École affiliée IMT

Université de Lyon

Université Jean Monnet Saint-Étienne

Cycle Initial en Technologies de l'Information de Saint-Étienne

SYNTHÈSE TP AT33

Eva Maturana - Lucas Lescure

Table des Contenus

1. Programmes de base

1.1. Manipulation des arguments en ligne de commande

1.1.a) Exercise 1

On veut créer un programme qui accepte 3 arguments en ligne de commande et les affiche. Dans le cas où il n'y a pas 3 arguments on désire retourner le code d'erreur `EXIT_FAILURE` ainsi qu'un message d'erreur. Pour pouvoir

```
#include <stdio.h>

int main (int argc, char* argv[]) {
    if(argc != 4){
        printf("\nerreur %s, il faut 3 arguments, vous en avez passe %d", arcgv[0], argc - 1);
        return EXIT_FAILURE;
    } else {
        for(int i = 0; i < 4; i++){
            printf("\nargv[%d] = %s", i, argv[i]);
        }
        return EXIT_SUCCESS;
    }
}
```

Figure 1.1. Code fonctionnel de l'exercise

manipuler des arguments en ligne commande, on définit la fonction `main()` en faisant passer les variables `argc`, qui compte le nombre d'arguments passés, et `argv[]` qui sauvegarde sous forme de `const char*` les arguments passés dans un tableau.

Pour afficher un message à l'utilisateur dans la console il faut utiliser la fonction `printf()` car la fonction `cout` n'existe que en `C++`.

Après avoir codé le programme on utilise le compilateur `gcc` avec `-o <fichier>.o` en argument pour créer un objet executable par l'utilisateur. Il ne suffit donc plus que d'exécuter ce dernier avec la command `./ex1.o`

En utilisant ceci on retrouve les exécutions suivantes:

```
pi@raspberry:~$  ./ex1.o il faut 3 arguments
erreur ./ex1.o, il faut 3 arguments, vous en avez passé 4

pi@raspberry:~$  ./ex1.o faut 3 arguments
argv[0] = ./ex1.o
argv[1] = faut
argv[2] = 3
argv[3] = arguments
```

1.1.b) Exercise 2

On veut créer un programme qui accepte 2 arguments(entiers) en ligne de commande, retourne `EXIT_FAILURE` dans le cas échéant, et affiche les deux entier ainsi que leur somme, leur produit et leur différence.

On construit le programme ci-dessous en utilisant les notions vues précédemment et cette fois-ci en faisant intervenir un nouvelle fonction `atoi()` dans la librairie `stdlib.h` permettant de retourner un `int` d'un `const char*` passé en paramètre.

En stockant la valeurs des arguments en ligne de commande dans les entiers `A` et `B` on peut alors traiter les deux entiers dans notre programme et y effectuer des opérations.

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

int main (int argc, char* argv[]) {
    if(argc != 3) {
        printf("erreur %s, il faut 2 arguments", argv[0]);
        return EXIT_FAILURE;
    } else {
        int A = atoi(argv[1]); int B = atoi(argv[2]);

        printf("\nA = %d; B = %d", A, B);
        printf("\nA + B = %d", A + B);
        printf("\nA * B = %d", A * B);
        printf("\nA - B = %d", A - B);

        return EXIT_SUCCESS;
    }
}
```

Figure 1.2. Code fonctionnel de l'exercise

Exemple d'exécution:

```
pi@raspberry:~$  ./ex2.o 5 1245
A = 5; B = 1245
A + B = 1250
A * B = 6225
A - B = -1240
```

1.2. Pilotage de GPIO en sortie

1.2.a) Exercise 1

On veut construire un programme qui permet de piloter le niveau logique du `GPIO19`, tel que l'argument passé en ligne de commande corresponde à l'état logique de la LED.

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

int main (int argc, char* argv[]) {
    if(argc != 2) {
        printf("erreur %s, il faut 1 argument", argv[0]);
        return EXIT_FAILURE;
    } else {
        FILE* fichier;
        char path_d[100]; char path_v[100];
        int val = atoi(argv[1]);

        sprintf(path_d, "/sys/class/gpio/gpio19/direction");
        sprintf(path_v, "/sys/class/gpio/gpio19/value");

        fichier = fopen(path_d, "w");
            fprintf(fichier, "out");
        fclose(fichier);

        fichier = fopen(path_v, "w");
            fprint(fichier, "%d", val);
        fclose(fichier);

        return EXIT_SUCCESS;
    }
}
```

Figure 1.3. Code fonctionnel de l'exercise

Pour ce faire il faut pouvoir naviguer dans les repertoires du système embarqué. On va donc faire recours aux fonctions `fopen()`, `fprintf()` et `fclose()` permettant respectivement, d'affecter à un pointeur de fichier (`FILE* fichier`) l'emplacement et le mode d'ouverture du fichier, écraser et écrire sur le fichier, et fermer le fichier suite aux opérations effectuées avec.

