Interruptions sur Olinuxino A13micro – module noyau Linux

J.-M Friedt, 20 mars 2018

Les interruptions fournissent une méthode fondamentale pour déclencher la gestion d'un évènement qui ne peut attendre. Cependant, l'interruption ne peut être gérée, en présence d'un système d'exploitation, que par le noyau. En effet, l'accès aux ressources matérielles ne doit pas se faire directement depuis l'espace utilisateur, mais être géré par le noyau qui distribuera ensuite l'information aux programmes intéressés par l'information.

1 Par un *character device*

La stratégie que nous allons mettre en œuvre pour appréhender la gestion des interruptions par le noyau Linux en vue de distribuer l'information aux processus au travers de signaux est

1. structure de base d'un module noyau Linux,
2. méthodes d'interaction du module avec l'espace utilisateur (lecture, écriture, `ioctl()`),
3. interaction avec le matériel depuis le module noyau, directement ou en exploitant les fonctionnalités fournies par un autre module noyau,
4. enregistrement d'une interruption et gestionnaire associé,
5. transmission d'un signal à l'espace utilisateur depuis le gestionnaire d'interruption,
6. retarder l'émission du signal depuis une tâche en dehors du gestionnaire d'interruption.

La programmation en espace noyau impose quelques contraintes qui rapprochent des concepts habituellement proposés en système embarqués : parcimonie des ressources occupées, peu de bibliothèques accessibles (on notera en particulier que l'affichage passe par `printk()`), interdiction d'utiliser le calcul sur des nombres à virgule flottante.

Nous avons déjà abordé, dans `http://jmfriedt.free.fr/kernel_module_A13.pdf`, les bases du module noyau. Nous allons approfondir nos connaissances dans les exemples qui suivent en complétant les exemples par les ressources nécessaires à enregistrer une liste de processus intéressés par un évènement matériel détecté au travers d'une interruption. Un domaine potentiel d'application, bien que quelquepeu obsolète sur les PCs modernes, est l'application à l'interruption matérielles accessible depuis le port parallèle [1].

1.1 La méthode `ioctl` – utilisation de `udev`

Bien que toutes les interfaces du monde unix aient pour vocation d'apparaître comme fichier, une méthode additionnelle qui rompt avec ce principe [2] a été implémentée sous la nomenclature de *I/O control* ou `ioctl`. Cette méthode permet, par exemple, de configurer la vitesse d'acquisition d'une carte son, méthode qui ne saurait se configurer par une lecture (des signaux acquis depuis le microphone) ou une lecture (vers le haut parleur).

Par ailleurs, l'exemple ci-dessous modifie la méthode de connexion au pseudo-fichier de `/dev` afin de créer dynamiquement cette entrée (méthode `udev` sur PC notamment – noter que dans ce cas il faut prendre soin de ne **pas** avoir de `/dev/jmf` défini à la main). Il n'est donc plus nécessaire ici d'explicitement créer le nœud au moyen de `mknod` : on vérifiera que l'entrée `/dev/jmf` est créée au chargement du module, et effacée au retrait du module.

```
1  // bien verifier que /dev/jmf N'EXISTE PAS pour ne pas entrer en conflit avec misc dev
2
3  #include <linux/module.h> /* Needed by all modules */
4  #include <linux/kernel.h> /* Needed for KERN_INFO */
5  #include <linux/init.h> /* Needed for the macros */
6  #include <linux/fs.h> // define fops
7  #include <asm/uaccess.h> // get_user, put_user
8  #include <linux/miscdevice.h> // allocation dynamique de /dev
9
10 static int dev_open(struct inode *inod,struct file *fil);
11 static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off);
12 static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off);
13 // static int dev_ioctl(struct inode *inod,struct file *fil, unsigned int cmd, unsigned long arg); // pre-2.6.35
```

1. `http://friedtj.free.fr/modules.ps`
2. cette erreur d'implémentation d'unix a donné lieu au projet Plan9 tel que décrit à `http://plan9.bell-labs.com/sys/doc/net/net.html`

```
14 static long dev_ioctl(struct file *f, unsigned int cmd, unsigned long arg); // actuel
15 static int dev_rls(struct inode *inod,struct file *fil);
16
17 struct miscdevice jmfdev;
18
19 int hello_start(void);
20 void hello_end(void);
21
22 static struct file_operations fops=
23 {.read=dev_read,
24  .open=dev_open,
25  .unlocked_ioctl=dev_ioctl,
26  .write=dev_write,
27  .release=dev_rls,};
28
29 int hello_start() // init_module(void)
30 {// int t=register_chrdev(90,"jmf",&fops); // major = 90
31  // if (t<0) printk(KERN_ALERT "registration failed\n");
32  // else printk(KERN_ALERT "registration success\n");
33  jmfdev.name = "jmf"; // /dev/jmf de major pris dans la classe misc
34  jmfdev.minor = MISC_DYNAMIC_MINOR;
35  jmfdev.fops = &fops;
36  misc_register(&jmfdev); // creation dynamique de l'entree /dev/jmf
37
38  printk(KERN_INFO "Hello ioctl\n");
39  return 0;
40 }
41
42 void hello_end() // cleanup_module(void)
43 {printk(KERN_INFO "Goodbye\n");
44  // unregister_chrdev(90,"jmf");
45  misc_deregister(&jmfdev);
46 }
47
48 static int dev_rls(struct inode *inod,struct file *fil)
49 {printk(KERN_ALERT "bye\n");
50  return 0;
51 }
52
53 static int dev_open(struct inode *inod,struct file *fil)
54 {printk(KERN_ALERT "ioctl open\n");
55  return 0;
56 }
57
58 static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off)
59 {char buf[15]="Hello read\n\0";
60  int dummy,readPos=strlen(buf);
61  printk(KERN_ALERT "read\n");
62  // readPos=0;while (len && (buf[readPos]!=0)) {put_user(buf[readPos],buff++);readPos++;len--;}
63  dummy=copy_to_user(buff,buf,readPos);
64  return readPos;
65 }
66
67 static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off)
68 {int dummy,mylen;
69  char buf[15]="Hello\0";
70  printk(KERN_ALERT "write %d",(int)len);
71  if (len>14) mylen=14; else mylen=len;
72  dummy=copy_from_user(buf,buff,mylen); // for (l=0;l<mylen;l++) get_user(buf[l],buff+l);
73  buf[mylen]=0;
74  printk(KERN_ALERT "%s",buf);
75  return mylen;
76 }
77
78 // static int dev_ioctl(struct inode *inod,struct file *fil, unsigned int cmd, unsigned long arg) // pre-2.6.35
```

```
79 static long dev_ioctl(struct file *f, unsigned int cmd, unsigned long arg)
80 {static char var[10];
81  int dummy;
82  printk(KERN_ALERT "ioctl␣CMD%d",cmd);
83  switch ( cmd )
84  {case 0: dummy=copy_from_user(var,(char*)arg,5);
85          printk(KERN_ALERT "ioctl0:␣%s\n",(char*)var); // NE PAS acceder a arg qui est en userspace
86          break;
87   case 1: printk(KERN_ALERT "ioctl1:␣%s\n",(char*)var); // ... idem
88          dummy=copy_to_user((char*)arg,var,5);
89          break;
90   default:printk(KERN_ALERT "unknown␣ioctl");break;
91  }
92  return 0;
93 }
94
95 module_init(hello_start);
96 module_exit(hello_end);
```

