Le bouton poussoir un composant banal, ô combien, étonnant.

https://www.framboise314.fr/le-bouton-poussoir-un-composant-banal-o-combien-etonnant/

Publié le 29 août 2016 - par Patrice SEIBEL

Le bouton poussoir (BP) existe sous de nombreuses tailles, formes et technologies. Il est la principale Interface de communication entre l'Homme et la Machine (IHM). Son emploi est si courant que la plupart des utilisateurs ne soupçonnent même pas les phénomènes liées à sa mise en œuvre. Le bouton poussoir voit son origine dans la sphère de l'électromécanique.

Le BP est un "escargot" en terme de vitesse (à cause de l'humain qui l'actionne) par rapport au circuit électronique (analogique, logique ou numérique) qu'il commande. Les humains capables d'actionner un BP en moins de 10 ms sont rares.

Contenu

Introduction

Le vocable bouton poussoir regroupe également les interrupteurs et autres actionneurs qui ne sont qu'une forme particulière de ce composant.

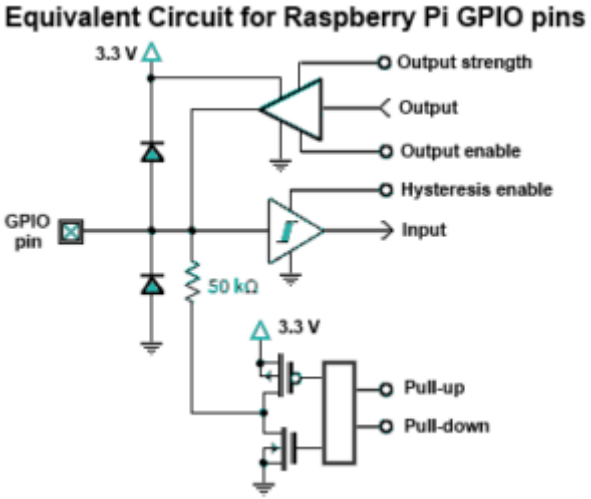
image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure02.png

 La photo représente quelques spécimens de BP, interrupteurs, inverseurs et autres encodeurs mécaniques. On trouve également des BP à commande magnétique ou sous ampoule de verre/plastique avec des liquides conducteurs et autres contacts de relais. Les contacts peuvent à ouverture ou fermeture de circuit. La qualité de ces organes de commande est également très importante dans la mise en œuvre. Le prix de ces composants peut aller de 1 à 20 en fonction de la qualité. Malheureusement, les deux ne sont pas toujours liés.

Théorie

Un peu de pseudo-théorie, le processeur ou plus exactement le SoC (System on Chip) qui équipe la platine Raspberry Pi intègre un nombre important d'entrées/sorties. Ces ports GPIO (General Purpose Input/Output) permettent à notre SoC d'interagir avec d'autres périphériques ou circuits électroniques.

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure03-300x251.png



Dans ce schéma de principe, on retrouve principalement la fonction sortie (output) ou la fonction entrée (input).
Lorsqu'on active la fonction entrée sur une broche du GPIO, on a une broche connectée à rien c'est à dire "en l'air". Cette entrée est alors sensible aux perturbations ou aux bruits ambiants. L'entrée, en question, peut être soit au niveau 1 (+3.3V), au niveau 0 (0V) ou carrément indéfinie. Pour éviter ce phénomène, on peut fixer le potentiel d'entrée de la broche à 1 ou 0 par l'intermédiaire d'une résistance de soutirage.
Cette résistance (50 kΩ sur le schéma) est connectée à la pin GPIO via des transistors mosfet utilisés en commutateur et pilotées par les entrées pull-up et pull-down du schéma (la représentation des mosfet n'est pas un sinogramme).
Si on relie la résistance au +3.3V notre entrée présente un niveau 1 sur son entrée et un niveau 0 dans le cas contraire.

Remarques :
– La valeur de 50 kΩ est un ordre de grandeur, car dans la pratique, cette

valeur est comprise entre 40 et 100 kΩ,
– On considère que le niveau 1 correspond à un niveau de tension supérieure à 2V (difficile de trouver une documentation satisfaisante sur le sujet),
– On considère que le niveau 0 correspond à un niveau de tension inférieure à 1.2V,
– Entre ces deux niveaux de tension, on se trouve dans un “no man’s land” donc à éviter.

Nota : Les deux diodes connectées à la broche GPIO **ne sont pas des diodes de protection** de l’entrée mais des diodes parasites (aléas de la fabrication). **Les broches ne tolèrent pas une tension de 5V**. Si l’on applique une tension supérieure à 3.3V sur une entrée, on risque, au mieux, la destruction de l’entrée/sortie et, au pire, la destruction de notre Raspberry Pi.

Schémas électroniques

En fonction de ce que l'on a abordé dans le chapitre précédent, il est possible d'opter pour quatre configuration de schéma. Dans tous les cas, notre cher bouton poussoir sera raccordé à la broche 22 du GPIO.

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure04a-300x293.png

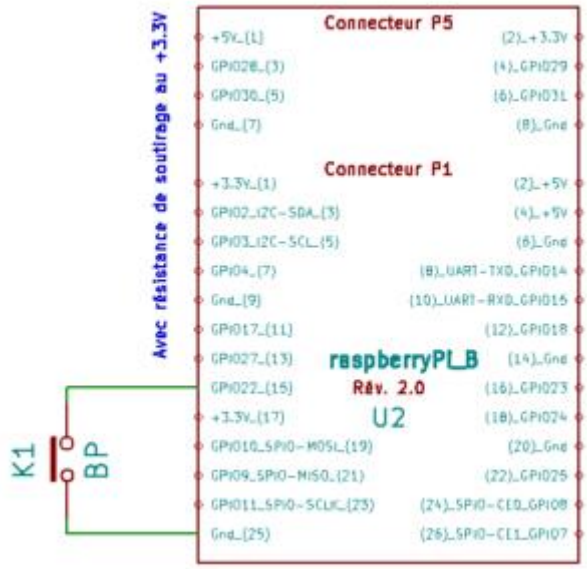


Figure 4a

Dans ce schéma de connexion, le BP est connecté entre l'entrée GPIO22 et l'alimentation 3.3V. Le potentiel de repos de la broche d'entrée est fixé par la résistance interne de soutirage à la masse.
Lorsque le BP n'est pas appuyé le potentiel de l'entrée se trouve à 0V.
Lorsque le BP est appuyé le potentiel de l'entrée se trouve à +3.3V.

Dans ce schéma de connexion, le BP est connecté entre l'entrée GPIO22 et la masse. Le potentiel de repos de la broche d'entrée est fixé par la résistance interne de soutirage au +3.3V.
Lorsque le BP n'est pas appuyé le potentiel de l'entrée se trouve à +3.3V.
Lorsque le BP est appuyé le potentiel de l'entrée se trouve à 0V.

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure04b-300x291.png

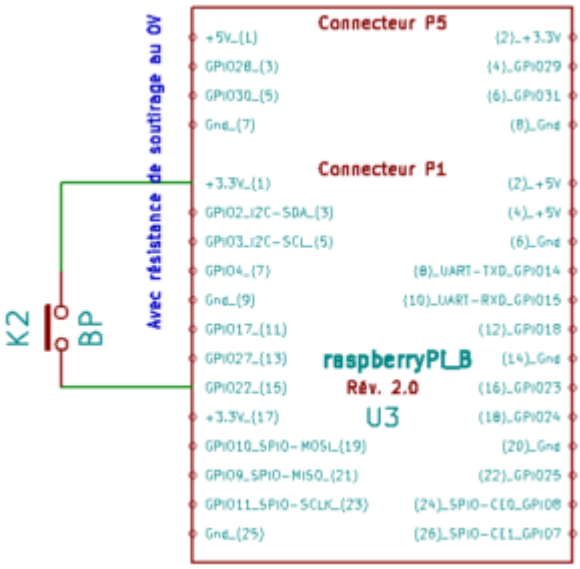


Figure 4b

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure04c-300x297.png

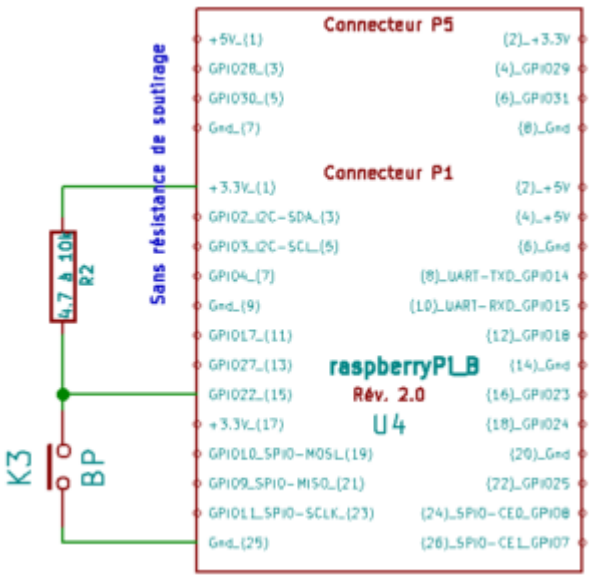


Figure 4c

Dans ce schéma de connexion, le BP est connecté entre l'entrée GPIO22 et la masse. Le potentiel de repos de la broche d'entrée est fixé par la résistance externe reliée au +3.3V. La résistance de soutirage interne n'est pas activée. Lorsque le BP n'est pas appuyé le potentiel de l'entrée se trouve à +3.3V.
Lorsque le BP est appuyé le potentiel de l'entrée se trouve à 0V.

