcontrolleur)

Avec le code de `http://www.elektronikladen.de/en_usb08.html` (30% de la flash) :

```
$ tail -f /var/log/syslog
cheyenne1 kernel: Manufacturer: MCT Elektronikladen
cheyenne1 kernel: Product: USB08 Evaluation Board
cheyenne1 kernel: usb.c: unhandled interfaces on device
cheyenne1 kernel: usb.c: USB device 31 (vend/prod 0xc70/0x0) is
not claimed by any active driver.
```

Couche matérielle de USB

Signal à 3.3 V

Bus $\Rightarrow$ assignation dynamique d'adresse aux périphériques additionnelles

Encapsulation des données dans des paquets

Notion de *endpoint* (les "ports" en UDP sur ethernet)

Pourquoi développer sous linux ?

Disponibilité des sources à `http://www.kernel.org`
$\Rightarrow$ pas de reverse-engineering de binaires existant

Distribution libre des codes produits

Documentation plus ou moins disponible selon les sujets

On ne paie que pour le matériel, pas pour le logiciel (outils de développement)

La structure de fichier sous unix

Sous unix, tout est fichier (incluant l'accès aux périphériques)

Un répertoire dédié : `/dev`

Dans `/dev` : un major number (pilote) et un minor number (périphériques utilisant le même pilote) définissent l'accès au périphérique

```
jmfriedt@tml2:~$ ls -l /dev/ttyS*
crw-rw----    1 root     dialout    4,  64 2002-03-14 22:51 /dev/ttyS0
crw-rw----    1 root     dialout    4,  65 2002-03-14 22:51 /dev/ttyS1
crw-rw----    1 root     dialout    4,  66 2002-03-14 22:51 /dev/ttyS2
crw-rw----    1 root     dialout    4,  67 2002-03-14 22:51 /dev/ttyS3
```

De l'utilité du driver

Communication entre le noyau et l'utilisateur

Couche d'abstraction du matériel (portabilité)

Fournir à l'utilisateur des méthodes plus faciles à utiliser que réinventer la roue

Notion de module : code indépendant du noyau qui sait lui communiquer pour y ajouter ses fonctionnalités (souplesse au développement, économie de mémoire). Existe depuis les kernel 2.x sous linux.

Structure d'un driver

Les méthodes :

1. `write()` (ex. : jouer un son, `cat /bin/ls > /dev/audio`)
2. `read()` (ex. : acquisition d'un son)
3. `ioctl()` (ex. : controler le format audio, la vitesse d'échantillonnage ...)
4. `open()` et `close()` (ex. : initialiser et quitter la carte son)

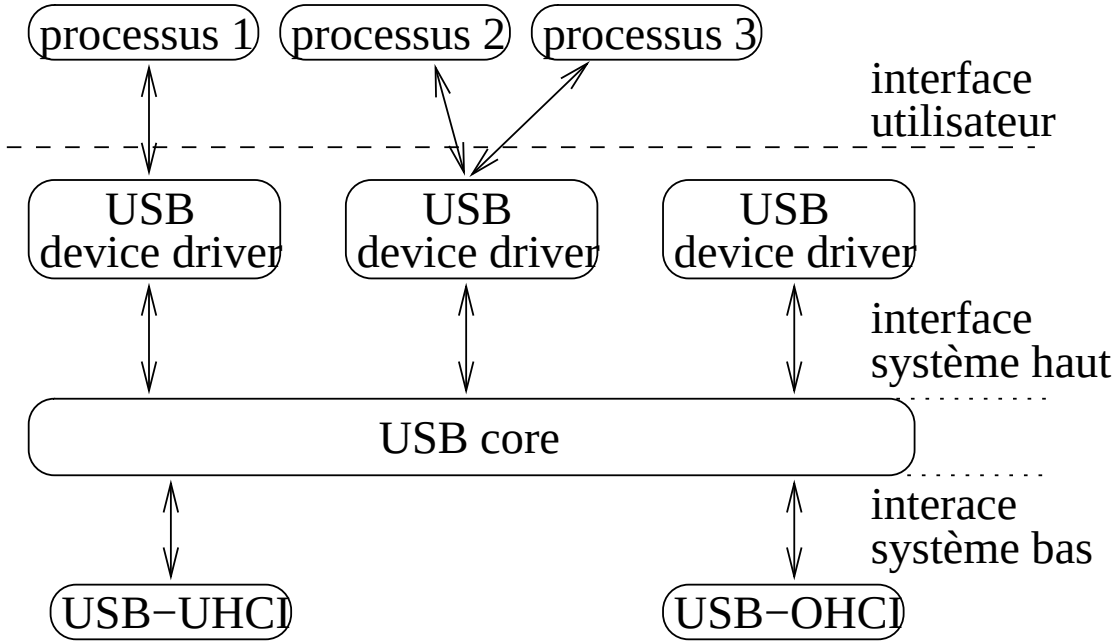
Exemple d'ioctls dans `/usr/include/linux/soundcard.h` :

```
#define SNDCTL_DSP_SPEED          _SIOWR('P', 2, int)
#define SNDCTL_DSP_STEREO         _SIOWR('P', 3, int)
#define SNDCTL_DSP_SAMPLESIZE     SNDCTL_DSP_SETFMT
#define SNDCTL_DSP_CHANNELS       _SIOWR('P', 6, int)
```

