Utilisation avancée de l'interface USB-RS232 FT232

Gwenhaël Goavec-Merou

29 octobre 2021

Avec la disparition des ports parallèles des ordinateurs personnels, une source de `GPIOs` (*General Purpose Input/Output*) a également disparu. Il existe bien des convertisseurs USB-parallèles mais ils sont spécifiquement dédiés à une tâche précise : communiquer avec une imprimante. Ainsi, l'amateur ou le débutant qui souhaiterait se familiariser avec le développement de composants ludiques tels que les écrans `LCD`, découvrir des protocoles de communications comme le `SPI` ou l'`I2c`, doit passer par l'usage d'un microcontrôleur entraînant, de-facto, une complication supplémentaire.
Toutefois il est possible de contourner ce problème par l'utilisation, ou plutôt le détournement, de composants dont certaines fonctionnalités sont peu ou pas connues. C'est le cas des convertisseurs `USB-UART` tels que les `Silicon Labs CP2102` ou les `prolific PL2303`, mais également ceux de chez `FTDI`, le `FT232`. C'est à ce dernier composant que nous allons nous intéresser dans la suite.
Il est généralement utilisé comme convertisseur série vers USB et réciproquement. Toutefois il ne se limite pas uniquement à cette fonction mais est également une source d'entrées sorties à petit prix.

1 Présentation

Depuis la disparition de certains ports des PCs, sont arrivées des solutions pour, depuis un ordinateur personnel, réaliser du prototypage rapide ou évaluer un composant. Une des solutions est le *bus pirate* [OS2] [1]. Mais il est également possible d'exploiter du matériel type `arduino` ou même une quelconque carte à base de micro-processeur.

Un point commun à ces diverses approches est que dans la plupart des cas la communication entre la carte et le PC se fait au travers d'un convertisseur `USB-UART`. Dans le cas du *bus pirate* (version 3), ce convertisseur est justement un `FTDI FT232RL`[FT232DS] (c'est également le cas de la première version des `arduinos`).

Dans l'utilisation en tant que convertisseur de protocoles de communication `USB-UART`, 4 broches (nommées `CBUS` sur la figure 1(b)) sont disponibles pour des fonctions diverses telles qu'un témoin de transmission/réception ou un mode de réveil du convertisseur. Toutefois une autre configuration, moins connue, est disponible : le fonctionnement en tant que `GPIO`. Avec cette configuration, les 4 broches sont pilotables depuis la machine hôte en tant que port d'entrée-sortie généraliste numérique, par exemple pour allumer ou éteindre une LED.

Le second mode d'utilisation possible est le `bitbang`[AN232R]. Dans ce mode de fonctionnement, l'ensemble des lignes de l'`UART` fonctionnent comme des `GPIOs` et peuvent être mises en entrée ou en sortie indépendemment. Attention : le mode `bitbang` et le pilotage des broches `CBUS` sont mutuellement exclusifs.

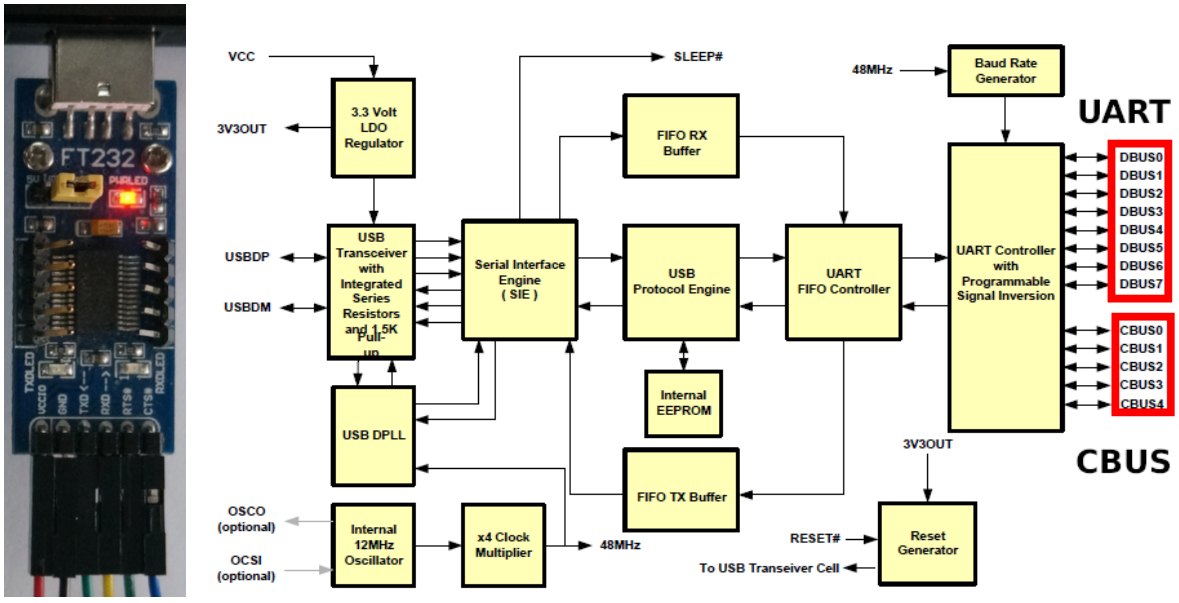
Deux solutions existent pour l'utilisation de ce composant :

— classiquement avec le module noyau `ftdi_sio`. Il donne accès au convertisseur, au travers d'un `/dev/ttyUSBx`, uniquement dans sa fonction première, sans contrôle des broches, ni mode `bitbang` ;

— avec la `libftdi`. Elle prend la main sur la convertisseur, en forçant le noyau à détacher le pilote. Tout, ensuite, est fait depuis l'espace utilisateur. Cette bibliothèque offre non seulement des fonctions pour l'utilisation en mode classique, mais permet de piloter les broches `CBUS` et de passer le `FT232` en mode `bitbang`.

L'utilisation du mode `bitbang`, qui sera abordée dans la seconde partie de cette prose, se fait de manière logicielle par l'envoi d'une commande particulière. En ce qui concerne les broches `CBUS`, la

1. http://dangerousprototypes.com/docs/Bus_Pirate



(a) (b)

FIGURE 1 – 1(a) : exemple de module à base de `FT232`. Il présente l'intérêt de rendre toutes les broches accessibles. 1(b) schéma interne du `FT232`, les broches `DBUS` correspondent à l'`UART` complet. @ftdichip.

configuration de la direction et du niveau se fait également par logiciel mais ces broches doivent être, au préalable, configurées au niveau de l'`EEPROM` en mode `IO`.

Notes : Lorsque la `libftdi` prend la main sur un périphérique, celui-ci est automatiquement détaché et `/dev/ttyUSBx` disparaît. À la fin de l'exécution de l'application, le composant n'est pas réattaché. Plusieurs solutions existent : déconnecter et reconnecter, physiquement, le `FTDI` ; utiliser, en root, la commande : `echo 1 > /sys/bus/usb/drivers_probe` pour forcer la redécouverte ; une autre solution sera traitée dans la section sur la programmation des microcontrôleurs.

2 Outils et compilation

Afin de manipuler la configuration du composant `FTDI` et en modifier l'`EEPROM`, il est nécessaire d'installer la `libftdi` et `ftdi_eeprom`. Dans le cas d'une distribution Debian, les paquets *pkgconf*, *libftdi1-2*, *libftdi1-dev* et *ftdi-eeprom* seront nécessaires à cet effet.

Ce sont les seuls prérequis pour l'utilisation du `FT232` pour des applications autres que de la conversion `USB-UART`.

La bibliothèque `libftdi` fournit le fichier `libftdi1.pc` utilisé par *pkg-config* pour simplifier la ligne de compilation en cherchant automatiquement l'emplacement de la bibliothèque et de ses entêtes :

```
gcc `pkg-config --cflags --libs libftdi` leds.c -g -Wall -o leds
```

Le paramètre *–libs* passé à *pkg-config* retourne les bibliothèques nécessaires pour la compilation et le *–cflags* les répertoires contenant les entêtes.

3 Configuration du FT232RL

Tel que présenté dans [LM125], un outil nommé `ftdi_eeprom` est fourni avec les sources de la `libftdi` (ou dans un paquet éponyme dans certaines distributions). Depuis cet article, l'outil a bien évolué, rendant dès lors l'utilisation des – affreux – logiciels propriétaires inutile.

Celui-ci permet :

— de lire et de stocker le contenu de l'`EEPROM` dans un fichier binaire ;
— d'écraser le contenu de l'`EEPROM` à partir d'un fichier binaire ;
— de générer une nouvelle `EEPROM` à partir d'un fichier de configuration `ASCII`.