Dans ce schéma de connexion, le BP est connecté entre l'entrée GPIO22 et l'alimentation +3.3V. Le potentiel de repos de la broche d'entrée est fixé par la résistance externe reliée à la masse. La résistance de soutirage interne

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure04d-300x295.png

n’est pas activée. Lorsque le BP n’est pas appuyé le potentiel de l’entrée se trouve à 0V. Lorsque le BP est appuyé le potentiel de l’entrée se trouve à +3.3V.

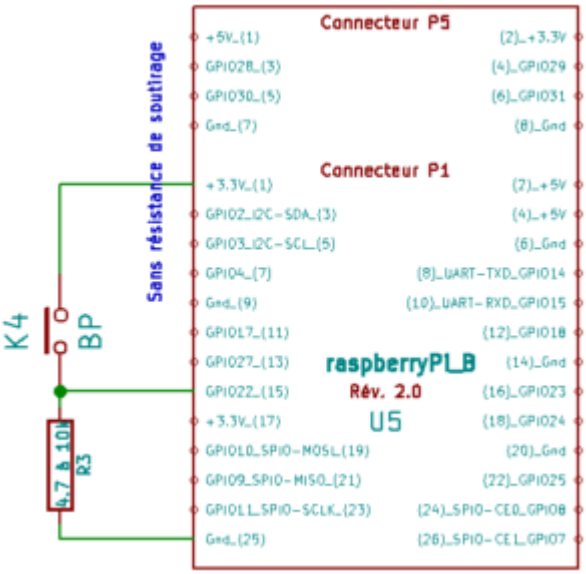


Figure 4d

L’utilisation des schémas des figures 4c et 4d est préférable dans un milieu à fortes perturbations électromagnétiques car il présente une meilleure immunité aux bruits. Il est, en effet, plus facile de perturber une entrée reliée à un potentiel de référence avec une résistance de 50 ou 100 kΩ qu’avec une résistance de 4.7 ou 10kΩ.
L‘utilisation des montage des figures 4a et 4c semble plus opportun. En effet, si l’on déporte le BP (entrée et masse), il y a moins de risques de court circuit accidentel qu’avec les schémas 4b et 4d (entrée et +3.3V).

Pour nos essais, nous avons opté pour un schéma dérivant de la figure 4b. En effet, si vous vous levez le matin avec sieur Murphy à vos côtés, il est nécessaire de prendre quelques précautions. Si accidentellement, vous programmez notre broche en sortie, alors l’appui sur le BP risque de provoquer des dégâts irréversibles (court-circuit). On peut se protéger de ce risque en insérant une résistance de 1kΩ en série avec l’entrée du port GPIO qui limitera le courant à une valeur non destructrice pour le SoC. L’auteur utilise ce type de câblage lors des essais de programmation directe des registres internes du SoC, car on a souvent des surprises.

En reliant le BP au +3.3V, on aura également une vision plus digeste des actions. En appuyant sur le BP (1), la broche d’entrée se retrouve au niveau +3.3V soit un niveau logique 1. Si l’on fait l’inverse (figures 4a et 4c), il faudra faire une gymnastique intellectuelle car un appuie sur le BP (1), on aura l’entrée qui va passer au niveau 0V, soit un 0 logique.

Toutes ces explications pour arriver à un schéma aussi simple !

image: https://www.framboise314.fr/wp-

content/uploads/2016/08/figure05-300x285.png L’avantage de cette présentation, c’est qu’elle liste toutes les possibilités de raccordement de notre bouton poussoir.

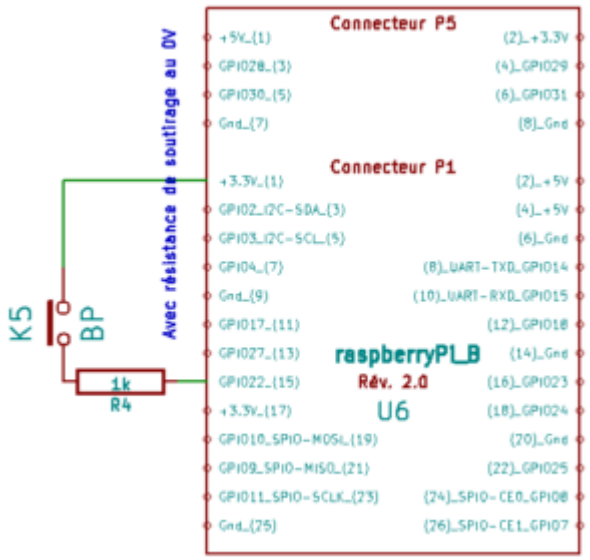
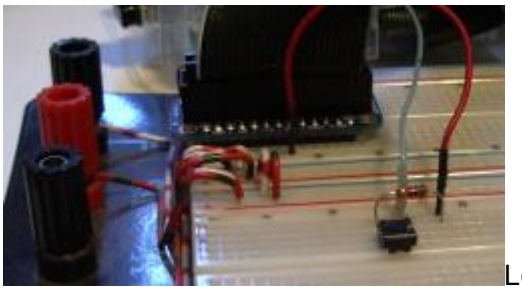


Figure 5

Schéma de câblage

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure01.png



Le câblage est simple et ne demande que peu de composants. On peut le réaliser avec un cobbler comme sur la photo ci-contre ou par une connexion directe sur le port GPIO du raspberry moyennant un petit montage volant (ne pas oublier l’isolation).
Le cobbler est un accessoire qui permet de déporter les pins du GPIO vers une plaque d’essai (breadboard).

Cahier des charges

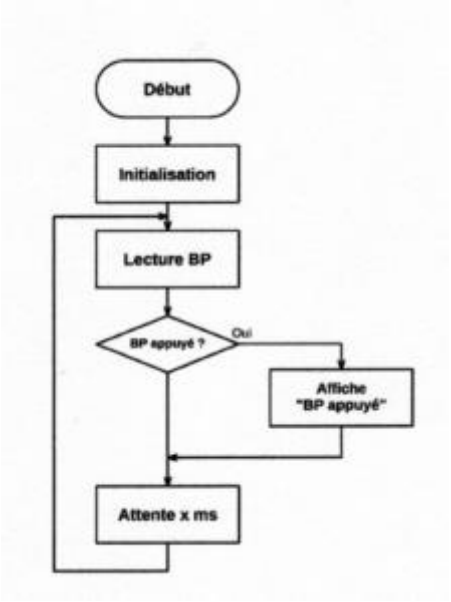
Tout programmeur qui se respecte doit disposer ou établir un cahier des charges qui définit les différentes opérations à réaliser. Dans notre cas, il sera plus que simple. Notre cahier des charges :

1) lire l’état de l’entrée,
2) si BP appuyé alors afficher un message dans la console ,

3) attente x ms,
4) continuer en 1) (on répète indéfiniment la boucle)

Organigramme

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure07-223x300.png



La représentation graphique ci-contre représente le déroulement de notre programme. Il comporte les différentes étapes qui permettront une transcription dans un langage donné.
La première phase consiste à initialiser la pin GPIO22 en entrée pour lire l'état du BP.
La phase suivante consiste à lire l'état de l'entrée donc du BP. Si c'est le cas, on affiche dans la console le message « BP appuyé », sinon on continue.
Pour finir, on a une petite attente avant de reboucler sur la phase lecture BP.

Programme

Les différents programmes abordés dans la suite peuvent être exécuter sur toutes les versions du raspberry Pi (type A, B, jusqu'au RPI3 y compris le zéro).
La saisie des programmes se fait via un éditeur de texte graphique (par exemple geany) ou en mode console (par exemple nano).

L'accès au Raspberry Pi peut se faire via son interface écran HDMI / clavier ou bien par l'intermédiaire d'un connexion à distance (mode console ou graphique). L'auteur utilise une connexion console via une liaison ssh.

