

Systèmes embarqués 2/7

J.-M Friedt

FEMTO-ST/département temps-fréquence

`jmfriedt@femto-st.fr`

transparents à `jmfriedt.free.fr`

6 septembre 2017

Système d'exploitation : pourquoi les pilotes

- Fournir une abstraction du matériel : cacher les fonctions bas niveau pour que le développeur se focalise sur la fonction du périphérique → un point d'entrée avec des appels système (`open`, `read`, `write`, `close`) qui cache l'accès au matériel
- Homogénéiser les interfaces à tous les périphériques (“tout est fichier”)
- Seul le noyau a accès à toutes les ressources de la machine (DMA, interruptions)
- Distribuer les ressources et s'assurer de la cohérence des accès
- Ajout de fonctionnalités au noyau Linux : les **modules**

Mémoire virtuelle/mémoire matérielle

- Espace utilisateur : mmap sur le descripteur de /dev/mem
- Espace noyau : fonction ioremap après avoir réservé une plage d'adresses.

```
#include <linux/io.h>          // ioremap
#define IO_BASE 0xe000a000 // UG585 p.1347
```

et dans la fonction d'initialisation

```
if (request_mem_region(IO_BASE,0x2e4,"GPIO test")==NULL)
    printk(KERN_ALERT "mem request failed");
jmf_gpio=(u32)ioremap(IO_BASE, 0x2e4); // UG585 p.1349
writel(1<<9,jmf_gpio+0x204); // dir. of MI09
```

Pour libérer la ressource :

```
release_mem_region(IO_BASE, 0x2e4);
```

Tester si la plage mémoire requise est disponible (sinon kernel panic)

Accès au matériel depuis l'espace utilisateur

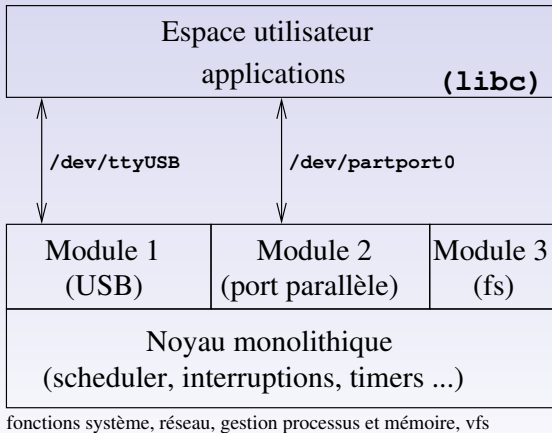
Au travers de /dev

- écriture, lecture ou contrôle (*ioctl*)
- passage au travers du module noyau qui implémente les diverses méthodes
- chaque périphérique est identifié par sa classe (*major number*) et son indice (*minor number*)

```
brw-rw---- 1 root    disk      8,    0 Feb 28 06:21 sda
brw-rw---- 1 root    disk      8,    1 Feb 28 06:21 sda1
brw-rw---- 1 root    disk      8,    2 Feb 28 06:21 sda2
brw-rw---- 1 root    disk      8,    3 Feb 28 06:21 sda3
[...]
crw-rw---- 1 root    dialout   4,  64 Feb 28 07:21 ttyS0
crw-rw---- 1 root    dialout   4,  65 Feb 28 07:21 ttyS1
crw-rw---- 1 root    dialout   4,  66 Feb 28 07:21 ttyS2
crw-rw---- 1 root    dialout   4,  67 Feb 28 07:21 ttyS3
```

En l'absence d'une entrée (par udev) dans /dev : `mknod`

Architecture d'un OS compatible POSIX¹



1. Répertoire /dev :
pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap10.html

Le répertoire /dev

- contient des pseudo-fichiers faisant le lien entre un module noyau et l'espace utilisateur
- chaque *device* est défini par un *major number* et, dans sa classe, un *minor number*
- du côté du noyau, un pilote (*driver*) s'est lié au même major number et fournit des méthodes standard

(appels système)

- open()
- close()
- write()
- read()
- ioctl()

```
static struct file_operations fops =  
{.read:          qadc_read,  
 .open:         qadc_open,  
 .unlocked_ioctl: qadc_ioctl,  
 .release:      qadc_release,  
 // pas de .write sur un ADC !  
};
```

```
static int __init qadc_init (void)  
{...  
 register_chrdev (qadc_major, "ppp", &fops);  
 ...}
```

```
module_init (qadc_init);  
module_exit (qadc_exit);
```

