Montage 19

Interactions entre Arduino et Raspberry Pi

Tous les mois ou presque, une nouvelle carte à microcontrôleur ou un nouveau descendant de la famille Raspberry Pi voit le jour. En introduction à cet ouvrage, j'ai mentionné que certains percevaient le Raspberry Pi comme une menace pour l'Arduino. Mais tout le monde ne partage pas cet avis. En effet, qu'est-ce qui empêche de bâtir un duo solide à partir des deux cartes ? Cela me paraît être une si bonne idée que j'ai décidé de consacrer un montage à ce sujet.

Au sommaire :

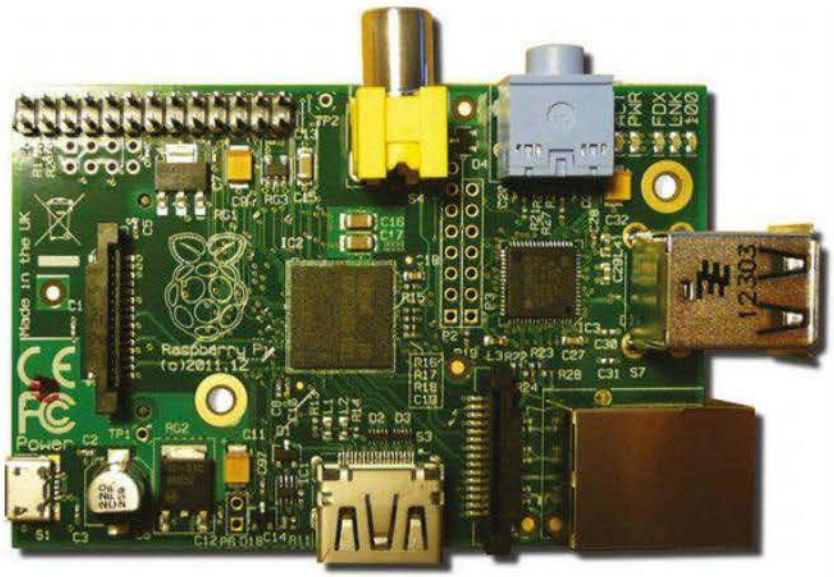
- le Raspberry Pi ;
- l'environnement de développement du Raspberry Pi et d'Arduino ;
- la communication au moyen de Python et de pyFirmata via le port USB ;
- la communication au moyen de Python et de pyFirmata via une liaison série TTL.

Réveillons l'Arduino sommeillant dans tout Raspberry Pi

Certes, le sujet de ce livre n'est pas le Raspberry Pi, mais il me paraît opportun d'étudier la carte d'un peu plus près afin d'en souligner les points forts. Il s'agit d'un ordinateur monocarte de la taille d'une carte bancaire qui a été développé par la fondation britannique Raspberry Pi. Il coûte environ 35 €, ce qui est donc très abordable. Mais ce n'est pas pour cette raison que les gens manifestent de l'intérêt pour ce nano-ordinateur. Cet ordinateur – puisqu'il s'agit bien d'un ordinateur à part entière – réunit tout ce qui est nécessaire pour s'aventurer dans l'univers de l'informatique et de la programmation. Il possède

un processeur Broadcom BCM2835 de type ARM11 cadencé à 70 MHz, une mémoire vive de 256 Mo ou 512 Mo, un port Ethernet (modèle B) pour le raccordement à un réseau, ainsi que deux ports USB. Comme support d'amorçage, le nano-ordinateur utilise une carte SD sur laquelle peuvent aussi être installés différents systèmes d'exploitation (Linux ou Android) compatibles avec l'architecture ARM.

Figure 19-1 ▶
Le Raspberry Pi



Si l'on veut raccorder un clavier ou une souris, il est préférable d'utiliser des modèles sans fil avec dongle USB afin que les deux périphériques n'occupent qu'un seul port USB. Comme la majorité des écrans TFT ou des écrans plats disposent aujourd'hui d'une connexion HDMI, vous pouvez donc facilement les raccorder au port HDMI de type A *(full size)* du Raspberry Pi. Si votre écran est un modèle plus ancien, sachez qu'il existe aussi des adaptateurs HDMI-DVI pour convertir le signal envoyé sur un port DVI.

Voyons maintenant les préparatifs nécessaires en vue du raccordement d'une carte Arduino Uno à un Raspberry Pi afin que les deux systèmes puissent échanger des informations.

Installation de l'IDE Arduino sur le Raspberry Pi

Vous pouvez évidemment continuer à programmer votre carte Arduino Uno dans votre environnement de développement habituel depuis votre ordinateur sous Windows, Mac ou Linux. Mais, comme tôt ou tard, nous allons raccorder les deux cartes, je vais vous montrer ici comment le faire directement depuis le Raspberry Pi. En effet, qu'est-ce qui vous empêche d'utiliser l'IDE Arduino sur le Raspberry Pi ?! Ouvrez une fenêtre de terminal sur le Raspberry Pi et saisissez les deux lignes suivantes :

```
# sudo apt-get update
# sudo apt-get install arduino
```

Pour l'installation, nous utiliserons le gestionnaire de paquets APT *(Advanced Packaging Tool)*. La première ligne actualise les listes de paquets et la deuxième installe l'IDE Arduino. Une fois l'installation réussie, une nouvelle commande apparaît dans le menu Développement : il s'agit de l'IDE d'Arduino. Notez que la version actuelle 1.0.1 ne prend pas en charge la carte Arduino Yún.

