Embedded Linux

*Device driver development*

Sébastien Bilavarn
Ch1 – Introduction to Linux
Ch2 – Linux kernel overview
Ch3 – Linux for Embedded Systems
Ch4 – Embedded Linux distributions
Ch5 – Case study: Xilinx PowerPC Linux
Ch5 bis – Case study: Xilinx Zynq-7000 Linux
Ch6 – Device driver development
Device driver development

- Device drivers
  - User / device interaction
  - Example 1: basic module
  - Example 2: character driver
  - Example 3: « ioctl » method
  - Example 4: the « mmap » method
Linux device driver

- Definition
  - A device driver is a small piece of software which describes to the operating system how to control and communicate with a piece of hardware (peripheral).
    - A driver is often made up of detailed considerations about synchronization and bit level operations.
    - Assembly level (or low level C) is often more suited for their development.
    - A driver is associated to a specific hardware device, and is fully dependent of an operating system.

- Usefulness
  - A driver provides an abstraction of a device and its operating details, it makes the use of a device easier for a system or a user.

- Linux possible implementations
  - Fully part of the kernel
  - External (loadable) modules
Kernel space – user space

- **Kernel space**
  - Linux manages all computer hardware resources and provides common services for application/user programs.
  - Runs in a privileged mode (kernel space) where it has all permission.
  - Misoperations in *kernel space* can cause important damages to the system.

- **User space**
  - User programs don’t have full privileges in order to protect the system (they can’t access hardware/peripheral registers).
  - They can still interact with the hardware indirectly by using of specific functions called *device drivers*. 
Linux device driver

Kernel space – user space

- **Various layers**
  - User space: applications, system components and libraries.
  - Kernel space: system calls, kernel parts, *device drivers*.

<table>
<thead>
<tr>
<th>User mode</th>
<th>User applications</th>
<th>System daemons: systemd, runit, logind, network, soundd, ...</th>
<th>Windowing system: X11, Wayland, Mir, SurfaceFlinger (Android)</th>
<th>Other libraries: GTK+, Qt, EFL, SDL, SFML, FLTK, GNUstep, etc.</th>
<th>Graphics: Mesa, AMD Catalyst, ...</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>open(), exec(), brk(), socket(), fopen(), malloc(), ... (up to 2000)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>C standard library: subroutines</td>
<td>glibc aims to be POSIX/SUS-compatible. uClibc targets embedded systems, bionic written for Android, etc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Kernel mode</td>
<td>Linux kernel</td>
<td>stat. splice, dup, read, open, ioctl, write, mmap, close, exit, etc. (about 300 system calls)</td>
<td>The Linux kernel System Call Interface (SCI, aims to be POSIX/SUS-compatible)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Process scheduling subsystem</td>
<td>IPC subsystem</td>
<td>Memory management subsystem</td>
<td>Virtual files subsystem</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Other components: ALSA, DRI, evdev, LVM, device mapper, Linux Network Scheduler, Netfilter</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Linux Security Modules: SELinux, TOMOYO, AppArmor, Smack</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Hardware (CPU, main memory, data storage devices, etc.)</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Kernel space – user space

- **Various layers**
  - User space: applications, system components and libraries.
  - Kernel space: system calls, kernel parts, *device drivers*.
Differents types of drivers

- Character device driver
  - Intended for most current devices where exchanging data is based on stream of bytes (example: RS-232 serial communication).

- Block device driver
  - Intended for storage devices that has to move large data blocks (hard disk drive, CDROM, DVD, memory disk, etc.)

- Network device driver
  - Intended for network controllers (example: Ethernet), and protocol stacks.

