CSEL

Rapports des TP

Git du projet : https://github.com/Mathistis/csel-workspace

Fribourg, 15 juin 2022

Auteurs
Macherel Rémy
remy.macherel@master.hes-so.ch
Raemy Mathis
mathis.raemy@master.hes-so.ch

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

Version: 1.0

Table des matières

Raemy Mathis, Macherel Rémy, CSEL

1. Introduction

Ce rapport décrit l'architecture ainsi que le développement et les fonctionnalités du mini-projet réalisé lors du cours *MA-CSEL* suivi lors du semestre de printemps 2022 du master MSE. Ce projet consiste à mettre en pratique les notions vues dans le cours par l'intermédiaire de l'implémentation d'un gestionnaire de ventilateur pour le processeur de la cible. Notre cible n'ayant pas de réel ventilateur, son fonctionnement sera simulé par le clignotement d'une LED symbolisant la fréquence de celui-ci ainsi qu'un écran OLED affichant quelques valeurs importantes.
Le but du travail est donc de concevoir une application permettant de simuler la gestion de la vitesse de rotation d'un ventilateur en fonction de la température du processeur. Les fonctionnalités suivantes seront donc implémentées :

— Supervision de la température du processeur et la gestion de la vitesse de clignotement de la LED à l'aide d'un module noyau.

— Un daemon en espace utilisateur qui offrira des services pour une gestion manuelle et prendra en compte la gestion des appuis sur les boutons afin d'augmenter la vitesse de rotation, de la diminuer et de passer du mode manuel à automatique.
Ce daemon pourra également, à l'aide d'une interface IPC, communiquer avec une application de type *CLI* pour la gestion du clignotement et du mode.

— Une application *CLI* pour piloter le système via l'interface IPC.