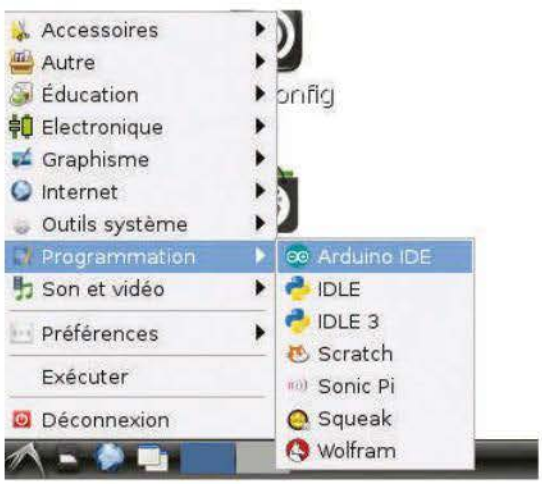


Figure 19-2
L'IDE Arduino dans le menu Développement

Lorsque vous démarrez l'IDE à l'aide de cette commande, la fenêtre bien connue s'ouvre au bout de quelques instants. Vous devez encore choisir le port série correct. Sous Linux, il s'agit de */dev/ttyACM0* pour la carte Uno. Les modèles plus anciens utilisent */dev/ttyUSB0*.

Figure 19-3 ▶
Version 1.0.1 de l'environnement de développement

Vous pouvez vérifier que tout a bien fonctionné en lançant le sketch Blink que j'ai également chargé. Revenons-en à la communication entre l'Arduino et le Raspberry Pi. Nous allons utiliser ici Firmata, un protocole de communication entre ordinateurs ou programmes. Voyons maintenant comment établir une liaison entre l'Arduino et le Raspberry Pi afin que les deux cartes puissent échanger des données.

Firmata

Côté Arduino, nous disposons déjà de tout le nécessaire. Firmata fait partie de l'environnement de développement Arduino et il suffit de le charger en tant que firmware sous la forme d'un sketch. Côté Raspberry Pi, ce n'est pas tout à fait la même chose. Le langage habituel du Raspberry Pi, Python, fait déjà partie de la distribution Linux Debian Wheezy que je recommande aux débutants. Il nous faut donc

uniquement installer Firmata. Le paquet Python correspondant se nomme pyFirmata.

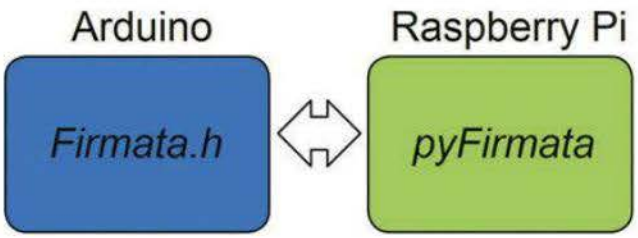


Figure 19-4
Firmata sur l'Arduino et le Raspberry Pi

Comme Firmata fait déjà partie de l'environnement de développement Arduino, il suffit de s'assurer de son bon fonctionnement en ajoutant le fichier d'en-tête `Firmata.h`. Côté Raspberry Pi, nous avons quelques bricoles à installer. Je vais vous présenter deux méthodes employant deux gestionnaires de paquets différents.

Méthode 1 : avec Mercurial

Exécutez les commandes suivantes dans le terminal afin d'installer pySerial, pour la communication série sous Python, et Mercurial, qui est un outil de gestion de versions :

```
# sudo apt-get install python-serial mercurial
```

Vous pouvez maintenant télécharger pySerial et l'installer sous Python en saisissant les lignes suivantes dans le terminal :

```
# hg clone https://bitbucket.org/tino/pyfirmata
# cd pyfirmata
# sudo python setup.py install
```

La première ligne de commande charge les sources requises depuis l'adresse saisie sur le Raspberry Pi. Après le téléchargement, elles se trouvent dans le dossier `pyfirmata`. La commande `cd` de la deuxième ligne permet d'accéder au dossier et la troisième ligne démarre l'installation de pyFirmata sous Python. Lorsque l'installation est terminée, il n'est pas nécessaire de conserver les sources et vous pouvez donc les supprimer à l'aide des lignes suivantes :

```
# cd ..
# sudo rm -r pyfirmata
```

Méthode 2 : avec GitHub

Avant toute chose, si ce n'est pas encore fait, vous devez installer Git (autre logiciel de gestion de versions décentralisé) pour pouvoir télé-

charger les sources de GitHub sur le Raspberry Pi. Pour ce faire, saisissez la commande suivante :

```
# sudo apt-get install git
```

Ensuite, vous pouvez installer pyFirmata à l'aide des lignes suivantes :

```
# git clone https://github.com/tino/pyFirmata.git
# cd pyFirmata
# sudo python setup.py install
```

Comme avec la première méthode, vous pouvez ici aussi supprimer les sources dans le dossier pyFirmata.

