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: messgae ste 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)

```
void perfmon_ioctl_start(void){

  // Disable all individual counters
  asm volatile("mov r0, #0x8000000F");
  asm volatile("mcr p15, 0, r0, c9, c12, 2");

  // Disable the PMNC
  asm volatile("MRC p15, 0, r0, c9, c12, 0"); // Read PMNC
  asm volatile("ORR r0, r0, #0x0"); // Disable
  asm volatile("MCR p15, 0, r0, c9, c12, 0"); // Write PMNC

  // Select register and event
  asm volatile("mov r0, #0x0");
  asm volatile("mcr p15, 0, r0, c9, c12, 5");
  asm volatile("mov r0, #0x55");
  asm volatile("mcr p15, 0, r0, c9, c13, 1");

  // Enable all individual counters
  asm volatile("mov r0, #0x8000000F");
  asm volatile("mcr p15, 0, r0, c9, c12, 1");

  // Enable PMNC
  asm volatile("MRC p15, 0, r0, c9, c12, 0"); // Read PMNC
  asm volatile("ORR r0, r0, #0x7"); // Enable and re-set
  asm volatile("MCR p15, 0, r0, c9, c12, 0"); // Write PMNC

  // Run test function
  printk("<1>Start Profiling!\n");
  // Read CCNT (Cycle CouNT) Register
  asm volatile("mrc p15, 0, %0, c9, c13, 0" : "=r" (val1));
  printk(" CCNT = 0x%08x\n", val1);
}
```

Figure 7- perfmon_ioctl_start() function

```
void perfmon_ioctl_stop(void){

  // Read CCNT (Cycle CouNT) Register
  asm volatile("mrc p15, 0, %0, c9, c13, 0" : "=r" (val2));
  printk(" CCNT = 0x%08x\n", val2);

  printk(" EXEC TIME  = %d CYCLES\n", val2-val1);
}
```

Figure 8- perfmon_ioctl_stop() function

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

```c
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: messgae ste 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;
  case IOCTL_PERFMON_START:
      perfmon_ioctl_start();
    break;
  case IOCTL_PERFMON_STOP:
      perfmon_ioctl_stop();
    break;
  default:;
}
  return 0;
}
```

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.

```c
void ioctl_start_perf(int file_desc){
  ioctl(file_desc, IOCTL_PERFMON_START, "");
}

void ioctl_stop_perf(int file_desc){
  ioctl(file_desc, IOCTL_PERFMON_STOP, "");
}
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_start_perf(mydevice_file);
  for (int i = 0; i < 100; i++)
   ioctl_set_msg(mydevice_file, msg_passed);
  ioctl_stop_perf(mydevice_file);
  ioctl_get_msg(mydevice_file, msg_received);
  close(mydevice_file);

  return 0;
}
```

Figure 10- test_driver userspace program

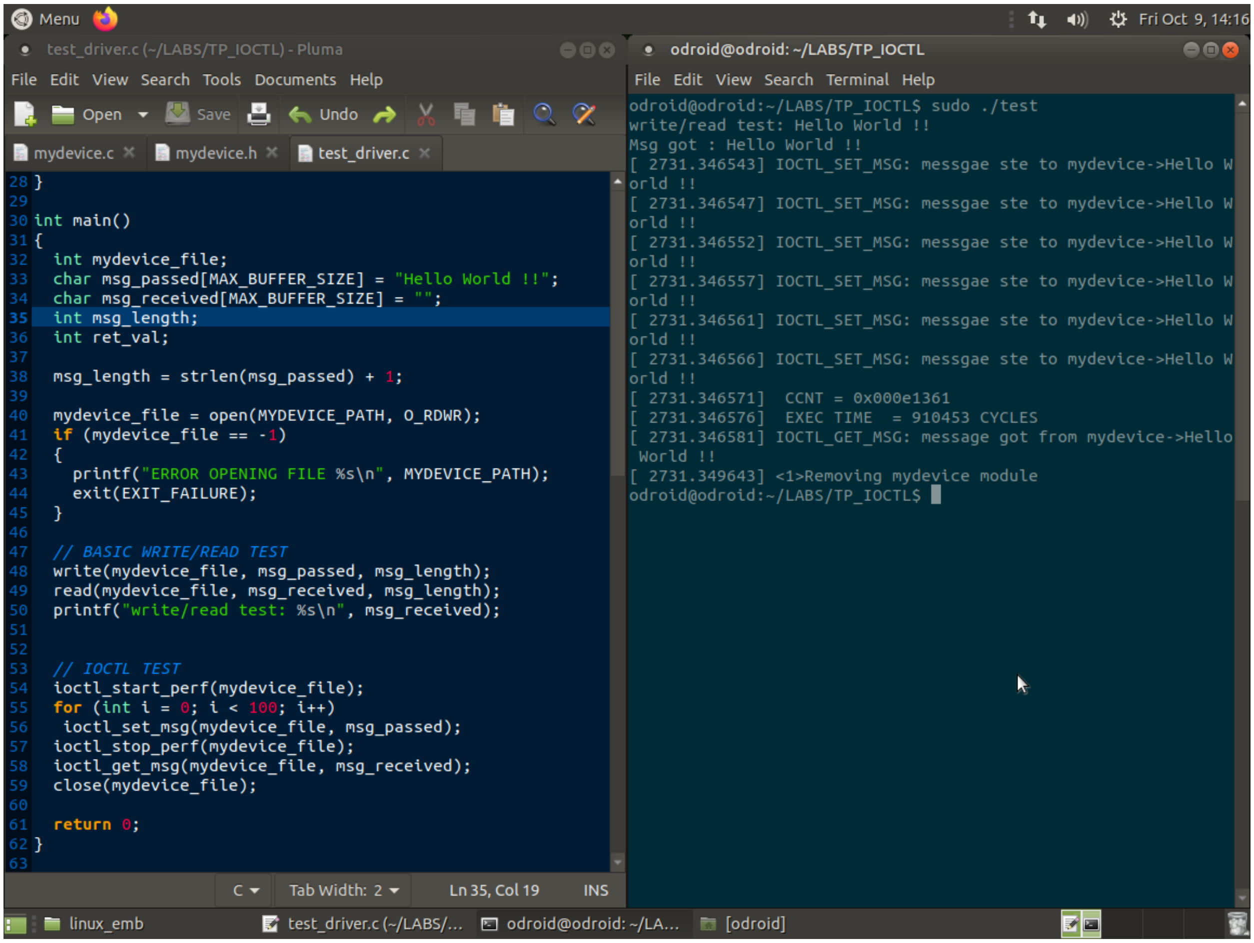

Figure 11- result of the performance monitoring

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.