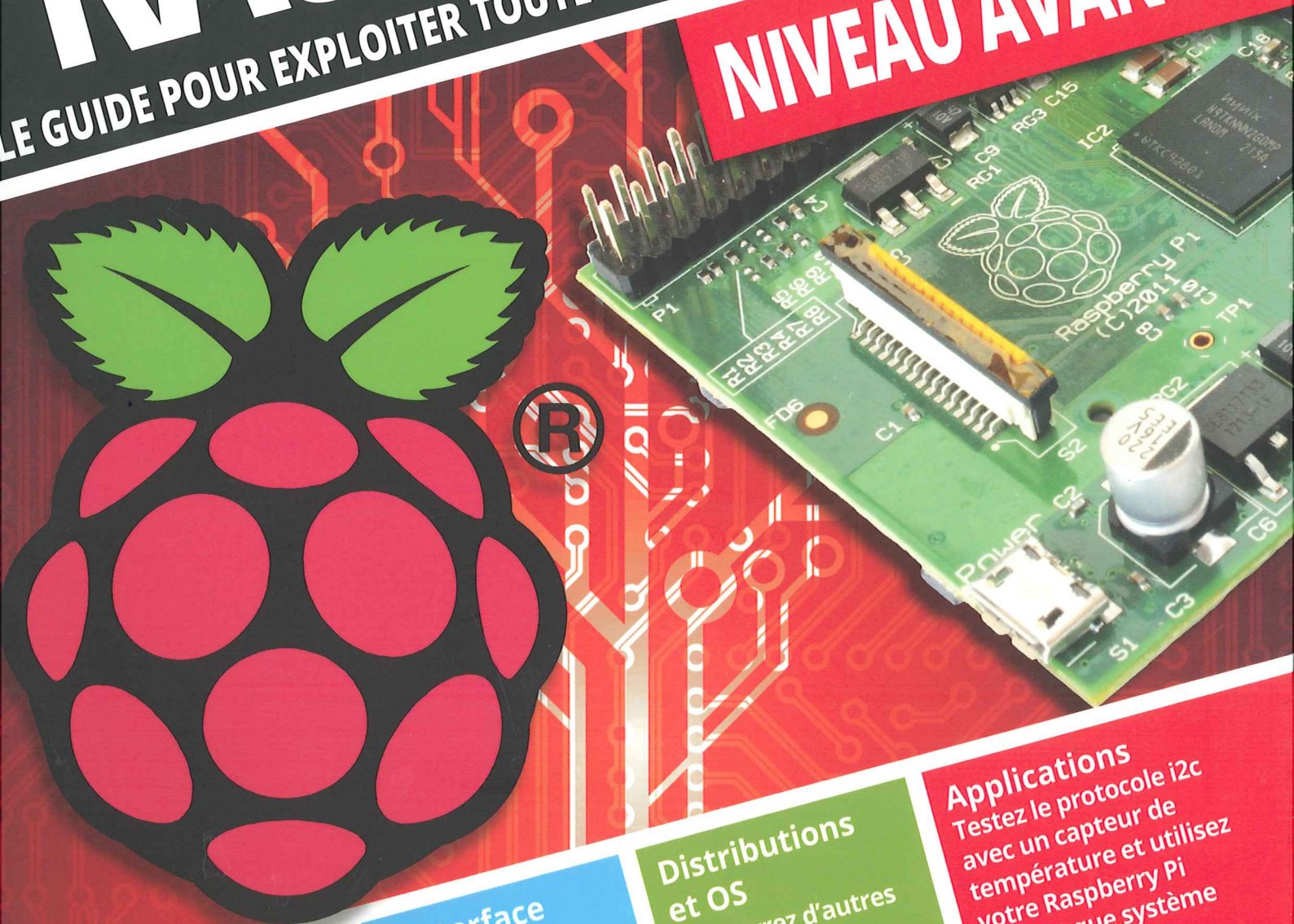
LES GUIDES DE
GNU LINUX MAGAZINE / FRANCE
HORS-SÉRIE
N°75

France METRO : 12,90 € — CH : 18,00 CHF — BEL/PORT.CONT : 13,90 € — DOM TOM : 13,90 € — CAN : 18,00 $ cad — MAR : 130 MAD

RASPBERRY PI

LE GUIDE POUR EXPLOITER TOUTE LA PUISSANCE DE LA RASPBERRY PI !

NIVEAU AVANCÉ



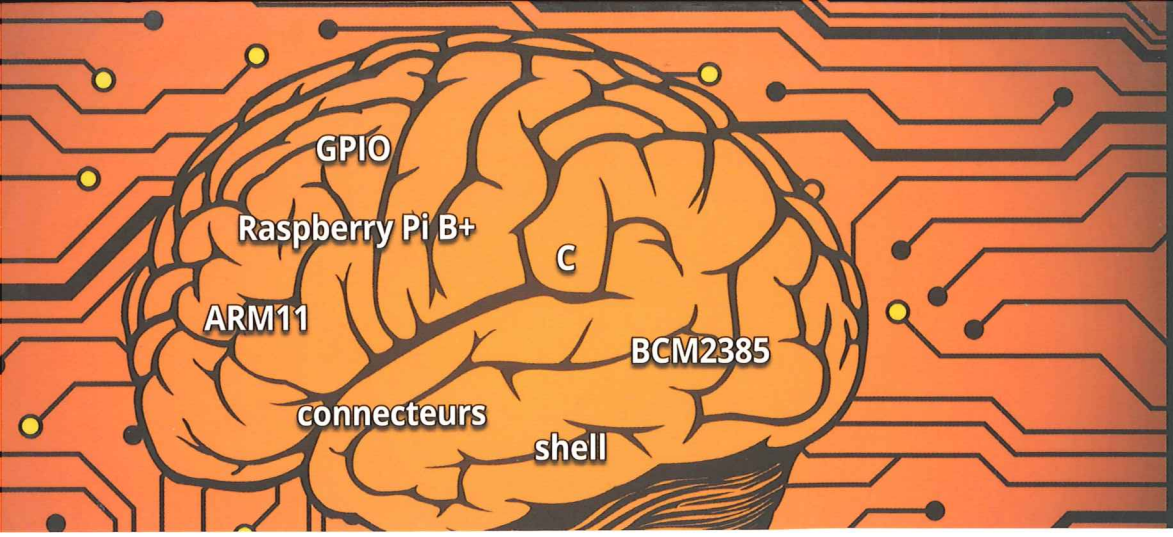
La carte et ses ports GPIO
Apprenez à utiliser la nouvelle Raspberry Pi B+ et ses ports GPIO en shell et en C

L'interface SPI
Utilisez le protocole SPI pour dialoguer rapidement et en full-duplex avec des périphériques

Distributions et OS
Découvrez d'autres systèmes que Raspbian ainsi que la compilation croisée depuis un autre système

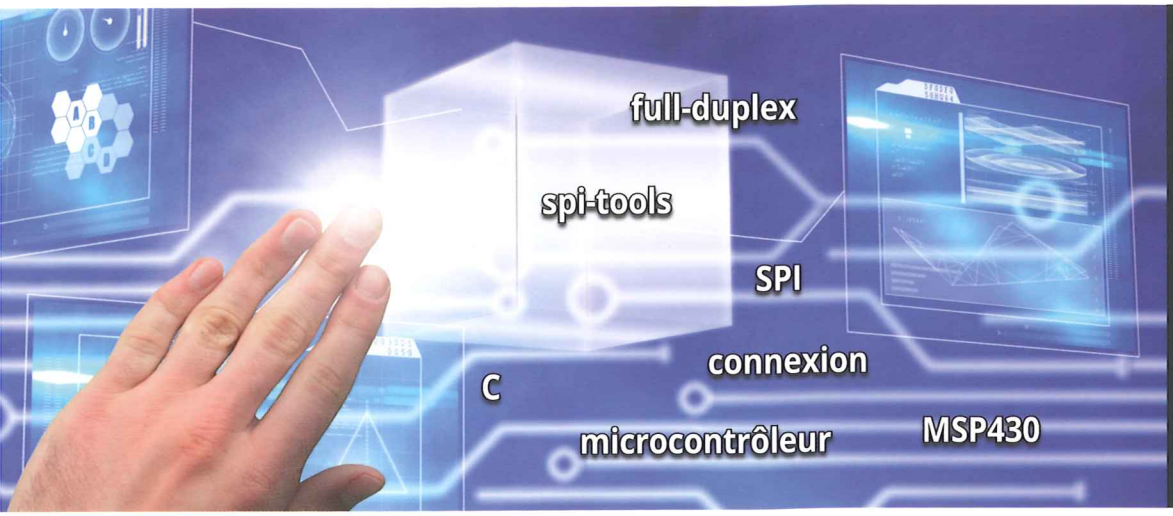
Applications
Testez le protocole i2c avec un capteur de température et utilisez votre Raspberry Pi en tant que système temps réel

Édité par Les Éditions Diamond
L 15066 - 75 H - F: 12,90 € - RD
www.ed-diamond.com



La carte et ses ports GPIO

Apprenez à utiliser la nouvelle Raspberry Pi B+ et ses ports GPIO en shell et en C

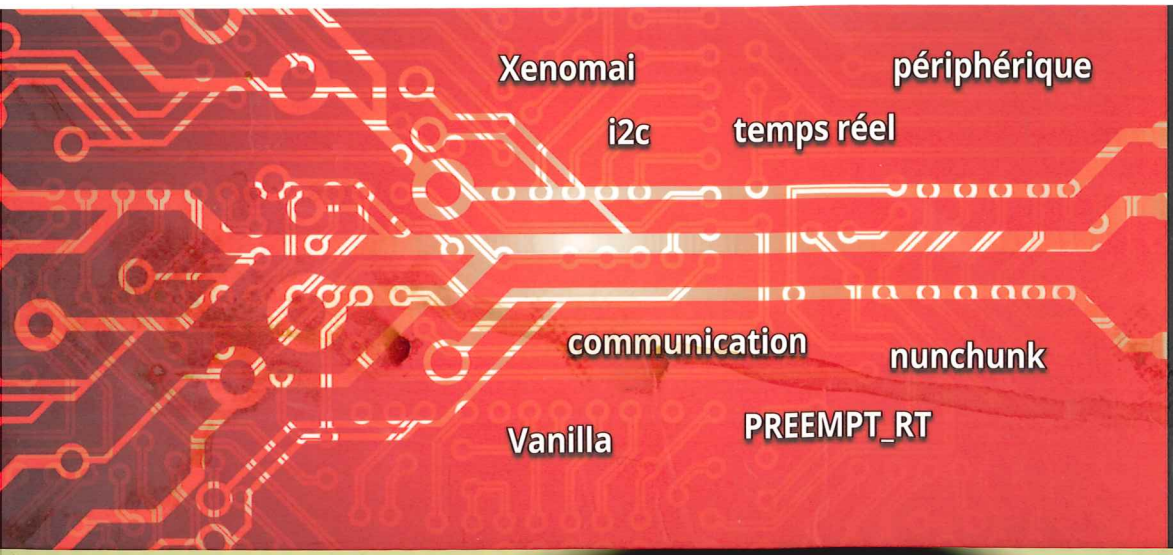


L'interface SPI

Utilisez le protocole SPI pour dialoguer rapidement et en full-duplex avec des périphériques

Distributions et OS

Découvrez d'autres systèmes que Raspbian ainsi que la compilation croisée depuis un autre système



Applications

Testez le protocole i2c avec un capteur de température et utilisez votre Raspberry Pi en tant que système temps réel

Retrouvez toutes nos publications
LES ÉDITIONS DIAMOND
sur **www.ed-diamond.com**

GNU/Linux Magazine Hors-Série
est édité par **Les Éditions Diamond**

B.P. 20142 / 67603 Sélestat Cedex
Tél. : 03 67 10 00 20 / **Fax :** 03 67 10 00 21
E-mail : cial@ed-diamond.com
lecteurs@gnulinuxmag.com
Service commercial : abo@gnulinuxmag.com
Sites : www.gnulinuxmag.com
www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Chef des rédactions : Denis Bodor
Rédacteur en chef : Tristan Colombo
Secrétaire de rédaction : Fleur Brosseau
Conception graphique : Kathrin Scali
Responsable publicité : Tél. : 03 67 10 00 27
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, Allemagne
Distribution France :
(uniquement pour les dépositaires de presse)
MLP Réassort :
Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier.
Tél. : 04 74 82 63 04
Service des ventes :
Distri-médias : Tél. : 05 34 52 34 01
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : A parution
N° ISSN : 0183-0864
Commission Paritaire : K78 976
Périodicité : Bimestrielle
Prix de vente : 12,90 Euros

PRÉFACE

Il y a quelques mois, nous proposions un guide de *Linux Pratique* sur le Raspberry Pi qui exposait les bases pour prendre en main ce mini-ordinateur. Depuis, l'été et ses nombreux apéritifs vous ont probablement permis d'expérimenter la fabrication d'un distributeur à cocktails. De même, vos nombreux invités ont pu tester vos talents culinaires mais également la précision de vos cuissons grâce à votre compte à rebours ultra design (et un peu geek) réalisé avec votre petite carte « magique ». En bref, vous avez pu tester les multiples possibilités de ce petit mais non moins efficace outil qu'est le Raspberry Pi.

Or, saviez-vous que depuis juillet 2014, une nouvelle version de la carte a été commercialisée (la version B+) avec de nombreuses améliorations matérielles ? Pour en savoir plus, rendez-vous en page 08.

Certains d'entre vous ont donc profité de l'été pour se familiariser avec cette petite carte et vont pouvoir approfondir leurs découvertes dans ce numéro. Quant aux autres, ceux qui connaissaient déjà la carte, ce numéro devrait aussi leur apporter de nouvelles connaissances.

En effet, pour tous il est temps d'aller voir un peu plus en profondeur ce que l'on peut faire avec ce Raspberry Pi conçu à l'origine dans un but éducatif et qui, pour de nombreux électroniciens, est devenu l'outil favori d'expérimentations et de réalisations de projets.

Ainsi, dans ce nouveau guide vous pourrez :

⇨ découvrir les nouveautés du Raspberry Pi B+ ;

⇨ approfondir vos connaissances sur les différentes broches d'entrées/sorties, que ce soient les ports GPIO, ou SPI ;

⇨ ajouter un écran tactile nomade à votre carte (écran Adafruit au format du Raspberry Pi) ;

⇨ utiliser le bus i2c pour communiquer avec un capteur de température externe ;

⇨ faire du « temps réel » avec votre carte en étudiant plusieurs possibilités ;

⇨ utiliser la cross-compilation ;

⇨ et bien d'autres choses encore.

Vous avez dans les mains le guide qui vous ouvrira de nouvelles perspectives de développements, et qui vous permettra d'exploiter encore au mieux toute la puissance de la petite carte à la framboise !

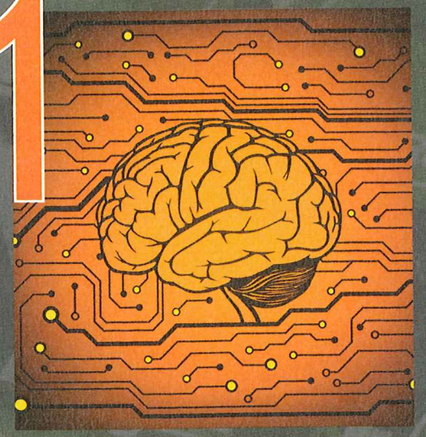
La rédaction

Sommaire

GNU/Linux Magazine
Hors-Série
N°75

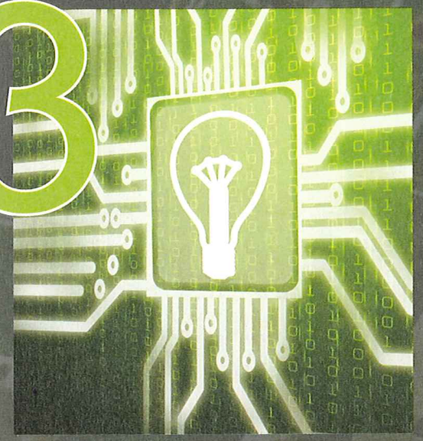
RASPBERRY PI

NIVEAU AVANCÉ

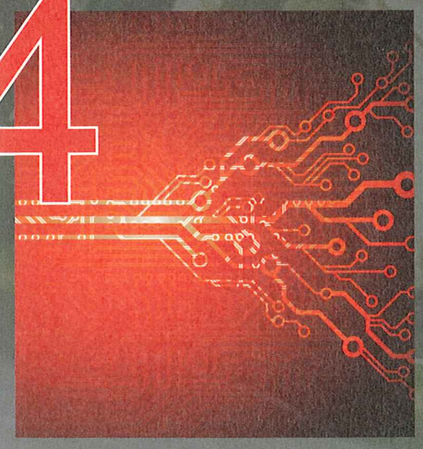


1 LA CARTE ET SES PORTS GPIO

2 L'INTERFACE SPI



3 DISTRIBUTIONS ET OS



4 APPLICATIONS

1 LA CARTE ET SES PORTS GPIO

À découvrir dans cette partie...

1

LA CARTE ET SES PORTS GPIO

SORTIE DE LA NOUVELLE RASPBERRY PI B+

Pierre Ficheux

La fondation Raspberry Pi avait jusqu'à présent produit deux versions de sa carte, la A et la B. La version B était une évolution importante par rapport à la A. La version B+ est une simple mise à jour de la B, mais mérite cependant quelques explications.

La fondation Raspberry Pi a présenté cet été le nouveau modèle de sa célèbre carte à base de processeur ARM11 [1]. L'architecture même de la carte est assez peu modifiée (même CPU, mêmes périphériques et même prix [2] !). Les principales évolutions concernent l'aspect physique et matériel de la carte [3] :

- 4 ports USB (2.0) au lieu de 2 sur la B ;
- Augmentation du nombre de GPIO disponibles tout en conservant la compatibilité avec la B (connecteur 40 broches au lieu de 26) ;
- Utilisation d'une carte microSD en remplacement de la SD ;
- Nouveau connecteur audio/vidéo composite ;
- Amélioration de l'audio ;
- Réduction de la consommation ;
- Aspect mécanique amélioré.

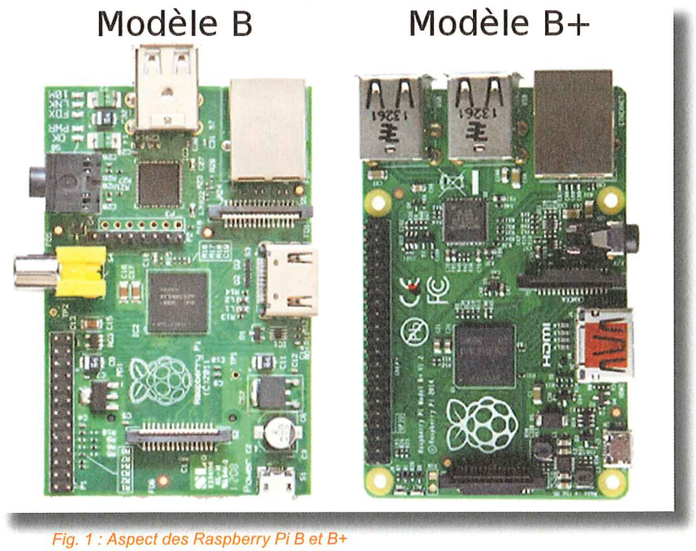


Fig. 1 : Aspect des Raspberry Pi B et B+

L'utilisation de la microSD permet d'avoir un aspect plus agréable et une plus grande sécurité de fonctionnement grâce à un meilleur contact dans le connecteur.

Au niveau logiciel, nous avons réalisé quelques tests de compatibilité et la seule modification nécessaire est la mise à jour du « firmware » [4] fourni par Broadcom. Cette modification est bien entendu prévue sous forme de mise à jour par les fournisseurs des distributions classiques comme la Raspbian. Seul (petit) bémol, il n'y a plus que deux leds disponibles (PWR et ACT), mais elles sont toujours accessibles comme des GPIO (respectivement aux numéros 35 et 47) [5]. ■

RÉFÉRENCES

[1] La B+ sur le site de la fondation : **http://www.raspberrypi.org/introducing-raspberry-pi-model-b-plus**

[2] La Raspberry Pi B+ en vente chez Farnell : **http://fr.farnell.com/raspberry-pi**

[3] Différences entre B et B+ par Adafruit : **https://learn.adafruit.com/introducing-the-raspberry-pi-model-b-plus-plus-differences-vs-model-b/overview**

[4] Firmware de le Raspberry Pi : **https://github.com/raspberrypi/firmware**

[5] Article de Christophe Blaess sur la B+ : **http://www.blaess.fr/christophe/2014/08/06/b**

1 LA CARTE ET SES PORTS GPIO

DÉCOUVREZ ET UTILISEZ LES BROCHES D'ENTRÉES-SORTIES DU RASPBERRY PI

Yann Guidon

Un des facteurs du succès du Raspberry Pi est son port d'entrées-sorties générales. Pour preuve, le nouveau modèle B+ fournit encore plus de broches GPIO ! On peut y accéder au moyen de quasiment tous les langages : nous allons voir ici comment le faire en Bash et en C, puis nous étendrons le nombre de signaux en sortie.

Le Raspberry Pi n'est pas juste un nano-ordinateur économique tournant sous Linux. Ses broches d'entrées-sorties l'ouvrent au monde des interfaces et c'est donc un microcontrôleur de luxe ! Mais ce n'est pas non plus un microcontrôleur comme sur les platines Arduino.

Tout d'abord, les caractéristiques électriques sont différentes : les tensions vont de 0 à 3,3V au lieu de 5V et on peut atteindre des vitesses bien plus élevées : les broches SPI peuvent monter à plusieurs dizaines de mégahertz ! Il faut donc soigner les circuits si on veut exploiter à fond le potentiel de la puce Broadcom, mais si vous désirez juste allumer une LED ou lire un bouton-poussoir, il n'y a pas de différence avec une autre carte.

1. LES CONNECTEURS GPIO DU RASPBERRY PI

Il existe plusieurs révisions du Raspberry Pi. Le modèle A et le modèle B utilisent le même circuit imprimé, qui a évolué en trois ans, avant la refonte majeure du modèle B+. Dans cet article, nous allons nous concentrer uniquement sur les 26 broches de la figure 1, car elles seront utilisées quel que soit le modèle.

Par la suite, nous utiliserons uniquement la numérotation *logique* des GPIO. On trouve des projets qui numérotent les signaux en fonction des broches de la puce Broadcom, ou du connecteur P1. En pratique, la numérotation logique facilite l'écriture des programmes et la partie électronique se débrouille avec un amas de fils volants pour remettre les signaux dans l'ordre.

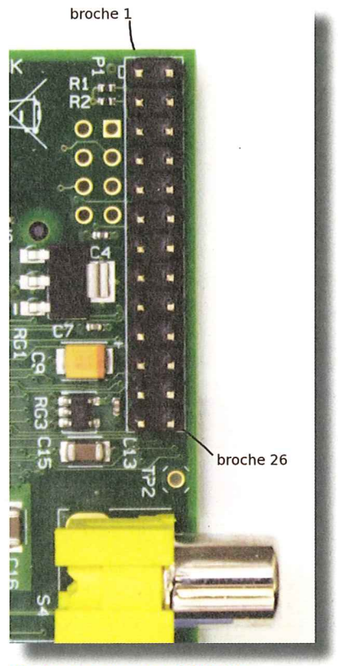


Fig. 1 : Le connecteur GPIO d'un Raspeberry Pi, révision 2013-2014

L'assignation de certaines broches a évolué : les premières révisions fournissaient les GPIO n°0, 1 et 21, qui ont été remplacées par les GPIO n°2, 3 et 27. Ces cartes de première génération sont maintenant rares et nous supposerons que vous disposez d'une révision récente.

Sur la figure 1, on aperçoit un groupe de huit pastilles à gauche : c'est le connecteur P5, destiné à être éventuellement soudé par l'utilisateur, sur l'autre face du circuit. Si vous trouvez ces pastilles sur votre carte, cela signifie aussi que vous avez une révision récente, qui dispose des GPIO n°2, 3 et 27. Le connecteur P5 fournit 4 signaux GPIO et des alimentations, mais il est légèrement décalé par rapport au connecteur principal, ce qui l'empêche d'être utilisé avec des plaques standards au pas de 2,54 mm.

Le nouveau modèle B+ a supprimé ce port, bricolable seulement par des experts, donc impopulaire. Heureusement, les concepteurs ont eu la bonne idée d'allonger le connecteur P1, qui passe à 40 broches et qui conserve un format très pratique. Les GPIO n°28 à 31 ne sont plus disponibles, car P5 a disparu, mais on gagne au change ! L'ordre des GPIO est toujours tiré par les cheveux (certainement pour des raisons de routage des pistes du circuit imprimé), mais leurs numéros sont maintenant consécutifs. Le modèle B+ permet désormais d'accéder à toutes les entrées-sorties de 2 à 27. Il n'y a plus de « trous » et l'assignation des broches est donc beaucoup plus facile. Le routage d'une carte fille reste un peu compliqué, mais le logiciel peut être moins complexe, donc plus efficace et rapide.

Dans cet article, nous nous intéresserons seulement à la fonction GPIO des broches, c'est-à-dire « *General Purpose Input-Output* », donc entrée-sortie généraliste. Cela signifie que l'utilisateur peut contrôler l'état de la broche et lire un signal connecté dessus. La plupart des broches de la puce BCM2835 fournissent aussi plusieurs *fonctions alternatives*, dont les plus intéressantes sont disponibles sur le connecteur P1. Les plus populaires sont le port série, le port SPI et le port I²C ; on trouve aussi une sortie PWM et I2S (complété par feu P5). De plus, chaque broche peut servir de source d'interruption.

Commençons par allumer une simple LED. Justement, Linux allume la LED ACT au moyen d'une broche GPIO, mais comment fait-il ?

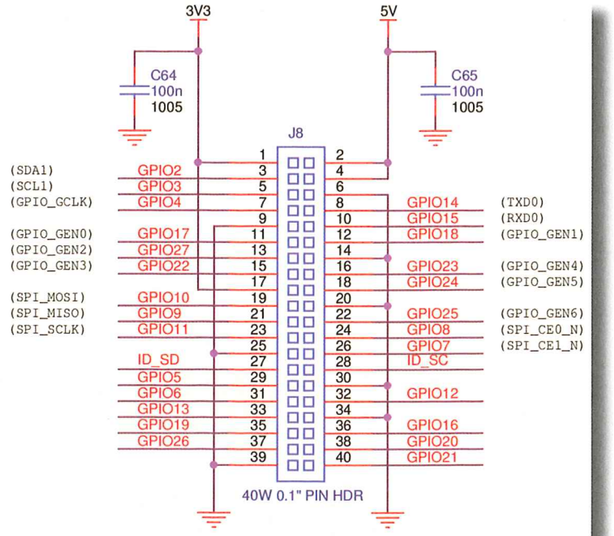


Fig. 2 : Assignation des broches du connecteur P1 du modèle B+
Source : http://www.raspberrypi.org/documentation/hardware/raspberrypi/schematics/Raspberry-Pi-B-Plus-V1.2-Schematics.pdf

2. L'INTERFACE GPIO DE LINUX

Quand il s'agit juste de bricoler un petit montage électronique, lent de surcroît, un langage interprété est le plus pratique à mettre en œuvre. Et si vous n'êtes pas encore converti au Python, voici comment lire et écrire sur les broches en ligne de commandes. C'est un peu lent, mais c'est très utile et Bash est déjà installé sur tous les systèmes GNU/Linux !

L'interpréteur de ligne de commandes Bash n'est pas réputé pour sa vitesse, d'autant plus qu'il doit passer par des appels système et des fichiers virtuels pour effectuer la moindre opération. Pourtant, cela suffit largement si vous voulez juste allumer une LED pour informer l'utilisateur d'une condition du système (réception d'un e-mail ou charge de processeur trop élevée), ou lire des interfaces (bouton-poussoir, capteur physique, ou communiquer avec un autre montage). Dans mon cas, cela me permet de manipuler les broches interactivement, en les connectant à un montage dont je veux m'assurer du fonctionnement, avant de commencer à coder en C.

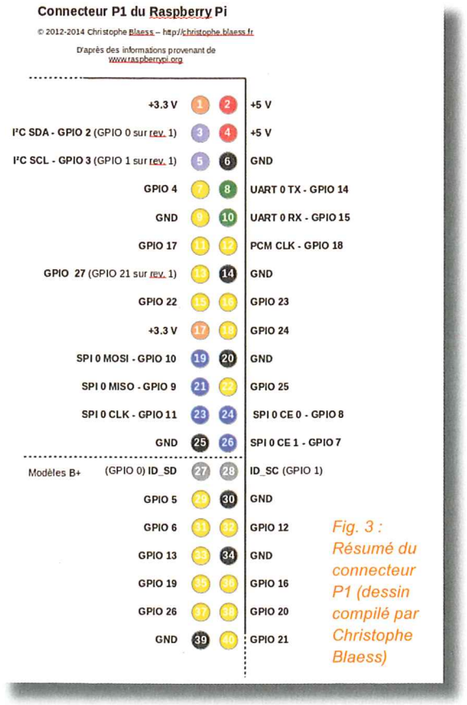


Fig. 3 : Résumé du connecteur P1 (dessin compilé par Christophe Blaess)

2.1 L'interface GPIO de Linux

La clé de cette partie, c'est l'accès aux broches GPIO au travers des fichiers virtuels de Linux. La distribution standard Raspbian fournit un module du noyau, qui ajoute un répertoire contenant des fichiers spéciaux. Ceux-ci configurent et contrôlent les broches au moyen de simples caractères, ce qui est très pratique en ligne de commandes ou dans un script.

2.1.1 Configuration

Tout d'abord, n'oublions pas que les GPIO ne sont accessibles qu'à partir du compte root. Toutes les commandes doivent être précédées par **su**. Sinon, vous pouvez activer le bit **setuid** d'un script contenant ces commandes, en plus de l'attribut d'exécution, avec **chmod**. Mais pour commencer, le plus simple reste :

Terminal
```
pi@pi:~$ sudo su
```

Ensuite, allons faire un tour du côté de **/sys/class/gpio** :

Terminal
```
root@pi:/home/pi# cd /sys/class/gpio/
root@pi:/sys/class/gpio# ls -al
total 0
drwxr-xr-x  2 root root    0 juil. 31 13:35 .
drwxr-xr-x 37 root root    0 juil. 31 13:35 ..
--w-------  1 root root 4096 juil. 31 13:35 export
lrwxrwxrwx  1 root root    0 juil. 31 13:35 gpiochip0 -> ../../devices/
virtual/gpio/gpiochip0
--w-------  1 root root 4096 juil. 31 13:35 unexport
```

Le fichier **export** (en écriture seule) permet d'ajouter des fichiers et répertoires virtuels correspondant à des broches du processeur central. Quand on écrit un nombre décimal valide dedans, un nouveau répertoire va apparaître. Par exemple, si on s'intéresse à la broche GPIO4 :

Terminal
```
root@pi:/sys/class/gpio# echo 4 > export
root@pi:/sys/class/gpio# ls -al
total 0
drwxr-xr-x  2 root root    0 juil. 31 13:35 .
drwxr-xr-x 37 root root    0 juil. 31 13:35 ..
--w-------  1 root root 4096 juil. 31 13:43 export
lrwxrwxrwx  1 root root    0 juil. 31 13:43 gpio4 -> ../../devices/virtual/gpio/gpio4
lrwxrwxrwx  1 root root    0 juil. 31 13:35 gpiochip0 -> ../../devices/virtual/gpio/
gpiochip0
--w-------  1 root root 4096 juil. 31 13:35 unexport
```

L'écriture du nombre 4 dans le fichier **unexport**, comme vous le devinez, effacera le nouveau répertoire.

Le contenu de ces répertoires est très intéressant :

Terminal
```
root@pi:/sys/class/gpio# ls -al gpio4/
total 0
drwxr-xr-x 3 root root    0 juil. 31 13:43 .
drwxr-xr-x 4 root root    0 juil. 31 13:35 ..
```

```
-rw-r--r-- 1 root root 4096 juil. 31 13:49 active_low
-rw-r--r-- 1 root root 4096 juil. 31 13:49 direction
-rw-r--r-- 1 root root 4096 juil. 31 13:49 edge
drwxr-xr-x 2 root root    0 juil. 31 13:49 power
lrwxrwxrwx 1 root root    0 juil. 31 13:43 subsystem -> ../../../../class/gpio
-rw-r--r-- 1 root root 4096 juil. 31 13:43 uevent
-rw-r--r-- 1 root root 4096 juil. 31 13:49 value
```

On voit des fichiers qui servent à la gestion des interruptions (**active_low**, **edge**), mais celui qui nous intéresse ici est **direction**. Comme son nom l'indique, ce qu'on y écrira déterminera si la broche est une entrée ou une sortie.

2.1.2 Lecture

La broche est en lecture par défaut :

Terminal
```
root@pi:/sys/class/gpio# cat gpio4/direction
in
```

On peut toujours forcer la direction, cela ne provoque pas d'erreur :

Terminal
```
root@pi:/sys/class/gpio# echo in > gpio4/direction
```

L'état de la broche est lu dans le fichier virtuel **value** :

Terminal
```
root@pi:/sys/class/gpio# cat gpio4/value
0
```

Et c'est tout. Qui a dit que ça devait être compliqué ?

2.1.3 Écriture

Corollairement, avant de changer l'état d'une broche, nous devons d'abord indiquer qu'elle doit être en sortie, avec le mot-clé **out** :

Terminal
```
root@pi:/sys/class/gpio# echo out > gpio4/direction
```

Et comme vous l'aurez peut-être deviné, la valeur de la broche est aussi contrôlée en écrivant dans le fichier **value** :

Terminal
```
root@pi:/sys/class/gpio# echo 0 > gpio4/value
root@pi:/sys/class/gpio# cat gpio4/value
0
root@pi:/sys/class/gpio# echo 1 > gpio4/value
root@pi:/sys/class/gpio# cat gpio4/value
1
```

Si vous avez connecté une LED dans le bon sens sur GPIO4 (broche 7 du connecteur P1), elle devrait s'allumer.

Ça y est, vous savez utiliser les entrées-sorties !

2.2 Des scripts

Pour réduire le nombre de caractères à écrire, donc le nombre d'erreurs, ces commandes sont souvent intégrées dans de petits scripts. Quelques vérifications sont ajoutées, mais on laisse le noyau détecter si le numéro du port est correct.

La première étape consiste à s'assurer que le répertoire de la broche existe bien. Une expression conditionnelle le détecte, et s'il n'est pas trouvé, il est créé :

Fichier
```
[ -d /sys/class/gpio/gpio$1 ] ||
  echo $1 > /sys/class/gpio/export
```

⇨ **-d** retourne une valeur *vraie* si le répertoire existe ;

⇨ **||** est l'expression OU : si la première partie est fausse, alors la deuxième partie est exécutée.

À ce point, si l'argument est valide, le répertoire de la broche doit exister. Si l'argument est absent, non numérique ou ne correspond pas à une broche accessible, le noyau ne va pas le créer. C'est ce que nous vérifions dans la deuxième étape : peut-on écrire dans le fichier **direction** ?

On vérifie cela avec **-w**. Ce test assure non seulement que le répertoire existe, mais aussi que l'utilisateur a le droit d'y accéder (car seul root le peut). En cas de succès, le bloc de code suivant (délimité par des accolades) est exécuté après l'expression ET, écrite avec **&&**.

Le premier script **GPIO_in.sh** lit l'état de la broche et reprend tous ces éléments :

Fichier
```
#!/bin/bash
[ -d /sys/class/gpio/gpio$1 ] ||
  echo $1 > /sys/class/gpio/export
[ -w /sys/class/gpio/gpio$1/direction ] && {
  echo in > /sys/class/gpio/gpio$1/direction
  echo -n "GPIO$1 ="; cat /sys/class/gpio/gpio$1/value
}
```

GPIO_on.sh, comme son nom l'indique, met la broche en argument à 1. Il commence de la même manière, mais change l'entrée en sortie avec le mot-clé **out** :

Fichier
```
#!/bin/bash
[ -d /sys/class/gpio/gpio$1 ] ||
  echo $1 > /sys/class/gpio/export
[ -w /sys/class/gpio/gpio$1/direction ] && {
  echo out > /sys/class/gpio/gpio$1/direction
  echo 1 > /sys/class/gpio/gpio$1/value
  echo "GPIO$1 à 1"
}
```

GPIO_off.sh met la broche à zéro et ne diffère que par la valeur envoyée sur le port :

Fichier
```
#!/bin/bash
[ -d /sys/class/gpio/gpio$1 ] ||
  echo $1 > /sys/class/gpio/export
[ -w /sys/class/gpio/gpio$1/direction ] && {
  echo out > /sys/class/gpio/gpio$1/direction
  echo 0 > /sys/class/gpio/gpio$1/value
  echo "GPIO$1 à 0"
}
```

N'oubliez pas de rendre ces fichiers exécutables, et même par les utilisateurs normaux (avec le bit **setuid**) :

Terminal
```
root@pi~# chmod +xs GPIO_*
```

Grâce à ces quelques petits scripts, les GPIO ne devraient plus vous faire peur. Ce type d'accès est d'ailleurs proposé sur d'autres plateformes que le Pi, car c'est une fonctionnalité standard du noyau Linux [1].

2.3 Une petite application simple

Puisque le module GPIO du noyau nous abstrait des détails de bas niveau, la complexité de codage dépend du langage choisi et Bash n'est pas un exemple de clarté. Pourtant, il permet de réaliser facilement beaucoup de choses, sans rien installer d'autre, et c'est encore plus puissant quand on le combine avec d'autres logiciels en ligne de commandes. Il faut juste ne pas avoir peur de lire la page du manuel.

Dans cet exemple, nous allons utiliser **sox**, le « couteau suisse de la manipulation sonore », pour jouer un son. Notre script lira un fichier tant qu'une broche sera à un certain état, ce qui est la base de nombreux montages comme des installations interactives. La latence est trop grande pour une utilisation musicale, mais on peut déjà beaucoup s'amuser.

