```c
#include <linux/module.h>       /* Needed by all modules */
#include <linux/kernel.h>       /* Needed for KERN_INFO */
#include <linux/init.h>         /* Needed for the macros */
#include <linux/fs.h>           /* Needed for file_operations */
#include <linux/slab.h>         /* Needed for kmalloc */
#include <linux/uaccess.h>      /* copy_(to|from)_user */

#include <linux/string.h>

#include <linux/mm.h>
#include <linux/io.h>

#define MAJOR_NUM       97
#define DEVICE_NAME     "md5_socf"


typedef volatile unsigned int vuint;

// Adresses
#define FPGA_BASE_ADDR          0xFF200000
#define AXI_LIGHT_BASE_ADDR     FPGA_BASE_ADDR
#define AXI_LIGHT_SIZE          4096

// Offsets
#define ADDR_WB_BASE_OFF            0x0
#define ADDR_WB_LAST_OFF            0x3C
#define ENABLE_OFF                  0X100
#define BUSY_MD5_OFF                0x104
#define FOOTPRINT_BASE_OFF          0x200

// Tailles
#define SIZE_MAX_WRITE          32768

#define WB_SIZE                 512
#define NB_BYTE_PARQUET         (512 / 8)

#define FOOTPRINT_SIZE          16



// Adresse virtuelle du bus AXI light
void * addr_base_axi_light;

// Nombre de bit sur 64 bits
unsigned long long total_size;



// Ouverture du device
static int md5_socf_open(struct inode* node, struct file * f)
{

    // Adresse de l'enable
    vuint* addr_enable = addr_base_axi_light+ENABLE_OFF;


    // Si le device est ouvert pour une écriture
    if ( (f->f_flags & O_ACCMODE) == O_WRONLY){

        // Active l'IP MD5
        *addr_enable = 1;
        total_size = 0;
    }


    return 0;
}

// Fermeture du device
static int md5_socf_close(struct inode* node, struct file * f)
```

```
{
    /* Récupération des adresses virtuelles */
    vuint* addr_busy = addr_base_axi_light+BUSY_MD5_OFF;
    vuint* addr_enable = addr_base_axi_light+ENABLE_OFF;
    vuint* addr_wb_base = addr_base_axi_light+ADDR_WB_BASE_OFF;
    vuint* addr_wb_last = addr_base_axi_light+ADDR_WB_LAST_OFF;

    int mod;

    // Si le device est ouvert pour une écriture
    if ( (f->f_flags & O_ACCMODE) == O_WRONLY){

        // Si les données sont alignées, envoie le dernier paquet
        mod = total_size%WB_SIZE;
        if ( mod == 0 ){

            /* Met toutes les données à 0 */
            char tabPadding[64];
            memset(tabPadding, 0, 64);

            // Place la premier valeur à 128
            tabPadding[0] = 128;
            // Ajoute la taille en bit de la donnée
            *((long long *)(tabPadding+56)) = (long long)total_size;

            // Envoie les premiers bytes
            memcpy( (void*)addr_wb_base, (void*) tabPadding, 60);

            // Tant que c'est busy
            while ( (*addr_busy & 0x1) ) {
            };

            // Envoie le dernier mot
            *addr_wb_last = *((unsigned int *)(tabPadding+60));

        }

        // S'assure que le footprint est bien généré
        while ( (*addr_busy & 0x1) ){
        };


        // Désactive l'IP MD5
        *addr_enable = 0;   // Sinon, change de footprint
    }




    return 0;
}

// Lecture sur le device du driver (Ex: cat /dev/votre_device_MD5 )
static ssize_t
md5_socf_read(struct file *filp, char __user *buf,
              size_t count, loff_t *ppos)
{
    unsigned int i,j;
    unsigned char footprint[16];
    char buf_out[34] ;

    /* Adresse du footprint  */
    vuint* addr_footprint = addr_base_axi_light+FOOTPRINT_BASE_OFF;

    if (buf == 0 || count < sizeof(buf_out)) {
        return 0;
    }

    if (*ppos >= sizeof(buf_out)) {
        return 0;
```

```
    }

    // Vérifie que l'écriture dans un buffer user soit OK
    if ( !access_ok( buf, count)){
        printk(KERN_ERR "Erreur access_ok for read..\n");
         return -1;
    }

    // Lecture du footprint md5 (inversement afin de l'afficher comme souhaité)
    for(j=0; j < (FOOTPRINT_SIZE/4); ++j){
            *((unsigned int *)(footprint+(j*4))) = *(addr_footprint+3-j);
    }

    // Transforamtion en Hexa en string pour l'affichage
    for(i=0; i < sizeof(footprint); ++i){
        sprintf(buf_out + i*2, "%02X", footprint[i]);
    }
    sprintf(buf_out+(sizeof(buf_out)-2) , "\n");

    // Ecriture du footprint md5 en hexa dans le buffer user
    if (copy_to_user(buf, buf_out, sizeof(buf_out)) != 0){
        printk(KERN_ERR "Erreur read..\n");
    }

    // Met à jour la taille du buffer
    *ppos = sizeof(buf_out);

    return sizeof(buf_out);
}

// Ecriture sur le device du driver (Ex: cat mon_fichier > /dev/votre_device_MD5 )
// Exemple de padding md5 : https://fthb321.github.io/MD5-Hash/MD5OurVersion2.html
static ssize_t
md5_socf_write(struct file *filp, const char __user *buf,
             size_t count, loff_t *ppos)
{
    /* Récupération des adresses virtuelles */
    vuint* addr_wb_base = addr_base_axi_light+ADDR_WB_BASE_OFF;
    vuint* addr_wb_last = addr_base_axi_light+ADDR_WB_LAST_OFF;
