Programmation et interfaçage d'un microcontrôleur par USB sous linux : le 68HC908JB8

Jean-Michel Friedt (friedtj@free.fr), Simon Guinot (simon@sequanux.org)
Association Projet Aurore, 11 novembre 2005

1 Introduction

Classiquement, les microcontrôleurs communiquent avec les ordinateurs personnels par le port série (RS232). Or la tendance actuelle de l'évolution du matériel informatique force aux constats suivants :

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

Nous nous sommes par conséquent fixés comme objectifs de maîtriser un microcontrôleur possédant un port USB [1, 2, 3], d'effectuer le développement totalement avec des outils libres sous GNU/Linux, et de développer un logiciel de base permettant de tester le microcontrôleur sous ce même système d'exploitation.

2 Le microcontrôleur Freescale 68HC908JB8

Nous avons sélectionné, parmi les rares microcontrôleurs possédant un port USB, le Freescale 68HC908JB8 selon les critères suivants :

- disponible chez les principaux revendeurs européens de matériel électronique [1],
- basé sur un noyau connu (le 6808) et donc avec des outils de développement OpenSource disponibles (assembleur `as6808` de l'ensemble d'outils `asxxxx` [2], et `asl`),
- et finalement ne nécessitant qu'un nombre minimum de composants passifs pour sa mise en œuvre (mémoire flash ne nécessitant pas de programmeur dédié).

Le 68HC908JB8 possède une mémoire FLASH (8 KB), RAM (256 B), un port USB 1.1, est disponible en packages CMS SOIC28 ou DIL20 et est cadencé par un résonateur à 6 MHz. Cependant, un inconvénient majeur est l'absence de convertisseur analogique-numérique qui limite ses applications en terme d'instrumentation.

Nous avons utilisé un circuit de test (Fig. 1), en partie inspiré de http://elmicro.com/hc08web/usb08/images/usb08_schema.gif, comprenant le strict minimum de composants passifs : une paire de diodes pour utiliser une broche comme port RS232 bidirectionnel, mise en œuvre du port USB, et les résistances pour la mise au niveau des signaux de RESET et interruption matérielle IRQ#. Ce montage nous permet déjà d'effectuer quelques tests préliminaires de programmation pour nous familiariser avec le protocole de communication, l'accès à la mémoire flash et les assembleurs disponibles sous Linux.

3 La programmation du 68HC908

Nous avons initialement (début 2001) recherché les outils disponibles en opensource pour programmer le 68HC908 pour n'identifier alors qu'un outil, `spgmr08` [3]. Cet outil, très complet, ne nous donnait pas l'opportunité d'appréhender en détail le fonctionnement du microcontrôleur et nous avons décidé d'écrire quelques petits utilitaires simples pour mettre en évidence les méthodes d'initialisation du microcontrôleur : il s'agit dans ce document de cette première partie, qui sera

[1] références 3480252, 3480264 et 4133020 chez Farnell ; référence 445-6744 chez Radiospares pour environ 6 euros/pièce

[2] http://shop-pdp.kent.edu/ashtml/asxget.htm ou alternativement installé sous Debian avec le package `sdcc` – http://sdcc.sourceforge.net/ – qui fournit aussi un compilateur C que nous n'avons pas testé. Dans ce dernier cas, l'assembleur s'appelle `as-hc08` au lieu de `as6808`

[3] http://sourceforge.net/projects/spgmr08/

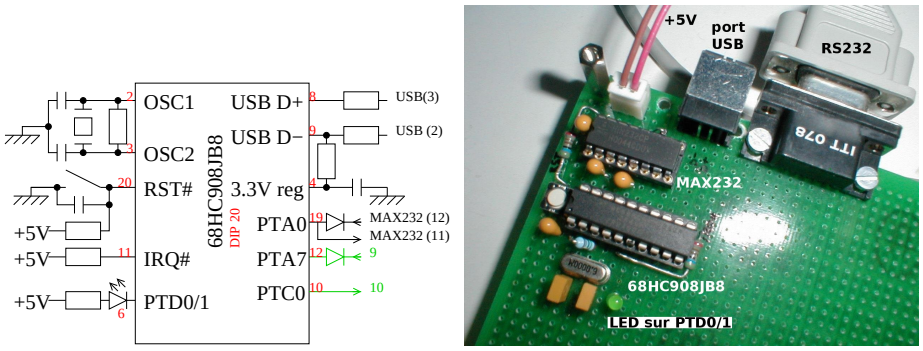

FIG. 1 – Gauche : schéma de principe pour faire fonctionner un 68HC908. On y retrouve les éléments fondamentaux que sont le résonateur à 6 MHz, les résistances définissant certains signaux de contrôle (reset RST#, interruption matérielle IRQ#), la gestion de l'USB sur les broches USB D+ et D- (la résistance entre D- et 3,3 V indique à l'hôte auquel se connecte le microcontrôleur que la communication USB est lente). Finalement quelques composants accessoires tel que les diodes connectées à PTA0 pour émuler un port de communication série et une LED sur PTD0/1 qui nous sera utile comme indicateur visuel de l'état du système. Noter que sur la version CMS du composant, PTD0 et PTD1 sont séparées (broches 6 et 7) alors que sur la version DIL utilisée ici ces deux signaux sont connectés en interne (broche 6). Droite : montage prototype sur plaque pré-percée illustrant qu'aucun matériel de réalisation de circuits imprimés spécifique n'est nécessaire pour exploiter ce microcontrôleur.

suivie dans une second temps par une exploitation sous GNU/Linux du code fourni en exemple par Freescale [4] pour la communication par bus USB.

Plus récemment, Benoît Demaine s'est intéressé à ce microcontrôleur dans le cadre d'une étude universitaire qu'il a documentée à http://stwp.demaine.info/, et a notamment écrit lui aussi un outil complet de programmation de ce microcontrôleur qu'il met à disposition à http://sourceforge.net/projects/monitor-68hc08/.

Nous allons ici décrire le mode de programmation de ce microcontrôleur afin de maîtriser totalement la chaîne de développement qui va se conclure par l'écriture en mémoire non-volatile d'un code de gestion de la communication par USB.