Tout est maintenant prêt pour procéder au premier essai. Nous allons dire bonjour à Arduino et nous ferons clignoter sa LED 13. Comment allons-nous faire ?

Préparation de l'Arduino

Vous devez évidemment munir l'Arduino du firmware de Firmata en chargeant le sketch correspondant et en le téléversant sur la carte. Activez la commande *Fichiers>Exemples>Firmata>StandardFirmata* pour ouvrir le sketch dans l'environnement de développement, puis téléversez-le sur la carte Arduino. Vous n'avez rien de plus à faire pour l'instant.

Préparations du Raspberry Pi

Comme nous travaillerons avec le langage de programmation Python sur le Raspberry Pi, il est conseillé d'installer un environnement de développement Python. Actuellement, j'utilise SPE *(Stani's Python Editor)* que vous pouvez installer à l'aide des lignes suivantes :

```
# sudo apt-get update
# sudo apt-get install spe
```

Toutefois, cet environnement de développement ralentit le Raspberry Pi, comme vous pourrez le constater si vous jetez un œil à l'indicateur de performances du processeur dans la barre des tâches. Au lieu de SPE, vous pouvez aussi utiliser le simple éditeur de texte Nano qui s'installe à l'aide de la ligne de commande suivante :

```
# sudo apt-get install nano
```

Pour ouvrir l'éditeur, il suffit ensuite de saisir la commande suivante dans le terminal :

```
# nano
```

Nous allons néanmoins continuer avec SPE. Après son installation, vous le trouverez dans le dossier Développement du menu Démarrer de Linux.

Figure 19-5
Ouverture de Stani's Python Editor

Vous pouvez saisir le code suivant dans l'éditeur pour faire clignoter la LED 13. Ne vous inquiétez pas, nous n'en resterons pas là ! Nous ferons aussi des choses un peu plus compliquées. Cet exercice nous permet simplement de nous mettre en jambe :

```
#!/usr/bin/python

from time import sleep              # importation de la fonction sleep
from pyfirmata import Arduino, util # importation de la fonction Arduino, util

# communication avec la carte Arduino via le port série
arduinoboard = Arduino('/dev/ttyACM0')

# programmation de la broche Arduino
pin13 = arduinoboard.get_pin('d:13:o')
while True:
    pin13.write(1) # LED allumée
    sleep(1)       # pause 1 seconde
    pin13.write(0) # LED éteinte
    sleep(1)       # pause 1 seconde
```

Examinons la signification des différentes lignes du code :

Ligne 1

Afin que le script Python soit correctement détecté par l'interpréteur, la première ligne commence sur les systèmes Linux par la séquence de caractères `#!` suivie du chemin d'accès absolu à l'incontournable interpréteur.

```
#!/usr/bin/python
```

Cette ligne se nomme le *shebang*. L'emplacement de l'interpréteur Python varie selon les systèmes Linux. Par conséquent, la ligne telle qu'elle est présentée ici n'est pas universellement valable. Il existe une meilleure méthode qui utilise le programme `env`.

```
#!/usr/bin/env python
```

Le code `env` charge les variables d'environnement standard de la configuration du système d'exploitation qui prend aussi en charge la variable d'environnement *PATH*. Sur mon ordinateur, l'interpréteur Python se trouve sous `/usr/bin/python`.

Ligne 3

Comme nous avons besoin de la fonction `sleep` pour insérer une pause, celle-ci est importée à la ligne 3 par l'instruction `from/import`.

Ligne 4

Pour nous permettre d'utiliser la multitude de fonctions de pyFirmata après son installation, le paquet est inséré dans le code à la ligne 4 par l'instruction `from/import`.

Ligne 7

Comme nous voulons communiquer et contrôler la carte Arduino via le port série, nous devons nous y connecter. Il s'agit du même *device* (appareil) */dev/ttyACM0* que celui déjà utilisé pour programmer la carte Arduino. Pour que Python puisse avoir accès au port série, nous avions précédemment installé le paquet *python-serial* pendant la phase préparatoire.

```
arduinoboard =Arduino('/dev/ttyACM0')
```

Par cette ligne, pyFirmata crée une instance de *pyfirmata* que nous avons nommée `arduinoboard`. Comme argument, nous transmettons précisément le nom d'appareil que je viens d'indiquer.

