

# EMBEDDED LINUX KERNEL DEVELOPMENT

## REPORT 3

### 1 INTRODUCTION

In this laboratory, we will develop a driver and application using the *ioctl* method.

The *ioctl* method allows data transfer between the kernel space and the user space.

The Linux kernel runs on an Exynos board.

### 2 TEST PROJECT

We are using the *ioctl* template already available on the board.

This code provides a template for developing driver, with init and close function for example.

We also use the test script. This script allows us to install the driver, run the userspace application, then remove the driver.

Before running the script, we need to use the command "**mknod /dev/mydevice c 259 0**".

```
#mknod /dev/mydevice c 259 0
insmod mydevice.ko
./test_driver
rmmod mydevice.ko
dmesg|tail
```

Figure 1- test script

### 3 IOCTL METHOD

Using the previous template, we fill the *mydevice\_ioctl* method.

```
long mydevice_ioctl(struct file *filp, unsigned int ioctl_num, unsigned long ioctl_param) {
    switch(ioctl_num){
        case IOCTL_SET_MSG:
            copy_from_user(mydevice_buffer, (char*)(ioctl_param), MAX_BUFFER_SIZE);
            printk("IOCTL_SET_MSG: message sent to mydevice->%s\n", mydevice_buffer);
            break;
        case IOCTL_GET_MSG:
            copy_to_user((char*)(ioctl_param), mydevice_buffer, MAX_BUFFER_SIZE);
            printk("IOCTL_GET_MSG: message got from mydevice->%s\n", mydevice_buffer);
            break;
    }
}
```

Figure 2- ioctl method with send/receive message

```
/* Command numbers of the device driver */
#define IOCTL_SET_MSG _IOW(MYDEVICE_MAJOR, 0, char *)
#define IOCTL_GET_MSG _IOR(MYDEVICE_MAJOR, 1, char *)
```

Figure 3- macros definition for ioctl switch

With this one function, we can send and receive a message to and from the driver. Using the *copy\_from\_user/copy\_to\_user* methods, we can avoid pointer manipulation in the driver code.

The IOCTL\_SET\_MSG/IOCTL\_GET\_MSG values are computed with kernel macros to avoid potential conflict.

```
void ioctl_set_msg(int file_desc, char* msg){
    ioctl(file_desc, IOCTL_SET_MSG, msg);
}

void ioctl_get_msg(int file_desc, char* msg){
    ioctl(file_desc, IOCTL_GET_MSG, msg);
    printf("Msg got : %s\n",msg);
}

int main()
{
    int mydevice_file;
    char msg_passed[MAX_BUFFER_SIZE] = "Hello World !!";
    char msg_received[MAX_BUFFER_SIZE] = "";
    int msg_length;
    int ret_val;

    msg_length = strlen(msg_passed) + 1;

    mydevice_file = open(MYDEVICE_PATH, O_RDWR);
    if (mydevice_file == -1)
    {
        printf("ERROR OPENING FILE %s\n", MYDEVICE_PATH);
        exit(EXIT_FAILURE);
    }

    // BASIC WRITE/READ TEST
    write(mydevice_file, msg_passed, msg_length);
    read(mydevice_file, msg_received, msg_length);
    printf("write/read test: %s\n", msg_received);

    // IOCTL TEST
    ioctl_set_msg(mydevice_file, msg_passed);
    ioctl_get_msg(mydevice_file, msg_received);

    close(mydevice_file);

    return 0;
}
```

Figure 4- userspace application test for ioctl

We can write a simple userspace application to test if the driver behaves normally. As we can see bellow, the driver received the HelloWorld message and sent it back correctly.

```
odroid@odroid:~/LABS/TP_IOCTL$ sudo ./test
write/read test: Hello World !!
Msg got : Hello World !!
```

Figure 5- userspace result

```
[ 2731.346566] IOCTL_SET_MSG: messgae ste to mydevice->Hello W
orld !!

[ 2731.346581] IOCTL_GET_MSG: message got from mydevice->Hello
World !!
```

