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 13, 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

Ce module kernel a pour but de gérer le refroidissement de notre système embarqué, qui est simulé par le clignotement d'une LED. Le module propose deux modes de fonctionnement :

- Manuel : L'utilisateur peut choisir la fréquence de clignotement de la LED.
- Automatique : La fréquence de clignotement de la LED est calculée en fonction de la température du microprocesseur.

Le module met à disposition une interface de contrôle à travers *sysfs* permettant de modifier le mode de fonctionnement, la fréquence de la Modulation de largeur d'impulsion (PWM) et de lire la température du microprocesseur.

Toute la communication avec le module se fait à travers les attribus de la class. Nous en mettons trois à disposition :

- *auto_config* : Permet de lire et modifier le mode de fonctionnement du module. Il peut prendre deux valeurs : 0 pour le mode manuel et 1 pour le mode automatique.
- *frequency_Hz* : Permet de lire et modifier la fréquence en Hz du PWM si le mode manuel est activé. La valeur 0 permette d'arrêter le PWM et tout autre valeur positive fixe la fréquence du PWM,
- *temperature_mC* : Permet de lire la température du microprocesseur en millidegré celsius.

La mise en place des attributs se fait de la manière suivante :

```
DEVICE_ATTR(frequency_Hz, 0664, show_frequency_Hz, store_frequency_Hz);
DEVICE_ATTR(auto_config, 0664, show_auto_config, store_auto_config);
DEVICE_ATTR(temperature_mC, 0444, show_temperature_mC, NULL);

static int __init my_module_init(void){
  /* ... */
  device_create_file(my_device, &dev_attr_frequency_Hz);
  device_create_file(my_device, &dev_attr_auto_config);
  device_create_file(my_device, &dev_attr_temperature_mC);
  /* ... */
}
```

Dans le code ci-dessus, les fonction *show_*()* et *store_*()* sont des fonctions qui permettent de lire et modifier les attributs. Elles sont appelées automatiquement par le système d'exploitation lorsqu'un utilisateur lit ou modifie un attribut. Elles sont définies selon le modèle suivant :

```
ssize_t store_attr(struct device *dev, struct device_attribute *attr, const char *buf, size_t
    count){
  int new_value;
  sscanf(buf, "%d", &new_value);
  my_device_attribute.value = new_value;
  return count;
}

ssize_t show_attr(struct device *dev, struct device_attribute *attr, char *buf){
  sprintf(buf, "%d\n", my_device_attribute.value);
  return strlen(buf);
}
```

Etant donné que ces attributs permettent de contrôler le module, une logique supplémentaire est intégré dans les fonctions ci-dessus, afin de contrôler la cohérence des valeurs. Par exemple, la fréquence ne peut pas être modifié si le module est en mode auto ou être négative.

Le module utlise deux timers indépendants pour gérer le clignotement de la LED et la lecture de la température. Les deux timers sont périodiques, le premier est configuré avec la fréquence donnée par l'attribut et le second avec une période de 500 milisecondes.
Pour les configurer, nous utilisons une structure *timer_list* qui contient les informations nécessaires à la gestion du timer. Nous utilisons la fonction *setup_timer*() pour initialiser la structure avec la callback voulue et la fonction *mod_timer*() pour démarrer le timer. Ce timer a comme unité de temps les *jiffies* qui est une unité de temps définie par le système d'exploitation et représente le temps entre deux ticks d'horloge successives. Voici l'example de configuration du timer de température :

```
static struct timer_list timer_temprature;

void run_timer(struct timer_list *timer, unsigned long period_us){
  /* ... */
  mod_timer(timer, jiffies + usecs_to_jiffies(period_us));
}

void timer_temprature_callback(struct timer_list *timer){
  /* ... */
  // run timer again
  run_timer(&timer_fan, my_device_attribute.period_us_d2);
}

static int __init my_module_init(void){
  /* ... */
    timer_setup(&timer_temprature, timer_temprature_callback, 0);
    run_timer(&timer_temprature, TEMP_PERIOD_US);
  /* ... */
}
```