- Other drivers
  - There are other types of Linux drivers, mostly dedicated to buses (e.g. PCI) or specific APIs (USB, video V4L, V4L2)
Device driver development

- Device drivers
  - User / device interaction
  - Example 1: basic module
  - Example 2: character driver
  - Example 3: « ioctl » method
  - Example 4: the « mmap » method
User / device interaction

- **User space / kernel space** interaction
  - In a Linux system, devices are regarded as files by a user / program.
  - A specific *device file* is used:
    - `/dev/mydevice`
  - Access a device from *user space*
    - File access system calls are used to exchange data through the *device file*.
    - `open`, `close`, `read`, `write`
  - Corresponding operations in *kernel space*
    - When a user/program accesses a device through the *device file*, the driver executes a kernel mode function associated to the system call (`open`, `close`, `read`, `write`, etc.).
    - These functions (and others) are described in the device driver. A programmer defines exactly what the driver does in these functions.
User / device interaction

User / kernel driver functions correspondance

<table>
<thead>
<tr>
<th>Actions</th>
<th>User functions</th>
<th>Kernel functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load module</td>
<td>insmod</td>
<td>module_init()</td>
</tr>
<tr>
<td>Open device</td>
<td>open</td>
<td>file_operations: open</td>
</tr>
<tr>
<td>Read device</td>
<td>read</td>
<td>file_operations: read</td>
</tr>
<tr>
<td>Write device</td>
<td>write</td>
<td>file_operations: write</td>
</tr>
<tr>
<td>Close device</td>
<td>close</td>
<td>file_operations: release</td>
</tr>
<tr>
<td>Remove module</td>
<td>rmmod</td>
<td>module_exit()</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

System call

Corresponding function executed in kernel space
Device driver development

- Device drivers
- User / device interaction
  - Example 1: basic module
  - Example 2: character driver
  - Example 3: « ioctl » method
  - Example 4: the « mmap » method
Example 1: basic module

- A module is C code, but not standard C code
- A module does not have a `main`
- A module is installed for the duration of the interaction with a device, removed when interaction is finished
  - `insmod / rmmod`
- Compilation is different than standard C code
  - `Makefile`
- Produces a `.ko` file (kernel object)
- Full kernel headers are mandatory to produce a module that will effectively run on the Linux distribution
The « Hello world » driver

Compilation

- Makefile:

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
MODULE_LICENSE("Dual BSD/GPL");

int hello_init(void);
void hello_exit(void);

module_init(hello_init);
module_exit(hello_exit);
```

- Command

```
$ make
```

- Full kernel headers are mandatory to produce a working module.

- Generates a hello.ko module in current directory ($PWD)

```
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>

MODULE_LICENSE("Dual BSD/GPL");

int hello_init(void);    
void hello_exit(void);

module_init(hello_init);
module_exit(hello_exit);

