Dialogue en SPI avec un MSP430*

Christophe BLAESS

Nous avons vu dans l'article précédent le support SPI proposé par le Raspberry Pi. Nous allons mettre ceci en pratique en communiquant avec un petit microcontrôleur simple à programmer : le MSP430...

Microcontrôleur MSP430

Le microcontrôleur *Texas Instruments MSP430* a déjà été présenté dans Linux Magazine, il est simple à programmer, très répandu et peu coûteux. Les outils de programmation (compilateur, débogueur, etc.) sont directement disponibles sous forme de packages pour la plupart des distributions, y compris pour la Raspbian.

Il existe une petite carte de développement simple et bon marché nommée *MSP430 Launchpad*, que l'on peut commander pour 9,99$ directement sur le site web de Texas Instruments (**http://www.ti.com/tool/msp-exp430g2**). Cette carte se connecte par un port USB sur le PC de développement ce qui permet de l'alimenter et de programmer le microcontrôleur.

Voici comment installer les outils de programmation du MSP430 sur une distribution dérivée de Debian.

```
$ sudo apt-get install gcc-msp430
[…]
The following extra packages will be installed:
  binutils-msp430 msp430-libc msp430mcu
[…]
Do you want to continue [Y/n]? y
$ sudo apt-get install mspdebug
```

J'utiliserai ci-après ces outils de compilation en ligne de commande (car c'est le plus simple à transmettre par écrit), mais il est tout à fait possible d'ajouter un *plug-in* pour Eclipse sur PC Linux afin d'avoir un environnement de développement graphique confortable et agréable d'emploi. Pour cela il faut sélectionner dans le menu ***Help*** l'option ***Install new software***, puis saisir l'URL **http://eclipse.xpg.dk** dans le champ ***Work with***. Ceci doit être réalisé après l'installation des outils ci-dessus.

À l'opposé,on peut également choisir de programmer le MSP430 directement depuis le Raspberry Pi (puisque les outils sont disponibles avec Raspbian), mais je ne le conseille pas forcément pour débuter, afin d'éviter les confusions d'environnements.

La documentation sur la programmation des microcontrôleurs de la famille MSP430 se trouve sur le site de *Texas Instruments*, il s'agit d'un document PDF intitulé **`slau144`**.

Compilation d'un programme C pour MSP430

Les exemples de cet articles se trouvent sur un dépôt *Github*, on peut les télécharger ainsi :

```
pi@raspberrypi:~$ git  clone  https://github.com/cpb-/Article-RPi-MSP430.git
  [...]
pi@raspberrypi:~$ cd  Article-RPi-MSP430/
```

Voici un premier programme qui réécrit en permanence sur sa sortie **`MISO`** les données qu'il a reçues sur son entrée **`MOSI`**. On peut remarquer que le bit de configuration **`UCCKPH`** représente l'inverse de la phase **`CPHA`** SPI habituelle. C'est une particularité de ce microcontrôleur.

```
// msp430-spi-1.c :

#include <stdlib.h>
#include <msp430g2553.h>
```

* Cet article est paru dans Gnu/Linux Magazine Hors Série numéro 75 « *Raspberry Pi Avancé* » en novembre 2014, disponible sur `https://boutique.ed-diamond.com/anciens-numeros/789-gnulinux-magazine-hs-75.html`

```
int main(void)
{
   int val = 0;

   // Arrêter le watchdog.
   WDTCTL = WDTPW + WDTHOLD;

   // Attendre que l'horloge SPICLK soit au repos (niveau bas).
   while ((P1IN & BIT4)) ;

   // Fonctions secondaires pour les broches P1.1 (MISO) P1.2 (MOSI) P1.4 (CLK)
   P1SEL  = BIT1 + BIT2 + BIT4;
   P1SEL2 = BIT1 + BIT2 + BIT4;

   // Réinitialiser et placer le contrôleur SPI en mode configuration.
   UCA0CTL1 |= UCSWRST;

   // Configuration SPI (voir slau144 p.445)
   // UCCKPH = 0
   // UCCKPL = 0
   //   SPI Mode 0 :  UCCKPH * 1 | UCCKPL * 0
   //   SPI Mode 1 :  UCCKPH * 0 | UCCKPL * 0  <-- Notre choix.
   //   SPI Mode 2 :  UCCKPH * 1 | UCCKPL * 1
   //   SPI Mode 3 :  UCCKPH * 0 | UCCKPL * 1
   // UCMSB  = 1 -> Bit poids fort en premier.
   // UC7BIT = 0 -> 8 bits, 1 -> 7 bits.
   // UCMST  = 0 -> esclave, 1 -> maître.
   // UCMODE_0 -> 3-pin SPI,
   // UCSYNC = 1 -> Mode synchrone (SPI).
   //
   UCA0CTL0 = UCCKPH*0 | UCCKPL*0 | UCMSB*1 | UC7BIT*0
         | UCMST*0 | UCMODE_0 | UCSYNC*1;

   // Activer l'UART.
   UCA0CTL1 &= ~UCSWRST;

   while (1) {
      // Attendre que le contrôleur SPI soit libre.
      while (UCA0STAT & UCBUSY)
         ;
      val = UCA0RXBUF;
      UCA0TXBUF = val;
   }
   return 0;
}
```