Les modules nécessaires à la communication USB

`usb-uhci` ou `usb-ohci` fournissent l'interface avec le controleur USB (le maître) (ne *pas* utiliser `uhci.o` !)

`usbcore` définit l'interface avec les autres modules (notamment le notre !)



Syntaxe de la définition des méthodes

Les opérations possibles : interaction avec le système de fichiers

```
static struct file_operations hc08_fops = {
        owner:          THIS_MODULE,
        read:           hc08_read,
        write:          hc08_write,
        ioctl:          hc08_ioctl,
        open:           hc08_open,
        release:        hc08_release,
};
```

Exemple d'enregistrement d'un character device :

```
if ((hc08_major = register_chrdev (hc08_major, "hc08", &hc08_fops)) < 0)
```

Syntaxe de la définition des méthodes (2)

Identification : enregistrement du pilote auprès du système USB (`usbcore`)

```
static struct usb_driver hc08_driver = {
        name:           "hc08",
        probe:          hc08_probe,
        disconnect:     hc08_disconnect,
        fops:           &hc08_fops,
        minor:          USB_HC08_MINOR_BASE,
        id_table:       hc08_table,
};
```

Exemple d'enregistrement d'un périphérique USB :

```
result = usb_register(&hc08_driver);
```

Spécificité de USB : la méthode `probe`.

La communication avec le microcontrôleur

Le périphérique USB s'identifie par un paire d'indices uniques :

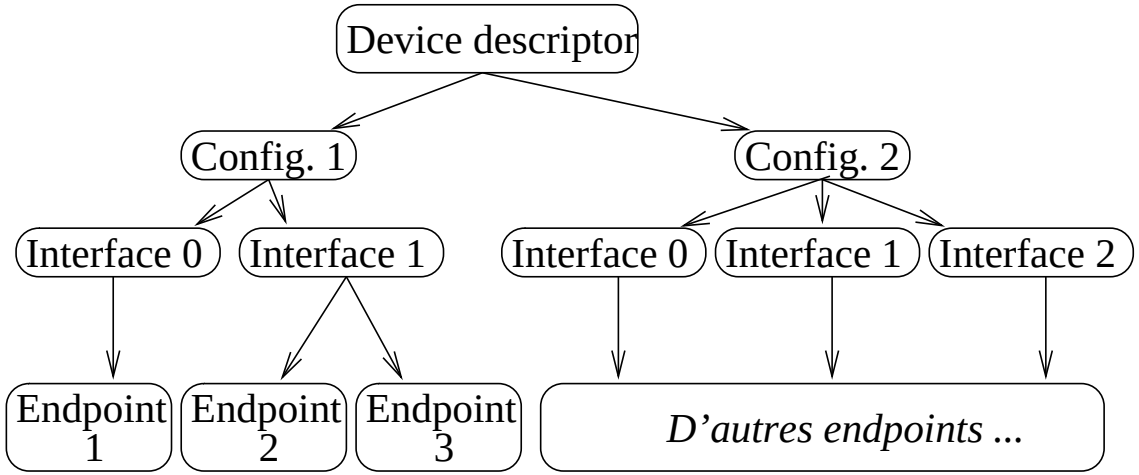
```
#define USB_HC08_VENDOR_ID      0x0c70
#define USB_HC08_PRODUCT_ID     0x0000

static struct usb_device_id hc08_table [] = {
        { USB_DEVICE(USB_HC08_VENDOR_ID, USB_HC08_PRODUCT_ID) },
        { }
};
```

Liste connectant les identifiants de périphériques avec les drivers associés (lue à toute nouvelle connexion).