3.1 Premiers pas : le bootloader (mode MONITOR)

À l'allumage, le microcontrôleur lance un moniteur (présent en ROM), communiquant par RS232. Ce moniteur comprend un certain nombre de commandes nous donnant l'accès aux registres du microcontrôleur (dont les ports d'entrée/sortie), tel que décrit dans [5, section 10]. Ainsi, nous pourrons stocker puis exécuter en RAM un petit bout de code chargé de récupérer par RS232 les octets à placer en mémoire flash. Les commandes du bootloader que nous utilisons pour stocker un programme en mémoire flash sont (dans l'ordre d'utilisation) :

- écrire (0x49)
- exécuter (0x28)
- lire et définir l'adresse du pointeur de programme (0x0C)

Une action préliminaire à l'envoi d'ordres au bootloader est de s'identifier par la transmission d'un code de sécurité [5, p.175]. Ce code de sécurité est la séquence des octets stockée dans les vecteurs d'interruption situés en 0xFFF6-0xFFFD. Sur un microcontrôleur neuf, ces valeurs sont toutes égales à 0xFF donc il nous suffit d'initialiser la communication en transmettant une séquence de 8 octets égaux à 0xFF. Ultérieurement, lorsque ces vecteurs d'interruption auront été modifiés (ce sera sûrement le cas puisqu'ils comprennent un timer, une interruption matérielle et l'interface de communication USB) il faudra prendre soin d'envoyer la bonne séquence d'octets (les nouvelles

adresses des vecteurs d'interruption) faute de quoi le mode monitor ne sera pas activé.

Dans la suite de ce document, les programmes qui seront décrits sont mis à la disposition du lecteur sur notre site web à http://projets.sequanux.org/membres/sim/usb/ et http://jmfriedt.free.fr. La fonction qui initialise la communication avec le bootloader pour lancer le mode monitor se trouve ainsi dans le programme `hc08.c` – fonction `init_hc08mon()`.

Une façon simple de tester si le matériel fonctionne, sans même passer par une phase de programmation/assemblage en 6808, est de changer la valeur stockée dans le registre associé à un port matériel dont nous visualisons par exemple l'état par une diode électroluminescente (LED). La diode étant connectée au port D, son allumage/extinction est commandé par la valeur stockée dans le registre d'adresse 0x0003 (PTD) après avoir défini ce port comme étant une sortie, en stockant la valeur 0xFF dans le registre d'adresse 0x0007 (DDRD). Faire clignoter une diode depuis le mode moniteur se résume donc à l'envoi des commandes suivantes depuis un programme de communication par le port série du PC :

```
lo=0x07; hi=0; writ_hc08(fd,hi,lo,0xFF);
while(1) {
    lo=0x03; hi=0; writ_hc08(fd,hi,lo,0xFF);
    sleep(1);
    lo=0x03; hi=0; writ_hc08(fd,hi,lo,0x00);
    sleep(1);
}
```

où nous avons implémenté les fonctions de base de communication entre le 68HC908 et le PC – à savoir ici une écriture en mémoire du HC908 `writ_hc08()` – par l'envoi sur le port RS232 (ouvert en lecture/écriture avec un descripteur de fichier `fd`) de la séquence suivante :

1. le PC envoie le code de commande 0x49 (`WRITE`)
2. le PC écoute la réponse du HC908 qui lui confirme que la commande a été comprise en répondant 0x49
3. le PC envoie l'octet de poids fort de l'adresse du registre à modifier, et écoute la réponse du HC908 qui confirme la réception de cette donnée. La séquence est alors la même pour l'octet de poids faible.
4. finalement, l'octet à écrire (la donnée) est transféré par le PC puis confirmé par le HC908.

Ce programme est disponible dans notre archive sous le nom `hc08test.c`.

Attention : le 68HC908 ne possède pas d'UART pour gérer la communication RS232. Une émulation logicielle de ce protocole est effectuée, basée sur une *unique* broche (PTA0) connectée aux broches d'émission et de réception du port série du PC par deux diodes protégeant chaque signal des fluctuations de niveau de l'autre signal (voir Fig. 1). La conséquence est que tout signal émis par le PC (signal de transmission de l'UART) est immédiatement vu par le PC sur sa ligne de réception. L'UART place en mémoire cette donnée avant de recevoir la réponse du 68HC908 acquittant la réception de l'ordre. En conséquent, il faut toujours lire *2 octets* pour vérifier la bonne transmission d'une donnée du PC vers le 68HC908, le premier octet étant directement transmis de la voie d'émission vers la voie de réception (sans intérêt) et le second étant la réponse du 68HC908 (intéressant). La procédure pour envoyer la commande 0x49 est donc (en supposant que le port série a été ouvert et associé à un descripteur de fichier `fd` et que `buf` est un tableau de caractères : `buf[0]=0x49 ;write(fd,buf,1) ;read(fd,buf,2) ;printf("%d ",buf[1]&0xff) ;`

De la même façon nous pouvons stocker une valeur en RAM et la relire :

```
lo=0x57; hi=0; read_hc08(fd,hi,lo);
lo=0x57; hi=0; writ_hc08(fd,hi,lo,0xaa);
lo=0x57; hi=0; read_hc08(fd,hi,lo);
```

Ici la fonction `read_hc08` fonctionne de la même façon que vu précédemment à l'exception du dernier octet – la donnée lue en mémoire du HC908 et transmise au PC – qui ne nécessite une lecture que d'un octet unique puisqu'il ne s'agit pas d'une réponse à un ordre issu du PC.

3.2 La communication RS232 et la mémoire flash

Un premier exemple simple (programme en Table 1) consiste à faire clignoter cette même LED connectée au port PTD0/1 au moyen d'un programme exécuté depuis la mémoire volatile (RAM). Ce premier exemple très simple permet de se familiariser avec la syntaxe de l'assembleur 6808 et l'architecture de ce cœur de microprocesseur (un unique accumulateur A sur 8 bits, un index d'adresse H :X qui peut éventuellement fournir un second registre de travail sur 8 bits, un pointeur de pile et un pointeur de position d'exécution du programme [6, p.24]) puis avec la séquence de compilation et de transfert du programme en RAM.

```
start:  ldhx #0x0140            ; TXS : (SP)<-(H:X)-1 => STACK=0x013f
        txs                     ; reset stack pointer
        ; mov  #0x01,CONFIG1    ; disable COP watchdog, CONFIG1=0x001f
        clra                    ; clear accumulator
        mov  #0x0f,0x0007       ; DDRD: port D as output

loop:   eor  #0x0f              ; toggle diode
        sta  0x0003             ; store accumulator on PortD
        bsr  delay
        bra  loop

delay:  psha
        pshx
        clrx                    ; 256*0,9375ms=240ms
loopx:  clra                    ; 9*256=2304 (0,9375ms @ 2,4576MHz)
loopa:  nsa                     ; [3]
        nsa                     ; [3]
        dbnza loopa             ; [3]
        dbnzx loopx             ;
        pulx
        pula
        rts
```

Tab. 1 – Premier programme permettant de faire clignoter une diode connectée au port PTD0/1 du 68HC908JB8.

