Embedded systems 2/7

J.-M Friedt

FEMTO-ST/département temps-fréquence

jmfriedt@femto-st.fr

slides at `jmfriedt.free.fr`

September 9, 2020

Operating system: the need for drivers

- Hardware abstraction: hide low level functions so that the developer can focus on the functionalities provided by the peripheral
  $\rightarrow$ a single entry point providing system calls (`open`, `read`, `write`, `close`) hiding access to hardware
- Homogeneous interface to all peripherals ("Everything is a file")
- Only the kernel can access hardware resources (DMA, interrupts)
- Share resources and make sure only one process can access a given hardware function
- Add functionalities to the Linux kernel: **modules**

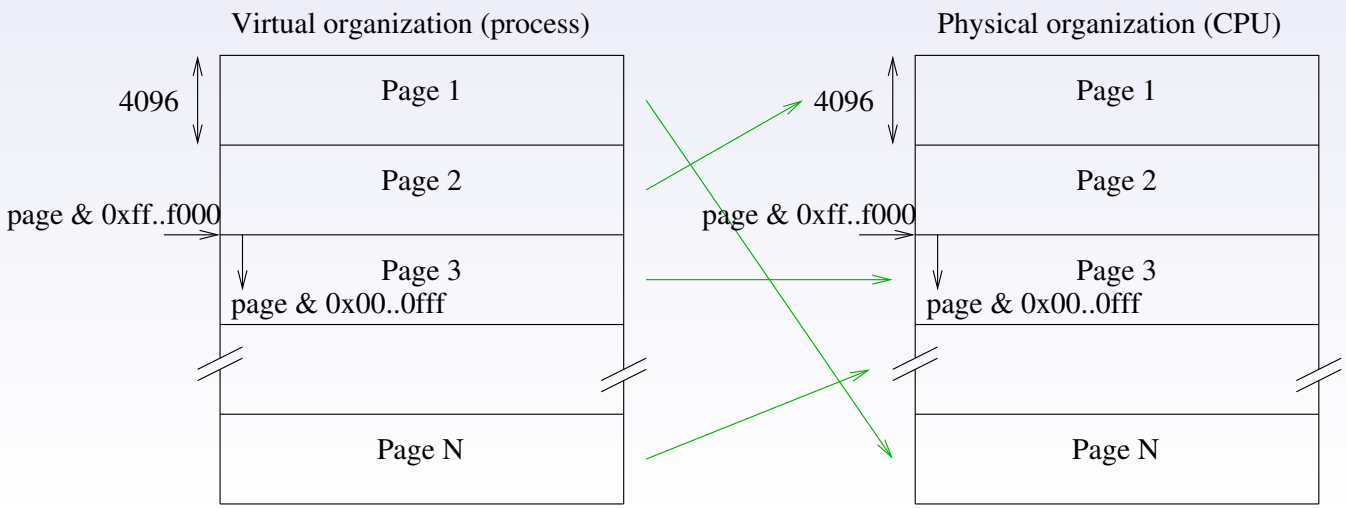
Virtual memory/hardware memory

Hardware memory addressing

- hardware memory: a value on the address bus identifies which peripheral is active
- each peripheral decodes the address bus to detect whether it is the target of a message
- only one peripheral must match a given physical address (otherwise, conflict)

Virtual memory addressing

- each process has its own address space
- memory organization independent of physical constraints
- dynamic loading binaries and associated libraries
- MMU: translates between hardware and virtual memory addresses



Hardware access from userspace

Through /dev/mem

- advantage: not going through kernel abstraction layers (fast)
- drawback: not going through kernel abstraction layers (no handling of concurrent memory access)

```
#include <fcntl.h>
#include <sys/mman.h>
#define MAP_SIZE 4096UL            // MMU page size
#define MAP_MASK (MAP_SIZE-1)      // mask

int main(int argc, char **argv) {
 int fd; void *map_base, *virt_addr;
 unsigned long read_result, writeval;
 off_t target=0x12345678;          // physical @
 fd = open("/dev/mem", O_RDWR | O_SYNC); // MMU access
 map_base=mmap(0, MAP_SIZE, PROT_READ | PROT_WRITE, \
                    MAP_SHARED, fd, target & ~MAP_MASK);
 virt_addr=map_base+(target & MAP_MASK); // virt. @
 read_result=*((unsigned long *)virt_addr); // read mem
 printf("0x%X (%p): 0x%X\n",target,virt_addr,read_result);
 // *((unsigned long *) virt_addr) = writeval; // write
 munmap(map_base, MAP_SIZE); close(fd); return 0;
}
```