Commençons par installer **sox** :

Terminal
```
root@pi# apt-get install sox
```

sox utilise l'interface OSS (*Open Sound System*) qui n'est pas installée par défaut. Ce détail est réglé avec la commande suivante :

Terminal
```
root@pi# modprobe snd-pcm-oss
```

Ensuite, vous devez sélectionner le périphérique qui restituera le son. Pour la prise casque (Jack 3,5 mm), la commande est la suivante :

Terminal
```
root@pi# sudo amixer cset numid=3 1
```

Le son peut aussi être envoyé sur la sortie HDMI, si vous disposez d'un convertisseur adapté ou d'une télévision :

Terminal
```
root@pi# sudo amixer cset numid=3 2
```

Il est aussi possible d'utiliser une carte son connectée à l'un des ports USB :

Terminal
```
root@pi# sudo amixer cset numid=3 3
```

Maintenant, vous pouvez lire un fichier, comme les sons fournis par ALSA pour tester les haut-parleurs :

Terminal
```
root@pi# play /usr/share/sounds/alsa/Front_Center.wav
```


Il ne reste plus qu'à attendre qu'une des broches GPIO soit à l'état désiré. Une boucle en Bash fait l'affaire :

Fichier
```
#!/bin/bash

[ -d /sys/class/gpio/gpio$1 ] ||
  echo $1 > /sys/class/gpio/export
echo in > /sys/class/gpio/gpio$1/direction

# boucle infinie
while true
do
  # attend que la broche $1 passe à 1 :
  until /bin/grep 1 /sys/class/gpio/gpio$1/value >> /dev/null
  do
    /bin/sleep .1 # libère le CPU
  done

  /usr/local/bin/play /usr/share/sounds/alsa/Front_Center.wav
done
```

Dans cette version, le fichier est lu tant que la broche reste à 1. Mais aussi, la lecture continue lorsque la broche retourne à 0 avant la fin du fichier. C'est un fonctionnement de type *monostable non réarmable* : la lecture ne repart pas du début lorsqu'on rappuie en plein milieu.

La version suivante lance **play** en tâche de fond au moyen du caractère **&** : on peut ensuite l'interrompre avec la commande **killall** pour arrêter la lecture lorsque la broche change. On détecte cela en répliquant la boucle d'attente, mais avec une condition inverse :

Fichier
```
#!/bin/bash

echo $1 > /sys/class/gpio/export
echo in > /sys/class/gpio/gpio$1/direction

# boucle infinie
while true
do
  # attend que la broche $1 passe à 1 :
  until /bin/grep 1 /sys/class/gpio/gpio$1/value >> /dev/null
  do
    /bin/sleep .1 # libère le CPU
  done

  # lance la lecture en tâche de fond
  /usr/local/bin/play /usr/share/sounds/alsa/Front_Center.wav &

  # attend que la broche $1 retourne à 0 :
  until /bin/grep 0 /sys/class/gpio/gpio$1/value >> /dev/null
  do
    /bin/sleep .1
  done

  # interrompt la lecture (si elle n'est pas terminée)
  /usr/bin/killall play
done
```

La lecture n'est pas interrompue immédiatement pour deux raisons. D'une part, l'attente d'un dixième de seconde ajoute une certaine incertitude (Christophe Blaess présente plus loin une méthode plus efficace qui exploite les interruptions). D'autre part, les tampons de

données sonores mettent approximativement autant de temps à se vider. Heureusement, il existe de nombreuses situations où cela ne dérange pas, autrement il faudrait faire appel à des systèmes bien plus complexes.

La situation se complique si vous voulez lire plusieurs sons, car on ne peut démarrer la lecture suivante que si la précédente s'est bien terminée. En d'autres termes, il faut savoir si `play` s'est bien arrêté, ce qui est un peu plus compliqué que s'il était lancé en tâche d'avant-plan (sans `&`). Bash nous fournit une solution avec la commande `jobs` qui liste les programmes lancés en arrière-plan. Cette commande retourne une chaîne vide si le programme s'est terminé, que ce soit tout seul ou bien forcé par `kill`. On peut donc scruter la broche GPIO simultanément et tester des conditions plus complexes.

Il serait aussi bienvenu de ne plus faire appel à `grep` pour comparer la valeur de la broche. Encore une fois, Bash le permet avec une syntaxe plus obscure : `$(< fichier )` lit le fichier indiqué et met son contenu dans une variable que Bash peut comparer comme une chaîne de caractères normale, au moyen de l'opérateur `==` entouré des délimiteurs `[[` et `]]`. Cela consomme moins de ressources, puisque Bash ne lance plus de commandes externes. Tout cela est condensé dans la fonction `playwhile()` :

Fichier

```
#!/bin/bash
SON_ON=/usr/share/sounds/alsa/Front_Center.wav
SON_OFF=/usr/share/sounds/alsa/Front_Left.wav

[ -d /sys/class/gpio/gpio7 ] ||
  echo 7 > /sys/class/gpio/export
echo in > /sys/class/gpio/gpio7/direction &&

function playwhile() {
  # $1 : état à tester
  # $2 : fichier à lire

  # ne commence la lecture que si l'état est différent :
  [[ $1 == "$(< /sys/class/gpio/gpio7/value)" ]] || /usr/local/bin/play -q
$2 &

  # attend la fin de la lecture
  while [[ "$(jobs)" ]]
  do
    jobs > /dev/null # le script ne fonctionne pas sans cette ligne
redondante, je ne sais pas pourquoi.
    [[ $1 == "$(< /sys/class/gpio/gpio7/value)" ]] && /usr/bin/killall /
usr/local/bin/play >> /dev/null 2>&1
    /bin/sleep .1
  done
}

# Le corps du programme :
while true
do
  playwhile 1 $SON_OFF
  playwhile 0 $SON_ON
done
```

D'autres comportements plus complexes sont envisageables à partir de ces briques de base. Une fois que tout est mis au point, le fichier script pourra être exécuté au démarrage du système si on ajoute son chemin complet à la fin du fichier `/etc/rc.local`. Dans le script, les noms des programmes sont aussi donnés avec leur chemin absolu, car `$PATH` n'est pas encore initialisé à ce moment-là.

3. ACCÉDEZ DIRECTEMENT AUX GPIO EN C

Ces scripts sont donc relativement portables et vous pourrez réutiliser une grande partie de votre code sur d'autres types d'ordinateurs. Mais pour réaliser des fonctions plus complexes ou beaucoup plus rapides, le codage en langage C devient nécessaire, bien que le code soit moins portable.

Je vous propose maintenant une petite bibliothèque de code en C pur, destinée à contrôler les broches de votre carte à la framboise. C'est une évolution du code présenté en 2013 dans *Open Silicium* n°6 [2], je vous invite à consulter cet article pour y trouver les détails techniques. Depuis, le code a été remanié, corrigé et amélioré : l'expérience a permis de trouver un bug indigne, d'ajouter des fonctionnalités et d'augmenter le confort d'utilisation. Ce fichier source est à la base de plusieurs bibliothèques et projets, faisons donc dès maintenant les présentations.

Le code est fourni ici intégralement, dans un souci d'exhaustivité et parce qu'il n'est pas très long, mais vous pouvez aussi le télécharger sur le dépôt GitHub du magazine. Les plus curieux compareront avec la version originale pour découvrir quel bug stupide a bien pu être oublié précédemment...

3.1 Le principe

La puce BCM2835 au cœur du Raspberry Pi accède à ses broches au travers de circuits situés dans un espace mémoire particulier, à l'adresse physique `GPIO_BASE`. Seul root peut y lire et écrire, après avoir projeté cet espace dans sa mémoire virtuelle au moyen de la fonction `mmap()`. Ceci est effectué par la fonction `PI_IOmmap()`.

La zone d'entrées-sorties contient de nombreux registres qui configurent chaque broche. Par exemple, il y a jusqu'à huit fonctions possibles par broche (dont `BCM_GPIO_IN` et `BCM_GPIO_OUT`), ce qui est encodé avec 3 bits. La fonction `PI_GPIO_config()` calcule l'adresse des bits correspondant au port à configurer et met à jour le registre de fonctions.

Enfin, le plus important est l'accès aux broches. Trois macros sont proposées : `GPIO_LEV_N(N)` retourne la valeur de la broche, alors que `GPIO_SET_N(N)` et `GPIO_CLR_N(N)` la modifient. Pour des raisons d'atomicité des opérations, il n'est pas possible d'indiquer directement la valeur d'une broche : le registre pourrait être en cours de modification par un autre programme, ce qui entraînerait l'instabilité du système (puisque certaines broches GPIO contrôlent directement des périphériques tels que la carte SD).

3.2 Les améliorations

Voici une liste d'améliorations :

- La fonction d'initialisation devait être explicitement appelée, mais elle est maintenant intégrée dans la fonction de configuration d'une broche. Chaque appel à `PI_GPIO_config()` teste si la zone mémoire des GPIO est bien projetée dans l'espace utilisateur. Ainsi, il n'y a plus de risque d'oublier un appel de fonction en début de programme, qui est aussi plus court d'une ligne.
- La configuration des ports d'entrées-sorties est aussi utilisée par ma bibliothèque SPI. D'ailleurs, la fonction `PI_IOmmap()` peut aussi être appelée par d'autres morceaux de code. Il n'est même plus nécessaire de se souvenir quel code a initialisé quoi en premier.
- L'appel à `mmap()` posait quelques soucis, car la valeur de retour n'était pas correctement testée. Une mise à jour du système d'exploitation en mai 2013 a révélé des problèmes dans le code de Gert & Dom, qui sont maintenant résolus en utilisant le symbole `MAP_FAILED`.

⇨ Les messages d'erreurs ont été adaptés et on peut fournir notre propre routine en jouant avec les **#define**. Habituellement, j'utilise deux fonctions : **erreur()** utilise **perror()** si un appel système échoue ; **err()** utilise juste **printf()** pour les erreurs internes.

3.3 Le parachute

La dernière nouveauté est l'ajout de la fonction **GPIO_parachute()** qui remet automatiquement en entrées les broches configurées par le programme. Cela réduit le code que le développeur doit écrire et augmente aussi la sécurité du système : si l'application plante, l'électronique ne reste pas figée dans un état potentiellement dommageable (par exemple, si une impulsion ne doit pas durer trop longtemps pour ne pas griller un composant).

Pour réaliser cela, le code utilise la fonction standard **atexit()**, qui ajoute notre fonction de nettoyage **GPIO_parachute()** à une liste. Cette liste sera balayée dans le sens inverse lorsque le programme se terminera. Le seul cas où cela ne se produira pas est si le programme est interrompu avec le signal **SIGKILL** (avec la commande **kill -9**), car il ne peut être intercepté.

PI_GPIO_config() mémorise dans la variable **GPIO_used** toutes les broches qu'on lui demande de configurer. Au premier appel, il installe aussi le parachute : **GPIO_parachute()** consultera la variable en fin de programme pour désactiver les broches qui ne sont pas déjà en entrées. Le parachute utilise aussi **PI_GPIO_config()**, ce qui crée une situation de poule et d'œuf, résolue par une pré-déclaration de la fonction.

Afin de s'assurer que le parachute sera déployé dans tous les cas possibles, on connecte la plupart des signaux à **exit()**, au moyen de la fonction **signal()**. L'utilisateur peut toujours ajouter son propre gestionnaire de signaux, tant qu'il se termine par un appel à **exit()**. Mais le plus simple est d'enregistrer une ou plusieurs autres fonctions avec **atexit()**, car cela préservera aussi l'ordre d'appel.

Cette fonctionnalité peut être désactivée en définissant le symbole **GPIO_NO_ATEXIT**.

3.4 Le code source

Le code suivant est le code du projet. Il a été abondamment commenté pour en comprendre le fonctionnement :

Fichier

```
01: /*
02:  PI_GPIO.c  (c) Yann Guidon 20130204
03:  Accès aux E/S de la carte Raspberry Pi
04:  Dérivé du code de Dom & Gert @ http://elinux.org/RPi_Low-level_
peripherals v. 20130101
05:
06:   20140907 : tout sur stderr
07: */
08:
09: #ifndef PI_GPIO
10: #define PI_GPIO
11:
12: #include <stdio.h>
13: #include <sys/mman.h>
14: #include <fcntl.h>
15: #include <stdlib.h>
16:
```


Fichier

```
17: int mmap_fd=0;
18:
19: #define GPIO_BASE  (0x20200000)
20: #define BLOCK_SIZE (4096) // correspond à la taille d'une page de MMU (unité de
gestion mémoire)
21:
22: // si la fonction erreur() n'est pas déjà fournie
23: #ifdef PI_GPIO_ERR
24: #include <errno.h>
25:
26: /* sortie avec un message d'erreur contextualisé */
27: void erreur(char *msg) {
28:   perror(msg);
29:   exit(EXIT_FAILURE);
30: }
31:
32: // 2ème message d'erreur, sans perror
33: void err(char *msg) {
34:   fputs(msg, stderr);
35:   fputc('\n', stderr);
36:   exit(EXIT_FAILURE);
37: }
38: #endif
39:
40: unsigned * PI_IOmmap(off_t where) {
41:    void* map;
42:
43:    // ne réouvre pas /dev/mem si on l'a déjà ouvert
44:    if (mmap_fd <= 0) {
45:      // ouvre /dev/mem
46:      if ((mmap_fd = open("/dev/mem", O_RDWR|O_SYNC) ) < 0)
47:        erreur("Echec à l'ouverture de /dev/mem");
48:    }
49:
50:    // projette les registres GPIO dans la mémoire de notre programme
51:    map = mmap(
52:       NULL,             // projette où ça arrange le kernel
53:       BLOCK_SIZE,       // les registres de contrôle tiennent dans une seule page
54:       PROT_READ|PROT_WRITE, // on veut lire et écrire
55:       MAP_SHARED,       // partagé avec d'autres processus
56:       mmap_fd,          // la mémoire
57:       where             // adresse de la zone à accéder
58:    );
59:
60:    if (map == MAP_FAILED)
61:      erreur("Echec de mmap()");
62:
63:    return (unsigned*)map;
64: }
65:
66:
67: volatile unsigned *PI_gpio=NULL; // DOIT être volatile pour éviter que les
optimisations du compilateur n'altèrent notre code
68: // Pour mettre des broches à 1 ou à 0 :
69: #define RPI_GPSET0   (7)
70: #define RPI_GPCLR0   (10)
71: #define RPI_GPLEV0   (13)
72: #define GPIO_SET    *(PI_gpio+RPI_GPSET0)
73: #define GPIO_CLR    *(PI_gpio+RPI_GPCLR0)
74: #define GPIO_LEV    *(PI_gpio+RPI_GPLEV0)
75:
76: #define GPIO_SET_N(N) GPIO_SET = (1 << N)
77: #define GPIO_CLR_N(N) GPIO_CLR = (1 << N)
78: #define GPIO_LEV_N(N) (((GPIO_LEV) >> N) &1)
79:
```

Fichier

```
80: #define BCM_GPIO_IN   (0)  // mode entrée
81: #define BCM_GPIO_OUT  (1)  // mode sortie
82: #define BCM_GPIO_ALT0 (4)  // fonctions alternatives
83: #define BCM_GPIO_ALT1 (5)
84: #define BCM_GPIO_ALT2 (6)
85: #define BCM_GPIO_ALT3 (7)
86: #define BCM_GPIO_ALT4 (3)
87: #define BCM_GPIO_ALT5 (2)
88: #define BCM_GPIO_ALT3 (7)
89:
90:
91: #ifndef GPIO_NO_ATEXIT
92: unsigned long long int GPIO_used=0; // Liste des broches à désactiver
93: void GPIO_parachute(); // pré-déclaration
94:
95: #include <signal.h>
96: #endif
97:
98: // à appeler obligatoirement avant d'accéder aux broches !
99: void PI_GPIO_config(int port, int mode) {
100:   int registre, offset, temp;
101:
102:   // le premier appel lance le mmap (pour pas oublier)
103:   if (PI_gpio == NULL)
104:     PI_gpio = PI_IOmmap(GPIO_BASE); // adresse des ports d'entrée-sortie
105:
106:   if ((port >= 0) && (port < 32) // Ce code fonctionne jusqu'à 54 GPIO,
107:       // mais les macros ne supportent que 32 GPIO
108:    && (mode >= 0) && (mode < 8 )) {
109:     // Calcul du numéro de registre :
110:     registre = port/10;
111:
112:     // lecture dudit registre :
113:     temp = *(PI_gpio+registre);
114:
115:     // calcul de l'offset :
116:     offset = port - (registre*10); // modulo déguisé
117:     offset *= 3; // 3 bits par port
118:
119:     // Effacer les bits précédents :
120:     temp &= ~(7 << offset);
121:     // ajouter le mode désiré :
122:     temp |= mode << offset;
123:
124:     // réécriture du résultat :
125:     *(PI_gpio+registre) = temp;
126:
127: #ifndef GPIO_NO_ATEXIT
128:     if (mode != 0) {
129:       // enregistrement de la broche pour la désactivation
130:       if ( GPIO_used == 0 ) {
131:         atexit(GPIO_parachute);
132:         // En cas de signal, termine le programme
133:         // et indirectement appelle GPIO_parachute :
134:         signal(SIGHUP,  exit);
135:         signal(SIGINT,  exit);
136:         signal(SIGQUIT, exit);
137:         signal(SIGSTOP, exit);
138:         signal(SIGTERM, exit);
139:         signal(SIGABRT, exit);
140:         signal(SIGKILL, exit);
141:       }
142:       GPIO_used |= 1UL << port;
143:     }
144: #endif
145:   }
146:   else {
```


Fichier

```
147:     fprintf(stderr,"Mauvais numéro de port (%d) ou de mode (%d)\n",
port, mode);
148:     exit(EXIT_FAILURE);
149:   }
150: }
151:
152: #ifndef GPIO_NO_ATEXIT
153:
154: void GPIO_parachute() {
155:   unsigned int GPIO_cache = GPIO_used;
156:   unsigned int mask = 1;
157:   int i=0;
158:
159:   fputs("\nRemet les broches", stderr);
160:   while (GPIO_cache) {
161:     if (GPIO_cache & mask) {
162:       PI_GPIO_config(i, BCM_GPIO_IN);
163:       GPIO_cache &= ~mask;
164:       fprintf(stderr, " %d", i);
165:     }
166:     mask += mask; // décalage à gauche
167:     i++;
168:   }
169:
170:   fputs(" en entrée\n", stderr);
171: }
172:
173: #endif
174: #endif
```

3.5 Un petit exemple

La bibliothèque facilite beaucoup l'écriture de code, même en C. Pour preuve, voici le code source d'un petit programme qui recopie l'état d'une broche dans une autre. Il n'y a presque rien d'autre à gérer et puisque plusieurs fichiers standards sont déjà inclus, pas besoin de les déclarer !

Fichier

```
01: /*  test_in_out.c
02:     version sam. juil. 26 10:05:03 CEST 2014
03:
04: copie l'état d'une broche d'entrée vers une broche de sortie
05:
06: gcc -Wall -o test test_in_out.c
07: */
08:
09: #define PI_GPIO_ERR
10: #include "PI_GPIO.c"
11: #define PI_IN  (4)
12: #define PI_OUT (25)
13:
14: int main(int argc, char *argv[]) {
15:   PI_GPIO_config(PI_IN,  BCM_GPIO_IN);
16:   PI_GPIO_config(PI_OUT, BCM_GPIO_OUT);
17:
18:   while (1) {
19:     printf("GPIO%d=%d\n", PI_IN, GPIO_LEV_N(PI_IN));
20:     if (GPIO_LEV_N(PI_IN))
21:       GPIO_SET_N(PI_OUT);
22:     else
23:       GPIO_CLR_N(PI_OUT);
24:   }
25: }
```

Évidemment, pour que ce programme fonctionne, vous devez le lancer avec l'utilisateur root :

Terminal

```
pi@pi:~$ gcc -Wall -o test test_in_out.c
pi@pi:~$ sudo ./test
GPIO4=0
GPIO4=0
GPIO4=0
...
```

4. EXTENSION FIABLE DU PORT GPIO

Voilà, nous pouvons interfacer notre microcontrôleur de luxe ! Cependant, les débutants peuvent être découragés par quelques détails qui font la différence entre un montage qui fait à peu près ce qu'on lui demande, et un montage fiable. Une différence qui ne manque pas de se révéler lorsqu'on passe d'un montage bricolé sur le coin d'une table à une installation définitive sur le terrain. Nous allons maintenant voir comment ajouter des signaux au connecteur pour contrôler des composants électroniques externes sans aucun risque.

Un des luxes des PC de l'an 2000, c'était le port d'imprimante parallèle, que l'on pouvait utiliser pour un nombre incroyable d'autres applications. Trois broches en sortie, cinq en entrée et huit bidirectionnelles, permettaient de connecter des appareils parfois farfelus (une interface Ethernet ?), expérimentaux (comme les innombrables montages publiés par *Électronique Pratique*) ou grand public (scanner, lecteur ZIP). Un exemple d'utilisation sous Linux a été décrit dans *GNU/Linux Magazine France* [3].

Le PC a malheureusement abandonné cette interface, au profit (douteux) du port USB, qui apporte son lot d'avantages... et d'inconvénients (en particulier la latence, ainsi que la portabilité des logiciels et des pilotes). Le Raspberry Pi arrive sur les talons d'Arduino et une nouvelle ère d'interfaçage commence, nous permettant de redécouvrir les joies du *bitbanging*, débarrassé des complexités accumulées par les PC modernes.

Mais rappelons-nous que les PC utilisent le port parallèle pour à peu près tout et n'importe quoi. En particulier, certains modèles y envoient des codes lors du démarrage, afin de diagnostiquer les problèmes de BIOS. Ce qui signifie qu'un montage électronique recevra des informations incohérentes ou indésirables si le montage est allumé en même temps que l'ordinateur.

Une parade consiste à doter le périphérique d'une certaine « intelligence », pour par exemple reconnaître des codes d'identification et d'activation, ce qui complexifie évidemment l'électronique, alourdit le développement et augmente le coût.

Qu'en est-il du Raspberry Pi ? Bien que le SoC BCM2835 soit assez bien documenté, ce n'est pas toujours clair ou utile, car le firmware propriétaire ainsi que le kernel et la configuration de l'utilisateur peuvent tout perturber. Les nouveaux « HATs » introduits avec le modèle B+ [4] utilisent une EEPROM I²C pour la configuration, mais ce nouveau standard n'est ni obligatoire, ni disponible sur les révisions antérieures.

4.1 État à l'allumage

Comment s'assurer de l'état des broches dès la mise sous tension, et qu'elles ne changeront pas avant qu'on le décide ? Pour en avoir le cœur net, j'ai observé les broches des ports d'entrées-sorties d'un Modèle B, révision 2, à l'aide d'un oscilloscope, ce qui donne le tableau suivant.

Port P1 :

Broche	État à l'allumage	Fonction	Remarques
1		Alim 3,3V	
3	1	GPIO2	I2C SDA : Pull-up 1.8K
5	1	GPIO3	I2C SCL : Pull-up 1.8K
7	0	GPIO4	
9		GND	
11	0	GPIO17	
13	0	GPIO27	impulsions parfois observées à l'allumage (SPI/série ?)
15	0	GPIO22	
17		Alim 3,3V	
19	0	GPIO10	SPI_MOSI
21	0	GPIO9	SPI_MISO
23	0	GPIO11	SPI_CLK
25		GND	
-----	----	----------	
2		Alim 5V	
4		Alim 5V	
6		GND	
8	1	GPIO14	TxD => console, impulsions (voir `/boot/cmdline.txt`)
10	1	GPIO15	RxD (si c'est la réception, pourquoi est-il à 1 ?)
12	0	GPIO18	
14		GND	
16	0	GPIO23	
18	0	GPIO24	
20		GND	
22	0	GPIO25	
24	0	GPIO8	SPI_CE0
26	0	GPIO9	SPI_CE1 (observé à 1 une fois ?)

Port P5 :

Broche	État	Fonction
1		Alim 5V
3	0	GPIO28
5	0	GPIO30
7		GND
----------	----------	
2		Alim 3,3V
4	0	GPIO29
6	0	GPIO31
8		GND

Je ne dispose pas encore d'un modèle B+, sur lequel il faudra refaire les tests.

Évidemment, cela dépend aussi des logiciels installés, ainsi que des modules du noyau, statiques ou dynamiques. Par exemple, le port série a été désactivé sur mon système (en enlevant toute mention de `/dev/ttyAMA0` dans `/boot/cmdline.txt` et `/etc/inittab`) et malgré cela, quelques impulsions sont encore observées à l'allumage sur GPIO14.

1

Attention

Parfois, le 74HCT273 se déclenche et change de sortie lorsque des parasites sont injectés dans l'alimentation. Par exemple lors du branchement d'un autre appareil, ou doté d'une alimentation différente. Filtrez bien les alimentations !

4.2 Le registre 74HCT273

Les changements et signaux transitoires sont très embêtants si vous voulez contrôler des périphériques comme des moteurs ou des lumières, ou tout autre actionneur qui n'aime pas que ses broches soient chatouillées. Particulièrement si des combinaisons de signaux sont interdites et pourraient endommager les circuits ou les données.

Une approche classique consiste à utiliser des interfaces de type série. Un microcontrôleur connecté sur `ttyAMA0` ferait l'affaire, mais ce serait superflu. On dispose aussi du port SPI (interfaçable avec un registre à décalage 74HCT595 par exemple) ou I²C. Par contre, ce n'est pas idéal pour piloter des circuits à relativement haute vitesse, avec une latence inférieure à une microseconde.

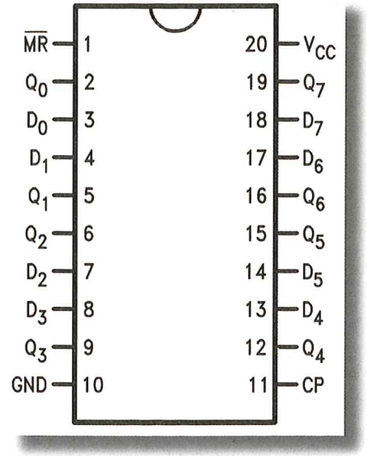


Fig. 4 : Brochage du 74HCT273. Attention à l'ordre des broches, qui diffère de la plupart des circuits similaires !

La solution présentée ici utilise un autre circuit ; le 74HCT273 est un registre à 8 bits qui mémorise l'entrée lors d'un front montant sur la broche d'horloge et surtout, qui dispose d'une entrée de remise à zéro. Je choisis ce circuit au lieu d'un 373/573/574 lorsque l'état interne *doit* être contrôlé par un signal externe. Par exemple, les 573 disposent d'une sortie à 3 états, donc des résistances de rappel au 0V peuvent amener le signal à 0, mais c'est moins fiable et il faut plus de soudures. De plus, cela ne fait que déplacer le problème : la broche /OE, qui contrôle l'état de la sortie, doit elle-même être contrôlée par un autre circuit devant être remis à zéro...

Grâce à la broche /RESET du 273, nous pouvons être sûrs qu'un actionneur ne va pas se mettre en route si la carte plante, se rallume ou subit une interruption accidentelle d'alimentation. L'inconvénient est que si vous avez besoin de signaux initialisés à l'état haut, vous devrez ajouter un inverseur. Un simple transistor peut suffire, mais une vraie porte logique est nécessaire pour les signaux à haute vitesse.

La technologie HCT est similaire à HC (faible consommation, très peu de courant sur les broches d'entrées, fonctionne sous 5V) ; la différence est que ses entrées sont compatibles avec les circuits 3,3V comme le Raspberry Pi.

Si je devais parler des inconvénients de ce circuit, je dirais juste que son brochage est inhabituel. Vérifiez bien la fonction de chaque broche, car l'organisation diffère d'autres circuits plus répandus comme le 245, 373, 573 ou 574. Heureusement, mon Pi a survécu à quelques erreurs de manipulation, où des broches en sorties se sont retrouvées connectées entre elles... Cela n'arrive pas qu'aux autres !

4.3 Soignez votre circuit de remise à zéro !

Évidemment, cette promesse de sécurité faite au début de cette section ne tient que si le circuit est correctement remis à zéro. Or, même éteinte, la carte conserve environ 1,4V sur le rail 5V et 0,5V sur le rail 3,3V. Cela se produit sans qu'on le voie, lorsqu'un écran HDMI est connecté et allumé, ce qui injecte des courants parasites...

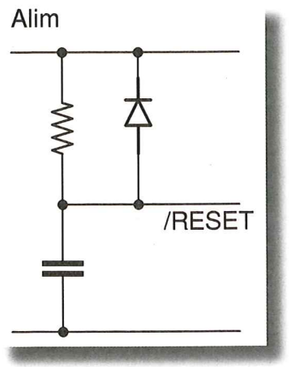


Fig. 5 : Un circuit de remise à zéro avec une diode, un condensateur et une résistance. Économique, mais pas très fiable.

Un circuit de reset « analogique » risque de conserver plus de 1V à ses bornes et mal effectuer le reset, surtout si une perte d'alimentation est courte (voir figure 5).

Le principe est simple : à la mise sous tension, le condensateur est déchargé et il se charge au travers de la résistance. Les valeurs sont choisies afin que la tension de déclenchement soit atteinte en un dixième de seconde environ. Par exemple, une résistance d'1MΩ et un condensateur de 100nF font l'affaire.

Lorsque l'alimentation est coupée, le condensateur se décharge au travers de la diode, mais cela laisse environ 0,7V à ses bornes, avant que la résistance n'ait d'effet. Et cela suppose que l'alimentation redescende effectivement à 0V, suffisamment longtemps. D'autre part, la lente montée de la tension n'est pas toujours compatible avec les broches très sensibles des circuits intégrés, ce qui risque de provoquer plusieurs impulsions.

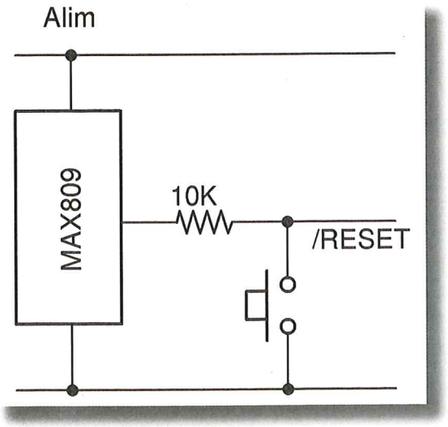


Fig. 6 : Un circuit de remise à zéro avec un circuit spécialisé, assisté d'un bouton-poussoir.

Des circuits intégrés spéciaux sont prévus pour initialiser correctement un système numérique. Différentes versions existent, avec temporisation, drain ouvert, des tensions variées, un choix de niveaux actifs, fournis par de nombreux fabricants : MAX809, LM809, TCM809, TPS3809...

On peut facilement combiner cela avec un reset manuel. Comme le montre la figure 6, un bouton-poussoir ou tout autre capteur peut être mis en parallèle avec le circuit de remise à zéro, si la sortie de celui-ci est protégée par une résistance. Cela fournit une fonction d'arrêt d'urgence pour un montage sensible. Par exemple, une forte impulsion sur l'alimentation du Pi peut le forcer à rebooter sans exécuter les fonctions de parachute. *Ça sent le vécu...*

Ce circuit commence maintenant à ressembler à un système « sérieux ».

4.4 Exemple d'application

Nous allons illustrer le principe du registre externe avec un petit montage qui fournit 8 bits (ou plus). Ce registre peut contrôler des circuits sensibles ou bien en activer d'autres (au moyen d'une entrée d'inhibition), ce qui compense un peu le faible nombre de bits disponibles.

Parmi les inconvénients du Raspberry Pi *original*, on trouve l'ordre des broches GPIO : elles sont *un peu mélangées* et surtout, il n'y a pas beaucoup de numéros consécutifs. Pour la révision 2 du modèle B (la version actuellement la plus répandue), on a juste 4 broches successives (22, 23, 24, 25), ce qui ne permet pas d'exploiter confortablement les 8 bits du registre 273.

Si vous avez vraiment besoin de 8 bits, vous pouvez compléter le bus de données avec les 4 bits consécutifs du connecteur P5, avec deux gros bémols :

- Les broches sont décalées par rapport au connecteur principal P1, donc l'accès avec un connecteur standard est difficile (il faut souder des fils ou raboter un connecteur) ;
- Les quatre broches supplémentaires (28, 29, 30, 31) ont des numéros GPIO consécutifs, mais séparés des autres (22, 23, 24, 25) ce qui complique le code.

Le modèle B+, qui remplace le modèle B depuis juillet 2014, fournit plus de broches et « bouche des trous » dans la liste des numéros logiques, ce qui permet d'accéder à la séquence de GPIO consécutives de 2 à 27. Évidemment, selon les autres types de périphériques que vous utilisez (I²C, SPI, série...), vous ne pourrez en utiliser qu'une partie, mais l'amélioration est significative et le nouveau modèle mérite d'être adopté.

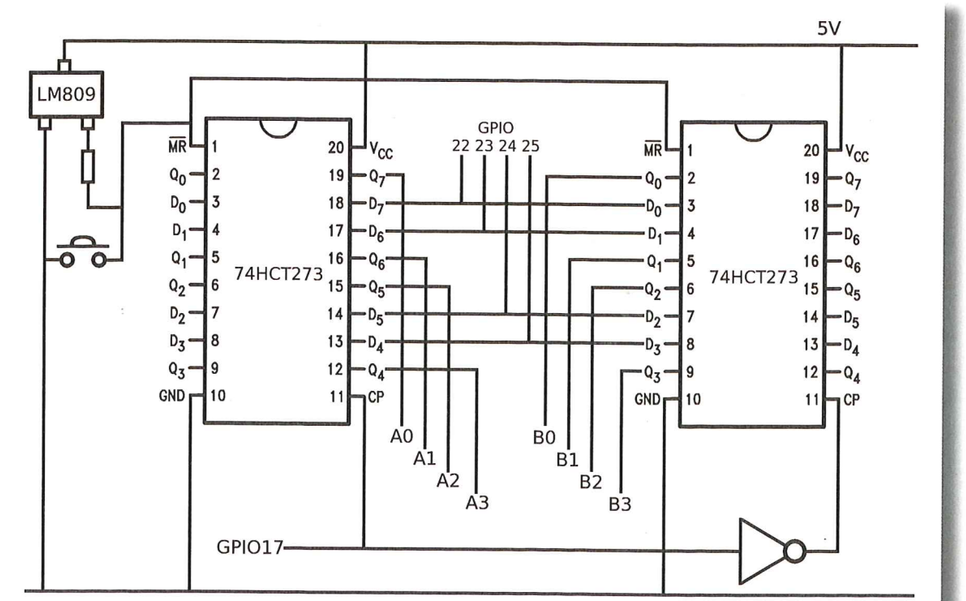


Fig. 7 : Une paire de registres 273, échantillonnés sur un front différent d'un signal de contrôle, double le nombre de sorties et garantit un allumage propre du circuit.

Autrement, si le connecteur P5 est inaccessible, il reste possible d'obtenir quand même 8 bits au moyen d'une petite astuce, même si la moitié des broches du 273 est inutilisée. En effet, l'horloge est sensible au front montant uniquement, et nous devrons appliquer une impulsion, donc deux fronts, sur cette broche. Le deuxième front peut adresser un autre 273 si on ajoute un petit circuit inverseur (voir figure 7).

Ainsi, avec 5 bits de sortie, nous en obtenons 8. Seul le bit d'horloge doit nécessairement être *propre* au démarrage ; les broches de données peuvent être à n'importe quel état puisque le 273 sera remis à zéro.

Pour l'inverseur, j'ai utilisé un 74LVC1G14, mais beaucoup d'autres circuits peuvent faire l'affaire, en fonction de ce dont vous disposez. Un simple 74HCT00 (quadruple NAND) fonctionne aussi, si l'une des broches d'entrée est connectée à l'alimentation positive. Bref, débrouillez-vous et faites attention aux tensions !

4.5 Plus d'extensions

Ce circuit fonctionne sans souci avec d'autres 273 qui partagent le bus de données (de 4 ou 8 bits, si vous utilisez P5 ou un modèle B+). Chaque broche GPIO libre peut adresser une autre paire de registres, tant qu'elle ne danse pas la samba à l'allumage.

Si vous avez vraiment besoin de beaucoup plus de sorties, alors un circuit décodeur de type 74HCT138 peut venir à la rescousse. Cette puce est un petit décodeur 3 bits vers 8, qui prend une combinaison de 3 bits pour activer une seule des 8 sorties (ce qui économise 5 des précieuses GPIO du Pi).

Le signal en sortie du 138 est inversé (actif à l'état bas), mais il suffit d'échanger la fonction des registres d'une paire pour compenser la polarité.

Il faut toutefois tenir compte des éventuelles impulsions parasites lors d'un changement de code : si on sélectionne la sortie 0 après la sortie 7, d'autres sorties pourraient recevoir des micro-impulsions et perturber les autres registres.

La solution consiste à allouer une broche supplémentaire du Pi pour activer une des broches d'inhibition du 138. Cela réduit l'économie de broches (il faut 4 broches d'adressage pour sélectionner une des 8 paires de 273) mais au moins, il n'est plus absolument obligatoire de les changer simultanément. Le code d'adresse peut être sur 3 GPIO consécutives (2, 3 et 4, dont celles utilisées par le bus I²C) et la broche d'inhibition (qui contrôle indirectement l'impulsion d'horloge) peut être affectée à une broche solitaire. On obtient alors jusqu'à 64 sorties fiables avec seulement 8 GPIO.

4.6 Codage

Le code qui fait tourner tout cela n'est pas très compliqué, mais il pose une nouvelle question : comment modifier simultanément plusieurs bits sans modifier les autres ? L'interface GPIO permet de mettre simultanément plusieurs bits à 0, ou bien à 1, mais on ne peut pas faire les deux en même temps...

Il faut nécessairement deux accès, un pour mettre à 0 les éventuels bits à 1, l'autre pour mettre à 1 les bits à 0. Pour éviter de se compliquer la vie, le code suivant met tous les bits à 0 (lignes 21, 27) avant d'écrire la valeur binaire (lignes 22, 28). Dans d'autres situations, cela peut créer des impulsions parasites durant le très court instant où la sortie passe à 0, mais ici, la valeur est verrouillée en aval par un des 273 externes.

Ensuite, comment envoyer une impulsion courte, mais pas trop, mais courte quand même, sans écrire beaucoup de code ? D'un côté, le cœur ARM tourne à 700 MHz, soit environ 1,4 ns par instruction en théorie. Si on veut attendre environ 20 ns, il faut forcer l'exécution de 14 instructions dans le vide...

La solution suivante utilise un autre moyen de temporisation : elle repose sur la latence du bus interne dédié aux entrées-sorties et certains périphériques lents. Ce bus périphérique est cadencé à *seulement* 250 MHz et est plus étroit (probablement 8 ou 16 bits de large), donc il faut plusieurs cycles pour accéder aux GPIO. Les différents tampons sur les bus ajoutent encore de la latence, et des attentes sont imposées lorsque le sens de transfert change. Pour des transferts efficaces, il est préférable de coder plusieurs lectures consécutives, suivies d'écritures consécutives, au lieu de les mélanger.

Mais puisque nous voulons attendre, nous allons justement faire l'inverse : intercaler une lecture au milieu des écritures. Nous n'avons pas besoin de la valeur lue, mais il ne faut pas non plus que le compilateur élimine le code correspondant, donc elle sera mélangée à une autre valeur volatile. C'est le rôle de la macro **SETTLE** (ligne 12) qui mélange le registre **GPIO_LEV** à la variable **Dummy** avec l'opérateur XOR.

Une fois que tout cela est testé et mesuré (grâce à un oscilloscope), il ne reste plus qu'à ajouter une autre fonction parachute, **reg_cleanup()**, qui effacera les registres à la fin du programme.