Figure 6- driver log

## 4 APPLICATION: IOCTL BASED PERFORMANCE MONITORING

We want to use our ioctl driver to test the performance monitor of our Cortex based microprocessor. The performance monitor uses specific registers, setting these registers requires a kernel level access. In our *mydevice* module, we define two new functions :

- `perfmon_ioctl_start()` that sets the registers to start monitoring performance
- `perfmon_ioctl_stop()` that reads and prints the cycle count register value (number of cycles since `perfmon_ioctl_start()` has been called)

```

171 void perfmon_ioctl_start(void){
172
173     // Disable all individual counters
174     asm volatile("mov r0, #0x8000000F");
175     asm volatile("mcr p15, 0, r0, c9, c12, 2");
176
177     // Disable the PMNC
178     asm volatile("MRC p15, 0, r0, c9, c12, 0"); // Read PMNC
179     asm volatile("ORR r0, r0, #0x0"); // Disable
180     asm volatile("MCR p15, 0, r0, c9, c12, 0"); // Write PMNC
181
182     // Select register and event
183     asm volatile("mov r0, #0x0");
184     asm volatile("mcr p15, 0, r0, c9, c12, 5");
185     asm volatile("mov r0, #0x55");
186     asm volatile("mcr p15, 0, r0, c9, c13, 1");
187
188     // Enable all individual counters
189     asm volatile("mov r0, #0x8000000F");
190     asm volatile("mcr p15, 0, r0, c9, c12, 1");
191
192     // Enable PMNC
193     asm volatile("MRC p15, 0, r0, c9, c12, 0"); // Read PMNC
194     asm volatile("ORR r0, r0, #0x7"); // Enable and re-set
195     asm volatile("MCR p15, 0, r0, c9, c12, 0"); // Write PMNC
196
197     // Run test function
198     printk("<1>Start Profiling!\n");
199     // Read CCNT (Cycle CouNT) Register
200     asm volatile("mrc p15, 0, %0, c9, c13, 0" : "=r" (val1));
201     printk(" CCNT = 0x%08x\n", val1);
202 }
203

```

Figure 7- `perfmon_ioctl_start()` function

```

204 void perfmon_ioctl_stop(void){
205
206     // Read CCNT (Cycle CouNT) Register
207     asm volatile("mrc p15, 0, %0, c9, c13, 0" : "=r" (val2));
208     printk(" CCNT = 0x%08x\n", val2);
209
210     printk(" EXEC TIME = %d CYCLES\n", val2-val1);
211 }
212

```

Figure 8- `perfmon_ioctl_stop()` function

These functions are called using the special request codes `IOCTL_PERFMON_START` and `IOCTL_PERFMON_STOP`.

```

213 long mydevice_ioctl(struct file *filp, unsigned int ioctl_num, unsigned long ioctl_param) {
214     switch(ioctl_num){
215         case IOCTL_SET_MSG:
216             copy_from_user(mydevice_buffer, (char*)(ioctl_param), MAX_BUFFER_SIZE);
217             printk("IOCTL_SET_MSG: message set to mydevice->%s\n", mydevice_buffer);
218             break;
219         case IOCTL_GET_MSG:
220             copy_to_user((char*)(ioctl_param), mydevice_buffer, MAX_BUFFER_SIZE);
221             printk("IOCTL_GET_MSG: message got from mydevice->%s\n", mydevice_buffer);
222             break;
223         case IOCTL_PERFMON_START:
224             perfmon_ioctl_start();
225             break;
226         case IOCTL_PERFMON_STOP:
227             perfmon_ioctl_stop();
228             break;
229         default:;
230     }
231     return 0;
232 }