The devmem tool

- Fast prototyping when accessing processor registers
- Available in busybox[1]
- Use:

```
Read/write from physical address
        ADDRESS Address to act upon
        WIDTH   Width (8/16/...)
        VALUE   Data to be written
```

[1] $BUILDROOT/output/build/busybox-1.30.1/miscutils/devmem.c

Hardware access from the kernel

- Userspace: `mmap` on the `/dev/mem` pseudo-file
- Kernel space: `ioremap` function after requesting the address range used by the peripheral

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

and in the initialization function

```
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 MIO9
```

To free the resource:

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

Check if the requested memory range is available (otherwise, kernel panic)

Basic kernel module structure

Kernel module = "plugin" requiring strarting and ending functions

```
#include <linux/module.h>     /* Needed by all modules */
#include <linux/kernel.h>     /* Needed for KERN_INFO */
#include <linux/init.h>       /* Needed for the macros */

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

int hello_start()  // init_module(void)
{printk(KERN_INFO "Hello\n");
 return 0;
}

void hello_end() // cleanup_module(void)
{printk(KERN_INFO "Goodbye\n");
}

module_init(hello_start);
module_exit(hello_end);
```

No `printf` (write to `/var/log/messages` with `printk` accessible with `dmesg`), no floating point operation, no file operations.

Kernel module compilation

A kernel module must link to the running kernel:

- requires kernel sources (`$BR/output/build/linux*`) or at least configured headers (`linux-headers-X.Y.Z-amd64` on Debian/GNU Linux)
- dedicated `Makefile` calling the Linux kernel Makefile `modules` method on PC:

```
obj-m += mymod.o
all:
	make -C /lib/modules/$(shell uname -r)/build \
	                M=$(PWD) modules
```

or for cross-compiling, linking to the Buildroot `$BR`

```
obj-m +=mymod.o
all:
	make ARCH=arm CROSS_COMPILE=arm-linux- \
	             -C $(BR)/output/build/linux-XXX     \
	             M=$(PWD) modules
```

Communicating from userspace with the kernel

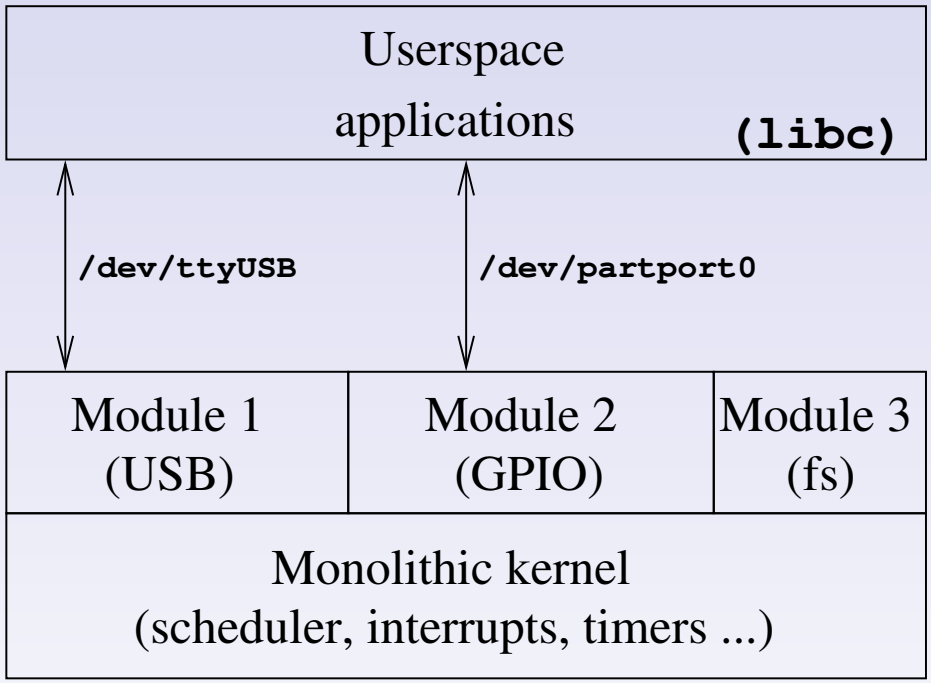
Through `/dev`

- write, read, or control (`ioctl`)
- each kernel module implements its own answer to the system calls
- each peripheral is identified by its class (*major number*) and its index (*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
```

In case an entry is missing (e.g. created by `udev`) in `/dev` : `mknod`

Architecture of a POSIX [2] compliant OS



system fonctions, networking, process scheduling and memory sharing, vfs

[2] /dev directory:
pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap10.html

The /dev directory

- includes pseudo-files linking the kernel with the user space
- each device is defined with a *major number* and, within each class, a *minor number*
- on the kernel side, a *driver* links to the same major number and provides implementations of the system calls

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