Fichier

```
01: #include "PI_GPIO.c"
02:
03: #define PIN_273_SEL  (17)  // con.: pin 11
04:
05: #define PIN_273_DAT0 (22)  // con.: pin 15
06: #define PIN_273_DAT1 (23)  // con.: pin 16
07: #define PIN_273_DAT2 (24)  // con.: pin 18
08: #define PIN_273_DAT3 (25)  // con.: pin 22
09:
10: volatile unsigned int Dummy;
11: // délai de 120 ns environ :
12: #define SETTLE  Dummy^=(GPIO_LEV)
13: // Le xor vers une valeur volatile ne devrait pas être optimisé
14: // par le compilateur, l'accès aux registres GPIO non plus.
15: // Attention, car la durée dépend aussi des accès précédents
16: // aux registres d'entrées-sorties, en lecture comme en écriture.
17:
18: // dure approx. 500ns :
19: void set_273(unsigned val) {
20:   // moitié basse
21:   GPIO_CLR = ( 15        << PIN_273_DAT0);
22:   GPIO_SET = ((15 & val) << PIN_273_DAT0);
23:   SETTLE;
24:   GPIO_SET_N(PIN_273_SEL);
25:   SETTLE;
26:   // moitié haute
27:   GPIO_CLR = ( 15             << PIN_273_DAT0);
28:   GPIO_SET = ((15 & (val>>4)) << PIN_273_DAT0);
29:   SETTLE;
30:   GPIO_CLR_N(PIN_273_SEL);
31:   SETTLE;
32: }
33:
34: void reg_cleanup() {
35:   set_273(0);
36:   fputs("\nregistres remis à 0 ", stderr);
37: }
38:
39: void reg_init() {
40:   PI_GPIO_config(PIN_273_SEL,  BCM_GPIO_OUT);
41:   PI_GPIO_config(PIN_273_DAT0, BCM_GPIO_OUT);
42:   PI_GPIO_config(PIN_273_DAT1, BCM_GPIO_OUT);
43:   PI_GPIO_config(PIN_273_DAT2, BCM_GPIO_OUT);
44:   PI_GPIO_config(PIN_273_DAT3, BCM_GPIO_OUT);
45:   set_273(0);
46:   atexit(reg_cleanup);
47: }
```

Avec un temps d'exécution de 500 ns, la vitesse maximale est d'environ deux millions de rafraîchissements par seconde. Une temporisation plus courte améliorerait cette performance, mais lorsque ce n'est pas critique, on préfère optimiser la compacité du code. On réservera aussi des GPIO pour les signaux les plus rapides.

CONCLUSION

Les GPIO sont une ressource essentielle du Raspberry Pi, car ils lui ouvrent les portes vers d'innombrables périphériques sur mesure, qui n'existent pas sur d'autres bus. J'espère que ces codes sources vous rendront de grands services. Ils sont aussi la base pour d'autres fonctions plus évoluées, comme ma bibliothèque de fonctions SPI, ou la plupart de mes projets à base de Raspberry Pi. Amusez-vous bien ! ■

RÉFÉRENCES

[1] La documentation de l'interface GPIO du kernel Linux : **https://www.kernel.org/doc/Documentation/gpio/**

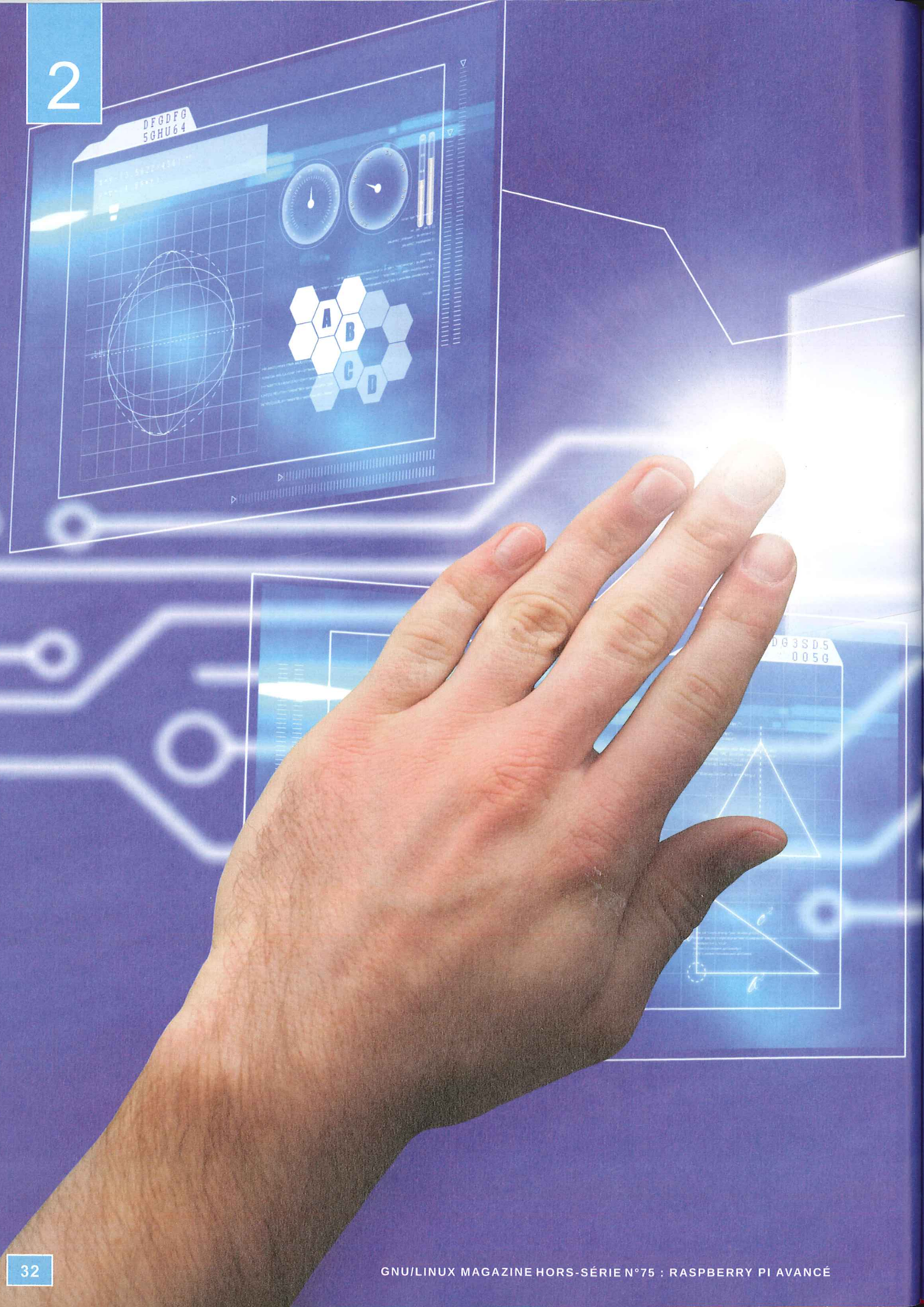
[2] Guidon Y., « Comment contrôler les GPIO du Raspberry Pi par HTTP en C », *Open Silicium* n°6, mars 2013. **http://connect.ed-diamond.com/Open-Silicium/OS-006/Comment-controler-les-GPIO-du-Raspberry-Pi-par-HTTP-en-C**

[3] Guidon Y., « Interfaçage de GHDL avec le port parallèle sous Linux », *GNU/Linux Magazine France* n°133, décembre 2010, p. 74. **http://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-133/Interfacage-de-GHDL-avec-le-port-parallele-sous-Linux**. Code source : **http://ygdes.com/GHDL/io_port/**

[4] Adams J., « Introducing Raspberry Pi HATs », **http://www.raspberrypi.org/introducing-raspberry-pi-hats/**

Pour l'assignation des broches du connecteur et la correspondance avec le numéro de GPIO, consultez **http://www.raspberrypi.org/documentation/hardware/raspberrypi/README.md** (qui renvoie vers l'incontournable **http://elinux.org/RPi_Low-level_peripherals**).

Voir aussi **http://www.blaess.fr/christophe/files/article-2014-08-07/Connecteur_P1.pdf** et **http://www.blaess.fr/christophe/2014/08/06/b/**.



2 INTERFACE SPI

À découvrir dans cette partie...

2 INTERFACE SPI

SPI ET RASPBERRY PI

Christophe Blaess

La communication suivant le protocole SPI est très rapide, car les fréquences sont élevées et le dialogue totalement bidirectionnel « full-duplex ». Ce type de communication sert typiquement pour établir un lien entre un processeur et des périphériques (capteurs, etc.), mais on peut aussi l'utiliser pour dialoguer avec un microcontrôleur, comme nous le ferons dans l'article suivant.

1. SPI

Le protocole SPI (*Serial Peripheral Interface*) implémente une liaison série synchrone entre un maître et un esclave. Lorsqu'un seul esclave est employé, trois signaux seulement (outre la masse) sont nécessaires.

Le maître produit une horloge (signal `SCLK`) envoyé à l'esclave. Sur certaines transitions de cette horloge, l'esclave lira (sur le signal nommé `MOSI` – *Master Out Slave In*), ou écrira (sur le signal nommé `MISO` – *Master In Slave Out*) des données. Il existe plusieurs dénominations suivant les fabricants de matériel pour décrire ces signaux. Il est recommandé d'utiliser la notation `MISO` / `MOSI` (la plus répandue), car elle supprime toute ambiguïté : la broche `MOSI` d'un maître doit toujours être reliée à la broche `MOSI` d'un esclave, et de même pour leurs broches `MISO`.

Si plusieurs esclaves doivent être reliés au même hôte, il pourront être branchés en parallèle (toutes les broches `MISO` reliées entre elles et toutes les broches `MOSI` également), mais il faudra un signal supplémentaire (`CS` – *Chip Select*) pour chacun d'eux, afin de choisir avec lequel la communication est établie à un moment donné (voir figure 1).

On peut noter qu'il existe également un schéma de connexion multi-esclave nommé « *daisy chain* », où ils se transmettent les données en série, la sortie `MISO` de chaque esclave (sauf le dernier de la chaîne) étant reliée à l'entrée `MOSI` du suivant. Mais ceci sort du propos de notre article.

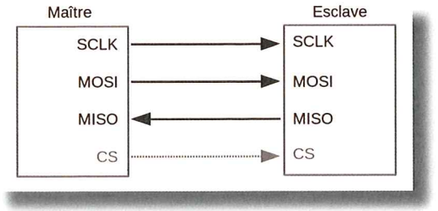


Fig. 1 : Connexions entre maître et esclave

Le noyau Linux Vanilla n'implémente que le côté maître du protocole, même si le support de la partie esclave existe sous forme de patches indépendants que l'on peut ajouter à des drivers de contrôleurs SPI bien spécifiques.

1.1 Signal d'horloge

Le signal d'horloge `SCLK` est essentiel, car c'est lui qui cadence tous les échanges. Il est produit par le maître pour séquencer les transmissions de bits sur les lignes `MOSI` et `MISO`. Le protocole `SPI` étant un standard de fait, sans véritable norme sous-jacente, plusieurs modes de fonctionnement sont tolérés pour l'utilisation du signal d'horloge.

Il existe essentiellement quatre modes possibles. Pour que deux composants puissent dialoguer en SPI, il est donc indispensable de commencer par préciser quel mode nous souhaitons utiliser. La plupart des composants évolués sont capables d'utiliser les quatre modes SPI classiques. Le problème est que suivant les constructeurs, les modes ne sont pas toujours numérotés de manière identique. Nous verrons d'ailleurs une différence entre la notation utilisée par Linux et celle employée dans la documentation du microcontrôleur Texas Instruments MSP430.

Pour déterminer le mode SPI, il est nécessaire de prendre en considération deux paramètres : la polarité (nommée habituellement `POL`) et la phase de l'horloge (nommée `PHA`). En les notant sous forme binaire, nous pourrons obtenir la valeur des quatre modes possibles.

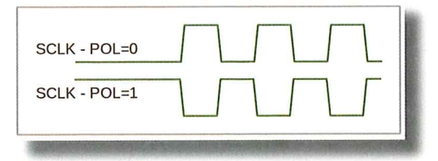


Fig. 2 : Polarité d'horloge

Lorsqu'aucun échange ne se déroule, le signal `SCLK` est au repos. Il peut être au niveau haut ou au niveau bas. C'est ce que l'on nomme la **polarité d'horloge** (voir figure 2). Dans un mode de fonctionnement avec

une horloge à polarité positive (`POL=1`), `SCLK` est au repos au niveau haut, et descend au niveau bas pour indiquer un créneau actif. Inversement, dans un mode à polarité négative (`POL=0`) `SCLK` est au niveau bas au repos, et monte au niveau haut en début de créneau.

En résumé, si `POL=0`, le premier front du signal `SCLK` est un front montant et le second un front descendant. Inversement, dans le cas où `POL=1`, le premier front est descendant et le second montant.

Le second paramètre pour spécifier le mode de communication SPI utilisé est la **phase d'horloge** (`PHA`) et indique sur quels fronts de celle-ci on cadencera les communications MISO et MOSI.

Lorsque la phase `PHA=1`, les signaux seront établis en sortie (du côté *Slave Out* ou *Master Out*) lors du premier front et lus en entrée (du côté *Master In* ou *Slave In*) sur le second front.

Inversement, dans le cas où la phase `PHA=0`, les signaux seront lus en entrée sur le premier front et établis en sortie sur le second front.

Sur la figure 3, on a représenté par une flèche vers le bas le moment où le signal est écrit en sortie et par une flèche vers le haut l'instant où il est lu en entrée.

On remarque que lorsque `PHA=0`, le périphérique esclave doit écrire sa sortie sur la ligne `MISO` avant le premier front d'horloge. Pour cela, il doit déclencher son écriture à l'activation du signal *Chip Select* qui précède toujours le premier front d'horloge, ceci au prix d'un câblage supplémentaire. Pour économiser les lignes de communication et garder un modèle le plus simple possible, nous utiliserons dans la suite de cet article et dans le suivant une configuration avec `PHA=1`.

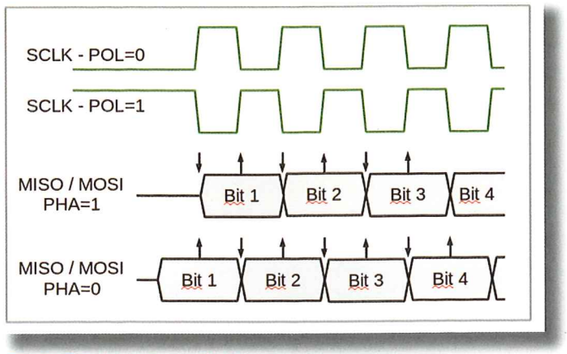


Fig. 3 : Polarité et phase d'horloge

POL	PHA	Mode SPI
0	0	0
0	1	1
1	0	2
1	1	3

Pour simplifier la mise en œuvre des communications SPI, on regroupe généralement les composantes de polarité et de phase en une seule valeur : le **mode** SPI. Cette notation est bien répandue et relativement standardisée, mais elle n'est pas employée par tous les constructeurs.

1.2 MISO et MOSI

Une fois la configuration de `SCLK` réalisée, une seconde surprise attend le programmeur qui découvre le protocole SPI. Il s'agit d'une communication bidirectionnelle *full-duplex*. Autrement dit, à chaque fois que le maître envoie un bit vers un esclave, il en reçoit un en retour en provenance de cet esclave.

L'image qui me vient à l'esprit est celle d'une chaîne de vélo : à chaque maillon qui part vers le pignon de la roue arrière (chaque bit émis sur le signal `MOSI`), un maillon revient sur le plateau du pédalier (un bit est reçu sur `MISO`).

Lorsqu'on voudra faire une simple opération de lecture sur un périphérique, il faut donc être conscient que celui-ci recevra également des données de notre part. De même, le schéma classique « écriture d'une commande » « lecture du résultat » « écriture de la commande suivante », etc., doit être modifié pour prendre en considération le fait que la lecture du premier résultat s'accompagnera déjà de l'écriture de la seconde commande.

2. SPI SUR RASPBERRY PI

Comme le montre la figure 4, le Raspberry Pi propose deux ports SPI accessibles sur son connecteur d'extension P1.

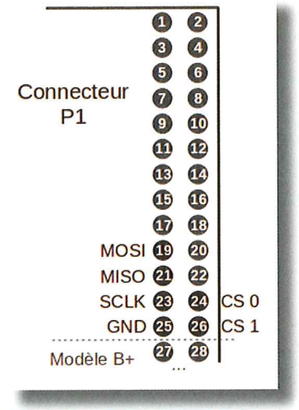


Fig. 4 : Connecteur d'extension du Raspberry Pi

Les broches `MOSI` (19), `MISO` (21) et `SCLK` (23) sont reliées directement à celles des composants esclaves. Chaque broche `CS0` (24) ou `CS1` (26) ne peut être connectée qu'à l'entrée `CS` d'un seul esclave.

Le contrôleur SPI est intégré dans le *system-on-chip* Broadcom 2835 (de la famille 2708) au cœur du Raspberry Pi. Il faut donc charger dans le noyau le module capable de le piloter. Celui-ci se nomme `spi-bcm2708.ko`. Par défaut, les distributions Linux comme la Raspbian empêchent son chargement au démarrage, car il ne concerne qu'un nombre très réduit d'utilisateurs et occupe des ressources système (de la mémoire notamment).

Si l'on veut charger le module dynamiquement, il suffit pour cela de saisir :

Terminal
```
~$ sudo modprobe spi-bcm2708
```

Si on préfère que le module soit chargé automatiquement au démarrage, il faut éditer le fichier `/etc/modprobe.d/raspi-blacklist.conf` et commenter la ligne suivante (en la précédant d'un dièse `#`) :

Fichier
```
blacklist  spi-bcm2708
```

Pour accéder aux interfaces SPI depuis l'espace utilisateur, le noyau nous fournit des points d'entrée sous forme de fichiers spéciaux dans `/dev`. Ceci nécessite le chargement d'un second module particulier :

Terminal
```
~$ sudo modprobe spidev
```

Dès le chargement de ce module, les fichiers spéciaux suivants apparaissent :

Terminal
```
~$ ls -l /dev/spi*
crw-rw---T 1 root spi 153, 0 Aug 11 20:26 /dev/spidev0.0
crw-rw---T 1 root spi 153, 1 Aug 11 20:26 /dev/spidev0.1
```

Naturellement, le fichier `/dev/spidev0.0` correspond au périphérique esclave sélectionné par la broche `CS0` et le fichier `/dev/spidev0.1` à celui correspondant à la broche `CS1`.

Pour configurer les paramètres de communication sur un port SPI, le noyau met à notre disposition un appel système `ioctl()`. Pour la communication proprement dite, on peut combiner des appels système `read()` et `write()`, mais cela ne permet pas de tirer pleinement parti du transfert *full-duplex*. En pratique, on préférera un appel `ioctl()` qui permet de communiquer en s'appuyant sur des structures `spi_ioc_transfer` décrites dans le fichier `<linux/spi/spidev.h>`.

2.1 Le projet spi-tools

La programmation *full-duplex* en C avec les `ioctl()` est parfois un peu complexe. Pour établir rapidement un lien SPI avec un périphérique depuis un script shell par exemple, je vous propose d'utiliser un petit package que j'ai développé récemment, et qui permet de simplifier la configuration et la communication bidirectionnelle. Il s'agit d'un projet libre nommé `spi-tools`. Il n'est pas encore intégré dans les distributions, et sera donc téléchargé et compilé sur le Raspberry Pi. Il n'y a pas de dépendances particulières, la compilation est très simple.

Commençons par télécharger le projet :

Terminal
```
~$ git clone https://github.com/cpb-/spi-tools.git
Cloning into 'spi-tools'...
remote: Counting objects: 53, done.
remote: Total 53 (delta 0), reused 0 (delta 0)
Unpacking objects: 100% (53/53), done.
```

Puis, on le compile très simplement ainsi :

Terminal
```
~$ cd spi-tools/
~/spi-tools$ make
cc -Wall -DVERSION=\"0.3.0\" spi-config.c -o spi-config
cc -Wall -DVERSION=\"0.3.0\" spi-pipe.c -o spi-pipe
```

Le projet évolue encore, la version que vous téléchargerez sera probablement différente de celle ci-dessus.

Par défaut, les exécutables sont installés dans `/usr/sbin`, ce qui réclame les droits *root* (toutefois, il est possible de modifier ce répertoire en remplissant la variable d'environnement `INSTALL_DIR` avant le `make install`).

Terminal
```
~/spi-tools$ sudo make install
install spi-config spi-pipe "/usr/sbin/"
```

Le premier outil est `spi-config`, qui permet de consulter ou de modifier les paramètres de communication sur un port SPI :

Terminal
```
~/spi-tools$ spi-config
spi-config: no device specified (use option -h for help).
```

Demandons de l'aide pour voir la liste des options :

Terminal
```
~/spi-tools$ spi-config -h
usage: spi-config options...
  options:
    -d --device=<dev>  use the given spi-dev character device.
    -q --query         print the current configuration.
    -m --mode=[0-3]    use the selected spi mode.
       0: low iddle level, sample on leading edge
       1: low iddle level, sample on trailing edge
       2: high iddle level, sample on leading edge
       3: high iddle level, sample on trailing edge
    -l --lsb={0,1}     LSB first (1) or MSB first (0)
    -b --bits=[7...]   bits per word
    -s --speed=<int>   set the speed in Hz
    -h --help          this screen
    -v --version       display the version number
```

L'option `-q` permet donc de consulter la configuration actuelle, et l'option `-d` de préciser le port concerné.

Terminal
```
~/spi-tools$ spi-config -q -d /dev/spidev0.0
/dev/spidev0.0: mode=0, lsb=0, bits=8, speed=500000
```

Que signifient ces paramètres ?

- `mode` : il s'agit bien sûr du mode SPI. En consultant le tableau plus haut, nous voyons que `POL=0` et `PHA=0`.
- `lsb` : les communications en SPI se font généralement en transmettant le bit de poids fort en premier. En indiquant `lsb=1`, on peut inverser le sens de transmission des octets.
- `bits` : le nombre de bits par caractère.
- `speed` : la vitesse de transmission (fréquence du signal d'horloge) indiquée en Hz. Les communications SPI se font généralement à des vitesses plutôt élevées (plusieurs MHz), ce qui limite la distance entre les équipements pour éviter les effets parasites sur les signaux. Ici, l'horloge est configurée pour une fréquence de 500 kHz.

Comme précisé plus haut, nous allons choisir un mode SPI où la phase de l'horloge est telle que l'écriture des données `MISO` et `MOSI` se fasse sur le premier front. Par exemple, le mode 1.

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

Le second outil du package, `spi-pipe`, permet de dialoguer en *full-duplex* avec un périphérique, en redirigeant son entrée standard et sa sortie standard vers le port SPI indiqué. Nous en verrons des exemples d'utilisation dans le prochain article.

CONCLUSION

Le Raspberry Pi nous permet de dialoguer facilement avec un ou deux périphérique(s) en utilisant le protocole SPI, par l'intermédiaire des ports accessibles directement sur ses broches d'extensions. Dans le prochain article, nous allons établir une communication avec un microcontrôleur. ■

2

INTERFACE SPI

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...

1. MICROCONTRÔLEUR MSP430

Le microcontrôleur Texas Instruments MSP430 a déjà été présenté dans *GNU/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 :

Terminal

```
$ 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 commandes (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`.

1.1 Compilation d'un programme C pour MSP430

Les exemples de cet article se trouvent sur un dépôt GitHub, on peut les télécharger ainsi :

Terminal

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

Fichier

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

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

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 :

Terminal

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

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

Terminal

```
~/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 sur microcontrôleur d'utiliser des suffixes explicites (`.hex`, `.bin`, `.elf`, etc.) afin d'éviter les confusions lors du transfert vers la cible.

1.2 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 :

Terminal

```
$ 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` :

Terminal

```
$ 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.

1.3 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.

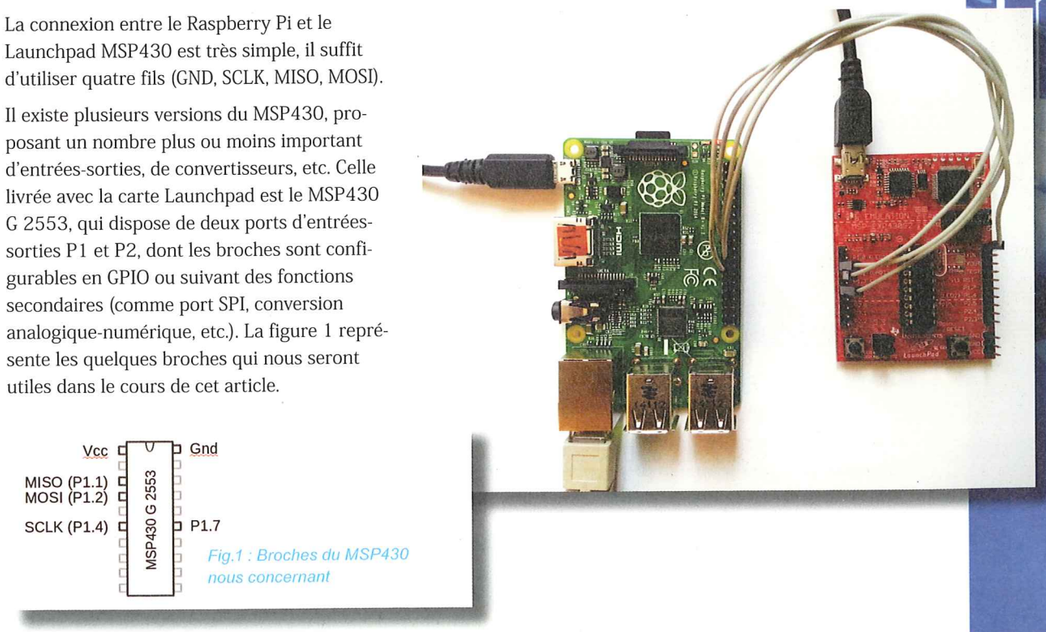


Fig.1 : Broches du MSP430 nous concernant

2. 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.

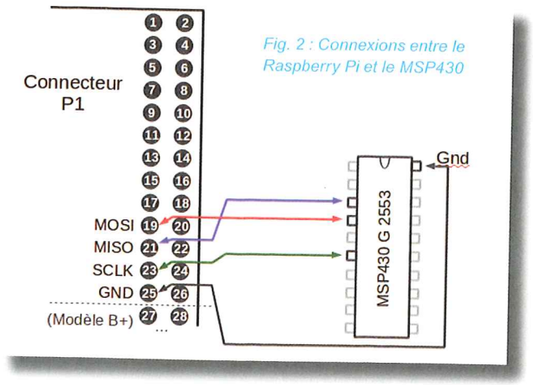


Fig. 2 : Connexions entre le Raspberry Pi et le MSP430

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 outils **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.

2.1 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 reçues depuis la ligne **MISO** du SPI.

Le principe général d'utilisation est :

Terminal
```
$ <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.

Terminal
```
~$ spi-config -d /dev/spidev0.0 -m 1
~$ 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 ligne 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).

Terminal
```
~$ 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 :

Terminal
```
~$ 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 :

Fichier
```
// 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 :

Terminal
```
~$ printf "AZERTY" | spi-pipe -d /dev/spidev0.0 | hexdump -C
00000000  01 42 5b 46 53 55                                 |.B[FSU|
~$ 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 :

Fichier
```
// 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 :

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

2.2 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'interruptions. Dans la syntaxe de programmation du MSP430, on remplacera la boucle active ainsi :

Fichier

```
// 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 :

Terminal

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

2.3 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 kiloohms par exemple entre la masse et la broche Vcc, et relier son point central à l'entrée `P1.7` comme sur la figure 3.

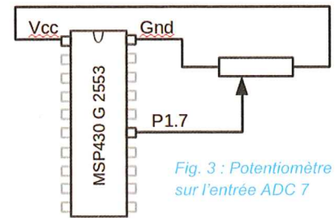


Fig. 3 : Potentiomètre sur l'entrée ADC 7

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 appelé 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 :

Fichier

```
// 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 10 bits est décalée vers la droite de 2 bits (perdant ainsi les 2 bits de poids faible) afin de tenir dans le registre 8 bits. Si on voulait conserver la résolution, il faudrait envisager de transmettre la valeur sur deux octets successifs (2 bits de poids forts et 8 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.

Terminal

```
~$ 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) :