Les callbacks des timers nous permettent de mettre à jour la temperature avec laquelle nous recalculons la nouvelle frequence du PWM et de créer un signal PWM avec un duty cycle de 50% et la fréquence voulue.
Nos deux callbacks sont les suivantes :

```
void timer_temprature_callback(struct timer_list *timer){
  struct thermal_zone_device *tzd;
  int temperature, ret, last_f;
  run_timer(&timer_temprature, TEMP_PERIOD_US);

  tzd = thermal_zone_get_zone_by_name("cpu-thermal");
  ret = thermal_zone_get_temp(tzd, &temperature);
  if(ret){
    pr_err("Pilotes Fan_ctl: Error in thermal_zone_get_temp\n");
    return;
  }
  my_device_attribute.temperature_mC = temperature;
  if(my_device_attribute.auto_config){
    last_f = my_device_attribute.frequency_Hz;
    if(temperature < TEMP_THR_1){
      my_device_attribute.frequency_Hz = TEMP_FRQ_1;
    }else if(temperature < TEMP_THR_2){
      my_device_attribute.frequency_Hz = TEMP_FRQ_2;
    }else if(temperature < TEMP_THR_3){
      my_device_attribute.frequency_Hz = TEMP_FRQ_3;
    }else{
      my_device_attribute.frequency_Hz = TEMP_FRQ_4;
    }
    my_device_attribute.period_us_d2 = S_IN_US / (2 * my_device_attribute.frequency_Hz);
    if(last_f == 0){
      run_timer(&timer_fan, my_device_attribute.period_us_d2);
    }
  }
}

void timer_fan_callback(struct timer_list *timer){
  static int state = 0;
  if(my_device_attribute.period_us_d2 == 0){
    state = 0;
  }else{
    run_timer(&timer_fan, my_device_attribute.period_us_d2);
    state = (state + 1) % 2;
  }
  gpio_set_value(GPIO_FAN, state);
```

```
40 }
```

La fonction *thermal_zone_get_zone_by_name*() permet de récupérer la zone thermique du CPU. La fonction *thermal_zone_get_temp*() permet de récupérer la température de la zone thermique, que nous stockons dans l'attribut *temperature_mC*.

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.

Nous avons réutiliser le code fournis dans le cours comme point de départ pour la création du daemon.

Nous avons repartie le code de notre programme dans differentes librairies afin de le rendre plus lisible, voici les fichier qui le compose, et leux contenue :

- *main.c*
  Ce fichier contien le main de notre programme, il initialise le daemon et contient le thread de reception des message du socket.
- *daemonfanlib.c/h*
  Cette librairie contient les fonction necessaire a l'initialisation du socket et de la creation du daemon. Le constante qui indique le port du socket.
- *gpio_utility.c/h*
  Cette librairies permet d'acceder au GPIO de la carte, ainsi qu'au information fournis par le module kernel. le header contient divers constante qui permettent de definir les GPIO utilisé.
- *ssd1306.c/h*
  Cette librairies fournis les fonction utiles à l'utilisation de l'écran. Cette librairies nous à été fournis dans le cours.

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 :

```c
static void *threadSocket(void *arg)
{
  int client_fd = 0;
  char buffer[SOCKET_BUFFER_SIZE] = {0};
  // get the parameters
  socketParamThread *param = (socketParamThread*) arg;
  int addresslen = sizeof(param->address);
  //listen on the socket
  if((client_fd = accept(param->server_fd, (struct sockaddr*)&param->address, ((socklen_t*) &
    addresslen))) < 0) {
    syslog(LOG_ERR, "accept");
    exit(EXIT_FAILURE);
  }

  syslog(LOG_INFO, "threadSocket started\n");
  while(1) {
    int valread = read(client_fd, buffer, SOCKET_BUFFER_SIZE);
    if (valread == 0) {
      syslog(LOG_INFO, "client disconnected\n");
      close(client_fd);
      client_fd = accept(param->server_fd, (struct sockaddr*)&param->address, ((socklen_t*) &
    addresslen));
    } else {
      if (buffer[0] == 'M') {
        *param->mode = buffer[1] - '0';
        writeMode(*param->mode);
        if (*param->mode == 0) {
          writeFreq(*param->freq);
        }
      } else if (buffer[0] == 'F') {
        *param->freq = atoi(&buffer[1]);
        writeFreq(*param->freq);
      }
      syslog(LOG_INFO, "received: %s\n", buffer);
    }
  }
  free(param);
  return NULL;
}
```