```
static struct file_operations fops=
{.read:           qadc_read,
 .open:           qadc_open,
 .unlocked_ioctl: qadc_ioctl,
 .release:        qadc_release,
// no .write on an 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)

- mechanism breaking the "Everything is a file" philosophy
- configures a peripheral by providing "what" index and as an argument the "value"
- no standardized command: the header file common to the driver and the userspace program provides the list of IOCTL calls
- example: OSS (Open Sound System [3]) in `include/uapi/linux/soundcard.h` of the kernel source:

```
/* 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)
...
```

- Userspace:

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

[3] http://www.linuxdevcenter.com/pub/a/linux/2007/08/02/an-introduction-to-linux-audio.html

IOCTL: kernel

```
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]; // DON'T read arg which is located
 int dummy;           // ... in userspace: copy_from_user
 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: userspace

```c
#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, msg);
  ret_val = ioctl(file_desc, IOCTL_GET_MSG, msg);
}
```

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
```

Kernel timer: direct GPIO access

```
#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.

Kernel timer: configuration and callback function

Timer configuration and releasing resources when leaving the module

```
 timer_setup(&exp_timer,do_something,0);  // since 4.14
 exp_timer.expires = jiffies + delay * HZ;
// HZ specifies number of clock ticks generated per second
 add_timer(&exp_timer);
}

void hello_end() // cleanup_module(void)
{release_mem_region(IO_BASE2, 0x2e4);
 del_timer(&exp_timer);
}
```

Callback function:

```
static void do_something(unsigned long data)
{[...]
 mod_timer(&exp_timer, jiffies + HZ); // relaunch timer
}
```

Accessing kernel functions requires compliance with the GPL License [4]:

```
MODULE_LICENSE("GPL");
```

[4] J. Corbet, *Unexporting kallsyms_lookup_name()*, 28/02/2020 @ https://lwn.net/Articles/813350/

Kernel timer: gpiolib

- Rather than accessing the low level registers, use existing kernel functions
- GPIO access: `gpiolib`
- kernel configuration (`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
```

- we have selected *not to* compile static libraries but to use modules in order to keep the ability to deactivate the libraries by removing the module from the kernel (`rmmod`)
- as was seen in userspace, MIOx is refered to by the index 906+x

Kernel timer – gpiolib

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

int hello_start()
{int jmf_gpio=906+0;  // 0 = orange, 7 = red (heartbeat)
 err=gpio_is_valid(jmf_gpio);
 err=gpio_request_one(jmf_gpio, GPIOF_OUT_INIT_LOW, "→
     ↪jmf_gpio");
 // see linux-XXX/linux/gpio.h for available functions

 timer_setup(&exp_timer,do_something,0);  // since 4.14
 exp_timer.expires = jiffies + delay * HZ;
// HZ specifies number of clock ticks generated per second
 add_timer(&exp_timer);
 return 0;
}

void hello_end() // cleanup_module(void)
{gpio_free(jmf_gpio); del_timer(&exp_timer);
}
```

output pin handling: `gpio_set_value(jmf_gpio,jmf_stat);`

Using `gpiolib`: the licence (GPL) of the module matters !

Summary

- Two input and output functions:
  `module_init (beginning_function);` and
  `module_exit (final_function);`
- initialize the link between the module and userspace
  `register_chrdev (90 , "jmf" ,&fops );`
- print message in `/var/log/syslog` by
  `printk (KERN_ALERT "formated message");` (*no coma !*)
- ability to dynamically the dev entry point
  `insmod module` creates `/dev/jmf` by

```
struct miscdevice jmfdev;

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

  which concludes with (`rmmod module`):

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

Lab session

1. compile a simple module (`init`, `exit`) on the PC – `Makefile` linking to the kernel source,

```
obj-m +=mon_module.o
all:
    make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
```

since

```
$ ls -l /lib/modules/4.12.0-1-amd64/
 build -> /usr/src/linux-headers-4.12.0-1-amd64
```

2. compile the *same* module for the Redpitaya by linking with the kernel sources found in `buildroot` – transfer and execute on the Zynq
3. add communication functions`open`, `close` – create the `/dev` entry point with `mknod` and test from shell and a C program
4. add IOCTL function and call from a C program using `ioctl()`

Use conditional compilation x86 v.s ARM : `#ifdef __ARMEL__`[5]

[5] 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 [6] 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.

$\Rightarrow$ 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.

[6] liste des appels systèmes POSIX dans `/usr/include/x86_64-linux-gnu/asm/unistd_64.h`

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=%lx\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
```

Ne fonctionne plus de puis le noyau 4.17.0 qui n'exporte plus `sys_close`, cf `https://github.com/f0rb1dd3n/Reptile`