Ayant assemblé ce programme (`as6808 -o programme.asm`) et extrait les opcodes (Table 2) du fichier ainsi généré (`grep ^T programme.rel | cut -c9-80 > programme.out`) nous sommes prêts à transmettre le programme en RAM (`hc08ram programme.out`) et à l'exécuter. Cette opération nécessite une dernière exploration du mode de fonctionnement du processeur. Le bootloader nous fournit une commande RUN (commande 0x28) dont le fonctionnement n'est pas expliqué dans le manuel du microcontrôleur [5] mais dans le programme `mongp32.c`, fonction `mon_runpc()` de `spgmr08`. Il faut demander au bootloader quelle est la position actuelle du pointeur de pile (stack pointer SP), et placer en SP+4 l'octet de poids fort de l'adresse de début de programme et en SP+5 l'octet de poids faible, avant d'appeler la fonction 0x28. Dans le cas d'un programme exécuté depuis la RAM, nous plaçons donc 0x00 en SP+4 et 0x40 en SP+5 lors de l'exécution de la commande 0x0C.

Enfin, un dernier point important à considérer lors de l'exécution d'un programme depuis la RAM est que le bootloader du 68HC908JB8 définit sa propre pile en 0x00FF. Ainsi, tout programme en RAM qui se trouve à cet emplacement sera écrasé lors de l'exécution de commandes du bootloader : nous n'avons donc pas 256 octets de RAM disponibles pour nos tests depuis la RAM (0x40 à 0x140) mais seulement 186 octets (de 0x40 à 0xFA, si on se laisse un peu de marge). Nous comprenons ainsi la nécessité de placer le programme de gestion de la communication USB (l'exemple Freescale prend 1,8 KB) en mémoire flash.

Ceci nous indique donc la procédure à suivre pour mettre en pratique la procédure de programmation de la mémoire flash : nous devrons dans un premier temps remplir la RAM du code

```
45 01 40 94 4F 6E 0F 07 A8 0F C7 00 03
AD 02 20 F7 87 89 5F 4F 62 62 4B FC 5B
F9 88 86 81
```

Tab. 2 – Opcodes issus de l'assemblage du programme présenté en Table 1 : il s'agit là de la séquence d'octets prête à être transférée en RAM pour exécution par `hc08ram`. Il peut être instructif de se familiariser avec le processus d'assemblage en l'effectuant à la main [6, pp.79-87] sur cet exemple court.

permettant de programmer la mémoire flash (en effet nous ne disposons que de 256 octets de RAM – insuffisant pour contenir tout le code de gestion de la communication USB – et il nous faut absolument accéder aux 8 KB de mémoire flash pour utiliser efficacement ce microcontrôleur), puis exécuter ce code afin de lire le programme plus volumineux sur le port série et le stocker en flash. Nous avons mentionné auparavant que que l'absence d'UART rend la lecture sur le port série plus complexe que sur la majorité des microcontrôleurs : il nous faut donc émuler un tel périphérique de façon logicielle dans le programme de réception des octets à stocker en mémoire flash, tel que présenté dans le programme de la Table 3.

Attention : la RAM du 68HC908 commence en 0x40, les 64 premiers octets étant réservés pour les registres de gestion du matériel. Tout programme destiné à être exécuté depuis la RAM devra donc être compilé pour commencer à cette adresse. De même, étant donné que nous avons 256 octets de RAM, la pile est placée à la fin de cette RAM (placer une valeur sur la pile *décrémente* le pointeur de pile) donc nous chargeons la pile en 0x140 par la séquence `ldhx #0x0140` et `txs` qui commencera tout nouveau programme (syntaxe `asxxxx`).

Une fois équipé de cette procédure de communication RS232 (remplacer ce code 3 chargé de l'écriture par une lecture est une opération simple), nous pouvons nous atteler au problème de récupérer des octets sur le port série tels que transmis par le PC et les stocker en mémoire flash. La mémoire flash commence en 0xDC00 donc :

1. tout programme destiné à être exécuté depuis la flash doit être recompilé pour commencer à cette adresse (redéfinition des sauts en adresses absolues `JMP`)
2. nous commencerons à écrire à cet emplacement les octets reçus du PC lors du remplissage de la flash.

Plutôt qu'utiliser les procédures fournies en ROM par le 68HC908 pour accéder à la flash (lecture/écriture), relativement contraignantes en terme de gestion de la RAM partagée avec notre programme, nous avons décidé d'implémenter quelques routines de base pour écrire en flash tel que décrit dans [5, section 4]. L'ensemble de ces routines, fastidieuses à décrire ici, sont disponibles dans l'archive mise à disposition sur http://projets.sequanux.org/membres/sim/usb/ sous les noms `flash_write.asm` pour recevoir un programme du PC et le stocker en flash, `flash_read.asm` pour lire le contenu de la flash et transmettre au PC, et finalement `flash_erase.asm` pour effacer le contenu de la flash, opération nécessaire avant toute nouvelle écriture. Ainsi la procédure pour stocker un programme en mémoire flash avec nos outils est

1. `hc08flash flash_erase.out` pour effacer la flash
2. `hc08flash flash_write.out programme.out` pour écrire en flash la séquence d'opcodes `programme.out`
3. `hc08flash` (sans argument) pour exécuter le contenu de la flash