Ligne 10

Les différentes broches de la carte peuvent être contrôlées ou configurées par la méthode `get_pin`. La configuration s'effectue par le biais d'une séquence de caractères au format suivant :

```
(a|d:<PinNr>:i|o|p|s)
```

Les trois informations nécessaires sont énumérées en étant séparées par des deux-points. En voici la signification :

- L'instruction est-elle destinée à une broche analogique (*a*) ou numérique (`d`) ?
- De quelle broche s'agit-il (`PinNr`) ?
- Quel mode faut-il utiliser (`i` : entrée, `o` : sortie, `p` : MLI, `s` : servo) ?

C'est donc ce que nous faisons à la ligne 10.

```
pin13 = arduinoboard.get_pin('d:13:o')
```

Une variable intitulée `pin13` est initialisée à l'aide de la méthode `get_pin` afin de nous permettre de manipuler cette broche (numérique, 13, sortie) conformément aux instructions transmises aux lignes 11 à 15.

Lignes 11 à 15

Une boucle `while` permet d'exécuter en continu les instructions énoncées dans le corps de la boucle :

```
while True:
  pin13.write(1) # LED allumée
  sleep(1) # pause 1 seconde
  pin13.write(0) # LED éteinte
  sleep(1) # pause 1 seconde
```

Pour associer différents niveaux à la LED qui est connectée à la broche 13, nous utilisons la méthode `write` avec l'argument 1 pour le niveau HIGH ou 0 pour le niveau LOW. Entre les deux, nous plaçons la fonction `sleep` qui interrompt l'exécution du programme pendant une durée d'une seconde. La boucle `while` est exécutée tant que l'instruction suivante est vraie (`True`). Nous n'avons pas employé de variable comme instruction, mais la constante `True`, ce qui signifie que la boucle est exécutée à l'infini ou jusqu'à ce que l'utilisateur interrompe manuellement l'exécution du script à l'aide du raccourci *Ctrl* + *C*. Si vous avez démarré le script depuis le Raspberry Pi, observez la LED RX de la carte Arduino. Elle s'allume et s'éteint avec un intervalle d'une seconde. On peut donc en conclure que des

informations sont transmises depuis le Raspberry Pi à l'Arduino via le port série avec le même intervalle pour contrôler la LED.

Commande par MLI

Nous allons maintenant voir comment commander une LED raccordée à l'une des broches MLI à l'aide d'un signal MLI (voir la section « Que signifie MLI ? » du chapitre 10). Je voudrais ouvrir une interface graphique sur le Raspberry Pi afin d'envoyer le signal MLI à la carte Arduino à l'aide d'un potentiomètre linéaire comme celui illustré ci-après.

Le potentiomètre linéaire permet de choisir des valeurs comprises entre 0 et 100 qui correspondent aux pourcentages de MLI. Attention toutefois, car la fonction `write` de pyFirmata, qui s'occupe de générer le signal MLI, accepte des valeurs comprises entre 0,0 et 1,0. J'ai donc intentionnellement employé des valeurs à virgule flottante, car la fonction attend des valeurs de type *float*.

Examinons le script Python de plus près. Python ne peut afficher de but en blanc des éléments graphiques, comme des boutons, des étiquettes, des potentiomètres linéaires ou autres. Pour ce faire, nous utilisons une bibliothèque nommée Tkinter. Qu'est-ce donc ? Il s'agit de la première boîte à outils d'interface graphique pour Python. Elle permet de créer sous Python des programmes ayant une interface graphique. La boîte à outils Tk a initialement été développée pour le langage Tcl *(Tool Command Language)*, mais entre-temps, elle a pris place dans la bibliothèque standard de Python. Le module Tkinter (*Tk-Interface*) permet à l'utilisateur de programmer très facilement des applications Tk sans devoir installer au préalable des logiciels ou des bibliothèques supplémentaires. Examinons le script Python que j'ai divisé en blocs pour une meilleure lisibilité.

Initialisation

Pendant la phase d'initialisation, nous importerons aussi bien `pyfirmata` que *Tkinter*. Nous utilisons encore `arduinoboard`, comme dans l'exemple précédent. La broche 3, sur laquelle la diode est

raccordée, doit être initialisée en tant que sortie MLI, ce que nous faisons à l'aide du mode `p`.

```
1  #!/usr/bin/env python
2  import pyfirmata
3  from Tkinter import *
4
5  arduinoboard = pyfirmata.Arduino('/dev/ttyACM0')
6
7  pin3 = arduinoboard.get_pin('d:3:p') # sortie MLI sur la broche 3
```

Fonctions requises

Les fonctions `cleanup` et `setPWM` sont utilisées lors de l'exécution du script – on s'en serait douté. La fonction `cleanup` est exécutée au moment où vous fermez l'interface graphique par un clic sur la croix dans le coin supérieur droit. Elle fait en sorte que la LED soit éteinte au moyen de la fonction `write`. La fonction `setPWM` commande la LED et elle est exécutée tant que vous modifiez la position du curseur du potentiomètre.

```
9  def cleanup():
10     # LED 3 éteinte
11     pin3.write(0)
12     arduinoboard.exit()
13
14 def setPWM(pwm):
15     # commande par MLI de la LED 3
16     # accepte des valeurs comprises entre 0 et 1
17     pin3.write(float(pwm)/100.0)
```

La fonction `write` de la commande par MLI attend des valeurs comprises entre 0 et 1, ce qui signifie que l'argument doit être de type *float*. Comme le potentiomètre transmettra par la suite, en réponse au déplacement du curseur, une valeur du paramètre `pwm` comprise entre 0 et 100, cette valeur doit donc être divisée par 100.