Cependant, il n'est pas possible de limiter la modification à seulement quelques paramètres : le fichier de configuration doit fournir la totalité des informations sous peine de comportements inattendus. Cette approche rend son utilisation laborieuse puisqu'il faut respecter la configuration courante du composant, ce qui par ailleurs complique la configuration de plusieurs convertisseurs si l'on souhaite conserver le numéro de série original, par exemple.

Dans le cas qui nous intéresse, à savoir changer la fonction de quelques broches, ce qui tient en 1 ou 2 octets sur les 128 de l'`EEPROM` d'un `FT232`, cet outil est trop complexe. Comme les fonctions nécessaires pour manipuler l'`EEPROM` sont fournies par la bibliothèque et non l'application, il est possible de créer notre propre outil[2]. Celui-ci se bornant, selon le besoin, à changer uniquement quelques options telles que la fonction des broches. Le code 1 permettra également de se familiariser avec l'utilisation de la `libftdi`.

```
1 #include <ftdi.h>
  #include <libusb.h>
3
  int main(void)
5 {
    int ret;
7   struct ftdi_context *ftdi;

9   if ((ftdi = ftdi_new()) == NULL) return EXIT_FAILURE;
    if (ftdi_usb_open(ftdi, 0x403, 0x6010) != 0) goto cleanup;
11  ret = ftdi_read_eeprom(ftdi);
    ret = ftdi_eeprom_decode(ftdi, 1);
13
    ret = ftdi_set_eeprom_value(ftdi, CBUS_FUNCTION_0, CBUS_IOMODE);
15
    ret = ftdi_eeprom_build(ftdi); /* generate new eeprom */
17  ret = ftdi_write_eeprom(ftdi); /* flash */

19  libusb_reset_device(ftdi->usb_dev);
    ftdi_free(ftdi);
21  return EXIT_SUCCESS;
}
```

Listing 1 – Exemple d'application prenant la main sur un `FT232` pour configurer, dans l'EEPROM, certaines broches du composant.

La première étape est le point d'entrée de toute application faisant usage de la `libftdi` (code 1 l.7-10) :

- `ftdi_new` (l.9) initialise le contexte `ftdi` nécessaire à l'utilisation du `FT232` depuis l'espace utilisateur. Il contient entre autres un pointeur vers la `libusb` utilisée par la `libftdi` ;
- la seconde fonction (l.10) détache le périphérique, spécifié par son `vendor_id` et son `product_id`, du pilote `ftdi_sio`. À partir de cette instruction, le composant n'est plus accessible par le pilote (disparition du nœud `/dev/ttyUSB*`).

Comme le but de cette application est de modifier l'`EEPROM`, et non de la reconstruire, l'étape suivante est d'obtenir le contenu courant pour n'en remplacer que certaines parties (1 l.11-12) :

- `ftdi_read_eeprom` (l.11) lit le contenu binaire de l'`EEPROM` et le stocke dans un tableau interne de la sous-structure `ftdi_eeprom` contenue dans la structure `ftdi_context` ;
- `ftdi_eeprom_decode` (l.12) analyse le contenu du tableau et remplit l'ensemble des champs de la structure `ftdi_eeprom` du contexte courant.

À ce stade, si tout s'est bien déroulé (récupération de la configuration du `FT232` et décodage de celle-ci) il devient possible de changer, dans un premier temps les valeurs des divers champs de la structure utilisée pour générer le nouveau contenu de l'`EEPROM` (l.14). La fonction permettant d'atteindre ce résultat est `ftdi_set_eeprom_value` dont la signature est :

```
int ftdi_set_eeprom_value(struct ftdi_context *ftdi, enum ftdi_eeprom_value value_name,→
    ↪ int  value);
```

avec `value_name` qui prend, dans notre cas, la `CBUS_FUNCTION_X` où X correspond au numéro de la broche. L'argument `value`, dans le cas de la configuration des broches `CBUS`, peut valoir :

- BUS_TXDEN ;
- CBUS_PWREN ;
- CBUS_RXLED

2. Utilitaire disponible dans le dépôt https://github.com/trabucayre/ft232_cbus_config

— CBUS_TXLED
— CBUS_TXRXLED
— ... ;
— CBUS_IOMODE ;
— ...

Les modifications réalisées grâce à la fonction ci-dessus ne sont appliquées que sur les champs de la structure `ftdi_eeprom`, le tampon et le `FT232` ne sont pas impactés. L'étape suivante va donc être d'utiliser `ftdi_eeprom_build`(l.16) pour reconstruire le binaire, puis de l'écrire physiquement avec `ftdi_write_eeprom`(l.17)

Avant de libérer les ressources et de s'arrêter, le programme fait appel à la fonction `libusb_reset_device` (l.19), fonction de la `libusb`, qui nécessite l'utilisation directe de l'attribut `usb_dev` (de type `libusb_context`). Cette fonction a pour but de forcer la réinitialisation du périphérique, qui ainsi prend immédiatement en compte les modifications apportées précédemment (dans le cas contraire il serait nécessaire de débrancher puis de rebrancher le `FT232` pour avoir le même résultat).

Finalement, le programme appelle la fonction `ftdi_free` (l.20) pour libérer le périphérique, désallouer l'ensemble des ressources allouées dynamiquement dans la structure `ftdi_context`, et libérer celle-ci.

Attention : Dans la portion de code ci-dessus les tests ont été supprimés pour réduire sa taille. Dans la réalité les valeurs de retour des fonctions `ftdi_read_eeprom`, `ftdi_decode_eeprom`, `ftdi_eeprom_build` et `ftdi_write_eeprom` doivent être absolument vérifiées. En effet, si une de ces étapes venait à échouer, le tampon utilisé en interne par la `libftdi` pourrait être corrompu ou contenir exclusivement des '0'. [a]. L'ensemble des paramètres, dont le `VENDOR_ID` et le `PRODUCT_ID`, seraient mis à '0', rendant le composant inexploitable dans le mode classique. Bien entendu ce n'est pas une situation irréversible grâce à la `libftdi` [b].

a. Effet équivalent à http://hackaday.com/2014/10/22/watch-that-windows-update-ftdi-drivers-are-killing-fake-chips/
b. C'est d'ailleurs ainsi que certains ont pu faire revivre leurs composants http://tech.scargill.net/ftdi-bricked-chips-fix/

4 Clignotement de LEDs

Très classiquement, comme pour la découverte de tout nouveau matériel électronique numérique, la première étape est de jouer avec les `GPIOs` (l'autre option étant la communication au travers du port série, mais dans le cas d'un convertisseur `USB-UART`, cette approche présente un intérêt quelque peu limité).

Dans notre cas, ce premier exemple va permettre de valider la configuration de l'`EEPROM` du `FT232`, et de voir comment changer l'état d'une des broches `CBUS`. Les LEDs sont connectées aux broches `CBUS0` et `CBUS1`.

L'ouverture, depuis l'espace utilisateur, du `FT232` étant exactement comme pour l'outil de configuration, cette partie n'est plus présentée ici (code 2). Ensuite,

```
1 struct ftdi_context ftdic;
  unsigned char value=0;
3 /* ouverture du ftdi */
  open_device(&ftdic);
5
  for (i = 0; i< 10;i++) {
7   value = (value +1)&0x03;
    ftdi_set_bitmode(&ftdic, 0x30 | value, BITMODE_CBUS);
9   sleep(1);
  }
```

Listing 2 – Utilisation des broches CBUS

le changement de l'état des broches se fait à l'aide de la fonction ftdi_set_bitmode(struct ftdi_context →*desc, unsigned char bitmask, unsigned char mode);. Le `bitmask` est composé de la façon suivante :

— les quatre bits de poids fort donnent la direction (0 entrée, 1 sortie) pour chaque broche,
— les quatre bits de poids faible donnent l'état des broches.

Cela se traduit, dans notre cas, par les broches `CBUS0` et `CBUS1` en sortie, les autres en entrée.

L'exploitation des broches `CBUS` du composant est bien pratique, surtout en parallèle de l'utilisation du port série (communication exclusivement faite grâce à la `libftdi` tant que l'application est en cours d'utilisation et que le pilote n'a pas repris la main sur le périphérique, le nœud dans `/dev` ayant disparu), mais les possibilités sont relativement limitées par le faible nombre de `GPIO` disponibles et leur utilisation

relativement lente. La section suivante présente un cas d'utilisation où ces broches présentent un réel intérêt et permettent de gagner du temps en manipulant les broches d'un microcontrôleur pour le passer du mode exécution au mode programmation.