Dans le cas particulier de l'utilisation des interruptions, nous avons écrit un programme additionnel qui est volontairement conservé distinct des autres procédures de gestion de la flash du fait du danger associé à son utilisation. En effet, les vecteurs d'interruption se trouvent en mémoire non-volatile dans la plage 0xFFF0-0xFFFF, donc dans une zone disjointe de la flash. Lors de l'initialisation (mise sous tension) du 68HC908, le bootloader teste le contenu du vecteur d'interruption de Reset en 0xFFFE-0xFFFF et ne s'exécute *que* si ces deux octets sont à 0xFFFF. Ainsi, avec notre montage électronique qui ne permet *pas* d'appliquer une tension de

```
start:  ldhx    #0x0140         ; TXS : (SP)<-(H:X)-1 => STACK=0x013f
        txs                     ; reset stack pointer
        ; mov   #0x01,CONFIG1   ; disable COP watchdog, CONFIG1=0x001f

        mov     #0x00,0x0003    ; PTD0/1 lo => LED lit
        mov     #0x03,0x0007    ; DDRD: PTD0/1 as output

        mov     #0x01,0x0000    ; PTA: PTA0 hi
        mov     #0x01,0x0004    ; DDRA: PTA0 as output
        ldx     #0h00
loop:   incx                    ; increment counter
        txa
        bsr  send               ; send value of counter to serial port
        bra  loop

send:   pshx
        ldx     #0x08           ;   snd through PTA0 the content of Acc (@9600)
        mov     #0x00,0x0000    ;   PTA: PTA0 lo : START bit
looprs: bsr     delay           ; X
        bsr     delay           ; X
        rora                    ; 1 rotate right Acc through carry
        bcc     bit0            ; 3 branch if carry is clear (is A&1=0)
        mov     #0x01,0x0000    ; 4 PTA0=hi
        bra     bit1            ; 3
bit0:   mov     #0x00,0x0000    ; 4 PTA0=lo
bit1:   dbnzx   looprs          ; 3   --> sum=11 or 14
fin:    bsr     delay
        bsr     delay
        mov     #0x01,0x0000    ; PTA: PTA0 hi : STOP bit
        bsr     delay
        bsr     delay
        pulx
        rts

delay:  pshx                    ; 2+4 for bsr   (12+2*((X*9)+15))*.3333=833
        ldx     #0h0f           ; 3                                    104
loopx:  nsa                     ; 3    => Xinit=0x88 for 1200
        nsa                     ; 3           =0x0f for 9600
        dbnzx loopx             ; 3
        pulx                    ; 2
        rts                     ; 4
```

TAB. 3 – Exemple de programme transmettant à la vitesse de 9600 bauds les valeurs d'un compteur sur le port série. La vitesse de transmission est définie par la valeur du délai dans la fonction `delay` qui est utilisée pour définir de façon logicielle la durée de chaque bit transmis afin d'être compatible avec le protocole asynchrone : une valeur de 0x0F dans le registre X permet une transmission à 9600 bauds tandis qu'il faut utiliser une valeur de 0x88 pour une transmission à 1200 bauds.

8 V sur la broche IRQ# (condition de désactivation de ce test initial), le 68HC908 devient inutilisable si une valeur est écrite dans ces deux registres. Nous avons donc écrit un petit programme autonome, `flash_irqs.asm`, qui prend en entrée une liste de 16 valeurs qui sont les 16 octets définissant les vecteurs d'interruption, mais ne stocke en mémoire que les 14 premières valeurs afin de s'assurer de ne pas modifier le vecteur de reset. Ce programme s'utilise donc à la fin de la séquence de programmation vue plus haut par `hc08flash flash_irqs.out programme.irqs` avec `programme.irqs` contenant les 16 octets à placer dans les vecteurs d'interruption.

4 La communication USB sur le 68HC908

Un circuit de développement est vendu par la société MCT Elektronikladen (http://hc08web.de/usb08/) et un exemple d'application communiquant par le port USB est distribué gratuitement sur leur site web. C'est de cette application que nous allons partir pour comprendre le fonctionnement du bus USB et développer un driver GNU/Linux communiquant avec le microcontrôleur. Nous nous affranchissons ainsi d'une description détaillée du programme exécuté sur le microcontrôleur en utilisant un code prêt à l'emploi. Ce programme, écrit en C, est proposé sous forme de code source et fichier S19 après compilation par divers compilateurs commerciaux sous Windows. Nos tentatives de porter ce code à `sdcc` n'ont pas été fructueuses : nous nous contenterons donc d'exploiter les fichiers compilés (S19) et n'utiliserons le code C que pour comprendre le sens du programme après désassemblage – ce code désassemblé est mis à disposition dans notre archive [4] et pourra servir de base aux applications spécifiques dont chaque lecteur pourra avoir besoin. Alternativement, une application totalement opensource basée sur le 68HC908JB8 est USB-IR-Boy tel que présentée à http://usbirboy.sourceforge.net/ sur lequel nous ne nous attarderons pas ici puisqu'il est le sujet d'un autre article dans ce même numéro.

Un point fondamental que nous préciserons ultérieurement est que le bus USB est très sensible aux délais. Or il apparaît que le code C compilé avec le compilateur commercial ICC (http://www.imagecraft.com/ sous Windows) tel que proposé par Motorola ne respecte pas ces conditions et ne fonctionne par correctement avec les drivers `ohci` et `usb-uhci` de GNU/Linux. Il faut donc prendre soin de flasher le code compilé avec le compilateur C Cosmic (http://www.cosmic-software.com/ sous Windows) tel que fourni par Motorola. Nous convertissons donc le fichier au format S19 fourni dans l'archive http://hc08web.de/usb08/files/usb08_fwv102_cosmic.zip pour générer la suite d'opcodes transférée en mémoire flash selon la méthode présentée plus haut.

Notre approche a donc été la suivante : partir du code d'exemple fourni par MCT Elektronikladen, vérifier son bon fonctionnement, puis désassembler ce code afin de pouvoir modifier l'action associée à la transmission de trames USB sans toucher au gestionnaire de communication lui même, pour finalement re-assembler le code résultant. Le passage d'une suite d'opcodes (le fichier .S19 fourni en exemple) au programme assembleur étant un opération bijective, nous évitons tout problème de version de compilateur de langage évolué (ici le C) en travaillant systématiquement en assembleur : le programme assemblé après nos modifications aura la même latence dans la communication USB que le code original (tant que nous ne touchons pas aux routines de gestion des interruptions associées à la communication USB).

Ainsi, sans même savoir ce qu'est un driver et sans maîtriser le protocole USB, nous pouvons vérifier si le client USB fonctionne après programmation du microcontrôleur et déjà valider l'aspect matériel.

Avec le code de http://www.elektronikladen.de/en_usb08.html qui occupe 30% de la mémoire flash, le client USB est reconnu sous GNU/Linux par le message suivant :

```
$ 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.
```

Nous identifions là deux champs qui nous seront utiles : les identificateurs vendeur (0x0c70) et produit (0x0000).

5 Couche matérielle de USB

Un bus USB porte l'alimentation sous 5 V permettant d'alimenter le microcontrôleur, ainsi qu'un signal asynchrone bidirectionnel sous une tension de 3,3 V.