2. Architecture logicielle

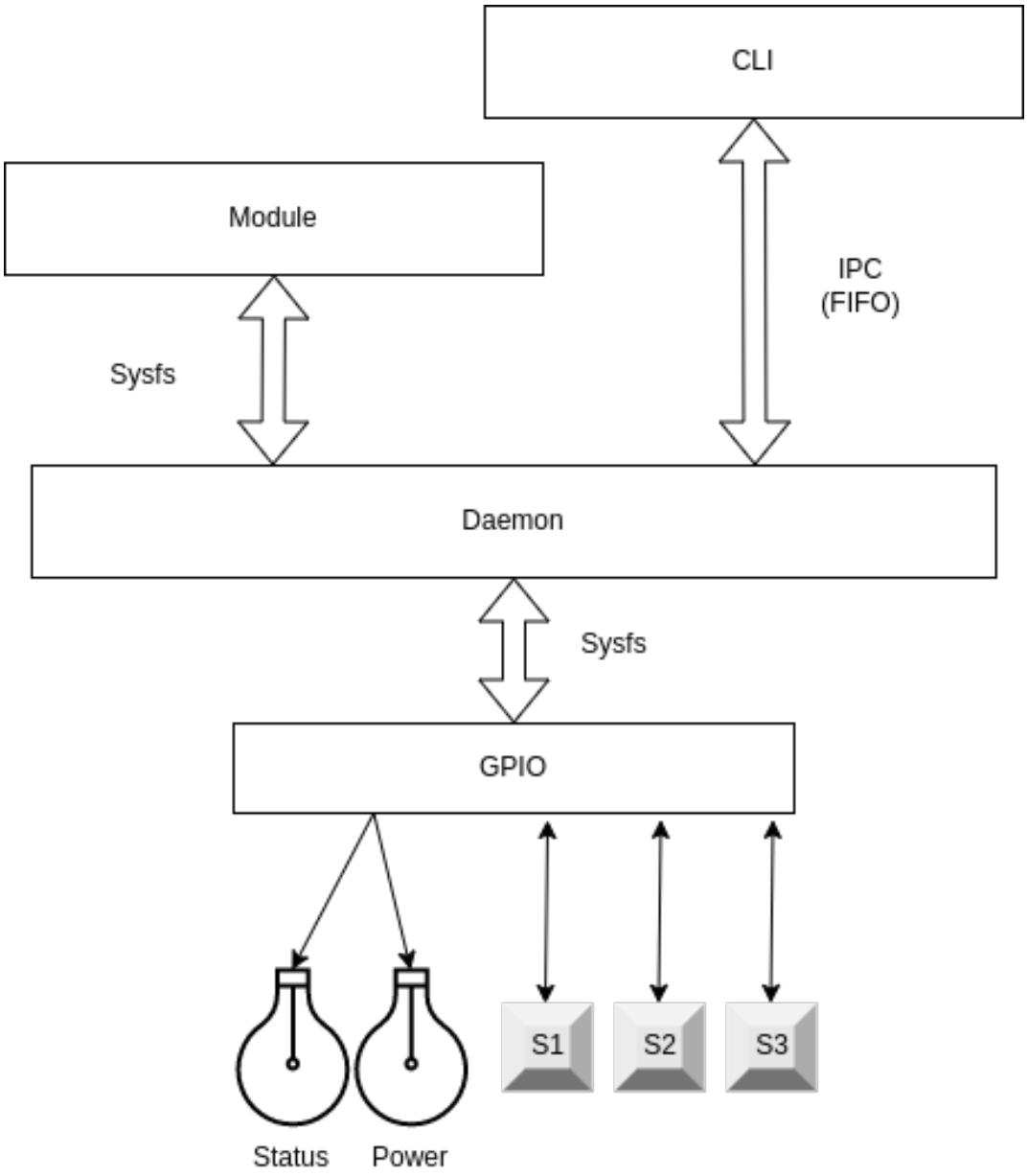

FIGURE 2.1: Diagramme de principe de l'application

3. Conception du module

Ce chapitre traite du développement du module noyau permettant de surveiller la température du processeur ainsi que la gestion du clignotement de la LED status. Ce module est dans le répertoire class et est composé de deux attributs :

— is_manu

— freq

Ces deux attributs sont représentés dans le userspace par des fichiers du sysfs. Le but de ce module est de faire clignoter la LED en mode automatique (si is_manu est à 0) ou en mode manuel (si is_manu est à 1). Le module implémente un timer (<*linux/timer.h*>) (voir fichiers *timer_controller.h et .c*) qui est créé avec une fonction à exécuter à chaque fois que son décompte est terminé. Lorsque le temps est écoulé, le programme vérifie le mode actuel grâce à l'attribut *is_manu* et si le mode actuel est manuel, il va regarder la fréquence inscrite dans le fichier *freq* puis calculer le prochain *jiffies* correspondant à la période de la fréquence de la manière suivante :

```
#define NEXT_JIFFIE_FROM_FREQ(freq) (jiffies + (HZ / freq))
```

Il va ensuite changer l'état de la LED et réamorcer le timer.
Pour le mode automatique, il va simplement lire la température du microprocesseur et obtenir la fréquence correspondante (voir figure 3.1), puis écrire cette valeur dans le fichier freq et réamorcer le timer.

- Température $< 35^{\circ}C \rightarrow$ fréquence de 2Hz
- Température $< 40^{\circ}C \rightarrow$ fréquence de 5Hz
- Température $< 45^{\circ}C \rightarrow$ fréquence de 10Hz
- Température $\geq 45^{\circ}C \rightarrow$ fréquence de 20Hz