int hello_init(void) {
    printk("<1> Hello world!
");
    return 0;
}

void hello_exit(void) {
    printk("<1> Bye, cruel world
");
}
```
Example 1: basic module

- Installing and removing modules (*kernel space*)
  - *module_init/module_exit*
    - These functions are programmed by the user.
    - They are in charge of usual operations associated to the device (reset memory allocation, initialisation, interrupts, etc.)
    - Correspond to the following *user space commands*: `insmod` and `rmmod`
    - `insmod` and `rmmod` will result in practice to the call of *module_init* and *module_exit* associated functions
    - The effective install / remove functions of the module are described in the source code of the module: `hello_init` and `hello_exit`.
    - They are provided as function parameters to *module_init* and *module_exit*. 
Example 1: basic module

- Kernel messages
  - `Printk("<1> Hello world !! ");`
    - Similar to user space function `printf`, but dedicated to kernel space.
    - Linux kernel produces diagnostic messages in the message buffer of the kernel. These messages are accessible using a `dmesg` command.
    - `printk` displays a string using the message buffer of the kernel.
    - Symbol `<1>` is the message priority.

- Execution
  - Installing / loading a module (root)
    - `$ insmod hello.ko`
  - List installed modules
    - `$ lsmod`
  - Removing a module
    - `$ rmmod hello.ko`
Device driver development

- Device drivers
  - User / device interaction
  - Example 1: basic module
  - Example 2: character driver
  - Example 3: « ioctl » method
  - Example 4: the « mmap » method
Example 2: character driver

- The « memory » driver example
Example 2: character driver

- Strictly speaking, not a real device driver
  - Does not provide the full management of a device
  - This module is just used to illustrate the communication of a stream of bytes between *user space* and *kernel space*.

- Mehanisms for data exchange *user space* ↔ *kernel space*
  - Use of a device file (associated to the module)
    - The user accesses the device (through the device file) from *user space* using system calls:
      - open/close
      - read/write
    - The device (driver) accesses the device file from *kernel space* using:
      - file_operations: open / file_operations:release
      - file_operations: read / file_operations:write
Example 2: character driver

/* Necessary includes for device drivers */
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h> /* printk() */
#include <linux/slab.h> /* kmalloc() */
#include <linux/fs.h> /* everything... */
#include <linux/errno.h> /* error codes */
#include <linux/types.h> /* size_t */
#include <linux/proc_fs.h>
#include <linux/fcntl.h> /* O_ACCMODE */
#include <asm/uaccess.h> /* copy_from_user, copy_to_user */
#include "mydevice.h"

MODULE_LICENSE("Dual BSD/GPL");

/* Declaration of driver functions */
int mydevice_open(struct inode *inode, struct file *filp);
int mydevice_release(struct inode *inode, struct file *filp);
ssize_t mydevice_read(struct file *filp, char *buf, size_t count, loff_t *f_pos);
ssize_t mydevice_write(struct file *filp, char *buf, size_t count, loff_t *f_pos);

void mydevice_exit(void);
int mydevice_init(void);

/* Structure that declares the usual file access functions */
struct file_operations mydevice_fops = {
  .read = mydevice_read,
  .write = mydevice_write,
  .open = mydevice_open,
  .release = mydevice_release
};

/* Declaration of the init and exit functions */
module_init(mydevice_init);
module_exit(mydevice_exit);

/* Global variables of the driver */
/* Buffer to store data */
char *mydevice_buffer;

...
Example 2: character driver

Access to the device

- Regarded as a file
- Accessed from user space using file management system calls:
  - open, close, read, write

- Corresponding driver functions
  - mydevice_open,
  - mydevice_release,
  - mydevice_read,
  - mydevice_write

- The file_operations structure defines the list of functions (operations) available for the driver

```c
struct file_operations mydevice_fops = {
    read: mydevice_read,
    write: mydevice_write,
    open: mydevice_open,
    release: mydevice_release
};
```
Example 2: character driver

Data exchange using a character driver

Use of device file

- All device files are located in /dev
- To use the device file from the driver, a specific mechanism based on a major number and a minor number is used. These two numbers associate the device file and device driver.
- The device file is created with a mknod command
  
  ```
  $ mknod /dev/mydevice c 60 0
  ```

- c specifies a character device where 60 is the major number, 0 is the minor number.
  - The major number identifies the type of device (list all devices: cat /proc/devices)
  - The minor number identifies the instance of the driver. This allows to handle several identical devices with the same driver (same major number).
- Within the driver, function register_chrdev is used to associate the device file /dev/mydevice with the device driver.
Example 2: character driver

Driver initialisation
- **mydevice_init**
  - Declaration of driver variables
    - register_chrdev
    - Parameters
      - Major number
      - Driver name
    - **file_operations** structure
  - Memory allocation
    - kmalloc
    - Here, MAX_BUFFER_SIZE bytes are allocated for communicating messages with the driver
    - **GFP_KERNEL**: flag get free pages

```c
int mydevice_init(void) {
    int result;

    /* Registering device */
    result = register_chrdev(MYDEVICE_MAJOR, "mydevice", &mydevice_fops);
    if (result < 0) {
        printk("<1>mydevice: cannot obtain major number %d\n", MYDEVICE_MAJOR);
        return result;
    }

    /* Allocating memory for the buffer */
    mydevice_buffer = kmalloc(MAX_BUFFER_SIZE, GFP_KERNEL);
    if (!mydevice_buffer) {
        result = -ENOMEM; goto fail;
    }
    memset(mydevice_buffer, 0, MAX_BUFFER_SIZE);
    printk("<1>Inserting mydevice module\n");
    return 0;

fail:
    memory_exit();
    return result;
}
```
Example 2: character driver

Driver elimination

- mydevice_exit
  - Implements mechanisms to keep the kernel « clean »
    - unregister_chrdev
      - Frees the major number for the kernel
    - parameters
      - Driver name
    - file_operations structure
  - kfree
    - Frees the memory buffer allocated for communicating with the driver