Input/Output Control (ioctl)

Introduction

- mécanisme qui rompt avec la philosophie “tout est fichier”
- passage de la configuration d'un périphérique sous forme “nature de la configuration” et argument “valeur”
- pas de commande standardisée : voir dans le fichier d'entête commun au pilote et programme utilisateur les IOCTL disponibles
- exemple OSS (Open Sound System²) dans

```
include/uapi/linux/soundcard.h des sources du noyau :  
/* Use ioctl(fd, OSS_GETVERSION, &int) to get the version number of  
 * the currently active driver. */  
...  
/* IOCTL commands for /dev/dsp and /dev/audio  
 */  
#define SNDCTL_DSP_RESET                _SIO ( 'P', 0)  
#define SNDCTL_DSP_SYNC                 _SIO ( 'P', 1)  
#define SNDCTL_DSP_SPEED                _SIOWR('P', 2, int)  
#define SNDCTL_DSP_STEREO               _SIOWR('P', 3, int)  
...
```

- Espace utilisateur :
sample_rate = 48000;
if (ioctl (fd, SNDCTL_DSP_SPEED, &sample_rate) == -1)
 { perror ("SNDCTL_DSP_SPEED"); exit (-1); }

2. <http://www.linuxdevcenter.com/pub/a/linux/2007/08/02/an-introduction-to-linux-audio.html>

IOCTL : noyau

```
static long dev_ioctl(struct file*, unsigned int, unsigned →  
    ↪ long);
```

```
static struct file_operations fops=  
{.read=dev_read ,  
 .open=dev_open ,  
 .unlocked_ioctl=dev_ioctl ,  
 .write=dev_write ,  
 .release=dev_rls ,};
```

```
static long dev_ioctl(struct file *f, unsigned int cmd, →  
    ↪ unsigned long arg)  
{static char var[10]; // NE PAS acceder a arg  
 int dummy; // ... qui est en userspace  
 printk(KERN_ALERT "ioctl CMD%d",cmd);  
 switch ( cmd ) // one of the known ioctl  
{case 0: dummy=copy_from_user(var, (char*)arg, 5);  
     printk(KERN_ALERT "ioctl0: %s\n", (char*)var);  
     break;  
     case 1: printk(KERN_ALERT "ioctl1: %s\n", (char*)var);  
     dummy=copy_to_user((char*)arg, var, 5);  
     break;  
     default: printk(KERN_ALERT "unknown ioctl"); break;  
}  
return 0;}
```


IOCTL : utilisateur

```
#include <sys/ioctl.h> /* ioctl */
#define IOCTL_SET_MSG 0
#define IOCTL_GET_MSG 1

main(int argc, char **argv)
{ int file_desc, ret_val;
  char msg[30] = "Message passed by ioctl\n";

  file_desc = open("/dev/jmf", 0);
  msg[4]=0;
  ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);
  ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);
}
```

donne (dmesg)

```
[ 943.551282] Hello ioctl
[ 946.385700] ioctl open
[ 946.385757] ioctl CMD0
[ 946.385758] ioctl0: Mess
[ 946.385762] ioctl CMD1
[ 946.385763] ioctl1: Mess
[ 946.385766] bye
```

Timer noyau – accès direct GPIO

```
#define mio 0

int hello_start()
{int delay = 1;
 unsigned int s;

 if (request_mem_region(IO_BASE1,0x12c,"GPIO cfg")==NULL)
     printk(KERN_ALERT "mem request failed");
 jmf_gpio = (void __iomem*)ioremap(IO_BASE1, 0x4);
 s=readl(jmf_gpio+0x12c); // PAS assignation (crash): mask
 s|=(1<<22); // ... GPIO clock
 writel(s,jmf_gpio+0x12c);
 release_mem_region(IO_BASE1, 0x12c);

 if (request_mem_region(IO_BASE2,0x2E4,"GPIO test")==NULL)
     printk(KERN_ALERT "mem request failed");
 jmf_gpio = (void __iomem*)ioremap(IO_BASE2, 0x2E4);
 writel(1<<mio,jmf_gpio+0x204); // direction
 writel(1<<mio,jmf_gpio+0x208); // enable
 writel(1<<mio,jmf_gpio+0x40); // value
```

User Guide : Appendix B (Register Details) p.785 → gpio @ 0xE000A000
→ DATA_0 en 0x040, DIRM_0 en 0x204 et OEN_0 en 0x208.