La compilation s'effectue tout simplement avec **`gcc`**, en précisant le type de microcontrôleur :

```
$ msp430-gcc  -Wall  -mmcu=msp430g2553    msp430-spi-1.c   -o msp430-spi-1.elf
```

Sinon, le fichier **`Makefile`** accompagnant les exemples effectue la compilation

```
pi@raspberrypi:~/Article-RPi-MSP430$ make
  [...]
```

Le nom du fichier de sortie importe peu. Comme il ne s'agit pas d'un exécutable qu'on lance directement sur le système, on a souvent coutume pour le développement à microcontrôleur d'utiliser des suffixes explicites (**`.hex`**, **`.bin`**, **`.elf`**, etc.) afin d'éviter les confusions lors du transfert vers la cible.

Programmation du microcontrôleur

Pour programmer l'image dans le microcontrôleur, on utilise l'outil **`mspdebug`** qui, comme son nom l'indique, permet également de faire du débogage basé sur GDB.

On commence par effacer la mémoire flash du MSP430 :

```
$ mspdebug  rf2500  erase
```

```
[…]
Erasing...
```

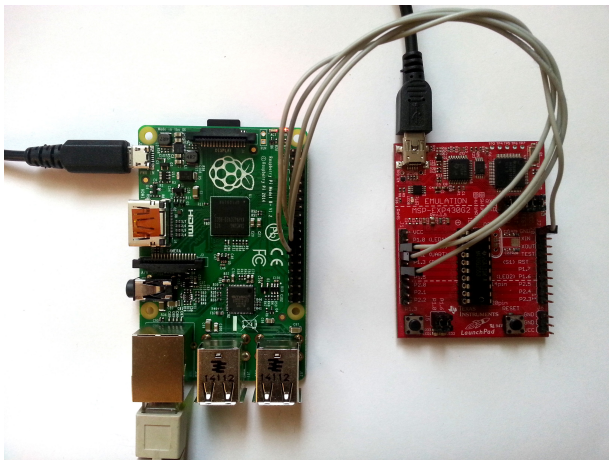
L'option **`rf2500`** précise le type de programmateur que l'on utilise, ici il s'agit du *Launchpad*.

Ensuite, on envoie le fichier image. Attention à bien regrouper la commande **`prog`** et le nom du fichier entre des guillemets, afin que cela ne constitue qu'un seul argument pour la commande **`mspdebug`** :

```
$ mspdebug   rf2500   "prog msp430-spi.elf"
[…]
Writing  188 bytes to c000 [section: .text]...
Writing   32 bytes to ffe0 [section: .vectors]...
Done, 220 bytes written
```

Le MSP430 redémarre immédiatement et le programme est prêt à dialoguer en SPI.

Connexions



La connexion entre le Raspberry Pi et le Launchpad MSP430 est très simple, il suffit d'utiliser quatre fils (GND, SCLK, MISO, MOSI).