Préparation de l'interface graphique GUI et du potentiomètre linéraire

L'interface graphique est initialisée à la ligne 20, qui prépare plus précisément l'instance `master` de la boîte de dialogue `wm_protocol`, ferme la fenêtre en effaçant son contenu, opération qui s'achève par l'exécution de la fonction `cleanup` qui éteint la diode. À la ligne 22, `wm_title` permet de nommer l'application, nom qui sera affiché dans la barre de titre. Le potentiomètre est initialisé avec les valeurs correspondantes et son exécution démarre à la ligne 24. `from_` et `to` définissent la plage de valeurs transmises par le potentiomètre. Quand le

curseur est actionné, il faut afficher immédiatement le résultat, ce qui est assuré par l'instruction `command` suivie de la fonction exécutée. L'orientation est définie par `orient` ; ici, elle est horizontale. Ensuite, nous précisons encore la longueur (`length`) et nous affichons le nom du potentiomètre au moyen d'une étiquette (`label`).

```
# GUI
master = Tk()
master.wm_protocol("WM_DELETE_WINDOW", cleanup)
master.wm_title('PWM control')

# initialisation du potentiomètre
scale = Scale(master,
    from_ = 0,
    to = 100,
    command = setPWM,
    orient = HORIZONTAL,
    length = 400,
    label = 'PWM-Value')
```

Démarrage du programme

Le gestionnaire d'interface se sert de `pack` pour centrer le potentiomètre à l'intérieur de la boîte de dialogue. L'activation de `mainloop` affiche la boîte de dialogue et maintient le programme dans une boucle sans fin, en attendant que se produisent des événements intéressants comme le déplacement du curseur qui déclenche l'action programmée.

```
scale.pack(anchor = CENTER) # centré
master.mainloop()           # démarrage de TK Event-Loop
```

Il n'y a pas grand-chose d'autre à ajouter sur ce script assez simple. Tkinter est bien plus performant que ce que j'ai pu montrer dans ces quelques pages et je ne peux que vous conseiller la lecture d'ouvrages spécialisés ou la consultation de ressources sur Internet.

Comme vous vous êtes maintenant familiarisé avec le fonctionnement d'une broche MLI, vous allez pouvoir commander un servomoteur au moyen d'un potentiomètre.

Attention !

Si vous utilisez la version 3 de Python, vérifiez que vous écrivez bien *tkinter* (avec un *t* minuscule) lors de son importation. La bibliothèque a été renommée.

Commande d'un servomoteur

Nous avons vu précédemment comment commander un servomoteur. Ici, vous apprendrez à régler précisément l'angle du servomoteur à l'aide d'un potentiomètre.

Pour en savoir davantage sur le brochage d'un servomoteur, je vous invite à relire le montage n° 14 « Le moteur pas-à-pas ». J'aimerais commander le moteur au moyen de la broche MLI 3, mais il est aussi possible d'utiliser une broche qui ne soit pas dotée de cette fonctionnalité de modulation, comme la broche 7. Examinons le code Python qui est très proche de celui de l'exemple précédent. Comme vous pourrez le constater, une fois que l'on a compris le principe de base, il suffit souvent de modifier un détail pour accéder à d'autres fonctionnalités.

```
#!/usr/bin/env python
import pyfirmata
from Tkinter import *

arduinoboard = pyfirmata.Arduino('/dev/ttyACM0')

it = pyfirmata.util.Iterator(arduinoboard)
it.start()

pin3 = arduinoboard.get_pin('d:3:s') # servo sur la broche 3
```

La seule différence dans ce segment de code réside dans le changement de mode qui passe à `s` pour la commande d'un servomoteur. Les deux fonctions suivantes sont quasiment inchangées.

```
def cleanup():
    # désactivation de la broche 3
    pin3.write(0)
    arduinoboard.exit()

def moveServo(a):
    # commande du servo par la broche 3
    pin3.write(a)
```

La valeur du potentiomètre est ici aussi transmise au paramètre de la fonction `moveServo` pour commander le moteur à l'aide de la méthode `write`. L'interface du programme est créée de la même façon que précédemment et je me suis contenté d'en changer le nom.

```
# GUI
master = Tk()
master.wm_protocol("WM_DELETE_WINDOW", cleanup)
master.wm_title('Servo-Control')
```