La méthode `probe`

Appelée lorsqu'il y a correspondance avec les indices dans la table des identifiants → prend-on en charge ce périphérique ?

```
static void * hc08_probe (struct usb_device *udev, unsigned int ifnum,
                          const struct usb_device_id *id)
  interface = &udev->actconfig->interface[ifnum];
```



Le 68HC908 fournit

- un port de contrôle (endpoint 0) : bidirectionnel, buffer 8 B I/O
- un port en émission (endpoint 1) : buffer 8 B O
- un port bidirectionnel (endpoint 2) : buffer 8 B **I**/O

⇒ on a ici une interface *lente* [4, p.126]

La méthode `probe` (2)

Dans `/usr/src/linux-2.4.22/include/linux/usb.h` : `usbcore` renvoie la configuration du nouveau périphérique

```
struct usb_interface_descriptor {
  __u8  bNumEndpoints     __attribute__ ((packed));
  struct usb_endpoint_descriptor *endpoint;
  ...

struct usb_endpoint_descriptor {
 __u8  bEndpointAddress  __attribute__ ((packed));
 __u8  bmAttributes      __attribute__ ((packed));
 __u16 wMaxPacketSize    __attribute__ ((packed));
 __u8  bInterval         __attribute__ ((packed));
 ...
```

... pour en arriver à :

```
if ((endpoint->bEndpointAddress & 0x80) && ((endpoint->bmAttributes & 3)==0x03))
    /* we found a interrupt in endpoint */
```

Les endpoints et les modes de communication

Endpoint 0 est *toujours* le port lié au contrôle du port USB (c'est par là que le maître donne une adresse à l'esclave lorsqu'il se connecte)

Communication par paquets de 8 (lent), 16, 32 ou 64 octets (rapide)

- **Interrupt** (débit constant, ré-essaie jusqu'à succès, paquets$<$ 64 B)
- Isochronous (débit constant, latence bornée, pas de corrections)
- Bulk (prend toute la bande passante, garantit le transfert mais pas la latence)

Notion de URB (USB Request Block) : le paquet contenant les transactions à effectuer *via* USB (voir `/usr/src/linux/Documentation/usb/URB.txt`)

Un URB est soumis par `usb_submit_urb(urb)` tel que décrit dans ce même document.

La communication avec le microcontrôleur : la lecture

Dans `open`

```
static int hc08_open (struct inode *inode, struct file *file)
{...
 FILL_INT_URB (dev->read_urb, dev->udev,
      usb_rcvintpipe (dev->udev, dev->int_in_endpointAddr),
      dev->int_in_buffer, dev->int_in_size,
      hc08_read_int_callback, dev, dev->int_in_interval);
}
```

puis `usbcore` appelle à chaque lecture (gestion des URBs)

```
static void hc08_read_int_callback (struct urb *urb)
{...
 memcpy (dev->read_buffer_start + dev->read_buffer_size,
 dev->int_in_buffer, urb->actual_length);
 ...
 wake_up_interruptible (&dev->int_in_wait);
```

La communication avec le microcontrôleur : la lecture (2)

La lecture est *bloquante*

```
static ssize_t hc08_read (struct file *file, char *buffer,
                          size_t count, loff_t *ppos)
{ [...}
 while (dev->read_buffer_size == 0)
 wait_event_interruptible (dev->int_in_wait, dev->read_buffer_size);
 bytes_reads=(count>dev->read_buffer_size)?dev->read_buffer_size:count;
 copy_to_user (buffer, dev->read_buffer_start, bytes_reads);
```

La communication avec le microcontrôleur : l'écriture

```
static ssize_t hc08_write (struct file *file, const char *buffer,
                           size_t count, loff_t *ppos)
{ [...]
 bytes_written=(count>dev->int_out_size)?dev->int_out_size:count;
 copy_from_user(dev->int_out_buffer, buffer, bytes_written);
 FILL_INT_URB (dev->write_urb, dev->udev,
               usb_sndintpipe (dev->udev, dev->int_out_endpointAddr),
               dev->int_out_buffer, dev->int_out_size,
               hc08_write_int_callback, dev, dev->int_out_interval);
 retval = usb_submit_urb (dev->write_urb);
```

La communication avec le programme utilisateur

Le driver est la partie difficile. S'il est bien fait, le reste est trivial :

```
while (1)
{trans=write (desc, data_write, SIZE); // ecriture
 memset (data_read, 0x00, SIZE);
 trans=read (desc, data_read, SIZE);   // lecture
 printf ("buffer : ");
 for (j=0; j<SIZE; j++) printf ("%x ", data_read[j] & 0xff);
 printf ("\n");
 if (*data_write) {memset (data_write, 0x00, SIZE);}
    else {memset (data_write, 0xff, SIZE);}
 sleep (1);
}
```