Il existe plusieurs versions du MSP430, proposant un nombre plus ou moins important d'entrées-sorties, de convertisseurs, etc. Celle livrée avec la carte Launchpad est le MSP430 G 2553, qui dispose de deux ports d'entrées-sorties P1 et P2 dont les broches sont configurables en GPIO ou suivant des fonctions secondaires (comme port SPI, conversion analogique-numérique, etc.). La figure 1 représente les quelques broches qui nous seront utiles dans le cours de cet article.

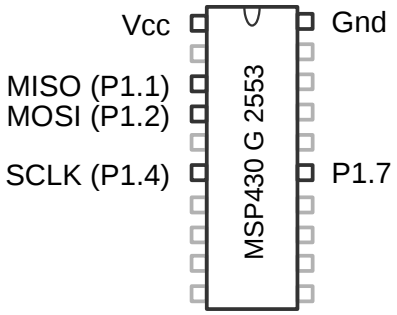


Fig.1 – Broches du MSP430 nous concernant.

Communications SPI

Nous relions les broches MISO, MOSI, SCLK et Gnd avec celles portant les mêmes noms sur le connecteur P1 du Raspberry Pi, comme sur la figure 2.

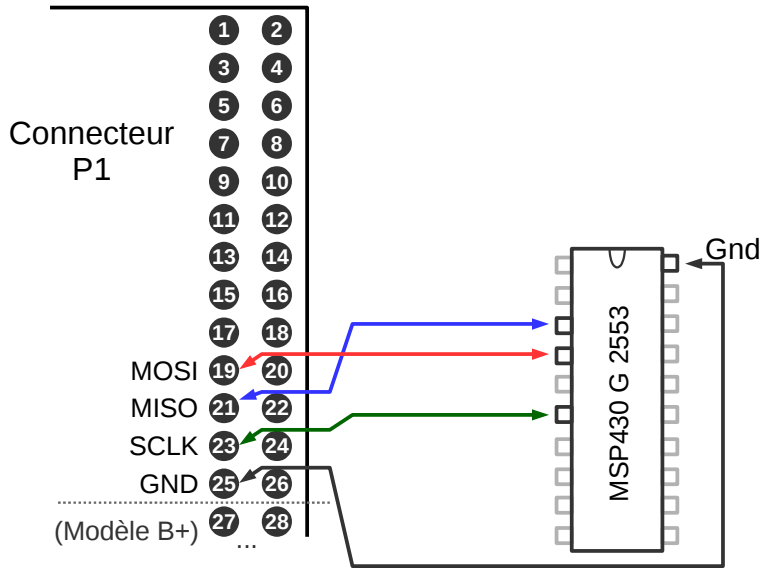


Fig. 2 - Connexions entre le Raspberry Pi et le MSP430

Les deux cartes étant alimentées, nous allons envoyer des caractères depuis le Raspberry Pi et vérifier s'ils nous sont bien renvoyés par le MSP430.

On considère que les modules **`spi-bcm2708`** et **`spidev`** ont été chargés dans le noyau du Raspberry Pi, comme indiqué dans l'article précédent, et que les outil **`spi-tools`** ont été compilés et installés.

Du côté du MSP430, nous supposons que le programme **`msp430-spi-1`** (voir plus haut) a été compilé et transféré dans la mémoire du microcontrôleur.

Outil spi-pipe

Le programme spi-pipe permet d'envoyer sur la ligne **`MOSI`** du SPI les données qu'il reçoit sur son entrée standard, tout en affichant simultanément sur sa sortie standard les données reçue depuis la ligne **`MISO`** du SPI.

Le principe général d'utilisation est :

```
$ <commande-1> | spi-pipe [options...] | <commande-2>
```

Rappelons que les trois membres du pipeline s'exécutent en parallèle, chacun dans un processus distinct en se synchronisant sur les données qui circulent entre eux.

Je configure le port en mode SPI numéro 1 comme nous l'avons prévu dans l'article précédent car cela évite de câbler le signal *Chip Select* lorsqu'il n'y a qu'un seul périphérique connecté.

```
pi@raspberrypi:~$ spi-config  -d /dev/spidev0.0  -m 1
pi@raspberrypi:~$ spi-config  -d /dev/spidev0.0  -q
/dev/spidev0.0: mode=1, lsb=0, bits=8, speed=500000
```

Je vais envoyer une chaîne de six caractères vers le MSP430. J'utiliserai la commande *shell* **`printf`** plutôt que **`echo`** car cela évite d'ajouter des caractères de sauts de lignes superflus.