5 Utilisation pour programmer un microcontrôleur

L'utilisation des broches CBUS est une solution pratique mais présente des limitations tant en terme de GPIOs à disposition, qu'en terme de vitesse. Un autre usage de ces broches est clairement plus intéressant

Celui-ci, qui fut à l'origine de l'étude de ce composant, est de simplifier la programmation d'un microcontrôleur. En effet, n'importe qui ayant déjà travaillé avec un composant tel que le `STM32`[3], l'`ADuC7026`[4], ou le `SAM3` par exemple, aura remarqué rapidement l'aspect pénible de devoir, à chaque chargement du firmware par le port série, manipuler un cavalier ou un bouton puis appuyer sur `reset` pour passer le CPU dans le mode programmation. Il apparaît très vite qu'une solution pour automatiser cette manipulation est nécessaire (la flemme est une qualité), mais aussi pour permettre une programmation à distance du composant sans la présence d'un opérateur à proximité du circuit, par exemple pour une télémaintenance. Un autre cas où l'automatisation du passage en mode programmation est utile est lorsque le composant se trouve dans un boîtier : en effet, dans ce type de situations, pouvoir accéder aux boutons, ou jumpers, nécessite l'ouverture de la boîte, là encore une situation pénible à la longue.

La première approche consisterait à utiliser les signaux de contrôle du port UART (qui, souvent, ne sont pas exploités pour les communications à courte distance) pour réaliser cette étape. Toutefois, un problème que nous avons rencontré est qu'alors qu'à l'initialisation ces broches sont configurées à un certain état (généralement un état haut ou bien haute impédance), certains émulateurs de terminaux tels que `minicom`, même si les signaux de contrôles ne sont utilisés, vont mettre ces broches à l'état inverse, pouvant entraîner un passage en mode programmation non souhaité[5]. C'est justement dans ce type de situation que les broches `CBUS` apparaîssent comme une bonne solution car elles ne sont manipulées par personne !

Ayant identifié la solution, il ne reste plus qu'à adapter les outils à cet usage. Trois solutions sont envisageables :

- modifier ou ajouter un module dédié. L'intérêt est d'avoir une solution assez simple, que ce soit pour la modification de l'outil dédié ou la création d'un simple script. Un contributeur au noyau Linux à proposé une telle modification[6]. Cependant, à ce jour, elle n'est pas disponible dans le noyau. Il est bien évidemment possible d'appliquer le patch correspondant mais ceci devra être fait à chaque mise à jour et pour chaque ordinateur. Ceci peut rapidement devenir laborieux et ne constitue pas une solution pérenne dans le temps ;
- la deuxième solution consiste à modifier l'outil de programmation pour utiliser entièrement la `libftdi`. Au-delà de la complexité de ce travail, les mises à jour de l'outil risquent d'être difficiles. Par ailleurs ce travail devra être réalisé pour chaque outil, ce qui n'est pas une option réaliste quand on a plusieurs microcontrôleurs différents ;
- la dernière solution consiste en la création d'un outil simple qui gérera les deux broches : il sera exécuté avant le lancement de l'outil de programmation et après la fin du chargement pour remettre le CPU dans l'état d'exécution.

La dernière solution, celle retenue car actuellement la plus simple implique plusieurs difficultés. Plusieurs `FTDI` peuvent être connectés : il faut donc prendre la main sur celui correspondant au nœud fourni à l'outil de programmation. Le second problème est que la `libftdi` détache le pilote empêchant à l'outil de réaliser sa tâche.

Automatiser le passage en mode programmation d'un microcontrôleur ne consiste pas simplement à raccorder une ou deux broches et à écrire un petit utilitaire. Il est également nécessaire de s'assurer que le niveau des signaux appliqués, par défaut, au microcontrôleur lui permette de démarrer dans le mode normal. En effet, du côté `FTDI` les broches vont avoir un certain niveau après des potentielles transitions

3. section 3 de www.st.com/resource/zh/application_note/cd00164185.pdf
4. section 7 de www.analog.com/aduc7xxxgetstarted?doc=ADuC7120-7121.pdf
5. ce problème est présent sur les premiers `arduino`
6. https://lkml.org/lkml/2015/6/20/205

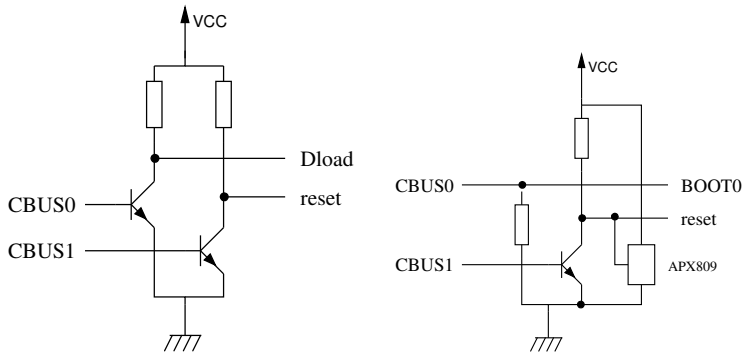
et du côté CPU les tensions appliquées doivent avoir un niveau bien particulier pour qu'il puisse démarrer normalement.

Le code complet est disponible sur le dépôt https://github.com/trabucayre/ftdi_cpu_prog.

5.1 Aspects matériels

Pour l'aspect matériel nous allons donner deux exemples de montage.

Le premier microcontrôleur, l'`ADuC7026`, passe en mode programmation en appliquant sur la broche `DLOAD` un état bas, puis en validant par un passage à l'état bas de la broche `reset`. Les broches du `FT232` étant, avant lecture de l'`EEPROM`, à l'état bas et ensuite flottantes (sans doute en entrée), nous avons utilisé des transistors `2N2222` montés en inverseur pour garantir que l'état de `DLOAD` et `reset` soient bien haut par défaut. Ce montage est présenté sur la figure 2(a). Par défaut ou si les broches `CBUS` sont à l'état bas les bases des transistors ne sont pas saturées, les broches sont tirées à `VCC` par des résistances. Lorsqu'une broche `CBUS` passe à l'état haut, le transistor devient passant, ramenant la broche de `l'ADuC` à la masse. Il nous faudra donc, dans le cadre de l'outil, définir les broches du `FT232` comme étant inactives à l'état bas.



(a) Montage pour un `ADuC7026`. (b) Montage pour un `STM32`.

FIGURE 2 – Montage pour l'adaptation des signaux de `FT232` pour la programmation d'un `ADuC7026` 2(a) et d'un `STM32` 2(b).

Le second cas, celui du STM32 (figure 2(b)), diffère sur deux points :

— la broche `BOOT0` (équivalente à `DLOAD`) est active à l'état haut. Compte tenu de l'état par défaut des broches `CBUS` une simple résistance de `pulldown` suffira. Ainsi, comme ce signal n'est pas inversé, les mêmes états actifs doivent être fournis à l'outil.

— le second point est relatif à l'instabilité des broches au moment du démarrage. Le `STM32` semble analyser `BOOT0` très rapidement et passe directement en mode programmation lors de l'alimentation de la carte (le même problème se présente pour le `SAM3`). Nous avons ajouté un circuit de reset (un `APX809`) afin de retarder le démarrage du microcontrôleur.

5.2 Partie logicielle

Une fois les signaux du convertisseur correctement adaptés aux contraintes du processeur cible, il reste encore à réaliser la partie logicielle. Globalement cette application ne présente pas de grandes nouveautés par rapport aux exemples de manipulations de broches `CBUS`.

Pour cette application nous voulons :

— accéder à un composant particulier même si plusieurs composants du même type sont connectés à un même ordinateur. Ceci doit se faire à partir du nœud fourni pour l'outil de programmation ;

— laisser l'utilisateur choisir quelle broche du `FTDI` correspond à quelle broche du microcontrôleur et leur état respectif au repos. Ceci pour que l'outil soit le plus généraliste possible.

Nous n'allons pas, dans la suite, tout détailler mais plutôt nous focaliser sur les points nouveaux : comment prendre la main sur le composant à partir de son nœud et comment rendre le contrôle au pilote `ftdi_sio` après exécution de l'outil.

Prendre la main sur un convertisseur au travers de son entrée dans `/dev` n'est pas nativement supporté par la `libftdi`, celle-ci ne peut prendre en charge les composants qu'à partir des identifiants ou du numéro de bus. Ceci présente un inconvénient majeur puisqu'il est difficile d'identifier un convertisseur particulier si plusieurs sont connectés sur l'ordinateur. D'autre part il est nécessaire de faire le lien entre le pseudo-fichier utilisé par l'outil de programmation (stm32flash [7], aducloader [8], bossa [9], ...) et le convertisseur.

