Compte-rendu TP "Prise en main de Linux embarqué - Partie 2"

Mewen Michel et Sander Ricou - MI11 UTC

Exercice 1 : Hello Word

On a créé notre fichier main.c, puis on le compile:

```
gcc main.c -o main
```

Question 1.1 : *Que constatez vous ? Pourquoi ce fichier ne peut-il pas s'exécuter sur la cible ?*

```
file main
main: ELF 64-bit LSB  executable, x86-64, version 1 (SYSV), dynamically linked
(uses shared libs), for GNU/Linux 2.6.24,
BuildID[sha1]=467d07238021bd56c90ae0d41f6a489ccd874689, not stripped
```

On constate alors que la compilation a été effectuée pour notre ordinateur de travail (x86-64) et non pas pour la cible embarquée sous ARM.

Cross-compilez maintenant votre programme avec la commande suivante :

```
arm-poky-linux-gnueabi-gcc main.c -o main
```

Cependant cette ligne de commande ne fonctionne pas dans un premier temps car nous n'utilisons pas le script fourni avec la chaîne de compilation croisée.

Question 1.2 : *Que faut-il faire avant de pouvoir lancer cette commande ?*
il faut donc sourcer le script et faire un unset sur LDFLAGS pour s'assurer que le comportement du linker n'est pas changé:

```
source /opt/poky/1.7.3/environment-setup-armv7a-vfp-neon-poky-linux-gnueabi
unset LDFLAGS
arm-poky-linux-gnueabi-gcc main.c -o main
```

Question 1.3 : *Utilisez de nouveau la commande file, que constatez vous ?*
Vérifiez ensuite le bon fonctionnement sur la cible.

```
file main
main: ELF 32-bit LSB executable, ARM, EABI5 version 1 (SYSV), dynamically
linked (uses shared libs), for GNU/Linux 2.6.32,
BuildID[sha1]=b17d26ffaef10e5340617cd2f0c7ce088fe0d6d6, not stripped
```

On constate alors que cette fois ci la compilation a bien été effectuée pour un processeur ARM.

on le copie sur la cible:

```
scp main root@192.168.1.6:/home/root/
```

puis après s'être connecté en ssh sur la cible on exécute:

```
./main
Hello World
```

Question 1.4 : *Fournissez le code dans le compte rendu et le résultat obtenu dans le terminal.*

Notre code:

```
#include <stdio.h>

int main () {
    printf("Hello World");
    return 1;
}
```

Résultat dans le terminal:

```
Hello World
```

Exercice 2 : Clignotement des LEDs

Nous allons maintenant manipuler les périphériques de la cible. Pour que ce soit visuel, nous allons
nous attaquer aux traditionnelles LEDs. Trois sont présentes sur la carte de développement, la rouge
étant réservée à la fonction POWER. Les deux LEDs vertes sont accessibles et correspondent aux
LEDs système et utilisateur.

Question 2.1 : *Rappelez comment accéder aux LEDs en manipulant des fichiers.*
Dans le terminal, affichez les valeurs de ces fichiers et modifiez-les.

Sur notre cible, on va modifier le fichier brightness des différents dossiers suivants:

`/sys/class/leds/user_led/brightness` (LED Utilisateur)
ou
`/sys/class/leds/sys_led/brightness` (LED Système)

On affiche les valeurs initiales puis on les changent à 1, les deux LEDs vertes s'allument:

```
cat /sys/class/leds/user_led/brightness
0
cat /sys/class/leds/sys_led/brightness
0
echo 1 > /sys/class/leds/user_led/brightness
echo 1 > /sys/class/leds/sys_led/brightness
```

Question 2.2 : *Fournissez les 4 lignes de commandes permettant d'allumer et d'éteindre les 2 LEDs.*

```
echo 1 > /sys/class/leds/user_led/brightness
echo 1 > /sys/class/leds/sys_led/brightness
echo 0 > /sys/class/leds/user_led/brightness
echo 0 > /sys/class/leds/sys_led/brightness
```

Créez maintenant un nouveau fichier led.c qui devra allumer en alternance la LED utilisateur et la LED système chaque seconde. Compilez ce code avec la chaîne de compilation croisée et testez-la sur la cible. Pour rappel, les entrées sorties étant vu comme des fichiers sous Linux, vous pouvez utiliser les fonctions open, read, write pour accéder aux IO et contrôler les LEDs.