Les deux fichiers qui nous intéresse pour la manipulation des GPIOs sont `direction`, qui permet d'établir le sens (entrée/sortie - "in"/"out") de la GPIO à utiliser, et `value` sur lequel est stocké l'état de la GPIO.

On se permet aussi d'utiliser la fonction `sprintf()` pour affecter le chemin des fichiers `direction` et `value` de la GPIO19 au chaine de charactères, `path_d` et `path_v`.

Exemple d'exécution:
`pi@raspberry:~$ ./ex3.o 1`
La LED de la GPIO19 s'éteind.

`pi@raspberry:~$ ./ex3.o 0`
La LED de la GPIO19 s'allume.

1.2.b) Exercise 2

On veut comme dans l'exercise précédent pouvoir piloter le niveau logique d'une GPIO, cette fois-ci on utilisera un argument en plus pour désigner le numéro de la GPIO à utiliser.

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

int main (int argc, char* argv[]) {
    if(argc != 3){
        printf("erreur %s, il faut 2 arguments", arcgv[0]);
        return EXIT_FAILURE;
    } else {
        FILE* fichier;
        char path_d[100]; char path_v[100];
        int num = atoi(argv[1]);
        int val = atoi(argv[2]);

        sprintf(path_d, "/sys/class/gpio/gpio%d/direction", num);
        sprintf(path_v, "/sys/class/gpio/gpio%d/value", num);

        fichier = fopen(path_d, "w");
             fprintf(fichier, "out");
        fclose(fichier);

        fichier = fopen(path_v, "w");
             fprintf(fichier, "%d", val);
        fclose(fichier);

        return EXIT_SUCCESS;
    }
}
```

Figure 1.4. Code fonctionnel de l'exercise

Pour réaliser ceci on stocke le numéro de la GPIO dans la variable `num`, et son état dans la variable `val`.

Pour se diriger vers le bon emplacement de la GPIO on utilise dans la fonction `sprintf()`, l'argument `%d` qui est un spécificateur de format permettant de remplacer celui-ci par la valeur de la variable `num` de type entier dans la chaine de charactères à affecter aux chemins `path_d` et `path_v`. On peut faire de même avec la fonction `fprintf()`.

Exemple d'exécution:
`pi@raspberry:~$ ./ex4.o 26 1`
La LED de la GPIO26 s'éteind.
`pi@raspberry:~$ ./ex4.o 23 0`
La LED de la GPIO23 s'allume.

1.2.c) Exercise 3

On veut établir un programme qui permet de faire clignoter la LED d'une GPIO dont le numéro sera passé en argument en ligne de commande jusqu'à l'interruption de l'exécution par la commande CTRL+C.

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main (int argc, char* argv[]) {
    if(argc != 2){
        printf("erreur %s, il faut 1 argument", arcgv[0]);
        return EXIT_FAILURE;
    } else {
        FILE* fichier;
        char path_d[100]; char path_v[100];
        int num = atoi(argv[1]);

        sprintf(path_d, "/sys/class/gpio/gpio%d/direction", num);
        sprintf(path_v, "/sys/class/gpio/gpio%d/value", num);

        fichier = fopen(path_d, "w");
             fprintf(fichier, "out");
        fclose(fichier);

        while(1){
            fichier = fopen(path_v, "w");
                fprintf(fichier, "1");
            fclose(fichier);
            sleep(1);

            fichier = fopen(path_v, "w");
                fprintf(fichier, "0");
            fclose(fichier);
            sleep(1);
        }

        return EXIT_SUCCESS;
    }
}
```

Figure 1.5. Code fonctionnel de l'exercise

On va donc utiliser une boucle `while(1){...}` de façon exécuter le programme en permanence, et on fera recours à la fonction `sleep` de la librairie `unistd.h` ainsi que les notions précédemment vues.

Dans notre boucle on va donc ouvrir le fichier `value` une première fois pour mettre l'état de la `GPIO` à 1 et une deuxième fois pour la remettre à 0, avec une pause de 1 seconde entre chaque écriture.

Exemple d'exécution:
`pi@raspberry:~$ ./ex5.o 26`
La LED de la `GPIO26` clignote à une fréquence de $0.5\ H_z$

1.2.d) Exercise 4

En gardant le même fonctionnement que l'exercise précédent, on veut pouvoir modifier la fréquence de clignotement à partir d'un second argument passé en ligne de commande.