Terminal

```
~$ 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.

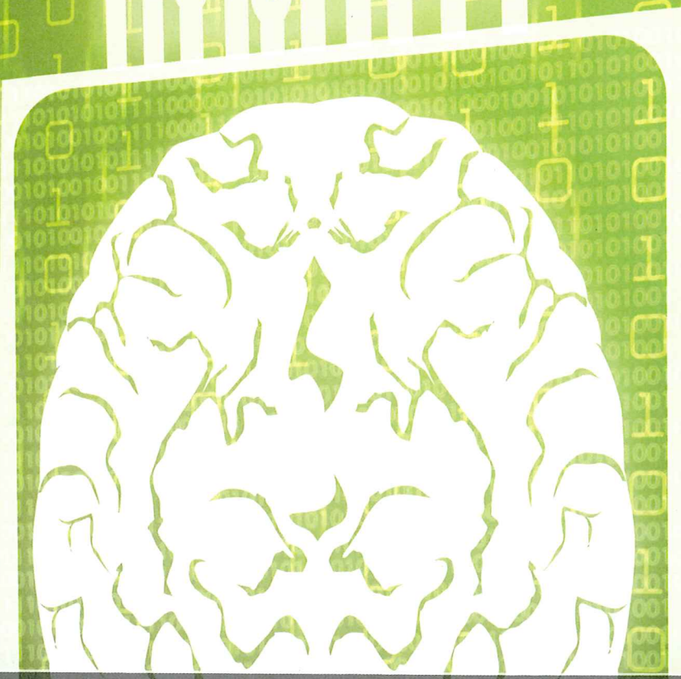
En outre, il peut être intéressant de sous-traiter les opérations d'entrées-sorties automatiques (comptage, décodage de protocoles, calcul de checksum, etc.) à un microcontrôleur pour laisser le processeur du Raspberry Pi libre d'effectuer des traitements plus complexes (interface utilisateur, réseau,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

3 DISTRIBUTIONS ET OS

À découvrir dans cette partie...

3 DISTRIBUTIONS ET OS



LA COMPILATION CROISÉE AVEC VOTRE RASPBERRY PI

Yann Morère

Le Raspberry Pi est une plateforme bon marché, mais ne possède pas un processeur très puissant. Un simple programme « Hello World ! » met plusieurs secondes à compiler.... Et un noyau Linux plus de 15 heures... On se rend bien compte qu'il n'est pas vraiment possible de développer directement sur cette plateforme sans une perte de temps considérable. Pour pallier cela, nous allons utiliser la compilation croisée ou « cross compilation » pour réduire drastiquement les temps de génération des programmes.

La compilation croisée sur Raspberry Pi a déjà été traitée par Christophe Blaess dans les numéros de *GNU/Linux Magazine* 155 et 158 à l'aide de Buildroot et plutôt dédiée aux systèmes « from scratch ». Je vous propose ici de découvrir pas à pas sa mise en œuvre sur des exemples simples à l'aide d'autres outils.

D'après **[1]**, La compilation croisée est la possibilité, sur une machine possédant une architecture matérielle spécifique et un système d'exploitation particulier, de compiler des programmes pour d'autres architectures et/ou systèmes d'exploitation.

On peut donc compiler un programme sur sa machine de bureau (vraisemblablement à base de processeur x86/64) à destination de son téléphone mobile, qui lui, utilise un autre système d'exploitation et un processeur ARM. Dans notre cas, il s'agit de compiler des programmes pour le système Linux utilisant des processeurs ARM.

La compilation croisée peut être intéressante à mettre en œuvre à plus d'un titre :

- Dans le cas d'architectures identiques mais de systèmes d'exploitation différents, cela vous évite de redémarrer la machine pour compiler vos binaires sur le bon système.
- Elle devient indispensable, lorsque les outils de développement et de compilation ne sont pas disponibles sur la plateforme de destination. Les compilateurs sont plutôt rares sur les smartphones et même s'ils existaient, leur mise en œuvre serait plus que fastidieuse. On en comprend aussi tout l'intérêt, lorsque l'on développe pour des microcontrôleurs (l'Arduino en est un exemple).
- La puissance de calcul des plateformes cibles est souvent plus modeste que celle de votre machine, et la compilation croisée vous apportera une rapidité de génération des programmes.
- Des problèmes d'espace disque peuvent aussi vous obliger à utiliser la compilation croisée, dans le cas de plateforme de destination utilisant des unités de stockage de faible capacité (carte SD par exemple).
- Elle vous permet aussi de générer des exécutables pour un système dont vous ne possédez pas la licence, vous évitant ainsi des coûts supplémentaires (par exemple, pour Windows on utilisera Wine pour tester les exécutables produits).

Comme pour la compilation standard, il n'est pas garanti que le programme compilé fonctionne, et vous devrez faire des tests à l'aide d'émulateurs ou de machines possédant l'architecture et le système d'exploitation de destination.

Le « cross compilateur » sera alors capable de générer du code exécutable pour une autre architecture que celle sur laquelle il s'exécute. Cependant, pour pouvoir compiler pour une architecture spécifique des programmes complexes utilisant de nombreuses librairies externes, il vous faudra les librairies compilées pour cette architecture.

Autre point important, le compilateur seul ne suffit pas, une série de programmes et d'utilitaires qui permettent la création de l'exécutable final est nécessaire : cela se nomme la chaîne de compilation (« compilation toolchain »). Dans notre cas, nous utiliserons la chaîne de compilation GNU. Elle comporte les éléments suivants :

- `binutils` : une collection d'utilitaires utilisés pour le traitement des fichiers binaires. Il contient l'assembleur qui transforme le pseudo-code généré par la compilation en instructions comprises par le processeur cible, et l'éditeur des liens pour lier les bibliothèques utilisées par un programme.
- `gcc` : les compilateurs C et C++ de la collection ;
- `glibc` : bibliothèque C du système utilisée pour les appels au noyau et le traitement des processus de bas-niveau ;
- les en-têtes du noyau requis par la bibliothèque glibc (`linux-headers` dans le cas du noyau Linux) ;
- `gdb` : composant optionnel pour le débogage d'une chaîne binaire déjà compilée.

3

À savoir

Si vous désirez reproduire l'ensemble des manipulations de cet article, il vous faudra un PC sous Linux (j'utilise ici une distribution Arch Linux, mais tout est reproductible sur une autre distribution) et un Raspberry Pi muni d'un système Raspbian. Les deux machines devront avoir une configuration réseau fonctionnelle (accès à Internet, connexion SSH) et se trouver dans le même sous-réseau pour plus de simplicité.

Dans la suite de cet article, nous présenterons l'installation d'une chaîne de compilation pour Arch Linux et les outils Linaro dédiés au Raspberry Pi (basé sur Debian et Raspbian). Ensuite, nous découvrirons trois exemples de compilation et les méthodes associées.

1. INSTALLATION DE LA CHAÎNE DE COMPILATION

1.1 Sur Arch Linux pour Arch Linux

L'installation avec Arch Linux est très simple, car les outils sont disponibles sur les archives AUR (*Arch User Repository*). Il suffit de lancer la commande suivante :

Terminal
```
$ yaourt -S arm-linux-gnueabi-gcc
```

La suite sera bien plus longue ; la compilation des outils passe par 3 étapes différentes, avec pour chacune d'elles des compilations et des installations. Finalement, vous obtenez une série d'outils dont les exécutables commencent tous par `arm-*`. Voici, à titre d'exemple, le compilateur C pour architecture ARM :

Terminal
```
$ file /usr/bin/arm-linux-gnueabi-gcc-4.9.0
/usr/bin/arm-linux-gnueabi-gcc-4.9.0: ELF 64-bit LSB executable,
x86-64, version 1 (SYSV), dynamically linked (uses shared libs),
for GNU/Linux 2.6.32, BuildID[sha1]=d29d7410d3c6028d4a8f9c1aea81bd0
8d4e64fd3, stripped
```

Le compilateur est un programme fonctionnant sur une architecture x86 64 bits et permettra de produire des exécutables pour l'architecture ARM au format 32 bits.

Testons maintenant nos nouveaux outils. On commence par créer un fichier source C `hello.c` :

Fichier
```
#include <stdio.h>
int main(void)
{
  printf("Hello, cross-compilation world !\n");
  return 0;
}
```

Ensuite, on le compile à l'aide de la commande suivante :

Terminal
```
$ arm-linux-gnueabi-gcc -o hello hello.c
```

Notre fichier étant prévu pour le Raspberry Pi, il ne pourra être exécuté sur la machine ayant servi à le compiler :

Terminal
```
$ ./hello
-bash: ./hello: cannot execute binary file: Erreur de format pour exec()
```


La commande `file` nous le confirme :

Terminal
```
$ file ./hello
./hello: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 3.13.3, not stripped
```

On peut alors tester notre programme en le téléversant sur notre Raspberry Pi :

Terminal
```
$ scp hello pi@192.168.1.67:temp
pi@192.168.1.67's password:
hello                                   100% 5346     5.2KB/s   00:00
```

Puis, après s'être connecté sur la framboise, on lance notre programme, qui maintenant fonctionne :

Terminal
```
pi@raspberrypi ~/temp $ ./hello
Hello, cross-compilation world !
```

Il est important de noter qu'il est préférable d'utiliser les outils de compilation croisée dédiés à chaque distribution Linux lorsqu'il s'agit d'applications utilisant de nombreuses bibliothèques : par exemple, il est préférable d'utiliser la chaîne de compilation Arch Linux si le système d'exploitation de destination est aussi une Arch Linux. Le cas échéant, vous pouvez rencontrer des problèmes d'exécution des programmes binaires à cause de versions de bibliothèques différentes (par exemple, binaires compilés sur Arch Linux et exécutés sur Raspbian).

Pour la compilation d'un noyau, cela n'a pas d'importance, car peu de bibliothèques sont mises à contribution.

Dans la suite, je destine mes programmes au système Raspbian, je vais donc utiliser des outils prévus pour cette distribution.

1.2 Sur Arch Linux pour Raspbian

Nous allons utiliser dans la suite le compilateur GCC Linaro pour l'architecture Raspberry Pi. « Linaro est une association internationale à but non lucratif travaillant à la consolidation et à l'optimisation des logiciels libres à base Linux pour les processeurs d'architecture ARM » [2]. Linaro GCC est une branche spécifique de la version stable actuelle de GCC et comprend des « backports », des améliorations et corrections de bugs que Linaro et d'autres contributeurs ont réalisé.

La disponibilité des outils en version 64 bits est assez récente et, lors de mes tests, seuls les programmes en version 32 bits étaient disponibles. Donc, afin de pouvoir utiliser les outils de compilation croisée Linaro sur mon système Arch Linux 64 bits, il m'a fallu installer quelques bibliothèques supplémentaires pour pouvoir utiliser des applicatifs 32 bits [5].

On commence par ajouter le dépôt `multilib` à Pacman. On édite le fichier `/etc/pacman.conf` et on ajoute les lignes suivantes :

Fichier
```
[multilib]
SigLevel = PackageRequired
Include = /etc/pacman.d/mirrorlist
```

À savoir

Crosstool-NG vous permettra de simplifier la construction d'une chaîne de compilation pour une architecture spécifique. Je vous renvoie à l'adresse [3] pour la compilation et la mise en œuvre de ces outils dans le cas de l'architecture du Raspberry Pi. Il est possible de construire les outils de compilation Linaro comme décrit à l'adresse [4].

On peut alors installer les outils nécessaires :

Terminal
```
$ sudo pacman -S multilib/lib32-gcc-libs multilib/lib32-zlib
```

Ensuite, on récupère l'ensemble des outils simplement à partir du dépôt GitHub :

Terminal
```
$ git clone git://github.com/raspberrypi/tools.git --depth 1
```

Une fois les outils téléchargés, on peut faire un test de compilation :

Terminal
```
$ ../tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-gcc -o hello32
hello.c
$ file ../tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-gcc-4.8.3
../tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/bin/arm-linux-gnueabihf-gcc-4.8.3: ELF
32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/
Linux 2.6.15, stripped
```

On peut aussi compiler le programme avec la version 64 bits des outils :

Terminal
```
$ ../tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin/arm-linux-gnueabihf-gcc
-o hello64 hello.c
$ file ../tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin/arm-linux-gnueabihf-
gcc-4.8.3
../tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian-x64/bin/arm-linux-gnueabihf-
gcc-4.8.3: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked (uses shared
libs), for GNU/Linux 2.6.24, BuildID[sha1]=04d49c97b46fb3d971e496e6721137f6bab56342, stripped
```

On remarquera que pour un code source identique, la taille des 2 exécutables est différente :

Terminal
```
$ ls -al
total 36
drwxr-xr-x 2 yann users 4096 19 août  15:02 .
drwxr-xr-x 8 yann users 4096 19 août  14:37 ..
-rwxr-xr-x 1 yann users 5985 19 août  15:02 hello32
-rwxr-xr-x 1 yann users 5993 19 août  15:02 hello64
-rw-r--r-- 1 yann users  102 19 août  13:51 hello.c
$ file hello32
hello32: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked (uses shared
libs), for GNU/Linux 2.6.26, BuildID[sha1]=2b3cba1fcfa41d80263837926992bdab5595c607, not stripped
[yann@archery test]$ file hello64
hello64: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically linked (uses shared
libs), for GNU/Linux 2.6.26, BuildID[sha1]=71b3cc3efcb928f2e02e413e516eea6dc73c0023, not stripped
```

Voyons maintenant comment compiler un projet un peu plus gros : un noyau !

2. COMPILATION CROISÉE DE VOTRE NOYAU

2.1 Les préparatifs

Il est bien sûr possible de compiler un noyau directement sur le Raspberry Pi ; cela fonctionne, mais prend tout de même un peu de temps (entre 15h et 17h pour être plus exact). On peut tenter d'« overclocker » le processeur ARM afin de gagner quelques précieuses minutes. Mais si vous possédez sur votre bureau une machine récente, il est bien plus rapide de lui faire faire le travail.

Nous allons voir la compilation et l'installation d'un noyau à travers un exemple concret : la correction d'un bug d'un module pour le support d'un encodeur joystick Arcade.

Il s'agit d'un module électronique qui s'interface en USB et permet de connecter très facilement deux joysticks et des boutons arcade à votre système. Le modèle utilisé est visible sur la page [6]. Cependant, sous Linux, le module noyau gérant ce dernier possède un bug : les valeurs renvoyées pour les directions « Haut » et « Gauche » des joysticks sont en dehors des plages prévues et le noyau les ignore. La conséquence est que les actions « Haut » et « Gauche » sont sans effet dans les émulateurs de jeux. Le descriptif du problème est détaillé dans [7].

Heureusement, un « patch » est disponible et permet de corriger le problème [8]. Voyons comment l'appliquer et créer notre nouveau noyau.

On commence par récupérer les sources du noyau :

Terminal
```
$ mkdir kernel
$ cd kernel/
$ git init
$ git clone --depth 1 git://github.com/raspberrypi/linux.git
```

Ensuite, on télécharge le « patch » :

Terminal
```
$ wget 'http://ithink.ch/blog/files/xin-mo/0001-hid-Add-
new-driver-for-non-compliant-Xin-Mo-devices.patch'
```

Puis, on l'applique :

Terminal
```
$ patch -p1 < 0001-hid-Add-new-driver-for-non-compliant-
Xin-Mo-devices.patch
patching file drivers/hid/Kconfig
Hunk #1 succeeded at 728 with fuzz 2 (offset -15 lines).
patching file drivers/hid/Makefile
Hunk #1 succeeded at 111 (offset 1 line).
patching file drivers/hid/hid-core.c
Hunk #1 succeeded at 1817 (offset 81 lines).
patching file drivers/hid/hid-ids.h
Hunk #1 succeeded at 890 (offset 3 lines).
patching file drivers/hid/hid-xinmo.c
```

2.2 La compilation

Avant de construire le noyau, on s'assure que l'arborescence de compilation est propre :

Terminal
```
$ make mrproper
```

Ensuite, on récupère la configuration courante du noyau Linux sur une distribution fonctionnelle et à jour de votre Raspberry Pi.

Terminal
```
pi@raspberrypi ~ $ zcat /proc/config.gz > .config
```

On la copie dans le répertoire `linux` de la machine qui va réaliser la compilation croisée :

À savoir

Les correctifs de ce bug ont depuis été intégrés dans les sources officielles du noyau et la compilation n'est plus nécessaire depuis le noyau 3.12.25.

À savoir

Les essais ont été faits avec un noyau 3.10.y ; si vous réalisez les manipulations avec un noyau de version supérieure à 3.12.25, la commande `patch` vous indiquera que les sources sont déjà modifiées.

Terminal
```
$ pwd
/home/yann/raspberry/kernel/linux
[yann@archery linux]$ scp pi@192.168.1.67:.config .
pi@192.168.1.67's password:
.config                   100%   85KB  85.0KB/s   00:00
```

On lance la configuration en mode console pour activer le support de notre module Xin-Mo dans le noyau en tant que module :

Terminal
```
$ make oldconfig
```

On répond par défaut à toutes les questions, sauf celle concernant le module Xin-Mo :

Terminal
```
Xin-Mo non-fully compliant devices (HID_XINMO) [N/m/y/?] (NEW)
```

Il est aussi possible d'utiliser l'interface « dialog » grâce à la commande :

Terminal
```
$ make menuconfig
```

Ensuite, dans le menu ***Device Drivers > HID support > Special HID drivers > Xin-Mo non-fully compliant devices***, on active le module comme le montre la figure 1*.

Finalement, on sélectionne le bouton ***Exit*** et on enregistre la nouvelle configuration dans le fichier `.config`.

Voilà, tout est prêt pour la génération d'un nouveau noyau corrigeant le bug de notre module arcade !

Dans le cas de mon installation par l'archive AUR, les exécutables créés sont localisés dans `/usr/bin` et sont tous préfixés par `arm-linux-gnueabi-`. Le compilateur recherché par défaut pour la construction sera `gcc` et les outils associés (`ar`, `nm`, etc.).

Cependant, comme nous voulons utiliser la version du compilateur pour l'architecture ARM, il est nécessaire de renseigner la variable `CROSS_COMPILE` avec `arm-linux-gnueabi-`, afin que ce préfixe soit ajouté à `gcc` de manière à appeler les bons programmes de compilation.

Deux autres cas peuvent survenir :

- Les outils de compilation ARM sont dans un répertoire spécifique et possèdent des liens symboliques de type `gcc`, `g++`, etc. On renseigne alors la variable comme suit :

Fichier
```
export CCPREFIX=/path/to/your/compiler/binary/
```

- Les outils de compilation ARM sont dans un répertoire spécifique et ne possèdent pas de liens symboliques de types `gcc`, `g++`, etc. :

Fichier
```
export CCPREFIX=/path/to/your/compiler/binary/prefix-of-binary-
```

Dans mon cas, il n'y a pas besoin d'ajouter le chemin vers les outils ARM, puisque tout est dans `/usr/bin`. Je peux « simplement » utiliser :

Terminal
```
$ export CCPREFIX=arm-linux-gnueabi-
$ make ARCH=arm CROSS_COMPILE=${CCPREFIX} -j2 ; make ARCH=arm CROSS_
COMPILE=${CCPREFIX} -j2 modules
```


Ou encore :

Terminal
```
$ make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- -j2 ; make ARCH=arm
CROSS_COMPILE=arm-linux-gnueabi- -j2 modules
```

Par contre, si vous désirez utiliser les outils Linaro pour la compilation de votre noyau, il faudra renseigner de la manière suivante :

Terminal
```
$ make ARCH=arm CROSS_COMPILE=/home/yann/raspberry/tools/arm-bcm2708/gcc-
linaro-arm-linux-gnueabihf-raspbian-x64/bin/arm-linux-gnueabihf- -j2 ; make
ARCH=arm CROSS_COMPILE=/home/yann/raspberry/tools/arm-bcm2708/gcc-linaro-
arm-linux-gnueabihf-raspbian-x64/bin/arm-linux-gnueabihf- -j2 modules
```

Il est maintenant temps de remplir votre wiki pour documenter vos différentes activités du jour... ou bien aller boire un café avec les collègues !

La compilation terminée, passons à l'installation de notre nouveau noyau.

2.3 Installation du noyau

Avant de transférer noyau et modules vers le Raspberry Pi, pour plus de simplicité et éviter de copier de nombreux fichiers inutiles, nous allons réaliser une installation dans un répertoire de la machine de compilation :

Terminal
```
$ export CCPREFIX=/usr/bin/arm-linux-gnueabi-
$ export MODULES_TEMP=/home/yann/raspberry/kernel/linux/modules
$ make ARCH=arm CROSS_COMPILE=${CCPREFIX} INSTALL_MOD_PATH=${MODULES_TEMP}
modules_install
```

Les commandes précédentes créent le répertoire `modules` contenant les répertoires `lib` et `firmware`. Il suffira ensuite de remplacer les répertoires existant sur le Raspberry Pi par ceux fraîchement créés.

Le noyau compilé se trouve dans le répertoire `arch/arm/boot/Image`. Nous allons créer une archive contenant noyau et modules et la transférer sur le Raspberry Pi :

Terminal
```
$ cp arch/arm/boot/Image ./kernel-3.10.36-xinmo
$ tar czf kernel-3.10.36-xinmo.tar.gz kernel-3.10.36-xinmo modules
$ scp kernel-3.10.36-xinmo.tar.gz pi@192.168.1.67:
pi@192.168.1.67's password:
kernel-3.10.36-xinmo.tar.gz          100%   15MB   3.7MB/s   00:04
```

Avant de décompresser l'archive, pensez à mettre à l'heure votre système ! Le cas échéant, vous obtiendrez de nombreuses erreurs de *timestamp* (le Raspberry Pi ne possède pas d'horloge RTC et sa mise à l'heure se fait via le réseau et un serveur de temps).

Terminal
```
pi@raspberrypi ~ $ sudo ntpdate 192.168.1.254
 9 Apr 07:13:50 ntpdate[2171]: step time server 192.168.1.254 offset 315265.763132 sec
pi@raspberrypi ~ $ tar xzf kernel-3.10.36-xinmo.tar.gz
```

L'étape suivante consiste à copier le nouveau noyau dans le répertoire `/boot` :

Terminal
```
pi@raspberrypi ~ $ sudo cp kernel-3.10.36-xinmo /boot/
```

Ensuite, on fait une sauvegarde (pour revenir en arrière en cas de problème pour des versions de noyau identiques) des répertoires `lib` et `firmware` :

Terminal
```
pi@raspberrypi ~ $ cd /lib
pi@raspberrypi /lib $ sudo cp -r firmware firmware_old
pi@raspberrypi /lib $ sudo cp -r modules modules_old
pi@raspberrypi ~ $ cd
```

Puis, on copie les nouvelles versions :

Terminal
```
pi@raspberrypi ~ $ sudo cp -R modules/lib/firmware /lib/
pi@raspberrypi ~ $ sudo cp -R modules/lib/modules /lib/
```

Afin d'utiliser ce nouveau noyau au démarrage, on va ajouter (ou modifier) le ligne suivante au (du) fichier `/boot/config.txt` :

Fichier
```
kernel=kernel-3.10.36-xinmo
```

Il ne reste plus qu'à redémarrer et à tester notre module Xin-Mo :

Terminal
```
$ sudo reboot
```

On se reconnecte et on vérifie la nouvelle version du noyau :

Terminal
```
$ ssh pi@192.168.1.63
pi@192.168.1.63's password:

Welcome in your RaspberryPI (Moebius Release)
$ uname -a
Linux raspberrypi 3.10.36+ #1 PREEMPT Tue Apr 8 17:13:47 CEST
2014 armv6l GNU/Linux
```

On branche le module et on vérifie qu'il est reconnu grâce à la commande `dmesg` :

Fichier
```
[  285.035573] usb 1-1.3.2: new low-speed USB device number 8 using dwc_
otg
[  285.145233] usb 1-1.3.2: New USB device found, idVendor=16c0,
idProduct=05e1
[  285.145298] usb 1-1.3.2: New USB device strings: Mfr=1, Product=2,
SerialNumber=0
[  285.145320] usb 1-1.3.2: Product: THT Arcade console 2P USB Player
[  285.145336] usb 1-1.3.2: Manufacturer: THT
[  285.214536] input: THT THT Arcade console 2P USB Player as /devices/
platform/bcm2708_usb/usb1/1-1/1-1.3/1-1.3.2/1-1.3.2:1.0/input/input3
[  285.216642] xinmo 0003:16C0:05E1.0004: input,hidraw3: USB HID v1.01
Joystick [THT THT Arcade console 2P USB Player] on usb-bcm2708_usb-1.
```

On contrôle les modules chargés et on remarque que le module `hid_xinmo` est bien présent :

Terminal
```
$ lsmod
Module                  Size  Used by
hid_xinmo               1108  0
[...]
```

À savoir

Normalement, il est recommandé de mettre à jour le « firmware » du GPU et des bibliothèques. Je ne l'ai pas fait dans ce cas et cela a fonctionné. Cependant, cela est nécessaire en cas de changement de version majeure de noyau. Par exemple, dans le cas d'un noyau 3.10 vers 3.12 c'est obligatoire, car l'interface du firmware a changé. La commande `rpi-update` vous permettra de réaliser l'opération simplement [9].

Finalement, on teste le fonctionnement du module à l'aide du programme `evtest`, ou encore le programme `jstest` (respectivement paquet `evtest` et `joystick`), afin de vérifier que tout est opérationnel.

Passons maintenant à la compilation d'applications utilisant des bibliothèques graphiques.

3. CROSS-COMPILEZ VOS APPLICATIONS GRAPHIQUES !

Si vous désirez créer et recompiler des programmes graphiques (ou des programmes utilisant des bibliothèques tierces) en compilation croisée afin de gagner du temps, il vous faudra disposer de ces bibliothèques sur la machine de compilation, mais les binaires devront être au format ARM afin de pouvoir réaliser l'édition de lien de votre programme final.

Deux solutions s'offrent à vous :

- Recompiler toutes les librairies nécessaires en cross-compilation vers l'architecture ARM et les installer dans la chaîne de compilation ;
- Récupérer sur la machine de compilation l'arborescence complète des bibliothèques de la distribution qui se trouve sur le Raspberry Pi. Il faut bien sûr que vous ayez installé sur ce système l'ensemble des outils et bibliothèques nécessaires à la compilation de vos programmes (bibliothèques et paquets de développement).

C'est cette seconde option que nous allons choisir, car elle permet de gagner du temps et de s'assurer que notre programme est compilé avec les versions de bibliothèques installées sur notre système ARM. Si vous désirez utiliser la première option, je vous conseille la lecture de [10].

Dans le cadre de cet article, nous allons compiler un programme simple utilisant la bibliothèque graphique GTK+.

Dans un premier temps, il faut installer sur la Raspberry Pi la bibliothèque de développement GTK 3 :

Terminal
```
pi@raspberrypi ~ $ sudo apt-get install libgtk-3-dev
```

Cela installe en même temps toutes les dépendances nécessaires. Il faut maintenant récupérer l'ensemble des répertoires `/lib` et `/usr` afin de pouvoir compiler notre programme. Afin de ne pas être obligés de récupérer l'ensemble des fichiers à chaque mise à jour du système de notre Raspberry, nous allons utiliser `rsync`. On l'installe sur la machine de compilation et sur la Raspberry Pi :

Terminal
```
pi@raspberrypi ~ $ sudo apt-get install rsync
$ sudo pacman -S rsync
```

Ensuite, on récupère l'ensemble des deux répertoires dans le répertoire `raspberry` créé précédemment :

Terminal
```
$ rsync -rl pi@192.168.1.67:/lib .
pi@192.168.1.63's password:
$ rsync -rl pi@192.168.1.67:/usr .
pi@192.168.1.63's password:
```

Les options `-rl` permettent de parcourir l'ensemble des sous-répertoires et de conserver les liens symboliques (très importants dans le cas des bibliothèques partagées).

Pour notre exemple, nous utiliserons le code source du programme GTK du tutoriel « Increase - Decrease », proposé par le site **zetcode.com** [11]. Ce programme simple affiche une fenêtre avec 2 boutons d'incrémentation et décrémentation d'un compteur que l'on affichera.

Construisons maintenant le fichier `Makefile` nécessaire à la compilation [12]. Ce dernier comportera les configurations nécessaires à l'utilisation du compilateur de compilation croisée et des chemins vers les bibliothèques au format ARM que l'on vient de récupérer.

Fichier

```
ARM_PREFIX=/home/yann/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-
raspbian/bin/arm-linux-gnueabihf-
CC = $(ARM_PREFIX)gcc
SRC += gtktest.c
TARGET = gtktest

LIBRARY += gtk-3
LIBRARY += gdk-3
LIBRARY += atk-1.0
LIBRARY += gio-2.0
LIBRARY += pangocairo-1.0
LIBRARY += gdk_pixbuf-2.0
LIBRARY += cairo-gobject
LIBRARY += pango-1.0
LIBRARY += cairo
LIBRARY += gobject-2.0
LIBRARY += glib-2.0
LIBRARY += rt
LIBRARY += pcre

LIBRARYDIR += $(HOME)/raspberry/lib/arm-linux-gnueabihf
LIBRARYDIR += $(HOME)/raspberry/usr/lib/arm-linux-gnueabihf
LIBRARYDIR += $(HOME)/raspberry/lib
LIBRARYDIR += $(HOME)/raspberry/usr/lib

XLINK_LIBDIR += $(HOME)/raspberry/lib/arm-linux-gnueabihf
XLINK_LIBDIR += $(HOME)/raspberry/usr/lib/arm-linux-gnueabihf

INCLUDEDIR += $(HOME)/raspberry/usr/include/gtk-3.0
INCLUDEDIR += $(HOME)/raspberry/usr/include/pango-1.0
INCLUDEDIR += $(HOME)/raspberry/usr/include/gio-unix-2.0/
INCLUDEDIR += $(HOME)/raspberry/usr/include/atk-1.0
INCLUDEDIR += $(HOME)/raspberry/usr/include/cairo
INCLUDEDIR += $(HOME)/raspberry/usr/include/gdk-pixbuf-2.0
INCLUDEDIR += $(HOME)/raspberry/usr/include/freetype2
INCLUDEDIR += $(HOME)/raspberry/usr/include/glib-2.0
INCLUDEDIR += $(HOME)/raspberry/usr/lib/arm-linux-gnueabihf/glib-2.0/include
INCLUDEDIR += $(HOME)/raspberry/usr/include/pixman-1
INCLUDEDIR += $(HOME)/raspberry/usr/include/libpng12

OPT = -O0
DEBUG = -g
WARN= -Wall
PTHREAD= -pthread

INCDIR  = $(patsubst %,-I%,$(INCLUDEDIR))
LIBDIR  = $(patsubst %,-L%,$(LIBRARYDIR))
LIB     = $(patsubst %,-l%,$(LIBRARY))
XLINKDIR = $(patsubst %,-Xlinker -rpath-link=%,$(XLINK_LIBDIR))

all:
  $(CC) $(OPT) $(DEBUG) $(WARN) $(LIBDIR) $(PTHREAD) $(INCDIR) $(XLINKDIR) $(LIB)
$(SRC) -o $(TARGET)

clean:
  rm -rf $(TARGET)
```


Le première ligne donne le chemin complet vers les outils de compilation Linaro précédemment installés. On définit ensuite les sources à compiler, ainsi que le nom du programme. On liste ensuite les différentes bibliothèques nécessaires à la création de notre programme, puis on donne l'ensemble des chemins vers ces différents composants, ainsi que vers les composants système nécessaires à l'édition de lien. Viennent ensuite les chemins vers les inclusions, puis les options du compilateur. Finalement, on termine par la génération des différents chemins complets.

L'utilisation de la fonction `patsubst` [13] va nous permettre de remplacer les chemins par défaut d'inclusion de fichier et d'accès aux bibliothèques pour un programme GTK (c'est-à-dire ceux de l'ordinateur de compilation) par ceux que nous avons définis et qui nous permettront de compiler pour l'architecture ARM.

On termine par la ligne permettant la génération de notre exécutable.

Tout est prêt, on lance la commande `make` et nous obtenons une erreur !

Terminal

```
$ make
/home/yann/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-
raspbian/bin/arm-linux-gnueabihf-gcc -O0 -g -Wall  -L/home/yann/raspberry/
lib/arm-linux-gnueabihf -L/home/yann/raspberry/usr/lib/arm-linux-gnueabihf
-L/home/yann/raspberry/lib -L/home/yann/raspberry/usr/lib -pthread -I/
home/yann/raspberry/usr/include/gtk-3.0 -I/home/yann/raspberry/usr/
include/pango-1.0 -I/home/yann/raspberry/usr/include/gio-unix-2.0/ -I/
home/yann/raspberry/usr/include/atk-1.0 -I/home/yann/raspberry/usr/
include/cairo -I/home/yann/raspberry/usr/include/gdk-pixbuf-2.0 -I/
home/yann/raspberry/usr/include/freetype2 -I/home/yann/raspberry/usr/
include/glib-2.0 -I/home/yann/raspberry/usr/lib/arm-linux-gnueabihf/glib-
2.0/include -I/home/yann/raspberry/usr/include/pixman-1 -I/home/yann/
raspberry/usr/include/libpng12 -Xlinker -rpath-link=/home/yann/raspberry/
lib/arm-linux-gnueabihf -Xlinker -rpath-link=/home/yann/raspberry/usr/lib/
arm-linux-gnueabihf -lgtk-3 -lgdk-3 -latk-1.0 -lgio-2.0 -lpangocairo-1.0
-lgdk_pixbuf-2.0 -lcairo-gobject -lpango-1.0 -lcairo -lgobject-2.0
-lglib-2.0 -lrt -lpcre gtktest.c -o gtktest
/home/yann/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-
raspbian/bin/../lib/gcc/arm-linux-gnueabihf/4.8.3/../../../../arm-linux-
gnueabihf/bin/ld: cannot find /lib/arm-linux-gnueabihf/libpthread.so.0
/home/yann/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-
raspbian/bin/../lib/gcc/arm-linux-gnueabihf/4.8.3/../../../../arm-linux-
gnueabihf/bin/ld: cannot find /usr/lib/arm-linux-gnueabihf/libpthread_
nonshared.a
collect2: erreur: ld a retourné 1 code d'état d'exécution
Makefile:53: recipe for target 'all' failed
make: *** [all] Error 1
```

Cela vient du fait qu'un préfixe `sysroot` est configuré dans le compilateur [14].
On peut le vérifier avec la commande suivante :

Terminal

```
$ /home/yann/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-
raspbian/bin/arm-linux-gnueabihf-gcc -print-sysroot
/home/yann/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-
raspbian/bin/../arm-linux-gnueabihf/libc
```

Les fichiers des bibliothèques système seront alors recherchés dans ce répertoire.
On peut s'en affranchir en modifiant les fichiers `./usr/lib/arm-linux-gnueabihf/libpthread.so` et `./usr/lib/arm-linux-gnueabihf/libc.so`. Il suffit d'enlever le chemin absolu qui se trouve dans les fichiers de bibliothèque.

Les fichiers deviennent les suivants :

Fichier

```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
OUTPUT_FORMAT(elf32-littlearm)
/*GROUP ( /lib/arm-linux-gnueabihf/libc.so.6 /usr/lib/arm-linux-gnueabihf/
libc_nonshared.a  AS_NEEDED ( /lib/arm-linux-gnueabihf/ld-linux-armhf.so.3
) )*/
GROUP ( libc.so.6 libc_nonshared.a  AS_NEEDED ( ld-linux-armhf.so.3 ) )
```

et

Fichier

```
/* GNU ld script
   Use the shared library, but some functions are only in
   the static library, so try that secondarily.  */
OUTPUT_FORMAT(elf32-littlearm)
/*GROUP ( /lib/arm-linux-gnueabihf/libpthread.so.0 /usr/lib/arm-linux-
gnueabihf/libpthread_nonshared.a )*/
GROUP ( libpthread.so.0 libpthread_nonshared.a )
```

Maintenant, la compilation se déroule comme prévu et nous obtenons un fichier exécutable pour l'architecture ARM.

Terminal

```
$ make
/home/yann/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-
raspbian/bin/arm-linux-gnueabihf-gcc -O0 -g -Wall  -L/home/yann/raspberry/
lib/arm-linux-gnueabihf -L/home/yann/raspberry/usr/lib/arm-linux-gnueabihf
-L/home/yann/raspberry/lib -L/home/yann/raspberry/usr/lib -pthread -I/
home/yann/raspberry/usr/include/gtk-3.0 -I/home/yann/raspberry/usr/
include/pango-1.0 -I/home/yann/raspberry/usr/include/gio-unix-2.0/ -I/
home/yann/raspberry/usr/include/atk-1.0 -I/home/yann/raspberry/usr/
include/cairo -I/home/yann/raspberry/usr/include/gdk-pixbuf-2.0 -I/
home/yann/raspberry/usr/include/freetype2 -I/home/yann/raspberry/usr/
include/glib-2.0 -I/home/yann/raspberry/usr/lib/arm-linux-gnueabihf/glib-
2.0/include -I/home/yann/raspberry/usr/include/pixman-1 -I/home/yann/
raspberry/usr/include/libpng12 -Xlinker -rpath-link=/home/yann/raspberry/
lib/arm-linux-gnueabihf -Xlinker -rpath-link=/home/yann/raspberry/usr/lib/
arm-linux-gnueabihf -lgtk-3 -lgdk-3 -latk-1.0 -lgio-2.0 -lpangocairo-1.0
-lgdk_pixbuf-2.0 -lcairo-gobject -lpango-1.0 -lcairo -lgobject-2.0
-lglib-2.0 -lrt -lpcre gtktest.c -o gtktest
$ file
gtktest    gtktest.c  Makefile
$ file gtktest
gtktest: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV),
dynamically linked (uses shared libs), for GNU/Linux 2.6.26, BuildID[sha1]
=d69d1a4fd237277a4be5febb6298e0c4eb61e5f7, not stripped
```

Sur une distribution Ubuntu, j'ai eu une autre erreur :

Terminal

```
/home/yann/Bureau/tp1_rpi/tools/arm-bcm2708/gcc-linaro-arm-linux-
gnueabihf-raspbian/bin/../lib/gcc/arm-linux-gnueabihf/4.8.3/../../../../
arm-linux-gnueabihf/bin/ld: warning: libgraphite2.so.2.0.0, needed by /
home/yann/Bureau/tp1_rpi/usr/lib/arm-linux-gnueabihf/libharfbuzz.so.0, not
found (try using -rpath or -rpath-link)
```


Le compilateur ne semble pas trouver la bibliothèque `graphite2`, alors qu'elle est bien présente dans l'arborescence. On commence donc par ajouter l'utilisation de la bibliothèque dans le `Makefile`, comme pour `rt` et `pcre` :

Fichier

```
LIBRARY += graphite2
```

Cependant, la compilation ne fonctionne toujours pas et l'erreur est toujours présente. On peut pallier ce problème rapidement en créant le lien symbolique suivant :

Terminal

```
$ ln -s libgraphite2.so.2.0.0 libgraphite2.so
```

Une autre manière de s'en sortir est d'installer le paquet `libgraphite2-dev` sur le Raspberry Pi et de synchroniser à nouveau les répertoires `lib` et `usr` du PC de compilation :

Terminal

```
pi@raspberrypi /usr $ sudo apt-get install libgraphite2-dev
```

En effet, le lien symbolique est présent dans le paquet de développement :

Terminal

```
pi@raspberrypi /usr $ dpkg -L libgraphite2-dev
[snip]
/usr/lib/libgraphite2.so
```

Pour le tester, on le transfère sur notre Raspberry Pi à l'aide de `scp`. Puis, on lance notre application (voir figure 2*).

Si vous ne disposez pas de serveur graphique sur votre Raspberry, vous pouvez utiliser une connexion SSH qui redirigera les programmes graphiques et vous obtiendrez la figure 3* :

Terminal

```
$ ssh -X pi@192.168.1.67
pi@192.168.1.67's password:
```

Voilà pour cette partie. Cependant, si vous ne voulez pas faire de mise à jour de l'architecture présente sur votre machine de bureau à chaque mise à jour du système de votre Raspberry Pi, il est possible d'utiliser une autre solution : lancer la compilation sur le Raspberry Pi, mais envoyer par le réseau sur une/des machine(s) plus puissante(s) les calculs de cette compilation. C'est ce que nous allons faire avec `distcc`. C'est une façon très élégante de réaliser la compilation croisée.

4. DISTCC, POUR DISTRIBUER VOS COMPILATIONS CROISÉES

Distcc [15] est un programme qui permet de distribuer la compilation de code (Objective) C, (Objective) C++ sur plusieurs machines d'un réseau. Malgré les transferts de données sur le réseau, la plupart du temps cela permet de réduire considérablement les temps de compilation.

Bien que **distcc** existe dans les dépôts des différentes distributions Linux, nous allons le recompiler. En effet, la version 3.1 disponible souffre d'une limitation [16] : la variable **DISTCC_IO_TIMEOUT** est fixée dans le code à 300 secondes, il n'est donc pas rare d'obtenir l'erreur suivante lorsque le transfert de données dure plus de 300 s :

Fichier

```
(dcc_select_for_write) ERROR: IO timeout
```

Alors **distcc** considère que le serveur n'est pas fonctionnel et lance la compilation en local. Nous allons donc utiliser la version 3.2, qui permet de fixer cette valeur à travers une variable d'environnement.

Il faut installer **distcc** sur toutes les machines : le Raspberry Pi qui lance la compilation et toutes les machines faisant office de serveurs de compilation. Je n'utiliserai qu'une seule machine pour cet exemple.

4.1 Sur le Raspberry Pi

La bibliothèque **libiberty**, nécessaire à la compilation de **distcc**, n'est pas présente dans les dépôts Raspbian, nous allons donc la recompiler rapidement :

Terminal

```
pi@raspberrypi ~ $ sudo apt-get install libpopt-dev
pi@raspberrypi ~ $ wget https://toolbox-of-eric.googlecode.com/files/libiberty.tar.gz
pi@raspberrypi ~ $ tar xzf libiberty.tar.gz
pi@raspberrypi ~ $ cd libiberty
pi@raspberrypi ~ $ ./configure && make && sudo make install
```

Si l'on désire suivre l'activité de **distcc** avec un moniteur graphique, il est nécessaire d'installer GTK+-2.0 :

Terminal

```
pi@raspberrypi ~/$ sudo apt-get install libgtk2.0-dev
```

On récupère les dernières sources de **distcc** et on installe le programme comme suit :

Terminal

```
pi@raspberrypi ~ $ svn checkout http://distcc.googlecode.com/svn/trunk/ distcc-read-
only
pi@raspberrypi ~/distcc-read-only $ cd distcc-read-only
pi@raspberrypi ~/distcc-read-only $ ./autogen.sh
pi@raspberrypi ~/distcc-read-only $ ./configure --with-gtk && make && sudo make install
```

NOTE

Omettez l'option **--with-gtk** pour compiler sans le moniteur graphique.

Ensuite, on crée des liens symboliques pour être sûr que lorsque **gcc** ou **g++** seront utilisés sur le Raspberry Pi, ils le seront à travers **distcc**. Les liens sont créés dans le répertoire **/usr/local/bin**, puis on modifie la variable **PATH**, afin de prendre en priorité la version **distcc** des compilateurs :

Terminal

```
pi@raspberrypi ~/distcc-read-only $ which gcc
/usr/bin/gcc
pi@raspberrypi ~/distcc-read-only $ which distcc
/usr/local/bin/distcc
pi@raspberrypi ~/distcc-read-only $ sudo ln -s /usr/local/bin/distcc /usr/local/bin/gcc
pi@raspberrypi ~/distcc-read-only $ sudo ln -s /usr/local/bin/distcc /usr/local/bin/cc
pi@raspberrypi ~/distcc-read-only $ sudo ln -s /usr/local/bin/distcc /usr/local/bin/g++
pi@raspberrypi ~/distcc-read-only $ sudo ln -s /usr/local/bin/distcc /usr/local/bin/c++
pi@raspberrypi ~/distcc-read-only $ sudo ln -s /usr/local/bin/distcc /usr/local/bin/cpp
pi@raspberrypi ~/distcc-read-only $ export PATH=/usr/local/bin:$PATH
pi@raspberrypi ~/distcc-read-only $ which gcc
/usr/local/bin/gcc
```

Vous pouvez aussi placer la commande **export** dans votre fichier **.bashrc** pour rendre cette opération permanente. On y placera aussi les configurations suivantes :

Fichier

```
###
# distcc section
###
#export usr/local/bin for distcc
export PATH=/usr/local/bin:$PATH

# The remote machines that will build things for you. Don't put the ip of the Pi unless
# you want the Pi to take part to the build process.
# The syntax is : "IP_ADDRESS/NUMBER_OF_JOBS IP_ADDRESS/NUMBER_OF_JOBS" etc...
# The documentation states that your should set the number of jobs per machine to
# its number of processors. I advise you to set it to twice as much. See why in the test
paragraph.
# For example:
export DISTCC_HOSTS="192.168.1.99/2"

# When a job fails, distcc backs off the machine that failed for some time.
# We want distcc to retry immediately
export DISTCC_BACKOFF_PERIOD=0

# Time, in seconds, before distcc throws a DISTCC_IO_TIMEOUT error and tries to build the file
# locally ( default hardcoded to 300 in version prior to 3.2 )
export DISTCC_IO_TIMEOUT=3000
# Don't try to build the file locally when a remote job failed
#expor DISTCC_SKIP_LOCAL_RETRY=1
```

La directive **DISTCC_HOSTS** permet de configurer les différents serveurs de compilation, ainsi que le nombre de compilations en parallèle sur chaque machine. Dans mon cas, j'utiliserai une seule machine avec 2 processus, mais il suffit d'ajouter les adresses IP d'autres machines munies des outils **distcc** pour multiplier la puissance de compilation.

C'est terminé pour la partie client, passons à la partie serveur sur votre PC de bureau ou portable puissant.

4.2 Sur le PC

Sur ma distribution Arch Linux, j'utilise le dépôt AUR pour installer la dernière version de **distcc**.

Terminal