Nous en arrivons à l'initialisation du potentiomètre qui doit transmettre des valeurs comprises entre 0 et 179 à la fonction `moveServo`. Ensuite, le script démarre avec `mainloop`.

```
# initialisation du potentiomètre
scale = Scale(master,
    from_ = 0,
    to = 179,
    command = moveServo,
    orient = HORIZONTAL,
    length = 400,
    label = 'Angle')

scale.pack(anchor = CENTER) # centré
mainloop()                  # démarrage de TK Event-Loop
```

Interrogation d'un bouton-poussoir

Jusqu'ici, nous avons toujours transmis des informations à la carte Arduino. Nous allons maintenant faire l'inverse en interrogeant un bouton-poussoir qui est raccordé à la broche 8. Comme procède-t-on avec pyFirmata ? Le code est assez évident :

```
#!/usr/bin/env python
import pyfirmata

arduinoboard = pyfirmata.Arduino('/dev/ttyACM0')

pin8 = arduinoboard.get_pin('d:8:i') # entrée sur broche 8

it = pyfirmata.util.Iterator(arduinoboard)
it.start()
pin8.enable_reporting()

while True:
    pin8_state = pin8.read() # lecture de l'état
    if pin8_state == True:
        print 'bouton enfoncé'
    if pin8_state == False:
        print 'bouton non enfoncé'
    arduinoboard.pass_time(0.5) # Pause
```

Ligne 6

Pour pouvoir interroger l'état d'un bouton-poussoir connecté sur une broche, nous devons évidemment la programmer en tant qu'entrée, ce que nous faisons de ce pas avec `d:8:i` (`i` correspond à entrée).

Lignes 8 et 9

Pour lire une broche d'entrée sous pyFirmata, vous ne pouvez pas tout simplement activer une fonction `read`, comme nous le ferons à la ligne 13. Il faut implémenter un `Iterator Thread` qui veille à ce que les broches de la carte Arduino communiquent la valeur courante lors de leur interrogation. Cela évite aussi que ne se produise un débordement de *buffer* qui bloquerait toute la communication sur le port série.

Ligne 10

Par `enable_reporting`, vous indiquez à pyFirmata que vous voulez surveiller la broche.

Lignes 12 et 13

La boucle `while` interroge l'état de la broche 8 en continu en utilisant la méthode `read`.

Lignes 14 à 17

Les instructions `if` interrogent l'état `True` qui correspond au bouton enfoncé et l'état `False` qui correspond au bouton non enfoncé, et affichent le résultat par l'instruction `print`.

Ligne 18

La méthode `pass_time` prévoit une pause d'une demi-seconde après chaque affichage.

J'ai un petit problème : au démarrage du script, rien n'indique que le bouton n'est pas enfoncé. Pourquoi le message correspondant ne s'affiche-t-il pas ?

Vous faites bien d'en parler, Ardus. Quand le bouton n'a pas encore été enfoncé, l'état correspond à `None`. Libre à vous de compléter le code afin que cet état soit aussi interrogé et qu'un message s'affiche.

Interrogation d'un port analogique

Pour finir, nous allons voir comment interroger une entrée analogique et les aspects à prendre en considération. Là encore, le script est assez évident (voir page suivante).

```
#!/usr/bin/env python
import pyfirmata
from time import sleep

arduinoboard = pyfirmata.Arduino('/dev/ttyACM0')

pin0 = arduinoboard.get_pin('a:0:i') # entrée analogique sur broche

it = pyfirmata.util.Iterator(arduinoboard)
it.start()
pin0.enable_reporting()

while True:
    value = pin0.read() # lecture de la valeur analogique
    print value         # affichage
    sleep(1)            # pause 1 seconde
```

Ligne 7

La broche analogique 0 est désignée par un `a` (pour *analogique)* et elle est programmée en tant qu'entrée par le biais du mode `i`.

Lignes 13 à 16

La valeur analogique est lue sur la broche 0 à l'intérieur de la boucle `while` au moyen de la méthode `read`. Le résultat est compris entre 0,0 et 1,0. Ensuite, le programme marque une courte pause d'1 seconde.

Eh là, il y a un problème ! Voilà les valeurs qui sont affichées lorsque je fais lentement tourner le potentiomètre de la gauche vers la droite.

```
/home/pi/pyfirmata_analog001.py
None
0.0
0.7019
0.825
0.8563
0.914
1.0
1.0
Script stopped by user (ok).
```

Ah oui, Ardus ! Je crois avoir compris où tu voulais en venir. Au tout début de l'affichage, on peut lire `None`, ce qui signifie qu'aucune valeur valide n'a pu être lue. Cela se produit de temps en temps au début de l'interrogation. Voici comment l'éviter :

```
while True:
    value = pin0.read() # lecture de la valeur analogique
    if value != None:
        print 'Valeur : %f' % value # affichage
        sleep(1)                    # pause 1 seconde
```