Avertissement :
Tous les programmes de cette présentation ont une vocation découverte ou initiatique, donc faire découvrir le monde merveilleux de la communication avec le milieu externe de notre Raspberry Pi via son port GPIO.
Certains programmes peuvent présenter une charge CPU importante voir atteindre 100 %. Ce type de fonctionnement ne présente aucun danger pour notre carte.
Toutes les remarques sur le sujet des saga blink restent valables.

Pour nos premiers tests, nous utiliserons le langage Python avec la bibliothèque RPi.GPIO. L'approche multi-langages se fera dans les chapitres suivants de ce didacticiel.

Pour saisir ce programme, il faut saisir dans la console :

```
1nano push01.py
```

Cette commande ouvre un fichier texte vide appelé push01.py (pour la sauvegarde faire ctrl+o et pour sortir ctrl+x).
Le contenu du programme et de ses commentaires sont représentés ci-dessous.

```
1 #!/usr/bin/python3
2 # -*- coding:utf-8 -*-
3 """
  Programme classique lecture entrée GPIO avec la bibliothèque RPi.GPIO
4
  utilisation de la fonction GPIO.input()
5
  Bouton poussoir raccordé entre GPIO22 et +3.3V
6
  (avec résistance de protection de 1k en série)
7
  nom programme       : push01.py
8
  logiciel            : python 3.4.2
9
  cible               : raspberry Pi
1 date de création    : 18/08/2016
0
  date de mise à jour : 18/08/2016
1
  version             : 1.0
1
1
  auteur              : icarePetibles
1
2 référence           :
  """
1
1
3 #---------------------------------------------------------------------
  -------
1
```

```
# Bibliothèques
#------------------------------------------------------------------------
-------
import RPi.GPIO as GPIO                 #bibliothèque RPi.GPIO
import time                             #bibliothèque time
#------------------------------------------------------------------------
-------
pin = 22             #broche utilisé en entrée
#temps = 1                              #valeur attente en msec
#temps = 10
temps = 100
#temps = 100
#temps = 1000

GPIO.setwarnings(False)                 #désactive le mode warning
GPIO.setmode(GPIO.BCM)                  #utilisation des numéros de ports
du
                                        #processeur
GPIO.setup(pin, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
                                        #mise en entrée du port GPIO 22
                                        #et activation résistance
soutirage
                                        #au ground
if __name__ == '__main__':
    """
    Programme par défaut
    """
    print("Début du programme")         #IHM
    print("Sortie par ctrl-c\n")        #IHM
    try:
        while True:                     #boucle infinie

```

```
            entree = GPIO.input(pin)    #lecture entrée
            if (entree == True):        #si touche appuyée
                print("BP appuyé")      #IHM
            time.sleep(temps / 1000)    #attente en msec
    except KeyboardInterrupt:           #sortie boucle par ctrl-c
        GPIO.cleanup()                  #libère toutes les ressources
        print("\nFin du programme\n")   #IHM
```

```
8

4
9

5
0

5
1
```

Les lignes 20 et 21 importent les bibliothèques nécessaires à notre programme.
L'initialisation des différents paramètres est effectuée dans les lignes 23 à 41.
Le programme principal s'effectue dans le module while True: (boucle sans fin). A la ligne 44, on lit l'entrée de notre broche et si la valeur lue (ligne 46) est à 1 (3.3V ou True) alors on affiche dans la console le message « BP appuyé » (ligne 46). Enfin ligne 47, on attend quelques millisecondes avant de boucler.

Par rapport à notre organigramme, le programme gère une sortie « propre » de la boucle sans fin.

Exécution du programme

Pour exécuter ce programme, il faut lancer la commande suivante dans la console Linux :

```
1python3 push01.py
```

Pour sortir de la boucle infinie, il suffit de faire un Ctrl+c au clavier.

Le programme affiche dans la console les messages suivants :

```
1Début du programme

2Sortie par ctrl-c
```

Si vous essayez ce programme en appuyant sur le bouton poussoir, vous constaterez qu'il fonctionne mais fort mal. En effet, chaque appui sur le BP affiche plusieurs lignes indiquant que le BP a été actionné (même pour un appui unique). Vous pouvez également tester ce programme en changeant les valeurs de l'attente (lignes 24 à 27). Certaines valeurs donneront un résultat qui semble acceptable mais on risque, dans ces cas, de ne pas détecter toutes les actions sur le BP.

Commentaires

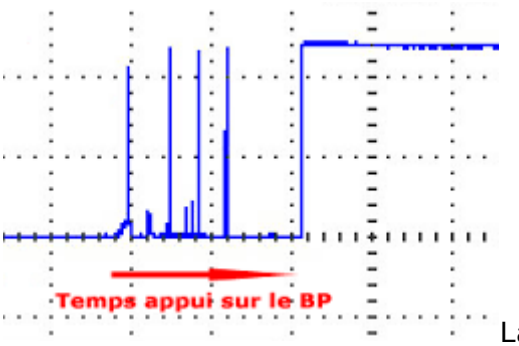
Mais pourquoi notre programme fonctionne-t-il si mal ?
Nous avons deux phénomènes qui viennent perturber notre programme lorsque l'on cherche à lire un contact mécanique.

Le premier est que le programme exécute très rapidement par rapport à notre échelle de temps de pauvre humain. Vous avez pu mettre en évidence ce paramètre lors des différents essais de notre programme.

Le deuxième phénomène est causé par notre bouton poussoir qui n'est pas parfait. D'un point de vue théorique, on avait admis que notre BP avait deux états (non actionné ou actionné). La réalité est tout autre, pour passé de l'état repos à l'état appuyé, il passe par une multitude de rebondissements. Il en va de même pour le passage de l'état appuyé à l'état repos.

Bouton poussoir réel

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure08-1.png



La figure ci-contre représente un passage réel d'un bouton poussoir de l'état repos à l'état actionné. On constate que des pics sont générés par les rebondissements entre dans la zone de basculement de l'entrée.
D'un point de vue physique, ce phénomène de rebondissements est logique puisqu'il faut absorber l'énergie du contact mobile lors de sa frappe sur la partie fixe. Sans entrer dans la théorie des chocs, on se trouve devant une situation que vous connaissez bien. En effet, une balle rebondit plusieurs fois lorsqu'elle tombe au sol, de même, lorsque vous frappez avec un marteau sur une enclume, il rebondit.

Maintenant que faire pour remédier à ce problème ?

Les solutions pour remédier aux rebondissements des BP suscitent de nombreux débats, depuis plus de 30 ans, sans avoir réussi à départager les protagonistes. Deux clans s'affrontent, *d'un côté du ring les «softeux»* et le l'autre *côté du ring les «hardeux»*.
Pour les softeux, on a un problème, il faut le traiter, et pour les hardeux, on a un problème, il faut l'éliminer.
Pas facile de trancher puisque cela fonctionne dans les deux cas.

Pour reprendre notre comparaison précédente, on se trouve dans la situation suivante :
– les softeux ferment les yeux pendant que la balle ou le marteau rebondissent,
– les hardeux utilisent une balle ou un marteau sans inertie.

Côté soft :
Pour faire simple, on lit deux fois l'entrée correspondant au BP avec un temps d'attente entre les deux lectures. Ce temps d'attente correspond au temps nécessaire pour stabiliser les contacts du bouton poussoir (bounce time).

Côté hard :
On met un condensateur (entre 10 et 100nF) en parallèle sur le BP et on lit l'entrée.

Nota : Il y a une scission dans le clan des hardeux entre ceux qui veulent mettre une résistance en série avec le condensateur pour limiter le courant (de charge ou de

décharge) et ceux qui trouvent cela inutile. Personnellement, je trouve cette résistance inutile car ce n'est pas un composant parfait. Le schéma équivalent du condensateur est composé d'un condensateur parfait et de deux résistances (une en parallèle avec le condensateur et l'autre en série). Si l'on ajoute à ces résistances toutes les résistances parasites du circuit la limitation de courant se fera naturellement.

Remarque : Nous n'avons pas abordé, volontairement, toutes les techniques de traitement des anti-rebonds utilisés en logique classique (bascules RS, monostables, triggers de Schmitt, etc…).

Maintenant que vous savez dans quel camp se trouve l'auteur et pour ne pas passer pour un sectaire ou un dogmatique, on utilisera le traitement logiciel des rebonds du bouton poussoir.

Structure des programmes

Un petit rappel sur la structure des programmes sur notre ordinateur ou carte embarquée.

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure09-1-144x300.png

Sur notre ordinateur de bureau ou nano-ordinateur un programme “classique” peut être représenté par l’organigramme ci-contre.

Le programme démarre par une phase d’initialisation puis, on entre dans le traitement de l’applicatif (traitement de texte, dessin, navigateur internet, messagerie, etc…).

