PERI - Rapport des TPs

== Participants : 2 ==

HOSPITAL Alexandra - 3401862 - alexandra.hospital@etu.upmc.fr
TOUMLILT Ilyas - 3261538 - toumlilt.ilyas@gmail.com

== Codes sources ==

Les sources des nos TPs sont disponibles dans le dossier Codes_Sources (soumission)
et également en open source sur github à l'adresse :
-> https://github.com/ilyasToumlilt/M1_PERI
-> Les codes y sont clairement documentés et accompagnés de main de
test et de Makefile.

== Etat d'avancement ==

TP 01 - OK
TP 02 - OK
TP 03 - OK
TP 04 - OK
TP 05 - OK
TP 06 - OK
TP 07 - OK
TP 08 - Projet - En Cours

TP01 - Communication par tubes

-> Aucun rapport n'a été demandé pour ce TP.
-> Exercice assez facile dont la seule difficulté est syntaxique,
pour ceux qui ne sont pas familier avec python.
-> Code C : **TME01/src/fake.c**
-> Code Py : **TME01/src/server.py**
-> un **make** execute le tout (serveur en tache de fond)

TP02 - Outils de développement et GPIO :

1. Prise en mains des outils de développement: Hello World !

-> Dans ce premier exercice il est demandé de créer un simple programme
Hello World (see **src/hello_world.c**) et de pouvoir le cross compiler
et l'exécuter sur la raspberry.
-> La directive **make hello_world** du makefile permet de faire la compilation
en utilisant la toolchain (à rajouter dans le $PATH).
-> La directive **make upload** permet de charger le fichier sur la raspberry
(en ssh).

2. Contrôle de GPIO en sortie

-> Dans cet exercice on nous propose de manipuler une pin du GPIO en mode "sortie"
pour contrôler le clignotement d'une LED à une fréquence donnée.
-> Pour celà j'utilise la libgpio donnée avec le TP, le fichier **src/lab1.c**
fait clignoter la LED sur GPIO_04 10 fois à une fréquende de 1Hz, sauf
si on lui passe une période de clignotement en argument.
-> La directive **make lab1** du Makefile permet de cross compiler ce programme
(et make upload pour le charger sur la RPi)
-> On remarque d'ailleurs que plus on augmente la fréquence (réduit la période
en argument), plus vite la LED clignote. Logique.

3. Contrôle de plusieurs GPIO en mode "sortie"

-> Ici on reprend l'exo précédent, mais en utilisant les LEDs sur les broches
GPIO 4, 7, 22 et 27.
-> source : **src/lab2.c** ; cc : **make lab2**

4. Lecture de la valeur d'une entrée GPIO

-> Cette fois-ci on veut lire les entrée depuis les boutons poussoirs connectés
aux GPIO pins 18 et 23, on configure donc les deux GPIO en entrée, et on
lance une boucle d'échantillonnage où on lit et affiche les valeurs des
deux entrée (voir **src/lab3.c**).
-> On observe que l'entrée est effectivement égale à 0 quand on appuie sur le
bouton et 1 sinon (Pull Up).
-> On a décidé d'écrire un programme qui échantillonne avec une fréquence égale à
100Hz, je fais donc un sleep de 100 millisecondes entre chaque deux lectures.
-> La directive **make lab3** permet de compiler le fichier.

5. Manipulation de registres bas-niveau

-> Dans cet exercice il nous est demandé de réécrire les fonctions de la libgpio,
pour celà j'ai créé les fichiers :
- **src/gpio_setup.c** : Ce fichier gère le mapping sur la zone mémoire des
registres GPIO, pour initialiser l'adresse de base j'ai écris une fonction
setGpioBaseAdrFromPiRevision qui initialise l'adr selon le type de RPi,
ceci préserve la compatibilité de nos programmes (par exemple chez moi
je travaille sur une RPi v2 vs RPi v1 en TP), tous nos programmes