La solution pour pallier ce problème, grandement inspirée de `USBDetach`[LM157], consiste à utiliser la `libudev` pour récupérer l'ensemble des informations nécessaires pour réaliser la prise de contrôle du périphérique de manière automatique. Cette série de manipulations peut sembler complexes mais n'est finalement pas trop volumineuse en terme de code.

Pour le premier point nous n'allons pas rentrer dans les détails car, d'une part, il est largement inspiré de `USBDetach`, et d'autre part, c'est un aspect qui dépasse le simple cadre de l'utilisation du `FT232`. Par ailleurs, les informations sont présentées dans l'article de `Denis Bodor` et les sources étant disponibles sur un dépôt `github`, le lecteur intéressé par cet aspect pourra s'y référer pour de plus amples détails (fichier `serial_ftdi.c`).

Pour résumer, la série d'opérations suivantes est nécessaire afin d'obtenir les informations utilisées par la `libftdi` pour accéder à un composant particulier sans ambiguïté, à partir d'un nœud, si plusieurs convertisseurs sont connectés :

- la fonction `stat` permet, à partir du chemin absolu d'un nœud, d'obtenir le couple `majeur/mineur` de celui-ci ;
- ces informations permettent d'utiliser `udev` pour retrouver les champs `vid`, `pid` et `serial` ;
- finalement en utilisant la fonction ftdi_usb_open_desc(ftdi , vid, pid, NULL, serial) nous prenons la main sur le composant.

La technique semble assez laborieuse, mais nous n'avons pas, à l'heure actuelle, trouvé de méthode plus simple pour réaliser la même chose.

Par rapport à `USBDetach` nous n'utilisons pas le numéro de device qui est relatif au bus sur lequel est connecté le composant. Sachant que deux composants peuvent avoir le même numéro mais être sur deux bus différents nous risquerions de ne pas prendre la main sur le bon périphérique.

Ayant accès à notre composant, il devient maintenant possible de réaliser la séquence correspondante à la mise en mode programmation du microcontrôleur. Comme précisé précédemment et afin d'avoir un outil souple, ces informations sont fournies sur la ligne de commande. Cet aspect dépasse largement le cadre de cette prose (se référer à la fonction `get_check_params` sur le dépôt).

Arrive maintenant le moment de rendre la main et surtout de faire en sorte que le pilote associé reprenne le contrôle afin de faire réapparaître le pseudo-fichier `/dev/ttyUSBx`.

Pour ce faire, nous avons copié et modifié la fonction `ftdi_usb_close` (renommée en `serial_close`) afin d'ajouter :

```
if (ftdi->module_detach_mode == AUTO_DETACH_SIO_MODULE) {
  rtn = libusb_attach_kernel_driver(ftdi->usb_dev, ftdi->interface);
}
```

Nous cassons l'aspect "opaque" de la structure `ftdi_context` en faisant un accès direct à `module_detach_mode`, afin de savoir si le module est automatiquement détaché et à `usb_dev` qui est le context `libusb` utilisé par la `libftdi`. La fonction `libusb_attach_kernel` a pour rôle de forcer le noyau à redécouvrir le périphérique et ainsi, de rendre la main au pilote `ftdi_sio`.

Nous avons maintenant un outil qui pilote les `CBUS` associées aux broches du microcontrôleur, soit pour le passage en mode programmation, soit pour un simple *reset* après l'écriture du firmware. Afin de rendre transparente l'utilisation de l'application nous pouvons, par exemple, renommer l'outil dédié et créer un script du nom de celui-ci. Par exemple pour le `STM32` nous avons un script correspondant à :

```
#!/bin/sh
DEV=""
for i in $@; do
    echo $i | grep "tty" > /dev/null
    if [ $? = 0 ]; then
        DEV=$i
    fi
done
```

7. https://sourceforge.net/projects/stm32flash
8. https://cyclerecorder.org/aducloader/
9. https://github.com/shumatech/BOSSA

```
9 ftdi_cpu_prog -r 2 -rd 0 -b 1 -bd 0 -d $DEV -m 0
  _stm32flash $@
11 ftdi_cpu_prog -r 2 -rd 0 -b 1 -bd 0 -d $DEV -m 1
```

Dans ce script nous cherchons le `/dev/ttyxx`, nous appelons ensuite notre outil en specifiant le numéro de la broche de reset (-r), son état par défaut/inactif (-rd), celle connectée à `BOOT0` (-b et -bd), le pseudo fichier (-d) et specifions 0 pour le passage en mode programmation (-m). Nous faisons ensuite appel à l'application, puis faisons à nouveau appel à l'outil avec le dernier paramètre à 1 pour réaliser un simple reset.

À l'usage, nous avons également pu exploiter cet outil pour d'autres cartes telles que les `teensy3.1` et les `HUZZAH ESP8266`.

6 Utilisation en mode BitBang

L'utilisation des broches `CBUS` est une solution pratique pour disposer d'une poignée de broches, tout en ayant à disposition un port `UART`. Toutefois ce mode présente des limitations tant au niveau du nombre de `GPIOs` à disposition que de la vitesse. Bien entendu, tel que présenté précédemment, dans le cas de la configuration d'un microcontrôleur, ces limitations n'ont pas d'importance car il n'y a pas de réelles contraintes temporelles et seulement 2 broches sont nécessaires.

Le mode `Bitbang` pallie partiellement cette limitation car il permet d'utiliser les 8 broches de l'UART comme de simples `GPIOs` et de transférer non plus un état mais une suite de transitions. Dans ce mode de fonctionnement il n'est plus possible d'exploiter le FTDI en tant que convertisseur USB-UART, mais il n'est également pas possible d'exploiter les broches `CBUS` en `GPIO`. Elles seront donc réservées pour une exploitation selon une des autres fonctions disponibles pour celles-ci.

Pour montrer l'intérêt de ce mode et son usage pour du prototypage rapide, nous allons présenter deux cas d'utilisation :

- la mise en œuvre d'un afficheur compatible `HD44780`, qui présente l'avantage de communiquer sur un bus parallèle ;
- l'utilisation d'un écran TFT Nokia à base de contrôleur `Epson S1D15G00` qui va nécessiter l'émulation du protocole `SPI`.

6.1 Utilisation basique du mode BitBang

Mais avant cela, nous allons reprendre l'exemple précédent pour faire clignoter une LED connectée sur la broche TX du convertisseur (code. 3).

Dans ce mode, la fonction `ftdi_set_bitmode` n'est plus utilisée pour configurer la direction et l'état des broches. Son rôle dans ce mode est exclusivement de donner la direction pour les 8 broches. En conséquence elle ne sera appelée qu'une seule fois, au début de l'application (code 3 l.12) :

- l'octet en second paramètre fixe la direction de chacune des 8 broches du port UART ;
- le mode va modifier le comportement du composant. Dans le cas de `BITMODE_BITBANG` les broches du port vont passer en mode parallèle.

La fonction `ftdi_set_bitmode` ne réalisant que le changement de mode et la configuration de la direction, c'est la fonction `ftdi_write_data` (code 3 l.15) qui est utilisée pour changer l'état des broches. Cette fonction, utilisée en temps normal pour fournir le ou les octets à transférer sur la broche TX va permettre, pour chaque octet du tableau, de fournir l'état de l'ensemble des broches. Avec le mode `bitbang` il est ainsi possible de transférer dans la mémoire du `FT232` une suite d'états que vont prendre les broches, et non plus exclusivement un seul ensemble, ceci permettant d'avoir un meilleur rapport temps de transfert/changement d'états.