Question 2.3 : *Fournissez le code source led.c et faites valider le fonctionnement de l'application par le chargé de TP.*

```
#include <fcntl.h>
#include <unistd.h>

int main() {
	char* sysLed = "/sys/class/leds/sys_led/brightness";
	char* userLed = "/sys/class/leds/user_led/brightness";
	char* on = "1";
	char* off = "0";
	int fSys, fUser;

	while (1){
		fSys = open( sysLed, O_RDWR | O_TRUNC );
		fUser = open( userLed, O_RDWR | O_TRUNC );
		if(fSys != -1 && fUser != -1) {
			write(fSys, on, 1);
			write(fUser, off, 1);
		}
		close(fSys);
		close(fUser);

		sleep(1);

		fSys = open( sysLed, O_RDWR | O_TRUNC );
		fUser = open( userLed, O_RDWR | O_TRUNC );
		if(fSys != -1 && fUser != -1) {
			write(fSys, off, 1);
			write(fUser, on, 1);
		}
		close(fSys);
		close(fUser);

		sleep(1);
	}
}
```

on le compile puis le copie sur la cible pour ensuite l'exécuter en ssh, on s'aperçoit du bon fonctionnement du code:

```
arm-poky-linux-gnueabi-gcc led.c -o led
scp led root@192.168.1.6:/home/root/

./led
```

Exercice 3 : Boutons poussoirs

Question 3.1 : *Combien y a t-il de boutons poussoirs sur la cible ? Comment y accède t-on ?*

Il y a 4 boutons poussoirs sur la cible:

```
User manual p. 40 :
S1 (HOME) : User-defined key
S2 (MENU) : System menu key
S3 (BACK) : System back key
S4 (SWITCH PUSH BUTTON) : Power Switch button
```

Nous y accèdons à travers le fichier `/dev/input/event1` qui interface le GPIO.

Faites un essai de lecture de l'état des boutons en ligne de commande.
Question 3.2 : *Donnez les commandes utilisées. Quelles sont les valeurs des différents événements ?*
Nous pouvons tester leur fonctionnement grâce à l'utilitaire `evtest`, qui affiche le contenu du fichier `event1` (manual p.73):

```
evtest /dev/input/event1
```

On accède alors au différentes valeurs des boutons lorsqu'on les déclenche:

```
evtest /dev/input/event1
Input driver version is 1.0.1
Input device ID: bus 0x19 vendor 0x1 product 0x1 version 0x100
Input device name: "gpio-keys"
Supported events:
  Event type 0 (EV_SYN)
  Event type 1 (EV_KEY)
  Event code 1 (KEY_ESC)
  Event code 59 (KEY_F1)
  Event code 102 (KEY_HOME)
Properties:
Testing ... (interrupt to exit)
Event: time 1492183091.889940, type 1 (EV_KEY), code 102 (KEY_HOME), value 1
Event: time 1492183091.889944, -------------- SYN_REPORT ------------
Event: time 1492183092.598825, type 1 (EV_KEY), code 102 (KEY_HOME), value 0
Event: time 1492183092.598827, -------------- SYN_REPORT ------------
Event: time 1492183093.327590, type 1 (EV_KEY), code 59 (KEY_F1), value 1
Event: time 1492183093.327593, -------------- SYN_REPORT ------------
Event: time 1492183093.775184, type 1 (EV_KEY), code 59 (KEY_F1), value 0
Event: time 1492183093.775186, -------------- SYN_REPORT ------------
Event: time 1492183104.686032, type 1 (EV_KEY), code 1 (KEY_ESC), value 1
Event: time 1492183104.686035, -------------- SYN_REPORT ------------
Event: time 1492183104.913303, type 1 (EV_KEY), code 1 (KEY_ESC), value 0
Event: time 1492183104.913305, -------------- SYN_REPORT ------------
```

On voit par exemple dans les dernières lignes que le bouton poussoir "retour" (KEY_ESC) est déclenché.

Écrivez maintenant un programme réagissant aux actions sur les boutons, en affichant un message et/ou allumant une LED par exemple. Pour cela, vous devez stocker le résultat de la lecture du fichier dans une structure de type input_event, qui contiendra donc les informations liées à l'événement d'une touche.

Question 3.3 : *Faites une recherche dans la cross-toolchain pour trouver où est déclarée la struct input_event.*
Notre recherche:

```
grep -r "input_event" /opt/poky/1.7.3/
/opt/poky/1.7.3/sysroots/armv7a-vfp-neon-poky-linux-
gnueabi/usr/include/linux/input.h:struct input_event { ...
```

Cette structure est donc déclarée dans le fichier `input.h`.

on découvre la structure suivante:

```
struct input_event {
    struct timeval time;
    __u16 type;
    __u16 code;
    __s32 value;
};
```

Cross-compilez votre programme et testez le.
Question 3.4 : *Fournissez le code source et faites valider le fonctionnement de l'application par le chargé de TP.*

Code source, button.c:

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

int main(int argc, char *argv[]) {
    char* btn = "/dev/input/event1";
    char* sysLed = "/sys/class/leds/sys_led/brightness";
    char* userLed = "/sys/class/leds/user_led/brightness";
    int fSys, fUser, fBtn, rd;

    char* on = "1";
    char* off = "0";
    struct input_event evt;
    int evtSize = sizeof (struct input_event);

    while (1){
     // fSys = open( sysLed, O_RDWR | O_TRUNC );
     // fUser = open( userLed, O_RDWR | O_TRUNC );

     fBtn = open ( btn, O_RDONLY );

     if ( (rd = read (fBtn, &evt, evtSize)) >= evtSize) {


       if (evt.value != ' ' && evt.type == EV_KEY && evt.code == KEY_ESC){ //
Only read the key press event
         fUser = open( userLed, O_RDWR | O_TRUNC );
```