A tout moment, notre programme vérifie si une action de sortie de l’applicatif est générée. Si c’est le cas, on quitte le programme en libérant les ressources mises en œuvre.

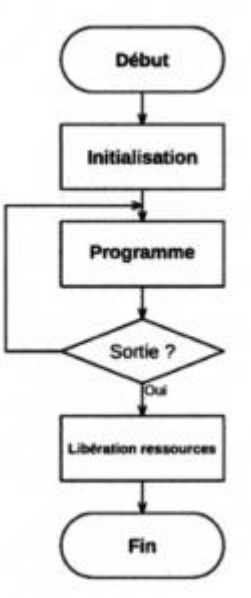


Figure 9

Dans le cas d’une application de type domotique ou contrôle de processus, le programme s’exécute de manière continue sans condition de sortie.

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure10-1-197x300.png

La figure ci-contre représente un organigramme simplifié de la structure du programme.

Notre programme s’exécute dans une boucle infinie et traite les événements pour lesquels il a été développé.
Pour un applicatif de production tout se passe bien, mais pour des tests initiatiques ou d’apprentissage, il faut avoir la possibilité de sortir “proprement” de la boucle infinie sans laisser les ressources dans un état indéfini.

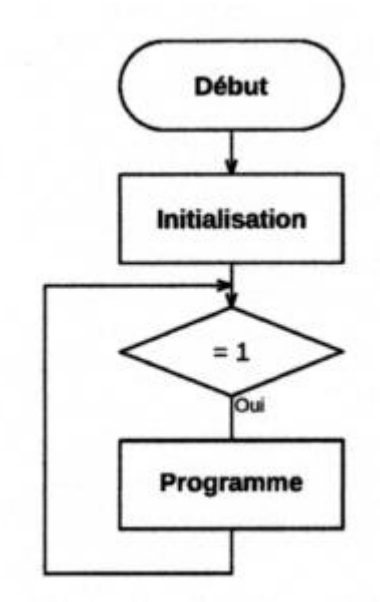


Figure 10

En effet, une broche du GPIO qui reste en sortie peut provoquer un court-circuit fatal à notre entrées/sortie ou au Raspberry Pi. Lorsqu'on relance le logiciel, on risque également d'avoir un message d'erreur parce que l'on alloue une ressource qui est encore occupée.

Dans la suite de nos différentes "saga", nous adopterons l'organigramme ci-dessous.

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure11-300x225.png

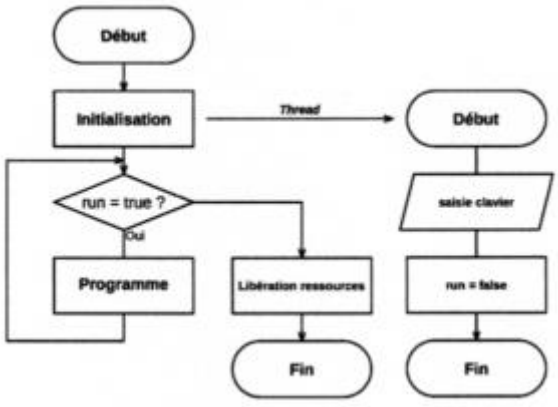


Figure 11 : Organigramme saisie BP

La partie gauche de l'organigramme ressemble étrangement à l'organigramme de la figure 10 avec un contrôle de la boucle infinie (run = true). Si cette variable run passe de true (vrai) à false (faux), on sort de la boucle en libérant les ressources.
Pour changer la valeur de cette variable, nous utiliserons un thread (créé dans la

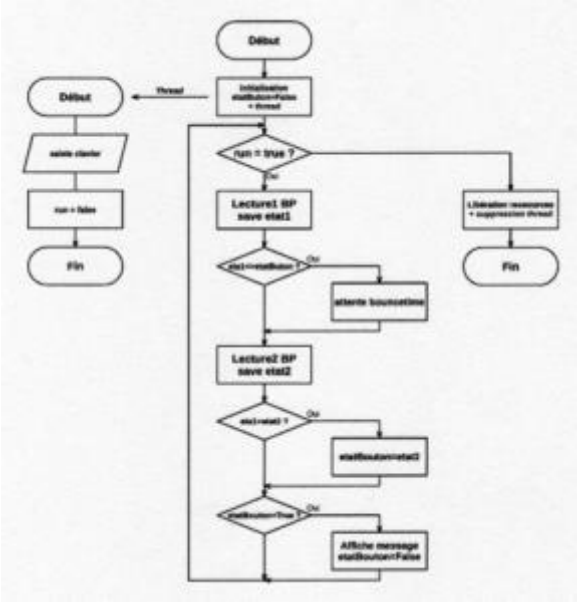
phase initialisation) qui permettra une saisie clavier, en occurrence l'appuie sur la touche <Entrée> du clavier. Pour le fonctionnement des threads, on pourra consulter le chapitre 6 de la saga blink, la suite.
Le passage d'un programme selon l'organigramme de la figure 11, à un programme selon l'organigramme de la figure 10 sera commenté dans le paragraphe suivant.

Programme (anti-rebond logiciel)

Pour la suite, nous continuons à utiliser le langage Python avec la bibliothèque RPi.GPIO.

Organigramme

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure12-287x300.png



Dans un premier temps, on initialise le port GPIO ainsi que les variables et le thread. L'étape suivante consiste à lire l'état du BP et de le sauvegarder. Le premier test vérifie etat1 est différent de etatBouton. Si c'est le cas on attend que les régimes transitoires soient terminés. Puis on a une seconde lecture de l'état du BP avec sauvegarde de l'état. Ensuite, on compare etat1 et etat2, si les deux variables sont égales alors on a un appui BP (etatBouton = etat2).
Et pour finir, si etatBouton = True alors on affiche notre message et on remet notre variable etatBouton à False.
En faisant <Entrée> au clavier, on sort de la boucle de scrutation. A la sortie, on libère les ressources et l'on supprime le thread.

Programme

Pour saisir ce programme, il faut saisir dans la console :

```
1nano push05.py
```

Cette commande ouvre un fichier texte vide appelé push05.py (pour la sauvegarde faire ctrl+o et pour sortir ctrl+x).
Le contenu du programme et de ses commentaires sont représentés ci-dessous.

```
#!/usr/bin/python3
# -*- coding:utf-8 -*-
"""
Programme classique lecture entrée GPIO avec la bibliothèque RPi.GPIO
utilisation anti-rebond logiciel
Bouton poussoir raccordé entre GPIO22 et +3.3V
(avec résistance de protection de 1k en série)
nom programme       : push05.py
logiciel            : python 3.4.2
cible               : raspberry Pi
date de création    : 18/08/2016
date de mise à jour : 18/08/2016
version             : 1.0
auteur              : icarePetibles
référence           :
"""
#-------------------------------------------------------------------------------
# Bibliothèques
#-------------------------------------------------------------------------------
import RPi.GPIO as GPIO                 #bibliothèque RPi.GPIO
import time                             #bibliothèque time
from threading import *       #bibliothèque thread
#-------------------------------------------------------------------------------
pin = 22              #broche utilisé en entrée
debounce = 100           #attente anti-rebond
etatBouton = False          #bouton au repos

GPIO.setwarnings(False)                 #désactive le mode warning
```

```
GPIO.setmode(GPIO.BCM)                    #utilisation des numéros de ports du
                                          #processeur
GPIO.setup(pin, GPIO.IN, pull_up_down=GPIO.PUD_DOWN)
                                          #mise en entrée du port GPIO 22
                                          #et activation résistance soutirage
                                          #au ground
enMarche = True           #sortie boucle


class Saisie(Thread):
    """Lanceur de fonction"""
    def __init__(self, fonction):
        Thread.__init__(self, None, fonction)
        self._fonction = fonction
        self.running = True

    def run(self):                        #lance le programme
        while self.running:
            self._fonction()              #notre fonction programme

    def stop(self):                       #arrêt du programme
        self.running = False

def saisieClavier():
    """Saisie <Entrée>"""
    global enMarche
    while enMarche:                       #tant que en marche
        input()                           #attente saisie clavier
        sortie = True        #on sort
```

```
        enMarche = False       #sortie boucle

if __name__ == '__main__':
    """
    Programme par défaut
    """
    print("Début du programme")          #IHM
    print("Sortie par <Entrée>\n")       #IHM
    thread_1 = Saisie(saisieClavier)     #création thread
    thread_1.start()                     #démarre le thread
    while enMarche:                      #boucle infinie contrôlée
        lecture1 = GPIO.input(pin)       #première lecture entrée
        etat1 = lecture1                 #mémorise la lecture
        if(etat1 != etatBouton):         #si première lecture différente
                                         #etatBouton
            time.sleep(debounce / 1000) #attente fin rebondissement
contact
        lecture2 = GPIO.input(pin)       #deuxième lesture entrée
        etat2 = lecture2                 #mémorise la lecture
        if(etat1 == etat2):              #si etat1 = etat2
            etatBouton = etat2           #alors changement état BP

        if(etatBouton == True):          #si BP appuyé
            print("BP appyué")           #alors affiche message
            etatBouton = False     #BP relaché

    thread_1.stop()                      #arrêt thread
    GPIO.cleanup()                       #libère toutes les ressources
    print("Fin du programme")            #IHM
```

```
2

7
3

7
4

7
5

7
6

7
7

7
8

7
9

8
0

8
1

8
2

8
3

8
4
```

Dans le programme ci-dessus, on retrouve les différentes phases de notre organigramme.