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 :

```c
void initLeds()
{
  int f = open(GPIO_UNEXPORT, O_WRONLY);
  write(f, LED, strlen(LED));
  close(f);

  // export pin to sysfs
  f = open(GPIO_EXPORT, O_WRONLY);
  write(f, LED, strlen(LED));
  close(f);

  // config pin
  f = open(GPIO_LED "/direction", O_WRONLY);
  write(f, "out", 3);
  close(f);
  g_led_fd = open(GPIO_LED "/value", O_WRONLY);
  syslog(LOG_INFO, "leds initialized\n");
}
```

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*.
Pour finir un simple appel a la fonction *void writeLed(int value)* permet de changer l'etat de la LED rouge.

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. afin de rendre le code main plus lisible, nous avons écrit une fonction *int initButtonsAndTimer()* qui permet d'initialiser les boutons, et un timer pour etre utiliser avec les epoll. Cette fonction fais appel au fonction epoll_create1(), epoll_ctl(), et timerfd_create() pour créer les epoll et le timer. puis elle retourne un descripteur de fichier qui permet de lire les evenement sur les boutons, et le timer.

```
// init the buttons S1, S2 and S3, and the timer
epfd = initButtonsAndTimer();
```

L'attente des evenement sur les bouton s'effectue dans le main avec la fonction *epoll_wait()*. Voici un extrait du code main, qui permet de lire les evenement sur les boutons :

```
while (1) {
  struct epoll_event event_arrived[NUM_EVENTS];
  syslog(LOG_INFO, "waiting for event epoll\n");
  writeLed(LED_OFF);
  int nr = epoll_wait(epfd, event_arrived, NUM_EVENTS, -1);
  syslog(LOG_INFO, "event arrived\n");
  if (nr == -1) {
    // printf("error epoll_wait: %s\n", strerror(errno));
    syslog(LOG_ERR, "epoll_wait");
    exit(EXIT_FAILURE);
  }
  for(int i = 0; i < nr; i++){
    my_context *ctx = event_arrived[i].data.ptr;

    switch (ctx->ev){
    case EV_BTN_1: // increase frequence
      syslog(LOG_INFO, "button 1 pressed\n");
      if(ctx->first_done == 0){
        ctx->first_done = 1;
        break;
      }
      freq++;
      writeFreq(freq); // will fail if in auto mode
      writeLed(LED_ON);
    break;
    // ...
```

Dans le epoll, nous avons aussi ajouté le timer, qui permet de raffraichir l'affichage de la temperature, et de la frequence en mode automatique. Le timer est initialisé avec une periode de 200ms, il est possible de le modifier dans le fichier gpio_utility.h, en modifiant la constante *DEFAULT_PERIOD*. Voici un extrait de la gestion de l'evevement timer :

```
case EV_TIMER:
// read the actual mode
syslog(LOG_INFO, "timer expired\n");
updateTempCPU();
if(mode == 1) // if in auto mode
{
  // read the actual freq
  freq = readFreq();
  // show it on the screen
  writeLCDFreq(freq);
}
else // if in manual mode
{
  // syslog(LOG_INFO, "manual mode\n");
}
break;
```

3.4 Concurence

Comme notre programme est multithreadé (thread socket et thread main), il est possible d'avoir des problèmes de concurence. Pour eviter cela nous avons ajouter un mutex ces les commande envoyé au LCD. Les variable de mode et de frequence sont aussi partagé, et peuvent etre modifié par le thread socket, et par le main. mais dans ce cas, les risques sont assez faible, il faudrait qu'un requette soit envoyé au moment au quelqun écrit sur le socket. Exemple de protection dans la fonction *void writeLCDFreq(int freq)* :

```
void writeLCDFreq(int freq)
{
    char str[32] = {0};
    sprintf(str, "Freq: %03d Hz", freq);
    pthread_mutex_lock(&g_mutex_lcd);
    ssd1306_set_position (0 ,4);
    ssd1306_puts(str);
    pthread_mutex_unlock(&g_mutex_lcd);
}
```

Sans cela deux thread essayait d'écrire en meme temps sur le LCD, et le texte n'était pas affiché correctement.

4 Conclusion

VROOM VROOM ...