Noter par ailleur le passage des arguments entre l'espace utilisateur et l'espace noyau par les fonctions `get_user()` et `put_user` (pour un échange de scalaires) ou leurs versions étendues `copy_from_user()` et `copy_to_user()` (pour un échange de tableaux) [3]. Du point de vue de l'espace utilisateur, la fonction `ioctl` transfère l'appel au travers du tuyau vers le gestionnaire des entrées/sorties du module noyau.

```
1  #include <fcntl.h> /* open */
2  #include <unistd.h> /* exit */
3  #include <sys/ioctl.h> /* ioctl */
4  #include <stdio.h>
5  #include <stdlib.h>
6
7  #define IOCTL_SET_MSG 0
8  #define IOCTL_GET_MSG 1
9  #define DEVICE_FILE_NAME "/dev/jmf"
10
11 ioctl_set_msg(int file_desc, char *message)
12 { int ret_val;
13   ret_val = ioctl(file_desc, IOCTL_SET_MSG, message);
14   printf("set_msg␣message:␣%s\n", message);
15 }
16
17 ioctl_get_msg(int file_desc,char *message)
18 { int ret_val;
19   ret_val = ioctl(file_desc, IOCTL_GET_MSG, message);
20   printf("get_msg␣message:␣%s\n", message);
21 }
22
23 main(int argc,char **argv)
24 { int file_desc, ret_val;
25   char msg[30] = "Message␣passed␣by␣ioctl\n";
26
27   file_desc = open("/dev/jmf", 0);
28   printf("%d\n",file_desc);
29   if (argc>1) sprintf(msg,argv[1]);
30   msg[4]=0;
31   ioctl_set_msg(file_desc,msg);
32
33   sprintf(msg,"Hello␣World");
34   ioctl_get_msg(file_desc,msg);
35   close(file_desc);
36 }
```

Afin de bien comprendre la séquence des appels système, lancer le programme en espace utilisateur précédé de l'observation des fonctions appelées (`strace`) permet de finement suivre le déroulement de l'exécution :

3. https://www.kernel.org/doc/htmldocs/kernel-hacking/routines-copy.html

```
# ../strace ./ioctl
execve("./ioctl", ["./ioctl"], [/* 12 vars */]) = 0
readlinkat(AT_FDCWD, "/proc/self/exe", "/root/a13-kernel-[ 1683.575556] ioctl open
modules/ioctl", 4096) = 30
mmap2(NULL, 4096, PROT_READ|PROT_WRI[ 1683.583473] ioctl CMD0TE, MAP_PRIVATE|MAP_ANONYMOUS|MAPa
e000
open("/lib/libc.so.1", O_RDONLY)        = 3
fstat(3, {st_[ 1683.591629] ioctl CMD1mode=S_IFREG|0755, st_size=358176, ...}) = 0
mmap2(NULL, 4096, [ 1683.604613] bye
PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_UNINITIALIZED, -1, 0) = 0xb6fad000
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0‘\265\0\0004\0\0\0"..., 4096) = 4096
mmap2(NULL, 409600, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6f3e000
mmap2(0xb6f3e000, 353856, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED, 3, 0) = 0xb6f3e000
mmap2(0xb6f9c000, 4856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x56000) = 0xb6f9c000
mmap2(0xb6f9e000, 15952, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 00
close(3)                                = 0
munmap(0xb6fad000, 4096)                = 0
stat("/lib/ld-uClibc.so.0", {st_mode=S_IFREG|0755, st_size=28570, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_UNINITIALIZED, -1, 0) = 0
set_tls(0xb6fad490, 0xb6fad000, 0xb6fadb38, 0xb6fad490, 0xb6faff70) = 0
mprotect(0xb6f9c000, 4096, PROT_READ)   = 0
mprotect(0xb6faf000, 4096, PROT_READ)   = 0
ioctl(0, TCGETS, {B115200 opost isig icanon echo ...}) = 0
ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
open("/dev/jmf", O_RDWR)                = 3
write(1, "3\n", 23
)                       = 2
ioctl(3, _IOC(0, 0x00, 0x00, 0x00), 0xbec19cd4) = 0
write(1, "set_msg message: Mess\n", 22set_msg message: Mess
) = 22
ioctl(3, FIBMAP, 0xbec19cd4)            = 0
write(1, "get_msg message: Hello World\n", 29get_msg message: Hello World
) = 29# ../strace ./ioctl
execve("./ioctl", ["./ioctl"], [/* 12 vars */]) = 0
readlinkat(AT_FDCWD, "/proc/self/exe", "/root/a13-kernel-[ 1683.575556] ioctl open
modules/ioctl", 4096) = 30
mmap2(NULL, 4096, PROT_READ|PROT_WRI[ 1683.583473] ioctl CMD0TE, MAP_PRIVATE|MAP_ANONYMOUS|MAPa
e000
open("/lib/libc.so.1", O_RDONLY)        = 3
fstat(3, {st_[ 1683.591629] ioctl CMD1mode=S_IFREG|0755, st_size=358176, ...}) = 0
mmap2(NULL, 4096, [ 1683.604613] bye
PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_UNINITIALIZED, -1, 0) = 0xb6fad000
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0(\0\1\0\0\0‘\265\0\0004\0\0\0"..., 4096) = 4096
mmap2(NULL, 409600, PROT_NONE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xb6f3e000
mmap2(0xb6f3e000, 353856, PROT_READ|PROT_EXEC, MAP_PRIVATE|MAP_FIXED, 3, 0) = 0xb6f3e000
mmap2(0xb6f9c000, 4856, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED, 3, 0x56000) = 0xb6f9c000
mmap2(0xb6f9e000, 15952, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 00
close(3)                                = 0
munmap(0xb6fad000, 4096)                = 0
stat("/lib/ld-uClibc.so.0", {st_mode=S_IFREG|0755, st_size=28570, ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS|MAP_UNINITIALIZED, -1, 0) = 0
set_tls(0xb6fad490, 0xb6fad000, 0xb6fadb38, 0xb6fad490, 0xb6faff70) = 0
mprotect(0xb6f9c000, 4096, PROT_READ)   = 0
mprotect(0xb6faf000, 4096, PROT_READ)   = 0
ioctl(0, TCGETS, {B115200 opost isig icanon echo ...}) = 0
ioctl(1, TCGETS, {B115200 opost isig icanon echo ...}) = 0
open("/dev/jmf", O_RDWR)                = 3
write(1, "3\n", 23
)                       = 2
ioctl(3, _IOC(0, 0x00, 0x00, 0x00), 0xbec19cd4) = 0
write(1, "set_msg message: Mess\n", 22set_msg message: Mess
) = 22
ioctl(3, FIBMAP, 0xbec19cd4)            = 0
write(1, "get_msg message: Hello World\n", 29get_msg message: Hello World
) = 29
close(3)                                = 0
exit_group(0)                           = ?
+++ exited with 0 +++
```

1.2 Accès au matériel

Un module noyau se justifie pleinement pour accéder de façon cohérente aux ressources matérielles – contrairement à l'accès depuis l'espace utilisateur qui ne peut se faire qu'en administrateur et en traversant les sécurités proposées par le noyau, seul

le noyau est capable de distribuer de façon cohérente à tous les processus de l'espace utilisateur les ressources. Ainsi, le noyau peut s'assurer qu'un seul processus accède à un instant donné à une ressource qui ne peut être partagée (par exemple une broche de GPIO). Par ailleurs, le noyau ne peut pas être interrompu au cours de ses accès au matériel, contrairement à un processus en espace utilisateur qui peut être prémpté.

```
1  #include <linux/module.h> /* Needed by all modules */
2  #include <linux/kernel.h> /* Needed for KERN_INFO */
3  #include <linux/init.h> /* Needed for the macros */
4  #include <linux/ioport.h> // request_mem
5  #include <linux/io.h> // ioremap
6
7  #define IO_BASE 0x01c20800
8  //#define PWM_BASE 0x01c20e00 // http://linux-sunxi.org/PWM_Controller_Register_Guide != A13 datasheet !
9  #define PWM_BASE 0x01c20c00 // http://linux-sunxi.org/PWM_Controller_Register_Guide != A13 datasheet !
10
11 // IL FAUT PASSER PB2 en PWM : p.293 du manuel, PBconfig0 pour PB2 en PWM=010 bits 8:10
12
13 int jmf_gpio;
14 struct resource *sta; // http://makelinux.com/ldd3/chp-9-sect-4
15
16 int hello_start(void);
17 void hello_end(void);
18
19 int hello_start() // init_module(void)
20 {
21  printk(KERN_INFO "Hello\n");
22  // sta=request_mem_region(IO_BASE,36*('I'-'A'+1),"GPIO test"); // ('I'-'A'+1)=9
23  sta=request_mem_region(PWM_BASE,12,"PWM test"); // ('I'-'A'+1)=9
24  if (sta==NULL)
25     printk(KERN_ALERT "mem request failed");
26  else
27    {// jmf_gpio = (u32)ioremap(IO_BASE, 36*9); // 36 octets/port, et ports A-I
28     // writel(0x10,(void*)(jmf_gpio+36*6+0x04));
29     // writel(0x0200,(void*)(jmf_gpio+36*6+0x10));
30     jmf_gpio = (u32)ioremap(PWM_BASE, 12); // 36 octets/port, et ports A-I
31     writel((1<<4),(void*)(jmf_gpio)); // 24 MHz/120=200 kHz
32     writel((100<<16)+50,(void*)(jmf_gpio+4));
33     printk(KERN_ALERT "PWM success");
34    }
35  return 0;
36 }
37
38 void hello_end() // cleanup_module(void)
39 {printk(KERN_INFO "Goodbye\n");
40  if (sta!=NULL)
41     {// writel(0x0000,(void*)(jmf_gpio+36*6+0x10));
42      release_mem_region(PWM_BASE, 12);
43     }
44 }
45
46 module_init(hello_start);
47 module_exit(hello_end);
```