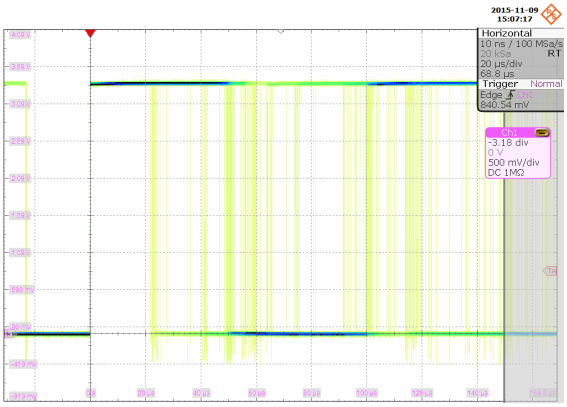
```
1 int main(void)
  {
3   int i;
    struct ftdi_context ftdic;
5   unsigned char buf = 0x00;

7   if (open_device(&ftdic) != 0)
      return EXIT_FAILURE;
9   if (ftdi_set_baudrate(&ftdic, 9600) < 0)
      return EXIT_FAILURE;
11
```

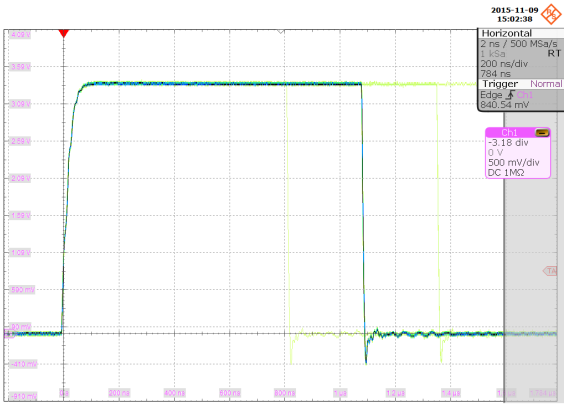
```
  ftdi_set_bitmode(&ftdic, 0x01, BITMODE_BITBANG);
13
  for (i = 0; i< 10000000;i++) {
15   ftdi_write_data(&ftdic, &buf,1);
     buf^=0xff;
17 }
  ftdi_disable_bitbang(&ftdic);
19
  ftdi_free(&ftdic);
21 return EXIT_SUCCESS;
}
```

Listing 3 – Exemple de mise en œuvre du mode `bitbang` sur un `FT232RL`.

À la différence des broches `CBUS`, la configuration du débit de communication (l.9-10 3) a un impact sur le signal généré (bien que dans le cas présent, et tel que présenté sur la figure 3, en ne transférant qu'un seul octet, ou un seul état pour l'ensemble des broches, ce soit l'ordinateur qui est le facteur limitant). Attention, tel que présenté dans le document [AN232R] en mode `bitbang` le `baudrate` est multiplié par 16, ce qui donnera un changement d'état des broches, si un tableau est envoyé, toutes les $\frac{1}{baud \times 16}$s (ce n'est pas magique mais du très certainement au multiplieur interne nécessaire à la synchronisation de l'UART).



(a) Signal carré réalisé par une série d'envois des états de la broche, avec appel systèmatique à la fonction `ftdi_write_data`.



(b) Signal carré généré par l'utilisation d'un tableau de 128 octets correspondant aux états successifs des broches du FT232

FIGURE 3 – Comparaison entre l'envoi de 128 octets successifs (3(a)) et par l'envoi d'un tableau décrivant les 128 états successifs (3(b). Le `baudrate` utilisé est de 57600 bps. Dans le premier cas, le changement d'état se fait (le plus souvent) toutes les 40 μs avec des fluctuations très importantes ne permettant pas de déterminer le pire cas. Dans le second cas, les transitions se font toutes les $1,1$ μs avec des fluctuations de ± 250 ns. La période correspond bien, dans le second cas, à $\frac{1}{57600 \times 16}$ s.

À la fin de l'exécution de l'application, et pour disposer à nouveau du composant dans sa fonction première, il est nécessaire de faire appel à la fonction `ftdi_disable_bitbang` (3 l.18).

Le résultat de ce premier exemple est présenté sur la figure 3(a).

Cet exemple, en utilisant un pointeur sur un simple `char` réalise la même fonction que son équivalent avec une broche `CBUS` et présente donc les mêmes problèmes de délais. Par ailleur il ne tire pas au mieux profit de la possibilité de transférer une suite d'état en utilisant les 128 octets de tampon interne du composant.

Attention : La documentation du composant fait référence à *FIFO RX Buffer* et à *FIFO TX Buffer*. Le nommage est du point de vu de l'USB. En d'autres termes le premier correspond au tableau pour l'envoi en UART et le second pour la réception.

En remplaçant les lignes 14 à 17 du code 3 par :

```
char buf=0x01;
2unsigned char buf2[128];
int i;
4for (i=0; i<DATA_SIZE; i++) {
```

```
    buf2[i]=buf;
    buf^=0x01;
}

for (i=0; i < 1000; i++)
    ftdi_write_data(&ftdic, buf2, 128);
```

il est possible d'obtenir des changements d'états tous les $\frac{1}{baud \times 16}$ s, toutefois une latence existe entre les transferts.

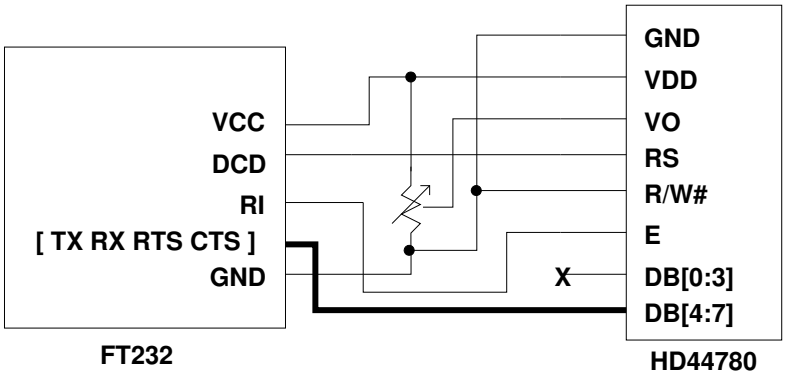
7 Application à un afficheur basé sur un HD44780

Au travers de l'exemple précédent nous avons vu comment utiliser le mode `bitbang` (configuration des broches et passage dans ce mode, envoi des états). La première application concrète d'utilisation du `FT232` est la communication avec un afficheur de type caractères basé sur un contrôleur compatible `HD44780`[HD44780]. Notre écran présente une ligne de 16 caractères, ce qui correspond, du point de vue du contrôleur, à 2 lignes de 8 caractères. En effet, le `HD44780` gère uniquement 8 caractères par ligne.

La communication avec un contrôleur compatible `HD44780` fait partie des applications les plus simples. En effet, il reçoit ses données en parallèle sur 4 ou 8 broches selon la configuration et nécessite deux broches supplémentaires pour le contrôle :

- `RS` (*Register Select*) pour définir si l'envoi est une commande ou une donnée ;
- `E` (*Enable*) pour valider, sur le front descendant du signal, l'octet (ou le demi-octet) présenté sur le bus de données.

Le schéma de câblage est présenté sur la figure 4(a) et les offsets de chaque broche sur la figure 4(b). N'ayant pas suffisamment de broches à disposition sur le convertisseur pour une liaison sur 8 bits, nous avons câblé l'écran pour des transferts en 4 bits (deux transmissions nécessaires par octet) et nous avons connectés `R/W#` à la masse (mode écriture uniquement).



(a) schéma de câblage entre l'afficheur et le `FT232`

HD44780	FT232	*offset*
DB4	TX	0
DB5	RX	1
DB6	RTS	2
DB7	CTS	3
RS	DCD	6
E	RI	7

(b) Correspondance entre les broches du `FT232`, celles de l'afficheur et l'offset de chaque broche

FIGURE 4 – Connexions entre un afficheur `HD44780` et un `FT232`. 4(a) : schéma de câblage, 4(b) : relation entre les broches et offset

La lecture de la documentation du contrôleur[HD44780] reporté et annoté sur le chronogramme 5 nous permet de déterminer qu'il n'y a pas de contraintes de temps entre `RS` et `DB`, qu'un délai minimum doit être respecté entre l'envoi de la donnée et la mise à l'état haut de `E` ainsi que sur la durée de cet état, et finalement qu'une durée minimale doit être respectée avant de modifier `DB`. Nous devons donc tenir compte du délai le plus long pour fixer le `baudrate` et donc séquencer correctement la production de données, du `FTDI`. Cette durée correspond à l'état haut de `E` soit 450 ns.

