
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

Table des matières

1	Introduction	2
2	Architecture logicielle	3
3	Conception du module	4
3.1	Difficultés rencontrées sur le module	7
4	Conception du daemon	8
4.1	Difficultées rencontrées sur le daemon	10
5	Application CLI	11
6	Conclusion	13

Table des figures

2.1	Diagramme de principe de l'application	3
3.1	Fréquences pour la led en mode automatique	4
3.2	Création des fichiers nécessaires dans le sysfs	5
3.3	Fonctions de lecture/écriture des fichiers du sysfs	6
4.1	Protocole défini pour l'IPC	8
4.2	Fonction fork_process	9
4.3	Capture des signaux	9
4.4	Handler des signaux	9
5.1	Application CLI	11
5.2	Cheminement d'une commande	12
6.1	Affichage final des valeurs sur l'écran OLED	13

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 un signal PWM pour piloter 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

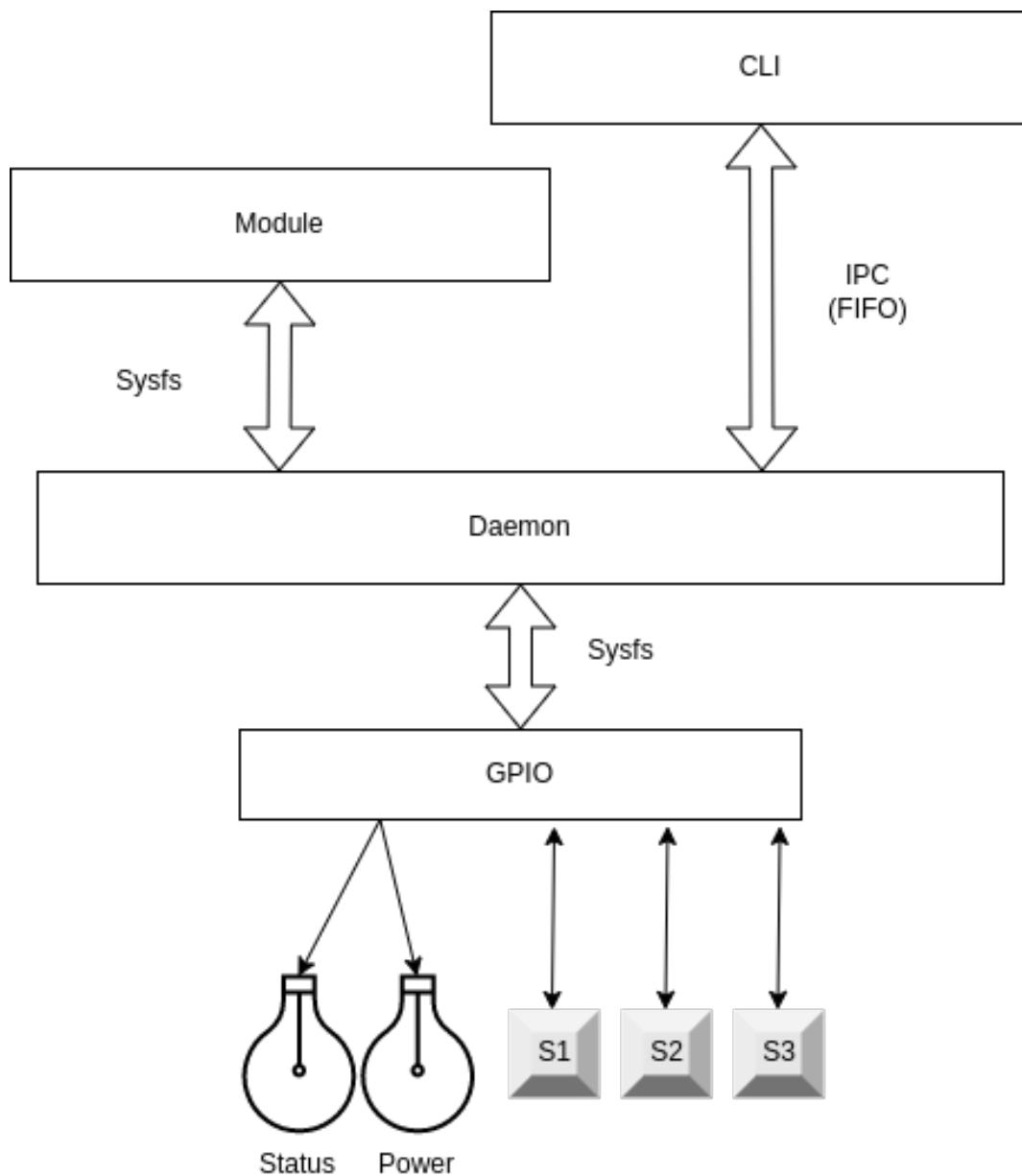


FIGURE 2.1: Diagramme de principe de l'application

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 `/sys/class/<mymodule>/...` 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 ou en mode manuel (si le mode manuel est actif, la valeur '1' est mise dans l'attribut `is_manu`, si la valeur de l'attribut est '0', le mode automatique est activé). Le module implémente un timer (`<linux/timer.h>`) (voir fichiers `timer_controller.h` et `.c`) qui est créé avec une fonction à exécuter lorsque son décompte est terminé. Lorsque le temps est écoulé, cette fonction vérifie le mode actuel grâce à l'attribut `is_manu` et si le mode actuel est manuel, elle 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))
```

Remarque : Jiffies est une variable globale qui compte le nombre de ticks qui se sont écoulés depuis le démarrage de la machine.

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 header file) 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 à gérer les différents composants du module noyau.

