MSE | MASTER OF SCIENCE IN ENGINEERING

Mini Projet
MA_CSEL

Quentin Rod

16 juin 2023

Table des matières

Introduction

Ce laboratoire a pour objectif d'exploiter les connaissances acquises au long du semestre. Il consiste à réaliser un gestionnaire de température automatique avec un ventilateur simulé par un clignotement de LED. Ce projet est constitué en trois parties

qui sont détaillées dans ce rapport. **Le code source est disponible dans le dossier annexe**.

1 Driver

Le driver réalisé permet de contrôler la vitesse du ventilateur, soit la fréquence de clignotement de la LED. Deux modes de fonctionnement sont présents :

— Mode **MANUAL** : L'utilisateur choisi la fréquence.

— Mode **AUTOMATIC** : La fréquence dépend de la température du CPU.

Par défaut, le mode automatique est activé. Afin d'observer le comportement du driver, l'ensemble des erreurs et des modifications au cours du temps sont répertoriées dans les logs situés au chemin **/var/log/messages**.

1.1 Clignotement de la LED

Un timer est dédié au clignotement de la LED. La structure utilisée est **struct timer_list** et est initialisée avec **timer_setup()**. Elle prend en paramètre la callback à appeler lorsque le temps est écoulé. L'intervalle de temps est indiqué avec la fonction **mod_timer()**. Afin de faciliter son utilisation, la fonction **timer_set_freq()** est réalisée et permet de choisir la fréquence d'intervalle. La fréquence choisie doit être le double de celle souhaitée afin d'avoir un rapport cyclique de 50%.

Dans la callback appelée lorsque le temps est écoulé, la fonction **timer_set_freq()** est utilisée pour relancer le timer. La fréquence spécifiée est celle contenue dans la variable globale **blinkingFreq**. C'est elle qui contient la fréquence à utiliser et est modifiée au cours du temps.

1.2 Contrôle de la température

Lorsque le gestionnaire est en mode **AUTOMATIC**, le driver doit contrôler la température et modifier la fréquence à utiliser en conséquence. La vérification est effectuée périodiquement grâce à un timer du même type que celui utilisé pour le clignotement de la LED.

Afin de pouvoir connaitre la température du CPU, un pointeur sur un **struct thermal_zone_device** doit être récupéré. Cette opération est effectuée à l'initialisation du module avec **thermal_zone_get_zone_by_name()**.

Dans la callback du timer de contrôle de température, la température est récupérée à l'aide de la fonction **thermal_zone_get_temp()** puis transformée de m°C à °C. En fonction de celle-ci, la variable contenant la fréquence du ventilateur à utiliser (**blinkingFreq**), est modifiée. Le timer du contrôle de température est également relancé. Dans le mode **AUTOMATIC**, l'intervalle de temps est de une seconde. Dans le mode **MANUAL**, la régulation ne doit plus être effectuée. Dans ce cas, la fonction **mod_timer()** est appelée avec un temps valant **LONG_MAX** ce qui correspond à la plus longue période possible.

1.3 Interface avec l'utilisateur

Une interface **sysfs** est mise à disposition par le driver afin d'échanger avec l'espace utilisateur. Dans le **sysfs**, une classe **mpcooling** est créée avec **class_create()**. Le gestionnaire nommé **controller** est ajouté comme appareil avec **device_create()**. Trois attributs sont crées grâce à la macro **DEVICE_ATTR()**. Ils sont les suivants.

— mode [RW] : '0' pour le mode **MANUAL** et '1' pour le **AUTOMATIC**.
— blinking [RW] : Fréquence de clignotement de la LED en ASCII.
— temperature [R] : Température du CPU en ASCII.

Les callbacks de lecture (**show**) et d'écriture (**store**) sont les mêmes pour l'ensemble des attributs. La discrimination est effectuée lors de la callback en utilisant le paramètre de la fonction, **struct device_attribute***. Il contient un membre **attr.name** pointant sur le nom de l'attribut concerné par la callback. Lors d'un **store**, la valeur reçue est convertie en nombre décimal avec **simple_strtol()**. Si la donnée est valide, elle est placée dans la variabe globale correspondante. Lors d'un **show**, la variable globale correspondante est transformée en ASCII avec **snprintf()** et placée dans le buffer de destination de l'espace utilisateur.

1.4 Sécurisation des accès

Les variables globales contenant l'état du gestionnaire (température, fréquence, mode) peuvent être accédées de manière concurrentes. Par exemple, la callback du timer de contrôle de la température peut être appelée en même temps que la lecture de la fréquence via le sysfs (**show**). Afin d'éviter des états incohérents, un mutex unique est utilisé pour protéger l'accès à ces variables globales.