Timer du noyau – accès direct GPIO (2)

```
init_timer_on_stack(&exp_timer);  
exp_timer.expires = jiffies + delay * HZ; // HZ specifies →  
    ↪ number of clock ticks generated per second  
exp_timer.data = 0;  
exp_timer.function = do_something;  
add_timer(&exp_timer);  
}  
  
void hello_end() // cleanup_module(void)  
{ release_mem_region(IO_BASE2, 0x2e4);  
  del_timer(&exp_timer);  
}
```

Timer et finir par relâcher les ressources

Timer du noyau – gpiolib

- Mieux : utiliser des bibliothèques existantes du noyau
- Accès aux GPIO : gpiolib
- configuration noyau (make linux-menuconfig)

```
CONFIG_GPIO_XILINX:
```

```
Say yes here to support the Xilinx FPGA GPIO device
```

```
Symbol: GPIO_XILINX [=m]
```

```
Type : tristate
```

```
Prompt: Xilinx GPIO support
```

```
Location:
```

```
-> Device Drivers
```

```
-> GPIO Support (GPIOLIB [=y])
```

```
-> Memory mapped GPIO drivers
```

- nous avons choisi de *ne pas* compiler les modules en statique afin de pouvoir les désactiver en les retirant du noyau (rmmod)
- comme en espace utilisateur, MIOx est appelé par 906+x

Timer du noyau – gpiolib

```
int hello_start(void);  
void hello_end(void);  
  
int hello_start()  
{int jmf_gpio=906+0; // 0 = orange, 7 = rouge (heartbeat)  
  err=gpio_is_valid(jmf_gpio);  
  err=gpio_request_one(jmf_gpio, GPIOF_OUT_INIT_LOW, "→  
    ↪jmf_gpio");  
  // voir dans linux-XXX/linux/gpio.h pour les appels  
  
  init_timer_on_stack(&exp_timer);  
  exp_timer.expires = jiffies + delay * HZ;  
  // HZ = number of clock ticks generated per second  
  exp_timer.data = 0;  
  exp_timer.function = do_something;  
  add_timer(&exp_timer);  
  return 0;  
}  
  
void hello_end() // cleanup_module(void)  
{gpio_free(jmf_gpio); del_timer(&exp_timer);  
}
```

manipuler la broche : `gpio_set_value(jmf_gpio, jmf_stat);`
Exploitation de gpiolib : la licence (GPL) du module importe !

Structure de base

- deux points d'entrée et de sortie :
`module_init (fonction_debut);` et
`module_exit (fonction_fin);`
- initialisation du point de liaison avec l'espace utilisateur
`register_chrdev (90 , "jmf" ,&fops);`
- des affichages au travers de `/var/log/syslog` par
`printk (KERN_ALERT "message formaté");` (*sans virgule!*)
- possibilité de création dynamique de l'entrée dev
`insmod module crée /dev/jmf` par
`struct miscdevice jmfdev;`

```
{jmfdev.name = "jmf"; // /dev/jmf: major = classe misc
  jmfdev.minor = MISC_DYNAMIC_MINOR;
  jmfdev.fops = &fops;
  misc_register(&jmfdev); // creation dynamique /dev/jmf
}
```

qui se conclut par (`rmmod module`) :

```
misc_deregister(&jmfdev);
```

Démonstration

- 1 compilation d'un module simple (`init`, `exit`) sur PC – Makefile faisant appel aux sources du noyau,

```
obj-m +=mon_module.o
```

```
all:
```

```
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