```

Figure 9- updated driver function

To use performance monitoring in a userspace diagram, we need to surround the monitored code with calls to ioctl using the IOCTL\_PERFMON\_START and IOCTL\_PERFMON\_STOP codes. For instance, in the userspace program below we monitor the performance of 100 messages being sent to the driver.

```

12 void ioctl_start_perf(int file_desc){
13     ioctl(file_desc, IOCTL_PERFMON_START, "");
14 }
15
16 void ioctl_stop_perf(int file_desc){
17     ioctl(file_desc, IOCTL_PERFMON_STOP, "");
18 }
19
30 int main()
31 {
32     int mydevice_file;
33     char msg_passed[MAX_BUFFER_SIZE] = "Hello World !!";
34     char msg_received[MAX_BUFFER_SIZE] = "";
35     int msg_length;
36     int ret_val;
37
38     msg_length = strlen(msg_passed) + 1;
39
40     mydevice_file = open(MYDEVICE_PATH, O_RDWR);
41     if (mydevice_file == -1)
42     {
43         printf("ERROR OPENING FILE %s\n", MYDEVICE_PATH);
44         exit(EXIT_FAILURE);
45     }
46
47     // BASIC WRITE/READ TEST
48     write(mydevice_file, msg_passed, msg_length);
49     read(mydevice_file, msg_received, msg_length);
50     printf("write/read test: %s\n", msg_received);
51
52     // IOCTL TEST
53     ioctl_start_perf(mydevice_file);
54     for (int i = 0; i < 100; i++)
55         ioctl_set_msg(mydevice_file, msg_passed);
56     ioctl_stop_perf(mydevice_file);
57     ioctl_get_msg(mydevice_file, msg_received);
58     close(mydevice_file);
59
60     return 0;
61 }
62

```

Figure 10- test\_driver userspace program

The screenshot shows a development environment with two windows. The left window is a code editor titled 'test\_driver.c (~LABS/TP\_IOCTL) - Pluma' showing the source code of a C program. The right window is a terminal titled 'odroid@odroid: ~/LABS/TP\_IOCTL' showing the execution output of the program.

```

28 }
29
30 int main()
31 {
32     int mydevice_file;
33     char msg_passed[MAX_BUFFER_SIZE] = "Hello World !!";
34     char msg_received[MAX_BUFFER_SIZE] = "";
35     int msg_length;
36     int ret_val;
37
38     msg_length = strlen(msg_passed) + 1;
39
40     mydevice_file = open(MYDEVICE_PATH, O_RDWR);
41     if (mydevice_file == -1)
42     {
43         printf("ERROR OPENING FILE %s\n", MYDEVICE_PATH);
44         exit(EXIT_FAILURE);
45     }
46
47     // BASIC WRITE/READ TEST
48     write(mydevice_file, msg_passed, msg_length);
49     read(mydevice_file, msg_received, msg_length);
50     printf("write/read test: %s\n", msg_received);
51
52
53     // IOCTL TEST
54     ioctl_start_perf(mydevice_file);
55     for (int i = 0; i < 100; i++)
56     {
57         ioctl_set_msg(mydevice_file, msg_passed);
58         ioctl_stop_perf(mydevice_file);
59         ioctl_get_msg(mydevice_file, msg_received);
60     }
61     close(mydevice_file);
62
63     return 0;
64 }

```

```

odroid@odroid:~/LABS/TP_IOCTL$ sudo ./test
write/read test: Hello World !!
Msg got : Hello World !!
[ 2731.346543] IOCTL_SET_MSG: messgae ste to mydevice->Hello W
orld !!
[ 2731.346547] IOCTL_SET_MSG: messgae ste to mydevice->Hello W
orld !!
[ 2731.346552] IOCTL_SET_MSG: messgae ste to mydevice->Hello W
orld !!
[ 2731.346557] IOCTL_SET_MSG: messgae ste to mydevice->Hello W
orld !!
[ 2731.346561] IOCTL_SET_MSG: messgae ste to mydevice->Hello W
orld !!
[ 2731.346566] IOCTL_SET_MSG: messgae ste to mydevice->Hello W
orld !!
[ 2731.346571] CCNT = 0x000e1361
[ 2731.346576] EXEC TIME = 910453 CYCLES
[ 2731.346581] IOCTL_GET_MSG: message got from mydevice->Hello
World !!
[ 2731.349643] <1>Removing mydevice module
odroid@odroid:~/LABS/TP_IOCTL$

```

Figure 11- result of the performance monitoring

## 5 CONCLUSION

In the previous LAB, we saw how to setup a driver/userspace-application communication using driver files.

In this LAB, we saw we can use the *ioctl* method to directly communicate with the driver. This method is simpler, but more error prone. Indeed, before we got it right, the driver made the system crash a few times.