2 Démon

Le démon réalisé fonctionne en arrière plan et permet de contrôler le gestionnaire depuis l'espace utilisateur. Des boutons sont mis à disposition de l'utilisateur afin de sélectionner le mode, incrémenter ou décrémenter la fréquence de la LED. Les différentes valeurs sont affichées au travers d'un écran. Egalement, le démon dispose d'un moyen de communication inter-processus permettant de configurer le gestionnaire depuis une application de l'espace utilisateur.

2.1 Configuration de l'interaction avec l'utilisateur

2.1.1 Boutons et LEDs

L'interface GPIO sysfs est utilisée. Elle est disponible au travers de la classe **gpio** et permet de contrôler les différentes pins. Il est premièrement nécessaire d'exporter les pins à configurer en écrivant leur numéro dans le fichier **export**. Un dossier **gpioX** est automatiquement créé, X étant le numéro de la pin exportée. Ce dossier contient les attributs paramétrables. Ceux utilisés sont **direction**, **edge**, **value** permettant respectivement de configurer la direction, le flanc d'interruption, la valeur.

2.1.2 Affichage sur l'écran

Les données affichées sur l'écran sont mises à jour périodiquement. Un timer est créé avec **timerfd_create()**. Le descripteur de fichier associé est retourné. L'intervalle entre deux interruptions est définie avec **timerfd_settime()**.

L'écriture des données sur l'écran est permise grâce à une librairie mise à disposition par le professeur. L'initialisation s'effectue avec **display_init()**.

Le device tree est modifié afin de pouvoir utilisé l'écran. Le fichier source utilisé par Buildroot est localisé dans **/buildroot/board/friendlyarm/nanopi-neo-plus2/nanopi-neo-plus2.dts**. L'élément suivant est ajouté afin d'autorisé l'I2C.

```
&i2c0 {
    status = "okay";
};
```

La commande **make** est appelée dans le dossier de buildroot afin de générer le nouveau contenu de la carte SD. Afin d'éviter de la reflasher intégralement, uniquement le device tree compilé au format **.dtb** est remplacé. La nouvelle version générée par buildroot se trouve dans **/buildroot/output/images**.

2.1.3 Communication inter-processus

Le mécanisme utilisé est le pipe nommé. Il est créé par le démon avec **mkfifo()** en spécifiant la localisation du fichier. Il est ensuite ouvert avec **open()** en spécifiant le mode **O_RDONLY** et **O_NONBLOCK**. Ces modes permettent respectivement d'autoriser uniquement la lecture et de ne pas bloquer lors de **open()** lorsque le pipe n'est pas ouvert en écriture.

2.2 Architecture du démon

2.2.1 Création du démon

Lors du lancement du programme, un démon est créé. Le code utilisé est basé sur celui fourni par le professeur. Les étapes principales effectuées sont les suivantes.

— Détachement complet avec deux **fork()** successifs.
— Interception des différents signaux.
— Fermeture des descripteurs de fichiers.
— Redirection des descripteurs **STDIN**, **STDOUT** et **STDERR**.

En utilisant le code fourni par le professeur, le comportement n'était pas celui escompté. Le problème semblait venir de la redirection de **STDIN**, **STDOUT** et **STDERR**. Le code étant le suivant.

```
if (open("/dev/null", O_RDWR) != STDIN_FILENO) {
    syslog(LOG_ERR, "ERROR while opening '/dev/null' for STDIN");
    exit(1);
}
if (dup2(STDIN_FILENO, STDOUT_FILENO) != STDOUT_FILENO) {
    syslog(LOG_ERR, "ERROR while opening '/dev/null' for STDOUT");
    exit(1);
```

```
}
if (dup2(STDIN_FILENO, STDERR_FILENO) != STDERR_FILENO) {
    syslog(LOG_ERR, "ERROR while opening '/dev/null' for STDERR");
    exit(1);
}
```

STDOUT et **STDERR** sont redirigés sur **STDIN** mais **STDIN** n'est pas redirigé sur **/dev/null**. La redirection a été ajoutée et les descripteurs fermés.

2.2.2 Mise en veille du processus

L'objectif est que le démon n'occupe pas le CPU inutilement et soit réveillé uniquement lorsqu'un événement intéressant est survenu. Le multiplexage des entrées proposé par **epoll** est utilisé. Un groupe est créé avec **epoll_create1()**. Les descripteurs de fichiers ayant un événement à surveiller sont ajoutés au groupe à l'aide de **epoll_ctl()** et le paramètre **EPOLL_CTL_ADD**. La variable privée permet de discriminer le descripteur de fichier concerné par l'événement. La valeur utilisée est le descripteur de fichier lui-même.

