

MA-CSEL

Conception système Embarqué Linux Mini-Projet

Kirill GOUNDIAEV & Tanguy DIETRICH

June 16, 2023



Contents

| | | |
|----------|---------------------------|----------|
| 1 | Introduction | 3 |
| 2 | Module Kernel | 3 |
| 3 | Daemon | 5 |
| 3.1 | IPC Server | 5 |
| 3.2 | IPC Client | 6 |
| 3.3 | Control des LED | 7 |
| 3.4 | Bouton | 7 |
| 3.5 | Concurrence | 8 |
| 4 | Conclusion | 9 |

1 Introduction

Le projet est composé de 2 parties :

- Un module kernel qui permet de contrôler la fréquence du processeur, et de changer le mode de fonctionnement du module kernel. Et il met à disposition les informations sur la température du CPU.
- Un daemon qui permet d'afficher les informations sur l'écran, et de contrôler 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 attributs 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 toute 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 :

```

1 DEVICE_ATTR(frequency_Hz, 0664, show_frequency_Hz, store_frequency_Hz);
2 DEVICE_ATTR(auto_config, 0664, show_auto_config, store_auto_config);
3 DEVICE_ATTR(temperature_mC, 0444, show_temperature_mC, NULL);
4
5 static int __init my_module_init(void){
6     /* ... */
7     device_create_file(my_device, &dev_attr_frequency_Hz);
8     device_create_file(my_device, &dev_attr_auto_config);
9     device_create_file(my_device, &dev_attr_temperature_mC);
10    /* ... */
11 }
```

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 :

```

1 ssize_t store_attr(struct device *dev, struct device_attribute *attr, const char *buf, size_t
    count){
2     int new_value;
3     sscanf(buf, "%d", &new_value);
4     my_device_attribute.value = new_value;
5     return count;
6 }
7
8 ssize_t show_attr(struct device *dev, struct device_attribute *attr, char *buf){
9     sprintf(buf, "%d\n", my_device_attribute.value);
10    return strlen(buf);
11 }
```

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

Le module utilise 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 millisecondes.

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'exemple de configuration du timer de température :

```

1 static struct timer_list timer_temprature;
2
3 void run_timer(struct timer_list *timer, unsigned long period_us){
4     /* ... */
5     mod_timer(timer, jiffies + usecs_to_jiffies(period_us));
6 }
7
8 void timer_temprature_callback(struct timer_list *timer){
9     /* ... */
10    // run timer again
11    run_timer(&timer_fan, my_device_attribute.period_us_d2);
12 }
13
14 static int __init my_module_init(void){
15     /* ... */
16     timer_setup(&timer_temprature, timer_temprature_callback, 0);
17     run_timer(&timer_temprature, TEMP_PERIOD_US);
18     /* ... */
19 }

```

Les callbacks des timers nous permettent de mettre à jour la température avec laquelle nous recalculons la nouvelle fréquence du PWM et de créer un signal PWM avec un duty cycle de 50% et la fréquence voulue.

