Hes·so
Haute Ecole Spécialisée de Suisse occidentale
Fachhochschule Westschweiz
University of Applied Sciences and Arts Western Switzerland

MSE | MASTER OF SCIENCE IN ENGINEERING

# MA-CSEL
# Conception système Embarqué Linux
# Mini-Projet

Kirill Goundiaev & Tanguy Dietrich

June 10, 2023

# Contents

# 1 Introduction

Le projet est composé de 2 parties :

- Un module kernel qui permet de controler la frequence du processeur, et de changer le mode de fonctionnement du module kernel. Et il met a disposition les informations sur la temperature du CPU.
- Un daemon qui permet d'afficher les informations sur l'ecran, et de controler le module kernel.

# 2 Module Kernel

TODO

# 3 Daemon

Un Daemon est un programme qui s'execute en arriere plan. Son objectif sera de gerer l'affichage des donnée sur l'ecran, tel que la temperature, frequence, et le mode de fonctionnement du module kernel. Pour faire cela, il dispose de 2 interface de controle :

- Les 3 bouton present sur la carte pour augmenter/diminuer la frequence, et changer le mode de fonctionnement.
- un IPC (Inter Process Communication) pour communiquer avec le module kernel. Dans notre cas, nous avons choisi d'utiliser un socket.

La création du daemon se fait en plusieurs étapes :
TODO WRITE ... Nous avons réutiliser le code fournis dans le cours comme point de départ pour la création du daemon.

## 3.1 IPC

La communication entre un processus externe et le daemon se fait par un socket. Ce qui permet de controller la carte depuis n'importe quel ordinateur connecté au même réseau. L'initialisation du socket se fait dans la fonction *void initSocket(int *mode, int *freq, pthread_t *thread_id, void* (*threadFunc)(void*))* qui est appelé dans la fonction *main*. La fonction prend un pointeur vers le mode, la frequence, et l'id du thread, ainsi qu'un pointeur vers la fonction qui sera executé par le thread. Afin de traiter les donnée recu. Le socket est initialisé avec l'adresse IP de la carte, et le port 8080.

Comme nous étions libre pour le choix du protocole de communication, nous avons choisi d'utiliser un protocole simple, qui permet de controller le mode et la frequence du module kernel. Le protocole est composé de 2 commandes :

- M*X* : Permet de choisir le mode de fonctionnement du module kernel. *X* peut prendre 2 valeurs : 0 pour le mode manuel, et 1 pour le mode automatique.
- F*XXX* : Permet de changer la frequence de clignotement de la LED. *XXX* est un nombre entre 0 et 999.

Par exemple envoyer "M0", puis "F5" aura pour effet de mettre le mode manuel, et de mettre la frequence de clignotement à 5 Hz.
Voici le code du thread qui vas s'occuper de traiter les donnée recu par le socket :

```
1 void *threadSocket(void *arg)
2 {
3         int client_fd = 0;
4         char buffer[SOCKET_BUFFER_SIZE] = {0};
5         socketParamThread *param = (socketParamThread*) arg;
6         int addresslen = sizeof(param->address);
7         //listen on the socket
8         if((client_fd = accept(param->server_fd, (struct sockaddr*)&param->address,
```

```
 9                 syslog(LOG_ERR, "accept");
10                 exit(EXIT_FAILURE);
11         }
12         syslog(LOG_INFO, "threadSocket started\n");
13         while(1) {
14                 int valread = read(client_fd, buffer, SOCKET_BUFFER_SIZE);
15                 if (valread == 0) {
16                         syslog(LOG_INFO, "client disconnected\n");
17                         close(client_fd);
18                         client_fd = accept(param->server_fd, (struct sockaddr*)&para
19                 } else {
20                         if (buffer[0] == 'M') {
21                                 *param->mode = buffer[1] - '0';
22                                 writeMode(*param->mode);
23                                 if (*param->mode == 0) {
24                                         writeFreq(*param->freq);
25                                 }
26                         } else if (buffer[0] == 'F') {
27                                 *param->freq = atoi(&buffer[1]);
28                                 writeFreq(*param->freq);
29                         }
30                         syslog(LOG_INFO, "received: %s\n", buffer);
31                 }
32         }
33         free(param);
34         return NULL;
35 }
```

Le thread vas attendre qu'un client se connecte, puis il vas lire les donnée recu, et les traiter. Si le client se déconnecte, le thread vas attendre qu'un nouveau client se connecte. Le code ne permet pas à plusieur client de se connecter en même temps, mais il serais possible de le faire en créant un thread qui attend des connection, et qui créer un nouveau thread pour chaque client qui se connecte.

## 3.2 Control des LED

L'acces a la LED par le daemon se fait par l'interface sysfs. Pour cela, nous avons créer une fonction qui permet d'initialiser les LED, et une fonction qui permet de choisir l'etat de la led rouge. Le prototype de la fonction d'écriture est : *void writeLed(int value)*. Voici le code d'initialisation des LED :

```
1 void initLeds()
2 {
3         int f = open(GPIO_UNEXPORT, O_WRONLY);
4         write(f, LED, strlen(LED));
5         close(f);
6
7         // export pin to sysfs
8         f = open(GPIO_EXPORT, O_WRONLY);
9         write(f, LED, strlen(LED));
10        close(f);
11
12        // config pin
13        f = open(GPIO_LED "/direction", O_WRONLY);
14        write(f, "out", 3);
15        close(f);
16        g_led_fd = open(GPIO_LED "/value", O_WRONLY);
17        syslog(LOG_INFO, "leds initialized\n");
18 }
```

Le code commence par exporter les LED, puis il configure les LED en sortie, en écrivant "out" dans le fichier direction. Enfin, il ouvre le fichier value, qui permet d'écrire dans la LED. Le descripteur de fichier est stocké dans la variable globale *g_led_fd*.

### 3.3 Bouton

Pour réaliser cette partie du TP, nous avons en partie réutilisé le code que nous avions écrit pour le rendu precedent. Notre code utilise les epoll pour gérer les interruptions des boutons. L'attente des evenement sur les bouton s'effectue dans le main avec la fonction *epoll_wait()*.

## 4 Conclusion