```
$ wget https://aur.ArchLinux.org/packages/di/distcc-svn/distcc-svn.tar.gz
$ tar xzf distcc-svn.tar.gz
$ cd distcc-svn
$ makepkg -s
$ sudo pacman -U distcc-svn-r791-1-x86_64.pkg.tar.xz
```

De la même manière que sur le Raspberry Pi, il est nécessaire de créer des liens symboliques afin que `distcc` puisse utiliser les outils de compilation Linaro pour l'architecture ARM :

Terminal
```
$ cd raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-raspbian/
bin/
$ ln -s arm-linux-gnueabihf-gcc gcc
$ ln -s arm-linux-gnueabihf-gcc cc
$ ln -s arm-linux-gnueabihf-c++ c++
$ ln -s arm-linux-gnueabihf-g++ g++
$ ln -s arm-linux-gnueabihf-cpp cpp
```

Ensuite, on exporte une nouvelle variable `PATH` afin d'utiliser par défaut les versions de compilation croisée des compilateurs :

Terminal
```
$ export PATH=$HOME/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-
gnueabihf-raspbian/bin:$PATH
$ which gcc
/home/yann/raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-gnueabihf-
raspbian/bin/gcc
```

Vous pouvez aussi ajouter cette commande dans votre `.bashrc`.

Tout est prêt, on exécute le serveur `distccd` sur le/les PC serveur(s) de compilation :

Terminal
```
$ distccd --daemon --jobs 2 --allow 192.168.1.63 --verbose --log-stderr
--no-detachdistccd[1083] (main) chdir to /tmp
distccd[1083] (dcc_setup_daemon_path) daemon's PATH is /usr/local/sbin:/
usr/local/bin:/usr/bin:/usr/bin/vendor_perl:/usr/bin/core_perl
distccd[1083] (dcc_listen_by_addr) listening on 0.0.0.0:3632
distccd[1083] (dcc_defer_accept) TCP_DEFER_ACCEPT turned on
distccd[1083] (dcc_standalone_server) 2 CPUs online on this server
distccd[1083] (dcc_standalone_server) allowing up to 2 active jobs
distccd[1083] (dcc_standalone_server) not detaching
distccd[1083] (dcc_new_pgrp) already a process group leader
distccd[1083] (dcc_log_daemon_started) preforking daemon started (3.2rc1
x86_64-unknown-linux-gnu, built Apr 11 2014 16:38:28)
distccd[1083] (dcc_create_kids) up to 1 children
distccd[1083] (dcc_create_kids) up to 2 children
```

L'option `--daemon` permet de mettre le serveur en attente de connexion. L'option `--jobs 2` fixe le nombre maximum de tâches en parallèle. L'option `--allow` permet d'autoriser notre Raspberry Pi à envoyer ses fichiers à compiler à notre serveur. `--verbose` passe en mode verbeux, `--log-stderr` permet d'afficher les erreurs sur la console et `--no-detach` permet de ne pas détacher le processus du terminal et ainsi, de suivre l'évolution de la compilation et les erreurs éventuelles. D'autres options sont disponibles dans la page de manuel.

4.3 La compilation

Tout est prêt, on peut maintenant tester notre nouvel outil de compilation distribuée. Sur le Raspberry Pi ,on lance la génération d'un émulateur SNES pour notre Raspberry Pi :

Terminal
```
pi@raspberrypi ~$ time make -j2
g++ -I/usr/include -I/opt/vc/include -I/opt/vc/include/interface/vcos/
pthreads -I/opt/vc/include/interface/vmcs_host/linux  -c -D_ZAURUS -O3
-march=armv6 -mfpu=vfp -mfloat-abi=hard -ffast-math -fstrict-aliasing
-fomit-frame-pointer  -I/usr/include -I/usr/include/SDL -I. -Iunzip
-Isdl -I/usr/include/glib-2.0 -I/usr/lib/arm-linux-gnueabihf/glib-2.0/
include -D__linux -DZLIB -DVAR_CYCLES -DCPU_SHUTDOWN -DSPC700_SHUTDOWN
-fpermissive -Wno-write-strings  -DSPC700_C  -DUNZIP_SUPPORT  -DNO_INLINE_
SET_GET cpuops.cpp -o cpuops.o
[snip]
gcc -I/usr/include -I/opt/vc/include -I/opt/vc/include/interface/vcos/
pthreads -I/opt/vc/include/interface/vmcs_host/linux  -o snes9x.gui unix/
frontend.o  -L/usr/lib/arm-linux-gnueabihf -L/opt/vc/lib -lbcm_host
-lGLESv2 -lEGL -lglib-2.0 -lSDL -lstdc++ -lz -lpthread

real 0m26.590s
user 0m10.180s
sys 0m2.960s
```

À titre de comparaison, on peut réaliser la même compilation en local sur le Raspberry Pi :

Terminal
```
pi@raspberrypi ~$ time make
[snip]
real 5m7.914s
user 5m3.300s
sys 0m3.740s
```

La machine utilisée comme serveur de compilation est un « vieux » Acer TravelMate 6292 (Core 2 Duo T7300). Dans ce cas précis, cela permet de compiler 12 fois plus rapidement. On comprend ici l'intérêt de distribuer la charge de compilation sur différentes machines plus puissantes à travers le réseau.

La figure 4* montre la compilation d'un noyau à l'aide de `distcc`. La compilation est lancée depuis le Raspberry Pi (grand terminal) et les calculs sont effectués sur la machine serveur `distcc` (petit terminal). On peut ainsi suivre la compilation en temps réel.

Il est aussi possible de suivre la compilation à l'aide des moniteurs présents dans le projet `distcc`. Ainsi, les programmes `distccmon-gnome` et `distccmon-text` permettent de suivre respectivement en mode graphique et en mode texte, l'évolution de la compilation de votre projet. La figure 5* décrit le suivi de la génération de la bibliothèque SDL 2 à l'aide de ces 2 applicatifs.

Dans le cas de l'utilisation de l'outil `make`, il n'est pas nécessaire de modifier votre `Makefile`, ou celui généré par le script `configure` du projet à compiler. La création des liens symboliques suffit pour l'utilisation de `distcc`. Cependant, si vous utilisez `cmake` pour générer les `Makefiles` (de nombreux projets utilisent maintenant cet outil), il faudra l'invoquer en lui indiquant que vous allez utiliser `distcc` pour compiler vos sources :

Terminal
```
$ cd ~/project
~/project$ make clean
~/project$ rm CMakeCache.txt
~/project$ CC="distcc gcc" CXX="distcc g++" cmake .
~/project$ make -j2
```

distcc vous permettra donc d'accélérer la génération de nombreuses applications.

4.4 Distcc avec une version spécifique de compilateur

Gcc est en version 4.6 par défaut sur la distribution Raspbian. La compilateur 4.7 est aussi disponible, mais non installé par défaut. Cependant, il est possible que certaines applications nécessitent une version 4.7 minimum, car leurs sources utilisent du code C++11 (le projet EmulationStation en est un exemple [17]). Vous installez alors ce nouveau compilateur à l'aide d'`apt-get`. Mais si vous tentez d'utiliser `distcc` pour la compilation distante, ça ne fonctionnera pas. En effet, ce dernier a été créé à partir de la version 4.6 de `gcc` et ne reconnaîtra pas les morceaux de code C++11 supportés par les versions ultérieures.

Il vous faudra alors recompiler `distcc` avec la nouvelle version du compilateur. Pour cela, on fait pointer les liens symboliques de `gcc` vers le nouveau compilateur :

Terminal
```
pi@raspberrypi ~ $ ls -al /usr/bin/gcc
lrwxrwxrwx 1 root root 7 Aug 20 01:08 /usr/bin/gcc -> gcc-4.7
pi@raspberrypi ~ $ ls -al /usr/bin/g++
lrwxrwxrwx 1 root root 7 Aug 20 01:08 /usr/bin/g++ -> g++-4.7
```

Puis, on recompile et on installe `distcc`. On vérifie ensuite que cette nouvelle version utilise bien la version 4.7 de `gcc` :

Terminal
```
pi@raspberrypi ~ $ distcc -v
[snip]
Thread model: posix
gcc version 4.7.2 (Debian 4.7.2-5+rpi1)
```

Il faut aussi que le compilateur croisé de votre machine de compilation ait une version supérieure ou égale à celle utilisée sur le Raspberry Pi. Ce qui est le cas avec les outils Linaro :

Terminal
```
[yann@archery ~]$ raspberry/tools/arm-bcm2708/gcc-linaro-arm-linux-
gnueabihf-raspbian/bin/gcc -v
[snip]
Modèle de thread: posix
gcc version 4.8.3 20140106 (prerelease) (crosstool-NG
linaro-1.13.1-4.8-2014.01 - Linaro GCC 2013.11)
```

Vous pouvez maintenant générer votre application à l'aide de `distcc` :

Terminal
```
$ rm CMakeCache.txt
$ CXX="distcc g++" cmake .
$ make -j2
```

CONCLUSION

Voici la fin de ce petit tour d'horizon de la compilation croisée pour votre Raspberry Pi. En résumé, pour les projets simples, l'utilisation seule des outils Linaro peut suffire, mais pour des projets complexes utilisant de nombreuses bibliothèques, l'utilisation de `distcc` s'avère élégante et très efficace. Bonne cross-compilation ;) ■

NOTE

* Les figures accompagnant cet article sont visibles sur le blog de GNU/Linux Magazine : **http://www.gnulinuxmag.com/**

RÉFÉRENCES

[1] Définition de la compilation croisée : **http://en.wikipedia.org/wiki/Cross_compiler**

[2] Présentation de l'association Linaro : **http://fr.wikipedia.org/wiki/Linaro**

[3] Mise en œuvre des outils Crosstool-NG : **http://www.chicoree.fr/w/Compilation_croisée_facile_pour_Raspberry_Pi**

[4] Construction des outils de compilation Linaro : **http://elinux.org/RPi_Linaro_GCC_Compilation**

[5] Utiliser des applications 32 bits sur un système Arch Linux 64 bits : **https://wiki.ArchLinux.fr/Utiliser_des_applications_32bits_avec_Arch64**

[6] Site officiel du module arcade Xin-Mo : **http://www.xin-mo.com/?page_id=34**

[7] Description complète en français du bug du module Xim-Mo : **http://www.morere.eu/spip.php?article176**

[8] Correctif du module noyau du module Xin-Mo : **http://ithink.ch/blog/2013/09/08/patching_the_linux_kernel_to_install_the_xin-mo_dual_arcade_driver_on_a_raspberry_pi.html**

[9] Réalisation de la mise à jour du firmware du Raspberry Pi : **http://elinux.org/RPi_Kernel_Compilation#Get_the_firmware**

[10] Compilation croisée des bibliothèques GTK pour ARM : **http://fatalfeel.blogspot.fr/2013/09/static-cross-compile-gtk-2166gtk.html**

[11] Programme de test en GTK3 : **http://zetcode.com/tutorials/gtktutorial/firstprograms/**

[12] Compilation croisée d'applications GTK pour Raspberry Pi : **http://hertaville.com/2013/07/19/cross-compiling-gtk-applications-for-the-raspberry-pi/**

[13] Description des fonctions de texte pour le programme Make : **http://www.gnu.org/software/make/manual/make.html#Text-Functions**

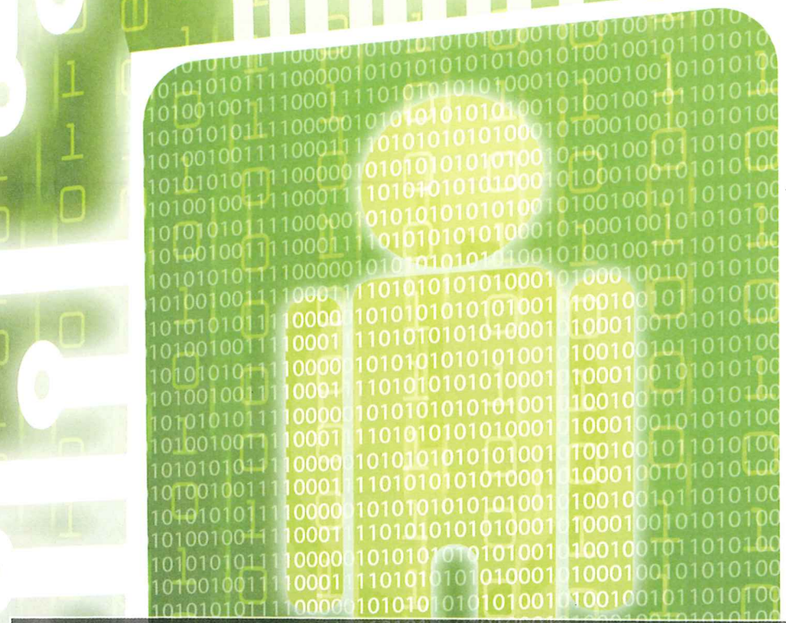
[14] Description et solution du bug de compilation croisée d'applications graphiques : **http://stackoverflow.com/questions/14207189/crosstools-ng-cant-find-pthread-so**

[15] Site officiel de Distcc : **https://code.google.com/p/distcc/**

[16] Installation et mise en œuvre de Distcc : **http://jeremy-nicola.info/portfolio-item/cross-compilation-distributed-compilation-for-the-raspberry-pi/**

[17] Dépôt GitHub d'EmulationStation : **https://github.com/Aloshi/EmulationStation**

3 DISTRIBUTIONS ET OS



ÉCRAN SPI POUR RASPBERRY PI

Pierre Ficheux

La carte Raspberry Pi (RPi) fut initialement conçue comme un « PC bon marché ». Elle dispose donc d'un contrôleur graphique performant permettant la connexion à un écran HDMI externe. Dans cette configuration, il est possible d'utiliser un mode accéléré (OpenGL) compatible avec des bibliothèques graphiques telles que Qt 5. Cependant, de nombreuses applications embarquées utilisent un écran local (fixé à la carte) souvent tactile et permettant d'utiliser une application métier. Dans cet article, nous allons décrire la mise en place et l'utilisation sous Qt et DirectFB d'un écran 320x240 2,8 pouces, connecté au bus SPI de la carte RPi.

L'écran présenté est commercialisé par la célèbre société Adafruit [1-4]. Sa taille est de 2,8 pouces et sa résolution de 320 sur 240 pixels. Malgré sa petite taille et son prix modique (environ $35 chez Adafruit), il est plutôt de bonne qualité. L'écran est tactile et utilise une technologie *résistive*, moins agréable que la technologie *capacitive* de nos téléphones modernes, mais également moins onéreuse. Lors de l'utilisation de l'écran, il faudra donc exercer une pression avec le doigt, ou mieux un stylet, pour que l'action soit bien prise en compte. Une version avec écran capacitif est désormais disponible même si elle est un peu plus chère ($45).

L'écran est piloté par le bus SPI (pour *Serial Peripheral Interface*), qui est assez peu performant mais simple à mettre en place. Il ne faut donc pas s'attendre à de l'affichage accéléré ou 3D comme avec le contrôleur graphique interne de la RPi. Cependant, ce petit écran peut rendre de bons services dans le cas de maquettes d'applications graphiques simples ou d'enseignements.

1. MONTAGE DU KIT

Le kit fourni contient l'écran, un connecteur HE10 26 points adapté à la RPi modèle B (pour la connexion directe à la carte), ainsi qu'un autre connecteur HE10 permettant d'utiliser une nappe pour la connexion à la carte (voir figure 1). L'écran est également compatible avec la nouvelle carte RPi B+, puisque les 26 premiers signaux du connecteur 40 points sont identiques à ceux de la version B.

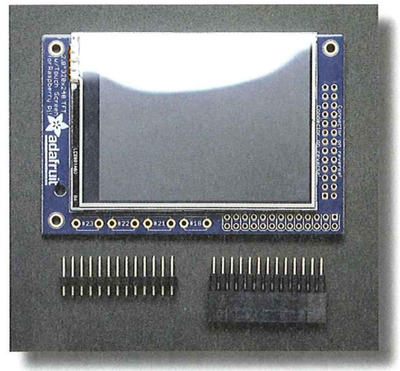


Fig. 1 : Kit de l'écran SPI

Comme à l'accoutumée, la section *learn* du site Adafruit [5-7] explique très clairement le montage du kit, puis la configuration logicielle pour les utilisateurs « classiques ». Selon le site Adafruit, l'écran est désormais fourni déjà monté, prêt à l'utilisation sur la RPi, *As of 8/15/2014 it comes fully assembled and ready to plug into your Pi!*.

Fig. 2 : Écran monté et installé sur la carte

Il faut noter que le connecteur vertical à droite de l'écran reprend les mêmes signaux que ceux du connecteur horizontal en bas à droite. Cela permet de conserver certains signaux non utilisés par l'écran, comme par exemple la console de la carte (TxD, RxD, 5V, masse), ou bien d'autres GPIO. Dans notre cas, nous avons simplement installé le connecteur horizontal (à connecter à la Rpi), puis horizontalement les 4 signaux permettant l'utilisation de la console comme le montre la figure 2. De bas en haut, nous avons donc deux fois le signal 5V, puis la masse, puis le signal TxD et enfin RxD.

On peut également ajouter des boutons (GPIO) situés en bas de l'écran, mais nous n'avons pas testé cette fonctionnalité. Pour la console, nous utilisons une fois encore le câble USB/Série proposé par Adafruit. Ce dernier permet l'utilisation de la console, ainsi que l'alimentation de la carte et de l'écran. La figure 3 indique la connectique à utiliser. Nous attirons l'attention sur le fait qu'une inversion des câbles rouge et noir (alimentation 5V et masse) aura pour effet de détruire la carte !

Fig. 3 : Branchement du câble USB/Série

Il est bien entendu possible d'utiliser la RPi avec un écran externe (HDMI ou composite), un clavier et une souris, mais la configuration est plus complexe sachant que le but final est d'utiliser l'écran SPI.

2. PREMIER TEST DE L'ÉCRAN

Il est conseillé d'effectuer le premier test avec une distribution classique comme la Raspbian. En effet, la documentation fournie par Adafruit permet la mise à jour du noyau Linux afin d'ajouter les pilotes de l'écran. Le site propose également des images binaires d'une version de la Raspbian intégrant directement les pilotes. Si l'on utilise la console comme indiqué précédemment, on obtient l'affichage suivant sur l'émulateur de terminal (le nom d'utilisateur est `pi` et le mot de passe `raspberry`) :

Terminal
```
...
Debian GNU/Linux wheezy/sid raspberrypi ttyAMA0
raspberrypi login: pi
Password: raspberry
Last login: Mon Feb  4 16:54:02 UTC 2013 on ttyAMA0
```

La documentation est disponible sur **https://learn.adafruit.com/adafruit-pitft-28-inch-resistive-touchscreen-display-raspberry-pi/software-installation**.

En résumé, il suffit de télécharger les paquet `.deb`, de les installer et de redémarrer la carte.

Terminal
```
$ wget http://adafruit-download.s3.amazonaws.com/libraspberrypi-bin-adafruit.deb
$ wget http://adafruit-download.s3.amazonaws.com/libraspberrypi-dev-adafruit.deb
$ wget http://adafruit-download.s3.amazonaws.com/libraspberrypi-doc-adafruit.deb
$ wget http://adafruit-download.s3.amazonaws.com/libraspberrypi0-adafruit.deb
$ wget http://adafruit-download.s3.amazonaws.com/raspberrypi-bootloader-adafruit-20140724-1.deb
$ sudo dpkg -i -B *.deb
```

Après redémarrage de la carte, on peut effectuer un test sous X11. Le principal est de détecter la carte sur le bus SPI, puis le pilote framebuffer spécifique correspondant à l'écran :

Terminal
```
$ sudo modprobe spi-bcm2708
$ sudo modprobe fbtft_device name=adafruitts rotate=90
$ export FRAMEBUFFER=/dev/fb1
$ startx
```

Nous remarquons que le framebuffer de l'écran correspond à `/dev/fb1`, car `/dev/fb0` est utilisé par le contrôleur graphique intégré. La suite est détaillée dans la documentation Adafruit et il n'est pas dans nos habitudes de paraphraser le travail des autres :-)

Dans la suite de l'article, nous étudierons donc l'ajout des pilotes de l'écran au noyau Linux utilisé pour la RPi, ainsi que l'intégration de l'ensemble dans une distribution créée grâce à l'outil Buildroot. Ce dernier nous permettra de tester aisément des bibliothèques graphiques comme DirectFB ou Qt.

3. UTILISATION DE L'ÉCRAN AVEC BUILDROOT

Buildroot [8] est un outil très répandu permettant de créer des distributions réduites optimisées pour l'utilisation d'un calculateur dans un environnement « embarqué ». Cet outil a fait l'objet de divers articles dans plusieurs numéros du magazine *Open Silicium* [9] [10]. Dans un premier temps, il est nécessaire de créer un « patch » correspondant aux modifications liées à l'intégration de l'écran. Une fois le patch intégré à Buildroot, nous pourrons ensuite sélectionner les paquets nécessaires à l'exploitation de l'écran (DirectFB, Qt,

etc.). La configuration Buildroot proposée permettra de créer *automatiquement* une image de carte SD pour la RPi. La configuration présentée utilise la version 2014.08 de Buildroot, mais devrait être compatible avec les versions plus anciennes sous réserve d'une légère adaptation.

Pour utiliser la configuration fournie et construire la distribution, il suffit d'exécuter la séquence de commandes qui suit :

Terminal
```
$ git clone git://git.buildroot.net/buildroot buildroot
$ cd buildroot
$ git checkout -b 2014_08 2014.08
$ git apply --whitespace=nowarn <path>/0001-Added-Adafruit-TS-support-to-
BR-2014.08.patch
$ make raspberrypi_qt4_ts_defconfig
$ make
```

À l'issue de la (longue) compilation, on devrait obtenir les fichiers suivants dans le répertoire `output/images` :

Terminal
```
$ cd output/images/
$ ls -l
total 321180
-rw-rw-r-- 1 pierre pierre 102318080 sept.  2 13:17 rootfs.ext2
lrwxrwxrwx 1 pierre pierre        11 sept.  2 13:17 rootfs.ext3 -> rootfs.ext2
-rw-rw-r-- 1 pierre pierre  83548160 sept.  2 13:17 rootfs.tar
-rw-rw-r-- 1 pierre pierre  33181488 sept.  2 13:17 rootfs.tar.gz
drwxr-xr-x 2 pierre pierre      4096 sept.  2 12:52 rpi-firmware
-rw-rw-r-- 1 pierre pierre 130023424 sept.  2 13:17 rpi-sdcard.img
-rwxrwxr-x 1 pierre pierre   3132136 sept.  2 13:17 zImage
```

Le fichier `rpi-sdcard.img` correspond à l'image de la carte SD. On peut donc créer une carte utilisable sur la RPi par la commande ci-dessous, en supposant que la carte SD est vue sur le périphérique `/dev/sdb` (attention aux erreurs sur le nom du périphérique, car elles sont **fatales pour les disques durs**) :

Terminal
```
$ sudo dd if=rpi-sdcard.img of=/dev/sdb
```

Si l'on démarre la RPi avec la carte SD ainsi créée, on obtient les messages suivants sur la console USB/Série :

Terminal
```
Uncompressing Linux... done, booting the kernel.
[    0.000000] Booting Linux on physical CPU 0
[    0.000000] Initializing cgroup subsys cpu
[    0.000000] Linux version 3.6.11-adafruit_ts+git (pierre@XPS-pf) (gcc version 4.6.3
(Sourcery CodeBench Lite 2012.03-57) ) #1 PREEMPT Mon Aug 25 15:16:02 CES T 2014
...
[    4.029741] Adafruit Industries' Raspberry Pi PWM driver v1.0
[    4.037480] fbtft_device:  SPI devices registered:
[    4.042291] fbtft_device:       spidev spi0.0 500kHz 8 bits mode=0x00
[    4.060685] fbtft_device:       spidev spi0.1 500kHz 8 bits mode=0x00
[    4.074373] fbtft_device:  'fb' Platform devices registered:
[    4.086250] fbtft_device:       bcm2708_fb id=-1 pdata? no
[    4.092123] bcm2708_i2c bcm2708_i2c.1: BSC1 Controller at 0x20804000 (irq 79)
 (baudrate 100k)
[    4.166299] fbtft_device: Deleting spi0.1 (spi0.1)
[    4.175888] stmpe-spi spi0.1: stmpe610 detected, chip id: 0x811
[    4.193286] fbtft_device: Deleting spi0.0 (spi0.0)
[    4.203595] fbtft_device:  GPIOS used by 'adafruitts':
[    4.215279] fbtft_device:    'dc' = GPIO25
[    4.225458] fbtft_device:  SPI devices registered:
[    4.237491] fbtft_device:       stmpe610 spi0.1 500kHz 8 bits mode=0x00
[    4.244028] fbtft_device:       fb_ili9340 spi0.0 16000kHz 8 bits mode=0x00
...
Welcome to Buildroot (Qt/TS)
rpi login:
```

On obtient également l'affichage du même invité de connexion sur l'écran graphique. Bien entendu, la connexion par ce biais nécessite de brancher un clavier USB à la carte. Après ce premier test, la suite de cette section s'attachera à décrire comment cette configuration Buildroot est construite.

3.1 Adaptation du noyau

La documentation Adafruit fait référence à un ensemble de patches permettant l'intégration des pilotes de l'écran au noyau de la RPi. Ces patches sont adaptés à la version 3.6.11. Après extraction de l'archive [11], on obtient les fichiers et répertoires suivants dans le dossier `adafruit-ts` :

Fichier

```
adafruit-ts/
|-- 0001-stmpe-ts-Don-t-report-empty-packets.patch
|-- 0002-stmpe-ts-Simulate-button-press-when-touch-happens.patch
|-- 0003-mach-bcm2708-Reserve-64-IRQs-for-peripherals.patch
|-- 0004-video-Add-fbtft-support.patch
|-- 0005-bcm2708-Add-rpi-pwm-driver-to-machine-definition.patch
|-- 0006-rpi-pwm-Add-driver-for-Raspberry-Pi-PWM-device.patch
|-- 0007-rpi-power-switch-Add-power-switch-module.patch
|-- config
`-- fbtft
    |-- 0001-fbtft_device-Allow-multiple-SPI-devices-to-be-insert.patch
    |-- 0002-fbtft-ili9340-Add-support-for-new-TFT.patch
    |-- 0003-fbtft-Add-support-for-Adafruit-touchscreen-module.patch
    `-- 0004-fbtft-Set-pullup-on-STMPE-IRQ-line.patch
```

La documentation n'est pas vraiment très explicite concernant la création du noyau modifié, car cela concerne surtout les utilisateurs *avancés*. De plus, nous avons eu quelques soucis avec les patches du répertoire `fbtft` et il nous a semblé plus judicieux de nous baser sur le dépôt Git du projet FBTFT [12] adapté à l'écran Adafruit (soit **https://github.com/xobs/adafruit-rpi-fbtft**).

Il faut tout d'abord réaliser un clone des sources du noyau Linux pour la RPi et se placer sur la version 3.6.11 :

Terminal

```
$ git clone https://github.com/raspberrypi/linux.git
$ cd linux
$ git checkout -b 3.6.11 2a8d45ec0883e3cbdce920855b3461ac77308a5f
```

On peut alors ajouter les pilotes FBTFT :

Terminal

```
$ git clone  https://github.com/xobs/adafruit-rpi-fbtft  drivers/video/
fbtft
$ git add drivers/video/fbtft/
```

On peut ensuite appliquer les patches fournis par Adafruit (hormis ceux pour FBTFT) :

Terminal

```
$ patch -p1 < ../adafruit-ts/0001-stmpe-ts-Don-t-report-empty-packets.patch
...
$ patch -p1 < ../adafruit-ts/0007-rpi-power-switch-Add-power-switch-module.patch
$ git add drivers/misc drivers/power
```


Enfin, on peut créer le patch à appliquer au noyau 3.6.11 :

Terminal

```
$ git commit -a -m "Added Adafruit TS support"
$ git format-patch 2a8d45ec0883e3cbdce920855b3461ac77308a5f
```

On obtient alors le fichier `0001-Added-Adafruit-TS-support.patch` que l'on peut intégrer à la configuration Buildroot.

3.2 Configuration de Buildroot

Le but est de créer une configuration permettant de démarrer la carte et d'initialiser automatiquement l'écran graphique :

- Chargement du module `spi-bcm2708` (SPI),
- Chargement du pilote `fbtft_device` (framebuffer),
- Calibration automatique avec `ts_calibrate` lors de la première utilisation,
- Affectation des variables d'environnement nécessaires à DirectFB ou Qt,
- Affichage de l'invité de connexion sur l'écran graphique.

Comme nous l'avons vu lors du premier test, la configuration est stockée dans le fichier `configs/raspberrypi_qt4_ts_defconfig`.

La configuration de la partie noyau est bien entendu l'une des plus importantes. On l'obtient en utilisant la commande `make menuconfig`, puis en sélectionnant la rubrique ***Kernel***. L'écran suivant permet de sélectionner la version de noyau à utiliser à partir du dépôt Git, mais également les patches à appliquer ainsi que la configuration du noyau. Ces fichiers sont localisés dans le répertoire `board/raspberrypi/adafruit_ts`.

Terminal

```
$ ls -1 board/raspberrypi/adafruit_ts/
config-linux-3.6.11-adafruit_ts+git
linux-3.6.11-0001-Added-Adafruit-TS-support.patch
overlay/
```

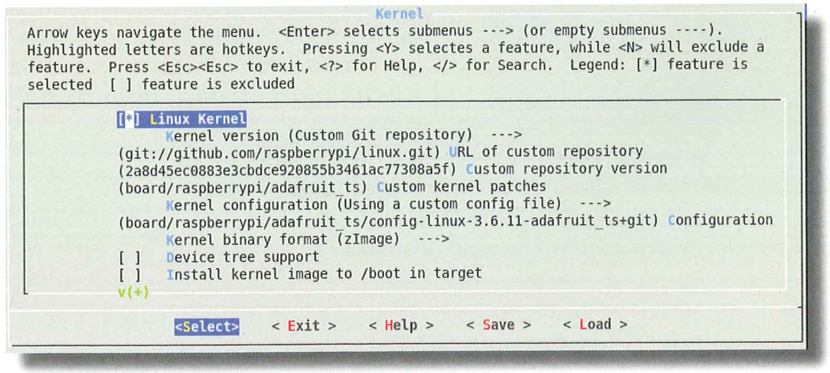


Fig. 4 : Configuration du noyau dans Buildroot

Le répertoire `overlay` contient une arborescence permettant d'ajouter ou de remplacer des fichiers du squelette de distribution par défaut. Ces fichiers sont spécifiques au fonctionnement de l'écran et nous présentons ci-dessous le contenu du répertoire :

Fichier

```
etc/udev/rules.d/95-stmpe.rules
etc/init.d/S11touchscreen
etc/fstab
etc/inittab
root/.profile
root/.directfbrc
```

Le fichier `95-stmpe.rules` est utilisé par UDEV afin de mettre en place un lien symbolique `/dev/input/touchscreen` pour l'accès à l'écran tactile.

Le fichier `S11touchscreen` est exécuté au démarrage du système et effectue le chargement des modules dédiés au pilotage de l'écran. En voici un extrait correspondant à la partie initialisation :

Fichier

```
start() {
  echo -n "Starting touchscreen: "
  modprobe spi-bcm2708
  modprobe fbtft_device name=adafruitts rotate=90
}
```

Les fichiers `.profile` et `.directfbrc` permettent de définir des variables d'environnement utilisées par le système graphique et nous y reviendrons lors du test des bibliothèques DirectFB et Qt. Les fichiers `/etc/fstab` et `/etc/inittab` n'ont aucun lien avec l'utilisation de l'écran et ils sont présents dans les autres configurations Buildroot pour la RPi.

Le script `build_sdcard.sh` présent sur `board/raspberrypi` permet de construire automatiquement l'image à copier sur la carte SD. Nous avons emprunté ce script au projet Yocto [13], qui est un outil similaire à Buildroot mais beaucoup plus complexe (et complet). Le script est lui-même assez complexe et nous l'avons légèrement modifié pour les besoins de Buildroot. Dans le cas de l'utilisation de l'écran, le fichier `board/raspberrypi/cmdline.txt` est copié par le script dans le répertoire `output/images/rpi-firmware` et remplace donc le fichier par défaut créé par le paquet `rpi-firmware`. Le script permet d'ajouter des paramètres de démarrage (options du noyau) par la variable `EXTRA_OPTS` remplacée par les paramètres définis lors de la configuration dans le menu ***System configuration*** de Buildroot. Dans le cas présent, ces paramètres correspondent à la police de caractères utilisée pour la console graphique sur l'écran tactile.

```
() Custom scripts to run before creating filesystem images
(board/raspberrypi/build_sdcard.sh) Custom scripts to run after creating fil
(fbcon=map:10 fbcon=font:MINI4x6) Extra post-{build,image} arguments
```

Fig. 5 : Options pour la console graphique

Les arguments `fbcon` saisis précédemment sont passés au script à partir du deuxième argument, le premier correspondant au répertoire `output/images` :

Fichier

```
# Update cmdline.txt with optional boot params
shift
cat $CURRENT_DIR/../../board/raspberrypi/cmdline.txt | sed -e "s/EXTRA_
OPTS/$*/g" > $CURRENT_DIR/rpi-firmware/cmdline.txt
```

Les fichiers produits sont installés sur la partition VFAT utilisée par la procédure de démarrage de la RPi.

4. TEST DES BIBLIOTHÈQUES GRAPHIQUES

La distribution ainsi construite permet de mettre en œuvre le *framebuffer* de l'écran SPI. La configuration de la distribution – accessible par `make menuconfig` - permet de tester différents composants activés dans le menu ***Graphic libraries and applications*** de Buildroot :

- Accès direct au framebuffer avec l'utilitaire d'affichage d'image `fbv`,
- Bibliothèque DirectFB,
- Bibliothèque Qt 4.

Bien entendu, Buildroot permet d'intégrer d'autres bibliothèques graphiques comme EFL (*Enlightenment Foundation Libraries*) ou SDL, mais nous ne les évoquerons pas dans cet article.

L'utilisation de bibliothèques graphiques nécessite de définir quelques variables d'environnement. Ceci peut être fait aisément grâce au fichier `/root/.profile` présenté ci-dessous :

Fichier

```
export TSLIB_TSDEVICE=/dev/input/touchscreen
export TSLIB_FBDEVICE=/dev/fb1
export FRAMEBUFFER=/dev/fb1

export QWS_MOUSE_PROTO=tslib:/dev/input/touchscreen
export QWS_DISPLAY=linuxfb:/dev/fb1

if [ ! -r /etc/pointercal ]; then
 ts_calibrate
fi
```

La première partie du fichier est utilisée par la bibliothèque TSLIB. Cette dernière permet de fournir une abstraction de l'écran tactile. Les variables `QWS_*` sont utilisées par Qt. Les trois dernières lignes du fichier permettent d'appeler automatiquement l'utilitaire de calibration `ts_calibrate` (fourni par TSLIB) lors de la première connexion. Comme nous l'avons vu précédemment, le périphérique d'entrée de l'écran tactile correspond au fichier spécial `/dev/input/touchscreen`. La figure 6 correspond à la procédure de calibration.

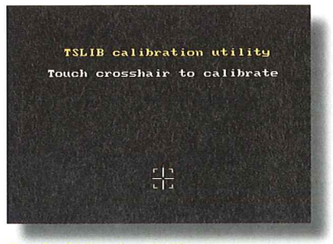


Fig. 6 : Calibration de l'écran

À l'issue de la calibration, les données sont sauvegardées dans le fichier `/etc/pointercal`.

4.1 Test de DirectFB

La bibliothèque DirectFB [14] est assez basique. Son but est de fournir une couche d'abstraction au framebuffer, mais également des primitives permettant la manipulation des fenêtres, des couleurs ou bien la prise en compte des périphériques d'entrée. Contrairement à Qt, cette bibliothèque ne fournit pas d'objets graphiques permettant de construire une IHM

(Interface Homme Machine). De ce fait, la bibliothèque est très légère et les programmes de test n'excèdent par 2 Mi de consommation en RAM. Elle est parfois utilisée sur des architectures légères, comme certaines versions de *set-top box*. L'ajout de DirectFB à la distribution s'effectue simplement en sélectionnant le paquet dans le menu ***Target packages / Graphic libraries and applications***. Outre la configuration principale de DirectFB, nous avons activé le périphérique d'entrée (écran tactile), le clavier USB, ainsi que quelques programmes d'exemple.

La configuration de la session s'effectue dans le fichier `/root/.directfbrc` contenant les deux lignes suivantes :

Fichier
```
disable-module=linux_input
tslib_devices=/dev/input/touchscreen
```

Nous avons sélectionné quelques applications de tests significatifs, comme `df_andi` (la multiplication des pingouins) ou `df_window` (manipulation de fenêtres). Le test de ces applications s'effectue simplement en tapant le nom de la commande sur la console (série) de la carte, ou bien sur une console graphique avec clavier USB. La figure 7 présente l'exécution de `df_window` permettant de manipuler des fenêtres colorées.

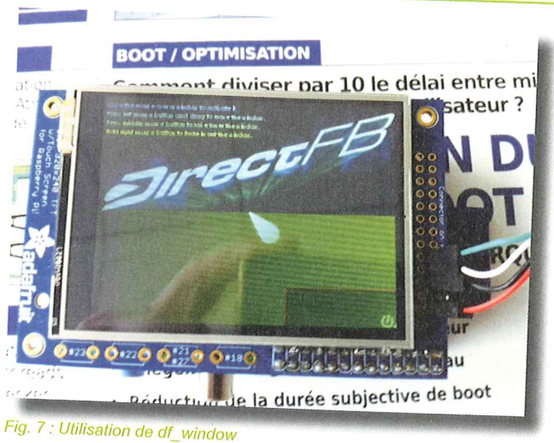


Fig. 7 : Utilisation de df_window

4.2 Test de Qt

La bibliothèque Qt [15] est un des composants de référence pour le développement graphique. La première version développée par la société norvégienne Trolltech est sortie en 1996. À l'époque, la bibliothèque utilise X11 et se fait connaître grâce au bureau graphique KDE. Qt est alors publiée sous double licence GPL et propriétaire. La bibliothèque est rapidement disponible sur les principaux systèmes du marché, soit UNIX, Windows et Mac OS X. Qt est plus qu'une bibliothèque graphique et ses concepteurs définissent le produit comme une interface de développement C++ pour le système cible permettant donc d'assurer la portabilité de l'application. En effet, Qt contient des classes graphiques, mais traite également les accès aux bases de données, la création de threads, etc.

En 2000, la version Qt 2 fait son apparition dans le monde de l'embarqué Linux grâce à son utilisation sur le célèbre PDA Zaurus de SHARP. En 2008, Nokia fait l'acquisition de Trolltech, qui devient *Qt software*. La bibliothèque est désormais disponible sous licence LGPL (et non plus GPL). Nokia adapte alors Qt à d'autres plateformes comme Symbian.

En 2012, l'activité Qt est cédée à la société norvégienne Digia, qui produit la version 5 de Qt début 2013. Cette dernière version est adaptée à des systèmes mobiles comme iOS ou Android. Actuellement, la version 4.8.5 cohabite avec la version 5, qui fonctionne uniquement (ou presque) avec des contrôleurs graphiques accélérés.

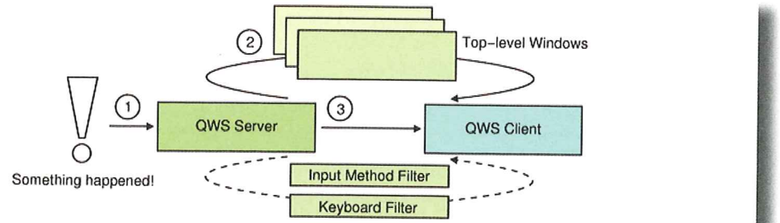


Fig. 8 : Principe de QWS

Dans le cas de notre petit écran SPI, nous avons utilisé Qt 4.8.6 et la couche QWS (*Qt Window System*). Cette interface permet un accès non accéléré au framebuffer, mais n'existe plus sur Qt 5 qui utilise QPA (*Qt Platform Abstraction*). Qt 5 utilise OpenGL (EGLFS) et cette extension est disponible pour le contrôleur graphique interne de la RPi en sélectionnant le paquet `rpi-userland` dans Buildroot. Nous avons réalisé quelques tests sur la RPi avec Qt 5 (et le contrôleur graphique interne), mais le comportement était relativement instable, de nombreux programmes de test se terminant avec une erreur de mémoire. D'après Digia, la portabilité des applications développées en Qt 4.8 est assurée dans Qt 5, puisque ces considérations concernent le support bas niveau.

Pour revenir à QWS, ce dernier utilise une approche assez similaire à *X Window System* (X11). Une des applications est démarrée en mode *serveur* en utilisant l'option `-qws` et les autres applications utilisent les ressources graphiques via le serveur.

Tout comme avec DirectFB, l'ajout de Qt s'effectue dans le menu ***Target packages / Graphic libraries and applications***. La configuration est plus complexe qu'avec DirectFB, mais nous avons sélectionné la configuration minimale afin de limiter le temps de compilation. Au niveau de l'affichage graphique dans le menu ***Graphics drivers*** en figure 9, seul le framebuffer Linux est sélectionné, car il est utilisé par QWS. Au niveau des périphériques de pointage (***Mouse drivers***) en figure 10, nous avons sélectionné toutes les possibilités, l'écran tactile étant pris en compte par la TSLIB. Nous avons fait de même pour le clavier (***Keyboard drivers***) en figure 11.