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

int main (int argc, char* argv[]) {
    if(argc != 3){
        printf("erreur %s, il faut 2 arguments", arcgv[0]);
        return EXIT_FAILURE;
    } else {
        FILE* fichier;
        char path_d[100]; char path_v[100];
        int num = atoi(argv[1]);

        float us_period = 1000000/(2*atoi(argv[2]));

        sprintf(path_d, "/sys/class/gpio/gpio%d/direction", num);
        sprintf(path_v, "/sys/class/gpio/gpio%d/value", num);

        fichier = fopen(path_d, "w");
             fprintf(fichier, "out");
        fclose(fichier);

        while(1){
            fichier = fopen(path_v, "w");
                fprintf(fichier, "1");
            fclose(fichier);
            usleep(us_period)

            fichier = fopen(path_v, "w");
                fprintf(fichier, "0");
            fclose(fichier);
            usleep(us_period)
        }

        return EXIT_SUCCESS;
    }
}
```

Figure 1.6. Code fonctionnel de l'exercise

En reprenant le code précédent on converti et stocke la période en μs dans le flottant `us_period`. Cette période est alors utilisé en conjonction avec la fonction `usleep()` pour établir la fréquence de clignotement demandée.

Exemple d'exécution:
`pi@raspberry:~$ ./ex6.o 23 5`
La LED de la `GPIO23` clignote à une fréquence de $5\ H_z$

1.3. Pilotage GPIO en entrée

1.3.a) Exercise 1

On cherche à écrire un programme permettant de lire l'état d'un bouton pressoir situé sur la `GPIO27` et d'afficher l'état sur la console.

Pour réaliser ceci on initialise la variable `num` à 27, et on configure le sens de la `GPIO` en entrée("in").

Dans un boucle on va donc lire l'état de la `GPIO` en ouvrant le fichier sous le mode lecture (`r+`), et en utilisant la fonction `fscanf()` qui permet d'extraire les données dans le fichier. Notamment avec le specificateur `%s` on relève la chaine de charactères qui est inscrite dans le fichier et on la stocke dans la chaine `sval[1]` placée en argument dans la fonction `fscanf()`. On converti ensuite cette chaine de charactères sous forme d'entier avec `state`.

On teste alors pour voir si la valeur relevé de l'état de la `GPIO` varie. Si c'est le cas alors on affiche cette valeur avec la fonction `printf()`, puis on sauvegarde cet état dans la variable `save` qui sera nouvellement comparé avec `state` pour savoir si l'état change. Sinon on continu de lire en boucle la valeur de l'état.

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

int main (int argc, char* argv[]) {
    if(argc != 1){
        printf("erreur %s, il faut pas d'argument", arcgv[0]);
        return EXIT_FAILURE;
    } else {
        FILE* fichier;
        char path_d[100]; char path_v[100];

        char sval[1];
        int num = 27; int save = 1;

        sprintf(path_d, "/sys/class/gpio/gpio%d/direction", num);
        sprintf(path_v, "/sys/class/gpio/gpio%d/value", num);

        fichier = fopen(path_d, "w");
             fprintf(fichier, "in");
        fclose(fichier);

        while(1){
            fichier = fopen(path_v, "r+");
                fscanf(fichier, "%s", sval);
            fclose(fichier);
            state = atoi(sval);

            if(save != state){
                printf("\nSTATE : %d", state);
                save = state;
            }

        }
        return EXIT_SUCCESS;
    }
}
```

Figure 1.7. Code fonctionnel de l'exercise

Exemple d'exécution:
`pi@raspberry:~$ ./ex7.o`
`STATE : 1`
(Appui du bouton pressoir pendant 2 secondes)
`STATE : 0`
(Fin des 2 secondes)
`STATE : 1`

1.3.b) Exercise 2

On veut faire de même que l'exercise précédent mais en passant en argument le numéro de la `GPIO` à surveiller.

En considérant que le sens des `GPIO` à lire sont préétablies, on initialise `num` au numéro de la `GPIO` passé en argument. On modifie également la fonction `printf()` pour satisfaire la demande.

Le reste du code n'as pas besoin d'être modifié.

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

int main (int argc, char* argv[]) {
    if(argc != 2){
        printf("erreur %s, il faut 1 argument", arcgv[0]);
        return EXIT_FAILURE;
    } else {
        FILE* fichier;
        char path_d[100]; char path_v[100];
        int num = atoi(argv[1]);

        char sval[1];
        int save = 1;

        sprintf(path_v, "/sys/class/gpio/gpio%d/value", num);

        while(1){
            fichier = fopen(path_v, "r+");
                fscanf(fichier, "%s", sval);
            fclose(fichier);
            state = atoi(sval);

            if(save != state){
                printf("\nSTATE GPIO&d : %d", num, state);
                save = state;
            }

        }
        return EXIT_SUCCESS;
    }
}
```

Figure 1.8. Code fonctionnel de l'exercise

Exemple d'exécution:

```
pi@raspberry:~$  ./ex8.o 19
STATE GPIO19 :  0
```

(On allume la LED de la GPIO19)

```
STATE GPIO19 :  1
```

(On éteind la LED)

```
STATE GPIO19 :  0
```

2. Mise en fonctions

2.1. Direction d'un GPIO

On cherche à réaliser une fonction qui aura le prototype : `void Ecrit_DIR_PORT(int num_port, int dir)`
Pour `dir` $= 1$ il faut que la `GPIO` soit mise en entrée, et pour 0, en sortie.