Simultanément j'afficherai avec la commande **`hexdump`** les données que je reçois du MSP430. Je les afficherai sous forme hexadécimale (à gauche) et ASCII (à droite).

```
pi@raspberrypi:~$ printf "AZERTY"  |  spi-pipe  -d /dev/spidev0.0  |  hexdump -C
00000000  00 41 5a 45 52 54                           |.AZERT|
```

Nous voyons que MSP430 nous a bien renvoyé nos caractères, en les précédant d'un caractère nul qui correspond à l'état du registre à l'initialisation du contrôleur SPI. Rappelons-nous qu'à chaque envoi d'un

caractère nous recevons celui qui a été écrit dans le registre de sortie du MSP430 à l'itération précédente.

Nous pouvons recommencer avec une nouvelle chaîne de caractères pour en avoir le cœur net :

```
pi@raspberrypi:~$ printf "QSDFGH" | spi-pipe -d /dev/spidev0.0 | hexdump -C
00000000  59 51 53 44 46 47                           |YQSDFG|
```

Le premier caractère reçu (**Y**) est bien le dernier envoyé la fois précédente. On peut modifier légèrement notre programme pour que le MSP430 fasse un petit traitement avant de nous retourner notre chaîne. Par exemple incrémenter chaque caractère. La boucle centrale devient :

```
// msp430-spi-2.c :

   [...]
   while (1) {
      while (UCA0STAT & UCBUSY)
         ;
      val = UCA0RXBUF;
      val = (val + 1) & 0xFF;
      UCA0TXBUF = val;
   }
   [...]
```

Après compilation et transfert nous obtenons le résultat attendu :

```
pi@raspberrypi:~$ printf "AZERTY"  |  spi-pipe  -d /dev/spidev0.0  |  hexdump -C
00000000  01 42 5b 46 53 55                           |.B[FSU|
pi@raspberrypi:~$ printf "QSDFGH" | spi-pipe -d /dev/spidev0.0 | hexdump -C
00000000  5a 52 54 45 47 48                           |ZRTEGH|
```

Chaque caractère est bien incrémenté d'une unité, le **Z** devenant **[**.

Le défaut de ce code, est qu'il passe son temps à copier le registre d'entrée dans celui de sortie. Il serait plus judicieux d'attendre d'avoir reçu un nouveau caractère. C'est ce que réalise l'exemple suivant.

```
// msp430-spi-3.c :

   [...]
   while (1) {
      // Attendre qu'un caractère soit reçu.
      while ((IFG2 & UCA0RXIFG) == 0) ;
      val = UCA0RXBUF;
      while (UCA0STAT & UCBUSY) ;
      UCA0TXBUF = val;
   }
   [...]
```

Vu depuis le Raspberry Pi le comportement est identique :

```
pi@raspberrypi:~$ printf "AZERTYUIOP"  |  spi-pipe -d /dev/spidev0.0  |  hexdump -C
00000000  00 41 5a 45 52 54 59 55  49 4f                 |.AZERTYUIO|
```

Fonctionnement en interruptions

Le code programmé jusqu'ici dans le MSP430 est utilisable et fonctionne bien, mais il présente une caractéristique que l'on essaye généralement d'éviter : le travail en boucle active. Le microcontrôleur boucle en effet sans cesse sur la ligne de test du registre **IFG2**, en attendant le changement du bit **UCA0RXIFG**.