    vuint* addr_busy = addr_base_axi_light+BUSY_MD5_OFF;

    int i;
    int last_ecriture = 0;
    unsigned long ret;
    char *message;
    // Taille en bit
    size_t sizeBit = count * 8;
    size_t mod512 = sizeBit % WB_SIZE ;
    size_t nbBoucle = (sizeBit / WB_SIZE);
    // Permet de savoir le nombre de valeur à ajouter au padding
    int addPad = 448 - mod512;

    /* Met à jour le nombre de bit total */
    total_size += sizeBit;

    /* Dernière lecture (possibilité d'erreur si le nombre de bit est un multiple de 32768) */
    if (sizeBit != SIZE_MAX_WRITE){
        last_ecriture = 1;
    }

    if (count == 0) {
        return 0;
    }

    // Vérifie que la lecture via un buffer user soit OK
    if ( !access_ok( buf, count)){
        printk(KERN_ERR "Erreur access_ok for write..\n");
        return -1;
    }
```

```
    *ppos = 0;

    // Si un paquet supplémentaire doit être ajouté pour le padding
    if (mod512 >= 448){
        nbBoucle++;
    }


    // Alloue le buffer pour récupérer les bytes du fichier
    message = kcalloc(  NB_BYTE_PARQUET , sizeof(char) , GFP_KERNEL);

    /* Pour chaque paquet de 512 bits (64 bytes) */
    for (i=0; i < nbBoucle ; ++i){

        // Lecture du buffer user dans le buffer kernel
        ret = copy_from_user((void*)message, buf+(i*NB_BYTE_PARQUET), NB_BYTE_PARQUET);
        if (ret != 0){
            printk(KERN_ERR "Erreur write..(%ld)\n", ret );
        }

        /* Si c'est le dernier paquet et qu'il faut ajouter le padding */
        if (addPad <= 0) {
            /* Remplis le reste du pauet de 0 */
            addPad = 8+(addPad/8);
            memset( (void*)message+NB_BYTE_PARQUET-addPad, 0, addPad);
            /* Ajoute 128 a la premiere valeur libre */
            *(message+NB_BYTE_PARQUET-addPad) = 128;
        }


        // Envoie les premiers bytes
         memcpy( (void*)addr_wb_base, (void*) message, 60);

        // Tant que c'est busy
         while ( (*addr_busy & 0x1) ) {
         };

        // Envoie le dernier mot
         *addr_wb_last = *((unsigned int *)(message+60));

    }

    /* Si il faut envoyer le dernier paquet */
    if (mod512 != 0 && last_ecriture == 1){

        /* S'il ne reste aucune donnée dans le buffer user */
        if (mod512 >= 448){
            // Remplis le paquet de 0
            memset(message, 0, 64);
        // Sinon
        } else {
            // Lecture du buffer user dans le buffer kernel
            ret = copy_from_user((void*)message, buf+(nbBoucle*NB_BYTE_PARQUET), mod512/8); //
        LINE FEDD TODO
            if (ret != 0){
                printk(KERN_ERR "Erreur write..(%ld)\n", ret );
            }
            /* Remplis le reste des données par des 0 */
            memset(message+(mod512/8), 0, (64-(mod512/8)));
            /* ajoute la valeur de 1 pour le padding */
            message[(mod512/8)] = 128;

        }

        // Place la taille en bit de la donnée dans le dernier paquet à envoyer
        *((long long *)(message+56)) = (long long)sizeBit;


        // Envoie les premiers bytes
        memcpy( (void*)addr_wb_base, (void*) message, 60);
```

```
        // Tant que c'est busy
        while ( (*addr_busy & 0x1) ) {
        };

        // Envoie le dernier mot
        *addr_wb_last = *((unsigned int *)(message+60));

    }

    return count;
}



const static struct
file_operations md5_socf_fops = {
    .owner          = THIS_MODULE,
    .read           = md5_socf_read,
    .write          = md5_socf_write,
    .open           = md5_socf_open,
    .release        = md5_socf_close,
};




static int
__init md5_socf_init(void)
{

    // Enregistrer un numéro majeur pour les dispositifs de caractères.
    register_chrdev(MAJOR_NUM, DEVICE_NAME, &md5_socf_fops);


    // Map l'adresse du bus AXI light
    addr_base_axi_light = ioremap(AXI_LIGHT_BASE_ADDR, AXI_LIGHT_SIZE);

    /* Si le mappage c'est mal passé */
    if (addr_base_axi_light == NULL) {
        pr_err("Failed to map memory!\n");
        /* Retourne (Invalid argument)*/
        return -EINVAL;
    }


    // Test la lecture cote fpga
    if (*((unsigned int*)addr_base_axi_light) != 0xdeadbeef){
        printk(KERN_ERR "Error read FPGA offset 0 (deadbeef)");
        return -EINVAL;
    }


    printk(KERN_INFO "md5_socf ready!\n");
    printk(KERN_INFO "mknod /dev/md5_socf c 97 2\n");
    printk(KERN_INFO "chmod 666 /dev/md5_socf\n");
    return 0;
}

static void
__exit md5_socf_exit(void)
{
    // Démappe l'adresse du bus AXI light
    iounmap(addr_base_axi_light);

    // Désinscrire et détruire le dispositifs de caractères.
    unregister_chrdev(MAJOR_NUM, DEVICE_NAME);

    printk(KERN_INFO "md5_socf bye!\n");
}
```

```
MODULE_AUTHOR("Spinelli Isaia");
MODULE_LICENSE("GPL");

module_init(md5_socf_init);
module_exit(md5_socf_exit);
```