```
[*] Linux Framebuffer
[ ] Transformed
[ ] Qt Virtual Framebuffer
[ ] VNC
[ ] multiscreen
[ ] directFB
```
Fig. 9 : Configuration de l'affichage

```
[*] pc
[*] linuxtp
[*] linux input
[*] tslib
[*] qvfb
    *** Mouse Options ***
[ ] Hide the mouse cursor
```
Fig. 10 : Configuration des périphériques de pointage

```
[*] tty
[*] linux input
[*] qvfb
```
Fig. 11 : Configuration du clavier

Une fois la carte démarrée, il est aisé de tester les exemples installés sur la distribution. Nous présentons ci-après un exemple de test de l'application **sliders**. Une fois de plus, l'option **-qws** est indispensable au fonctionnement de l'application puisqu'elle est la seule exécutée.

Terminal
```
# /usr/share/qt/examples/widgets/sliders/sliders -qws
```

En cas d'omission de cette option, on obtient l'erreur suivante :

Terminal
```
# /usr/share/qt/examples/widgets/sliders/sliders
QWSSocket::connectToLocalFile could not connect:: Connection refused
QWSSocket::connectToLocalFile could not connect:: Connection refused
```

En cas de démarrage correct, on constate sur l'écran l'affichage présenté en figure 12.

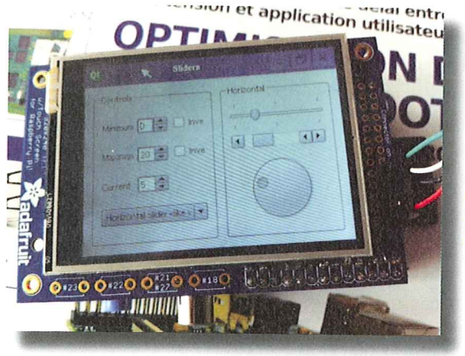


Fig. 12 : Test d'application Qt Sliders

5. DÉVELOPPEMENT AVEC QT

Même si ce n'est pas directement lié au sujet de l'article (donc pas spécifique à la RPi), il nous paraît intéressant de dire quelques mots sur le développement Qt et en particulier sur les notions de multi-plateforme et développement croisé. Nous allons prendre l'exemple d'une application simple, développée pour mettre en évidence les notions de *signaux* et de *slots* qui, sous Qt, remplacent avantageusement le principe très répandu du *callback*.

Dans cette application, un thread non graphique (classe **QThread**) envoie un signal à une interface graphique constituée d'une barre de progression, d'un label et d'un bouton de démarrage. On peut compiler puis tester cette application sous Linux/X11 en utilisant les commandes suivantes :

Terminal
```
$ qmake && make
$ ./Qthread_ex
Mise a jour de la valeur par le thread (emit) 1
Mise a jour de la valeur par le thread (emit) 2
Mise a jour de la valeur par le thread (emit) 3
Mise a jour de la valeur par le thread (emit) 4
Mise a jour de la valeur par le thread (emit) 5
Mise a jour de la valeur par le thread (emit) 6
...
```

On obtient alors l'affichage présenté en figure 13.

Pour compiler l'application pour la cible RPi, il suffit de changer le chemin d'accès la la commande **qmake**, puis de re-compiler l'application. La commande **qmake** a été produite par Buildroot.

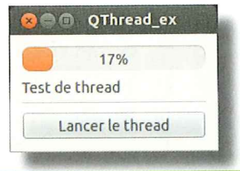


Fig. 13 : Application Qthread_ex sous X11

Terminal
```
$ export PATH=<BR path>/output/host/usr/bin:$PATH
$ make distclean
$ qmake && make
```

Après copie de l'application sur la carte, on obtient l'écran de la figure 14.

Fig. 14 : Application Qthread_ex sur RPi

CONCLUSION

Cet article nous a permis de décrire précisément la mise en place et l'utilisation d'un écran tactile sur la carte RPi. Ce modèle est assez limité de par sa taille et ses performances, mais il permet de s'initier au développement graphique embarqué pour seulement quelques dizaines d'euros. ■

RÉFÉRENCES

[1] Présentation de l'écran Adafruit : **https://www.adafruit.com/products/1601**

[2] Bouton tactile : **http://www.adafruit.com/products/1489**

[3] Boîtier : **https://www.adafruit.com/products/1892**

[4] Câble USB : **http://www.adafruit.com/products/954**

[5] Présentation de l'écran capacitif Adafruit : **https://learn.adafruit.com/adafruit-2-8-pitft-capacitive-touch**

[6] Présentation de l'écran résistif Adafruit : **https://learn.adafruit.com/adafruit-pitft-28-inch-resistive-touchscreen-display-raspberry-pi**

[7] Installation de l'écran résistif : **https://learn.adafruit.com/adafruit-pitft-28-inch-resistive-touchscreen-display-raspberry-pi/software-installation**

[8] Site du projet Buildroot : **http://buildroot.uclibc.org**

[9] Ficheux P., « Les distributions embarquées pour Raspberry Pi », *Open Silicium n°7*

[10] Bodor D., « Personnalisation de Buildroot », *Open Silicium n°10*

[11] Patches adafruit-ts : **http://adafruit-download.s3.amazonaws.com/adafruit-ts.zip**

[12] Fork du projet FBTFT : **https://github.com/xobs/adafruit-rpi-fbtft**

[13] Site du projet Yocto : **https://www.yoctoproject.org**

[14] Bibliothèque DirectFB : **http://directfb.org**

[15] Bibliothèque Qt : **http://qt-project.org**

3 DISTRIBUTIONS ET OS

RTEMS SUR RASPBERRY PI

Pierre Ficheux

La Raspberry Pi (RPi) est le plus souvent utilisée sous GNU/Linux. Il est cependant possible de l'utiliser avec des systèmes beaucoup plus légers comme des RTOS (Real Time Operating System). L'exécutif RTEMS est disponible depuis quelques mois sur la RPi. Après avoir mis en place la chaîne de compilation croisée, nous présenterons quelques exemples dont un pilote pour les GPIO de la RPi. Les tests sont réalisés sur les modèles B et B+ de la RPi.

RTEMS [1] est un exécutif temps réel initialement créé pour l'armée américaine dans les années 80 (*Real Time Executive for Missile Systems*, puis *Military Systems*). Il fut ensuite publié à partir de 1993 sur un serveur FTP. Le développement est désormais réalisé par la société OAR Corporation [2] et RTEMS signifie à présent *Real Time Executive for Multiprocessor Systems*. À la différence d'un système d'exploitation, un exécutif héberge en général une seule application, mais cette dernière peut être composée de plusieurs tâches (ou *threads*) gérées en temps réel. Le système final correspond donc à un seul exécutable contenant l'application liée au noyau de manière statique. La figure 1, extraite de la documentation de RTEMS, montre brièvement l'architecture interne de l'exécutif.

RTEMS est publié sous une licence GPL avec exception. Cette exception permet de ne pas diffuser le code source de l'application liée au noyau RTEMS, ce qui a la conséquence fâcheuse d'interdire l'intégration de code *purement* GPL dans le code de RTEMS.

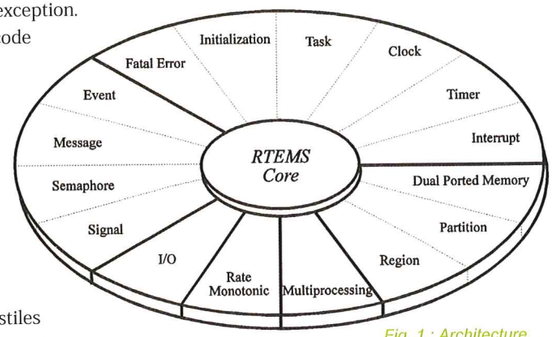


Fig. 1 : Architecture interne de RTEMS

Les domaines d'application de RTEMS sont très éloignés de ceux de la RPi. RTEMS est utilisé dans des applications aéronautiques et spatiales (ESA, NASA, EADS, etc.) [3], le plus souvent sur du matériel très onéreux capable de supporter des environnements hostiles (radiations cosmiques). Nous pouvons citer les cartes AEROFLEX (voir figure 2), à base d'architecture SPARC/LEON3 dont le prix est de plusieurs milliers d'euros !

Plusieurs sondes spatiales utilisent RTEMS pour gérer un ou plusieurs système(s). Une liste de références industrielles est disponible en référence [4]. Il existe cependant quelques applications originales de RTEMS comme la carte Milkymist (voir figure 3), basée sur du matériel ouvert (*open hardware*) et dédiée aux applications d'effets vidéo [5][6].

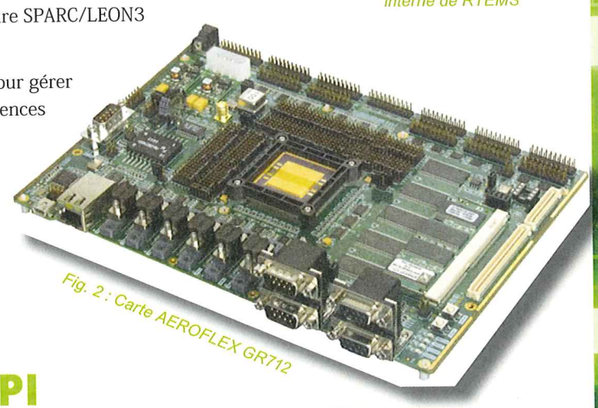


Fig. 2 : Carte AEROFLEX GR712

1. RTEMS ET LA RPI

La carte RPi n'est pas réellement adaptée à l'environnement industriel ciblé par RTEMS. Elle est de relativement bonne qualité, mais reste cependant réservée à l'enseignement, à des maquettes, ou bien aux hobbyistes et autres « geeks ».

Au début de l'année 2013, un ingénieur de la NASA (Alan Cudmore), déjà utilisateur de RTEMS, s'intéressa à la création d'un BSP RTEMS pour la RPi [7]. Ce travail fut réalisé à titre personnel et intégré au dépôt officiel RTEMS (4.11) au début de l'été 2013. Le BSP actuel exploite uniquement la console UART de la RPi, ce qui n'est pas foncièrement gênant, puisque ce système n'est pas réellement destiné à la création d'interfaces graphiques.

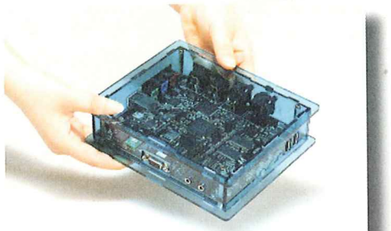


Fig. 3 : Carte Milkymist One

Pour effectuer les tests RTEMS, il est nécessaire de disposer d'un câble d'adaptation USB/TTL. Nous utilisons de nouveau le câble disponible chez Adafruit [8]. Le schéma de branchement, montré en figure 4, doit être scrupuleusement respecté sous peine de détruire la carte !

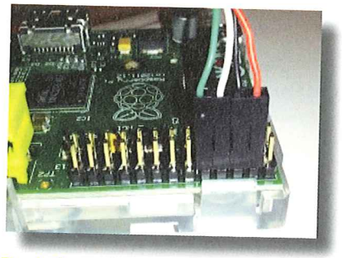


Fig. 4 : Connexion du câble UART

Hormis les GPIO, les autres périphériques (SPI, I2C, carte SD, USB et donc Ethernet) ne sont pas encore pris en compte par le BSP. Ceci étant, il est relativement intéressant de disposer d'une plateforme unique permettant de mettre en place des systèmes aussi variés que Linux en version complète ou embarquée, FreeRTOS, RTEMS et même Xinu [9] (un système UNIX minimaliste utilisé dans l'enseignement). N'oublions pas non plus que la RPi permet de réaliser du développement « bare metal » (sans OS) [10].

2. CHAÎNE DE COMPILATION

La chaîne de compilation croisée est un élément essentiel à mettre en place avant toute expérimentation. Le projet RTEMS fournit désormais un outil intégré (*RTEMS Source Builder*) [11] [12] permettant de produire facilement une chaîne croisée adaptée. L'utilisation de cet outil nécessite d'installer des paquets comme `cvs` ou `python2.7-dev` (sur Ubuntu).

En premier lieu, il est nécessaire de récupérer une copie du dépôt Git de l'outil :

Terminal
```
$ git clone git://git.rtems.org/rtems-source-builder.git
```

On peut alors vérifier l'environnement et démarrer la production de la chaîne croisée qui sera installée sur un répertoire `<install_path>`. La chaîne croisée est constituée des éléments classiques, comme le compilateur GCC, les utilitaires BINUTILS et le débogueur GDB.

Terminal
```
$ cd rtems-source-builder
$ ./source-builder/sb-check
RTEMS Source Builder - Check, v0.4.0
Environment is ok
$ ../source-builder/sb-set-builder --log=l-build.txt --prefix=<install_
path>  4.11/rtems-arm
```

La production de la chaîne est assez longue, mais on obtient finalement la chaîne sur le répertoire spécifié qui occupe environ 600 Mo.

Terminal
```
$ ls -l <install_path>
total 24
drwxr-xr-x  5 pierre pierre 4096 Aug 29 10:55 arm-rtems4.11
drwxr-xr-x  2 pierre pierre 4096 Aug 29 10:57 bin
drwxr-xr-x  3 pierre pierre 4096 Aug 29 10:57 include
drwxr-xr-x  4 pierre pierre 4096 Aug 29 10:57 lib
drwxr-xr-x  3 pierre pierre 4096 Aug 29 10:54 libexec
drwxr-xr-x 14 pierre pierre 4096 Aug 29 10:57 share
```

Dans la suite de l'article, nous partons du principe qu'un compilateur croisé est installé sur le système et qu'il est disponible dans l'environnement de l'utilisateur par ajustement de la variable `PATH`.

Terminal
```
$ export PATH=<install_path>/bin:$PATH
$ arm-rtems4.11-gcc -v
...
Thread model: rtems
gcc version 4.8.3 20140522 (RTEMS 4.11-RSB-2b4cd57ed84f53aebe3f62d284b4ab4
a8abeacec-1,gcc-4.8.3/newlib-19-Aug-2014) (GCC)
```

3. COMPILATION DE RTEMS

Le support de la RPi est intégré à l'arbre officiel de RTEMS, qui est géré sous Git. La compilation du BSP RTEMS s'effectue par les commandes ci-après. Une fois la copie du dépôt terminée, il faut utiliser le script `bootstrap` afin de créer les fichiers utilisés par les outils Autotools, soit `aclocal`, `autoconf` et `automake`.

Terminal
```
$ git clone git://git.rtems.org/rtems.git rtems
$ cd rtems
$ ./bootstrap
$ cd ..
$ mkdir b-rtems && cd b-rtems
$ <path>/rtems/configure --target=arm-rtems4.11 --disable-cxx -disable-
networking --enable-rtemsbsp=raspberrypi --prefix=<path>/target_rpi
$ make
$ make install
```

On peut utiliser ce nouveau BSP après affectation de la variable `RTEMS_MAKEFILE_PATH` :

Terminal
```
$ export RTEMS_MAKEFILE_PATH=<path>/target_rpi/arm-rtems4.11/raspberrypi
```

4. TEST D'UNE APPLICATION

La RPi utilise une carte SD (ou microSD) comme support de démarrage. Cette carte doit disposer d'une partition VFAT sur laquelle on installe un certains nombre d'éléments :

- des fichiers de démarrage binaires fournis par Broadcom,
- des fichiers de configuration (`config.txt`, `cmdline.txt`),
- une image de démarrage (noyau ou application) qui porte par défaut le nom `kernel.img`.

La première application testée est le traditionnel « Hello World », pour lequel nous utiliserons l'interface de programmation POSIX. Le code source est défini ci-après. Nous rappelons que RTEMS utilise une allocation *statique* des ressources en redéfinissant des constantes `CONFIGURE_*` du fichier `confdefs.h`.

Fichier
```
#include <stdio.h>
#include <stdlib.h>

void *POSIX_Init()
{
  printf ("Hello RTEMS/RPi\n");
  exit (0);
}
```

```
#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER

#define CONFIGURE_MAXIMUM_POSIX_THREADS          1
#define CONFIGURE_POSIX_INIT_THREAD_TABLE

#define CONFIGURE_INIT
#include <rtems/confdefs.h>
```

On compile simplement le programme par :

Terminal

```
$ make
test -d o-optimize || mkdir o-optimize
arm-rtems4.11-gcc --pipe -B//home/pierre/docs/articles-pf/lmf/hs_rpi/RTEMS/src/target_rpi/
arm-rtems4.11/raspberrypi/lib/ -specs bsp_specs -qrtems -Wall -O2 -g -mcpu=arm1176jzf-s -c -o
o-optimize/helloworld.o helloworld.c
arm-rtems4.11-gcc --pipe -B//home/pierre/docs/articles-pf/lmf/hs_rpi/RTEMS/src/target_rpi/
arm-rtems4.11/raspberrypi/lib/ -specs bsp_specs -qrtems -Wall -O2 -g -mcpu=arm1176jzf-s
-mcpu=arm1176jzf-s -o o-optimize/helloworld.exe  o-optimize/helloworld.o
arm-rtems4.11-objcopy -O binary --strip-all o-optimize/helloworld.exe o-optimize/helloworld.ralf
arm-rtems4.11-size o-optimize/helloworld.exe
   text    data     bss     dec     hex filename
  79464    1420 134093616  134174500  7ff5724 o-optimize/helloworld.exe
```

Une fois le programme compilé, nous obtenons les fichiers suivants dans le répertoire `o-optimize` :

Terminal

```
$ ls -l o-optimize/
total 3252
-rwxr-xr-x 1 pierre pierre 3203054 Aug 29 14:24 helloworld.exe
-rw-r--r-- 1 pierre pierre   40812 Aug 29 14:24 helloworld.o
-rwxr-xr-x 1 pierre pierre 1020360 Aug 29 14:24 helloworld.ralf
```

Pour tester sur la RPi, il suffit de copier le fichier `helloworld.ralf` sur la partition VFAT de la carte SD en tant que `kernel.img`, puis démarrer la carte.

Terminal

```
$ cp helloworld.ralf /media/Boot/kernel.img
```

Au démarrage de la carte, la console doit afficher le message prévu :

Terminal

```
Hello RTEMS/RPi
```

5. ACCÈS AUX GPIO DE LA RPI

La RPi dispose de plusieurs GPIO disponibles, comme décrit sur les schémas des figures 5a et 5b. Dans le cas de la RPi B+, le connecteur dispose de 40 broches, mais les 26 premières broches sont compatibles avec celles du modèle B [13].

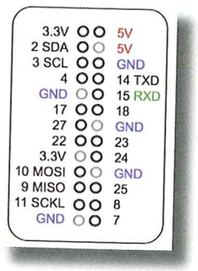


Fig. 5a : Répartition des GPIO sur Raspberry Pi B

Il existe plusieurs outils disponibles pour l'accès aux GPIO, le plus utilisé étant certainement la bibliothèque Python RPI.GPIO. Il n'est bien entendu pas question d'utiliser cela sous RTEMS, mais nous pouvons nous baser sur des macros permettant l'accès direct aux GPIO. Il est aisé d'obtenir l'adresse physique du registre de contrôle des GPIO, car cette adresse est définie dans le BSP RTEMS dans le fichier `include/raspberrypi.h`.

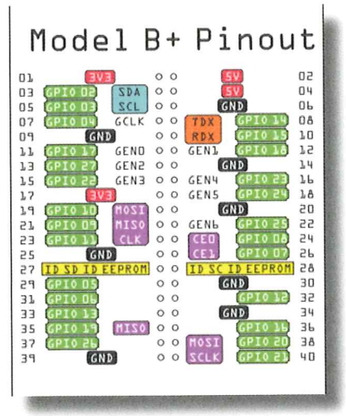


Fig. 5b : Répartition des GPIO sur Raspberry Pi B+

Fichier

```
#define BCM2835_GPIO_REGS_BASE    (0x20200000)
```

Les macros suivantes permettent de définir le mode de fonctionnement des GPIO, en entrée ou en sortie :

Fichier

```
#define INP_GPIO(g) *(gpio+((g)/10)) &= ~(7<<(((g)%10)*3))
#define OUT_GPIO(g) *(gpio+((g)/10)) |=  (1<<(((g)%10)*3))
```

Ces deux dernières macros permettent de changer l'état d'un GPIO en affectant à `GPIO_SET` ou `GPIO_CLR` un masque du type `(1 << num_gpio)`.

Fichier

```
#define GPIO_SET *(gpio+7)  // sets bits which are 1 ignores bits which are 0
#define GPIO_CLR *(gpio+10) // clears bits which are 1 ignores bits which are 0
```

Si le numéro de GPIO est supérieur ou égal à 32 (cas de la RPi B+), on doit utiliser les macros suivantes et affecter le masque `(1 << (num_gpio % 32))` :

Fichier

```
#define GPIO_SET_EXT *(gpio+8)
#define GPIO_CLR_EXT *(gpio+11)
```

Contrairement à Linux, RTEMS manipule directement les adresses *physiques* et il est donc aisé d'écrire une tâche périodique utilisant ces macros, ou encore mieux, d'adapter un exemple existant pour fonctionner sur la RPi. Le projet `examples-v2` disponible sur le dépôt RTEMS contient une suite d'exemples de manipulation de leds [14]. Dans le cas de la RPi, le GPIO numéro 16 correspond à la led ACT utilisée sous Linux lors de l'accès à la carte SD. Dans le cas de la B+, ce numéro est égal à 47.

L'adaptation des exemples à la RPi se fait simplement en ajoutant quelques lignes au fichier `examples-v2/led/led.h` afin de définir les macros `LED_INIT()`, `LED_ON()` et `LED_OFF()`.

Fichier

```
#elif defined(BCM2835_GPIO_REGS_BASE)
// Raspberry Pi
#define INP_GPIO(g) *(gpio+((g)/10)) &= ~(7<<(((g)%10)*3))
#define OUT_GPIO(g) *(gpio+((g)/10)) |=  (1<<(((g)%10)*3))
#define GPIO_SET *(gpio+7)  // sets bits which are 1 ignores bits which are 0
```

```
#define GPIO_CLR *(gpio+10) // clears bits which are 1 ignores bits which are 0
// For GPIO# >= 32 (RPi B+)
#define GPIO_SET_EXT *(gpio+8)  // sets bits which are 1 ignores bits which are 0
#define GPIO_CLR_EXT *(gpio+11) // clears bits which are 1 ignores bits which are 0

// RPi B => led 16
//#define LED_INIT()  do { unsigned int *gpio = (unsigned int *)BCM2835_GPIO_REGS_
BASE; OUT_GPIO(16);} while(0)
//#define LED_ON()  do { unsigned int *gpio = (unsigned int *)BCM2835_GPIO_REGS_BASE;
GPIO_CLR = 1 << 16;} while(0)
//#define LED_OFF()  do { unsigned int *gpio = (unsigned int *)BCM2835_GPIO_REGS_BASE;
GPIO_SET = 1 << 16;} while(0)

// RPi B+ => led 47
#define LED_INIT()  do { unsigned int *gpio = (unsigned int *)BCM2835_GPIO_REGS_BASE;
OUT_GPIO(47);} while(0)
#define LED_ON()  do { unsigned int *gpio = (unsigned int *)BCM2835_GPIO_REGS_BASE;
GPIO_CLR_EXT = 1 << (47 % 32);} while(0)
#define LED_OFF()  do { unsigned int *gpio = (unsigned int *)BCM2835_GPIO_REGS_BASE;
GPIO_SET_EXT = 1 << (47 % 32);} while(0)

#else
```

L'exécution de l'exemple du répertoire **led/timer** permet de constater l'affichage suivant et le clignotement de la led ACT toutes les secondes. Les autres exemples du répertoire sont bien entendu utilisables.

Terminal

```
*** LED BLINKER -- timer ***
```

6. ÉCRITURE D'UN PILOTE POUR LES GPIO

Plutôt qu'un accès direct aux registres, il est plus judicieux d'écrire un pilote que nous pourrons appeler depuis une application RTEMS. En effet, RTEMS dispose d'une API de programmation « UNIX like » permettant de rendre l'accès à un périphérique plus générique :

- Ouverture du pseudo-fichier **/dev/rpi_gpio**,
- Utilisation de l'appel système **ioctl()** pour piloter le GPIO.

Actuellement, nous avons défini les commandes suivantes pour **ioctl()** :

Fichier

```
#define RPI_GPIO_OUT      0    // GPIO is output
#define RPI_GPIO_IN       1    // GPIO is input
#define RPI_GPIO_SET      2    // Set GPIO
#define RPI_GPIO_CLR      3    // Clear GPIO
#define RPI_GPIO_READ     4    // Read GPIO state
```

La déclaration du nouveau pilote s'effectue grâce à une macro que l'on ajoutera à l'environnement de l'application, comme décrit précédemment dans l'article.

Fichier

```
#define RPI_GPIO_DRIVER_TABLE_ENTRY \
  { rpi_gpio_initialize, rpi_gpio_open, rpi_gpio_close, NULL, \
    NULL, rpi_gpio_control }

rtems_device_driver rpi_gpio_initialize(
  rtems_device_major_number,
  rtems_device_minor_number,
  void *
);

rtems_device_driver rpi_gpio_open(
  rtems_device_major_number,
  rtems_device_minor_number,
  void *
);

rtems_device_driver rpi_gpio_close(
  rtems_device_major_number,
  rtems_device_minor_number,
  void *
);

rtems_device_driver rpi_gpio_control(
  rtems_device_major_number,
  rtems_device_minor_number,
  void *
);
```

Dans notre cas, seule la fonction **rpi_gpio_control()** contient réellement du code lié à la RPi, directement déduit des macros précédentes. On utilise, de plus, une nouvelle macro afin de lire l'état d'un GPIO :

Fichier

```
#define GPIO_READ(g)  (*(gpio+13+((g)>>5)) & (1<<((g)&31)))

rtems_device_driver rpi_gpio_control(
  rtems_device_major_number major,
  rtems_device_minor_number minor,
  void *pargp
)
{
  int n, cmd;
  rtems_libio_ioctl_args_t *args = pargp;

  n = (int)(args->buffer);
  cmd = (int)(args->command);
  switch (cmd) {
 case RPI_GPIO_SET :
    if (n >= 32)
      GPIO_SET_EXT = 1 << (n % 32);
    else
      GPIO_SET = 1 << n;
    break;

  case RPI_GPIO_CLR :
    if (n >= 32)
      GPIO_CLR_EXT = 1 << (n % 32);
    else
      GPIO_CLR = 1 << n;
    break;

  case RPI_GPIO_OUT :
    OUT_GPIO(n);
    break;
```

```
    case RPI_GPIO_IN :
      INP_GPIO(n);
      break;

    case RPI_GPIO_READ :
      args->ioctl_return = ((GPIO_READ(n) & (1 << n)) != 0);
      return RTEMS_SUCCESSFUL;

    default:
      printk ("rpi_gpio_control: unknown cmd %x\n", cmd);

      args->ioctl_return = -1;
      return RTEMS_UNSATISFIED;
    }

    args->ioctl_return = 0;

    return RTEMS_SUCCESSFUL;
}
```

Afin de tester le pilote, nous écrivons une tâche périodique utilisant l'API POSIX. Le code complet est disponible sur le dépôt GitHub de *GNU/Linux Magazine* et nous commenterons uniquement les points essentiels propres à RTEMS. Un point important concerne la configuration de l'application, puisqu'il est nécessaire d'ajouter la définition du nouveau pilote à l'environnement de l'application. L'autre point est l'augmentation du nombre maximum de descripteurs de fichiers, puisque l'application en utilise un nouveau pour l'accès au pilote.

Fichier

```
/* configuration information */

#define CONFIGURE_APPLICATION_NEEDS_CONSOLE_DRIVER
#define CONFIGURE_APPLICATION_NEEDS_CLOCK_DRIVER
#define CONFIGURE_APPLICATION_EXTRA_DRIVERS RPI_GPIO_DRIVER_TABLE_ENTRY
#define CONFIGURE_LIBIO_MAXIMUM_FILE_DESCRIPTORS 5

#define CONFIGURE_MAXIMUM_POSIX_TIMERS          1
#define CONFIGURE_MAXIMUM_POSIX_THREADS  1

#define CONFIGURE_EXTRA_TASK_STACKS             (6 * RTEMS_MINIMUM_STACK_SIZE)

#define CONFIGURE_POSIX_INIT_THREAD_TABLE

#define CONFIGURE_INIT
#include <rtems/confdefs.h>
```

La principale fonction correspond à l'échéance du compteur POSIX permettant l'écriture sur le GPIO 25 (ou sur la led ACT). Cette même fonction est utilisée pour lire un GPIO (17) en entrée. Ce compteur est programmé avec une période de 10 ms. Lorsque le bouton change d'état, la période est multipliée ou divisée par 2.

Fichier

```
#define G_IN    17
#define G_OUT   25

// Led
//#define G_OUT   16 // RPi B
//#define G_OUT   47 // RPi B+
```



```
void got_signal (int sig)
{
  static int n = 0;
  int status, input;

  if (n % 2 == 0)
    status = ioctl (fd, RPI_GPIO_SET, G_OUT);
  else
    status = ioctl (fd, RPI_GPIO_CLR, G_OUT);

  if ((input = ioctl (fd, RPI_GPIO_READ, G_IN)) != gpio_input) {
    printf ("input= %d\n", ioctl (fd, RPI_GPIO_READ, G_IN));
    gpio_input = input;

    // Update timer
    if (gpio_input % 2)
      ti.it_interval.tv_nsec = PERIOD_NS;
    else
      ti.it_interval.tv_nsec = PERIOD_NS * 2;

    ti.it_value.tv_sec = 0;
    ti.it_value.tv_nsec = 1000000;
    ti.it_interval.tv_sec = 0;

    timer_settime(myTimer, 0, &ti, &ti_old);
  }

  n++;

  if (status)
    fprintf (stderr, "status= %d errno= %d => %s\n", status, errno,
strerror(errno));
}
```

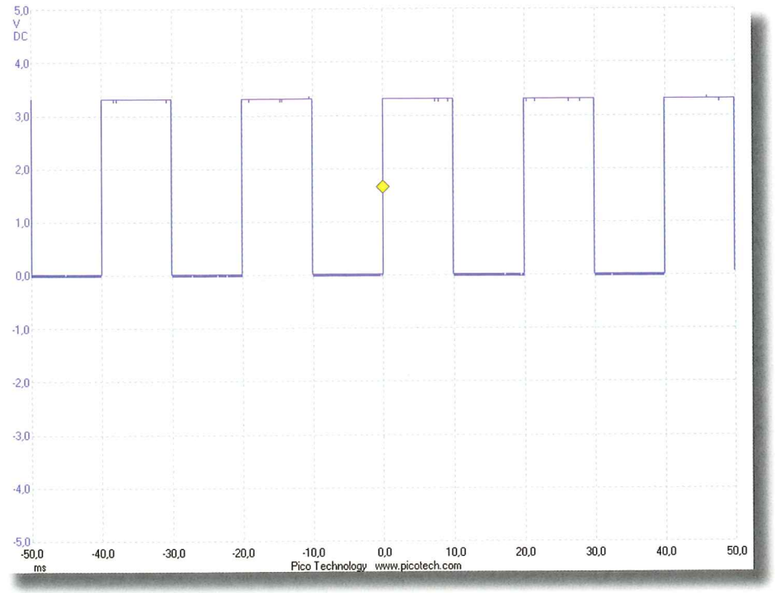


Fig. 6 : Périodicité de la tâche

L'initialisation s'effectue dans la fonction principale par le code suivant :

Fichier

```
if ((fd = open ("/dev/rpi_gpio", O_RDWR)) < 0) {
  fprintf (stderr, "open error => %d %s\n", errno, strerror(errno));
  exit (1);
}

ioctl(fd, RPI_GPIO_IN, G_IN);
ioctl(fd, RPI_GPIO_OUT, G_OUT);
```

Si l'on place un oscilloscope sur la broche 25, on obtient l'image de la figure 6 (page précédente) indiquant la périodicité de la tâche.

Afin d'exploiter le GPIO 17 en entrée, nous utilisons un montage classique basé sur une résistance de rappel de 10 KOhms, comme décrit sur la photo de la figure 7a et le schéma 7b. Le fil blanc correspond au GPIO 17, le rouge à l'alimentation 3,3V et le noir à la masse. Lors de l'action sur le bouton-poussoir, l'application affiche l'état de l'entrée lors de l'appel à `ioctl()`.

Terminal