Exécution du programme

Pour exécuter ce programme, il faut lancer la commande suivante dans la console Linux :

```
1python3 push05.py
```

Pour sortir de la boucle infinie, il suffit de faire un <Entrée> au clavier.

C'est magique, à chaque appui sur notre BP, on a l'affichage du message « BP appuyé »

Commentaires

Pour rendre notre programme compatible avec la version de la figure 10, il suffit de commenter (# en début de ligne) la ligne 22, les lignes 37 à 57, les lignes 63 à 66 et les lignes 82 à 84.

On aurait pu faire nettement mieux en utilisant une variable test ou debug (True ou False) pour avoir les deux versions dans le même fichier. Nous verrons cette option lors d'un autre chapitre.

Remarque :
Les puristes de python auraient adoptés une structure de la forme ci-dessous avec les exécutions spécifiques (lignes 4 et 5) :

```
1  #!/usr/bin/python3
2  # -*- coding:uft-8 -*-
3  #
4  # python3 -O test.py -> debug on
5  # python3 test.py    -> debug off
6  #
7  if __debug__:
       print("Debug ON")
8  else:
       print("Debug OFF")
9
10
```

Autres solutions

Lorsqu'on consulte les fonctions de notre bibliothèque RPi.GPIO, on constate qu'il existe d'autres possibilités pour lire l'état du BP. On peut explorer les fonctions GPIO.event_detected(), GPIO.wait_for_edge() et GPIO.add_event_detect (avec bounce time).

L’utilisation des bibliothèques permet une programmation plus concise et réduit considérablement le nombre de lignes de code à saisir. Par contre, leur utilisation n’est pas toujours très formatrice car elles cachent une partie du code utilisé et l’organigramme correspondant. Il ne faut néanmoins pas hésiter à consulter et re-consulter la documentation, parce qu’elles contiennent, presque toujours, des fonctions « miracles » pour résoudre vos problèmes.

Langage Python

Si l'on utilise le même organigramme, on aura toujours le même résultat peu importe le langage utilisé. Dans suite, on va mettre en œuvre une particularité de la bibliothèque gpiozero pour détecter l'appui sur le bouton poussoir. La méthode de la bibliothèque fait un travail similaire à notre organigramme.

gpiozero

Programme

Pour saisir ce programme, il faut saisir dans la console :

```
1nano push25.py
```

Cette commande ouvre un fichier texte vide appelé push25.py (pour la sauvegarde faire ctrl+o et pour sortir ctrl+x).
Le contenu du programme et de ses commentaires sont représentés ci-dessous.

```
1  #!/usr/bin/python3
2  # -*- coding:utf-8 -*-
3  """
4  Programme classique lecture entrée GPIO avec la bibliothèque gpiozero
5  utilisation anti-rebond logiciel
6  Bouton poussoir raccordé entre GPIO22 et +3.3V
7  (avec résistance de protection de 1k en série)
8  nom programme       : push25.py
9  logiciel            : python 3.4.2
10 cible               : raspberry Pi
11 date de création    : 18/08/2016
12 date de mise à jour : 18/08/2016
13 version             : 1.0
14 auteur              : icarePetibles
15 référence           :
16 """
17 #---------------------------------------------------------------------------
```

```
-------
# Bibliothèques
#-------------------------------------------------------------------------
-------
from gpiozero import Button             #bibliothèque gpiozero
import time                              #bibliothèque time
from threading import *       #bibliothèque thread
#-------------------------------------------------------------------------
-------
pin = 22              #broche utilisé en entrée
debounce = 0.1           #attente anti-rebond

bouton = Button(pin, pull_up=False, bounce_time=debounce)
                                         #instance bouton sur GPIO22 avec
                                         #résistance de soutirage vers la
masse
                                         #et temps de rebondissement de
100 ms
enMarche = True           #sortie boucle

class Saisie(Thread):
    """Lanceur de fonction"""
    def __init__(self, fonction):
        Thread.__init__(self, None, fonction)
        self._fonction = fonction
        self.running = True

    def run(self):                       #lance le programme
        while self.running:
            self._fonction()             #notre fonction programme
```

```
    def stop(self):                         #arrêt du programme
        self.running = False


def saisieClavier():
    """Saisie <Entrée>"""
    global enMarche
    while enMarche:                         #tant que en marche
        input()                             #attente saisie clavier
        sortie = True         #on sort
        enMarche = False       #sortie boucle


if __name__ == '__main__':
    """
    Programme par défaut
    """
    print("Début du programme")             #IHM
    print("Sortie par <Entrée>\n")          #IHM
    thread_1 = Saisie(saisieClavier)    #création thread
    thread_1.start()                        #démarre le thread
    while enMarche:                         #boucle infinie contrôlée
        if bouton.is_pressed:               #si BP appuyé
            print("BP appyué")              #alors affiche message
        time.sleep(0.15)                    #ralentit le programme


    thread_1.stop()                         #arrêt thread
    bouton.close()                          #libère la ressource
    print("Fin du programme")               #IHM
```

6
5

6
6

6
7

6
8

6
9

7
0

Notre programme n'a rien de compliqué car le rebondissement de contacts est géré par l'instance bouton (ligne 27).

La class Button possède de nombreuses méthodes (voir documentation : http://gpiozero.readthedocs.io/en/v1.2.0/api_input.html)

Exécution du programme

Pour exécuter ce programme, il faut lancer la commande suivante dans la console Linux :

```
1python3 push25.py
```

Pour sortir de la boucle infinie, il suffit de faire un <Entrée> au clavier.

Commentaires

Pour rendre notre programme compatible avec la version de la figure 10, il suffit de commenter (# en début de ligne) la ligne 22, les lignes 33 à 53, les lignes 59 à 62 et les lignes 68 à 70.

wiringpi et pigpio

Nous n'avons pas présenté d'exemples utilisant les bibliothèques Python pigpio ou wiringpi. Mais cela ne devrait pas vous poser de problèmes pour réaliser ces exercices.

Langage C

Nous utiliserons pour cet exemple (suivant l'organigramme de la figure 11) la bibliothèque wiringPi. Il existe également une bibliothèque bcm2835 qui, bien que très performante, n'est pas très avenante. Nous l'utiliserons certainement pour l'un de nos articles a venir.

Programme

Pour saisir ce programme, il faut saisir dans la console :

```
1nano push10.c
```

Cette commande ouvre un fichier texte vide appelé push10.c (pour la sauvegarde faire ctrl+o et pour sortir ctrl+x).
Le contenu du programme et de ses commentaires sont représentés ci-dessous.

```
1  /* ----------------------------------------------------------------------
   -------
2  Programme classique lecture entrée GPIO avec la bibliothèque wiringPi
3  Bouton poussoir raccordé entre GPIO22 et +3.3V
4  (avec une résistance de protection de 1k en série)
5  nom programme       : push10.c
6  os                  : RPi Linux 4.4.13+
7  logiciel            : gcc (Raspbian 4.9.2-10) 4.9.2
8  cible               : raspberry Pi
9  date de création    : 24/08/2016
10 date de mise à jour : 24/08/2016
11 version             : 1.0
12 auteur              : icarePetibles
13 référence           : www.wiringpi.com
14 Remarques           :
15
16 Bibliothèques
17 ----------------------------------------------------------------------
   ---- */
```

```
#include <stdio.h>                      //entrées/sorties
#include <wiringPi.h>                   //bibliothèque wiringPi
#include <stdlib.h>                     //conversion nombre
#include <string.h>                     //chaîne
#include <pthread.h>                    //thread

#define TEST                            //voir commentaire

#define PIN 3                           //numéro bouton = GPIO22
#define DEBOUNCE 100                    //temps rebondissement

typedef enum            //pour palier non existance type booléen
    {False=0, True=1}
    Bool;

volatile Bool enMarche = True;          //variable globale contrôle boucle

#ifdef TEST
void* saisieClavier(){                  //attente saisie clavier (thread)
    while(enMarche){                    //en fonctionnement
        getchar();                      //attente saisie clavier
        enMarche = False;               //arrêt
    }
    return 0;                           //valeur retour
}
#endif

int main(void){                         //programme principal
    Bool etatBouton = False;            //variable état du bouton