1.3 Interruption et accès au matériel par un autre module

Jusqu'ici, nous avons implémenté notre propre accès aux ressources bas niveau en écrivant directement dans le registre qui contrôle la LED. Cependant, il peut être judicieux d'utiliser les ressources fournies par un autre module noyau qui implémente déjà de telles fonctionnalités, dans notre cas `gpiolib.c` du répertoire `drivers/gpio/` des sources du noyau Linux. Nous allons exploiter cette fonctionnalité pour accéder au gestionnaire d'interruptions matérielles déclenchées par une transition d'état sur GPIO. Les méthodes fournies par le noyau Linux pour gérer les interruptions sont décrites en détail dans [1, chap. 10].

Pour avoir le droit d'accéder aux ressources fournies par un autre module noyau, notre propre module doit s'annoncer

comme respectant la licence GPL[4]. Pour ce faire, nous devons fournir `MODULE_LICENSE("GPL");`, faute de quoi les méthodes telles que `gpio_is_valid()` ou `gpio_to_irq()` ne seront pas accessibles. Réécrire le gestionnaire d'interruptions du processeur A13 serait une tâche excessivement fastidieuse qui justifie pleinement l'utilisation du système d'exploitation.

La principale subtilité dans ce code consiste à comprendre le mécanisme par lequel les GPIOs sont gérés par `gpiolib`.

1.3.1 Cas du `script.fex`

Afin d'afficher la configuration de la carte et des périphériques proposés au noyau, un fichier `script.bin` est disponible dans la partition de *boot*. Ce script, au format binaire illisible, est converti en script compréhensible par un utilitaire des `sunxi-tools` dont la compilation s'active dans `buildroot` par `Host utilities` → `sunxi-tools`. Le programme chargé de la conversion est `./output/host/usr/bin/bin2fex`.

1.3.2 Cas du `devicetree`

Dans le cas d'une description du matériel par `devicetree`, nous restons sur la représentation du type `broche=port×32+indice` pour accéder à la broche d'indice `indice` sur le port `port`.

Les diverses broches sont décrites dans la datasheet du processeur A13[5] et en particulier quelle interruption externe (EINT) est associée à quelle broche.

La séquence pour retrouver quelle interruption est active est

1. consulter la liste des broches de la carte Olinuxino A13 micro décrite à `https://www.olimex.com/Products/OLinuXino/A13/A13-OLinuXino-MICRO/resources/A13-OLINUXINO-MICRO.pdf`. Nous y constatons (page 26) que la broche 9 du connecteur GPIO2 (pin 6) est sur la broche 103 du processeur,
2. page 10 de `https://linux-sunxi.org/images/e/eb/A13_Datasheet.pdf` nous constatons que la broche 103 est associée à PB2/EINT16. Une autre représentation un peu plus simple mais partielle est proposée à `http://linux-sunxi.org/A13/PIO`.

La définition des plages d'adresses occupées par les GPIOs est fournie par le `devicetree` (décrit dans les sources du noyau Linux par `jrch/arm/boot/dts/sun5i.dtsi`) dans les champs

```
pio: pinctrl@01c20800 {
    reg = <0x01c20800 0x400>;
    interrupts = <28>;
    clocks = <&apb0_gates 5>;
    gpio-controller;
    interrupt-controller;
```

qui est utilisé dans le cas particulier de la carte A13 par la configuration `arch/arm/boot/dts/sun5i-a13.dtsi` :

```
&pio {
   compatible = "allwinner,sun5i-a13-pinctrl";
```

La définition des broches accessibles et de leurs fonctionnalités est décrite dans `drivers/pinctrl/sunxi/pinctrl-sun5i-a13.c` avec un extrait ci-dessous :

```
        SUNXI_PIN(SUNXI_PINCTRL_PIN(G, 9),
                  SUNXI_FUNCTION(0x0, "gpio_in"),
                  SUNXI_FUNCTION(0x1, "gpio_out"),
                  SUNXI_FUNCTION(0x2, "spi1"),          /* CS0 */
                  SUNXI_FUNCTION(0x3, "uart3"),         /* TX */
                  SUNXI_FUNCTION_IRQ(0x6, 9)),          /* EINT9 */
```

qui met en évidence la relation entre le registre de configuration de la *datasheet* (entrée par une configuration à 0x0, sortie par 0x1, modes étendus sinon) et les attributs de la broche, ainsi que la liste des broches actives.

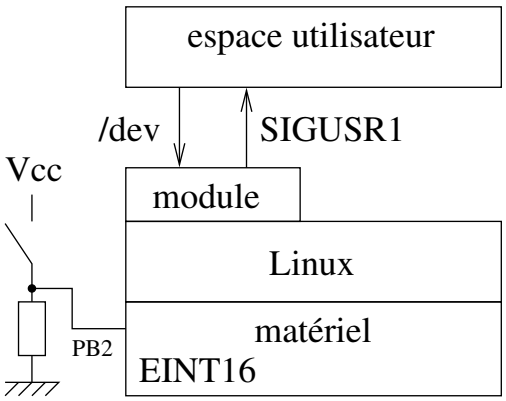
Afin de bien comprendre le mécanisme de gestion des GPIOs par `gpiolib`, nous avons tenté de requérir toutes les broches d'indice compris entre 0 et 7×32 : une erreur de 22 correspond à une requête invalide (EINVAL) tandis qu'une réponse de 16 correspond à une ressource déjà occupée (EBUSY)[6]. Dans le cas du fichier `script.fex`, nous constatons que les 15 premières requêtes aboutissent, en cohérence avec les entrées *gpio* du fichier de configuration.

Exercice : tester les réponses aux requêtes dans le cas de `devicetree`.

4. `http://www.gnu.org/licenses/gpl.html`
5. consulter pour ceci par exemple la *datasheet* du processeur A13 disponible à `https://linux-sunxi.org/images/e/eb/A13_Datasheet.pdf` – page 11
6. les définitions des messages d'erreur se retrouvent dans `/usr/include/asm-generic/errno-base.h`

1.4 Cas du `script.fex` (obsolète)

```
[gpio_para]
gpio_used = 1
gpio_num = 15
gpio_pin_1 = port:PB03<1><default><default><default>
gpio_pin_2 = port:PB04<1><default><default><default>
gpio_pin_3 = port:PB10<1><default><default><default>
gpio_pin_4 = port:PE04<0><default><default><default>
gpio_pin_5 = port:PE05<0><default><default><default>
gpio_pin_6 = port:PE06<0><default><default><default>
gpio_pin_7 = port:PE07<0><default><default><default>
gpio_pin_8 = port:PE08<1><default><default><default>
gpio_pin_9 = port:PE09<1><default><default><default>
gpio_pin_10 = port:PE10<1><default><default><default>
gpio_pin_11 = port:PE11<1><default><default><default>
gpio_pin_12 = port:PG09<1><default><default><default>
gpio_pin_13 = port:PG10<1><default><default><default>
gpio_pin_14 = port:PG11<1><default><default><default>
gpio_pin_15 = port:PB02<1><default><default><default>
```



Exercice : sachant que la LED verte de la carte OlinuxinoA13micro se trouve sur la broche PG9, exploiter la `gpiolib` fournie par le noyau Linux pour la faire clignoter lors du déclenchement d'un évènement par *timer* tel que nous l'avons vu sur le dernier exemple de `http://jmfriedt.free.fr/kernel_module.pdf`. On notera l'intérêt de cette méthode en terme de portabilité : toute plateforme pour laquelle le noyau Linux a été informé de la présence de GPIOs sera compatible avec ce pilote, indépendamment des détails matériels de l'emplacement du GPIO ou des registres permettant de les manipuler. Nous nous appuyons exclusivement sur la disponibilité de `gpiolib` et du travail de portable du noyau Linux à chaque plateforme pour exploiter cette ressource, sans se soucier des détails techniques de la mise en œuvre.