Le deuxième fichier `gpio.c` (également accompagné de son header file) 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* (le skeleton gère l'interface du module noyau dans le sysfs), 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 qui stockent la valeur des attributs 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: Crédit 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 fichiers qui nous serviront pour lire et écrire nos valeurs.

Nous avons également dans ce module déclaré quelques 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 fut 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.

Le daemon possède un timer qui lui permet de lire la température du processeur grâce au module *thermal* situé dans */sys/class/thermal/thermal_zone0/temp* ainsi que la fréquence dans notre propre module. Cette lecture s'effectue toutes les 500ms (valeur choisie arbitrairement pas nous-même), et lorsqu'il a pu obtenir ces valeurs, il va se charger de mettre à jour l'affichage sur l'écran OLED.

En plus de ce timer, il gère les boutons 1 à 3 afin de choisir le mode de fonctionnement ainsi que la fréquence de la led lorsque le mode est manuel. Le timer ainsi que les boutons et l'interface IPC sont gérés sous formes d'événements. Le multiplexeur d'entrées/sortie *epoll* permet de réveiller le programme lorsqu'un événement survient et donc d'utiliser efficacement les ressources CPU. Comme demandé dans la consigne, nous avons implémenté une FIFO afin de fournir une interface IPC (Inter Process Communication) qui est également utilisée par *epoll*. Pour la mise en place de cette interface, nous avons défini un simple protocole qui se présente de la manière suivante :

Chaque message est de taille 8 bytes, et se base sur la structure définie ci-dessous :

```
struct cmd
{
    int cmd;
    int value;
}
```

Les 4 premiers bytes sont utilisés pour la commande. Actuellement seules deux commandes sont disponibles, *SET_MANUAL* pour modifier *is_manual* et *SET_FREQ* pour modifier *freq* en mode manuel.