seront donc compatibles avec toutes les versions RPi.
La fonction gpio_setup effectue donc le mapping mémoire à partir de cette adresse, et la fonction gpio_teardown relache le mapping.
- **src/gpio_config.c** : contient une seule fonction gpio_config qui configure (ou initialise) un GPIO dont le numéro est passé en argument à la valeur passée, c'est grace à cette fonction qu'on configure un GPIO en entrée ou en sortie.
- **src/gpio_value.c** : implémentation de deux fonctions (qu'on pourrait qualifier d'entrée sortie) la première est gpio_value qui permet de récupérer la valeur lue sur une broche, et la deuxième qui permet d'envoyer une valeur sur un pin. Ces deux fonctions utilisent les registres SET pour l'écriture des 1, CLR pour l'écriture des 0, et LEV pour la lecture.

-> **make lib/libgpio.a**

6. Amusons-nous !

-> Le fichier **src/lib4.c** contient un joli programme qui utilise deux LEDs et deux BTNs pour implémenter un peitt automate déterministe, où chaque bouton s'occupe d'allumer/éteindre une LED. Le programme s'arrête quand on appuie sur les deux boutons à la fois.

-> Ce programme utilise les fonctions écrites dans l'exo précédent et est donc compatible avec les deux versions de RPi.

-> **make lab4** + **make upload**.

TP03 - Premier Pilote

1. Création et test d'un module noyau

-> Ce premier exercice consistait simplement à créer un module minimal, qui affiche dans le tampon des messages du noyau (accessible avec dmesg), "Hello World !" à l'insertion et "Goodbye World !" à la suppression.

-> Vous trouverez les sources de ce module dans **TME03/src/module.c**.

-> **NB:** Comme j'ai pas réussi à faire de la compilation séparée, une simple **make** compile tous les programmes de ce TP, et un **make clean** fait le grand ménage.

-> **NB2:** J'ai passé beaucoup de temps chez moi (environ une large matinée) à essayer de faire marcher le module sur ma RPi, car je n'arrivais pas à avoir les headers du Kernel compatibles entre celui installé sur la RPi et celui cross compilé sur ma machine. J'ai donc beaucoup cherché sur internet pour y arriver, vous trouverez des liens utiles vers les tutos les plus intéressants que j'ai trouvé dans **TME03/README.md**

2. ajout des paramètres au module

-> Notre module minimal doit maintenant prendre des paramètres à l'insertion,

on nous a proposé au TP de prendre les numéro des pins GPIO des LED/BTN, mais moi j'ai décidé de prendre le nombre de LEDs et le nombre de BTNs, ce qui m'est très utile car je n'ai pas le même nombre de LEDs et BTNs chez qu'en salle de TP.

-> sources du programme : **TME03/src/module_param.c**, le module prend deux arguments, nbLed et nbBtn, et affiche simplement les valeurs reçues (quand je dis affiche c'est plutôt des écritures dans le tampon des messages du noyau).

3. Création d'un driver qui ne fait rien mais dans le noyau

-> Le but de cet exercice est de charger un driver dans le module, et de voir le comportement des fonctions d'entrée/sortie (avec des simples print pour chacune), le code source est dans **TME03/src/ledbtn_lite.c**.

-> Petite trace d'execution :

```
$ sudo chmod a+rw /dev/ledbtn
$ echo "rien" > /dev/ledbtn
$ dd bs=1 count=1 < /dev/ledbtn
$ dmesg
.....
[ 2755.666469] Hello World !
[ 2830.446135] open()
[ 2830.446259] write()
[ 2830.446292] close()
[ 2849.568369] open()
[ 2849.574987] read()
[ 2849.575176] close()
```

4. Accès aux GPIOs depuis les fonctions du pilote

-> Sources : **TME03/src/ledbtn.c**

-> J'ai utilisé la structure représentant l'organisation des registres proposée dans l'énoncé du TP, et plus besoin d'implémenter une fonction pour initialiser l'adresse de base en fonction de la révison de la RPi, car l'adresse physique de base des GPIO est mappée dans l'espace virtuel du noyau à l'adresse _io_address.