Cette approche est d'autant plus intéressante que nous voulons exploiter des fonctionnalités avancées d'un périphérique, par exemple la détection d'un évènement par interruption. Nous sélectionnons la 15ème entrée pour `script.fex`, ou 34 dans la nomenclature `devicetree`, pour activer PB2 comme source d'interruption pour notre second exemple. La correspondance entre indice de broche et numéro d'interruption est gérée dans les sources du noyau Linux par `drivers/gpio/gpio-sunxi.c` qui mérite à être consulté.

```
1  #include <linux/module.h> /* Needed by all modules */
2  #include <linux/kernel.h> /* Needed for KERN_INFO */
3  #include <linux/init.h> /* Needed for the macros */
4
5  #include <linux/interrupt.h>
6  #include <linux/irq.h>
7  #include <linux/gpio.h>
8
9  static int dummy, irq, jmf_gpio, dev_id;
10
11 void hello_end(void); // cleanup_module(void)
12 int hello_start(void); // cleanup_module(void)
13
14 static irqreturn_t irq_handler(int irq, void *dev_id)
15 {
16        dummy++;
17        printk(KERN_INFO "plip␣%d",dummy);
18    return IRQ_HANDLED; // etait IRQ_NONE
19 }
20
21 int hello_start() // init_module(void)
22 {int err;
23
24  printk(KERN_INFO "Hello\n");
25  /* for (gpio =0;gpio<7*32;gpio++) {
26       err=gpio_is_valid(gpio);
27       err=gpio_request_one(gpio, GPIOF_IN, "jmf_irq");
28       if (err!=-22)
```

```
29         {printk(KERN_ALERT "gpio_request %d=%d\n",gpio,err);
30          irq = gpio_to_irq(gpio);
31          printk(KERN_ALERT "gpio_to_irq=%d\n",irq);
32          gpio_free(gpio); // libere la GPIO pour la prochaine fois
33         }
34    }
35 */
36  jmf_gpio =('B'-'A')*32+2; // PB2 : version devicetree ; 15 en version script.fex
37  err=gpio_is_valid(jmf_gpio);
38  err=gpio_request_one(jmf_gpio, GPIOF_IN, "jmf_irq");
39  if (err!=-22)
40    {printk(KERN_ALERT "gpio_request␣%d=%d\n",jmf_gpio,err);
41     irq = gpio_to_irq(jmf_gpio);
42     printk(KERN_ALERT "gpio_to_irq=%d\n",irq);
43     irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
44     err = request_irq(irq, irq_handler, IRQF_SHARED, "GPIO␣jmf", &dev_id);
45     printk(KERN_ALERT "finished␣IRQ:␣error=%d\n",err);
46     dummy=0;
47    }
48  return 0;
49 }
50
51 void hello_end() // cleanup_module(void)
52 {printk(KERN_INFO "Goodbye\n");
53  free_irq(irq,&dev_id);
54  gpio_free(jmf_gpio); // libere la GPIO pour la prochaine fois
55 }
56
57 module_init(hello_start);
58 module_exit(hello_end);
59
60 MODULE_LICENSE("GPL"); // NECESSAIRE pour exporter les symboles du noyau linux !
```

Dans cet exemple, le champ associé à la requête de contrôle de l'IRQ se retrouve lors du chargement du pilote dans

```
# cat /proc/interrupts
           CPU0
 16:     132057  sun4i_irq  22 Edge      sun4i_timer0
[...]
 46:          5  sunxi_pio_edge  16 Edge      GPIO jmf
```

Par ailleurs, on notera que lors de l'utilisation d'une interruption dont divers gestionnaires (*Interrupt Service Routine* – ISR) sont susceptibles de réagir aux stimuli matériels (drapeau `IRQF_SHARED`), il faut absolument mémoriser la structure de donnée obtenue lors de la requête `request_irq()`. Cette structure sera fournie en retour lors de la libération de la ressource par `free_irq()`, faute de quoi le noyau ne pourra maintenir le décompte des gestionnaires d'interruption et paniquera.

Dans cet exemple, l'interruption réagit aux deux fronts – montant et descendant – des impulsions sur la broche PB2 (`irq_set_irq_t`
On trouvera dans `linux/irq.h` les méthodes alternatives que sont `IRQ_TYPE_EDGE_RISING` et `IRQ_TYPE_EDGE_FALLING`.

1.5 Transfert de l'interruption de l'espace noyau vers l'espace utilisateur

Afin de prévenir un programme en espace utilisateur du déclenchement d'une interruption, nous désirons envoyer un signal depuis l'espace noyau vers un processus utilisateur qui s'enregistre auprès du noyau. La solution simple que nous proposons est d'écrire l'identifiant du processus (*process ID*) depuis l'espace utilisateur vers le noyau au moyen de la méthode `write`. Lorsque l'interruption se déclenche, le gestionnaire d'interruption envoie, en plus d'afficher un message, un signal au processus qui s'est ainsi déclaré.

```
1 #include <linux/module.h> /* Needed by all modules */
2 #include <linux/kernel.h> /* Needed for KERN_INFO */
3 #include <linux/init.h> /* Needed for the macros */
4 #include <linux/fs.h> // define fops
5 #include <asm/uaccess.h> // get_user, put_user
6
```

```
7  #include <linux/interrupt.h>
8  #include <linux/irq.h>
9  #include <linux/gpio.h>
10
11 #include <linux/sched.h>
12
13 int hello_start(void);
14 void hello_end(void);
15
16 static int dummy,gpio,id;
17
18 static int dev_open(struct inode *inod,struct file *fil);
19 static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off);
20 static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off);
21 static int dev_rls(struct inode *inod,struct file *fil);
22
23 static int irq;
24
25 static struct file_operations fops=
26 {.read=dev_read,
27  .open=dev_open,
28  .write=dev_write,
29  .release=dev_rls,};
30
31 int pid = 0;
32
33 static irqreturn_t irq_handler(int irq, void *dev_id)
34 {
35  struct siginfo sinfo;
36  struct task_struct *task;
37
38  memset(&sinfo, 0, sizeof(struct siginfo)); // on cherche PID au cas ou' le process aurait disparu
39  sinfo.si_signo = SIGUSR1; // depuis son enregistrement
40  sinfo.si_code = SI_USER;
41  task = pid_task(find_vpid(pid), PIDTYPE_PID);
42  if (task == NULL) {
43      pr_info("Cannot␣find␣PID␣from␣user␣program\r\n");
44     }
45  else send_sig_info(SIGUSR1, &sinfo, task);
46
47  dummy++;
48  printk(KERN_INFO "plip␣%d",dummy);
49  return IRQ_HANDLED;
50 }
51
52 int hello_start() // init_module(void)
53 {int t=register_chrdev(90,"jmf",&fops); // major = 90
54  int err;
55
56  if (t<0) printk(KERN_ALERT "registration␣failed\n");
57     else printk(KERN_ALERT "registration␣success\n");
58  printk(KERN_INFO "Hello\n");
59
60  gpio =('B'-'A')*32+2; // 15
61  err=gpio_is_valid(gpio);
62  err=gpio_request_one(gpio, GPIOF_IN, "jmf_irq");
63  if (err!=-22)
64     {printk(KERN_ALERT "gpio_request␣%d=%d\n",gpio,err);
65      irq = gpio_to_irq(gpio);
66      printk(KERN_ALERT "gpio_to_irq=%d\n",irq);
67      irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
68      err = request_irq(irq, irq_handler, IRQF_SHARED, "GPIO␣jmf", &id);
69      printk(KERN_ALERT "finished␣IRQ:␣error=%d\n",err);
70      dummy=0;
71     }
```

```
 return t;
}

void hello_end() // cleanup_module(void)
{printk(KERN_INFO "Goodbye\n");
 free_irq(irq,&id);
 gpio_free(gpio); // libere la GPIO pour la prochaine fois
 unregister_chrdev(90,"jmf");
}

static int dev_rls(struct inode *inod,struct file *fil)
{printk(KERN_ALERT "bye\n");
 return 0;
}

static int dev_open(struct inode *inod,struct file *fil)
{printk(KERN_ALERT "open\n");
 return 0;
}

static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off)
{char buf[15]="Hello read\n\0";
 int readPos=0;
 printk(KERN_ALERT "read\n");
 while (len && (buf[readPos]!=0))
   {put_user(buf[readPos],buff++);
    readPos++;
    len--;
   }
 return readPos;
}

static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off)
{int mylen;
 char buf[15];

 printk(KERN_ALERT "write ");
 if (len>14) mylen=14; else mylen=len;
 if (copy_from_user(buf, buff, mylen) == 0)
    sscanf(buf, "%d", &pid);

 printk(KERN_ALERT "PID registered: %d",pid);
 return len;
}

module_init(hello_start);
module_exit(hello_end);

MODULE_LICENSE("GPL"); // NECESSAIRE pour exporter les symboles du noyau linux !
```