Ce dernier représente l'occurrence d'une condition d'interruption. Il s'agit plus précisément de la disponibilité d'un nouveau caractère dans le registre d'entrée **MOSI**. Il serait préférable que le CPU ne boucle pas (ce qui le fait chauffer et consommer de l'énergie inutilement) mais s'endorme en attendant que le contrôleur SPI lui signale la présence d'une telle condition. Il suffit pour cela d'écrire un gestionnaire d'interruption. Dans la syntaxe de programmation du MSP430, on remplacera la boucle active ainsi :

```
// msp430-spi-4.c :

   [...]
   // Activer l'UART.
   UCA0CTL1 &= ~UCSWRST;
```

```
    // Valider les interruptions de réception SPI
    IE2 &= 0xF0;
    IE2 |= UCA0RXIE;

    // Passer en mode économie d'énergie (avec interruptions)
    __bis_SR_register(LPM4_bits + GIE);
    return 0;
}


// La fonction d'interruption de réception SPI.
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR (void)
{
    int val;
    while ((IFG2 & UCA0RXIFG) == 0) ;
    val = UCA0RXBUF;
    val = (val + 1) & 0xFF;
    while (UCA0STAT & UCBUSY) ;
    UCA0TXBUF = val;
}
```

Le comportement est toujours identique vu depuis le Raspberry Pi, bien que cette fois, le microcontrôleur soit le plus souvent au repos :

```
pi@raspberrypi:~$ printf "QSDFGHJKLM" | spi-pipe -d /dev/spidev0.0 | hexdump -C
00000000  00 52 54 45 47 48 49 4b  4c 4d                    |.RTEGHIKLM|
```

Conversion analogique-numérique

Il peut être intéressant de coupler une carte à microprocesseur, comme le Raspberry Pi, avec un microcontrôleur pour réaliser différentes choses :

- augmenter le nombre d'entrées-sorties numériques disponibles pour le traitement programmé dans le microprocesseur,
- gérer des interruptions de manière plus prédictible, notamment sur des systèmes soumis à des contraintes temps-réel,
- disposer de *timers*, de compteurs automatiques, de générateurs d'impulsions en PWM (*Pulse Width Modulation*), de convertisseurs analogique-numérique et inversement.

C'est de cette dernière possibilité dont nous allons profiter, en tirant parti du convertisseur ADC (*Analog Digital Conversion*) présent dans le MSP430. Il y a plusieurs modèles d'ADC présents dans la gamme des MSP430, la version G 2553 contient un ADC 10 bits.

Celui-ci peut prendre en entrée l'une des 8 broches **`P1.0`** à **`P1.7.`** J'ai choisi la dernière car c'est la plus éloignée physiquement des signaux SPI, ce qui diminue les risques de confusion lors de la connexion.

On peut choisir différentes références pour les tensions minimale et maximale. Pour simplifier le test, nous utiliserons les valeurs Gnd et Vcc comme limites.

Pour faire notre essai, nous allons donc simplement connecter un potentiomètre de quelques kilo-Ohm par exemple entre la masse et la broche Vcc, et relier son point central à l'entrée **P1.7** comme sur la figure 3.

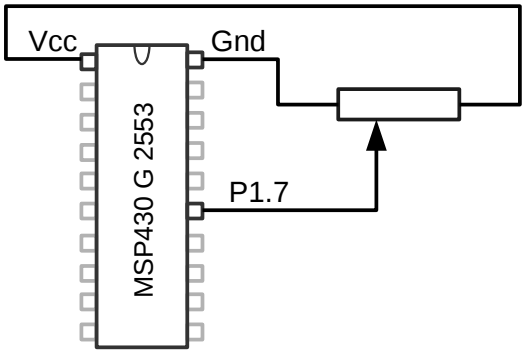


Fig. 3 – Potentiomètre sur l'entrée ADC 7

Bien entendu la connexion SPI avec le Raspberry Pi doit rester en place, même si elle n'est pas représentée sur la figure 3.

Le programme pour le MSP 430 est légèrement modifié pour configurer le convertisseur ADC10 et déclencher la première conversion. Lorsque celle-ci se termine, une interruption se produit, et notre routine de traitement ira remplir une variable globale avec la valeur mesurée.

Le gestionnaire d'interruption SPI qui est appelée lorsqu'un transfert est terminé ira lire la variable globale et l'écrire dans le registre de sortie **`MISO`**.

