
# MI11
# TP2 Linux Embarqué

Jeanneau Louis, Schulster Alex

Printemps 2022

# Table des matières

# 1 Travail préalable

Nous reprenons pour ce TP ce qui a été produit au TP précédent (machine virtuelle, noyaux, os, ...). Le `Joy-Pi-Note` n'étant pas le même, nous mettons à jour le chemin des fichiers présents dans `/tftpboot/`.

# 2 Hello World

Notre fichier objet `hello` compilé par `gcc` depuis du code en C est un fichier objet 64-bit pour architecture x86-64. Il ne peut donc pas s'exécuter sur la cible dont l'architecture est ARM.

```
mi11p22@mi11p22-VirtualBox:~/Documents/TP2$ file hello
hello: ELF 64-bit LSB shared object, x86-64
```

Pour pouvoir utiliser la chaîne de compilation croisée avec *$CC*, il faut d'abord l'installer. Nous l'installons avec :

```
. /opt/poky/3.1.16/cortexa7thf-neon-vfpv4/environment-setup-cortexa7t2hf-neon-vfpv4-poky-linuxgnueabi
```

En compilant avec la chaine croisée de *$CC*, nous obtenons un fichier 32-bit pour architecture ARM. Ce fichier s'éxecute parfaitement sur la cible.

```
mi11p22@mi11p22-VirtualBox:~/Documents/TP2$ file hello_arm
hello_arm: ELF 32-bit LSB shared object, ARM
```

```c
#include <stdio.h>

int main() {
  printf("Hello, World!\n");
  return 0;
}
```

Nous obtenons donc ce résultat à l'éxecution.

```
root@joypinote:/usr\# ./hello\_arm
Hello, World!
```

# 3 Clignotement des LEDs

Pour accéder aux LEDs, il faut écrire la valeur souhaitée dans le fichier représentant le périphérique. C'est possible dans le terminal avec `echo` :

```
root@joypinote:/ echo "valeur" \> /sys/class/leds/led\_num/brightness
```

Ainsi, pour nos 2 LEDs, nous avons 4 commandes :

```
echo 1 > /sys/class/leds/led_5/brightness
echo 1 > /sys/class/leds/led_6/brightness
echo 0 > /sys/class/leds/led_5/brightness
echo 0 > /sys/class/leds/led_6/brightness
```

Pour un programme en C, il nous faut utiliser `open` et `write`. Nous avons ainsi un programme qui fait clignoter les LEDs :

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

int main()
{
   // Chargement fichiers LEDs
   int led_5 = open("/sys/class/leds/led_5/brightness", O_WRONLY);
   // Verification de la bonne ouverture des fichiers
   if(led_5 == -1)
   {
      printf("Error!\n");
      exit(1);
   }

   int led_6 = open("/sys/class/leds/led_6/brightness", O_WRONLY);
   if(led_6 == -1)
    {
      printf("Error!\n");
      close(led_5);
      exit(1);
    }

  // Boucle infinie
   for (;;) {
    // Allumer led 5, eteindre la led_6
    write(led_5, "1", 1);
    write(led_6, "0", 1);
    sleep(1); // Attente de 1 seconde

    // Allumer la led_6, eteindre la led_5
    write(led_6, "1", 1);
    write(led_5, "0", 1);
    sleep(1); // Attente de 1 seconde
   }

  // Fermeture des fichiers LEDs
  close(led_5);
  close(led_6);
   return 0;
}
```

# 4 Boutons poussoirs

Pour les boutons poussoirs, nous utilisons `evtest` pour capter les différentes valeurs de messages.

```
root@joypinote:/usr# evtest
No device specified, trying to scan all of /dev/input/event*
Available devices:
/dev/input/event0: joypinote_keypad
/dev/input/event1: joypinote_joystick
Select the device event number [0-1]: 0
Input driver version is 1.0.1
Input device ID: bus 0x19 vendor 0x1 product 0x1 version 0x100
Input device name: "joypinote_keypad"
```

Il y a les événements de type 1 `EV_KEY` qui indique une interaction avec les boutons. Chaque touche a son code, par exemple le code 2 correspond à la touche numéro "1". Enfin, une valeur 1 indique que le bouton est enfoncé et une valeur 0 indique que le bouton est relâché.

Pour trouver le fichier *header* contenant la structure *input_event*, nous utilisons `grep`.