En espace utilisateur, le programme qui reçoit les informations est de la forme

```
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <string.h>

void signal_handler(int signum)
{
    if (signum == SIGUSR1)
        printf("J'ai eu un signal\r\n");
    return;
}

int main()
{
```

```
15  FILE *f;
16     signal(SIGUSR1, signal_handler);
17     printf("My␣PID␣is␣%d.\n", getpid());
18     f=fopen("/dev/jmf","w");
19     fprintf(f,"%d\n", getpid());
20     printf("send:␣%d\n", getpid());
21     while (1) {};
22     return 0;
23  }
```

1.6 Minimiser la durée d'exécution du gestionnaire d'interruption

Afin de minimiser le nombre d'opérations effectuées dans le gestionnaire d'interruption, nous exploitons une tâche [1, chap.7] que nous déclenchons lorsqu'une interruption survient. Cette tâche est enregistrée auprès de l'ordonnanceur pour être exécutée lorsque l'occasion se présentera. La lattence est donc potentiellement augmentée, mais nous respectons ici le précept de minimser le temps passé dans le gestionnaire d'interruption et de repousser dans le noyau les points qui ne sont pas critiques de la réaction à l'interruption – ici le transfert vers l'espace utilisteur d'un signal pour l'informer de l'évènement.

```
1  #include <linux/module.h> /* Needed by all modules */
2  #include <linux/kernel.h> /* Needed for KERN_INFO */
3  #include <linux/init.h> /* Needed for the macros */
4  #include <linux/fs.h> // define fops
5  #include <asm/uaccess.h> // get_user, put_user
6
7  #include <linux/interrupt.h>
8  #include <linux/irq.h>
9  #include <linux/gpio.h>
10
11 #include <linux/sched.h>
12 #include <linux/signal.h>
13
14 int hello_start(void);
15 void hello_end(void);
16
17 static int dummy,irq_id,gpio,irq;
18
19 static int dev_open(struct inode *inod,struct file *fil);
20 static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off);
21 static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off);
22 static int dev_rls(struct inode *inod,struct file *fil);
23
24 static struct work_struct irq_work;
25
26 static struct file_operations fops=
27 {.read=dev_read,
28  .open=dev_open,
29  .write=dev_write,
30  .release=dev_rls,};
31
32 int pid = 0;
33
34 static irqreturn_t irq_handler(int irq, void *dev_id)
35 {
36  schedule_work(&irq_work); // on le fera quand on aura le temps
37  dummy++;
38  printk(KERN_INFO "plip␣%d",dummy);
39  return IRQ_HANDLED;
40
41 }
42
43 void do_something(struct work_struct *data)
44 {
45  struct siginfo sinfo;
```

```
46  struct task_struct *task;
47
48  memset(&sinfo, 0, sizeof(struct siginfo)); // on cherche PID au cas ou' le process aurait disparu
49  sinfo.si_signo = SIGUSR1; // depuis son enregistrement
50  sinfo.si_code = SI_USER;
51  task = pid_task(find_vpid(pid), PIDTYPE_PID);
52  if (task == NULL) {
53      pr_info("Cannot␣find␣PID␣from␣user␣program\r\n");
54     }
55  else send_sig_info(SIGUSR1, &sinfo, task);
56
57  printk(KERN_INFO "plop");
58 }
59
60 int hello_start() // init_module(void)
61 {int t=register_chrdev(90,"jmf",&fops); // major = 90
62  int err;
63
64  if (t<0) printk(KERN_ALERT "registration␣failed\n");
65     else printk(KERN_ALERT "registration␣success\n");
66  printk(KERN_INFO "Hello\n");
67
68  gpio =('B'-'A')*32+2; // PB2 en devicetree ; 15 en script.fex
69  err=gpio_is_valid(gpio);
70  err=gpio_request_one(gpio, GPIOF_IN, "jmf_irq");
71  if (err!=-22)
72     {printk(KERN_ALERT "gpio_request␣%d=%d\n",gpio,err);
73      INIT_WORK(&irq_work, do_something);
74      irq = gpio_to_irq(gpio);
75      printk(KERN_ALERT "gpio_to_irq=%d\n",irq);
76      irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
77      err = request_irq(irq, irq_handler, IRQF_SHARED, "GPIO␣jmf", &irq_id);
78      printk(KERN_ALERT "finished␣IRQ:␣error=%d\n",err);
79      dummy=0;
80     }
81  return t;
82 }
83
84 void hello_end() // cleanup_module(void)
85 {printk(KERN_INFO "Goodbye\n");
86  free_irq(irq,&irq_id);
87  gpio_free(gpio); // libere la GPIO pour la prochaine fois
88  unregister_chrdev(90,"jmf");
89 }
90
91 static int dev_rls(struct inode *inod,struct file *fil)
92 {printk(KERN_ALERT "bye\n");
93  return 0;
94 }
95
96 static int dev_open(struct inode *inod,struct file *fil)
97 {printk(KERN_ALERT "open\n");
98  return 0;
99 }
100
101 static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off)
102 {char buf[15]="Hello␣read\n\0";
103  int readPos=0;
104  printk(KERN_ALERT "read\n");
105  while (len && (buf[readPos]!=0))
106    {put_user(buf[readPos],buff++);
107     readPos++;
108     len--;
109    }
110  return readPos;
```

```
111 }
112
113 static ssize_t dev_write(struct file *fil,const char *buff,size_t len,loff_t *off)
114 {int mylen;
115  char buf[15];
116
117  printk(KERN_ALERT "write ");
118  if (len>14) mylen=14; else mylen=len;
119  if (copy_from_user(buf, buff, mylen) == 0)
120     sscanf(buf, "%d", &pid);
121
122  printk(KERN_ALERT "PID registered: %d",pid);
123  return len;
124 }
125
126 module_init(hello_start);
127 module_exit(hello_end);
128
129 MODULE_LICENSE("GPL"); // NECESSAIRE pour exporter les symboles du noyau linux !
```

2 Par un *sysfs*

Le *character device* fournit beaucoup de méthodes qui ne sont pas utiles dans l'exemple qui nous intéresse. La tendance actuelle est à la représentation d'un périphérique matériel tel qu'un GPIO au travers de la structure *sysfs*.