```
input= 0
input= 1
input= 0
```

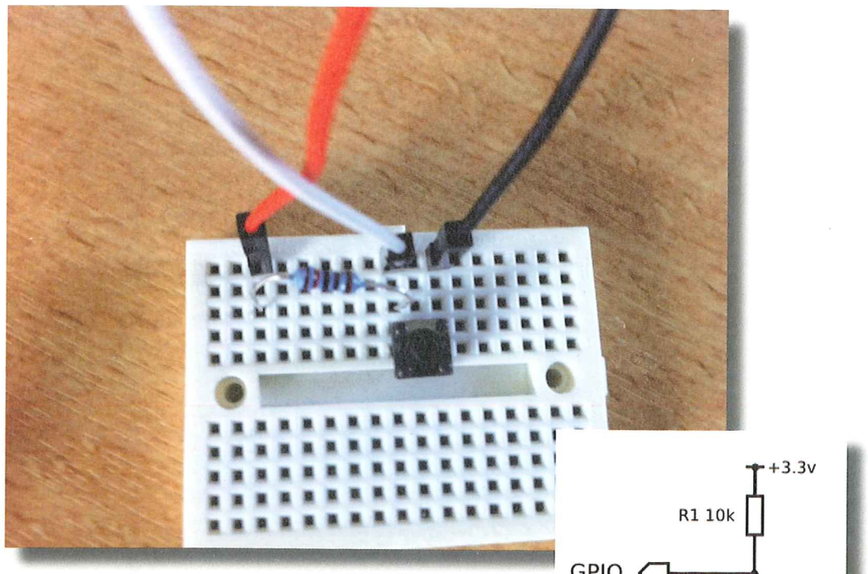


Fig. 7a & 7b : Test du GPIO en entrée

Actuellement, le GPIO est testé en *polling*, ce qui n'est pas la meilleure solution. La gestion des interruptions n'est cependant pas si simple sous RTEMS, car elle dépend de la plateforme utilisée.

CONCLUSION

Le BSP RTEMS pour la Raspberry Pi est encore relativement limité, car il reste à mettre en place le support pour les autres éléments de la carte (USB/Ethernet, SPI, carte SD, console graphique, etc.). La console UART fonctionne en « polling », car le mode interruption n'est pas encore au point et la période des compteurs semble limitée à 10 ms (?).

La licence un peu particulière de RTEMS (GPL avec exception) ne facilite pas le travail, car il n'est pas possible d'utiliser du code hérité d'un autre projet purement GPL. À titre d'exemple, le portage d'U-Boot pour la RPi (sous GPL) dispose de l'accès USB, Ethernet et carte SD. ■

RÉFÉRENCES

[1] Le site du projet RTEMS : **http://www.rtems.org**

[2] Site d'OAR Corp. : **http://www.oarcorp.com/rtems**

[3] RTEMS chez EADS Astrium : **http://ingenierie.openwide.fr/content/download/941/11986/file/RTEMS_ASTRIUM_2012-1.pdf**

[4] Références d'utilisation de RTEMS : **http://www.rtems.org/node/70**

[5] Le site M-Labs : **http://m-labs.hk/m1.html**

[6] La carte Milkymist : **http://2010.rmll.info/Milkymist-a-free-System-on-Chip-for-real-time-video.html?lang=fr**

[7] Utilisation de RTEMS sur Raspberry Pi : **http://alanstechnotes.blogspot.fr/2013/03/setting-up-rtems-development.html**

[8] Câble USB/série par Adafruit : **http://www.adafruit.com/products/954**

[9] Le site du projet Xinu sur Raspberry Pi : **http://xinu.mscs.mu.edu/Raspberry_Pi**

[10] Programmation « bare metal » sur Raspberry Pi : **http://en.wikibooks.org/wiki/Bare-metal_Raspberry_Pi_Programming**

[11] Outil de construction de la chaîne de compilation : **http://www.rtems.org/wiki/index.php/RTEMS_Source_Builder**

[12] Exemples de programmes RTEMS : **http://git.rtems.org/examples-v2**

[13] Les GPIO de la Raspberry Pi : **http://elinux.org/RPi_Low-level_peripheral**

[14] Une autre documentation sur cet outil : **http://www.rtems.org/ftp/pub/rtems/people/chrisj/source-builder/source-builder.html**

4 APPLICATIONS

À découvrir dans cette partie...

4 APPLICATIONS

COMMUNIQUER EN I²C AVEC UN CAPTEUR DE TEMPÉRATURE

Christophe Blaess

Le protocole i²c est le plus épuré des bus de communication entre un processeur et ses périphériques. Très simple à employer par le hobbyiste, nous allons le mettre en œuvre pour lire et programmer un capteur de température externe. Ce dernier ne nous servira qu'à titre d'exemple, tout périphérique i²c pouvant convenir.

On présente généralement i²c (*Inter Integrated Circuit*) comme le plus simple des bus de communication utilisés dans l'électronique moderne. Il s'appuie simplement sur deux signaux appelés SDA (*Serial Data*) et SCL (*Serial Clock*), sans oublier la masse commune entre les équipements. Il s'agit d'une communication bidirectionnelle *half-duplex* – où chacun ne parle qu'à son tour – reposant sur une communication série synchrone. Commençons par une présentation de ce protocole.

1. BUS I²C

Le protocole permet de mettre en communication un composant **maître** (généralement le microprocesseur) et plusieurs périphériques **esclaves**. Plusieurs maîtres peuvent partager le même bus, et un même composant peut passer du statut d'esclave à celui de maître, ou inversement. Toutefois, la communication n'a lieu qu'entre un seul maître et un seul esclave. Notons également que le maître peut également envoyer un ordre à tous ses esclaves simultanément (par exemple, une mise en sommeil ou une demande de réinitialisation).

1.1 Signaux électriques

Au niveau électrique, le protocole utilise des signaux alternant entre des niveaux bas et hauts, le plus fréquemment il s'agit de {0, 5V} ou de {0, 3.3V}. Le signal d'horloge SCL est produit par le maître. Le signal de données SDA est mis au niveau haut ou bas par le maître ou l'esclave, suivant la phase de communication (voir figure 1).

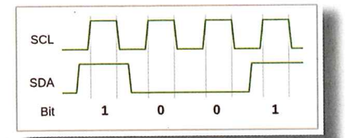


Fig. 1 : Exemple de signaux i²c

Pendant toute la durée du créneau haut du signal d'horloge SCL, le signal de données SDA doit être maintenu au niveau haut ou bas, suivant que l'on transmette un 1 ou un 0.

Enfin, comme le montre la figure 2, des configurations particulières des signaux (produits par le maître) permettent d'indiquer un début et une fin d'échange, en réalisant ce que l'on nomme les conditions START et STOP. Il s'agit d'une variation de signal SDA pendant un créneau de l'horloge.

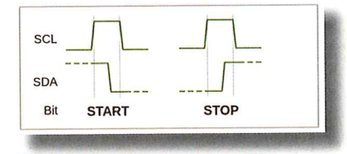


Fig. 2 : Conditions START et STOP

1.2 Protocole de communication

La communication s'établit toujours à l'initiative du maître. Celui-ci présente une condition START sur la ligne SDA, suivie de l'**adresse** (sur 7 bits) de l'esclave avec lequel il souhaite communiquer, puis un bit indiquant le sens (0 = écriture ou 1 = lecture) de la communication (voir figure 3).

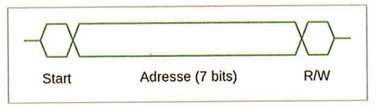


Fig. 3 : Initialisation de la communication

Le fait que les adresses soient sur 7 bits, mais toujours suivies d'un bit indiquant le sens de transmission, conduit parfois à considérer que chaque périphérique est doté de deux adresses unidirectionnelles sur 8 bits : l'adresse paire est destinée à l'écriture vers le périphérique et l'adresse impaire à la lecture.

Il existe quelques adresses particulières, notamment `0x00` qui envoie un message à destination de tous les esclaves. Plus généralement, les adresses inférieures à `0x08` et supérieures à `0x77` ont des rôles spécifiques.

Lorsqu'un périphérique reconnaît son adresse, il doit écrire sur la ligne SDA un bit 0, indiquant ainsi un acquittement (ACK).

Si le message est une écriture vers le périphérique, le maître peut alors envoyer sa commande, octet par octet, l'esclave acquittant chacun d'entre eux par un bit ACK.

Il existe naturellement des subtilités dans ce protocole, pour gérer les erreurs de transmission, les répétitions de messages, les changements de statut esclave/maître et les arbitrages de bus entre plusieurs maîtres.

Pour en savoir plus, je recommande l'article « i²c » de Wikipédia.

1.3 SMBus

La plupart des documentations font référence au protocole i²c, mais l'implémentation que l'on rencontre sur les systèmes monocartes sous Linux (Raspberry Pi, BeagleBone Black, etc.) est plutôt une spécialisation de ce protocole nommée **SMBus** (*System Management Bus*). Les différences essentielles sont des limitations dans les tensions autorisées (3.3V maxi.), ou les fréquences de communication (10 à 100 kHz seulement).

Le protocole SMBus est plus particulièrement utilisé pour communiquer entre un microprocesseur et les périphériques annexes sur la carte-même (ventilateur, capteur de température, etc.). Ceci explique que de nombreuses fonctions de l'API Linux pour le sous-système i²c soient préfixées par `smbus`.

2. LINUX, RASPBERRY PI ET I²C

Le protocole i²c est supporté par le noyau Linux depuis sa version 2.4. De nombreux périphériques sont reconnus par le kernel, notamment dans le sous-système **Hwmon** (*Hardware Monitor*).

L'accès depuis l'espace utilisateur est facilité par le module `i2c-dev` qui rend les bus i²c visibles dans le répertoire `/dev` sous forme de fichiers spéciaux représentant des périphériques en mode caractère.

2.1 Démarrage

Démarrons notre Raspberry Pi avec une distribution classique – j'utilise ici une Raspbian. Initialement, le noyau Linux ne détecte aucun contrôleur i²c, c'est normal.

Terminal
```
$ ls /sys/class/i2c-adapter/
$
```

Il nous faut charger dans le noyau le module qui gère le contrôleur i²c inclus dans le *system-on-chip* (de la famille Broadcom 2708) du Raspberry Pi. Ce module est nommé `i2c_bcm2708`. On peut le charger manuellement ainsi :

Terminal
```
$ sudo -i
# modprobe i2c_bcm2708
```

Pour éviter que ce driver ne soit chargé systématiquement au démarrage, le module `i2c_bcm2708` est inscrit dans la blacklist de `modprobe`. Comme peu d'utilisateurs s'en servent, cela évite de consommer inutilement des ressources. Pour le retirer de la blacklist, il suffit d'éditer le fichier `/etc/modprobe.d/raspi-blacklist.conf` et de mettre en commentaire la ligne :

Fichier
```
blacklist  i2c-bcm2708
```


en la précédant par un caractère dièse `#`. Ainsi, au prochain démarrage, le module sera automatiquement chargé.

Terminal
```
# ls /sys/class/i2c-adapter/
i2c-0  i2c-1
```

Le Raspberry Pi dispose de deux interfaces i²c. Le bus numéro 0 était accessible à travers le connecteur P5 qui était apparu dans la seconde version du Raspberry Pi modèle B (voir **http://elinux.org/Rpi_Low-level_peripherals#P5_header**) et qui a disparu depuis le modèle B+. De plus, ce connecteur étant présent sous forme de simples trous cuivrés, il était donc nécessaire de venir y souder des broches pour pouvoir l'utiliser. Les signaux du bus 0 sont toujours présents – mais pas très accessibles – sur les connecteurs J3 (*camera*) et J4 (*display*).

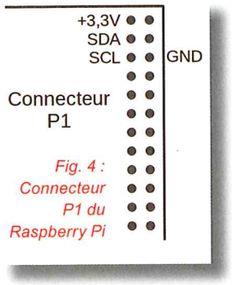


Fig. 4 : Connecteur P1 du Raspberry Pi

Nous allons plutôt nous intéresser au second bus, accessible via le port d'extension P1 (voir figure 4), sur deux broches (que l'on peut également employer pour des entrées/sorties par GPIO) identiques quel que soit le modèle de Raspberry. Il s'agit de la broche 3 (signal SDA) et de la broche 5 (signal SCL).

Le bus i2c-0 est vide, mais le bus i2c-1 contient déjà deux périphériques :

Terminal
```
# ls /sys/class/i2c-adapter/i2c-0/
delete_device  device  name  new_device  power  subsystem  uevent
# ls /sys/class/i2c-adapter/i2c-1/
1-003b  delete_device  name        power      uevent
1-004c  device         new_device  subsystem
# cat /sys/class/i2c-adapter/i2c-1/1-003b/name
wm8804
# cat /sys/class/i2c-adapter/i2c-1/1-004c/name
pcm5122
```

Les deux périphériques intégrés dans le *system-on-chip* sont un Wolfson Microelectronics WM8804 (transceiver audio) et un Texas Instruments PCM 5122 (convertisseur analogique/numérique audio).

2.2 Ajout d'un périphérique i²c

Nous allons établir une communication avec un capteur de température **DS 1621** de Maxim Integrated et dont le brochage est donné en figure 5. Ce composant peu coûteux présente l'avantage d'être largement disponible en boîtier DIP (*Dual Inline Package*) aisément utilisable sur la platine d'essai *breadboard* du bidouilleur amateur.

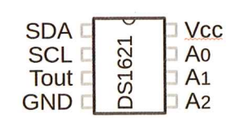


Fig. 5 : Brochage du boîtier DIP DS1621

Ce composant mesure la température ambiante (dans la plage -55°C à +125°C) et la communique, avec une résolution d'un demi-degré, par ses broches SDA et SCL. Il est accessible sur une adresse i²c, dont les 4 bits de poids fort sont toujours `1001` et les 3 bits de poids faible configurables suivant la valeur indiquée sur les broches A_0, A_1 et A_2. Les adresses sont donc entre `0x48` (`1001000b` avec $A_0 = A_1 = A_2 = 0$) et `0x4F` (`1001111b` avec $A_0 = A_1 = A_2 = 1$).

On peut également programmer par l'interface i²c un seuil de température en deçà duquel la sortie T_{out} est à 0 et au-delà duquel elle passe automatiquement à 1.

Un driver pour DS1621 est déjà disponible dans le noyau Linux depuis longtemps, mais nous n'allons pas l'utiliser ici, car c'est l'accès direct depuis l'espace utilisateur qui m'intéresse.

À savoir

Vous trouverez sur de nombreux montages utilisant des communications en i²c des résistances de tirage reliant chacune des broches SDA et SCL (qui sont en collecteur ouvert) avec Vcc pour éviter que la tension flotte quand personne ne force de niveau bas sur ces lignes. Ceci est déjà réalisé par le circuit électronique du Raspberry Pi, deux résistances de 1,8 kOhms étant présentes entre les broches du BCM 2835 et l'alimentation +3.3V.

Nous relions tout naturellement la broche SDA du Raspberry Pi avec son homologue du DS1621, ainsi que les broches SCL des deux composants. On relie les deux broches de masse GND, et on peut utiliser la broche +3,3V du Raspberry Pi pour alimenter le DS1621 sur son entrée Vcc.

On peut remarquer sur la photo de la figure 6 (où j'ai surligné les connexions établies par la breadboard), que les broches A_0, A_1, A_2 du DS 1621 sont toutes reliées au +Vcc par l'intermédiaire de petits *straps*, ceci lui attribue donc l'adresse **0x4F**.

Démarrons le Raspberry Pi et essayons de communiquer avec notre composant.

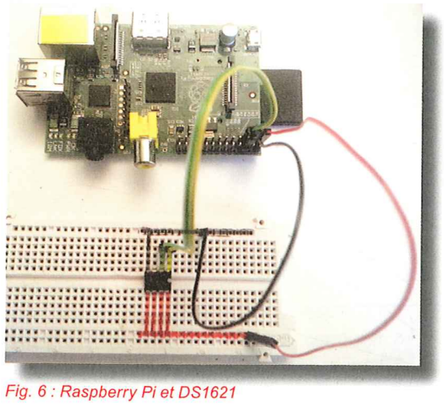


Fig. 6 : Raspberry Pi et DS1621

3. ACCÈS DEPUIS LE SHELL

Bien que le module soit chargé, le noyau ne voit pas le DS1621. Le bus i²c, à l'opposé d'un bus PCIe par exemple, ne permet pas d'énumérer et d'identifier automatiquement les périphériques présents. Pour que le kernel puisse le gérer directement, il faudrait lui indiquer la présence du DS1621 dans un fichier de configuration de la plateforme avant la compilation du kernel, ou dans un fichier de description *device tree* au moment du boot. Ceci n'est pas l'objet de cet article.

Après avoir chargé le module **i2c_bcm2708**, un second module va être nécessaire ; c'est celui qui va nous donner accès, depuis l'espace utilisateur, aux périphériques connectés au bus i²c. Il les rendra accessibles à travers une interface dans le répertoire **/dev**, ce qui donne au module le nom **i2c_dev**.

Terminal

```
# modprobe i2c_dev
# ls -l /dev/
total 0
[...]
crw-rw---T 1 root i2c      89,   0 Jul 13 16:19 i2c-0
crw-rw---T 1 root i2c      89,   1 Jul 13 16:19 i2c-1
[...]
```

3.1 Détection des périphériques

La manière la plus simple de communiquer depuis le shell est d'employer les utilitaires du package **i2c-tools**. Il faut tout d'abord installer ce dernier, car il n'est généralement pas inclus d'origine dans les distributions courantes :

Terminal

```
# apt-get install i2c-tools
```

Nous pouvons commencer par lister les bus présents, puis scanner celui qui nous intéresse à l'aide de l'outil **i2cdetect** :

Terminal

```
# i2cdetect -l
i2c-0   i2c             bcm2708_i2c.0                   I2C adapter
i2c-1   i2c             bcm2708_i2c.1                   I2C adapter
# i2cdetect  1
WARNING! This program can confuse your I2C bus, cause data loss and worse!
I will probe file /dev/i2c-1.
I will probe address range 0x03-0x77.
Continue? [Y/n] y
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- UU -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- 4f
50: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

Le nom de la commande **i2cdetect** est suivi d'un **1** pour indiquer le numéro de bus à parcourir. On peut affiner la plage de recherche en indiquant des adresses de départ et de fin. Comme l'opération peut perturber certains périphériques, **i2cdetect** nous demande confirmation (à laquelle nous répondons **y**). Si nous souhaitons éviter cette confirmation (pour inclure l'invocation dans un script par exemple), il suffit d'ajouter l'option **-y** sur la ligne de commandes.

Le tableau affiché nous indique les périphériques détectés (en remplissant la case avec leur adresse) et ceux dont la détection a été évitée car un driver s'en occupe déjà (avec les caractères '**UU**' comme c'est le cas du WM8804 à l'adresse **0x3b**).

Notre capteur de température a été détecté à l'adresse **0x4f**, ce qui est cohérent avec la configuration de ses entrées d'adresses.

3.2 Émission et réception de données

Pour envoyer des ordres à un périphérique i²c depuis le shell, on utilise la commande **i2cset**. Celle-ci est conçue comme une interface pour remplir des registres distants. On lui passe en argument le numéro de bus et l'adresse du périphérique suivis du numéro de registre et de la valeur à y inscrire. Suivant les périphériques, ceci pourra également être interprété comme un numéro de commande suivi d'un ou plusieurs argument(s).

Symétriquement, on emploiera **i2cget** pour lire l'état d'un registre distant. Ce qui peut aussi s'interpréter comme l'envoi d'une commande indiquant au périphérique de nous renvoyer immédiatement une valeur. Suivant les cas, on lira un ou plusieurs octet(s).

En consultant la documentation du DS1621, nous remarquons les commandes suivantes :

Nom commande	Numéro	Signification et arguments
START CONVERT	0xEE	Débute une mesure de température. Pas d'argument.
ACCESS CONFIG	0xAC	Lit l'état du convertisseur ou écrit sa configuration. La commande est suivie d'un seul octet. Le détail de la configuration est décrit dans la documentation du DS1621.
READ TEMPERATURE	0xAA	Demande à lire le résultat de la dernière acquisition de température.

Le DS1621 est un composant très simple. Il dispose de deux modes de fonctionnement : « one shot » (une seule mesure) ou « continu » (mesures de température en boucle permanente). Nous allons choisir le mode *one shot*.

Le principe de l'acquisition complète consistera donc à réaliser les étapes suivantes :

- écriture (avec la commande **ACCESS CONFIG**) dans le registre de configuration du bit **0x01** afin d'activer le mode *one shot* ;
- démarrage de la conversion avec **START CONVERT** ;
- lecture de l'état avec **ACCESS CONFIG** jusqu'à ce que le registre d'état contienne un bit de poids fort à 1 (**0x80**), ce qui indique que la mesure est terminée ;
- lecture du résultat avec **READ TEMPERATURE**.

Voici le script shell (**lecture-temperature.sh**) qui affiche la température ambiante :

Fichier

```
#! /bin/sh

# Adresse du DS 1621
I2C_BUS=1
I2C_ADDR=0x4F

# Commandes
START_CONVERT=0xEE
ACCESS_CONFIG=0xAC
READ_TEMPERATURE=0xAA

# Lire la configuration
status=$(i2cget -y ${I2C_BUS} ${I2C_ADDR} ${ACCESS_CONFIG})
        if [ $? -ne 0 ]; then exit 1; fi

# Ajouter le bit "One shot"
status=$((status | 0x01))

# Écrire la configuration
i2cset -y ${I2C_BUS} ${I2C_ADDR} ${ACCESS_CONFIG} $status
        if [ $? -ne 0 ]; then exit 1; fi

# Démarrer l'acquisition
i2cset -y ${I2C_BUS} ${I2C_ADDR} ${START_CONVERT}
        if [ $? -ne 0 ]; then exit 1; fi

while true
do
   # Lire l'état
   status=$(i2cget -y ${I2C_BUS} ${I2C_ADDR} ${ACCESS_CONFIG})
          if [ $? -ne 0 ]; then exit 1; fi

   # Si l'acquisition est terminée, sortir de la boucle
   if [ $(( $status & 0x80 )) -eq $((0x80)) ]; then break; fi
done

# Lire la température
temp=$(i2cget -y ${I2C_BUS} ${I2C_ADDR} ${READ_TEMPERATURE})
      if [ $? -ne 0 ]; then exit 1; fi

# Gérer les valeurs négatives
if [ $(($temp)) -gt 127 ]; then temp=$((temp - 256)); fi

# Écrire la température sur la sortie standard
echo $(($temp))
```

On peut faire divers essais pour vérifier le résultat. Par exemple, le petit script suivant permet d'enregistrer la température au cours d'une expérience en l'horodatant toutes les secondes :

Fichier

```
#! /bin/sh

PROG=./lecture-temperature.sh
d0=$(date +%s)

while true
do
    d=$(date +%s)
    printf "%d\t" $((d-d0))
    ${PROG}
    sleep 1
done
```


Il devient très facile et amusant de visualiser (avec Gnuplot) les fluctuations de la température lors de diverses manipulations (poser une tasse de café sur le composant, placer la breadboard au congélateur, etc.).

4. COMMUNICATION I²C AVEC D'AUTRES LANGAGES

4.1 Accès avec Python

Langage favori de nombreux développeurs et hackers actuels, Python offre une interface pour la communication en i²c qui s'appuie sur les fonctionnalités **i2c-dev** que nous avons installées précédemment.

Il existe une bibliothèque permettant un accès simple aux périphériques qui supporte le sous-ensemble SMBus d'i²c. Les fonctions de haut-niveau permettent une communication avec le périphérique en dissimulant les multiples émissions/réceptions d'octets sous-jacentes. Installons sur notre Raspberry Pi cette bibliothèque.

Terminal

```
~# apt-get install python-smbus
```

Les principales méthodes sont :

smbus.SMBus(*n*)	Renvoie un objet SMBus permettant l'accès au périphérique **/dev/i2c-*n***.
read_byte (adr) **write_byte (adr, val)**	Lit ou écrit un octet directement à l'adresse **adr**, sans préciser de registre.
read_byte_data (adr, reg) **write_byte_data (adr, reg, val)**	Lit ou écrit le contenu du registre **reg** du périphérique à l'adresse **adr** sous forme d'octet.
read_word_data (adr, reg) **write_word_data (adr, reg, val)**	Lit ou écrit un mot de deux octets dans le registre **reg** du périphérique à l'adresse **adr**.
read_bloc_data (adr, reg) **write_bloc_data (adr, reg, val[])**	Lit ou écrit un tableau d'octets (maximum 32) dans le registre **reg** du périphérique à l'adresse **adr**. Les données sont précédées de la longueur du tableau.

Voici un script Python (**lecture-temperature.py**) qui réalise le même travail que le précédent script shell :

Fichier

```
#! /usr/bin/python

import smbus

bus = smbus.SMBus(1)
addr= 0x4F

START_CONVERT=0xEE
ACCESS_CONFIG=0xAC
READ_TEMPERATURE=0xAA

bus.write_byte_data(addr, ACCESS_CONFIG, bus.read_byte_data(addr, ACCESS_
CONFIG)| 0x01)

# Démarrer l'acquisition
bus.write_byte(addr, START_CONVERT)
while (bus.read_byte_data(addr, ACCESS_CONFIG) & 0x80) == 0:
    pass

v=bus.read_byte_data(addr, READ_TEMPERATURE)
if v > 127:
    v -= 256
print v
```

4.2 Accès en C

Si l'on choisit le langage C pour communiquer en i²c avec un périphérique, ce sera probablement pour des raisons d'efficacité, de rapidité et de contrôle des opérations réalisées. L'API employée pourra néanmoins être proche de celle des autres langages.

Nous emploierons l'interface **i2c-dev**, car c'est la meilleure pour accéder depuis l'espace utilisateur, mais nous lui enverrons des ordres spécifiques en utilisant le mécanisme des ioctl (*I/O Controls*). Nous ouvrirons donc **open()** le bus /**dev/i2c-***n* qui nous intéresse, et ferons des accès avec l'appel-système **ioctl()** :

Fichier

```
int open (const char * filename, int flags, mode_t mode);
int ioctl (int fd, int request...);
```

Par exemple, pour indiquer une fois pour toute l'adresse du périphérique, nous utiliserons un appel avec l'argument **I2C_SLAVE** défini dans **/usr/include/linux/i2c-dev.h**. Il serait possible de coder tous les échanges au protocole SMBus avec des **ioctl()**, mais cela serait très fastidieux. Aussi existe-t-il une mini-bibliothèque, qui propose des fonctions *inline* s'appuyant sur les **ioctl()** bas-niveaux. La particularité de cette bibliothèque est qu'elle s'installe en modifiant le fichier **i2c-dev.h** du système pour y inscrire ses fonctions *inline*.
On l'installe ainsi :

Terminal

```
# apt-get install libi2c-dev
```

Nous disposons alors, entre autres, des fonctions :

Fichier

```
__s32 i2c_smbus_write_quick(int fd, __u8 val)
__s32 i2c_smbus_read_byte(int fd)
__s32 i2c_smbus_write_byte(int fd, __u8 val)
__s32 i2c_smbus_read_byte_data(int fd, __u8 cmd)
__s32 i2c_smbus_write_byte_data(int fd, __u8 cmd, __u8 val)
__s32 i2c_smbus_read_word_data(int fd, __u8 cmd)
__s32 i2c_smbus_write_word_data(int fd, __u8 cmd, __u16 val)
__s32 i2c_smbus_read_block_data(int fd, __u8 cmd, __u8 *val)
__s32 i2c_smbus_write_block_data(int fd, __u8 cmd, __u8 lg, const __u8 *val)
```

Les fonctions qui nous intéressent sont essentiellement celles suffixées par **_data**, car elles permettent d'indiquer un numéro de registre (de commande) et les données à lire ou écrire.

Voici donc le programme qui réalise le même travail que les précédents. Son écriture est plus longue, car les traitements d'erreur sont écrits de manière plus complète.

Fichier

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <linux/i2c-dev.h>
#include <sys/ioctl.h>

#define DS1621_I2C_BUS              "/dev/i2c-1"
#define DS1621_SLAVE_ADDR           0x4F
#define DS1621_ACCESS_CONFIG        0xAC
#define DS1621_START_CONVERT        0xEE
#define DS1621_READ_TEMPERATURE     0xAA
#define DS1621_ONE_SHOT             0x01
#define DS1621_DONE                 0x80
```



```
int main(int argc, char * argv[])
{
        int fd;
        int value;

        // Obtenir l'accès au bus i2c
        fd = open(DS1621_I2C_BUS, O_RDWR);
        if (fd < 0) {
                perror(DS1621_I2C_BUS);
                exit(EXIT_FAILURE);
        }

        // Fixer l'adresse de l'esclave avec qui communiquer
        if (ioctl(fd, I2C_SLAVE, DS1621_SLAVE_ADDR) < 0) {
                perror("Slave unreachable");
                exit(EXIT_FAILURE);
        }

        // Lire le registre ACCESS_CONFIG
        value = i2c_smbus_read_byte_data(fd, DS1621_ACCESS_CONFIG);
        if (value < 0) {
                perror("Read ACCESS_CONFIG");
                exit(EXIT_FAILURE);
        }

        value |= DS1621_ONE_SHOT;

        // Réécrire ACCESS_CONFIG
        if (i2c_smbus_write_byte_data(fd, DS1621_ACCESS_CONFIG, value) < 0) {
                perror("Write ACCESS_CONFIG");
                exit(EXIT_FAILURE);
        }

        // Démarrer l'acquisition
        if (i2c_smbus_write_byte(fd, DS1621_START_CONVERT) < 0) {
                perror("Write START_CONVERT");
                exit(EXIT_FAILURE);
        }
        // Attendre qu'une acquisition soit terminée
        do {
                value = i2c_smbus_read_byte_data(fd, DS1621_ACCESS_CONFIG);
                if (value < 0) {
                        perror("Read ACCESS_CONFIG");
                        exit(EXIT_FAILURE);
                }
        } while ((value & DS1621_DONE) == 0);

        // Lire la température
        value = i2c_smbus_read_byte_data(fd, DS1621_READ_TEMPERATURE);
        if (value < 0) {
                perror("Read TEMPERATURE");
                exit(EXIT_FAILURE);
        }

        if (value > 127)
                value -= 256;
        fprintf(stdout, "%d\n", value);

        return EXIT_SUCCESS;
}
```

On peut choisir de le *cross-compiler* sur un PC, puis de transférer le code exécutable sur la cible, à l'instar des pratiques usuelles en programmation embarquée. Il est sans doute plus simple de le compiler directement sur le Raspberry Pi, car la distribution Raspbian, comme la plupart des autres, inclut un compilateur **gcc**.

Terminal

```
# gcc lecture-temperature.c -o lecture-temperature -Wall
# ./lecture-temperature
26
```

CONCLUSION

Nous avons réussi à communiquer facilement avec un périphérique par l'intermédiaire du bus i²c présent sur le Raspberry Pi, en utilisant trois langages différents. Nous avons employé un composant DS1621 pour sa facilité de mise en œuvre, mais il existe de très nombreux périphériques que l'on peut piloter avec l'i²c (capteurs en tous genres, afficheurs, télécommandes, etc.) et la simplicité de la connexion rend aisée l'expérimentation pour le hacker amateur. ▮

POUR EN SAVOIR PLUS

La page i²c de Wikipédia en langue française (**http://fr.wikipedia.org/wiki/I2C**) est claire et complète.

La description du protocole SMBus sur son site officiel (**http://smbus.org**) est un peu ardue, mais on en trouve de nombreux résumés sur le Web.

La documentation technique du DS1621 (**http://datasheets.maximintegrated.com/en/ds/DS1621.pdf**) courte, complète, est facilement compréhensible, ce qui n'est pas souvent le cas dans les documentations de composants électroniques !

BUS I²C ET NUNCHUNK

Une autre utilisation amusante du bus i²c est la communication avec un « nunchunk », extension de manette de jeu pour la console Wii, qui regroupe un joystick analogique, deux boutons et un accéléromètre trois axes. Il s'agit en effet d'un périphérique i²c SMBus tout à fait standard. Il est assez facile de glisser des petits câbles fins à l'extérieur des languettes du connecteur (au format propriétaire), permettant ainsi une connexion temporaire sans avoir besoin de couper le câble. On notera que les deux broches centrales ne sont pas utilisées. Le nunchunk s'alimente en principe en +3V, mais j'ai constaté un fonctionnement plus fiable en le branchant sur le +5V (fourni par le Raspberry Pi).

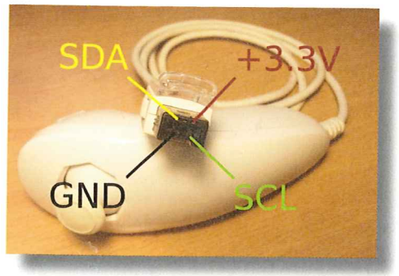


Fig. 7 : Branchements du Nunchunk

Il existe deux modèles principaux de nunchunks : le blanc et le noir, qui s'installent tous deux à l'adresse **0x52** mais utilisent des séquences d'initialisation différentes. Pour le nunchunk blanc, il faut écrire un **0x00** dans le registre **0x40**, tandis que pour le nunchunck noir, on doit écrire un **0x55** dans le registre **0xF0** suivi d'un **0x00** dans le registre **0xFB**.

Après quelques millisecondes d'initialisation, on pourra interroger périodiquement le nunchunk en lui envoyant d'abord un octet **0x00** pour demander une acquisition des capteurs, puis en lisant six octets décrits dans le tableau suivant :

Octet	Signification
1	Composante X du joystick (0-255)
2	Composante Y du joystick (0-255)
3	8 bits de poids fort de la composante X de l'accéléromètre
4	8 bits de poids fort de la composante Y de l'accéléromètre
5	8 bits de poids fort de la composante Z de l'accéléromètre
6	État des boutons (2 bits) + 2 bits de poids faible de chacune des composantes de l'accéléromètre

...SUITE BUS I²C ET NUNCHUNK

Comme on le voit, les composantes de l'accéléromètre sont fournies sur 10 bits (0-1023).
Le programme suivant permet d'afficher en permanence les différentes valeurs :

Fichier

```
#! /usr/bin/python

import smbus
import sys
import time

try:
        bus = smbus.SMBus(1)
except IOError:
        print "Please load i2C_bcm2708 and i2c_dev modules"
        sys.exit(1)

# Initialiser le nunchunk (adresse = 0x52)
# Écrire 0x00 dans le registre 0x40 pour le nunchunk blanc
# Écrire 0x55 dans le registre 0xF0 puis 0x00 dans 0xFB pour le noir
try:
        bus.write_byte_data(0x52, 0x40, 0x00)
except IOError:
        print "No nunchunk found"
        sys.exit(1)

# Attendre l'initialisation
time.sleep(0.01)

while True:
    try:
        # Envoyer un 0x00 pour demander l'acquisition
        bus.write_byte(0x52,0x00)
        # Attendre quelques millisecondes
        time.sleep(0.05)
        joystick_x      =  bus.read_byte(0x52)
        joystick_y      =  bus.read_byte(0x52)
        accelerometer_x =  bus.read_byte(0x52)
        accelerometer_y =  bus.read_byte(0x52)
        accelerometer_z =  bus.read_byte(0x52)
        miscellaneous   =  bus.read_byte(0x52)
        accelerometer_x = accelerometer_x << 2
        accelerometer_y = accelerometer_y << 2
        accelerometer_z = accelerometer_z << 2
        accelerometer_x += (miscellaneous & 0x0C) >> 2
        accelerometer_y += (miscellaneous & 0x30) >> 4
        accelerometer_z += (miscellaneous & 0xC0) >> 6
        if ((miscellaneous & 0x03) == 0):
            button_c = 0
            button_z = 1
        elif ((miscellaneous & 0x03) == 1):
            button_c = 1
            button_z = 0
        elif ((miscellaneous & 0x03) == 2):
            button_c = 1
            button_z = 1
        else:
            button_c = 0
            button_z = 0
        sys.stdout.write('Joystick: (%3d, %3d), ' % (
              joystick_x, joystick_y))
        sys.stdout.write('Accelerometer: (%4d, %4d, %4d), ' %(
              accelerometer_x, accelerometer_y, accelerometer_z))
        sys.stdout.write('Buttons: (%d,%d)\n'     % (
                    button_c, button_z))
    except KeyboardInterrupt:
        sys.exit(0)
    except IOError:
        print "Nunchunk disconnected"
        sys.exit(1)
```

4 APPLICATIONS

RASPBERRY PI ET TEMPS RÉEL

Christophe Blaess

La notion de temps réel est source de nombreuses controverses dans le monde du développement informatique. Je commencerai donc par qualifier les différentes classes de systèmes temps réel que l'on considère habituellement. Nous pourrons alors voir quelles sont les différentes solutions utilisables pour le Raspberry Pi et les niveaux de qualité que nous pouvons en attendre.

1. CONCEPT DE TEMPS RÉEL

La notion de temps réel fait référence à la possibilité pour un système de répondre à des stimuli extérieurs en prenant en compte précisément l'écoulement du temps, ceci indépendamment du flux d'instructions traité par le processeur. J'ai l'habitude de présenter ce concept en disant qu'un système est soumis à des contraintes temps réel si, lorsqu'il répond à un événement extérieur, l'instant auquel il parvient à fournir sa réponse est à prendre en considération dans la validité de celle-ci.

Les polémiques habituelles sur le temps réel tiennent essentiellement à la précision de la limite et aux tolérances qu'on accorde lors de sa prise en compte. Face à la multitude et la diversité de systèmes prétendant répondre aux critères du temps réel, j'ai choisi de les caractériser en définissant quatre catégories là où généralement on n'en prend que deux en considération (*hard realtime* et *soft realtime*).

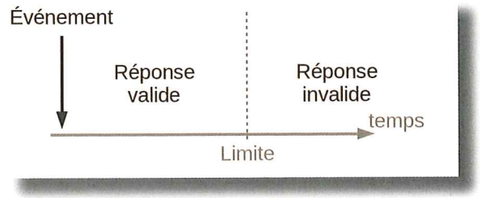


Fig. 1 : Notion de contrainte temps réel

1.1 Temps réel absolu

Dans un système temps réel **absolu**, les temps de réponse aux stimuli externes sont parfaitement connus et stables. Lorsqu'un événement se produit, le système saura y apporter une réponse dans un temps toujours identique, parfaitement **prédictible**, avec des fluctuations infimes (liées aux variations thermiques des composants par exemple) de l'ordre de la nanoseconde ou dizaine de nanosecondes.

Il s'agit du temps réel que peuvent proposer des systèmes entièrement électroniques sans aspects logiciels. Toute la logique est câblée et les temps de réponse sont des durées de propagation de signaux et de commutations électriques au sein des composants. Inutile d'espérer atteindre ce niveau de prédictibilité dans les temps de réponse du Raspberry Pi, ni d'ailleurs dans ceux d'autres cartes à microprocesseur.

Les seuls systèmes programmables et modifiables susceptibles de répondre à ce type de contrainte temps réel sont les FPGA, CPLD, PAL, etc., qui implémentent des arrangements configurables de composants logiques.

1.2 Temps réel strict certifiable

Avec un système apte à répondre à des contraintes de temps réel **strict** (*hard realtime*) **certifiables** (on parle également de temps réel **dur**), de légères fluctuations (de l'ordre de la centaine de nanosecondes ou de la microseconde) pourront intervenir dans les temps de réponse aux événements extérieurs. Ces fluctuations seront très faibles par rapport aux durées considérées, et elles seront **bornées**. S'il n'est pas possible de prédire exactement un temps de réponse absolu, on peut néanmoins savoir quelle sera sa valeur maximale. On peut, à l'étude du code applicatif, connaître la durée maximale d'exécution d'une portion de code, quoi qu'il se passe en dehors du système. C'est cet aspect qui permet à ce type d'application d'être employée dans des environnements où des certifications de sécurité sont réclamées (avionique, ferroviaire, automobile, médical, etc.).

Les systèmes répondant aux contraintes du temps réel strict certifiable sont généralement conçus en employant des microcontrôleurs. Ces systèmes sont mono-tâches ou ne comportent qu'un nombre limité (et figé dès la conception) de tâches.

La possibilité de réaliser des systèmes temps réel strict en employant des microprocesseurs modernes est sujet à controverse. De nombreux sous-systèmes de ces microprocesseurs limitent la prévisibilité des temps de réponse : MMU, caches mémoire, pipelines d'instructions, protocoles de synchronisation de caches dans les systèmes multicœurs, réduction de la consommation d'énergie, etc.

Il existe quelques systèmes d'exploitation visant ce type de performances. Pour la plupart, ils fonctionnent sur des microcontrôleurs et ne fournissent qu'un minimum de fonctionnalités, essentiellement liées au multitâche. Ces systèmes (dont le nom tourne souvent autour de **RTOS**, *Real-Time Operating System*) sont généralement propriétaires. On notera que la notion de système d'exploitation est limitée ici à son strict minimum, l'essentiel de la gestion des ressources (mémoire, périphériques, communications) étant statique, définie à l'initialisation et figée pour la suite.

Dans les systèmes temps réel strict certifiables capables de tirer parti des fonctionnalités d'un microprocesseur, nous pouvons citer **RTEMS** (*Real Time Executive for Multiprocessor Systems*). Ce dernier a été développé initialement pour l'armée américaine (ce qui explique que le « M » de son acronyme ait représenté successivement les mots « Missile » puis « Military ») et est distribué aujourd'hui librement sous une licence dérivée de la GPL.

On peut faire fonctionner RTEMS sur un Raspberry Pi même si cette plateforme n'est pas complètement supportée. On se reportera pour en savoir plus à l'article de Pierre Ficheux dans ce même numéro.

1.3 Temps réel strict non-certifiable

Les supports temps réel **stricts non-certifiables** sont des implémentations, dans de véritables systèmes d'exploitation, de mécanismes d'ordonnancement visant à **approcher** au mieux le temps réel strict.

Les systèmes considérés étant généralement multitâches (avec des tâches indépendantes les unes des autres), voire multi-utilisateurs, des mécanismes d'isolation mémoire (MMU) et de communication entre tâches sont proposés par le système.

La plupart du temps, il n'y a pas de vraies limites garanties pour les temps de réponse. De nombreux systèmes fonctionnent avec des notions de priorités entre tâches et garantissent qu'un traitement de priorité élevée ne sera pas interrompu par une tâche de priorité moindre.

Le noyau temps réel du système d'exploitation est prévu pour être le plus déterministe possible dans ses traitements. Les gestionnaires d'interruptions utilisés par le système sont les plus brefs possible.

Ce niveau de temps réel peut être obtenu sur un Raspberry Pi en utilisant **Xenomai** [1], comme nous le verrons plus loin. Le patch **`PREEMPT_RT`** [2] dont nous parlerons également permet d'approcher ce type de performances.

Sur une plateforme de type PC, on peut utiliser le projet **RTAI** (*Real-Time Application Interface*), qui ne supporte pas (encore) le Raspberry Pi.

1.4 Temps réel souple

Le temps réel **souple** (*soft realtime*) est une organisation sous forme de priorités entre les tâches applicatives. À chaque invocation de l'ordonnanceur, celui-ci choisit la tâche applicative dont la **priorité** est la plus élevée parmi toutes les tâches prêtes. Elle ne pourra être préemptée (interrompue) que pour laisser la place à une tâche de priorité plus élevée qui vient de se réveiller. Ce type d'ordonnancement est appelé « Fifo ». Avec l'ordonnancement nommé « Round Robin », on peut en outre avoir une rotation entre les tâches de même niveau de priorité.

Ce qui limite les performances du temps réel souple, ce sont les traitements – parfois longs, comme ceux des protocoles réseau, des systèmes de fichiers, etc. – réalisés directement par le système au détriment des tâches applicatives. Sur un système Linux avec un noyau standard sans extension (celui que l'on nomme le noyau *Vanilla*), un « ping » en provenance d'une machine distante sera immédiatement traité par le kernel (dans un gestionnaire d'interruptions, puis une *tasklet*), même s'il a interrompu une tâche considérée comme très prioritaire, retardant celle-ci d'autant.

1.5 Récapitulatif

En résumé, les quatre catégories que j'ai décrites ci-dessus sont caractérisées dans le tableau suivant :

Type de temps réel	Temps de réponse	Exemple avec Raspberry Pi
Absolu	Fixés	Non
Strict certifiable	Fluctuants mais bornés	RTEMS
Strict non-certifiable	Non garantis, mais indépendants de toute activité moins prioritaire du système.	Xenomai Linux **`PREEMPT_RT`**
Souple	Non garantis, mais indépendants des activités moins prioritaires en espace utilisateur.	Linux Vanilla

2. IMPLÉMENTATIONS POUR RASPBERRY PI

Pierre Ficheux ayant traité du portage de RTEMS sur Raspberry Pi, je vais me consacrer aux trois autres solutions : le noyau Linux Vanilla, le kernel modifié par le patch **`PREEMPT_RT`** et le système Xenomai. Je les traiterai par qualités croissantes du temps réel proposé.

2.1 Temps réel souple avec Linux

Le noyau Vanilla permet de modifier la priorité et le mode d'ordonnancement d'une tâche. Ceci grâce à l'appel système **`sched_setscheduler()`**, mais plus simplement avec la commande **`chrt`** depuis le shell.

L'appel **`chrt -f 90 ./commande`** permet de lancer la commande sous un ordonnancement Fifo (la tâche ne peut être préemptée que par une autre tâche de priorité strictement supérieure), avec la priorité 90 (sur une échelle allant de 1 à 99). De même, **`chrt -r 50 ./commande`** lance la commande avec un ordonnancement Round-Robin (après un certain temps d'exécution, la tâche peut être préemptée par une autre de même priorité) et la priorité 50.

Précisons qu'il existe depuis quelques mois une autre catégorie d'ordonnancement temps réel : **EDF** (*Earliest Deadline First*) où l'on réserve une partie du temps CPU disponible pour une exécution périodique de la tâche. Cet ordonnancement est implémenté dans le noyau Linux depuis sa version 3.14, mais pas encore supporté par les utilitaires de la Raspbian. La qualité de son comportement temps réel est sensiblement identique à celle de Fifo et Round-Robin.

Lorsqu'une tâche temps réel s'exécute sans discontinuer pendant plusieurs centaines de millisecondes, le noyau est susceptible de l'interrompre quelques instants pour laisser s'exécuter des tâches non temps réel. Ceci afin d'éviter qu'une tâche ne boucle indéfiniment en bloquant tout le processeur. Ce comportement est un peu surprenant, et peut sembler indésirable dans certains cas. Pour désactiver cette option du noyau, il faut exécuter (par exemple dans un script de démarrage du système) :

Terminal

```
echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

2.2 Temps réel classique avec PREEMPT_RT

Le patch **PREEMPT_RT** regroupe un ensemble de modifications que l'on peut apporter sur un noyau Linux Vanilla pour améliorer son comportement temps réel. Par exemple, il comprend un mode de fonctionnement « Fully Preemptible », où une tâche s'exécutant dans l'espace utilisateur est susceptible, si elle se réveille suite à l'arrivée d'une interruption, de préempter immédiatement une tâche moins prioritaire même si cette dernière exécutait du code dans l'espace noyau. L'amélioration consiste ici à réduire très sensiblement le délai de préemption par rapport à un noyau standard.