[4] http://projets.sequanux.org/membres/sim/usb/

Une fois le client USB connecté, une adresse lui est dynamiquement assignée et servira aux transactions ultérieures : dans l'exemple précédent, l'identifiant 31 nous avait été assigné.

Les données transitant entre le controlleur USB et les clients sont encapsulées dans des paquets dont le destinataire est défini par l'identitifiant vu précédemment, tandis qu'un certain nombre d'*endpoints* (voir section 6.3) jouent le même rôle que les ports sur un réseau TCP/IP ou UDP sur Ethernet. Nous n'aurons pas à nous préoccuper de la structure détaillée des paquets puisque le microcontrôleur gère leur assemblage de façon transparente pour l'utilisateur.

De façon générale, la communication sur bus USB se fait par paquets de 8 (lent), 16, 32 ou 64 octets (rapide) [7, p.41]. Un certain nombre de modes de communication ont été définis pour partager la bande passante du bus USB en fonction des besoins (et donc des applications) des divers périphériques USB. Nous les citons ici pour comparer les capacités du 68HC908JB8 (mode Interrupt) aux autres modes disponibles et ainsi mieux cerner son champ d'applications [1] :

- **Interrupt** (débit constant, réessaie 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)

6 Les drivers et la communication avec le matériel sous linux

6.1 Généralités sur les drivers

Nous allons présenter dans ce document les points importants de la réalisation d'un pilote USB pour les noyaux 2.4.x du système d'exploitation GNU/Linux [1, 8]. Dans notre archive, un port de ce module est également disponible pour les versions 2.6 du noyau. Même si ce pilote est dédié à la communication avec le microcontrôleur 68HC908JB8, il se veut aussi générique que possible. Cela signifie qu'il doit pouvoir être facilement adapté à un autre microcontrôleur implémentant l'USB en mode interrupt. Nous ne nous attarderons pas ici sur la définition d'un driver et l'intérêt de programmer un module pour la communication avec les périphériques (portabilité, simplicité de compilation, fournir des méthodes standard à l'utilisateur) [9]. Nous supposons connu le fait que tous les pseudo-fichiers interfaçant les processus utilisateurs avec un driver se trouvent dans le répertoire dédié `/dev` et sont définis par deux identificateurs, le nombre majeur (classe de périphérique) et le nombre mineur (indice dans cette classe de périphériques utilisant tous le même pilote).

Un driver doit fournir un certain nombres de méthodes pour permettre aux programmes utilisateurs d'accéder à un périphérique matériel : `write()`, `read()` et `ioctl()`. La première permet d'écrire des données sur le périphérique, la seconde d'y lire des données, et la troisième de le configurer. Deux autres méthodes standards d'initialisation sont `open()` et `close()`.

Il nous faut tout d'abord définir la structure de donnée présente dans tout module qui contient les pointeurs vers les fonctions permettant d'interagir 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,
};
```

Cette structure va permettre d'enregistrer auprès du noyau notre périphérique de type caractère (*char device* – nous communiquerons avec notre microcontrôleur octet par octet et non par blocs) :

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

6.2 Cas particulier des drivers USB

Une des particularités d'un pilote USB est qu'il n'interagit pas directement avec son matériel. Comme on peut le remarquer dans le schéma de la figure 2, drivers et périphériques sont séparés par des couches modulaires. Au niveau le plus bas, directement en contact avec le périphérique, on trouve une couche représentant le pilote du contrôleur USB maître (Host Controller). Le choix de ce pilote (`usb-uhci` ou `usb-ohci`) va être déterminé par le type de contrôleur. Pour un contrôleur UHCI (Universal Host Controller Interface d'Intel), le module `usb-uhci` sera utilisé tandis que pour un contrôleur OHCI (Open Host Controller Interface de Compaq, Microsoft et National Semiconductor), le module `usb-ohci` sera préféré.

Entre le Host Controller et les drivers USB, on trouve le module `usbcore`. Son rôle est de fournir aux pilotes de périphériques une interface standard indépendante du type de contrôleur utilisé. Il va également se charger d'arbitrer la transmission de données sur le bus commun à tous les périphériques USB. Ce module `usbcore` va faciliter le développement des pilotes USB en mettant à leur disposition un certain nombre de méthodes. La communication avec le périphérique (initialisation, transfert de données, ...) passera donc par des fonctions exportées par le module `usbcore`.

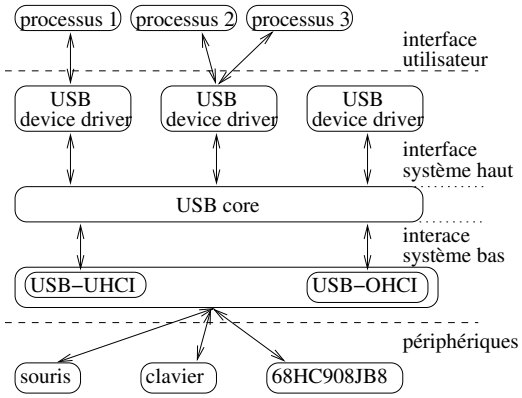

FIG. 2 – Implémentation par couches de l'USB sous GNU/Linux [1].

Pour pouvoir bénéficier de toutes ces ressources notre module va devoir s'enregistrer auprès des couches précédemment décrites. Pour cela une structure de type `struct usb_driver` est renseignée puis enregistrée auprès de `usbcore` à l'aide de la fonction `usb_register`.

```
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,
};

result = usb_register(&hc08_driver);
```

Mises à part les méthodes `probe` et `disconnect`, dont nous expliquerons le rôle plus tard (section 6.3), on peut remarquer la présence d'un tableau de type `id_table`. Il contient les identifiants vendeur et produit correspondant à notre périphérique, le microcontrôleur 68HC908JB8.