```c
void mydevice_exit(void) {
    /* Freeing the major number */
    unregister_chrdev(MYDEVICE_MAJOR, "mydevice");

    /* Freeing buffer memory */
    if (mydevice_buffer) {
        kfree(mydevice_buffer);
    }

    printk("<1>Removing mydevice module\n");
}
```
Example 2: character driver

Opening the driver

- **mydevice_open**

  - Driver function opening the device file in *kernel space*
    - In general, different initialisations are needed (variables, reset).
    - Field `open` of structure `file_operations`
    - Associated to the function `mydevice_open`
    - Corresponds to the `open` system call from user space.
    - Parameters:
      - `struct inode *inode`: kernel structure, not used by programmer.
      - `struct file *filp`: corresponds to `mydevice_fops` operations of the driver.

```c
int mydevice_open(struct inode *inode, struct file *filp) {
    /* Success */
    return 0;
}
```
Example 2: character driver

Closing the driver

- mydevice_release

  Driver function closing the device file in *kernel space*
  
  - In general, frees memory and variables reserved by the device opening function.
  - Field *release* of structure *file_operations*
  - Associated to the function mydevice_release
  - Corresponds to the *close* system call from user space.
  - Parameters:
    
    ```c
    struct inode *inode: kernel structure, not used by programmer
    struct file *filp: corresponds to the mydevice_fops operations of the driver.
    ```

```c
int mydevice_release(struct inode *inode, struct file *filp) {
    /* Success */
    return 0;
}
```
Example 2: character driver

Reading data from the driver

- `mydevice_read`
  - Field `read` of structure `file_operations`
  - Associated to the function `mydevice_read`
  - Parameters
    - `struct file *filp`: `mydevice_fops` of the driver
    - `char *buf`: user space buffer where data are copied.
    - `size_t count`: number of bytes to read.
    - `loff_t *f_pos`: pointer to the source data.
  - return: the number of bytes read.

```c
ssize_t mydevice_read(struct file *filp, char *buf, size_t count, loff_t *f_pos) {
    const int msg_length = strlen(mydevice_buffer); /* Transfering data to user space */
    copy_to_user(buf, mydevice_buffer, msg_length);
    return count;
}
```

Example

- Function `mydevice_read` copies `msg_length` bytes from `mydevice_buffer (kernel space)` to `buf (user space)` using function `copy_to_user`
Example 2: character driver

Reading data from the driver

- **copy_to_user**
  - Function `copy_to_user` is used to copy a block of data to **user space**.
  - **Parameters**
    - `to`: **user space** destination buffer.
    - `from`: kernel space source buffer.
    - `count`: number of bytes to copy
    - `return`: remaining message size to be copied.

```c
unsigned long copy_to_user(void __user *to, const void *from, unsigned long count)
```
Example 2: character driver

Writing data to the driver

- **mydevice_write**
  - **Field** `write` of the structure `file_operations`
  - **Associated to the function** `mydevice_write`
  - **Parameters**
    - `struct file *filp`: `mydevice_fops` of the driver
    - `char *buf`: user space buffer of data to copy to kernel space.
    - `size_t count`: number of bytes to write.
    - `loff_t *f_pos`: pointer to the destination data.
    - `return`: number of bytes written.

```c
ssize_t mydevice_write( struct file *filp, char *buf, size_t count, loff_t *f_pos) {
    copy_from_user(mydevice_buffer,buf,count);
    return 1;
}
```

- **Example**
  - **Function** `copy_from_user` copies `count` bytes from user space to kernel space.
Example 2: character driver

Writing data to the driver

- **copy_from_user**
  - Function `copy_from_user` is used to copy a block of data to *kernel space*.
  - **Parameters**
    - **to**: *kernel space* destination buffer.
    - **from**: *user space* source buffer
    - **count**: number of bytes to copy
    - **return**: remaining message size to be copied.