À la ligne 15, j'ai placé une instruction `if` qui intercepte la valeur `None`, le cas échéant. Pour améliorer la présentation, vous pouvez aussi

précéder l'affichage d'un texte ou de la mention du type de données, comme à la ligne 16. Vous obtenez alors le code suivant :

```
/home/pi/pyfirmata_analog002.py
Value : 0.000000
Value : 0.012700
Value : 0.326500
Value : 0.770300
Value : 0.849500
Value : 1.000000
Script stopped by user (ok).
```

Liaison série entre le Raspberry Pi et l'Arduino

Jusqu'ici, les informations échangées entre le Raspberry Pi et l'Arduino ont transité par la liaison USB qui a servi à l'acheminement des données série. Mais il est aussi possible d'établir directement une liaison TTL série à condition de respecter la précaution suivante : le Raspberry Pi fonctionne avec une tension d'alimentation maximale sur ses entrées et sorties GPIO de 3,3 V, tandis que l'Arduino utilise une tension de 5 V. Si vous raccordez les deux cartes l'une à l'autre sans prendre de précautions, c'est la loi du plus fort qui prévaut et l'un des protagonistes restera sur le carreau. Le Raspberry Pi recevra une tension trop élevée. Et comme ses broches GPIO sont directement reliées au processeur, sans protection, celui-ci grillera et la carte sera bonne à jeter. Ce n'est donc pas une bonne idée ! Comment l'éviter ? Nous pourrions utiliser un diviseur de tension afin de nous assurer que le Raspberry Pi reçoive bien une tension de 3,3 V. J'ai préféré employer un composant meilleur marché qui s'appelle un convertisseur logique ou *levelshifter*, comme le modèle proposé par Adafruit.

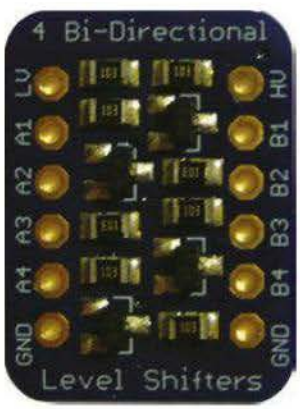


Figure 19-6
Convertisseur logique Adafruit

Ce composant permet non seulement de convertir la tension pour les liaisons RX/TX, mais aussi pour les bus I²C et SPI. Sur le côté gauche se trouvent les broches de raccordement en basse tension (LV ou *LOW Voltage*), et sur le côté droit, il y a les broches haute tension (HV ou *HIGH Voltage*). Voyons maintenant comment raccorder correctement le Raspberry Pi et l'Arduino pour que les deux cartes réussissent à communiquer via les ports série.

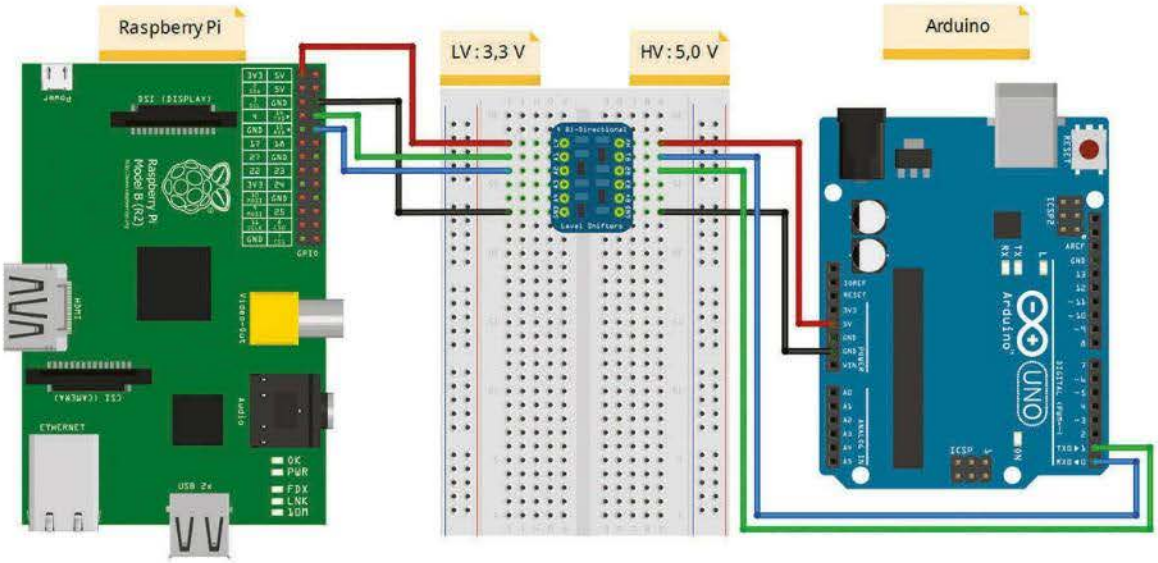


Figure 19-7 ▲
Le convertisseur logique sert d'intermédiaire entre le Raspberry Pi et l'Arduino.