```
mi11p22@mi11p22-VirtualBox:/opt/poky/3.1.16/cortexa7thf-neon-vfpv4$ grep -r "input_event" *
sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi/usr/include/linux/input.h:struct input_event
sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi/usr/include/linux/input.h:#define
    input_event_sec time.tv_sec
sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi/usr/include/linux/input.h:#define
    input_event_usec time.tv_usec
sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi/usr/include/linux/input.h:#define
    input_event_sec __sec
sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi/usr/include/linux/input.h:#define
    input_event_usec __usec
sysroots/cortexa7t2hf-neon-vfpv4-poky-linux-gnueabi/usr/include/linux/virtio_input.h:struct
    virtio_input_event
```

Ainsi, c'est le fichier `input.h` qui nous fournit la structure nécessaire à la réception des messages des boutons.

Notre programme est le suivant. Il allume la LED 6 (à gauche) lors d'un appui sur la touche "0" et la LED 5 (à droite) lors d'un appui sur la touche "+".

```c
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <linux/input.h>

int main()
{
   // Chargement fichiers LED_5
   int led_5 = open("/sys/class/leds/led_5/brightness", O_WRONLY);
   // Verification de la bonne ouverture du fichier
   if(led_5 == -1)
   {
      printf("Error led 5!\n");
      exit(1);
   }
   // On eteint la led_5
   write(led_5, "0", 1);
```

```
19
20    // Chargement fichiers LED_6
21    int led_6 = open("/sys/class/leds/led_6/brightness", O_WRONLY);
22    // Verification de la bonne ouverture du fichier
23    if(led_6 == -1)
24    {
25       close(led_5);
26       printf("Error led 6!\n");
27       exit(1);
28    }
29    // On eteint la led_6
30    write(led_6, "0", 1);
31
32    // On ouvre le fichier des inputs
33    int fd = open("/dev/input/event0", O_RDONLY);
34    // Verification de l'ouverture
35    if (fd == -1){
36       close(led_5);
37       close(led_6);
38       printf("Error input!\n");
39       exit(1);
40    }
41
42    // Variable pour enregistrer les evenements
43    struct input_event event;
44
45    // Boucle infinie de lecture du fichier input
46    while (read(fd, &event, sizeof(struct input_event))) {
47       // Verification que l'evenement corresponde bien au changement d'eta d'une touche
48       if (event.type == EV_KEY) {
49          printf("Etat du bouton %d et valeur : %d\n", event.code, event.value);
50          // Touche "+", correspond a la led 5
51          if (event.code == KEY_KPMINUS){
52             if (event.value)
53                write(led_5, "1", 1);
54             else
55                write(led_5, "0", 1);
56          }
57          // Touche "0", correspond a la led 6
58          if (event.code == KEY_0){
59             if (event.value)
60                write(led_6, "1", 1);
61             else
62                write(led_6, "0", 1);
63          }
64       }
65    }
66
67
68   // Fermeture des fichiers LEDs
69   close(led_5);
70   close(led_6);
71   close(fd);
72
73    return 0;
74 }
```

# 5 Charge CPU

Pour vérifier l'incidence de la charge CPU, nous faisons varier les paramètres de la fonction `stress`, pour générer un nombre variable de tâche inutiles. Pour 10 tâches générées avec `stress`, les résultats sont les suivants :

```
root@joypinote:/usr# ./caf
Temps d'execution: 10.667997s
Intervalle minimal: 0.001063s
Intervalle maximal: 0.001145s
```

Les temps minimum et maximum sont proches et l'ensemble est plutôt constant.

Pour 100 tâches, les choses ralentissent :