Le programme devient donc le suivant.

```
// msp430-spi-5.c :

#include <stdlib.h>
#include <msp430g2553.h>


int main(void)
{
   // Arrêter le watchdog.
   WDTCTL = WDTPW + WDTHOLD;

   // Attendre que l'horloge SPICLK soit au repos (niveau bas).
   while ((P1IN & BIT4)) ;

   // Fonctions secondaires pour les broches P1.1 (MISO) P1.2 (MOSI) P1.4 (CLK)
   P1SEL  = BIT1 + BIT2 + BIT4;
   P1SEL2 = BIT1 + BIT2 + BIT4;

   // Réinitialiser et placer le contrôleur SPI en mode configuration.
   UCA0CTL1 |= UCSWRST;

   // Configuration SPI(voir slau144 p.445)
   UCA0CTL0 = UCCKPH*0 | UCCKPL*0 | UCMSB*1 | UC7BIT*0
         | UCMST*0 | UCMODE_0 | UCSYNC*1;

   // Activer l'UART.
   UCA0CTL1 &= ~UCSWRST;

   // Valider les interruptions de réception SPI
   IE2 &= 0xF0;
   IE2 |= UCA0RXIE;

   // Entrée analogique A7 (broche P1.7)
   ADC10AE0 = BIT7;

   // Configuration ADC10 (voir slau144 p.553)
   ADC10CTL1 = INCH_7 | CONSEQ_2;
   ADC10CTL0 = ADC10ON | ADC10IE | ENC | ADC10SC;

   // Passer en mode économie d'énergie (avec interruptions)
   __bis_SR_register(LPM4_bits + GIE);
   return 0;
}

// Dernière valeur analogique mesurée
static int last_value = 0;

// La fonction d'interruption de réception SPI.
#pragma vector=USCIAB0RX_VECTOR
__interrupt void USCI0RX_ISR (void)
{
   int val;
   while ((IFG2 & UCA0RXIFG) == 0) ;
   val = UCA0RXBUF; // unused
   while (UCA0STAT & UCBUSY) ;
   UCA0TXBUF = (last_value >> 2); // 10 bits -> 8 bits
}
```

```
// La fonction d'interruption de fin de conversion ADC
#pragma vector=ADC10_VECTOR
__interrupt void ADC10_ISR(void)
{
   last_value = ADC10MEM;
   // Relancer une conversion ADC .
   ADC10CTL0 &= ~ENC;
   while (ADC10CTL1 & BUSY) ;
   ADC10CTL0 |= ENC | ADC10SC;
}
```

On notera qu'au moment de l'écriture sur le port SPI, la valeur mesurée sur dix bits est décalée vers la droite de deux bits (perdant ainsi les deux bits de poids faible) afin de tenir dans le registre huit bits. Si on voulait conserver la résolution, il faudrait envisager de transmettre la valeur sur deux octets successifs (deux bits de poids forts et huit bits de poids faibles).

Une fois ce programme flashé dans le microcontrôleur, nous pouvons l'interroger depuis le Raspberry Pi.

Comme le caractère reçu en entrée n'est pas utilisé (voir la fonction d'interruption de réception SPI), nous pouvons nous contenter de faire un simple appel système **`read()`** sur le fichier spécial **`/dev/spidev0.0`**. Ceci par exemple par l'intermédiaire du programme **`hexdump`** afin de voir les valeurs binaires reçues s'afficher en hexadécimal tandis que nous faisons varier la position du potentiomètre.

```
pi@raspberrypi:~$ hexdump -C /dev/spidev0.0
00000000  00 00 00 00 00 00 00 00  00 ff ff ff ff ff ff ff  |................|
00000010  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
00015000  ff ff ff ff ff ff ff fd  fd fd fd fd fd fd fd fd  |................|
00015010  fd fd fd fd fd fd fd fd  fd fd fd fd fd fd fd fd  |................|
*
00028000  fd fd fd fd fd fd f7 f7  f7 f7 f7 f7 f7 f7 f7 f7  |................|
00028010  f7 f7 f7 f7 f7 f7 f7 f7  f7 f7 f7 f7 f7 f7 f7 f7  |................|
*
00030000  f7 f7 f7 f7 f7 f7 f7 f7  f5 f5 f5 f5 f5 f5 f5 f5  |................|
00030010  f5 f5 f5 f5 f5 f5 f5 f5  f5 f5 f5 f5 f5 f5 f5 f5  |................|
*
[...]
```

Comme **`hexdump`** ne répète pas les lignes identiques (si on ne lui fournit pas l'option **`-v`**), cela nous permet de bien voir les fluctuations de la tension mesurée au fur et à mesure du mouvement du potentiomètre.