Toutefois, là encore la documentation précise ([HD44780] page 24) que selon les commandes envoyées, le temps d'exécution sera d'environ 37 μs pour la plupart de celles-ci, hormis lors de la remise à 0 du

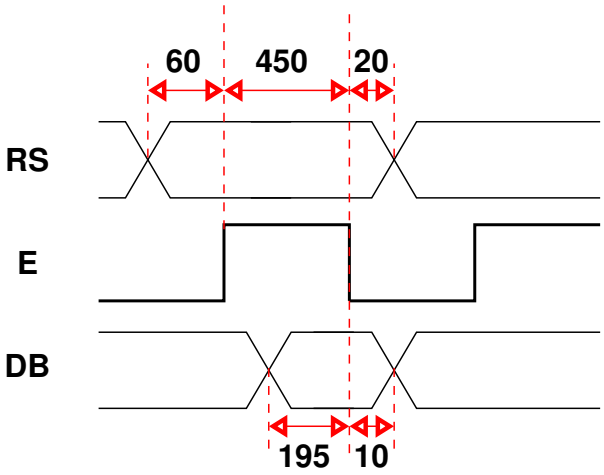


FIGURE 5 – Chronogramme de l'évolution des signaux dans une communication avec le contrôleur `HD44780`. Les durées correspondent au temps minimum entre la transition des signaux, et sont exprimées en ns.

pointeur sur la mémoire vidéo, dont la durée est de 1,57 ms. Ainsi, nous pouvons déterminer qu'une commande complète peut-être envoyée dans un seul paquet. Par contre, là encore compte tenu du temps d'exécution et de la durée entre la mise à jour de l'état des broches du `FT232`, vouloir envoyer toute une chaîne de caractères directement, dans un seul tampon, ne sera pas fonctionnel. Par ailleurs, en se référant aux tests présentés dans la figure 3(a), nous constatons qu'il n'est pas nécessaire d'ajouter de délais après l'envoi d'un ordre (hormis pour se repositionner en début de mémoire).

7.1 Implémentation

La base pour communiquer, que ce soit pour envoyer une commande ou une donnée, ne diffère que par l'état de la broche `RS`. Ainsi nous pouvons réaliser la fonction `hd44780_send` (code 4) qui va, au travers d'un tableau, réaliser l'ensemble des étapes successives pour l'envoi d'un ordre.

```
#define RS (1<<6)
void hd44780_send(struct ftdi_context *ftdic,
    uint8_t e, uint8_t rs) {
  unsigned char buffer[6];
  unsigned int cnt=0;
  /* msb */
  unsigned char val = (e >> 4) | rs;
  buffer[cnt++] = val;
  buffer[cnt++] = val | E;
  buffer[cnt++] = val;
  /* lsb */
  val = (e & 0x0F) | rs;
  buffer[cnt++] = val;
  buffer[cnt++] = val | E;
  buffer[cnt++] = val;
  /* flush */
  ftdi_write_data(ftdic, buffer, cnt);
}
void send_char(struct ftdi_context *ftdic, unsigned char e) {
  hd44780_send(ftdic, e, RS);
}
void send_cmd(struct ftdi_context *ftdic, unsigned char e) {
  hd44780_send(ftdic, e, 0);
}
```

Listing 4 – fonction de transfert de la série d'états des broches nécessaires pour la transmission d'un ordre vers l'afficheur

Cette fonction prend en paramètres un pointeur sur un `struct ftdi_context` pour l'accès au convertisseur, l'octet à envoyer, ainsi que la valeur de la broche `RS` (l.2-3). Elle définit un compteur ainsi qu'un

tableau pour stocker les états successifs des broches.

Le contrôleur s'attend à des commandes sur 8 bits, notre interface est en 4 bits : deux transferts seront donc nécessaires (l.7-10 et l.12-15). Mais comme dans le mode 4 bits l'ordre n'est pris en compte qu'après avoir reçu les bits de poids faible, un seul transfert entre l'ordinateur et le `FT232` est nécessaire (l.17).

Sachant que grâce à la configuration du `baudrate` nous respectons les durées de la séquence d'envoi nous pouvons simplement remplir un tableau qui correspond aux états de chaque broche pour chaque étape :

— la valeur de `DB` et de `RS` (l.7-8 et l.12-13) ;

— le passage à l'état haut (l.9 et 14) puis bas de `E` (l.10 et 15).

Ce remplissage se fait en premier lieu pour le quartet de poids fort de l'octet à envoyer, puis le quartet de poids faible.

Une fois le tampon rempli, il ne reste qu'à utiliser la fonction `ftdi_write_data` (l.17) pour envoyer les `cnt` octets.

Nous avons également défini deux fonctions pour envoyer les caractères ou des commandes (l.19-24) qui ne se différencient que par la valeur de l'argument `rs` (soit utilisation du "define" `RS` défini en l.1 soit 0).

La seconde étape (code 5) à réaliser est l'initialisation du contrôleur : la séquence qui doit être envoyée est documentée à la page 46 du document de référence[HD44780]. Le chronogramme est repris sur la figure **??**.

```
void init_lcd(struct ftdi_context *ftdic) {
  unsigned char buf[] = {0x03, 0x03 | E, 0x03};
  unsigned char buf2[] = {0x02, 0x02 | E, 0x02};
  ftdi_write_data(ftdic, buf, 3);
  usleep(4500);
  ftdi_write_data(ftdic, buf, 3);
  usleep(100);
  ftdi_write_data(ftdic, buf, 3);
  usleep(100);
  ftdi_write_data(ftdic, buf2, 3); /* 4 bits */
  usleep(100);
  send_cmd(ftdic, 0x28);
  send_cmd(ftdic, 0x0c); /* displ line and caract */
  send_cmd(ftdic, 0x01); /* clear displ */
}
```

Listing 5 – fonction d'initialisation du contrôleur pour une communication en 4 bits

Par défaut le `HD44780` est en 8 bits, c'est pourquoi les quatres premiers ordres sont envoyés avec la fonction `ftdi_write_data`. Toutefois nous envoyons la série d'états, de chaque commande, en un paquet.

À partir de l'émission de la commande `0x02`, l'afficheur passe en mode 4 bits. Il nous est désormais possible d'utiliser la fonction définie précédemment. La suite de commandes passée va configurer l'écran en 2 lignes (pour mémoire le contrôleur gère 8 caractères par ligne donc pour un écran affichant 16 caractères nous avons bien 2 lignes), affichage de caractères, pas de curseurs ni de clignotement, puis vide la mémoire (se reporter aux commentaires du code et à la documentation pour de plus amples explications).

Nous avons maintenant l'écran prêt à afficher des chaînes de caractères. Il ne reste donc plus qu'à écrire la fonction nécessaire (code 6) :

```
void sendString(struct ftdi_context *ftdic, char *string, int size) {
  int i;
  send_cmd(ftdic, 0x01); /* clear display */
  send_cmd(ftdic, 0x02); /* return home */
  usleep(1000);
  for (i=0; i< size; i++) {
    send_char(ftdic, string[i]);
    if (i==7) send_cmd(ftdic, 0xc0);
  }
}
```

Listing 6 – Fonction destinée à transférer une chaîne de caractères pour affichage

La fonction est assez simple : la mémoire est vidée et le pointeur d'adresse replacé à la position initiale (l.3-5), puis une attente est effectuée. Les caractéres sont ensuite envoyés.

`Note :` le contrôleur est capable de gérer jusqu'à 80 caractères ([HD44780] page 2 et [HD44780AddressingMode]) mais l'organisation des lignes est laissée à la discrétion du fabricant. Ainsi dans le cas de notre écran, qui présente une ligne de 16 caractères et qui est manifestement de Type 1 ([HD44780AddressingMode]), nous devons envoyer la commande `0xC0` qui correspond à l'adresse `0x40` dans le registre *set DDRAM address*(`0x80`) (en mode 2 lignes la première ligne commence à 0x00 jusqu'à 0x27 et la seconde va de 0x40 jusqu'à 0x67 (page 29)) pour passer du huitième au neuvième caractère.

Avec cette dernière fonction nous avons présenté une implémentation d'un code de communication avec le contrôleur `HD44780` au travers d'un `FT232`, permettant de manière simple de disposer d'un afficheur connecté à un ordinateur (figure 6).

L'approche mise en œuvre devrait, pour ceux ayant fait usage d'un port parallèle, sembler familière puisque la manipulation se fait avec des masques binaires pour gérer indépendamment les broches.

FIGURE 6 – Affichage d'un classique "hello world" sur un écran compatible `HD44780` piloté par un `FT232` en mode `bitbang`

8 Émulation d'un protocole série : écran Nokia SPI

L'exemple précédent nous a permis de mettre en œuvre, de manière relativement aisée, la communication avec un périphérique dont le protocole est de type parallèle. Toutefois, dans la panoplie des protocoles actuels, la plupart sont de type série et un des plus fréquents est le SPI (*Serial Peripheral Interface*).

Pour illuster la mise en œuvre de ce protocole, à l'aide d'un convertisseur `FT232`, nous allons nous pencher sur le cas de l'écran Nokia 6100 qui fut acheté chez `Sparkfun`[10]. Ce dernier présente une résolution de 12 bits (4 bits pour le rouge R, le vert G et le bleu B) pour une taille de 132x132 pixels et est contrôlé par un `Epson S1D15G00`.