```
#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) },
        { }
};
```

Fournir cette table au gestionnaire de périphériques USB lui permet de créer une liste reliant les identifiants de périphériques avec les drivers associés. À chaque fois qu'un matériel est connecté au bus USB, le noyau va essayer de déterminer s'il dispose d'un pilote susceptible de le prendre en charge. Pour cela, il va comparer l'identifiant du périphérique nouvellement connecté à la liste des identifiants déclarés par les pilotes USB. Si une correspondance est établie avec un pilote, alors la méthode `probe` de ce dernier est appelée.

6.3 La méthode `probe`

La méthode `probe` d'un pilote USB va tenter de déterminer avec précision si le périphérique arrivant dépend bien du pilote qu'elle représente. En effet, une authentification reposant uniquement sur les identifiants vendeur et produit est insuffisante. Notre travail va donc être de vérifier que le périphérique proposé par le gestionnaire USB est bien le microcontrôleur 68HC908JB8.

Si à l'issue de cette série de tests la fonction probe retourne un pointeur non nul, alors le noyau attribuera la gestion du périphérique à notre pilote. C'est grâce a cette méthode `probe` que linux va être capable de gérer le branchement à chaud d'un matériel USB. Chaque périphérique connecté au bus USB est décrit par un objet ou une hiérarchie de structures (Fig. 3).

La structure `struct usb_device` est le point d'entrée de cette arborescence qui contient toutes les informations relatives au périphérique USB. Lorsqu'elle est appelée, la méthode `probe` reçoit une telle structure en paramètre d'entrée. Cette dernière a été préalablement initialisée par le module `usbcore`. Nous allons donc parcourir et examiner cette arborescence afin d'établir ou non une correspondance avec le microcontrôleur 68HC908JB8.

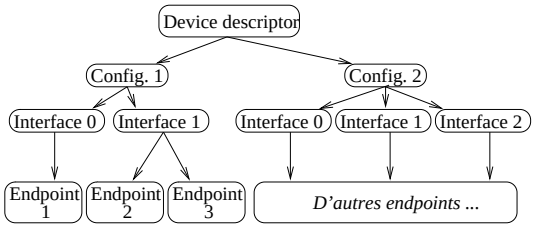

FIG. 3 – Représentation sous forme d'arbre d'un périphérique USB sous GNU/Linux.

– **device descriptor** :
Ce descripteur se présente sous la forme d'une structure `struct usb_device_descriptor`. Il regroupe toutes les informations d'ordre global sur le périphérique. On y trouvera entre autres les identifiants vendeur et produit. L'examen des ces deux champs est fondamental pour un pilote dans sa décision de s'attribuer ou non la gestion d'un périphérique. Cette structure contient également le nombre de configurations possibles pour le périphérique.

– **configuration descriptor** :
Chaque configuration du périphérique est décrite dans une structure `struct usb_config_descriptor`. Ce système de configuration sert à séparer les différents modes de fonctionnement du périphérique. Par exemple dans le cas d'une imprimante-scanner, on peut imaginer une configuration pour le mode imprimante et une autre pour le mode scanner. Dans le cas particulier du 68HC908JB8, une seule configuration est définie.

```
struct usb_config_descriptor {
[...]
        __u8  bNumInterfaces    __attribute__ ((packed));
        struct usb_interface *interface;
[...]
```

– **interface descriptor** :
Au sein d'une même configuration, un périphérique peut disposer de plusieurs interfaces. On peut comparer une interface à un réglage particulier du périphérique dans un mode de fonctionnement.

Par exemple dans le cas d'une caméra vidéo, on peut imaginer une interface pour chacun de ses modes d'acquisition. Encore une fois, le 68HC908JB8, en raison de sa simplicité ne définit qu'une seule interface.

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

– **endpoint descriptor** :
Ces objets permettent de représenter de manière logicielle, les ports d'entrées et de sorties utilisables pour communiquer avec le périphérique. Les endpoints sont au transfert USB ce que les sockets sont à la communication réseau. Une structure `struct usb_endpoint_descriptor` est associée à chacun des endpoints offerts par le périphérique. Cette structure va fournir des informations sur le mode de transfert (bulk, interrupt ou isochronous), sur le sens du transfert (in, out) ou encore la taille maximale des données pour un transfert...

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

Dans le cas du 68HC908JB8, les endpoints peuvent être définis comme des points d'accès présentant une correspondance directe avec un ensemble de registres du côté du microcontrôleur.

Le 68HC908JB8 fournit 3 endpoints :
– un port de contrôle (endpoint 0) : bidirectionnel, buffer 8 B I/O
L'endpoint 0 est toujours le port lié au contrôle du port USB : c'est par lui que le maître attribue une adresse à un périphérique esclave lorsqu'il se connecte au bus USB.
– un port en émission en mode interrupt (endpoint 1) : buffer 8 B O
– un port bidirectionnel en mode interrupt (endpoint 2) : buffer 8 B **I**/O

Une lecture du fichier d'entête `/usr/src/linux-2.4.22/include/linux/usb.h` fournira au lecteur curieux des informations plus détaillées sur le contenu de ces différentes structures.

Fort de toutes ces précieuses informations, voyons comment la fonction `hc08_probe` identifie ou non le microcontrôleur 68HC908JB8.

```
static void * hc08_probe (struct usb_device *udev, unsigned int ifnum,
                          const struct usb_device_id *id)
{
        struct usb_hc08 *dev = NULL;
        struct usb_interface *interface;
        struct usb_interface_descriptor *iface_desc;
        struct usb_endpoint_descriptor *endpoint;
        int i, endpoint_match = 0;

        [...]
```

On vérifie que les identifiants vendeur et produit correspondent bien à ceux du 68HC908JB8. Ce test peut être qualifié de paranoïaque puisque cette condition doit être vérifiée pour que `hc08_probe` soit appelée.

```
        /* See if the device offered to us matches what we can accept */
        if ((udev->descriptor.idVendor != USB_HC08_VENDOR_ID) ||
```

```
    (udev->descriptor.idProduct != USB_HC08_PRODUCT_ID))
{
        info ("unable to probe new device");
        return NULL;
}

[...]
```

Le microcontrôleur 68HC908JB8 n'étant pas un périphérique complexe, il ne dispose que d'une seule configuration et que d'une unique interface. Nous pouvons donc avancer dans l'arborescence directement jusqu'à l'interface active. Pour un périphérique plus sophistiqué, des tests complémentaires, sur sa classe par exemple, peuvent s'avérer utiles.

```
interface = &udev->actconfig->interface[ifnum];

[...]
```

Nous allons donc passer en revue les différents endpoints et voir si ils sont conformes à nos attentes. En fait, nous devons identifier 2 endpoints en mode interrupt, un étant configuré pour l'envoi de données (OUT) et l'autre pour la réception (IN). Un endpoint supplémentaire, de contrôle, doit également apparaître.