Une autre modification visible consiste à « threader » les gestionnaires d'interruptions, autrement dit à les exécuter dans des tâches (avec les privilèges du kernel) dont la priorité n'est plus infinie, mais au contraire modifiable par l'administrateur avec la commande **chrt**. Les *threaded interrupts* sont visibles dans les résultats de la commande **ps aux** sous forme **[36/irq-timer]** indiquant le numéro et le nom de l'interruption (que l'on peut retrouver dans **/proc/interrupts**). Leur priorité par défaut est 50 sous un ordonnancement Fifo. Une tâche de priorité 51 sera donc plus prioritaire que les traitements d'interruption.

Le patch **PREEMPT_RT** évolue régulièrement. Comme il s'applique sur les sources du noyau Vanilla, il est maintenu et disponible pour la plupart des versions stables du kernel.

2.3 Approche du temps réel strict

Avec le projet Xenomai, on atteint les performances du temps réel strict, tout en continuant de bénéficier des services d'un système d'exploitation très riche. Xenomai s'appuie sur une couche nommée **ipipe** (*interrupt pipeline*) qui capture les interruptions provenant du matériel avant Linux. Les interruptions sont transmises tout d'abord à un premier « domaine d'exécution », composé d'un petit ordonnanceur simple nommé *nucleus*, et qui pourra activer les tâches temps réel de Xenomai. Lorsqu'il ne reste plus aucune tâche Xenomai active, **ipipe** envoie les interruptions reçues au second domaine d'exécution : le noyau Linux.

Xenomai est constitué d'un patch que l'on applique sur les sources du noyau Vanilla, et de bibliothèques permettant de programmer des tâches temps réel. En outre, des outils de mesure sont fournis pour juger des performances du système.

3. INSTALLATIONS

Nous allons tester sur un Raspberry Pi successivement le noyau Vanilla, le patch **PREEMPT_RT** et Xenomai. Nous partons d'une distribution Raspbian, sur laquelle nous installerons successivement les noyaux modifiés compilés sur un PC.

3.1 Sélection d'une version

Pour pouvoir comparer les performances des trois systèmes, il est préférable de partir d'un noyau Linux standard de même niveau pour les trois expériences. Avec **PREEMPT_RT** et Xenomai, ce noyau sera modifié par un patch que nous lui appliquerons. Il faut donc vérifier quelles sont les versions de Linux supportées par les patches de ces deux systèmes. **PREEMPT_RT** est disponible pour pratiquement toutes les versions stables de Linux, mais le patch **ipipe** pour Xenomai est plus rare. C'est donc lui que nous allons vérifier en premier.

Téléchargeons la dernière version disponible au moment de la rédaction de ces lignes :

Terminal
```
$ git clone http://git.xenomai.org/xenomai-2.6.git
Clonage dans 'xenomai-2.6'...
```


Les patches sont groupés dans un répertoire qui dépend de l'architecture. En outre, il faudra ajouter ceux spécifiques pour le Raspberry Pi.

Terminal
```
$ ls xenomai-2.6/ksrc/arch/arm/patches/
beaglebone  ipipe-core-3.4.6-arm-4.patch  ipipe-core-3.5.7-arm-6.patch
ipipe-core-3.8.13-arm-3.patch  mxc  raspberry  README  zynq
$ ls xenomai-2.6/ksrc/arch/arm/patches/raspberry/
ipipe-core-3.8.13-raspberry-post-2.patch  ipipe-core-3.8.13-raspberry-pre-2.patch
```

Le support pour Raspberry Pi est donc proposé à partir de patches pour le noyau **3.8.13**. C'est donc la version que nous sélectionnons.

Les compilations des kernels seront réalisées sur un PC par *cross-compilation*, mais on pourrait – avec beaucoup de patience – faire le travail directement sur le Raspberry Pi avec une grosse carte SD ou un disque externe (il faut compter environ 2 Go libres pour compiler le noyau Linux).

3.2 Téléchargement

Téléchargeons un premier exemplaire des sources de Linux avec le support Raspberry Pi. Cette opération dure environ une demi-heure :

Terminal
```
$ git clone http://github.com/raspberrypi/linux  rpi-kernel-vanilla
Clonage dans 'rpi-kernel-vanilla'...
[...]
Checking out files: 100% (44957/44957), done.
```

Le répertoire **rpi-kernel-vanilla** contient l'ensemble des sources avec l'historique Git. Sélectionnons la bonne version du noyau :

Terminal
```
$ cd rpi-kernel-vanilla/
[rpi-kernel-vanilla]$ git checkout rpi-3.8.y
Checking out files: 100% (23274/23274), done.
La branche rpi-3.8.y est paramétrée pour suivre la branche distante rpi-3.8.y depuis origin.
Basculement sur la nouvelle branche 'rpi-3.8.y'
```

Nous sommes sur la branche stable 3.8, vérifions la version exacte :

Terminal
```
[rpi-kernel-vanilla]$ head -3 Makefile
VERSION = 3
PATCHLEVEL = 8
SUBLEVEL = 13
```

3.8.13, c'est parfait ! Dupliquons ce répertoire en deux autres exemplaires, afin de pouvoir appliquer les patches séparément sur les sources originales :

Terminal
```
[rpi-kernel-vanilla]$ cd ..
$ cp -R rpi-kernel-vanilla/ rpi-kernel-preempt
$ cp -R rpi-kernel-vanilla/ rpi-kernel-xenomai
```

3.3 Compilation et installation du noyau Vanilla

Pour le noyau standard Vanilla, aucune modification n'est à apporter ; nous allons simplement sélectionner une configuration pour le Raspberry Pi.

Terminal
```
$ cd rpi-kernel-vanilla/
[rpi-kernel-vanilla]$ make ARCH=arm bcmrpi_defconfig
...
# configuration written to .config
```

Si vous le souhaitez, vous pouvez examiner la configuration avec :

Terminal
```
[rpi-kernel-vanilla]$ make ARCH=arm menuconfig
```

Nous lançons à présent la compilation. J'ai installé sur mon PC de travail une chaîne de compilation croisée obtenue avec Buildroot, mais on peut utiliser n'importe quelle autre, pourvu qu'elle fonctionne pour le Raspberry Pi.

Sur ma machine, la chaîne de cross-compilation est installée ainsi :

Terminal
```
[rpi-kernel-vanilla]$ ls /usr/cross/rpi/usr/bin/arm-buildroot-linux-
gnueabi-*
/usr/cross/rpi/usr/bin/arm-buildroot-linux-gnueabi-addr2line
/usr/cross/rpi/usr/bin/arm-buildroot-linux-gnueabi-ar
/usr/cross/rpi/usr/bin/arm-buildroot-linux-gnueabi-as
...
/usr/cross/rpi/usr/bin/arm-buildroot-linux-gnueabi-gcc
...
```

J'ajoute le chemin d'accès à cette chaîne de compilation dans le PATH :

Terminal
```
[rpi-kernel-vanilla]$ PATH="$PATH:/usr/cross/rpi/usr/bin/"
```

J'indique dans la variable **CROSS_COMPILE** le préfixe à ajouter avant les noms standards des outils de compilation (attention à ne pas oublier le tiret final !) :

Terminal
```
[rpi-kernel-vanilla]$ export CROSS_COMPILE=arm-buildroot-linux-gnueabi-
```

Puis, je lance la compilation du noyau :

Terminal
```
[rpi-kernel-vanilla]$ make ARCH=arm
scripts/kconfig/conf --silentoldconfig Kconfig
  WRAP    arch/arm/include/generated/asm/auxvec.h
  ...
  LD      arch/arm/boot/compressed/vmlinux
  OBJCOPY arch/arm/boot/zImage
  Kernel: arch/arm/boot/zImage is ready
```

Le noyau est compilé, ainsi que ses modules. Regroupons ces derniers dans une arborescence facile à copier vers le Raspberry Pi :

Terminal
```
[rpi-kernel-vanilla]$ mkdir target
[rpi-kernel-vanilla]$ make ARCH=arm INSTALL_MOD_PATH=target/ modules_
install
  INSTALL arch/arm/mach-bcm2708/dmaer_master.ko
  INSTALL arch/arm/oprofile/oprofile.ko
...
  INSTALL target//lib/firmware/yam/9600.bin
  DEPMOD  3.8.13+
```


Je crée une petite archive regroupant tous les modules afin de les transférer aisément vers la cible :

Terminal
```
[rpi-kernel-vanilla]$ cd target/lib/modules
[modules]$ tar -cf 3.8.13.tar 3.8.13+/
[modules]$ cd ../../..
```

Pour installer le nouveau noyau et ses modules sur le Raspberry Pi, je les copie par le réseau via SSH. Mon Raspberry Pi se trouve à l'adresse **192.168.3.123**, il faudra évidemment adapter la commande à votre cas.

Terminal
```
[rpi-kernel-vanilla]$ scp arch/arm/boot/zImage root@192.168.3.123:/boot/
kernel-vanilla.img
root@192.168.3.123's password:
zImage                        100% 2885KB   2.8MB/s   00:0
[rpi-kernel-vanilla]$ scp target/lib/modules/3.8.13.tar
root@192.168.3.123:/lib/modules/
root@192.168.3.123's password:
3.8.13.tar                    100%   30MB   5.0MB/s   00:06
```

Je vais décompresser l'archive qui contient les modules du noyau en me connectant en SSH sur le Raspberry Pi :

Terminal
```
[rpi-kernel-vanilla]$ ssh root@192.168.3.123
root@192.168.3.123's password:
root@raspberrypi:~# cd /lib/modules/
root@raspberrypi:/lib/modules# tar xf 3.8.13.tar
```

Le noyau est copié sur la partition de *boot* du Raspberry Pi, mais il faut le renommer pour qu'il soit pris en considération par le bootloader. Je conserve par précaution l'ancienne version du noyau :

Terminal
```
root@raspberrypi:/lib/modules# cd /boot/
root@raspberrypi:/boot# ls
LICENSE.oracle bootcode.bin cmdline.txt config.txt fixup.dat
fixup_cd.dat fixup_x.dat issue.txt kernel-vanilla.img
kernel.img start.elf start_cd.elf start_x.elf
root@raspberrypi:/boot# cp kernel.img kernel-backup.img
root@raspberrypi:/boot# cp kernel-vanilla.img kernel.img
```

Je peux à présent tester le noyau fraîchement compilé :

Terminal
```
root@raspberrypi:/boot# reboot
```

Après re-démarrage et connexion sur le Raspberry Pi, je vérifie le numéro de version du noyau :

Terminal
```
root@raspberrypi:~# uname -a
Linux raspberrypi 3.8.13+ #1 PREEMPT Mon Sep 1 10:44:49 CEST 2014 armv6l
GNU/Linux
```

Attention aux confusions : le mot-clé **PREEMPT** affiché par **uname -a** indique que le noyau a été compilé en mode préemptible basique. Ceci n'est pas équivalent au patch **PREEMPT_RT** qui améliore très nettement cette préemptibilité.

3.4 Compilation et installation d'un noyau PREEMPT_RT

Avant de nous livrer aux expériences sur les performances du temps réel, nous pouvons commencer par installer les deux autres environnements, en commençant par l'application du patch **PREEMPT_RT**. Téléchargeons-le :

Terminal
```
$ cd rpi-kernel-preempt/
[rpi-kernel-preempt]$ wget https://www.kernel.org/pub/linux/kernel/
projects/rt/3.8/older/patch-3.8.13-rt16.patch.xz
[rpi-kernel-preempt]$ xz - d patch-3.8.13-rt16.patch.xz
[rpi-kernel-preempt]$ ls patch*
patch-3.8.13-rt16.patch
```

On peut, bien sûr, choisir la version avec un navigateur à partir de l'adresse **https://www.kernel.org/pub/linux/kernel/projects/rt/**.

On applique le patch pour modifier les sources du noyau :

Terminal
```
[rpi-kernel-preempt]$ patch -p1 < patch-3.8.13-rt16.patch
patching file Documentation/hwlat_detector.txt
patching file Documentation/kernel-parameters.txt
...
patching file drivers/misc/Kconfig
patching file drivers/misc/Makefile
Hunk #1 FAILED at 49.
1 out of 1 hunk FAILED
1 out of 1 hunk FAILED -- saving rejects to file drivers/misc/Makefile.rej
patching file drivers/misc/hwlat_detector.c
...
patching file scripts/mkcompile_h
```

Nous voyons une petite erreur dans l'application du patch sur un fichier **Makefile**. Ceci n'a pas d'importance dans notre cas, mais on pourrait éventuellement intervenir manuellement pour faire la modification notifiée dans le fichier **.rej** (l'ajout du fichier **hwlat_detector** en l'occurrence).

En pratique, je conseille d'appeler la commande **patch** une première fois avec l'option **--dry-run** pour faire un passage à vide en vérifiant les éventuelles erreurs, puis de la rappeler sans l'option pour faire effectivement les modifications.

Après sélection d'une configuration par défaut pour Raspberry Pi, nous faisons une petite modification :

Terminal
```
[rpi-kernel-preempt]$ make ARCH=arm bcmrpi_defconfig
...
[rpi-kernel-preempt]$ make ARCH=arm menuconfig
```

Dans le menu « Kernel features », pour l'option « Preemption model », sélectionnez la configuration « Fully Preemptible Kernel (RT) ».

Le reste de la compilation est identique au noyau Vanilla. Une fois celle-ci terminée et le **make modules_install** réalisé, pour le transfert vers la cible les noms de fichiers sont un peu modifiés :

Terminal
```
[rpi-kernel-preempt]$ cd target/lib/modules/
[modules]$ ls
3.8.13-rt16+
[modules]$ tar -cf 3.8.13-rt16.tar 3.8.13-rt16+/
[modules]$ cd ../../..
[rpi-kernel-preempt]$ scp target/lib/modules/3.8.13-rt16.tar
root@192.168.3.123:/lib/modules/
root@192.168.3.123's password:
3.8.13-rt16.tar                                100%   30MB   4.3MB/s   00:07
[rpi-kernel-preempt]$ scp arch/arm/boot/zImage root@192.168.3.123:/boot/
kernel-rt.img
root@192.168.3.123's password:
zImage                                         100% 2873KB   2.8MB/s   00:01
[rpi-kernel-preempt]$ ssh root@192.168.3.123
root@192.168.3.123's password:
root@raspberrypi:~# cd /lib/modules/
root@raspberrypi:/lib/modules# tar xf 3.8.13-rt16.tar
root@raspberrypi:/lib/modules# cd /boot/
root@raspberrypi:/boot# cp kernel-rt.img kernel.img
```

Pour que le système boote correctement sur un noyau **PREEMPT_RT**, il est nécessaire d'éditer sur le Raspberry Pi le fichier **/boot/cmdline.txt** pour y ajouter l'option (par exemple en début de ligne) **sdhci-bcm2708.enable_llm=0**.

Après redémarrage du Raspberry Pi, nous voyons l'option **RT** dans la commande **uname -a** :

Terminal
```
root@raspberrypi:~# uname -a
Linux raspberrypi 3.8.13-rt16+ #2 PREEMPT RT Mon Sep 1 15:04:57 CEST 2014
armv6l GNU/Linux
```

3.5 Compilation et installation d'un système Xenomai

Xenomai repose sur deux choses :

- un noyau Linux standard sur lequel on applique un patch pour intégrer **ipipe**,
- des bibliothèques pour la compilation (et éventuellement l'exécution) des tâches.

Nous devons appliquer trois patches sur le noyau (deux spécifiques au Raspberry Pi, et un générique pour l'architecture ARM). Puis, nous allons nous aider du script **prepare-kernel.sh** fourni avec Xenomai pour préparer la configuration.

Terminal
```
$ cd rpi-kernel-xenomai/
[rpi-kernel-xenomai]$ patch -p1 < ../xenomai-2.6/ksrc/arch/arm/patches/raspberry/ipipe-
core-3.8.13-raspberry-pre-2.patch
patching file kernel/trace/ftrace.c
[rpi-kernel-xenomai]$ patch -p1 < ../xenomai-2.6/ksrc/arch/arm/patches/ipipe-core-
3.8.13-arm-3.patch
...
[rpi-kernel-xenomai]$ patch -p1 < ../xenomai-2.6/ksrc/arch/arm/patches/raspberry/ipipe-
core-3.8.13-raspberry-post-2.patch
[rpi-kernel-xenomai]$ ../xenomai-2.6/scripts/prepare-kernel.sh --linux=.    --ipipe=../
xenomai-2.6/ksrc/arch/arm/patches/ipipe-core-3.8.13-arm-3.patch --arch=arm
[rpi-kernel-xenomai]$ make ARCH=arm bcmrpi_defconfig
[rpi-kernel-xenomai]$ make ARCH=arm menuconfig
```

Dans l'interface de configuration du noyau, deux options devront être modifiées :

- Dans le menu « Kernel Features », désactivez l'option « Enable -fstack-protector buffer overflow detection » ;
- Dans le menu « Kernel hacking », désactivez l'option « KGDB: kernel debugger ---> ».

La compilation et l'installation se feront comme précédemment. Après avoir redémarré sur le nouveau noyau, nous voyons deux nouvelles entrées dans `/proc` :

Terminal
```
root@raspberrypi:~# uname -a
Linux raspberrypi 3.8.13-ipipe+ #1 PREEMPT Tue Sep 2 00:58:06 CEST 2014
armv6l GNU/Linux
root@raspberrypi:~# ls /proc/ipipe/
Linux version Xenomai
root@raspberrypi:~# ls /proc/xenomai/
acct   faults  interfaces  latency   rtdm   schedclasses  timebases  timerstat
apc    heap    irq         registry  sched  stat          timer      version
```

Ceci nous fournit la partie « kernel » de Xenomai, mais il faut également disposer de la partie applicative (bibliothèques pour compiler et exécuter nos programmes, outils de tests, etc.). Pour compiler tout ceci, nous devons retourner dans le répertoire des sources de Xenomai. La variable d'environnement `PATH` doit à nouveau être configurée pour contenir le répertoire du *cross-compiler*.

Terminal
```
$ cd xenomai-2.6
[xenomai-2.6]$ ./configure --host=arm-buildroot-linux-gnueabi CFLAGS='-
march=armv6' LDFLAGS='-march=armv6' --enable-shared=no
...
[xenomai-2.6]$ make
...
[xenomai-2.6.$ make DESTDIR=$(pwd)/target install
...
```

On peut remarquer que j'ai ajouté l'option `--enable-shared=no` qui construira les outils de tests que nous emploierons avec une édition statique des liens. Ceci nous évite d'avoir à installer les bibliothèques dynamiques sur la cible.

Ainsi, je ne vais transférer sur le Raspberry que les répertoires contenant les exécutables :

Terminal
```
[xenomai-2.6]$ cd target/
[target]$ tar cf xenomai-user.tar usr/xenomai/bin/ usr/xenomai/sbin/
[target]$ scp xenomai-user.tar root@192.168.3.123:/
[target]$ ssh root@192.168.3.123
root@192.168.3.123's password:
root@raspberrypi:~# cd /
root@raspberrypi:/# tar xf xenomai-user.tar
```

4. OUTILS DE MESURE

Il existe de nombreux outils pour mesurer la qualité d'un système temps réel. Certains s'intéressent aux temps de commutation entre tâches, d'autres à la précision des *timers*, à la latence des interruptions, aux temps de prise d'un mutex, etc. Dans le cadre de cet article, j'ai choisi d'utiliser un outil bien connu et dont les résultats sont assez représentatifs du comportement temps réel global d'un système : `cyclictest`.

Initialement écrit par Thomas Gleixner, ce programme est maintenant intégré dans la suite `rt-tests` maintenue par Clark Williams. Il en existe également une version incorporée dans les outils de tests de Xenomai, ce qui nous permettra d'avoir une base uniforme de mesures entre nos systèmes.

Le principe de `cyclictest` est de vérifier la précision des déclenchements de tâches périodiques. Il programme une tâche qui doit être réveillée toutes les millisecondes. Lors de son activation, elle vérifie l'heure système et compare le temps écoulé depuis le dernier réveil et la période prévue. Après un nombre conséquent de déclenchements, on a un bon aperçu du comportement temps réel du système pour ce qui concerne un traitement périodique.

Compilons `cyclictest` sur le Raspberry Pi :

Terminal
```
root@raspberrypi:~# git clone git://git.kernel.org/pub/scm/linux/kernel/
git/clrkwllms/rt-tests.git
Cloning into 'rt-tests'
...
root@raspberrypi:~# cd rt-tests/
root@raspberrypi:~/rt-tests# make
...
root@raspberrypi:~/rt-tests# ls -l cyclictest
-rwxr-xr-x 1 root root 50683 Sep  6 02:27 cyclictest
```

Il y a plusieurs options de `cyclictest` qui nous intéressent :

`--duration`	Durée du test, avec un suffixe `m` (minutes), `h` (heures), voire `d` (jours).
`--quiet`	Ne rien afficher pendant l'exécution, seulement un compte-rendu à la fin (utile si on redirige le résultat dans un fichier.
`--mlockall`	S'assurer avant le démarrage du test que tout le code exécutable est chargé en mémoire physique et ne la quittera pas (indispensable en temps réel).
`--latency=0`	Écrire `0` dans `/dev/cpu_dma_latency` pour empêcher le processeur de s'endormir dans des sommeils plus profonds que *C0*.
`--policy=fifo`	Choisir un ordonnancement `other` (non temps réel) `fifo` ou `rr` (*round robin*) pour l'exécution du test. Les ordonnancements `fifo` et `rr` réclament les droits *root*.
`--priority=99`	Fixer une priorité temps réel de `99` (la plus élevée) pour les ordonnancements `fifo` ou `rr`. Pour l'ordonnancement `other`, seule la priorité `0` est possible.
`--nanosleep`	Par défaut `cyclictest` utilise des timers basés sur les signaux Unix avec `setitimer()`. Avec cette option, il emploiera plutôt une boucle autour de `clock_nanosleep()`.
`--system`	Combinée avec la précédente, cette option réclame d'utiliser l'appel système `nanosleep()` plutôt que `clock_nanosleep()` qui n'est pas toujours disponible.
`--histogram=10000`	Afficher à la fin de l'exécution un histogramme des latences mesurées jusqu'à 10000 microsecondes. Ceci nous sera très utile pour comparer les comportements.

Bien sûr, pour réaliser une série complète de mesures, j'utilise un script qui permet de tester successivement :

- les ordonnancements `other`, `fifo`, `rr`,
- et pour chacun d'eux, les fonctionnements avec `setitimer()`, `nanosleep()` et `clock_nanosleep()`.

Le script est exécuté (il dure quinze heures) successivement sur un noyau Vanilla, puis sur un noyau `PREEMPT_RT`. Il est disponible sur le dépôt GLMF.

4

À noté

Pour charger efficacement un système, j'ai l'habitude de procéder avec plusieurs scripts spécifiques qui solliciteront intensivement différents sous-systèmes du noyau. Par exemple, l'ordonnanceur (des dizaines ou centaines de processus en parallèle), le *Virtual File System* (des parcours incessants des fichiers de l'arborescence), le sous-système *Block* (des lectures/écritures volumineuses et aléatoirement réparties sur des périphériques blocs), la pile de protocole réseau (des transferts de données par TCP/IP et UDP/IP), les gestionnaires d'interruptions (avec un `ping` en mode *flood* depuis un autre poste ou un générateur basses fréquences connecté à une interruption GPIO).

En outre, j'aime bien que mes différents scripts s'exécutent périodiquement, avec des temps de repos pour le système. En effet, les perturbations les plus importantes surviennent souvent lors de périodes transitoires (montées en charge par exemple) plutôt que pendant les régimes permanents. Ceci est dû par exemple au temps de réveil du processeur mis ponctuellement en sommeil.

Pour éviter les effets liés à un fonctionnement cyclique, chaque script dispose d'une période de fonctionnement différente de celles des autres, et d'un rapport variable entre temps d'activité et temps de repos.

Pour que les mesures soient intéressantes et significatives, il faut qu'elles soient réalisées alors que le système est nettement plus chargé que pour son fonctionnement normal. Lorsque je dois vérifier le comportement d'un système temps réel pour un projet industriel, je fais fonctionner simultanément :

- l'application « métier » pour laquelle le système est développé (en vérifiant qu'elle tourne correctement dans toutes les circonstances),
- des outils de mesures comme `cyclictest`, qui vont vérifier les fluctuations du système (ces outils doivent être configurés pour s'exécuter avec une priorité supérieure à celle de l'application),
- des scripts de charge qui mettent le système sous pression.

Les vérifications durent habituellement plusieurs jours, si possible sur des machines différentes, afin d'obtenir au final un compte-rendu représentatif du comportement du système.

Il existe un script livré avec Xenomai dont la vocation est de forcer une charge assez importante sur un système. Son nom est explicite : `dohell` ! On peut très bien l'utiliser sur un noyau Vanilla ou `PREEMPT_RT`, puisqu'il ne fait appel qu'à des commandes système classiques (`cat`, `dd`, `nc`, `ps`, etc.).

Mon petit script `load.sh` invoque en boucle `dohell` pour des durées aléatoires de 15 à 25 secondes, suivies d'un temps de repos pour une période totale de 30 secondes. On fournit dans la variable `SERVER` l'adresse IP d'une machine du sous-réseau vers laquelle `dohell` enverra des trames TCP.

Le script s'exécute en arrière-plan. Pour l'arrêter, il suffit d'effacer le fichier `load.pid` qu'il crée dans son répertoire de lancement.

5. RÉSULTATS DES MESURES

5.1 Noyau Vanilla

Après avoir démarré le script `load.sh`, je lance le script qui invoque `cyclictest` pour toutes les situations décrites plus haut. Il s'exécute pendant quinze heures.

Terminal

```
root@raspberrypi:~# ./run-all-cyclictest.sh
Policy OTHER (1h tests)
  [08:58:25] setitimer
  [09:58:25] clock_nanosleep
  [10:58:26] nanosleep

Policy FIFO (2h tests)
  [11:58:26] setitimer
  [13:58:26] clock_nanosleep
  [15:58:26] nanosleep

Policy RR (2h tests)
  [17:58:26] setitimer
  [19:58:27] clock_nanosleep
```



```
  [21:58:27] nanosleep
root@raspberrypi:~# ls *3.8.13*
results-3.8.13+-fifo-clocknanosleep.txt
results-3.8.13+-fifo-nanosleep.txt
results-3.8.13+-fifo-setitimer.txt
results-3.8.13+-other-clocknanosleep.txt
results-3.8.13+-other-nanosleep.txt
results-3.8.13+-other-setitimer.txt
results-3.8.13+-rr-clocknanosleep.txt
results-3.8.13+-rr-nanosleep.txt
results-3.8.13+-rr-setitimer.txt
```

Chaque fichier contient 10.000 lignes comme celles-ci permettant de construire l'histogramme :

Fichier

```
000014  000005
000015  000379
000016  007943
```

Chaque ligne correspond à une classe dont l'amplitude est d'une microseconde et dont l'effectif est indiqué dans la seconde colonne. Cela signifie qu'à 5 reprises le timer était en retard de 14 microsecondes, à 379 reprises il l'était de 15 microsecondes et que 7943 fois il était retardé de 16 microsecondes.

Il y a également des lignes de commentaires et de statistiques commençant par un caractère `#` :

Terminal

```
root@raspberrypi:~# grep '^#' results-3.8.13+-rr-setitimer.txt
# /dev/cpu_dma_latency set to 0us
# Histogram
# Total: 007199162
# Min Latencies: 00024
# Avg Latencies: 00053
# Max Latencies: 19085
# Histogram Overflows: 00003
# Histogram Overflow at cycle number:
# Thread 0: 1181998 4732967 4807079
```

Ces lignes nous indiquent :

- La latence minimale est de 24 microsecondes. C'est le temps entre le déclenchement théorique du timer et l'exécution du *handler* ;
- La latence moyenne est de 53 microsecondes ce qui est tout à fait raisonnable ;
- La latence maximale (c'est la valeur qui nous intéresse dans les conceptions de systèmes temps réel) : 19,085 millisecondes ;
- Comme l'histogramme est dimensionné par une option du script de lancement avec un maximum de 10.000 microsecondes, soit 10 millisecondes, il a été dépassé à trois reprises ;
- Les numéros des cycles où l'histogramme est dépassé sont indiqués sur la dernière ligne.

La latence maximale est très médiocre, mais nous utilisons un noyau Vanilla. Espérons que les valeurs obtenues seront meilleures avec `PREEMPT_RT` et Xenomai.

Nous pouvons réaliser un histogramme des valeurs collectées. J'ai placé dans le dépôt GitHub de cet article un script nommé `draw_histogram.sh` qui crée avec `gnuplot` un fichier PNG à partir de la sortie de `cyclictest`.

L'histogramme représente en abscisse les latences par rapport au timer programmé, exprimées en microsecondes, et en ordonnée le nombre de mesures où la latence considérée a été observée. Ce graphique appelle deux remarques :

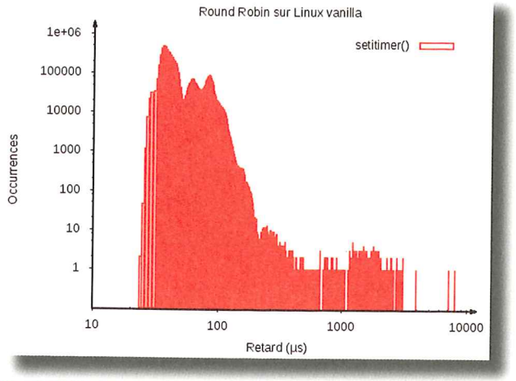


Fig. 2 : Histogramme de setitimer() en ordonnancement RR sur un système Linux Vanilla

- L'échelle des abscisses est logarithmique, car les comportements les plus intéressants se situent en-dessous de 100 microsecondes, mais il y a quand même des latences de plusieurs millisecondes. Ce mode de représentation permet d'avoir un aperçu lisible de la répartition des latences.
- L'échelle des ordonnées est également logarithmique, car certaines latences ont été rencontrées plusieurs centaines de milliers de fois (voire plusieurs millions dans les figures à venir), mais ce qui nous intéresse également ce sont les cas isolés, où une latence a été observée une seule fois (la valeur maximale par exemple).

Nous allons commencer par comparer les trois méthodes possibles pour programmer un traitement périodique : `setitimer()`, `nanosleep()` et `clock_nanosleep()`. Pour cela, un second script, `compare-results.sh`, superpose sur le même graphique les histogrammes – représentés sous formes de courbes – issus de plusieurs fichiers.

Voici le comportement de ces trois méthodes pour l'ordonnancement temps réel Fifo :

Nous voyons que les performances de `setitimer()` sont sensiblement moins bonnes que celles de `nanosleep()` et `clock_nanosleep()`. Cette dernière méthode est la meilleure, et c'est celle que nous utiliserons dans les comparaisons à venir.

Par exemple, nous pouvons examiner les fluctuations d'un système périodique programmé avec `clock_nanosleep()` sous les différents ordonnancements d'un noyau Vanilla.

L'ordonnancement `other` n'étant pas temps réel, il n'est présent sur cette figure que pour information. Nous voyons que les performances des ordonnancements Fifo et Round Robin sont globalement équivalentes.

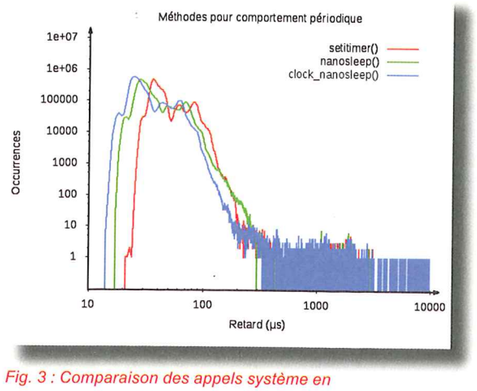


Fig. 3 : Comparaison des appels système en ordonnancement Fifo

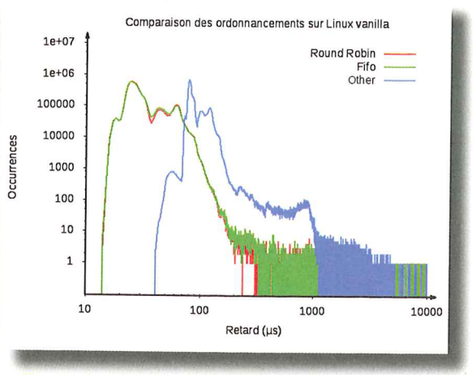


Fig. 4 : Comparaison des ordonnancements sur Linux Vanilla

5.2 Noyau Linux PREEMPT_RT

Nous allons refaire la même comparaison sur un noyau ayant été modifié par le patch `PREEMPT_RT`.

Cette fois, il y a une légère différence entre Round Robin et Fifo, ce dernier étant plus performant. On voit également que les performances globales des ordonnancements temps réel se sont sensiblement améliorées. Alors qu'avec un noyau Linux Vanilla, des fluctuations pouvaient dépasser la milliseconde, cette fois on dépasse à peine la centaine de microsecondes pour les ordonnancements temps réel.

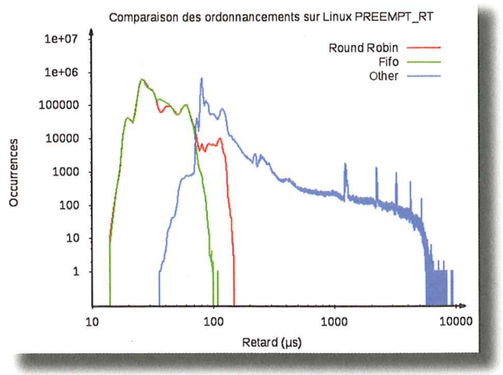


Fig. 5 : Comparaison des performances des ordonnancements sur Linux PREEMPT_RT

Terminal

```
root@raspberrypi:~# grep Max results-3.8.13-rt16+-*-clocknanosleep.txt
results-3.8.13-rt16+-fifo-clocknanosleep.txt:# Max Latencies: 00109
results-3.8.13-rt16+-other-clocknanosleep.txt:# Max Latencies: 15389
results-3.8.13-rt16+-rr-clocknanosleep.txt:# Max Latencies: 00150
```

5.3 Xenomai

Il existe une version de `cyclictest` pour Xenomai ayant un peu moins d'options que celle de `rt-test`, mais elle convient parfaitement pour nos expériences. Elle n'utilise que la méthode `clock_nanosleep()`, ce qui nous permet une comparaison significative avec les autres systèmes.

J'ai laissé le programme s'exécuter sous une forte charge système pendant douze heures.

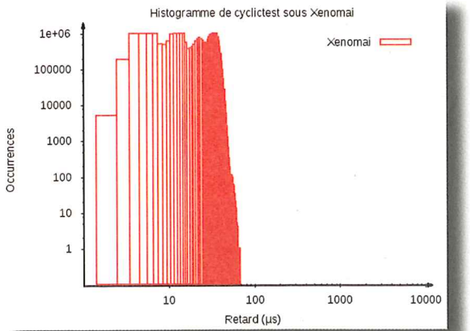


Fig. 6 : Histogramme obtenu avec Xenomai

Cette fois l'échelle des abscisses commence à zéro, car les latences sont beaucoup plus courtes. Nous voyons également que les résultats sont plus tassés, la valeur maximale étant nettement en-dessous de 100 microsecondes.

Terminal

```
[results]$ grep '#' results-3.8.13-ipipe+.txt
# Histogram
# Total: 043200000
# Min Latencies: 00002
# Avg Latencies: 00008
# Max Latencies: 00067
# Histogram Overflows: 00000
# Histogram Overflow at cycle number:
# Thread 0:
```

Nous pouvons zoomer sur le graphique pour voir son aspect complet, Figure 7, page suivante.

CONCLUSION

Nous voyons que le petit Raspberry Pi a un comportement tout à fait honorable en ce qui concerne les traitements temps réel. Naturellement, il n'est pas question de l'utiliser dans un contexte contrôlant la sécurité des personnes (pas plus que tout autre système sous Linux d'ailleurs), mais il peut très bien servir pour réaliser des tâches impliquant des contraintes temporelles.

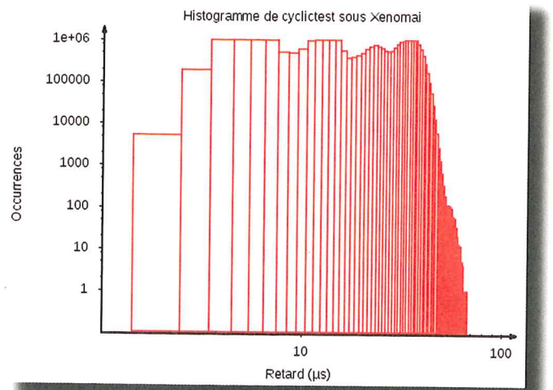


Fig. 7 : Histogramme obtenu avec Xenomai (zoom)

Si les tolérances sont de l'ordre de quelques millisecondes, un système Linux Vanilla peut suffire. Si les contraintes s'expriment plutôt en centaines de microsecondes, le patch **PREEMPT_RT** sera adapté. Enfin, pour les systèmes dont les tolérances sont en dizaines de microsecondes, on privilégiera Xenomai (et si les fluctuations maximales acceptables sont inférieures à la dizaine de microsecondes, on n'utilisera pas Linux !).

Bien entendu, il faut être conscient qu'aucune garantie n'est donnée que les valeurs maximales observées ne seront pas dépassées à un moment ou un autre, et que ce risque doit être pris en considération. Les conséquences d'un tel dépassement doivent être soigneusement étudiées avant de choisir un système temps réel. Échouer dans la production d'une pièce sur cent mille, par exemple, peut être un pari acceptable face à l'utilisation d'un système de contrôle peu coûteux. Si cet échec risque d'entraîner la destruction d'une partie de la chaîne de production, le pari est beaucoup moins raisonnable...

Pour terminer, précisons que les manipulations réalisées ci-dessus (applications des patches, compilations des noyaux, etc.) peuvent paraître complexes, elles sont surtout détaillées ici à titre pédagogique. Pour installer **PREEMPT_RT** ou Xenomai sur un système industriel, on fait généralement appel à des environnements de construction comme Buildroot ou Yocto, qui intègrent directement les opérations nécessaires. ▮

RÉFÉRENCES

[1] Site de référence de Xenomai : **http://www.xenomai.org**

[2] Site de référence pour le patch **PREEMPT_RT** : **rt.wiki.kernel.org**

POUR EN SAVOIR PLUS

Ficheux P., « *PREEMPT_RT sur Raspberry Pi* » :
http://www.linuxembedded.fr/2013/01/17/

Blaess C., « *Solutions temps réel sous Linux* », éd. Eyrolles, 2012.

Ficheux P., « *Linux embarqué* », éd. Eyrolles, 2012.