Nos deux callbacks sont les suivantes :

```

1 void timer_temprature_callback(struct timer_list *timer){
2     struct thermal_zone_device *tzd;
3     int temperature, ret, last_f;
4     run_timer(&timer_temprature, TEMP_PERIOD_US);
5
6     tzd = thermal_zone_get_zone_by_name("cpu-thermal");
7     ret = thermal_zone_get_temp(tzd, &temperature);
8     if(ret){
9         pr_err("Pilotes Fan_ctl: Error in thermal_zone_get_temp\n");
10        return;
11    }
12    my_device_attribute.temperature_mC = temperature;
13    if(my_device_attribute.auto_config){
14        last_f = my_device_attribute.frequency_Hz;
15        if(temperature < TEMP_THR_1){
16            my_device_attribute.frequency_Hz = TEMP_FRQ_1;
17        }else if(temperature < TEMP_THR_2){
18            my_device_attribute.frequency_Hz = TEMP_FRQ_2;
19        }else if(temperature < TEMP_THR_3){
20            my_device_attribute.frequency_Hz = TEMP_FRQ_3;
21        }else{
22            my_device_attribute.frequency_Hz = TEMP_FRQ_4;
23        }
24        my_device_attribute.period_us_d2 = S_IN_US / (2 * my_device_attribute.frequency_Hz);
25        if(last_f == 0){
26            run_timer(&timer_fan, my_device_attribute.period_us_d2);
27        }
28    }
29 }
30
31 void timer_fan_callback(struct timer_list *timer){
32     static int state = 0;
33     if(my_device_attribute.period_us_d2 == 0){
34         state = 0;
35     }else{
36         run_timer(&timer_fan, my_device_attribute.period_us_d2);
37         state = (state + 1) % 2;
38     }
39     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'exécute en arrière plan. Son objectif sera de gérer l'affichage des données sur l'écran, tel que la température, fréquence, et le mode de fonctionnement du module kernel. Pour faire cela, il dispose de 2 interfaces de contrôle :

- Les 3 boutons présents sur la carte pour augmenter/diminuer la fréquence, et changer le mode de fonctionnement.
- Un IPC (Inter Process Communication) pour recevoir des requêtes. Dans notre cas, nous avons choisi d'utiliser un socket.

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

Nous avons réparti le code de notre programme dans différentes bibliothèques afin de le rendre plus lisible, voici les fichiers qui le composent, et leur contenu :

- `main.c`
Ce fichier contient le main de notre programme, il initialise le daemon et contient le thread de réception des messages du socket.
- `daemonfanlib.c/h`
Cette bibliothèque contient les fonctions nécessaires à l'initialisation du socket et de la création du daemon. La constante qui indique le port du socket.
- `gpio_utility.c/h`
Cette bibliothèque permet d'accéder au GPIO de la carte, ainsi qu'à l'information fournie par le module kernel. Le header contient diverses constantes qui permettent de définir les GPIO utilisés.
- `ssd1306.c/h`
Cette bibliothèque fournit les fonctions utiles à l'utilisation de l'écran. Cette bibliothèque nous a été fournie dans le cours.

3.1 IPC Server

La communication entre un processus externe et le daemon se fait par un socket. Ce qui permet de contrôler 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 appelée dans la fonction `main`. La fonction prend un pointeur vers le mode, la fréquence, et l'id du thread, ainsi qu'un pointeur vers la fonction qui sera exécutée par le thread. Afin de traiter les données reçues. Le socket est initialisé sur le port 8080.

Comme nous étions libres pour le choix du protocole de communication, nous avons choisi d'utiliser un protocole simple, qui permet de contrôler le mode et la fréquence que le daemon appliquera au module kernel. Le protocole est composé de 2 commandes :

- `MX` : 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.
- `FXXX` : Permet de changer la fréquence 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 fréquence de clignotement à 5 Hz.

Voici le code du thread qui va s'occuper de traiter les données reçues par le socket :

```

1 static void *threadSocket(void *arg)
2 {
3     int client_fd = 0;
4     char buffer[SOCKET_BUFFER_SIZE] = {0};
5     // get the parameters
6     socketParamThread *param = (socketParamThread*) arg;
7     int addresslen = sizeof(param->address);
8     //listen on the socket
9     if((client_fd = accept(param->server_fd, (struct sockaddr*)&param->address, ((socklen_t) &
10         addresslen))) < 0) {
11         syslog(LOG_ERR, "accept");
12         exit(EXIT_FAILURE);
13     }
14     syslog(LOG_INFO, "threadSocket started\n");
15     while(1) {
16         int valread = read(client_fd, buffer, SOCKET_BUFFER_SIZE);
17         if (valread == 0) {
18             syslog(LOG_INFO, "client disconnected\n");
19             close(client_fd);
20             client_fd = accept(param->server_fd, (struct sockaddr*)&param->address, ((socklen_t) &
21                 addresslen));
22         } else {
23             if (buffer[0] == 'M') {
24                 *param->mode = buffer[1] - '0';
25                 writeMode(*param->mode);
26                 if (*param->mode == 0) {
27                     writeFreq(*param->freq);
28                 }
29             } else if (buffer[0] == 'F') {
30                 *param->freq = atoi(&buffer[1]);
31                 writeFreq(*param->freq);
32             }
33             syslog(LOG_INFO, "received: %s\n", buffer);
34         }
35     }
36     free(param);
37     return NULL;
38 }

```

Le thread vas attendre qu'un client se connecte, puis il vas lire les donnée reçu, et les traiter. Si le client se déconnecte, le thread vas attendre qu'un nouveau client se connecte. Le code ne permet pas à plusieurs 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 IPC Client

Le control du daemon se fait par un petit programme C qui permet d'envoyer des commande au daemon. Nous avons ecrit une petit librairie *command.c* qui contient les commande suivante pour controler le daemon en passant par le socket, voici le header de la librairie :

```

1 #define DAEMON_ADDR "127.0.0.1"
2 #define DAEMON_PORT 8080
3
4 void init_socket();
5 void send_mode(int mode);
6 void send_freq(int freq);
7 void close_socket();

```

Pour utiliser cette librairie il suffit d'appeler la fonction *init_socket()* au debut du programme, puis d'appeler les fonction *send_mode()* et *send_freq()* pour envoyer des commande au daemon, et enfin d'appeler la fonction *close_socket()* à la fin du programme.

Voici un exemple d'utilisation de la librairie :