-> J'ai donc repris mes fonctions d'opérations sur les GPIOs écrites au TME02.

-> Le module prend en paramètre le nombre de LEDs et le nombre de BTNs. Et il doit obligatoirement y avoir une LED ou un bouton pour qu'il fonctionne. (je lève une KERN_ERR sinon.)

-> Les LEDs sont configurés en sorties et les BTNs en entrée.

-> Les communications sont gérées grace aux fonction d'I/O, le read permet de lire les valeurs retournées par les boutons (un copy_to_user permet de transférer les données dans l'espace utilisateur), et le write de contrôler les LEDs, il prend donc une chaîne "binaire" de la forme "0101", à savoir qu'un 1 allume la led et un 0 l'éteint. (je vérifie bien que la taille de la chaîne est au moins égale au nombre de leds et qu'il n'y a que des 0/1, sinon je

lève une KERN_ERR).

TP04 - Pilotage d'un écran LCD en mode utilisateur

-> Un seul fichier source est fourni pour ce TP, il s'agit de **TME04/src/lcdRpi.c**
-> Pour réaliser ce TP j'ai utilisé mon propre matériel, à savoir une RPi v2
(le code fourni est compatible avec la révision 1 de RPi, comme tous les
programmes que je fournis), et un LCD HD44780 de version 16x2, contrairement
à ceux du TP qui contiennent 4 lignes. (et comme j'ai oublié ce détail,
mon programme est fonctionnel en salle de TP, mais ne prend en compte
que 2 lignes, et comme toutes les communications sont en sortie, je ne
peux pas lire la version du LCD pour la compatibilité ...).
-> Il y avait beaucoup de lecture de doc.

GPIO Pins :
-> J'ai respecté le branchement du TP, à savoir :
RS: GPIO 7
E : GPIO 8
D4: GPIO 22
D5: GPIO 23
D6: GPIO 24
D7: GPIO 25
Je n'ai pas eu de soucis particulier avec le raccordement, à part le fait
que mon LCD était un trop brillant et du coup j'ai rajouté un potentiomètre
pour réduire le contraste.

GPIO Operations :
-> J'ai simplement repris les fonctions implémentée aux TPs 2 et 3, pour le
controle des GPIOs (tous en sortie), à savoir la structure pour le
map des registres, les fonctions de configuration, de choix d'adresses
de base par rapport à la révision du RPi, l'initialisation des registres
en sortie, ainsi que la fonction d'écriture.

LCD's basic instructions :
-> J'ai choisi de représenter les instructions (et les données aussi d'ailleurs)
du LCD avec des chaînes de 8 caractères binaires.
-> Cette troisième partie du code contient des définitions de toutes les
instructions de base du LCD.
-> Ces instructions sont tirées de la doc.
-> Je n'utilise pas toutes ces définitions dans mon code, mais il est bon
de les avoir.

LCD's operations :

-> La partie la plus importante, elle implémente les opérations de contrôle du LCD, à savoir :

- binaryOR : un simple OR binaire entre deux chaînes de taille 8.
- decimal2binary : convertit un entier en chaîne binaire de taille 8.
- lcd_strobe : génère le signal E. En envoyant 0 puis 1 puis 0 sur la broche E avec 1 microseconde de sleep après chaque envoi.
- lcd_write4bits : écriture en mode 4bits, cette fonction prend en argument la commande/donnée à écrire et le mode (0 pour les commandes et 1 pour les données), ce mode sera envoyé sur RS au départ. On écrit ensuite les 4 premiers bits, on génére un signal E puis les 4 bits suivants, puis un autre E.
  Cette fonction est appelée par les deux fonctions suivantes.
- lcd_command : envoi d'une commande au LCD.
- lcd_data : envoi d'une donnée au LCD.
- lcd_init : éxecute les commandes d'initialisation.
- lcd_clear : clear display important avant écriture.
- lcd_message : écriture d'une chaîne sur l'ecran.