```

```
    Bool lecture1 = False, lecture2 = False;
    Bool etat1 = False, etat2 = False;
    wiringPiSetup();                    //numérotation wiringPi ou type Arduino
    pinMode(PIN, INPUT);                //pin en entrée
    pullUpDnControl(PIN, PUD_DOWN);     //résistance soutirage à la masse
#ifdef TEST
    pthread_t saisie;                   //instance thread
    printf("Début programme\n");        //IHM
    printf("sortie par touche <Entrée>\n\n");
                                        //IHM
    pthread_create(&saisie, NULL, saisieClavier, NULL);
                                        //création thread
#endif
    while(enMarche){                    //tant que l'on fonctionne
        lecture1 = digitalRead(PIN);    //lecture état BP
        etat1 = lecture1;               //sauve le résultat
        if(etat1 != etatBouton)         //si différent de l'état actuel
            delay(DEBOUNCE);            //attente fin rebondissement
        lecture2 = digitalRead(PIN);    //re-lecture état BP
        etat2 = lecture2;               //sauve le résultat
        if(etat1 == etat2)              //compare les 2 lectures sont identiques
            etatBouton = etat2;         //nouvel état


        if(etatBouton == True){         //si BP appuyé
            printf("BP appuyé\n");      //IHM
            etatBouton = False;         //remise à 0
        }
    }
#ifdef TEST
```

```
    printf("Fin du programme\n");       //IHM
#endif
    return(0);                          //code sortie
}
//---------------------------------------------------------------------------
-------
```

5

6
6

6
7

6
8

6
9

7
0

7
1

7
2

7
3

7
4

7
5

7
6

7
7

7
8

7
9

8
0

Notre programme est en tout point similaire au programme push05.py à la syntaxe près.

Exécution du programme

Pour exécuter ce programme, il faut, avant tout, le compiler par la commande suivante dans la console Linux :

```
1gcc -Wall -o push10 push10.c -lwiringPi -lpthread
```

Pour l'exécuter, il faut lancer le programme par la commande console :

```
1sudo ./push10
```

A chaque action sur le bouton poussoir, on verra le message console "BP appuyé".

Pour sortir de la boucle infinie, il suffit de taper sur la touche <Entrée> du clavier.

Commentaires

Le programme push10 s'exécute conformément à la structure de l'organigramme de la figure 11 pour le rendre exécutable selon l'organigramme de la figure 10, il suffit de commenter la ligne 24.

```
1//#define TEST                  //voir commentaire
```

Puis il faudra recompiler le programme comme décrit ci-dessus.
Toutes les lignes comprises entre les différents #ifdef TEST et #endif ne seront pas compilées.

Les seuls messages affichés dans la console seront "BP appuyé".
Pour sortir uniquement un ctrl-c viendra à bout de la boucle infernale.

Langage C++

Programme

Pour saisir ce programme, il faut saisir dans la console :

```
nano push20.cpp
```

Cette commande ouvre un fichier texte vide appelé push20.cpp (pour la sauvegarde faire ctrl+o et pour sortir ctrl+x).
Le contenu du programme et de ses commentaires sont représentés ci-dessous.

```
/* ------------------------------------------------------------------------
Programme classique lecture entrée GPIO avec la bibliothèque wiringPi
Bouton poussoir raccordé entre GPIO22 et +3.3V
(avec une résistance de protection de 1k en série)
nom programme       : push20.cpp
os                  : RPi Linux 4.4.13+
logiciel            : g++ (Raspbian 4.9.2-10) 4.9.2
cible               : raspberry Pi
date de création    : 24/08/2016
date de mise à jour : 24/08/2016
version             : 1.0
auteur              : icarePetibles
référence           : www.wiringpi.com
Remarques           :

Bibliothèques
------------------------------------------------------------------------
---- */
#include <cstdio>                       //entrées/sorties
#include <wiringPi.h>                   //bibliothèque wiringPi
#include <iostream>                     //entrées/sorties standard
```

```
#include <sstream>                          //buffer
#include <stdlib.h>                        //conversion nombre
#include <pthread.h>                       //thread

bool lectureBP(int pin, bool etatBouton);
                                           //prototype fonction lecture BP

using namespace std;

#define TEST                               //voir commentaire

#define PIN 3                              //numéro bouton = GPIO22
#define DEBOUNCE 100                       //temps rebondissement

volatile bool enMarche(true);             //variable globale contrôle boucle

#ifdef TEST
void* saisieClavier(void *){              //attente saisie clavier (thread)
    while(enMarche){                       //en fonctionnement
        cin.ignore();                      //attente saisie clavier
        enMarche = false;                  //arrêt
    }
    return 0;                             //valeur retour
}
#endif

int main(void){                           //programme principal
    bool etatBouton(false);               //variable état du bouton
    wiringPiSetup();                       //numérotation wiringPi ou type
Arduino
```

```
    pinMode(PIN, INPUT);                //pin en entrée
    pullUpDnControl(PIN, PUD_DOWN);     //résistance soutirage à la masse
#ifdef TEST
    pthread_t saisie;                   //instance thread
    cout << "Début programme" << endl;  //IHM
    cout << "sortie par la touche <Entrée>" << endl << endl;
                                        //IHM
    pthread_create(&saisie, NULL, saisieClavier, NULL);
                                        //création thread
#endif
    while(enMarche){                    //tant que l'on fonctionne
        if(lectureBP(PIN, etatBouton)){ //lecture BP
            cout << "BP appuyé" << endl;//IHM
            etatBouton = false;         //remise à 0
        }
    }
#ifdef TEST
    cout << "Fin du programme" << endl;
#endif
    return(0);                          //code sortie
}
//---------------------------------------------------------------------------
bool lectureBP(int pin, bool etatBouton){
    bool lecture1(false), lecture2(false);
    bool etat1(false), etat2(false);    //variables booléen et
initialisation
    lecture1 = digitalRead(pin);        //lecture état BP
    etat1 = lecture1;                   //sauve le résultat
    if(etat1 != etatBouton)             //si différent de l'état actuel
        delay(DEBOUNCE);                //attente fin rebondissement
```

```
    lecture2 = digitalRead(pin);        //re-lecture état BP
    etat2 = lecture2;                   //sauve le résultat
    if(etat1 == etat2)                  //compare les deux lectures
        etatBouton = etat2;             //nouvel état
    return etatBouton;                  //retourne état du bouton
}
//-------------------------------------------------------------------------
-------
```

```
4

8
5
```

Pour éviter la monotonie de tous ces programmes qui se ressemblent (à la syntaxe près), nous avons déporté la lecture du bouton poussoir dans un sous programme. Le fonctionnement reste, bien sur, le même.

Exécution du programme

Pour exécuter ce programme, il faut, avant tout, le compiler par la commande suivante dans la console Linux :

```
1g++ -Wall -o push20 push20.cpp -lwiringPi -lpthread
```

Pour l'exécuter, il faut lancer le programme par la commande console :

```
1sudo ./push20
```

A chaque action sur le bouton poussoir, on verra le message console "BP appuyé".

Pour sortir de la boucle infinie, il suffit de taper sur la touche <Entrée> du clavier.

Commentaires

L'utilisation de la directive TEST est la même que dans le chapitre précédent.

Langage Bash

Nous sommes émerveillés par les possibilités et la puissance des scripts bash. L'usage des scripts ne sera, certainement, pas très courant dans l'exploitation des entrées/sorties GPIO de la carte framboise. Donc juste pour le "fun", un programme similaire au précédent.