Dans ces conditions, au lieu de générer un signal pour déclencher de façon asynchrone un appel au *handler* depuis l'espace utilisateur, un processus crée un *thread* qui se met à l'écoute d'une entrée de *sysfs*. Lorsque l'interruption se déclenche, le *thread* gère rapidement l'évènement, puis peut informer son père de l'évènement.

Par ailleurs, nous proposons une alternative à la stratégie de communiquer à une liste de processus enregistrés auprès du pilote un signal qui indique le déclenchement de l'interruption, en exploitant cette fois la capacité d'un processus de scinder son activité en plusieurs `threads` indépendants. Dans ce contexte, une lecture bloquante sera réalisée par un `thread`, qui une fois débloqué par le déclenchement de l'interruption, en informera son père.

Les exemples ci-dessous, exploitant les fonctionnalités les plus récentes de l'API du noyau Linux, sont proposés par S. Guinot. En particulier, on notera le soin porté à l'indentation, respectant les consignes de syntaxe du noyau Linux, et les entêtes comportant la license d'utilisation.

Le premier exemple, listing 2, fournit une trame initiale qui illustre la forme du module noyau avec ses méthode d'initialisation et de libération des ressources : ce module se contente de requérir un GPIO selon les méthodes discutées auparavant.

```
1  /*
2   * GPIO simple driver
3   *
4   * This file is licensed under the terms of the GNU General Public
5   * License version 2. This program is licensed "as is" without any
6   * warranty of any kind, whether express or implied.
7   */
8
9  #include <linux/module.h>
10 #include <linux/gpio.h>
11
12 static int gpio = 15;
13 module_param(gpio, int, 0);
14 MODULE_PARM_DESC(gpio, "GPIO number");
15
16 static int __init gpio_simple_init(void)
17 {
18    int err = 0;
19
20    err = gpio_request_one(gpio, GPIOF_IN, "GPIO simple");
21    if (err)
22       return err;
23
```

```
24    pr_info("GPIO␣%d,␣value=%d\n",
25       gpio, gpio_get_value(gpio));
26
27    return 0;
28 }
29
30 static void __exit gpio_simple_exit(void)
31 {
32    gpio_free(gpio);
33 }
34
35 module_init(gpio_simple_init)
36 module_exit(gpio_simple_exit)
37
38 MODULE_DESCRIPTION("GPIO␣simple␣driver");
39 MODULE_LICENSE("GPL");
```

Le second exemple, listing 2, découle du premier en y ajoutant la gestion de l'interruption. Cependant, nous n'avons toujours pas pour le moment de méthode pour communiquer avec le pilote, qui vit sa vie indépendamment de l'espace utilisateur.

```
1  #include <linux/interrupt.h>
2  #include <linux/irq.h>
3
4  int dummy;
5
6  static irqreturn_t gpio_simple_irq_handler(int irq, void *dev_id)
7  {
8     pr_info("IRQ␣%d␣triggered␣on␣GPIO␣%d,␣value=%d\n",
9        irq, gpio, gpio_get_value(gpio));
10
11    return IRQ_NONE;
12 }
13
14 static int __init gpio_simple_init(void)
15 { [...]
16    irq = gpio_to_irq(gpio);
17    irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
18
19    err = request_irq(irq, gpio_simple_irq_handler,
20           IRQF_SHARED, "GPIO␣simple", &dummy);
21    if (err)
22       goto err_free_gpio;
23    [...]
24        return 0;
25 err_free_gpio:
26    gpio_free(gpio);
27
28    return err;
29 }
30
31 static void __exit gpio_simple_exit(void)
32 {
33        free_irq(gpio_to_irq(gpio), &dummy);
34        gpio_free(gpio);
35 }
```

L'absence d'interaction entre le module et l'utilisateur est corrigée dans l'exemple 2 dans lequel une entrée dans `/sys` est dynamiquement ajoutée lors du chargement du module. Ainsi, un `platform_device` initialise un point d'entrée dans `/sys`, ici nommé `gpio-simple`. Ce nom est aussi utilisé lors de la méthode `probe` qui associe une méthode d'initialisation au nom de la plateforme lors de sa création.

```
1  #include <linux/platform_device.h>
2  #include <linux/err.h>
3
4  static struct platform_device *pdev; // a la place de dummy
```

```
5
6  static int gpio_simple_probe(struct platform_device *pdev) // a la place de gpio_simple_init
7  { [...]
8    err = request_irq(irq, gpio_simple_irq_handler,
9            IRQF_SHARED, "GPIO simple", pdev); // a la place de &dummy
10   [...]
11 }
12
13 static int gpio_simple_remove(struct platform_device *pdev)
14 {
15    free_irq(gpio_to_irq(gpio), pdev);
16    gpio_free(gpio);
17
18    return 0;
19 }
20
21 static struct platform_driver gpio_simple_driver = {
22        .probe = gpio_simple_probe,
23        .remove = gpio_simple_remove,
24        .driver = {
25                .name = "gpio-simple",
26        },
27 };
28
29 static int __init gpio_simple_init(void)
30 {
31    int ret;
32
33    ret = platform_driver_register(&gpio_simple_driver);
34        if (ret)
35                return ret;
36
37    pdev = platform_device_register_simple("gpio-simple", 0, NULL, 0);
38    if (IS_ERR(pdev)) {
39       platform_driver_unregister(&gpio_simple_driver);
40       return PTR_ERR(pdev);
41    }
42
43    return 0;
44 }
45
46 static void __exit gpio_simple_exit(void)
47 {
48    platform_device_unregister(pdev);
49    platform_driver_unregister(&gpio_simple_driver);
50 }
```

Ajouter l'affichage d'un message dans la méthode `probe` et constater qu'elle est appelée lors de la création du platform_device. On pourra s'inspirer, sans prétention d'exaustivité, aux affichages de l'exemple 2 pour suivre le cheminement du programme lors des initialisations.

Nous ajoutons ensuite les points de communication avec l'espace utilisateur, qui sera accessible au travers de `/sys/bus/platform/drivers/gpio-simple`

Pour ce faire, une structure de données contenant les points de communication est initialisée au moyen de la macro `DEVICE_ATTR`.

```
1  static ssize_t
2  gpio_simple_show(struct device *dev, struct device_attribute *attr, char *buf)
3  {
4         return sprintf(buf, "%d\n", gpio_get_value(gpio));
5  }
6
7  static DEVICE_ATTR(value, 0444, gpio_simple_show, NULL);
8
9  static int gpio_simple_probe(struct platform_device *pdev)
10 { [... apres request_irq ...]
```

```
11 ]
12    err = device_create_file(&pdev->dev, &dev_attr_value);
13    if (err < 0)
14       goto err_free_irq;
15
16    return 0;
17
18 err_free_irq:
19    free_irq(irq, NULL);
20   [...]
21 }
22
23 static int gpio_simple_remove(struct platform_device *pdev)
24 {
25    device_remove_file(&pdev->dev, &dev_attr_value);
26   [...]
27 }
```

Le code devient plus complexe à lire si on ne prend pas pas soin d'analyser `.../buildroot-a13-olinuxino/output/build/linux-headers-3.12.5/include/linux/device.h` et d'y constater que la macro est définie selon

```
#define DEVICE_ATTR(_name, _mode, _show, _store) \
        struct device_attribute dev_attr_##_name = __ATTR(_name, _mode, _show, _store)
```

Ainsi, la macro crée dans notre cas une structure `dev_attr_value` (`value` étant le nom fourni en premier argument) qui est fournie en paramètre à `device_create_file()`.