```c
unsigned long copy_from_user(void *to, const void __user *from, unsigned long count)
```
Example 2: character driver

Testing the driver

- The following C test application:
  - Opens the « memory » driver device file.
  - Writes a « Hello world !! » message.
  - Reads a message from the driver device file.
  - Closes the device.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include "mydevice.h"

int main(){
    int mydevice_file;
    char *msg_passed = "Hello World !!\n";
    char *msg_received;
    int msg_length;
    msg_length = strlen(msg_passed);
    msg_received = malloc(msg_length);
    mydevice_file = open(MYDEVICE_PATH, O_RDWR);
    if (mydevice_file == -1) {
        printf("ERROR OPENING FILE %s\n", MYDEVICE_PATH);
        exit(EXIT_FAILURE);
    }
    write(mydevice_file, msg_passed, msg_length);
    read(mydevice_file, msg_received, msg_length);
    printf("write/read test: %s\n", msg_received);
    close(mydevice_file);
    free(msg_received);
    return 0;
}
```
Example 2: character driver

Testing the driver

- Device file creation
  
  $ mknod /dev/mydevice c 60 0
  
  $ chmod 666 /dev/mydevice

- Module installation
  
  $ insmod mydevice.ko

- Test application
  
  $ ./test

- Module elimination
  
  $ rmmod mydevice.ko
Device driver development

- Device drivers
- User / device interaction
- Example 1: basic module
- Example 2: character driver
- Example 3: « ioctl » method
- Example 4: the « mmap » method
User / kernel driver functions correspondence

<table>
<thead>
<tr>
<th>Actions</th>
<th>User functions</th>
<th>Kernel functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load module</td>
<td>insmod</td>
<td>module_init()</td>
</tr>
<tr>
<td>Open device</td>
<td>open</td>
<td>file_operations: open</td>
</tr>
<tr>
<td>Read device</td>
<td>read</td>
<td>file_operations: read</td>
</tr>
<tr>
<td>Write device</td>
<td>write</td>
<td>file_operations: write</td>
</tr>
<tr>
<td>Close device</td>
<td>close</td>
<td>file_operations: release</td>
</tr>
<tr>
<td>Remove module</td>
<td>rmmod</td>
<td>module_exit()</td>
</tr>
<tr>
<td>I/O Control</td>
<td>ioctl</td>
<td>file_operations: ioctl</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

System call  
Corresponding function executed in kernel space
Example 3: « ioctl » method

- **Device file**
  - The device file is used to abstract the device. A user/application interacts with the device using this file.

- **Most devices are used as an input or output.**
  - System calls are used in practice: `open`, `close`, `read`, `write`
  - This mechanism provides *data* exchange ability.

- **Device control**
  - Devices also require control information.
  - Examples: change the transmission speed of a serial communication, eject CD-ROM disk, etc.
  - A specific mechanism is used for the control of devices: `ioctl`
  - Each device should implement its own `ioctl` function.
Example 3: « ioctl » method

- **Device control:** `ioctl`
  - input/output control
    - An `ioctl` is a system call handling specific device control operation that can not be processed by the other system calls.
  - Defines specific device control operations
    - Examples: eject CD-ROM disk, read a CD-ROM.
    - Each possible operation (from *user space*) on the device is well defined and corresponds to a unique request code (eject, read, etc.).

- Application to the character driver example
  - write a function `mydevice_ioctl` offering the following driver functionalities to *user space*:
    - `IOCTL_SET_MSG`: to send a message to the driver
    - `IOCTL_GET_MSG`: to read a message from the driver
Example 3: « ioctl » method

Application to the character driver example

- Déclaration of mydevice_ioctl prototype.

```c
/* Declaration of memory.c functions */
int mydevice_open(struct inode *inode, struct file *filp);
int mydevice_release(struct inode *inode, struct file *filp);
ssize_t mydevice_read(struct file *filp, char *buf, size_t count, loff_t *f_pos);
ssize_t mydevice_write(struct file *filp, char *buf, size_t count, loff_t *f_pos);
int mydevice_ioctl(struct inode *inode, struct file *filp, unsigned int ioctl_num,
                   unsigned long ioctl_param);
void mydevice_exit(void);
int mydevice_init(void);
```