Main

-> Le programme principal prend en argument la chaîne à écrire sur l'ecran.

Directives du Makefile

-> make / make cleanall, classique.

-> make upload, pour charger le programme sur RPi.

Conclusion

-> Beaucoup de recherches personnelles pour réaliser ce TP, mais également beaucoup de plaisir car ma curiosité a été attisée sur pas mal d'aspects.

TP05 - LCD en mode kernel

-> Un seul fichier source : **TME05/src/lcdRpi.c**

-> Pour ce TP on reprend le TP précédent, mais en mode Kernel, un peu comme un passage du TP02 au TP03.

-> Le module prend en argument le nombre de lignes du LCD (pratique, vu que chez moi c'est un 2lignes, contre un 4lignes en salles de TP, et pour le coup je gère les deux formats cette fois).

-> Je fais toutes les initialisations au moment du chargement du module,
à savoir le paramètrage en sortie des GPIO, et l'execution des commandes
d'initialisation du LCD.
-> Les ecritures sur l'ecran se font grace aux focntions d'I/O, en particulier
le write, qui prend la chaîne écrite et l'envoi vers l'ecran LCD grace aux
fonctions implémentées au TP04.

Conclusion :
-> La réalisation de ce TP a été plus rapide que les précédents, car une fois
les fonctions de controle du LCD réalisées, et l'insertion des modules
maîtrisée, ça va vite.

TP06 - Arduino Base

-> Une première prise en main de l'arduino dans ce TP.

1. Faire clignoter une Led
-> Code source : **TME06/src/blink.ino**
-> Ce simple programme innitialise le pin 13 (celui de la LED)
en OUTPUT (appel à pinMode), et puis fait clignoter cette LED
(appel à digitalWrite avec HIGH/LOW), avec des délais de 500ms
entre les appels.

2. Photorésistance
-> Code source : **TME06/src/brightness.ino**
-> Ce programme allume la LED sur le pin 13 si la photorésistance capte
une faible lumière.
-> On définit le débit de données à 9600 bits par secondes (appel à Serial.begin)
photorésistance (A0) en INPUT, LED en OUTPUT, et à chaque fois on lit
la valeur d'A0 (analogRead) si c'est inférieur à 100 on allume la LED.

3. Simulation d'une serrure codée
-> Code source : **TME06/src/brightCodeLock.ino
-> On nous demande de réaliser une petite machine à états, où la photorésistance
se comporte comme un bouton sur lequel l'utilisateur peut composer son code
secret.
-> J'ai augmenté les temps de composition car il était difficile d'avoir une
précision correcte avec ce qui était donné. Le code devient donc :
masquer 4s, démasquer 4s, masquer 6s. Avec une marge d'imprécision
de 500ms

TP07 - Communication sans fil

1. Schéma de branchement

-> La photorésistance est sur le pin 14 (A0)
-> nRF24 :
- CE : 9
- CSN : 10
- MISO : 12
- MOSI : 11
- SCK : 13

2. Librairies et tests

-> On a commencé par une première exploration des exemples dans File/examples.

3. Ecran + Photorésistance

-> Code source : **TME07/src/brightOnScreen.ino**
-> Affiche la valeur lue sur la photorésistance (entrée A0), sur l'ecran, après une mise à l'échelle entre 0 et 100.
-> Pour ne pas avoir de petites valeurs (la salle n'est pas si eclairée que ça) j'ai mis la valeur max à 600.
-> Ces valeurs (MAX + échelle) sont modifiables à partir des defines au début du fichier.

4. Utilisation du module nRF24

-> On veut pouvoir afficher la valeur de la photorésistance cette fois sur l'ecran d'un autre Arduino en passant par la transmission nRF24.
-> Les deux entités se comporteront comme un producteur/consommateur, on utilise ici les méthodes de la classe radio, notamment le pipe nommé et les read/write.
-> Code Emetteur : **TME07/src/transmitter.ino**
-> Code Récepteur : **TME07/src/receiver.ino**