```
/* setup the endpoint information */
/* check out the endpoints */
iface_desc = &interface->altsetting[0];
for (i = 0; i < iface_desc->bNumEndpoints; ++i)
{
        endpoint = &iface_desc->endpoint[i];

        if ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
                        == USB_ENDPOINT_XFER_CONTROL)
                dbg ("%s - control endpoint found",
                                __FUNCTION__);

        [...]

        if (!(endpoint->bEndpointAddress & USB_DIR_IN) &&
           ((endpoint->bmAttributes & USB_ENDPOINT_XFERTYPE_MASK)
            == USB_ENDPOINT_XFER_INT))
        {
```

Une fois un endpoint détecté, une structure de données `struct usb_hc08`, définie et utilisée en interne par notre driver, est initialisée avec des valeurs propres à l'endpoint (adresse, taille du buffer de transfert, etc...). Un URB (USB Request Block) est également alloué. Sa fonction et son utilité seront présentés au chapitre suivant. Un compteur `endpoint_match`, correspondant au nombre d'endpoints identifiés est également incrémenté.

```
                /* we found an interrupt out endpoint */
                dev->write_urb = usb_alloc_urb(0);
                if (!dev->write_urb)
                {
                        err("No free urbs available");
                        goto error;
```

```
                        }

                        buffer_size = endpoint->wMaxPacketSize;
                        dev->int_out_size = buffer_size;
                        dev->int_out_endpointAddr = endpoint->bEndpointAddress;
                        /* as we want to write data packets one by one
                         * the interval used to poll urb transfers is
                         * set to 0
                         * that means ... only one transfer */
                        dev->int_out_interval = 0;
                        dev->int_out_buffer = kmalloc (buffer_size, GFP_KERNEL);
                        if (!dev->int_out_buffer)
                        {
                                err("Couldn't allocate int_out_buffer");
                                goto error;
                        }

                        endpoint_match ++;
                }

        }
```

Si le compte d'endpoints est bon, un pointeur non NULL est retourné au gestionnaire de l'USB. Ce dernier confiera la gestion du périphérique à notre pilote. À l'inverse, si la méthode `probe` détecte une anomalie, le pointeur NULL sera retourné, dégageant notre pilote de toutes obligations envers le matériel.

```
        /* if no endpoint is matching, just exit */
        if (endpoint_match < 2)
        {
                info ("only %d endpoints found instead of 2",
                                endpoint_match);
                goto error;
        }

        [...]

error:
        hc08_delete (dev);
        dev = NULL;

exit:
        up (&minor_table_mutex);
        return dev;
}
```

7 La communication sur bus USB

Afin d'illustrer la communication entre le pilote et le 68HC908JB8, nous présenterons la méthode `write` permettant de transmettre des données au microcontrôleur.

```
static ssize_t hc08_write (struct file *file, const char *buffer, size_t count, loff_t *ppos)
{
        struct usb_hc08 *dev;
        ssize_t bytes_written = 0;
        int retval = 0;
```

```
dev = (struct usb_hc08 *)file->private_data;
[...]
```

Dans un premier temps nous récupérons les données que le processus appelant souhaite transmettre. La recopie des données depuis l'espace utilisateur vers le noyau est réalisée à l'aide de la fonction `copy_from_user`.

```
[...]
/* we can only write as much as 1 urb will hold */
bytes_written = (count > dev->int_out_size) ?
                        dev->int_out_size : count;

/* copy the data from userspace into our urb */
if (copy_from_user(dev->int_out_buffer, buffer, bytes_written))
{
        retval = -EFAULT;
        goto exit;
}
[...]
```

Le noyau Linux utilise une structure appelée URB (USB Request Block) afin de définir les transactions sur le bus USB. Elle contient toutes les informations nécessaires à la transmission physique des données. Elle est décrite dans le fichier `/usr/src/linux/Documentation/usb/URB.txt` fourni avec les sources du noyau Linux. L'URB est défini sous la forme d'une structure de type `struct urb`. Étant donnée la complexité de cette dernière, le module `usbcore` met à disposition des pilotes des fonctions afin d'en faciliter l'initialisation. Nous utiliserons la fonction `usb_fill_int_urb` afin de construire un URB destiné à opérer un transfert en mode interrupt avec notre microcontrôleur.

```
[...]
/* set up our write urb */
usb_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);
[...]
```

Détaillons les paramètres de cette fonction :

- `dev->write_urb` : représente la `struct urb` en construction.
- `dev->udev` : désigne l'arborescence représentant le périphérique USB.
- `usb_sndintpipe (dev->udev, dev->int_out_endpointAddr)` : cette fonction permet d'associer un `pipe` ou canal de communication à l'endpoint 1 du microcontrôleur 68HC908JB8.
- `dev->int_out_buffer` : buffer contenant les données à transmettre.
- `dev->int_out_size` : taille en octets du transfert à effectuer.
- `dev->int_out_interval` : cette valeur précise l'intervalle de temps entre deux transferts de données. N'oublions pas que le mode de transmission interrupt est périodique. Dans notre application, répéter un même transfert périodiquement ne nous intéresse pas. C'est pourquoi `dev->int_out_interval` est initialisé a 0. Cela signifie que nous souhaitons procéder à un unique transfert.
- `hc08_write_int_callback` : ce paramètre est en fait un pointeur de fonction. Cette dernière sera appelée après la fin du transfert afin de le compléter. Son rôle est de vérifier que tout s'est déroulé normalement puis de réveiller tous les processus en attente de l'achèvement du transfert. Il est utile de préciser que la fonction `hc08_write_int_callback` est appelée dans un contexte d'interruption. Elle doit donc être définie avec les mêmes précautions qu'un gestionnaire d'interruption classique. Voici comment notre pilote la défini :

```
static void hc08_write_int_callback (struct urb *urb)
{
```

```
        struct usb_hc08 *dev = (struct usb_hc08 *)urb->context;

        dbg("%s - minor %d", __FUNCTION__, dev->minor);

        if ((urb->status == -ENOENT) ||
            (urb->status == -ECONNRESET))
        {
                dbg ("%s - write urb cancelled assynchronously or by user ", __FUNCTION__);
        }
        else
        {
                dbg ("%s - write int status : %d", __FUNCTION__, urb->status);
        }

        /* send a signal to the sleeping process,
         * waiting for write again */

        wake_up_interruptible (&dev->int_out_wait);

        return;
}
```

Après avoir initialisé l'URB, il ne reste plus qu'à le soumettre au module `usbcore` afin de procéder au transfert. Pour cela, la fonction `usb_submit_urb`, encore une fois exportée par `usbcore` est utilisée.