Programme

Pour saisir ce programme, il faut saisir dans la console :

```
1nano push30.sh
```

Cette commande ouvre un fichier texte vide appelé push30.sh (pour la sauvegarde faire ctrl+o et pour sortir ctrl+x).
Le contenu du programme et de ses commentaires sont représentés ci-dessous.

```
  #!/bin/bash
1
  #Programme classique lecture entrée GPIO via driver linux
2
  #Anti-rebond logiciel
3
  #Bouton poussoir raccordé entre GPIO22 et +3.3V
4
  #(avec résistance de protection de 1k en série)
5 #nom programme       : push30.sh
6 #logiciel            : bash
7 #cible               : raspberry Pi
8 #date de création    : 18/06/2016
  #date de mise à jour : 18/07/2016
9
  #version             : 1.0
10
  #auteur              : icarePetibles
11
  #référence           :
12
  #Remarques           : Pour fonctionner, il faut être sous super
13utilisateur
  #                      Il faut rendre le fichier exécutable avec
14
  #                      sudo chmod +x push30.sh
15
  #                      Pour exécuter le fichier, il faut faire
16
  #                      sudo ./push30.sh
```

```
#
BPPIN=22                                                #GPIO22
FALSE=0
TRUE=1
DEBOUNCE=0.05                                           #temps rebondissement
etatBouton=0                                            #BP relaché

#vérification si accès root
#if [ $EUID -ne 0 ]                                     #si $EUID différent
de 0
if [ $EUID != &quot;0&quot; ]                                        #si $EUID
différent de 0
then                    #alors
    echo &quot;Il faut être root pour exécuter le script. Essaye sudo
$0&quot;
    exit                #sortie
fi                      #fin if

#Procédure de netoyage, désactive l'entrée
cleanup()
{
    PIN=$1                                              #paramètre transmis
    echo $PIN &gt; /sys/class/gpio/unexport     #désactive port
    echo                #esthétique
    echo Fin script                                     #message de fin
    exit                #sortie
}

#programme principal
echo Début du script                                    #IHM
```

```
44 echo ctrl-c pour sortir de la boucle                #IHM
45 echo
46 #Setup pin et direction - Cature Control-C SIGHUP SIGKILL
47 echo $BPPIN &gt; /sys/class/gpio/export       #assigne pin 23
48 echo in &gt; /sys/class/gpio/gpio$BPPIN/direction   #pin 22 en entrée
49 trap &quot;cleanup $BPPIN&quot; SIGHUP SIGINT SIGTERM         #capture SIGNAL et lance la
50                                                      #la procédure cleanup avec
51                                                      #le paramètre 22
52 while true                  #boucle infinie
53   do                    #faire
54     lecture1=<code>cat /sys/class/gpio/gpio$BPPIN/value</code> #lecture entrée BP
55     etat1=$lecture1                                  #mémorise la lecture
56     if [ $etat1 -ne $etatBouton ]                   #si lecture &lt;&gt; état BP
57     then
58         sleep $DEBOUNCE                              #attente fin rebondissement
59     fi
60     lecture2=<code>cat /sys/class/gpio/gpio$BPPIN/value</code> #re-lecture entrée BP
61     etat2=$lecture2                                  #mémorise la lecture
62     if [ $etat1 -eq $etat2 ]                        #si etat1 = etat2
63     then
64         etatBouton=$etat2                            #alors changement état BP
65     fi
66
67     if [ $etatBouton -eq $TRUE ]                    #si BP appuyé
68     then
69
70
```

```
71          echo BP appuyé                                  #affiche message
72          etatBouton=$FALSE                               #BP relâché
73      fi
74  done                    #fin do
75  #Fin du script
```

Exécution du programme

Pour exécuter ce programme bash, il faut le rendre exécutable avec la commande :

```
1sudo chmod +x push30.sh
```

Et pour l'exécuter :

```
1sudo ./push30.sh
```

Pour sortir de la boucle infinie, il suffit de faire un ctrl-c au clavier.

Conclusion

image: https://www.framboise314.fr/wp-content/uploads/2016/08/figure13.png

Tout une histoire pour un simple bouton poussoir !!! Par contre, nous avons abordé des sujets complémentaires (schémas, logiciels et concepts) qui nous servirons dans d'autres articles de la "saga".
Il reste des sujets intéressants autour du BP comme la détection des appuis courts, longs ou incrémentant un compteur à vitesse progression et bien d'autres.
Peut être, une suite dans quelque temps, on ne sait jamais !

Les plus courageux pourront faire un mixte de la "sagaBlink" et "sagaPushButton" pour faire, par exemple :
– une led que l'on allume avec un bouton poussoir et que l'on éteint avec un autre bouton,
– une led que l'on allume et que l'on éteint avec le même bouton poussoir,
– deux leds clignotantes à des fréquences différentes que l'on fait clignoter avec chacune un BP,
– etc…

Le prochain épisode sera nettement plus intéressant et portera sur la **"saga I2C"**

A tous présents et à venir. Salut !

Sources

On pourra télécharger l'ensemble des sources sous :
http://psl.ibidouille.net/Images_forum/raspberryPi/sagaPush.zip

En savoir plus sur https://www.framboise314.fr/le-bouton-poussoir-un-composant-banal-o-combien-etonnant/#upVhLTDm1BMA38jM.99

---

19 réflexions au sujet de « Le bouton poussoir un composant banal, ô combien, étonnant. »

1.  ***fred*** 5 septembre 2016 à 15 h 24 min

   Ah ah voici donc la suite … bon j'ai fait une première lecture en diagonale j'y reviendrai plus tard …

   Au début je me suis dit … ATTENTION TERRAIN GLISSANT vais je devoir mettre un carton rouge (je déconne hein) en voyant de beau schémas avec des boutons poussoir sans anti-rebond … et non finalement y a un petit truc écrit dessus, étant ancien électronicien c'est le 1er truc qui me vient quand je vois ce genre de montage, j'ai du être traumatisé par mes profs

   donc je suppose que le reste doit être complet aussi …

   Dans tous les cas merci pour le travail

2.  ***S.POURRE*** 6 septembre 2016 à 17 h 59 min

   Bonjour Patrice,

   Je suis toute la saga d'un œil attentif et j'attends avec impatience la suite

   image: https://s.w.org/images/core/emoji/2.4/svg/1f609.svg

   Comme je l'ai déjà indiqué je suis un vieux crocodile formé à UNIX, au C de Ritchie &

Kernighan et au fork dans les années 85 (un autre millénaire).
Il y a donc de la remise à niveau à faire (j'ai bien essayé su apt-get dist-upgrade

image: https://s.w.org/images/core/emoji/2.4/svg/1f609.svg

).
Heureusement, avec UNIX (et donc Linux) les fondamentaux demeurent malgré certains choix plus ou moins heureux (comme l'abandon de sysinit au profit de systemd sur une nano-machine).
J'attends donc la saga de l'I2C (puis du SPI) en espérant y trouver une solution à l'emploi des interruptions. La bibliothèque wiringpi semble le prévoir expressément mais je n'ai blink-tempo-cross-FRICHEpas trouvé d'exemple et ce n'est pas trivial (un modo, adepte de robots, a bien essayé mais sans résultat malgré son très bon niveau). Le bus I2C et surtout le bus SPI présentent un autre débit qu'une IHM. Beaucoup de Circuits Intégrés I2C présentent une broche INT pour éviter de gaspiller des ressources CPU dans une boucle de polling.
AMHA, s'il y a eu peu de réactions à tes 2 derniers articles, ce n'est pas du à un manque d'intérêt mais au fait des vacances et que, maintenant, ta "cible" a été clairement définie.
en attendant, je teste mon environnement de cross compilation et je continue mes recherches sur le net.

Répondre ↓

1.

Bud Spencer 6 septembre 2016 à 22 h 02 min

Salut Sylvain.
Ca fait plusieurs fois que je te vois évoquer ton blocage sur la capture d'une interruption hard, mais c'est quoi au juste tes besoins ? La fréquence minimum de détection est si élevée que ca que tu ne t'en sort pas ?

3.

S.POURRE 6 septembre 2016 à 23 h 58 min

Bonsoir Bud,

C'est un projet à long terme qui me sert de fil rouge dans mes essais, pour ne pas partir dans tous les sens.
L'idée est de piloter entièrement une station radioamateur "faite maison" et portable car je suis en ville, sans grand dégagement.
Le tout serait alimenté par un panneau solaire orientable (pour améliorer le rendement) et une batterie et pourrait prendre la forme d'un coffret étanche, posé au pied du mat (pour limiter les pertes dans les câbles d'alimentation et coaxiaux) et piloté à distance.