Échanges microcontrôleur-utilisateur via le driver

Programme utilisateur	Microcontroleur

`trans=write (desc, data_write, SIZE)`	`uchar getUSB() {`
	` while(RxBuf_RdIdx == RxBuf_WrIdx) ; // RxBuffer empty`
	` c = RxBuffer[RxBuf_RdIdx];`
	` RxBuf_RdIdx = (RxBuf_RdIdx+1) & (MAX_RXBUF_SIZE-1);`
	` return c;}`

`trans=read (desc, data_read, SIZE)`	`void putUSB(uchar c) {`
	` newIdx = (TxBuf_WrIdx+1) & (MAX_TXBUF_SIZE-1);`
	` while(newIdx == TxBuf_RdIdx) ; // TxBuffer full`
	` TxBuffer[TxBuf_WrIdx] = c;`
	` TxBuf_WrIdx = newIdx;}`

`if (*data_write)`
`  {memset (data_write, 0x00, SIZE);}`
`else {memset (data_write, 0xff, SIZE);}`
	`void main() {`
	`  while(1) {n=0;`
	`  do {io_buffer[n++] = getPipe();} while(n<8);`
	`  if(io_buffer[0]==0) offLED(1); else onLED(1);`
	`  io_buffer[0]=1; // ...`
	`  n=0;do {putPipe(io_buffer[n++]);} while(n<8);`

Difficultés rencontrées

On s'impose de garder le code USB fixe (limiter les risques d'erreur) $\Rightarrow$ contraintes liées aux particularités de *cette* implémentation du client (exemple HC908EVB Motorola)

$\longrightarrow$ toute lecture nécessite une écriture et réciproquement

Amélioration : devfs

Gestion automatique et “plus propre” de /dev (arborescence)

```
static int __init usb_hc08_init(void)
...
if (devfs)
   {devfs_dir = devfs_mk_dir (NULL, "hc08", NULL);}

static void * hc08_probe
...
if (devfs)
  {sprintf (name, "hc08_%d", dev->minor);
   dev->devfs = devfs_register (devfs_dir, name,
                    DEVFS_FL_DEFAULT, USB_MAJOR,
                    USB_HC08_MINOR_BASE + dev->minor,
                    S_IFCHR | S_IRUSR | S_IWUSR | S_IRGRP
                    | S_IWGRP | S_IROTH, &hc08_fops, NULL);
  }
```

et à la fin

```
static void hc08_disconnect
...
if (devfs) {devfs_unregister (dev->devfs);}
```

Limitations de notre approche

USB en mode interruption : un paquet de max. 64 octets toutes les 12 ms
$\Rightarrow$ 1.5 Mb/s

Flux constant de données plutôt que gros paquets par intermittence

Interrupt est typiquement utilisé pour les souris, claviers, contrôle de robots ...

Conclusion

Notre objectif était :

- de nous familiariser avec USB
- de développer l'ensemble des outils pour le 68HC908 sous Linux
- de développer un module Linux pour l'EVB USB du 68HC908

Perspectives :

- Développer notre propre code USB en assembleur Motorola (pas de C)
- Trouver un microcontroleur supportant le mode bulk et se programmant par FLASH

Le module présenté ici s'adapte facilement à toutes sortes d'applications.

Applications : lecture de MultiMediaCard, ajout d'un ADC

Passer en kernel 2.6.x

Références

[1] “Programming Guide for Linux USB Device Drivers”, disponible à `http://wwwbode.cs.tum.edu/Par/arch/usb/usbdoc/`

[2] “USB08 Universal Serial Bus Evaluation Board Using the MC68HC908JB8 disponible à `http://e-www.motorola.com/files/microcontrollers/doc/ref_manual/DRM002.pdf`

[3] “Universal Serial Bus Specifications (rev. 1.1)” à `http ://www.usb.org/developers/docs/`

[4] “MC68HC908JB8 Technical Data (rev. 1.0)” à `e-www.motorola.com/files/microcontrollers/doc/data_sheet/MC68HC908JB8.pdf`

[5] “Linux Device Drivers, 2nd ed.”, A. Rubini & J. Corbet, O’Reilly Ed. (2001)

et `http://friedtj.free.fr/hc08_eng.pdf`