Il va sans dire que les lignes émettrices et réceptrices des deux cartes doivent être croisées, car si vous voulez que la liaison TX verte, par laquelle le Raspberry Pi transmet des données, réussisse à se faire entendre par l'Arduino, cette dernière doit être connectée à sa liaison réceptrice RX bleue, et inversement. Le montage suivant permet de faire clignoter une LED connectée à la broche 8 par le biais des deux liaisons RX/TX.

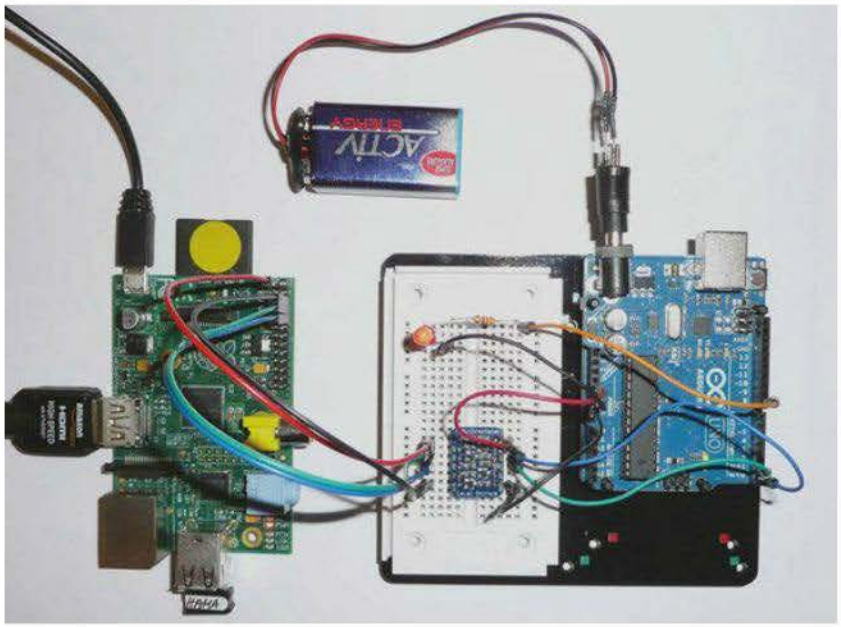


Figure 19-8
Le Raspberry Pi commande l'Arduino via les liaisons RX/TX.

Vous pouvez constater qu'il n'y a pas de connexion USB entre les deux cartes et que la communication passe exclusivement par l'interface UART. L'alimentation électrique de l'Arduino s'effectue au moyen d'une pile de 9 V. Quelles conditions doivent être remplies pour que cette forme de communication fonctionne ? Tout d'abord, je dois vous en dire davantage sur le port série du Raspberry Pi. Par défaut, le Raspberry Pi utilise ce port en tant qu'interface de console par l'intermédiaire duquel vous pouvez accéder au système depuis l'extérieur, même si vous n'y avez pas raccordé de moniteur, de souris ou de clavier – quasi *headless*. Mais ce moyen d'accès dont nous n'avons pas absolument besoin pour le moment bloque les broches RX/TX et nous empêche de poursuivre notre expérience. Nous devons donc couper ce lien. La solution réside dans le fichier */etc/inittab* que vous pouvez ouvrir dans l'éditeur de texte Nano avec la ligne suivante :

```
# sudo nano /etc/inittab
```

Faites défiler le texte vers le bas jusqu'à l'entrée.

```
T0:23:respawn:/sbin/getty -L ttyAMA0 115200 vt100
```

Cette ligne doit être commentée de façon à ne plus pouvoir bloquer l'interface série à la prochaine réinitialisation. Pour ce faire, il vous suffit d'insérer un dièse en début de ligne :

```
# T0:23:respawn:/sbin/getty -L ttyAMA0 115200 vt100
```

Ensuite, fermez l'éditeur à l'aide des raccourcis *Ctrl* + *X* et *J* + *Retour* et redémarrez le système. Il ne vous reste plus qu'à apporter quelques modifications au fameux script Blink.

```
1  #!/usr/bin/python
2
3  from time import sleep              # importation de la fonction sleep
4  from pyfirmata import Arduino, util # importation de la fonction Arduino, util
5
6  # communication avec la carte Arduino via le port série
7  arduinoboard = Arduino('/dev/ttyAMA0')
8
9  # programmation de la broche Arduino
10 pin8 = arduinoboard.get_pin('d:8:o')
11 while True:
12     pin8.write(1) # LED allumée
13     sleep(1)       # pause 1 seconde
14     pin8.write(0) # LED éteinte
15     sleep(1)       # pause 1 seconde
```

À première vue, le code ne semble pas avoir changé. Pourtant, il y a une différence majeure. Regardez attentivement la ligne 7 à laquelle le port série est initialisé.

Avant :

```
arduinoboard = Arduino('/dev/ttyACM0')
```

Après :

```
arduinoboard = Arduino('/dev/ttyAMA0')
```

Le *device* que nous avions utilisé précédemment se rapportait à l'interface USB. Après la correction, le *device* accède à l'interface UART.