L'attente passive est effectuée avec **epoll_wait()** en spécifiant le contexte. Lors du réveil, le descripteur de fichier associé à l'événement étant discriminé, la fonction de traitement correspondante est effectuée. Les événements attendus sont les suivants.

— Timer [EPOLLIN] : Affichage périodique des données sur l'écran.
— Bouton S1 [EPOLLIN — EPOLLET] : Appui sur le bouton S1.
— Bouton S2 [EPOLLIN — EPOLLET] : Appui sur le bouton S2.
— Bouton S3 [EPOLLIN — EPOLLET] : Appui sur le bouton S3.
— Pipe [EPOLLIN] : Message reçu du pipe nommé.

2.3 Gestion de l'interaction avec l'utilisateur

2.3.1 Affichage sur l'écran

Lorsque le timer est écoulé, le processus sort de **epoll_wait()** et la callback de traitement est appelée. La fonction **pread()** est utilisée sur les attributs **sysfs** du gestionnaire afin de récupérer le mode, la fréquence ainsi que la température. Les valeurs étant en ASCII, elles sont directement affichées à l'écran avec **ssd1306_set_position()** et **ssd1306_puts()**. La figure 1 montre l'affichage réalisé sur l'écran.

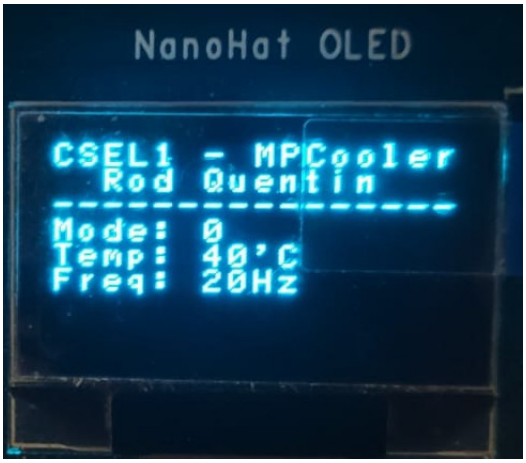

FIGURE 1 – Affichage des valeurs du gestionnaire sur l'écran

2.3.2 Appui sur un bouton

Lors de l'appui sur un bouton, le descripteur permet d'identifier lequel a été appuyé. Afin de modifier la valeur de l'attribut, elle est premièrement lue dans le fichier correspondant du **sysfs** avec **pread()**, Elle est ensuite transformée en décimale avec **strtol()**. S'il s'agit du premier ou du deuxième bouton, cela concerne la fréquence et la valeur est soit incrémentée ou décrémentée. Si il s'agit du dernier, la valeur est inversée car il s'agit du mode. La nouvelle valeur est convertie en ASCII avec **sprintf()** et écrite avec **pwrite()** en spécifiant le descripteur de l'attribut. Afin d'indiquer à l'utilisateur que l'appui a été traité, la LED power est mise à l'état **1** pendant **2ms**.

Sur la figure 2, suite à l'appui du bouton S1, la fréquence est bien augmentée. L'appui du bouton S2 entraine bien la diminution de la fréquence. Lorsque la fréquence demandée n'est pas valide, le driver l'indique et ne met pas à jour sa valeur. Le bouton S3 est ensuite utilisé pour passer en mode **AUTOMATIC**. Lors de l'appui du bouton S1 dans ce mode, le driver indique que l'action est interdite en mode **AUTOMATIC**.

```
Jan  1 04:56:38 csel daemon.info MPDaemon[271]: [MPDaemon] S1 button pushed
Jan  1 04:56:38 csel kern.info kernel: [17798.386950] [MPDriver] New blinking frequency : 10Hz
Jan  1 04:56:38 csel daemon.info MPDaemon[271]: [MPDaemon] S1 button pushed
Jan  1 04:56:38 csel kern.info kernel: [17798.662589] [MPDriver] New blinking frequency : 15Hz
Jan  1 04:56:43 csel daemon.info MPDaemon[271]: [MPDaemon] S1 button pushed
Jan  1 04:56:43 csel kern.info kernel: [17803.800228] [MPDriver] New blinking frequency : 20Hz
Jan  1 04:56:45 csel daemon.info MPDaemon[271]: [MPDaemon] S2 button pushed
Jan  1 04:56:45 csel kern.info kernel: [17805.249097] [MPDriver] New blinking frequency : 15Hz
Jan  1 04:56:47 csel daemon.info MPDaemon[271]: [MPDaemon] S2 button pushed
Jan  1 04:56:47 csel kern.info kernel: [17807.639394] [MPDriver] New blinking frequency : 10Hz
Jan  1 04:56:48 csel daemon.info MPDaemon[271]: [MPDaemon] S2 button pushed
Jan  1 04:56:48 csel kern.info kernel: [17808.274816] [MPDriver] New blinking frequency : 5Hz
Jan  1 04:56:48 csel daemon.info MPDaemon[271]: [MPDaemon] S2 button pushed
Jan  1 04:56:48 csel kern.err kernel: [17808.837721] [MPDriver] Impossible to set frequency 0Hz
Jan  1 04:59:06 csel daemon.info MPDaemon[271]: [MPDaemon] S3 button pushed
Jan  1 04:59:06 csel kern.info kernel: [17946.222751] [MPDriver] New mode : 1
Jan  1 04:59:10 csel daemon.info MPDaemon[271]: [MPDaemon] S1 button pushed
Jan  1 04:59:10 csel kern.err kernel: [17950.386967] [MPDriver] Forbidden to manually modify frequency
```