Finalement, nous ajoutons la gestion de l'interruption par :

```
1  static struct work_struct work;
2
3  static void gpio_simple_notify(struct work_struct *ws)
4  {
5     pr_info("IRQ␣%d␣triggered␣on␣GPIO␣%d,␣value=%d\n",
6        gpio_to_irq(gpio), gpio, gpio_get_value(gpio));
7  }
8
9  static irqreturn_t gpio_simple_irq_handler(int irq, void *dev_id)
10 {
11    schedule_work(&work);
12    return IRQ_NONE;
13 }
14
15 static int gpio_simple_probe(struct platform_device *pdev)
16 { [... apres gpio_request_one() ...]
17
18    INIT_WORK(&work, gpio_simple_notify);
19   [...]
20 }
21
22 static int gpio_simple_remove(struct platform_device *pdev)
23 {
24    device_remove_file(&pdev->dev, &dev_attr_value);
25    cancel_work_sync(&work);
26   [...]
27 }
```

puis la communication par le point d'entrée `value` dans le système de fichier associé dans `/sys` à notre pilote. Dans cet exemple, un mécanisme de déclenchement de la tâche est initié lorsqu'une interruption se déclenche. Cet évènement servira ensuite à débloquer la lecture d'un processus en utilisateur lorsque l'interruption se déclenchera.

```
1  static void gpio_simple_notify(struct work_struct *ws)
2  {
3     pr_info("Calling␣sysfs_notify␣%p:␣", &pdev->dev.kobj);
```

```
4     sysfs_notify(&pdev->dev.kobj, NULL, "value");
5     kobject_uevent(&pdev->dev.kobj, KOBJ_CHANGE);
6
7     pr_info("IRQ %d triggered on GPIO %d, value=%d\n",
8        gpio_to_irq(gpio), gpio, gpio_get_value(gpio));
9  }
10
11 static int gpio_simple_probe(struct platform_device *pdev)
12 { [...]
13
14    pr_info("Entering probe");
15
16         err=gpio_is_valid(gpio);
17    err = gpio_request_one(gpio, GPIOF_IN, "gpio-simple");
18   [...]
19 }
```

On notera, dans `gpio_simple_init(void)`, que `pdev = platform_device_register_simple("gpio-simple", 0, NULL, 0);` est appelé en premier puisque `pdev` est utilisé par `probe` qui est sollicité par `driver_register`.

Finalement, la conclusion de tout ce développement, dans lequel nous avons ajouté une attente bloquante afin que le processus en espace utilisateur arrête son exécution jusqu'au déclenchement de l'interruption, est résumée dans le programme fourni dans son intégralité :

```
1  /*
2   * GPIO simple driver
3   *
4   * This file is licensed under the terms of the GNU General Public
5   * License version 2. This program is licensed "as is" without any
6   * warranty of any kind, whether express or implied.
7   */
8
9  #include <linux/module.h>
10 #include <linux/gpio.h>
11 #include <linux/interrupt.h>
12 #include <linux/irq.h>
13 #include <linux/platform_device.h>
14 #include <linux/err.h>
15 #include <linux/sched.h>
16
17 static int gpio = 15;
18 module_param(gpio, int, 0);
19 MODULE_PARM_DESC(gpio, "GPIO number");
20
21 static struct platform_device *pdev;
22 static struct work_struct work;
23 static DECLARE_WAIT_QUEUE_HEAD(wq);
24 static atomic_t event = ATOMIC_INIT(0);
25
26 static void gpio_simple_notify(struct work_struct *ws)
27 {
28    sysfs_notify(&pdev->dev.kobj, NULL, "value");
29    kobject_uevent(&pdev->dev.kobj, KOBJ_CHANGE);
30    atomic_set(&event, 1);
31    wake_up(&wq);
32
33    pr_info("IRQ %d triggered on GPIO %d, value=%d\n",
34       gpio_to_irq(gpio), gpio, gpio_get_value(gpio));
35 }
36
37 static irqreturn_t gpio_simple_irq_handler(int irq, void *dev_id)
38 {
39    schedule_work(&work);
40
41    return IRQ_NONE;
42 }
```

```
43
44 static ssize_t
45 gpio_simple_show(struct device *dev, struct device_attribute *attr, char *buf)
46 {
47    int err;
48
49    err = wait_event_interruptible(wq, atomic_read(&event));
50    if (err < 0)
51       return err;
52
53    atomic_set(&event, 0);
54
55        return sprintf(buf, "%d\n", gpio_get_value(gpio));
56 }
57
58 static DEVICE_ATTR(value, 0444, gpio_simple_show, NULL);
59
60 static int gpio_simple_probe(struct platform_device *pdev)
61 {
62    int err = 0;
63    int irq;
64
65    err = gpio_request_one(gpio, GPIOF_IN, "GPIO␣simple");
66    if (err)
67       return err;
68
69    INIT_WORK(&work, gpio_simple_notify);
70
71    irq = gpio_to_irq(gpio);
72    irq_set_irq_type(irq, IRQ_TYPE_EDGE_BOTH);
73
74    err = request_irq(irq, gpio_simple_irq_handler,
75           IRQF_SHARED, "GPIO␣simple", pdev);
76    if (err)
77       goto err_free_gpio;
78
79    err = device_create_file(&pdev->dev, &dev_attr_value);
80    if (err < 0)
81       goto err_free_irq;
82
83    return 0;
84
85 err_free_irq:
86    free_irq(irq, NULL);
87 err_free_gpio:
88    gpio_free(gpio);
89
90    return err;
91 }
92
93 static int gpio_simple_remove(struct platform_device *pdev)
94 {
95    device_remove_file(&pdev->dev, &dev_attr_value);
96    cancel_work_sync(&work);
97    free_irq(gpio_to_irq(gpio), pdev);
98    gpio_free(gpio);
99
100   return 0;
101 }
102
103 static struct platform_driver gpio_simple_driver = {
104        .probe = gpio_simple_probe,
105        .remove = gpio_simple_remove,
106        .driver = {
107               .name = "gpio-simple",
```

```
108         },
109 };
110
111 static int __init gpio_simple_init(void)
112 {
113    int ret;
114
115    pdev = platform_device_register_simple("gpio-simple", 0, NULL, 0);
116    if (IS_ERR(pdev)) {
117       platform_driver_unregister(&gpio_simple_driver);
118       return PTR_ERR(pdev);
119    }
120
121    ret = platform_driver_register(&gpio_simple_driver);
122        if (ret)
123                return ret;
124
125    return 0;
126 }
127
128 static void __exit gpio_simple_exit(void)
129 {
130    platform_device_unregister(pdev);
131    platform_driver_unregister(&gpio_simple_driver);
132 }
133
134 module_init(gpio_simple_init)
135 module_exit(gpio_simple_exit)
136
137 MODULE_DESCRIPTION("GPIO simple driver");
138 MODULE_LICENSE("GPL");
```