Beaucoup de fonctions peuvent être lancées en daemon mais les actions de l'opérateur humain (changement de fréquence, de mode, rappel ou mise en mémoire, de direction des antennes…) doivent être prises immédiatement en compte (à l'échelle humaine, soit une poignée de millisecondes).
Pendant ce temps, le programme doit assurer pas mal de taches (afficher le niveau de réception, la puissance émise, la puissance réfléchie, faire tourner un SDR genre Gnuradio (très gros consommateur de CPU), afficher le spectre, décoder et afficher des modes numériques, calculer la correction de l'effet Doppler (pour les satellites)….
Sans se lancer dans du temps réel (même mou), Il n'est donc pas envisageable (pour moi) de faire une grosse boucle infinie de polling de tous les circuits et j'envisage une solution mixte à base de timer pour les fonctions non prioritaires et d'interruptions pour les autres.
Je compte utiliser ou recycler mes connaissances et donc réaliser ce projet en C, donc compilé (rapidité, opérateurs binaires..), en plusieurs fichiers (donc headers, prototypes et C), avec un gestionnaire de sources (versions) et de dépendances (CMAKE ?) pour ne recompiler que ce qui est nécessaire.
Ce projet me semble trop lourd (ambitieux ?) pour être géré sur un Raspberry (même 3) d'où mon choix de cross compiler sur ma tour (I5 + 24 Go de RAM). Je suis assez réfractaire aux IDE (genre éclipse) et VI me convient très bien .
Pour ne pas polluer cet article et faire fuir les vrais débutants, je te suggère soit:
– d'attendre l'article de Patrice et de réagir, comme tu l'as déjà fait, en apportant des précisions d'optimisation.
– de faire un article dédié, suite de ton premier article, sur l'exploitation des interruptions avec la bibliothèque Wiringpi.
– de me dire si tu préfères que j'ouvre un fil dans la rubrique programmation.
En attendant, je te remercie déjà de t'intéresser à mes problèmes (qui sont aussi ceux de certains qui exploitent des cartes I2Cde pilotage de servos).
Sylvain

1.

 ***Patrice SEIBEL*** *Auteur de l'article*7 septembre 2016 à 9 h 31 min

Bonjour à tous,
Pour avoir de véritables interruptions "hard", il faut titiller le noyau de Linux et cela sort du cadre que je me suis fixé. Mais c'est possible.
Par contre, il existe des solutions, un peu moins "real time" mais tout à fait acceptable. Par contre, on restera toujours devant le choix cornélien performance/charge processeur. C'est le choix que j'ai fait pour la suite de la saga. Je reste ouvert, bien sur, à d'autres pistes.
La dernière solution serait d'utiliser un composant spécifique pour le temps réel qui communiquerait avec les tâches de plus niveaux via un bus série. Une carte de la famille Arduino pourrait très bien remplir cette fonction (ajout à ma liste todo) à moindre coût (€ et développement).

nota :
Dans la préparation de cette article, j'avais prévu un chapitre sur l'exploitation

du BP par interruption mais pour réduire la taille, il est passé à la trappe. Il ne reste plus que la trace chapitre 6.5

1.

 ***S.POURRE*** 7 septembre 2016 à 11 h 26 min

Je comprends très bien qu'un article d'initiation ne puisse aborder des techniques "trop pointues", au risque de décourager le lectorat ciblé.

Si le $ est déjà écrit, ce serait tout de même dommage de ne pas le publier compte tenu de l'effort de recherche et de rédaction que cela a du te coûter.

Puis-je te suggérer de l'ajouter à la fin de l'article, comme voie à explorer pour aller plus loin ou de le mettre dans un commentaire ?

Merci encore pour ces articles.
Cordialement

Sylvain

4.

 ***François MOCQ*** 7 septembre 2016 à 9 h 14 min

Merci Patrice pour cet article
il y a juste une phrase avec laquelle je ne suis pas d'accord :
"Le prochain épisode sera nettement plus intéressant et portera sur la « saga I2C »"
à mon avis savoir gérer la lecture de l'état d'un BP est aussi intéressant que de gérer un composant I2C

image: https://s.w.org/images/core/emoji/2.4/svg/1f642.svg

c'est différent mais au moins aussi utile
@+

1.

 ***Didier Bon*** 5 octobre 2017 à 8 h 17 min

Une question sur la partie bash, pourquoi écrire:
lecture2=`cat /sys/class/gpio/gpio$BPPIN/value`
je ne connais pas cette balise
`merci pour l'explication car cela m'intrigue.`
`Bien cordialement`

1.

 ***Didier Bon*** 5 octobre 2017 à 8 h 25 min

je parlai de la balise code qui a sauté lors de la validation de mon post
cette fois je mets pas les “”
donc je voulais dire:

Une question sur la partie bash, pourquoi écrire:
lecture2= code cat /sys/class/gpio/gpio$BPPIN/value code
je ne connais pas cette balise code
merci pour l’explication car cela m’intrigue.
Bien cordialement

5.

 ***Denis Brion*** 7 septembre 2016 à 12 h 32 min

Il y a deux points qui m’ont intrigué:

a) pourquoi éditer un texte pour mettre en et hors service des portions de code c (++++++) : l’option -DTEST le fait très bien depuis un millénaire … et n’est pas plus compliquee à introduire pour des débitants que le lien avec une bibli
othèque (et pourra donc les inciter à trouver d’autres options)

b) quelle est la vitesse maximale que l’on peut échantillonner (ex : compter les changements d’état/ fronts montants). J’ai fait l’inverse avecun RPi couplé à .
.. un fréquencemètre et j’ai trouvé que, sans modifier les functions de delai (g ranularité : la milliseconde), il arrivait… à 500 hz (oéridoe ON : 1 ms/OFF 1 ms : peut ere peut on faire une PWM du pauvre..)

c) jamais 2 sans 3- quelle est la charge processeur induite (je crois qu’elle e st très faible, mais je ne sais pas très bien lire “top” sur un multicore

Cet article est suffisamment bon pour susciter la curiosité … et je vous en remercie.
Sincèrement

1.

 ***Bud Spencer*** 8 septembre 2016 à 19 h 40 min

500 Hz … tu est très loin du compte. Juste avec un PI1, en c,c++ tu peux tourner autour de 5 Mhz avec les lib's wiringPI ou bcm2835 et même dépasser les 20 Mhz en natif. A titre de complaisons, sur le même PI, RPI.GPIO et python passe difficilement les 50 Khz …

1.

 ***icare*** 8 septembre 2016 à 23 h 40 min

http://codeandlife.com/2012/07/03/benchmarking-raspberry-pi-gpio-speed/

1.

 ***Bud Spencer*** 9 septembre 2016 à 9 h 43 min

Ou mieux encore, la version mise à jour du bench qui compare PI1 et PI2
http://codeandlife.com/2015/03/25/raspberry-pi-2-vs-1-gpio-benchmark/

Pour comparer J'avais refait ces tests chez moi avec un oscillo pro et j'arrivais sensiblement au même résultat

6.

 ***Bud Spencer*** 7 septembre 2016 à 23 h 25 min

Sympa le projet (je suis Radioamateur aussi. F1— Promo 1990 ~ 1992, je ne sais plus exactement

image: https://s.w.org/images/core/emoji/2.4/svg/1f609.svg

). A mon avis tu fais tout un blocage sur un problème qui en réalité n'en est pas un. Lever en une 'poignée de milliseconde' quelques interruptions extérieures sur différentes entrées n'est pas un problème avec un PI3. Par contre, toutes les traiter correctement, ça,

peut vite devenir un casse-tête ;). Mais effectivement, ce n'est pas l'endroit pour parler de ça mais a l'occase, pourquoi pas un petit tuto 'expérimental' sur le forum.

1.

S.POURRE 8 septembre 2016 à 2 h 03 min

C'est pas grave, j'avais corrigé (comme lortograf de mon nom)

image: https://s.w.org/images/core/emoji/2.4/svg/1f609.svg

Je crois qu'on est plusieurs OM sur ce site dont le maître des lieux lui-même, François, en est un aussi (radioamateur oeuf corse).
Il va falloir qu'il modifie le formulaire d'inscription avec un champ "indicatif" et une case à cocher "acceptez-vous l'affichage de votre indicatif"

image: https://s.w.org/images/core/emoji/2.4/svg/1f609.svg

1.


François MOCQ 8 septembre 2016 à 8 h 20 min

ahah bonjour
oui je pense que le Raspberry Pi a de beaux jours chez les OM's
le site radioamateurs france reprend d'ailleurs souvent des articles de framboise314
http://www.radioamateurs-france.fr/?s=raspberry

7.

finopat 16 mai 2017 à 18 h 18 min

merci pour ces sagas, je les avais mises de côté par manque de temps mais j'y retourne.
J'en ai marre de mes pb de sonnettes sans fil qui ne fonctionnent plus au bout de qq mois (pb piles, pb réception …) du coup voilà une bonne application de cette saga : je vais tirer un câble jusqu'au portail (~20m) et avec le raspberry je pourrai faire ce que je veux (j'ai même vu qu'on pouvait envoyer des sms avec des plateformes comme twilio et ça me branche ce truc de geek).

Du coup si j'ai bien compris le plus fiable semble être le schéma 4c avec une résistance de 4,7k.
Pour le programme je vais partir sur du python (que je ne connais pas encore), pour économiser les ressources CPU et pouvoir faire autre chose de mon PI, est ce qu'il est judicieux de partir sur l'utilisation de la fonction wait_for_edge() de GPIO ?

j'ai trouvé des explications ici :
https://sourceforge.net/p/raspberry-gpio-python/wiki/Inputs/?SetFreedomCookie