- Parameters:
  - struct inode *inode: kernel structure, non used by programmer
  - struct file *filp: mydevice_fops operations of the driver
  - unsigned int ioctl_num: request code number (coding the operation process).
    Request codes are described in a header file (.h) because it is used by both the driver and
    the application.
  - unsigned long ioctl_param: parameter (optional)
Example 3: « ioctl » method

Application to the character driver example

- Add a function
  `mydevice_ioctl` to the list of operations supported by the driver.

```c
/* Structure that declares the usual file access functions */
struct file_operations mydevice_fops = {
  ioctl: mydevice_ioctl,
  read: mydevice_read,
  write: mydevice_write,
  open: mydevice_open,
  release: mydevice_release
};
```
Example 3: « ioctl » method

Application character driver example

- Request codes used:
  - IOCTL_SET_MSG
  - IOCTL_GET_MSG

- Declaration of request codes:
  - Request codes are created by the following macros:
    - _IO: ioctl function with no parameters
    - _IOR: ioctl function with read parameters
    - _IOW: ioctl function with write parameters
    - _IOWR: ioctl function with write and read parameters
  - In our example, IOCTL_SET_MSG inputs a write parameter while IOCTL_GET_MSG inputs a read parameter.
  - Request codes are generally placed in a header file (here mydevice.h) because they are used by both the driver and the application.

```
#include <linux/ioctl.h>
#define IOCTL_SET_MSG _IOW(MYDEVICE_MAJOR, 0, char *)
#define IOCTL_GET_MSG _IOR(MYDEVICE_MAJOR, 1, char *)
```
Exemple 3: « ioctl » method

Application to the character driver example

- **mydevice_ioctl**
  - IOCTL_SET_MESG
    - To send a message to the device
    - Use of function `copy_from_user`
  - IOCTL_GET_MESG
    - To read a message from the device
    - Use of function `copy_to_user`

```c
int mydevice_ioctl(struct inode *inode,
                   struct file *filp,
                   unsigned int ioctl_num,
                   unsigned long ioctl_param) {
    int msg_length;
    switch (ioctl_num) {
      case IOCTL_SET_MSG:
        msg_length = strlen((char *)ioctl_param);
        copy_from_user(mydevice_buffer, (char *)ioctl_param, msg_length);
        printk("IOCTL_SET_MSG: message set to mydevice -> %s\n", mydevice_buffer);
        break;
      case IOCTL_GET_MSG:
        msg_length = strlen(mydevice_buffer);
        copy_to_user((char *)ioctl_param, mydevice_buffer, msg_length);
        printk("IOCTL_GET_MSG: message got from mydevice -> %s\n", (char *)ioctl_param);
        break;
    }
    return 0;
}
```
Exemple 3: « ioctl » method

Testing the driver
- The following C test application:
  - Opens the device file
  - Invokes an ioctl system call to set a message to the device.
  - Invokes an ioctl system call to get a message from the device
  - Closes the device file

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include "mydevice.h"

int main(){
    int mydevice_file;
    char *msg_passed = "Hello World !!\n";
    char *msg_received;
    int msg_length;
    msg_length = strlen(msg_passed);
    msg_received = malloc(msg_length);
    mydevice_file = open(MYDEVICE_PATH, O_RDWR);
    if (mydevice_file == -1) {
        printf("ERROR OPENING FILE %s\n", MYDEVICE_PATH);
        exit(EXIT_FAILURE);
    }
    // IOCTL BASIC TEST
    int ret_val;
    ret_val = ioctl(mydevice_file, IOCTL_SET_MSG, msg_passed);
    ret_val = ioctl(mydevice_file, IOCTL_GET_MSG, msg_received);
    close(mydevice_file);
    free(msg_received);
    return 0;
}
```
Exemple 3: « ioctl » method

Note

- The `ioctl` function used in *user space* is a system call
  - Declared in header file:
    ```c
    #include <linux/ioctl.h>
    ```

- Header file `mydevice.h`
  - Defines data that are common to the driver and application
    - `IOCTL_SET_MSG`
    - `IOCTL_GET_MSG`
    - Etc.