FIGURE 2 – Résultats suite à l'appui de boutons

2.3.3 Message reçu de l'IPC

Lors de la réception de données sur le pipe, elles sont lues avec **pread()**. Elles sont ensuite découpées afin d'identifier la commande et la valeur. Par exemple, dans la chaîne **blinking=10**, **blinking** est la commande et **10** la valeur. La commande permet d'identifier le descripteur de l'attribut. La valeur est convertie en ASCII avec **sprintf()** et écrite avec **pwrite()** en spécifiant le descripteur de l'attribut.

Sur la figure 3, l'envoi de la commande **mode=0** provoque bien le changement en mode **MANUAL**. De même pour la commande **blinking=20** et **blinking=10** qui entraîne le changement de fréquence. Lors du mode **AUTOMATIC**, la modification de la fréquence n'est pas autorisée.

```
Jan  1 05:16:18 csel daemon.info MPDaemon[300]: [MPDaemon] From IPC, command received "mode=0"
Jan  1 05:16:18 csel kern.info kernel: [18978.731930] [MPDriver] New mode : 0
Jan  1 05:16:33 csel daemon.info MPDaemon[300]: [MPDaemon] From IPC, command received "blinking=20"
Jan  1 05:16:33 csel kern.info kernel: [18993.265352] [MPDriver] New blinking frequency : 20Hz
Jan  1 05:16:40 csel daemon.info MPDaemon[300]: [MPDaemon] From IPC, command received "blinking=10"
Jan  1 05:16:40 csel kern.info kernel: [19000.558950] [MPDriver] New blinking frequency : 10Hz
Jan  1 05:16:43 csel daemon.info MPDaemon[300]: [MPDaemon] From IPC, command received "mode=1"
Jan  1 05:16:43 csel kern.info kernel: [19003.265128] [MPDriver] New mode : 1
Jan  1 05:16:47 csel daemon.info MPDaemon[300]: [MPDaemon] From IPC, command received "blinking=50"
Jan  1 05:16:47 csel kern.err kernel: [19007.967310] [MPDriver] Forbidden to manually modify frequency
```

FIGURE 3 – Résultats suite à l'envoi de commandes via IPC

3 Programme utilisateur avec IPC

Le programme utilisateur réalisé est une interface en ligne de commande permettant d'envoyer des commandes au démon.

3.1 Fonctionnement du programme

Le pipe nommé est ouvert en écriture avec **open()** en spécifiant le mode **O_WRONLY**. Il effectue ensuite dans une boucle **while** une lecture du flux d'entrée avec **fgets()**. Les données sont directement écrites dans le pipe avec **write()**. Elles sont en effet parsées par le démon.

Sur la figure 4 figure le programme utilisateur ainsi que les commandes transmises dans la figure 3.

```
Commands:
- blinking=X
- mode=X
- stop
[MPIPC]# mode=0
[MPIPC]# blinking=20
[MPIPC]# blinking=10
[MPIPC]# mode=1
[MPIPC]# blinking=50
[MPIPC]# stop
```

FIGURE 4 – Interface en ligne de commande

L'ensemble des fonctionnalités ont été implémentées et sont fonctionnelles. J'ai rencontré des difficultés pour la création du démon. Le débogage est assez difficile mais j'y suis parvenu grâce à **syslog** et **strace -f** permettant de tracer un processus enfant. J'ai beaucoup apprécié ce travail pratique car il lie à la fois la partie driver et espace utilisateur. Ce mini-projet étant complet, il pourra servir de base pour d'autres projets futurs sous Linux.