```
car
```

```
$ ls -l /lib/modules/4.12.0-1-amd64/
```

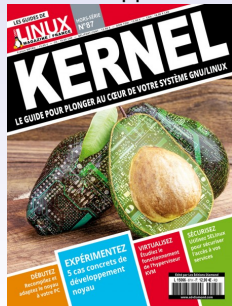
```
    build -> /usr/src/linux-headers-4.12.0-1-amd64
```

- 2 compilation du *même* module pour Redpitaya en faisant appel aux sources du noyau dans buildroot – transfert et exécution sur Zynq

- 3 ajout des fonctions `open`, `ioctl`, `close` – création de l'entrée dans `/dev` par `mknod` et test par un programme en C faisant appel à `ioctl()`

Pour proposer une compilation conditionnelle

x86 v.s ARM : `#ifdef __ARMEL__`³



3. J.-M Friedt, *Modification des appels systèmes du noyau Linux : manipulation de la mémoire du noyau*, GNU/Linux Magazine Hors Série 87 (Nov.-Déc 2016)

Détournement des appels système

```
$ strace mkdir toto  
[...]  
mkdir("toto", 0777) = 0
```

- Un appel système est une méthode fournie par le noyau pour accéder aux ressources gérées par le système (norme POSIX : `open`, `close`, `read`, `write`, `mkdir` ...)
- un programme utilisateur fait appel à un appel système ...
- ... qui correspond à une fonction en espace noyau.
- Une table des appels système fait la correspondance entre les deux.

⇒ manipuler les appels systèmes consiste en

- ① identifier l'emplacement en mémoire de cette table de correspondance
- ② modifier l'adresse de la fonction appelée vers notre fonction ou ...
- ③ ... modifier le contenu de la mémoire contenant l'adresse de destination.

Détournement des appels système

Attaque évidente contre lesquelles les protections sont

- interdiction d'écrire dans les pages mémoire contenant la table des appels système

```
/boot/System.map-4.6.0-1-amd64: ffffffff816001e0 R sys_call_table  
/proc/kallsyms : ffffffff816001e0 R sys_call_table
```

- ne pas exporter les symboles des appels systèmes (portée des fonctions : `static` dans le fichier, `EXPORT_SYMBOL`)

Question : quelles sont les fonctions exportées par le noyau vers les modules ?

```
linux-4.4.2$ grep -r EXPORT_SYMBOL * | grep \<(sys_  
[...]  
fs/open.c:EXPORT_SYMBOL(sys_close);  
kernel/time/time.c:EXPORT_SYMBOL(sys_tz);
```

Seul `sys_close()` est exporté

Détournement des appels système

```
#include <linux/module.h>
#include <linux/syscalls.h>

static unsigned long* cherche_table(void)
{
    unsigned long int offset = PAGE_OFFSET; // debut kernel
    unsigned long *sct;
    printk(KERN_INFO "PAGE_OFFSET=%1x\n", PAGE_OFFSET);
    while (offset < ULLONG_MAX) // recherche toute la mem
    {
        sct = (unsigned long *)offset;
        if (*sct == (unsigned long)&sys_close) // @ sys_close
            return sct; // on a trouve l'@ de debut
        offset += sizeof(void *);
    }
    return NULL;
}
[...]
```

```
[100266.370251] PAGE_OFFSET=ffff880000000000
[100266.433568] table: ffff8800016001f8
[100266.433570] NR close: 3
[100266.433571] NR mkdir: 83 => ffffffff812020d0
```

cohérent (16001f8-24=16001e0 car sys_close est appel 3) avec

```
/boot/System.map-4.6.0-1-amd64: ffffffff816001e0 R sys_call_table
/proc/kallsyms : ffffffff816001e0 R sys_call_table
```

Détournement des appels système

Pointeur de fonction

```
#include <linux/module.h>
#include <linux/syscalls.h>

asmlinkage // passage params par pile
long (*ref_sys_mkdir)(const char __user*,int);

asmlinkage
long mon_mkdir(const char __user *pnam, int mode)
{ printk(KERN_INFO "intercept: %s:%x\n", pnam, mode);
  return 0;
}

static int __init module_start(void)
{ addr_table = cherche_table();
  ref_sys_mkdir=(void *)addr_table[ __NR_mkdir--NR_close ];
  addr_table[ __NR_mkdir--NR_close ]=(unsigned long)→
    ↪mon_mkdir;
  return (0);
}
```

```
$ mkdir toto
[103669.045129] intercept: toto:1ff
```