```

1 int main()
2 {
3     // try to pass to manual mode
4     init_socket();

```

```

5 // set mode to manual
6 send_mode(0);
7 // set freq to 20Hz
8 send_freq(20);
9 usleep(1000000); // wait 1s
10 // set freq to 2Hz
11 send_freq(2);
12 // wait 2s
13 usleep(2000000);
14 // set mode to auto
15 send_mode(1);
16 close_socket();
17 return 0;
18 }

```

Le programme va passer en mode manuel, puis il va changer la fréquence de clignotement de la LED, et enfin il va passer en mode automatique.

3.3 Control des LED

L'accès à la LED par le daemon se fait par l'interface sysfs. Pour cela, nous avons créé une fonction qui permet d'initialiser les LED, et une fonction qui permet de choisir l'état 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`.

Pour finir un simple appel à la fonction `void writeLed(int value)` permet de changer l'état de la LED rouge.

3.4 Bouton

Pour réaliser cette partie du TP, nous avons en partie réutilisé le code que nous avons écrit pour le rendu précédent. 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 être utilisé avec les epoll. Cette fonction fait appel à la 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 événements sur les boutons, et le timer.

```

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

```

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

```

1 while (1) {
2     struct epoll_event event_arrived[NUM_EVENTS];
3     syslog(LOG_INFO, "waiting for event epoll\n");
4     writeLed(LED_OFF);

```

```

5  int nr = epoll_wait(epfd, event_arrived, NUM_EVENTS, -1);
6  syslog(LOG_INFO, "event arrived\n");
7  if (nr == -1) {
8      // printf("error epoll_wait: %s\n", strerror(errno));
9      syslog(LOG_ERR, "epoll_wait");
10     exit(EXIT_FAILURE);
11 }
12 for(int i = 0; i < nr; i++){
13     my_context *ctx = event_arrived[i].data.ptr;
14
15     switch (ctx->ev){
16     case EV_BTN_1: // increase frequency
17         syslog(LOG_INFO, "button 1 pressed\n");
18         if (ctx->first_done == 0){
19             ctx->first_done = 1;
20             break;
21         }
22         freq++;
23         writeFreq(freq); // will fail if in auto mode
24         writeLed(LED_ON);
25         break;
26         // ...

```

Dans le epoll, nous avons aussi ajouté le timer, qui permet de rafraichir l'affichage de la temperature, et de la frequency 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'evenement timer :

```

1  case EV_TIMER:
2  syslog(LOG_INFO, "timer expired\n");
3  updateTempCPU();
4  if(mode == 1) // if in auto mode
5  {
6      // read the actual freq
7      freq = readFreq();
8      // show it on the screen
9      writeLCDFreq(freq);
10 }
11 else // if in manual mode
12 {
13     // syslog(LOG_INFO, "manual mode\n");
14 }
15 break;

```

3.5 Concurrency

Comme notre programme est multithreadé (thread socket et thread main), il est possible d'avoir des problèmes de concurrence. Pour éviter cela nous avons ajouté un mutex sur les commandes envoyées au LCD. Les variables de mode et de fréquence sont aussi partagées, et peuvent être modifiées par le thread socket, et par le main. Mais dans ce cas, les risques sont assez faibles, il faudrait qu'une requête soit envoyée au moment où quelqu'un écrit sur le socket. Toutefois, nous devrions aussi protéger l'accès à ces variables. Exemple de protection dans la fonction `void writeLCDFreq(int freq)` :

```

1  void writeLCDFreq(int freq)
2  {
3      char str[32] = {0};
4      sprintf(str, "Freq: %03d Hz", freq);
5      pthread_mutex_lock(&g_mutex_lcd);
6      ssd1306_set_position(0, 4);
7      ssd1306_puts(str);
8      pthread_mutex_unlock(&g_mutex_lcd);
9  }

```

Sans cela deux threads essaieraient d'écrire en même temps sur le LCD, et le texte n'était pas affiché correctement.

4 Conclusion

Notre programme fonctionne correctement, le rafraichissement de la temperature est bien effectué. Et lorsque que nous somme en mode automatique, la frequence actuellement utilisé est aussi actualisé. Le control de la frequence en mode manuel fonctionne correctement, et nous avons ajouté une ligne d’affichage pour indiquer le mode actuel. Il es possible de controler notre daemon avec un client socket, et de changer le mode, la frequence. Comme la communication passe par un socket, il est possible de controler le daemon depuis un autre ordinateur. Ce qui peut etre une fonctionnalité interessante, ou un probleme de securité. Nous pourrions mettre une regle dans le firewall, pour n’autoriser que certaine adresse IP à se connecter au socket.

Dans un soucis de propreté, nous voulions eviter d’utiliser un timer dans le daemon pour rafraichir l’affichage car un timer est deja present dans le module kernel. Nous avons chercher des solutions pour detecter les changement de valeur de ces variables comme les epoll pour les fichier se trouvant dans /sys/class. Comme nous n’avons pas trouvé de solution satisfaisante. Nous avons donc opté pour un timer, qui permet de rafraichir l’affichage toutes les 200ms. Nous avons trouver ce projet interessant, il nous a permis de mettre en pratique beaucoup de concept vu en cours, et de nous familiariser avec les daemon.