```
root@joypinote:/usr# ./caf
Temps d'execution: 24.870433s
Intervalle minimal: 0.001021s
Intervalle maximal: 0.585806s
```

Nous voyons un grand écart entre le temps minimal et le temps maximal, d'un facteur de l'ordre de 500 ms. Le temps minimal est le même que pour l'essai précédent, mais le temps maximal a radicalement augmenté et est extrêmement long étant donné la vitesse des processeurs actuels. Cela s'explique par les nombreux changements de contextes que doit faire l'ordonnanceur de l'ordinateur avec le nombre de taches de `stress` qui augmente.

Pour améliorer ces résultats, il faudrait disposer de plusieurs cœurs dans le processeur, ou faire en sorte que l'attente du programme soit bloquante et n'appelle pas l'ordonnanceur.

Ci-dessous le code de notre programme de benchmark :

```c
#include <stdio.h>
#include <sys/time.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>

int main() {

    // Recuperation temps de debut du programme
    struct timeval beginning;
    gettimeofday(&beginning, NULL);

    // Timeval intermediaires pour processing dans boucle
    struct timeval loop;
    struct timeval loop_2;
    struct timeval loop_delta;
    struct timeval loop_min;
    loop_min.tv_usec = 999999;
    loop_min.tv_sec = 10;
    struct timeval loop_max;
    loop_max.tv_usec = 0;
    loop_max.tv_sec = 0;

    gettimeofday(&loop, NULL);
    gettimeofday(&loop_2, NULL);

    timersub(&loop_2, &loop, &loop_delta);
```

```
28
29      // 10 000 attentes de 1 ms
30      for (int i=0; i<10000; i++) {
31          // Attente de 1 ms
32          usleep(1000);
33
34          // Mesure du temps entre boucle precedente et actuelle
35          gettimeofday(&loop_2, NULL);
36          timersub(&loop_2, &loop, &loop_delta);
37
38          // Si temps le plus cours, on enregistre
39          if (loop_delta.tv_usec < loop_min.tv_usec) {
40              loop_min = loop_delta;
41          }
42
43          // Si temps le plus long, on enregistre
44          else if (loop_delta.tv_usec > loop_max.tv_usec) {
45              loop_max = loop_delta;
46          }
47
48          loop = loop_2;
49      }
50
51      // Recuperation temps de fin du programme
52      struct timeval ending;
53      gettimeofday(&ending, NULL);
54
55      // Calcul difference entre debut et fin du programme
56      struct timeval delta;
57      timersub(&ending, &beginning, &delta);
58
59      // Calcul interval moyen
60      int usec_moyen = delta.tv_sec * 1000000 / 10000 + delta.tv_usec / 10000;
61
62      // Impression resultats
63      printf("Temps d'execution: %d.%06ds\n", delta.tv_sec, delta.tv_usec);
64      printf("Intervalle minimal: %d.%06ds\n", loop_min.tv_sec, loop_min.tv_usec-1000);
65      printf("Intervalle maximal: %d.%06ds\n", loop_max.tv_sec, loop_max.tv_usec-1000);
66      printf("Intervalle moyen: %dus", usec_moyen-1000);
67  }
```

# 6 Joystick et écran LCD

Nous n'avons pas eu le temps de traiter cette partie lors du TP. Cette section du rapport n'est donc pas vérifiée en pratique.

Comme vu précédemment avec *evtest*, le fichier des joystick est `/dev/input/event1`. Pour ce qui est des caractéristiques du joystick, nous pensons qu'il fournit 2 valeurs différentes :

- une valeur de mouvement selon l'axe X
- une valeur de mouvement selon l'axe Y

Nous pensons que ces valeurs peuvent être positives ou négatives.

Pour ce qui est de l'écran LCD, nous supposons qu'il suffit d'écrire sur le fichier `/dev/lcd` pour que le texte s'affiche sur l'écran LCD. La commande devrait donc être :

```
1 echo texte > /dev/lcd
```

Faute de temps, nous n'avons pas de programme à proposer.

# 7 Conclusion

Au cours de ce TP, nous avons pu manier les différents périphériques du Joy-Pi-Note à l'aide de ligne de commande et de programmes compilés en C.