```
        if (evt.value == 1) {
          if(fSys != -1 && fUser != -1) {
            printf("Switch BACK on\n");
            write(fUser, on, 1);
            close(fUser);
          }
        } else {
          if(fSys != -1 && fUser != -1) {
            printf("Switch BACK off\n");
            write(fUser, off, 1);
            close(fUser);
          }
        }
      } else if (evt.value != ' ' && evt.type == EV_KEY && evt.code == KEY_F1)
{ // Only read the key press event
        fSys = open( sysLed, O_RDWR | O_TRUNC );
        if (evt.value == 1) {
          if(fSys != -1 && fUser != -1) {
            printf("Switch MENU on\n");
            write(fSys, on, 1);
            close(fSys);
          }
        } else {
          if(fSys != -1 && fUser != -1) {
            printf("Switch MENU off\n");
            write(fSys, off, 1);
            close(fSys);
          }
        }
      }

     }

    }
}
```

Notre programme ci-dessus nous permet d'allumer avec le bouton KEY_ESC la led utilisateur et avec le bouton KEY_F1 la led système.

Exercice 4 : Charge CPU

Cet exercice va vous permettre de vérifier l'incidence de la charge CPU sur une tâche périodique.

Ecrivez donc un programme réalisant 10 000 fois une attente de 1ms. Utilisez les fonctions gettimeofday et timersub pour mesurer le temps total. Utilisez maintenant la commande stress pour charger le CPU.

Exemple de l'utilisation de stress avec 100 workers CPU:

```
stress -c 100
```

Question 4.1 : *Quels sont les différents temps relevés ?*

```
Sortie sans stress :
10 sec 888306 µs

Avec 100 workers CPU :
15 sec 953746 µs

Avec 500 workers CPU :
99 sec 725782 µs
```

Améliorez votre programme afin de calculer et afficher les latences minimum, maximum et moyenne.

Question 4.2 : *Quels sont les différents temps relevés ? Quelles sont vos conclusions ? Comment pourrait-on améliorer les résultats ?*

Sur les 10000 itérations (nbLap) de notre boucle attendant 1ms, on relève (dans tvLap) le temps que prend chaque itération effectivement.
Puis on enregistre la latence la plus rapide et la plus longue. Finalement, on calcule la moyenne.

```
Sortie sans stress :
Min   1019        µs
Max   1699        µs
Mean  1088.830600 µs

Avec 100 workers CPU :
Min   1028        µs
Max   114652      µs
Mean  1595.374600 µs

Avec 500 workers CPU :
Min   1027        µs
Max   1668599     µs
Mean  9972.578200 µs
```

On observe que les minimums atteints pour chaque cas sont plutôt similaires, cependant les max diffèrent totalement, Notons que même sans stress le max est plus élevé de 50%. Il est donc préférable de regarder la moyenne pour juger de l'impact de la charge CPU sur notre programme. On se rend compte que notre programme ne supporte absolument pas ces charges CPUs et est de plus en plus retardé plus celles ci augmentent. La charge CPU a un effet considérable sur les tâches périodiques.

Il faut donc utilisé un OS temps réel (ex: Xenomai) qui permettrait alors d'exécuter les tâches en temps réel, si l'on veut que les tâches ne soient pas retardées.

Question 4.3 : *Fournissez le code source de votre programme.*

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

int main(int argc, char *argv[]) {
    const int nbLap = 10000;
    struct timeval tvRes;
    struct timeval tvLap[nbLap + 1];
    struct timeval intervals[nbLap + 1];
    int i;
    unsigned long max = 0;
    unsigned long min = 1000000; // 1s
    unsigned long tmp;
    double mean;

    printf("Start\n");

    gettimeofday(&tvLap[0], NULL);

    // Mesures
    for (i = 1; i < nbLap; i++) {
     usleep((useconds_t)1000);
     gettimeofday(&tvLap[i], NULL);
    }

    printf("End\n");

    // Calculs
    for (i = 0; i < nbLap - 1; i++) {
     timersub(&tvLap[i+1], &tvLap[i], &intervals[i]);

     tmp = intervals[i].tv_sec * 1000000 + intervals[i].tv_usec;

     if (tmp > max) {
       max = tmp;
     } else if (intervals[i].tv_usec < min) {
       min = tmp;
     }
    }

    timersub(&tvLap[nbLap-1], &tvLap[0], &tvRes);
    mean = ((double)(tvRes.tv_sec * 1000000 + tvRes.tv_usec)) / nbLap;

    printf("%ld sec %06ld µs\n", tvRes.tv_sec, tvRes.tv_usec);
    printf("Mean %f µs\n", mean);
    printf("Min %ld µs\n", min);
    printf("Max %ld µs\n", max);
}
```