Son originalité est liée à la configuration matérielle du contrôleur qui nécessite une communication par séquence de 9 bits (les 8 bits de poids faible correspondent à la commande/donnée et le bit de poids fort au type de transfert, équivalent à la broche `RS` du `HD44780`). L'envoi de pixels se fait toujours par deux, ainsi en 3 transferts de 9 bits nous obtenons bien les 24 bits des deux pixels consécutifs ([Nokia] p.7). Le contenu des trois transferts (hors bit de poids fort) sont le suivant :

1. R1R1R1R1G1G1G1G1
2. B1B1B1B1R2R2R2R2
3. G2G2G2G2B2B2B2B2

10. https://www.sparkfun.com/products/retired/569

Dans cette section, nous n'allons pas présenter l'ensemble de la mise en œuvre du code pour piloter le composant mais allons exclusivement nous concentrer sur les quelques fonctions nécessaires pour la communication. Le reste (configuration et traitements divers) est donné dans la documentation ou dans le dépôt lié à cet article (le résultat de l'exemple est visible sur la figure 7).

8.1 Implémentation

Contrairement au `HD44780` le volume de données à transférer est bien plus important : le pire cas, à savoir l'envoi de la totalité des pixels pour les 132 lignes × 132/2 colonnes × 3 mots (le /2 correspond au transfert des deux pixels consécutifs et le ×3 correspond au fait que cet envoi est réalisé en trois opérations), correspond à un total de 25344 paquets de 9 bits, soit, dans notre cas, un total de 506880 octets puisque chaque bit nécessite deux transitions de l'horloge ainsi que les deux transitions du *chip-select*. Pour limiter, au mieux, le nombre de transactions avec le convertisseur, nous avons tiré profit de la `FIFO` du composant. Grâce à elle nous avons obtenu, expérimentalement, un gain d'un facteur 16.

Nous allons donc mettre en place deux fonctions :

— une pour transférer le contenu d'un tampon vers le `FT232` (code 7) ;

— une seconde pour stocker dans le tableau un mot de 8 bits correspondant à l'état des broches du convertisseur. Si le tampon est plein, elle fait appel à la première fonction avant de réaliser le stockage (code 8).

L'intérêt d'une fonction d'envoi indépendante est de pouvoir, à tout moment, vider le tampon même s'il n'est pas plein. Ceci évitera des comportements surprenants.

Une fois ces deux fonctions disponibles, il nous sera possible de réaliser l'émulation du protocole SPI.

8.1.1 Transfert du tampon

```
static unsigned char data_buff[128];
static int buff_cnt;
void ftdi_flush(struct ftdi_context *ftdic){
  if (buff_cnt == 0) return;
  ftdi_write_data(ftdic, data_buff, buff_cnt);
  buff_cnt = 0;
}
```

Listing 7 – Fonction utilisée pour envoyer le contenu d'un tampon contenant les états des broches afin de réduire les latences de communications avec le convertisseur.

Cette fonction vérifie que le tampon n'est pas vide en testant la valeur de la variable globale représentant le nombre d'octets stockés. Si le tampon n'est pas vide, il est envoyé et le compteur est remis à 0.

8.1.2 Stockage des états des broches

```
void ftdi_add_char(struct ftdi_context *ftdic, unsigned char val) {
  if (buff_cnt == 255) ftdi_flush(ftdic);
  data_buff[buff_cnt++] = val;
}
```

Listing 8 – fonction utilisée pour stocker les états successifs des broches.

Là encore, la fonction vérifie l'état du compteur, mais cette fois-ci afin de ne pas dépasser la capacité du tampon. Si le tampon est plein, la fonction d'envoi est appelée. Ensuite le nouveau mot est stocké et le compteur incrémenté.

8.1.3 Émulation du protocole SPI

Le SPI émulé ici est en `Mode 0` signifiant que le signal d'horloge est à l'état bas au repos, les données échantilonnées sur le premier front (montant) et mises à jour sur le front suivant (descendant).

```
void ft232_emul_spi(struct ftdi_context *ftdic, unsigned char data,
    unsigned char rs) {
  unsigned char j, val=LCD_RES;
  if (rs == 1) val |= LCD_DIO;
  ftdi_add_char(ftdic, val);
  val |= LCD_SCK;
  ftdi_add_char(ftdic, val);
  for (j = 0; j < 8; j++) {
    if ((data & 0x80) == 0x80) val |= LCD_DIO;
    else val &= ~LCD_DIO;
```

```
      val &= ~LCD_SCK;
12    ftdi_add_char(ftdic, val);
      val |= LCD_SCK;
14    ftdi_add_char(ftdic, val);
      data <<= 1;
16  }
  val |= LCD_CS;
18  ftdi_add_char(ftdic, val);
}
```

Listing 9 – fonction d'émulation du protocole SPI en mode 0 pour des transferts de 9 bits.

Cette fonction prend en argument la commande à envoyer, codée sur 8 bits, ainsi que l'information liée au type de transfert (commande ou donnée).

L'ensemble des manipulations va consister à des `ou`, des `et` et des `non` pour changer l'état des broches de manière indépendante (comme on le ferait classiquement sur un port série ou sur un microcontrôleur). Nous utilisons donc une variable `val` de type char que nous assignons par défaut avec un état haut sur la broche de `reset`

8.1.4 Exemple d'utilisation : remplissage de l'écran

L'implémentation de toute la communication et de la configuration dépasse le cadre de cet article. Nous allons toutefois présenter un exemple de fonction assez représentatif du mécanisme mis en œuvre dans l'ensemble (envoi de commande et de données).

```
1 void fill_screen(struct ftdi_context *ftdic, int color)
  {
3   int x, y;
    LCDCommand(ftdic, PASET);
5   LCDData(ftdic, 0);
    LCDData(ftdic, 131);
7
    LCDCommand(ftdic, CASET);
9   LCDData(ftdic, 0);
    LCDData(ftdic, 131);
11
    LCDCommand(ftdic, RAMWR);
13  for (y=0; y < 131; y++) {
      for (x=0; x <132/2; x++) {
15      LCDData(ftdic, (color>>4)&0x00FF);
        LCDData(ftdic, ((color&0x0F)<<4)|(color>>8));
17      LCDData(ftdic, color&0x0FF);
      }
19  }
    ftdi_flush(ftdic);
21 }
```

Listing 10 – fonction chargée de remplir l'écran avec une couleur.

Le code de la fonction `fill_screen` (10) a pour rôle de faire le remplissage total de l'écran avec une seule couleur passée en paramètre.

Les deux fonctions utilisées `LCDCommand` et `LCDData` sont de simples facilités qui permettent d'éviter, à travers tout le code, de passer le paramètre `rs`.

Ce code est relativement simple à comprendre et n'est finalement pas du tout spécifique au `FT232` :

- de la ligne 4 à 10 : configuration de la zone mémoire qui va être addressée (PASET pour les pages mémoires et CASET pour les colonnes). En l'état, la totalité de l'écran est concerné.
- en ligne 12 la commande pour une écriture dans la mémoire est envoyée ;
- puis il ne reste plus qu'à transférer séquentiellement les deux pixels consécutifs dans l'ordre attendu dans la configuration de l'écran de test.

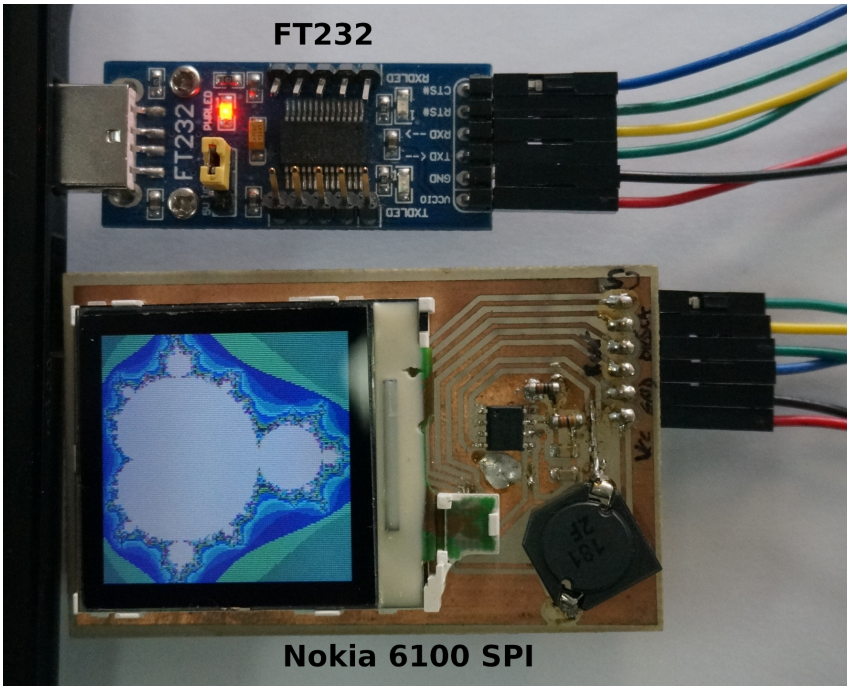


FIGURE 7 – Affichage d'une fractale de `Mandelbrot` sur un écran `Nokia 6100` basé sur un contrôleur `Epson S1D15G00`. Cet écran est piloté par un `FT232` en mode `bitbang` émulant le protocole `SPI`.