Nous ne contrôlons pas la valeur envoyée par le Raspberry Pi à chaque fois qu'il fait une lecture. Ceci n'est pas un problème dans notre cas, mais pourrait l'être dans d'autres situations. Pour cela on pourrait utiliser **`spi-pipe`** en lui fournissant la valeur à transmettre sur son entrée standard (par exemple zéro) :

```
pi@raspberrypi:~$ spi-pipe -d /dev/spidev0.0  < /dev/zero  | hexdump -C
00000000  ff d6 d6 d6 d6 d6 d6 d6  d6 d6 d5 d5 d5 d5 d5 d6  |................|
00000010  d6 d6 d6 d6 d6 d6 d6 d6  d6 d6 d6 d6 d6 d6 d6 d6  |................|
*
000001a0  d6 d6 d6 d6 d5 d5 d5 d5  d5 d5 d5 d5 d5 d5 d5 d4  |................|
000001b0  d4 d4 d4 d3 d3 d3 d3 d2  d2 d2 d1 d1 d0 d0 cf ce  |................|
000001c0  ce cd cc cc cb ca c9 c9  c8 c7 c6 c6 c5 c4 c3 c2  |................|
000001d0  c1 bf be bd bc bb b9 b8  b7 b6 b5 b3 b2 b1 b0 af  |................|
000001e0  ae ac ab aa a9 a7 a6 a5  a4 a3 a1 a0 9f 9e 9d 9c  |................|
000001f0  9b 99 98 97 96 95 94 93  92 91 90 8f 8e 8e 8d 8c  |................|
00000200  8b 8b 8a 89 89 88 87 87  86 85 85 84 84 83 83 82  |................|
```

```
00000210  82 81 81 80 80 7f 7e 7e  7d 7d 7d 7c 7b 7b 7a 7a  |......~~}}}|{{zz|
00000220  79 78 78 77 76 76 75 75  74 74 73 73 73 72 71 71  |yxxwvvuuttsssrqq|
00000230  71 71 71 71 71 71 71 71  71 71 71 71 71 71 71 71  |qqqqqqqqqqqqqqqq|
*
```

Conclusion

Nous avons vu qu'établir une connexion entre un Raspberry Pi et un microcontrôleur (MSP430 dans notre cas, mais cela pourrait être généralisable à la plupart des autres types) est plutôt simple à réaliser et permet des communications très rapides (on peut facilement augmenter la fréquence **`SCLK`** à plusieurs MHz).

L'étape la plus complexe en général est d'établir correctement la phase et la polarité d'horloge sur les deux processeurs. Ensuite, le dialogue est simple, si l'on prend bien en compte le fait qu'il s'établit toujours en full-duplex, chaque émission de donnée étant automatiquement accompagnée d'une réception.

L'intérêt, de ce type de montage est multiple : un microcontrôleur est généralement beaucoup moins coûteux qu'un microprocesseur, ses entrées-sorties sont mieux protégées contre les tensions parasites. Utiliser les GPIO d'un microcontrôleur externe plutôt que celles du Raspberry Pi permet de protéger ce dernier contre les inversions de polarité, les dépassements de plage de tensions, les courts-circuits, etc.

En outre, il peut être intéressant de sous-traiter les opérations d'entrées-sorties automatiques (comptage, décodage de protocoles, calcul de checksum, etc.) à un microcontrôleur pour laisser le processeur du Raspberry Pi libre d'effectuer des traitements plus complexes (interface utilisateur, réseau, statistiques, etc.)

L'interface SPI n'offre que la communication de bas-niveau et il est nécessaire d'ajouter un protocole de dialogue entre les applications. Je participe actuellement à un projet libre nommé LxMCU destiné à créer une API offrant côté Linux des périphériques caractères implémentant des files de messages, et côté microcontrôleur une bibliothèque de fonctions pour lire, écrire et être notifié des messages reçus. Le lecteur intéressé pourra me contacter directement pour plus de détails.

Christophe Blaess (christophe@blaess.fr)
www.blaess.fr/christophe