```
[...]
retval = usb_submit_urb (dev->write_urb);
[...]
```

Une fois le transfert initié, le processus utilisateur appelant est mis en sommeil. La fonction de complétion `hc08_write_int_callback` le réveillera une fois le travail achevé. La communication avec le 68HC908JB8 est donc bloquante. Cela signifie qu'un programme communiquant avec le microcontrôleur sortira des appels systèmes `read` et `write` uniquement après le transfert effectif des données.

```
while (dev->write_urb->status == -EINPROGRESS)
{
        retval = wait_event_interruptible (dev->int_out_wait,
                        dev->write_urb->status);
        if (retval == -ERESTARTSYS)
        {
                dbg ("%s - error wait_event_interruptible ()",
                                __FUNCTION__);
                goto exit;
        }
}
```

En raison de ses nombreuses similitudes avec `hc08_write`, la méthode `hc08_read`, permettant de lire les données transmises par le microcontrôleur, n'est pas présentée ici. Après avoir défini au niveau du driver, les méthodes de communication avec le 68HC908JB8, il ne reste plus qu'à les utiliser au niveau du programme utilisateur par l'intermédiaire des appels systèmes `read` et `write`. La communication avec le microcontrôleur se déroulera donc classiquement comme avec tout autre périphérique.

8 La communication avec le programme utilisateur

Une fois le drivers fonctionnel, le développement d'une application pour communiquer par USB avec le 68HC908JB8 est une simple formalité. Voici l'extrait principal du programme utilisateur permettant de faire clignoter la diode du circuit de test (PTD0/1) et de lire les valeurs renvoyées par le microcontrôleur :

```
/* read on the hc08 device */
while (num)
{
        /* trying to write... in hope of a reading */
        if ((trans=write (desc, data_write, SIZE)) < SIZE)
        {
                fprintf (stderr, "only %d bytes written instead of %d\n", trans, SIZE);
                perror ("write ()");
        }
        memset (data_read, 0x00, SIZE);
        if ((trans=read (desc, data_read, SIZE)) < SIZE)
        {
                fprintf (stderr, "only %d bytes read instead of %d\n", trans, SIZE);
                perror ("read ()");
        }
        printf ("buffer : ");
        for (j=0; j<SIZE; j++)
        {
                printf ("%.2x ", data_read[j] & 0xff);
        }
        printf ("\n");
        if (*data_write)
        {
                memset (data_write, 0x00, SIZE);
        }
        else
        {
                memset (data_write, 0xff, SIZE);
        }
        usleep (200000);
        num --;
}
```

Le lecteur attentif aura remarqué une alternance entre les lectures et les écritures sur le microcontrôleur 68HC908JB8. Cette bizarrerie est à attribuer à une limitation du firmware fourni par MCT Elektronikladen. En effet, le tampon utilisé en interne est le même pour les lectures et les écritures. Trois écritures remplissent le tampon et les suivantes sont bloquées. La seule méthode pour permettre la poursuite des transactions est de vider le tampon. Cette opération est réalisée par la lecture de données provenant du 68HC908JB8. L'inverse est également vrai : 3 lectures vident le tampon et les lectures suivantes sont bloquées. Cette fois, des écritures sont nécessaires pour recharger le tampon. En conclusion, pour pouvoir lire, il faut écrire et inversement.

9 Conclusion et perspectives

Nous avons présenté ici toutes les étapes de développement au moyen d'outils Opensource d'un périphérique USB basé sur le microcontrôleur Freescale 68HC908JB8. Nous avons présenté le protocole de programmation, les bases de l'assembleur 6808 au moyen de quelques exemples simples, pour finalement aboutir au stockage en mémoire non-volatile d'un programme d'exemple utilisant le port USB.

Nous avons ensuite abordé les méthodes de programmation de modules GNU/Linux pour communiquer sur le bus USB et avons insisté sur les diverses couches d'abstractions implémentées. Ces notions ont été appliquées au cas particulier de la communication avec le 68HC908JB8 : un programme a été présenté pour commander l'allumage ou l'extinction d'une diode *via* le port USB.

Nous nous sommes imposés de garder le code USB fixe afin de limiter les risques d'erreur, et avons focalisé nos efforts sur le développement d'un driver GNU/Linux communiquant avec un microcontrôleur 68HC908JB8 exécutant ce code, avec les contraintes liées à la réutilisation d'un code existant (dans ce cas, une subtilité du code distribué est que *toute lecture nécessite une écriture et réciproquement*).

Nous avons donc atteint nos objectifs de

– nous familiariser avec USB

– développer l'ensemble des outils pour le 68HC908 sous Linux
– développer un module Linux pour l'EVB USB du 68HC908

Le travail à réaliser à partir de là est le développement de notre propre code exécuté sur le 68HC908 répondant plus spécifiquement à une application donnée, et la modification du driver de façon conjuguée pour s'adapter au nouveau protocole de communication qui sera alors implémenté. Le module présenté ici s'adapte facilement à toutes sortes d'applications.

Remerciements

Nous remercions Benoît Demaine pour les discussions stimulantes au cours de cette étude. L'association de diffusion des logiciels libres sur la Franche-Comté – Sequanux (www.sequanux.org) est remerciée pour son support logistique.

Références

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

[2] M. Zerkus, J. Lusher & J. Ward, "USB Primer", Circuit Cellar **106** (Mai 1999), pp. 58-69, puis J. Lyle, "USB Primer", Circuit Cellar **107** (Juin 1999), pp68-73

[3] "Développez vos pilotes de périphériques USB", GNU/Linux Magazine Hors Série **17** (Novembre-Décembre 2003)

[4] "USB08 Universal Serial Bus Evaluation Board Using the MC68HC908JB8" disponible à http://www.freescale.com/files/microcontrollers/doc/ref_manual/DRM002.pdf

[5] "MC68HC908JB8, MC68HC08JB8, MC68HC08JT8 Technical Data (rev. 2.3)" (09/2005) à www.freescale.com/files/microcontrollers/doc/data_sheet/MC68HC908JB8.pdf

[6] "CPU08 Central Processor Unit Reference Manual (CPU08RM/AD Rev. 3, 2/2001)" à http://www.freescale.com/files/microcontrollers/doc/ref_manual/CPU08RM.pdf

[7] "Universal Serial Bus Specifications (rev. 1.1)" à http://www.usb.org/developers/docs/

[8] G. Kroah-Hartman, `/usr/src/linux/drivers/usb/usb-skeleton.c` dans les sources de Linux

[9] "Linux Device Drivers, 2nd ed.", A. Rubini & J. Corbet, O'Reilly Ed. (2001), disponible à http://www.xml.com/ldd/chapter/book/