FIGURE 3.1: Fréquences pour la led en mode automatique

Dans le module, nous avons séparé les codes en deux parties, un fichier *controller.c* (avec son .h dédié) qui représente une bibliothèque permettant la gestion des modes du programme. Ce controller permet d'initialiser les différents éléments utilisés dans le module (comme gpio, capteur de température, timers, etc.), il sert en quelque sorte d'interface de gestion des éléments du module noyau.
Le deuxième fichier *gpio.c* (également accompagné de son .h) permet quand à lui la gestion des gpio ainsi que leur initialisation.
Un autre code présent dans l'utilisation du module est le *temp_controller*, celui-ci permet à l'aide de la bibliothèque *linux/thermal.h* de récupérer la température actuelle du processeur.
Dans le fichier *skeleton.c* (en quelques sorte le main du module), nous appelons la fonction *init_controller* qui va permettre l'initialisation des gpio, du timer ainsi que de la lecture de

la température.
Ce module va également créer et initialiser dans le sysfs les fichiers nécessaires à la transmission de nos informations.
Notre module étant de type *class* il possède des attributs qui seront stocké dans le sysfs sous */sys/class/<mod_name/...* et lors de la déclaration des attributs du module (voir figure 3.2), on définit deux variables statiques au module ainsi que deux méthodes de lecture et écriture qui permettront de gérer ces deux fichiers.

```
static struct class *sysfs_class;
static struct device *sysfs_device;

DEVICE_ATTR(is_manu, 0664, sysfs_show_is_manu, sysfs_store_is_manu);
DEVICE_ATTR(freq, 0664, sysfs_show_freq, sysfs_store_freq);

static int __init skeleton_init(void)
{
    int status = 0;

    sysfs_class = class_create(THIS_MODULE, "led_class");
    sysfs_device = device_create(sysfs_class, NULL, 0, NULL, "led_device");
    if (status == 0)
        status = device_create_file(sysfs_device, &dev_attr_is_manu);
    if (status == 0)
        status = device_create_file(sysfs_device, &dev_attr_freq);
    if (status == 0)
        status = init_controller(&is_manu, &freq_manual);
    set_mode(MODE_AUTO);
    pr_info("Linux module skeleton loaded\n");
    return 0;
}
```

FIGURE 3.2: Création des fichiers nécessaires dans le sysfs

Dans l'initialisation du module (figure 3.2), on se charge également de créer le device pour le sysfs ainsi que les différents fichier qui nous servirons pour lire et écrire nos valeurs.

Nous avons également dans ce module déclaré quelque fonctions permettant la lecture ainsi que l'écriture des fichiers dans le sysfs.

```
static int is_manu = MODE_AUTO;
static int freq_manual = 1;

ssize_t sysfs_show_is_manu(struct device *dev,
                           struct device_attribute *attr,
                           char *buf)
{
    sprintf(buf, "%d\n", is_manu);
    return strlen(buf);
}
ssize_t sysfs_store_is_manu(struct device *dev,
                            struct device_attribute *attr,
                            const char *buf,
                            size_t count)
{
    is_manu = simple_strtol(buf, 0, 10) == 1 ? MODE_MANUAL : MODE_AUTO;

    return count;
}

ssize_t sysfs_show_freq(struct device *dev,
                        struct device_attribute *attr,
                        char *buf)
{
    sprintf(buf, "%d\n", freq_manual);
    return strlen(buf);
}
ssize_t sysfs_store_freq(struct device *dev,
                         struct device_attribute *attr,
                         const char *buf,
                         size_t count)
{
    sscanf(buf,
           "%d",
           &freq_manual);
    return count;
}
```

FIGURE 3.3: Fonctions de lecture/écriture des fichiers du sysfs

On observe dans la figure 3.3 que ces fonctions permettent de lire et écrire les différents fichiers correspondant par exemple au mode manuel ou l'affichage de la fréquence.

3.1 Difficultés rencontrées sur le module

Nous avons eu quelques soucis à fournir au début un code clair et bien séparé c'est pourquoi nous avons opté pour la solution des controllers ainsi que des fichiers séparés.
Une autre difficulté fût de bien comprendre que *DEVICE_ATTR* est en réalité une macro qui permet de créer plein de choses différentes concernant les fichiers du sysfs et au premier abord nous n'avions pas pleinement conscience de cela et avons eu un peu de peine à mettre en place ceci.

4. Conception du daemon

Ce chapitre traite du développement du deamon offrant les services permettant la gestion de la fréquence ainsi que le choix du mode.