```c
#include <stdio.h>

void Ecrit_DIR_PORT(int num_port, int  dir){
    FILE* fichier; char path[100];

    sprintf(path, "/sys/class/gpio/gpio%d/direction", num_port);

    fichier = fopen(path, "w");
        if(dir == 1){
            fprintf(fichier, "in");
        } else {
            fprintf(fichier, "out");
        }
    fclose(fichier);
}
```

Figure 2.1. Code direction d'une `GPIO`

En utilisant les notions précédemment vues on écrit le code ci-dessous, avec lequel on ajoute un test pour savoir en fonction de `dir` s'il faut écrire `"in"` ou `"out"`.

2.2. Écriture d'un GPIO

On cherche à écrire une fonction qui aura le prototype : `void Ecrit_ETAT_PORT(int num_port, int value)`

```c
#include <stdio.h>

void Ecrit_ETAT_PORT(int num_port, int value){
    FILE* fichier; char path[100];

    sprintf(path, "/sys/class/gpio/gpio%d/value", num_port);

    fichier = fopen(path, "w");
        fprintf(f,"%d", value);
    fclose(fichier);
}
```

Figure 2.2. Code écriture d'une `GPIO`

2.3. Lecture d'un GPIO

On veut avoir une fonction qui permet de retourner l'état d'une `GPIO` avec le prototype suivant:
`void Lire_ETAT_PORT(int num_port)`

En utilisant les notions vues dans **1. Programmes de base**, on écrit le code ci-dessus.

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

int Lire_ETAT_PORT(int num_port){
    FILE* fichier; char path[100]; char state[10]; int output;

    sprintf(path, "sys/class/gpio/gpio%d/value", num_port);

    fichier = fopen(path, "r+");
        fscanf(fichier, "%s", state);
    fclose(fichier);

    return output = atoi(state);
}
```

Figure 2.3. Code lecture d'une GPIO

2.4. Mise en bibliothèque

On veut réunir toutes les fonctions définies précédemment dans une même librairie que l'on appellera gpio.lib.c. On écrit alors:

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

void Ecrit_ETAT_PORT(int num_port, int value){
    FILE* fichier; char path[100];

    sprintf(path, "/sys/class/gpio/gpio%d/value", num_port);

    fichier = fopen(path, "w");
        fprintf(f,"%d", value);
    fclose(fichier);
}

void Ecrit_DIR_PORT(int num_port, int  dir){
    FILE* fichier; char path[100];

    sprintf(path, "/sys/class/gpio/gpio%d/direction", num_port);

    fichier = fopen(path, "w");
        if(dir == 1){
            fprintf(fichier, "in");
        } else {
            fprintf(fichier, "out");
        }
    fclose(fichier);
}

int Lire_ETAT_PORT(int num_port){
    FILE* fichier; char path[100]; char state[10]; int output;

    sprintf(path, "sys/class/gpio/gpio%d/value", num_port);

    fichier = fopen(path, "r+");
        fscanf(fichier, "%s", state);
    fclose(fichier);

    return output = atoi(state);
}
```

Figure 2.4. Bibliothèque gpio.lib.c

3. Programmes

3.1. Chenillard à 4 LED

On peut alors utiliser cette bibliothèque en rajoutant `#include gpio.lib.c` au début de notre programme. Pour faire un chenillard de fréquence modifiable on écrit le code suivant:

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

#include "gpio.lib.c"

int main(int argc, char* argv[]){
    if(argc != 2){
        printf("erreur %s, il faut 1 argument", arcgv[0]);
        return EXIT_FAILURE;
    } else {
        FILE* fichier;
        int ns_period = 1000000/atoi(argv[1]);

        int led[4] = {24,23,26,19};

        for(int i = 0; i < 4; i++){
            usleep(ns_period);
            Ecrit_ETAT_PORT(led[(i + 2) % 4], 1);
            Ecrit_ETAT_PORT(led[(i + 1) % 4], 1);
            Ecrit_ETAT_PORT(led[i], 0);
        }
    }
    return EXIT_SUCCESS;
}
```

Figure 3.1. Code chenillard avec `gpio.lib.c`