```c
// fichier prototype mydevice.h
#include <linux/ioctl.h>
#define MYDEVICE_MAJOR 60
#define MYDEVICE_PATH "/dev/mydevice"
#define MAX_BUFFER_SIZE 100

/* Command numbers of the device driver */
#define IOCTL_SET_MSG _IOW(MYDEVICE_MAJOR, 0, char *)
#define IOCTL_GET_MSG _IOR(MYDEVICE_MAJOR, 1, char *)
```
Example 3: « ioctl » method

Testing the driver

- Creation of the device file
  
  $ mknod /dev/mydevice c 60 0

  $ chmod 666 /dev/mydevice

- Loading the module
  
  $ insmod mydevice.ko

- Executing test application
  
  $ ./test

- Removing module
  
  $ rmmod mydevice.ko
Example 3: « ioctl » method

- Character driver example
Device driver development

- Device drivers
- User / device interaction
- Example 1: basic module
- Example 2: character driver
- Example 3: « ioctl » method
- Example 4: the « mmap » method
**Example 4: « mmap » method**

**User / kernel driver functions correspondance**

<table>
<thead>
<tr>
<th>Actions</th>
<th>User functions</th>
<th>Kernel functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load module</td>
<td>insmod</td>
<td>module_init()</td>
</tr>
<tr>
<td>Open device</td>
<td>open</td>
<td>file_operations: open</td>
</tr>
<tr>
<td>Read device</td>
<td>read</td>
<td>file_operations: read</td>
</tr>
<tr>
<td>Write device</td>
<td>write</td>
<td>file_operations: write</td>
</tr>
<tr>
<td>Close device</td>
<td>close</td>
<td>file_operations: release</td>
</tr>
<tr>
<td>Remove module</td>
<td>rmmod</td>
<td>module_exit()</td>
</tr>
<tr>
<td>I/O Control</td>
<td>ioctl</td>
<td>file_operations: ioctl</td>
</tr>
<tr>
<td>Memory map</td>
<td>mmap</td>
<td>file_operations: mmap</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**System call**

**Corresponding function executed in kernel space**

![Diagram](image.png)
Example 4: « mmap » method

Limit of character drivers
- This type of driver is suited for simple devices where data communication speed is not critical (stream of bytes of any size).
- Example: serial RS-232 communication driver.

mmap
- Example: driver for video
  - In case of video processing, very large amount of data are exchanged. Driver speed is critical in this case.
- The mmap function provides a way to directly access a memory region of the driver.
  - System call
  - The driver must implement a mmap function.
  - This function is more difficult to write because of the complexity of Linux virtual memory management.
  - Data accesses are much faster using this method.
Example 4: « mmap » method

Case study: the Linux framebuffer

- The Linux framebuffer has been developed to serve as a standard interface for display devices.
- The device file associated to the framebuffer is /dev/fb[0-9]
- The device file is accessed using the usual system calls:
  - open/close
  - read/write
  - ioctl
  - mmap

- The device file represents the memory footprint of the screen
- Reading the device file is equivalent to reading pixels of the screen, from left to right and from top to bottom.
- The number of bytes to be read for a pixel is determined by the color depth of the framebuffer.
Example 4: « mmap » method

Case study: the Linux framebuffer

Figure 4: Fonctionnement du frame buffer
Example 4: « mmap » method

Case study: the Linux framebuffer

- Color depth. Example 32-bit RGB888.

RGB888

<table>
<thead>
<tr>
<th>bits</th>
<th>31 - 24</th>
<th>23 - 16</th>
<th>15 - 8</th>
<th>7 - 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGB888</td>
<td>Transp.</td>
<td>R</td>
<td>G</td>
<td>B</td>
</tr>
</tbody>
</table>
Case study: the Linux framebuffer

Information related to the framebuffer can be accessed by an `ioctl` call (after opening the fb device file):

- Request code `FBIOGET_FSCREENINFO` provides the following information:

```c
struct fb_fix_screeninfo {
    char id[16]; /* identification string e.g. "TT"
    unsigned long smem_start; /* Start of frame buffer mem */
    /* (physical address) */
    __u32 smem_len;
    __u32 type;
    __u32 type_aux;
    /* see FB_TYPE_ */
    /* Interleave for interleaved Planes */
    __u32 visual;
    __u16 xpanstep;
    __u16 ypanstep;
    __u16 ywrapstep;
    __u32 line_length;
    unsigned long mmio_start; /* Start of Memory Mapped I/O */
    /* (physical address) */
    __u32 mmio_len;
    __u32 accel;
    /* Indicate to driver which */
    /* specific chip/card we have */
    __u16 reserved[3]; /* Reserved for future compatibility */
};
```
Example 4: « mmap » method

- Case study: the Linux framebuffer
  - Information related to the framebuffer can be accessed by an ioctl call (after opening the fb device file):
    - Request code
      
      ```c
      struct fb_var_screeninfo {
        __u32 xres;        /* visible resolution */
        __u32 yres;        /* virtual resolution */
        __u32 xres_virtual; /* offset from virtual to visible */
        __u32 yres_virtual; /* resolution */
        __u32 xoffset;     /* guess what */
        __u32 yoffset;     /* != 0 Graylevels instead of colors */
        __u32 bits_per_pixel;
        __u32 grayscale;
      };
      ```
      
      ```c
      struct fb_bitfield red;        /* bitfield in fb mem if true color, */
      struct fb_bitfield green;      /* else only length is significant */
      struct fb_bitfield blue;       /* transparency */
      struct fb_bitfield transp;     /* != 0 Non standard pixel format */
      __u32 nonstd;                  /* see FB_ACTIVATE */
      __u32 activate;                /* height of picture in mm */
      __u32 width;                   /* width of picture in mm */
      __u32 accel_flags;             /* (OBSOLETE) see fb_info.flags */
      */
      /* Timing: All values in pixclocks, except pixclock (of course) */
      __u32 pixclock;                /* pixel clock in ps (pico seconds) */
      */
      __u32 left_margin;            /* time from sync to picture */
      __u32 right_margin;           /* time from picture to sync */
      __u32 upper_margin;           /* time from sync to picture */
      __u32 lower_margin;
      __u32 hsync_len;              /* length of horizontal sync */
      __u32 vsync_len;              /* length of vertical sync */
      __u32 sync;                   /* see FB_SYNC */
      __u32 vmode;                  /* see FB_VMODE */
      __u32 reserved[5];            /* angle we rotate counter clockwise */
      */

      Polytech’Nice Sophia - Département Electronique - Université de Nice Sophia Antipolis - S. Bilavarn - 52 -
Example 4: « mmap » method

Case study: the Linux framebuffer

mmap

- The Linux framebuffer driver implements a mmap function.
- To use the Linux framebuffer:
  - Open the /dev/fb0 device file
  - Get framebuffer address (fb_ptr) by a call to mmap.
  - Framebuffer is then accessible by this pointer (fb_ptr).
  - Accessing a pixel is then much faster than read/write system calls to the device file.

```c
#define FRAME_BUFFER "/dev/fb0"

void my_function(void) {
    unsigned int *fb_ptr;
    int fb_dev; float R, G, B, T;
    fb_dev = open (FRAME_BUFFER, O_RDWR);
    struct fb_fix_screeninfo fb_fix_infos;
    struct fb_var_screeninfo fb_var_infos;

    ioctl (fb_dev, FBIOGET_FSCREENINFO, &fb_fix_infos);
    ioctl (fb_dev, FBIOGET_VSCREENINFO, &fb_var_infos);

    fb_ptr = (unsigned short *)mmap(NULL,
                                    fb_fix_infos.smem_len, PROT_READ | PROT_WRITE, MAP_SHARED, fb_dev, 0);

    unsigned int rgb_pixel =
        (((unsigned int)(B+0.5))&0x000000FF|
         (((unsigned int)(G+0.5)<<8)&0x0000FF00|
         (((unsigned int)(R+0.5)<<16)&0x00FF0000|
         (((unsigned int)(T+0.5)<<24)&0xFF000000);

    fb_ptr [0] = rgb_pixel;
    munmap(fb_ptr, fb_fix_infos.smem_len);
}
```
Example 4: « mmap » method

- Case study: the Linux framebuffer