```
#define FIFO_SET_MANUAL 0x14
#define FIFO_SET_FREQ 0x15
```

La valeur 0x14 est donc utilisée pour modifier *is_manu* et la valeur 0x15 pour modifier *freq*. Les 4 bytes suivants sont utilisés pour l'argument (ici *value*). Pour le choix du mode les valeurs sont 0 ou 1, pour la fréquence les valeurs sont entre 0 et *MAX_INT32*.

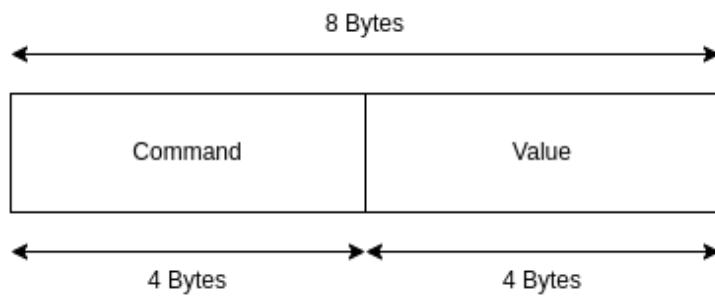


FIGURE 4.1: Protocole défini pour l'IPC

Lorsqu'une commande arrive, *epoll* réveille le daemon et la commande est traitée.

Afin de créer le daemon, nous avons suivi la procédure vue en cours qui consiste à créer un processus enfant puis de terminer le processus parent afin d'avoir un enfant détaché (daemon) qui est ensuite rattaché au processus *init* ceci est fait dans la fonction *fork_process()*.

```
static void fork_process()
{
    pid_t pid = fork();
    switch (pid) {
        case 0: break; // child process has been created
        case -1: syslog (LOG_ERR, "ERROR while forking"); exit (1); break;
        default: exit(0); // exit parent process with success
    }
}
```

FIGURE 4.2: Fonction fork_process

On crée ensuite une nouvelle session pour le processus avec le code suivant :

```
if (setsid() == -1) {
    syslog (LOG_ERR, "ERROR while creating new session");
    exit (1);
}
```

Le processus est maintenant session leader mais ce n'est pas encore notre but, nous allons alors encore une fois utiliser *fork_process()* afin de recréer un enfant et terminer son parent. une fois ceci fait, le processus démon n'est plus session leader et n'a également plus accès au terminal.

Nous avons ensuite déclaré la capture des signaux afin de les ignorer de la manière suivante :

```
struct sigaction act = { .sa_handler = catch_signal, };
sigaction (SIGHUP, &act, NULL); // 1 - hangup
sigaction (SIGINT, &act, NULL); // 2 - terminal interrupt
sigaction (SIGQUIT, &act, NULL); // 3 - terminal quit
sigaction (SIGABRT, &act, NULL); // 6 - abort
sigaction (SIGTERM, &act, NULL); // 15 - termination
sigaction (SIGTSTP, &act, NULL); // 19 - terminal stop signals
```

FIGURE 4.3: Capture des signaux

```
static void catch_signal (int signal)
{
    syslog (LOG_INFO, "signal=%d caught\n", signal);
    signal_caught++;
}
```

FIGURE 4.4: Handler des signaux

Nous mettons ensuite à jour le masque pour la création de fichiers avec la fonction *umask()* puis fermé tous les descripteurs de fichiers (*close*), redirigé *stdin,stdout,stderr* vers */dev/null* avec les fonctions *open* et *dup2*, puis enfin ouvert un fichier de logging avec *openlog()*.

Pour la gestion du daemon, nous avons à nouveau opté pour un principe de controller afin de gérer les différents événements et actions du daemon. Après la création du daemon, on utilise la fonction *start_controller()* qui se charge d'initialiser l'écran OLED, de créer les différents contextes *epoll* pour les boutons, initialiser la FIFO pour l'interface IPC, créer un contrôle *epoll* pour la FIFO et le timer (utilisé pour le rafraîchissement).

Pour la gestion de ces différents événements ainsi que les actions en découlant et par soucis de propreté de code nous avons déclaré quelques header files et fichiers dans lesquels nous avons défini différentes structures de données facilitant la compréhension du code (voir dans */workspace/src/miniprojet/daemon*).

Chaque gpio correspondant aux boutons S1,S2,S3 est configuré dans le fichier *controller.c* à l'aide de la fonction *open_btn()* récupérée d'un ancien TP. Le fichier source de cette fonction se trouvant au chemin suivant : */workspace/src/04_system/silly_led_control.c*. Comme dans cet ancien TP nous avons multiplexé les entrées (*epoll*) afin d'obtenir des événements (bouton, timer, fifo) et pour chaque événement nous avons défini une structure avec un pointeur de fonction permettant d'exécuter le code propre à chaque événement.

4.1 Difficultés rencontrées sur le daemon

Nous n'avons pas rencontré de difficultés particulières pour développer le daemon si ce n'est le debug car en effet un daemon n'offre pas vraiment de possibilités de print à l'écran (uniquement dans le fichier de logging créé mais ce n'est pas très pratique). C'est pourquoi nous avons tout d'abord développé son fonctionnement comme si c'était une application *CLI* classique et une fois assurés du bon fonctionnement de l'application, nous l'avons transformé en deamon avec la méthode vue précédemment.

5. Application CLI

Bien que rendue optionnelle sur la fin du projet, nous avons décidé d'implémenter une application fournissant à l'utilisateur une interface *CLI* pour la gestion du mode et le choix de la fréquence.

Cette application utilise la FIFO pour communiquer avec le daemon, pour ceci un fichier virtuel est créé au chemin suivant `:/workspace/src/miniprojet/ipc`. Ce fichier représente la fifo qui permet de communiquer entre la *CLI* et le daemon.

Afin de clarifier le code, nous avons établis la liste de tous les types de commandes que peut entrer l'utilisateur. Cette liste est un tableau de type *struct usr_command*. Chaque type de commande est composé d'un *string* pour reconnaître l'entrée de l'utilisateur, d'un nombre spécifique d'argumentset d'un pointeur de fonction est également présent dans la structure afin d'indiquer quelle fonction executer pour traiter cette commande. Cette structure est donc la suivante :