`Note :` La motif affichée sur l'écran dans la figure 7 est la fractale de Mandelbrot [mandelbrot] calculée par itération (avec un nombre maximum de 1000) de la suite complexe $z_{n+1} = z_n^2 + c$ avec c la cordonnée dans le plan complexe, donc d'abscisse sa partie réelle et d'ordonnée sa partie imaginaire. La fractale est affichée en attribuant une couleur pour l'indice n de l'itération pour laquelle le module de z_n dépasse un certain seuil qui indique la divergence de la suite – ce seuil est classiquement sélectionné à 2. Dans l'exemple de la figure, les points blancs (à l'intérieur de la fractale) indiquent que la suite ne diverge pas, tandis que les motifs vers l'extérieur indiquent une divergence de plus en plus rapide, *i.e.* en moins d'itérations pour atteindre le seuil critique. La fractale a la propriété de se retrouver lors de zooms dans diverses régions, tel que le lecteur pourra expérimenter par exemple avec `xaos`.

9 Remarque sur la lecture en mode `Bitbang`

Dans l'ensemble des exemples présentés dans cet article il n'est jamais fait de lectures. Il nous a semblé, malgré tout, important d'ajouter une note à ce sujet.

Le mode utilisé jusqu'à présent (`BITMODE_BITBANG`) est un mode asynchrone : dès lors que des données sont disponibles dans le tampon du `FT232` elles sont consommées. Ainsi, pour lire l'état d'une broche (`MISO` pour le SPI par exemple) il devient nécessaire de découper le transfert en portion afin de réaliser une lecture lors du front correspondant à l'échantillonnage (selon le mode). L'effet immédiat est de se retrouver dans une situation quasi équivalente à l'utilisation des broches `CBUS`.

Toutefois, le `FT232` propose un mode synchrone ([AN232R] section 3) utilisable avec la fonction `ftdi_set_bitmode` en lui passant la constante `BITMODE_SYNCBB` : dans ce cas le contenu du tampon d'envoi n'est consommé que suite à une requête de lecture (utilisation de la fonction `ftdi_read_data`). Il est à noter que le composant réalise un échantillonnage des signaux (et son stockage dans le tampon) avant de consommer une donnée du tampon d'écriture.

Ce mode, pour des transferts bi-directionnels, semble clairement plus intéressant pour ne pas perdre les avantages présentés dans les sections précédentes.

Pour illustrer ce mode de fonctionnement nous allons détailler un court exemple de gestion de la transmission d'un octet et de la réception de la réponse du périphérique (en l'occurence une eeprom `Atmel AT25DF`). Cet exemple n'est pas forcément optimal mais offre une vue d'ensemble de ce mode de

fonctionnement.

Pour ce faire la première étape est d'utiliser `BITMODE_SYNCBB` en lieu et place de `BITMODE_BITBANG` lors de l'appel de `ftdi_set_bitmode`.

Ensuite, ne reste qu'à réaliser une fonction telle que présentée en 11

```
1 uint8_t at25df_spi_write(struct ftdi_context *ftdic, uint8_t val) {
    uint8_t tmp[64], recv[64], r=0;
3   int j, cpt = 0;
    for (j = 0; j < 8; j++, val <<= 1) {
5     if ((val & 0x80) == 0x80) spi_val |= MOSI;
      else spi_val &=~ MOSI;
7     spi_val &= ~SCLK;
      tmp[cpt++] = spi_val;
9     spi_val|=SCLK;
      tmp[cpt++] = spi_val;
11  }
    ftdi_write_data(ftdic, tmp, cpt);
13  ftdi_read_data(ftdic, (uint8_t*)recv, cpt);
    for (j=1; j < cpt; j+=2)
15    r = (r << 1 ) | ((recv[j] & MISO)? 1 : 0);
    return r;
17 }
```

Listing 11 – fonction de transfert d'un octet avec gestion de la récupération d'un octet issu du périphérique.

Le début du code est identique à l'exemple précédent. La différence est ensuite :

- contrairement au mode `BITMODE_BITBANG` le fait d'envoyer des données dans le tampon du `FT232` ne déclenche pas le transfert au niveau des broches du composant. Celui-ci attends une requête de lecture (utilisation de la fonction `ftdi_read_data`) (l.13).
- lorsque cette fonction rends la main, le tableau `recv` contient les états successifs des broches pour chaque octet écrit. Ainsi il ne reste plus qu'à récupérer les états pertinents de la broche correspondant au signal `MISO`.

La partie reconstruction de l'octet reçut présente deux finesses :

1. la boucle commence au deuxième octet : en SPI, la première transaction consiste à mettre à jour la ligne de donnée (sans changement d'état de `SCLK` si `CPHA`=0 ou sur la première transition si `CPHA`=1). Le premier bit à échantillonner est donc toujours sur la seconde transaction.
2. l'itération se fait par pas de 2 : une transmission sur 2 étant la mise à jour des broches `MISO` et `MOSI` leur état ne nous intéresse pas (en fait elles sont dans un état, potentiellement, non stable).

Le reste est relativement trivial à comprendre : l'octet est reconstruit par ajouts successifs d'un bit (l.15) (dans le cas présent la transmission est considérée `MSB first`) dont la valeur correspond au contenu d'un élément du tableau lu.

10 Conclusion

Nous avons vu qu'un convertisseur USB-série – le `FT232` – utilisé en général pour la communication série asynchrone, dispose de fonctionnalités intéressantes telles que fournir des ports numériques d'interfaces généralistes (GPIO). Nous avons également vu comment se passer de Windows pour le configurer, comment s'interfacer avec des composants communiquant par protocole série synchrone SPI par exemple, mais aussi automatiser le passage en mode programmation d'un microcontrôleur. L'outil présenté, même si à l'origine développé pour les `FT232`, peut parfaitement grâce à l'utilisation du nœud au lieu de son couple `vid/pid`, être utilisé avec d'autres convertisseurs tels que le `FT230X` qui dispose également des broches `CBUS`. Les exemples n'ont porté globalement que sur l'utilisation en espace utilisateur, mais il est envisageable, modulo le changement de VID ou PID, de réaliser son propre pilote pour, par exemple, écrire sur le LCD au travers d'un pseudo fichier dans `/dev`.

Codes associés disponibles sur : https://github.com/trabucayre/lm_ftdi

Références

[FT232DS] FTDI, *datasheet du FT232* www.ftdichip.com/Support/Documents/DataSheets/ICs/DS_FT232R.pdf

[AN232R] FTDI, *Bit Bang Modes For The FT232R and FT245R Application Note AN_232R-01* http://www.ftdichip.com/Support/Documents/AppNotes/AN_232R-01_Bit_Bang_Mode_Available_For_FT232R_and_Ft245R.pdf

[HD44780] *datasheet du HD44780* https://www.sparkfun.com/datasheets/LCD/HD44780.pdf

[HD44780AddressingMode] *LCD Addressing* http://web.alfredstate.edu/weimandn/lcd/lcd_addressing/lcd_addressing_index.html

[LM125] D. Bodor, *Personnalisation d'un adaptateur série/USB FTDI*, GNU/Linux Magazine France **125** (mars 2010)

[OS2] D. Bodor, *Parlez 1-Wire, I2C, SPI, MIDI et bien plus avec un seul outil : Bus Pirate v3*, Open Sillicium **2** (avril-mai-juin 2011)

[LM157] D. Bodor, *Programmation udev et libusb en C : USBdetach*, GNU/Linux Magazine France **157** (février 2013) https://github.com/Lefinnois/USBdetach

[Nokia] James P. Lynch, *Nokia 6100 LCD Display Driver* https://www.sparkfun.com/tutorial/Nokia_6100_LCD_Display_Driver.pdf

[mandelbrot] H.O Peitgen, H. Jürgens, D. Saupe, *Chaos and Fractals – New frontiers of Science*, Springer (1992)