A Mise en œuvre sur PC

Les concepts de création du point d'accès à une entrée dans /sys s'applique évidemment sur PC. Un exemple trivial qui ne crée que l'entrée `cat < /sys/bus/platform/drivers/gpio-simple/gpio-simple.0/value*`

```
1  /*
2   sudo insmod gpio-pc.ko
3   sudo cat < /sys/bus/platform/drivers/gpio-simple/gpio-simple.0/value2
4   echo "hello" > /sys/bus/platform/drivers/gpio-simple/gpio-simple.0/value3
5   sudo cat < /sys/bus/platform/drivers/gpio-simple/gpio-simple.0/value2
6   sudo cat < /sys/bus/platform/drivers/gpio-simple/gpio-simple.0/value1
7   sudo rmmod gpio-pc.ko
8  */
9
10 #include <linux/module.h>
11 #include <linux/interrupt.h>
12 #include <linux/irq.h>
13 #include <linux/platform_device.h>
14 #include <linux/err.h>
15
16 char resultat[10]="resultat\0";
17
18 static ssize_t gpio_simple_show1(struct device *dev, struct device_attribute *attr, char *buf)
19 { return sprintf(buf, "Hello World 1\n"); }
20
21 static ssize_t gpio_simple_show2(struct device *dev, struct device_attribute *attr, char *buf)
22 { return sprintf(buf, "%s\n",resultat); }
23
24 static ssize_t gpio_simple_read1(struct device *dev, struct device_attribute *attr, const char *buf, size_t size)
25 { sprintf(resultat,"%s",buf); return(size); }
26
```

```
27 static DEVICE_ATTR(value1, 0444, gpio_simple_show1, NULL); // methode show
28 static DEVICE_ATTR(value2, 0444, gpio_simple_show2, NULL); // methode show
29 static DEVICE_ATTR(value3, 0220, NULL, gpio_simple_read1); // methode store
30
31 static struct platform_device *pdev;
32
33 static int gpio_simple_probe(struct platform_device *pdev)
34 {
35    int err = 0;
36    printk(KERN_INFO "probe\n");
37    err = device_create_file(&pdev->dev, &dev_attr_value1);
38    err = device_create_file(&pdev->dev, &dev_attr_value2);
39    err = device_create_file(&pdev->dev, &dev_attr_value3);
40    return err;
41 }
42
43 static int gpio_simple_remove(struct platform_device *pdev)
44 {
45    device_remove_file(&pdev->dev, &dev_attr_value1);
46    device_remove_file(&pdev->dev, &dev_attr_value2);
47    device_remove_file(&pdev->dev, &dev_attr_value3);
48    return 0;
49 }
50
51 static struct platform_driver gpio_simple_driver = {
52        .probe = gpio_simple_probe,
53        .remove = gpio_simple_remove,
54        .driver = {
55               .name = "gpio-simple",
56        },
57 };
58
59 static int __init gpio_simple_init(void)
60 {
61    int ret;
62    printk(KERN_INFO "hello\n");
63    ret = platform_driver_register(&gpio_simple_driver);
64        if (ret)
65               return ret;
66
67    pdev = platform_device_register_simple("gpio-simple", 0, NULL, 0);
68             // rappeler autant de fois que les peripheriques existent
69    if (IS_ERR(pdev)) {
70       platform_driver_unregister(&gpio_simple_driver);
71       return PTR_ERR(pdev);
72    }
73    return 0;
74 }
75
76 static void __exit gpio_simple_exit(void)
77 {
78    printk(KERN_INFO "bye\n");
79    platform_device_unregister(pdev);
80    platform_driver_unregister(&gpio_simple_driver);
81 }
82
83 module_init(gpio_simple_init)
84 module_exit(gpio_simple_exit)
85
86 MODULE_DESCRIPTION("GPIO simple driver");
87 MODULE_LICENSE("GPL");
```

Cet exemple crée trois fichiers dans le répertoire mentionné plus haut : deux en lecture et un en écriture. L'argument fourni par `value2` est modifié par une écriture dans `value3`. Noter le prototype légèrement différent de la fonction de lecture (répondant à une écriture – dernier argument de `DEVICE_ATTR` nommé méthode *store* fournissant la taille de la chaîne de

caractère transférée).

Tester le bon fonctionnement de ce périphérique par l'exécution des commandes fournies en commentaire en début du programme.

Le `Makefile` associé, pour compiler en natif sur le PC exécutant un noyau 3.16.0-4-686-pae (Debian testing en Mars 2015), ce module noyau est

```
obj-m := gpio-pc.o

all:
   make -C /usr/src/linux-headers-4.8.0-2-amd64 M=$(PWD) modules

clean:
   rm -fv *.o *.ko *.mod.c modules.order Module.symvers
```

B Exemple de lecture bloquante sur `/dev`

Dans cet exemple, une interruption est simulée par un *timer* périodique qui déclenche un évènement qui débloque la lecture en incrémentant un sémaphore.

```
// soit on bloque avec mutex
#define avec_mutex
// *soit* on bloque avec spinlock (l'un OU l'autre)
// #define avec_spin

// http://www.makelinux.net/ldd3/chp-5-sect-3
// http://blackfin.uclinux.org/doku.php?id=linux-kernel:semaphores
#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> // define fops
#include <asm/uaccess.h> // get_user, put_user
#include <linux/sched.h>
#include <linux/semaphore.h>
#ifdef avec_mutex
#include <linux/mutex.h>
#endif
#ifdef avec_spin
#include <linux/spinlock.h>
#endif
#include <linux/miscdevice.h> // allocation dynamique de /dev

struct timer_list exp_timer;
struct semaphore mysem;
#ifdef avec_mutex
struct mutex mymutex;
#endif
#ifdef avec_spin
static DEFINE_SPINLOCK(myspin);
#endif
char buf[15]="Hello␣read␣␣\n\0";
int compteur;

int hello_start(void);
void hello_end(void);

static int dev_open(struct inode *inod,struct file *fil);
static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off);
static int dev_rls(struct inode *inod,struct file *fil);

int is_open=0;

static int dev_rls(struct inode *inod,struct file *fil)
```

```
44 {printk(KERN_ALERT "bye\n");
45  is_open=0;
46  return 0;
47 }
48
49 static int dev_open(struct inode *inod,struct file *fil)
50 {printk(KERN_ALERT "open\n");
51  compteur=0;
52  // sema_init(&mysem, 0); // init the semaphore as empty
53  is_open=1;
54  return 0;
55 }
56
57 static ssize_t dev_read(struct file *fil,char *buff,size_t len,loff_t *off)
58 {int readPos=0;
59  down (&mysem); // JUSTE
60 #ifdef avec_mutex
61  mutex_lock(&mymutex);
62 #endif
63 #ifdef avec_spin
64  spin_lock(&myspin);
65 #endif
66  //down (&mysem); // FAUX : deadlock
67  printk(KERN_ALERT "read\n");
68  while (len && (buf[readPos]!=0))
69    {put_user(buf[readPos],buff++);
70     readPos++;
71     len--;
72    }
73 #ifdef avec_mutex
74  mutex_unlock(&mymutex);
75 #endif
76 #ifdef avec_spin
77  spin_unlock(&myspin);
78 #endif
79  return readPos;
80 }
81
82 static struct file_operations fops=
83 {.owner=THIS_MODULE,
84  .read=dev_read,
85  .open=dev_open,
86  .release=dev_rls,};
87
88 static void do_something(unsigned long data)
89 {printk(KERN_INFO "plop");
90 #ifdef avec_mutex
91  mutex_lock(&mymutex);
92 #endif
93 #ifdef avec_spin
94 // spin_lock(&myspin);
95 #endif
96  buf[11]=compteur+'0';compteur++;if (compteur==10) compteur=0;
97 #ifdef avec_mutex
98  mutex_unlock(&mymutex);
99 #endif
100 #ifdef avec_spin
101 // spin_unlock(&myspin);
102 #endif
103  if (is_open==1) up(&mysem);
104  mod_timer(&exp_timer, jiffies + HZ);
105 }
106
107 struct miscdevice jmfdev;
108
```

```
109 int hello_start() // init_module(void)
110 {int delay = 1;
111  // int t=register_chrdev(90,"jmf",&fops); // major = 90
112  jmfdev.name = "jmf"; // /dev/jmf de major pris dans la classe misc
113  jmfdev.minor = MISC_DYNAMIC_MINOR;
114  jmfdev.fops = &fops;
115  misc_register(&jmfdev); // creation dynamique de l'entree /dev/jmf
116
117  printk(KERN_INFO "Hello\n");
118  sema_init(&mysem, 1); /* init the semaphore as non-empty */
119 #ifdef avec_mutex
120  mutex_init(&mymutex);
121 #endif
122 #ifdef avec_spin
123  spin_lock_init(&myspin);
124 #endif
125 // ce timer simule le declenchement d'une interruption qui debloquera le semaphore
126  init_timer_on_stack(&exp_timer);
127  exp_timer.expires = jiffies + delay * HZ; // HZ specifies number of clock ticks generated per second
128  exp_timer.data = 0;
129  exp_timer.function = do_something;
130  add_timer(&exp_timer);
131
132  // return t;
133  return 0;
134 }
135
136 void hello_end()
137 {printk(KERN_INFO "Goodbye\n");
138  del_timer(&exp_timer);
139  misc_deregister(&jmfdev);
140 }
141
142 module_init(hello_start);
143 module_exit(hello_end);
144 MODULE_LICENSE("GPL");
```

Références

[1] J. Corbet, A. Rubini, & G. Kroah-Hartman, *Linux Device Drivers, 3rd Ed.*, O'Reilly, disponible à `http://lwn.net/Kernel/LDD3/`