```
struct usr_command
{
    char command[CMD_LENGTH];
    int nArgs;
    void (*fct)(int);
};
```

Dans cette structure nous avons un *string* correspondant à la commande entrée, un nombre d'arguments et une fonction permettant de traiter la commande.

C'est donc dans les fonctions de traitement des commandes que les valeurs entrées seront vérifiées puis envoyées au daemon via la fifo. Nous avons aussi imaginé deux commandes spéciales, *help* et *quit* qui permettent comme leur nom l'indique d'obtenir une aide sur la syntaxe ainsi que la liste des commandes disponibles ainsi que de quitter l'application. A chaque fois que l'utilisateur presse sur [ENTER], le programme parcourt la liste de commandes établie précédemment et compare les *string*. Si la commande entrée est identique à une des commandes prédéfinies, la fonction qui y est attachée est exécutée.

```
# ./CLI
CSEL project CLI - Macherel & Raemy
Enter command help for informations
command >> help
This CLI app has three commands possibilities which are :
    1. quit -> Quits the app
    2. manual <val> -> Enable/Disable manual mode (values are 0 or 1)
    3. freq <val> -> Sets LED blinking frequency (must be higher than 0)
command >> manual 1
command >> freq 5
command >> quit
```

FIGURE 5.1: Application CLI

On observe sur la figure 5.1 un exemple d'utilisation de l'application *CLI*, avec un premier appel à la commande *help*, puis une utilisation des commandes *manual* et *freq* afin de passer en mode manuel et modifier la fréquence. Et enfin l'utilisation de la commande *quit* pour terminer l'application.

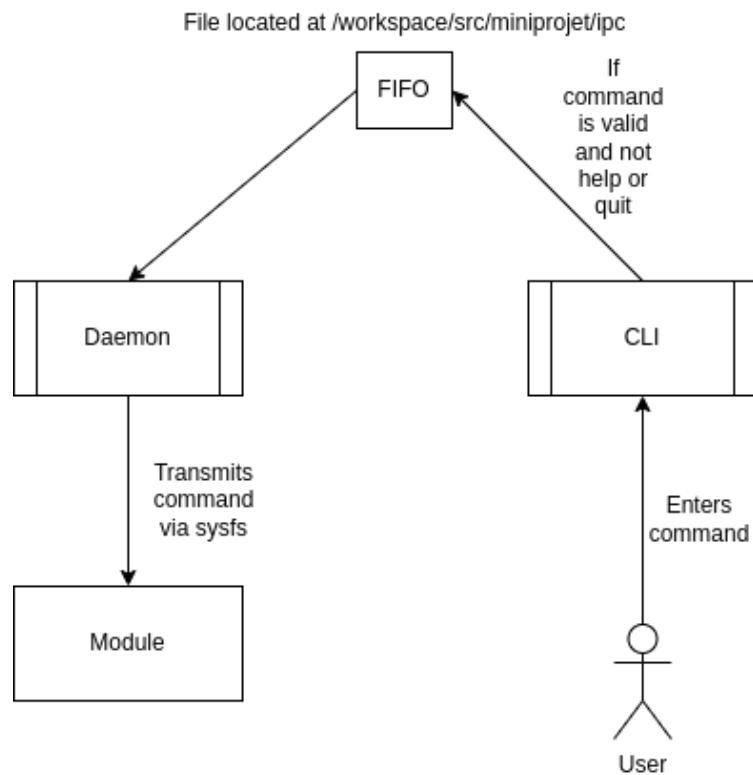


FIGURE 5.2: Cheminement d'une commande

On observe sur la figure 5.2 le cheminement complet d'une commande depuis sa saisie dans l'application *CLI* jusqu'au module noyau.

6. Conclusion

Nous avons beaucoup apprécié ce projet car il nous a vraiment permis d'utiliser toutes les notions vues en cours afin d'en faire un projet complet, fonctionnel et réel. Nous avons réussi à tout implémenter de manière fonctionnelle et sans problèmes et nous avons également essayé de fournir un code le plus propre possible afin de potentiellement pouvoir réutiliser celui-ci dans d'autres applications futures.

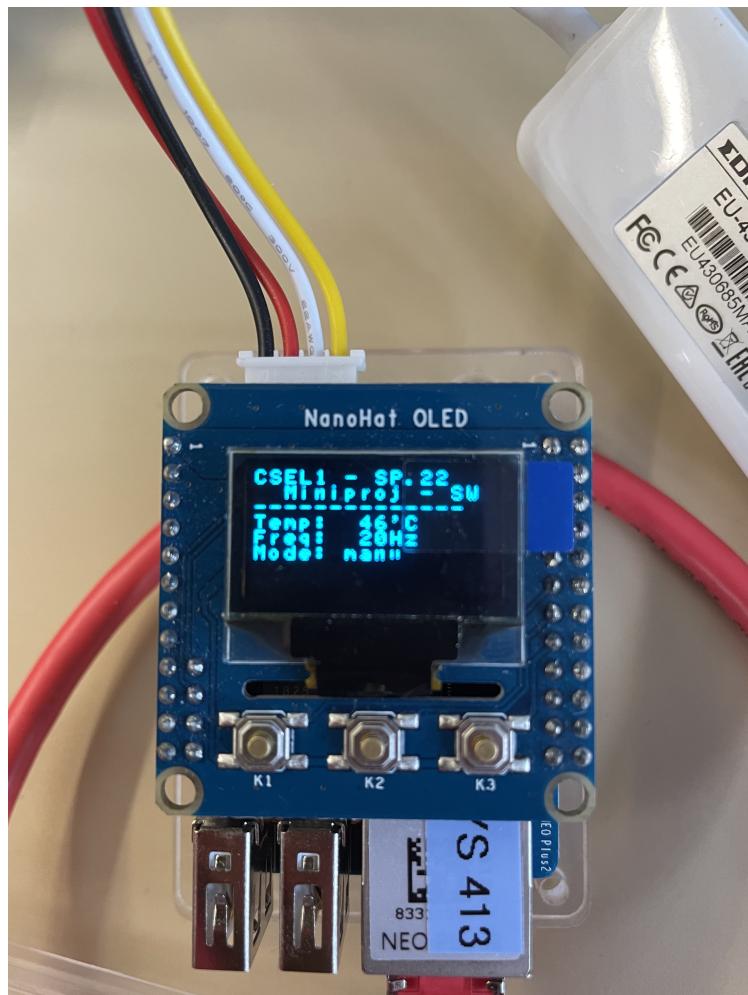


FIGURE 6.1: Affichage final des valeurs sur l'écran OLED