Linux Device Driver in Action (LDDiA)

Duy-Ky Nguyen **2015-04-30**

1. On Memory

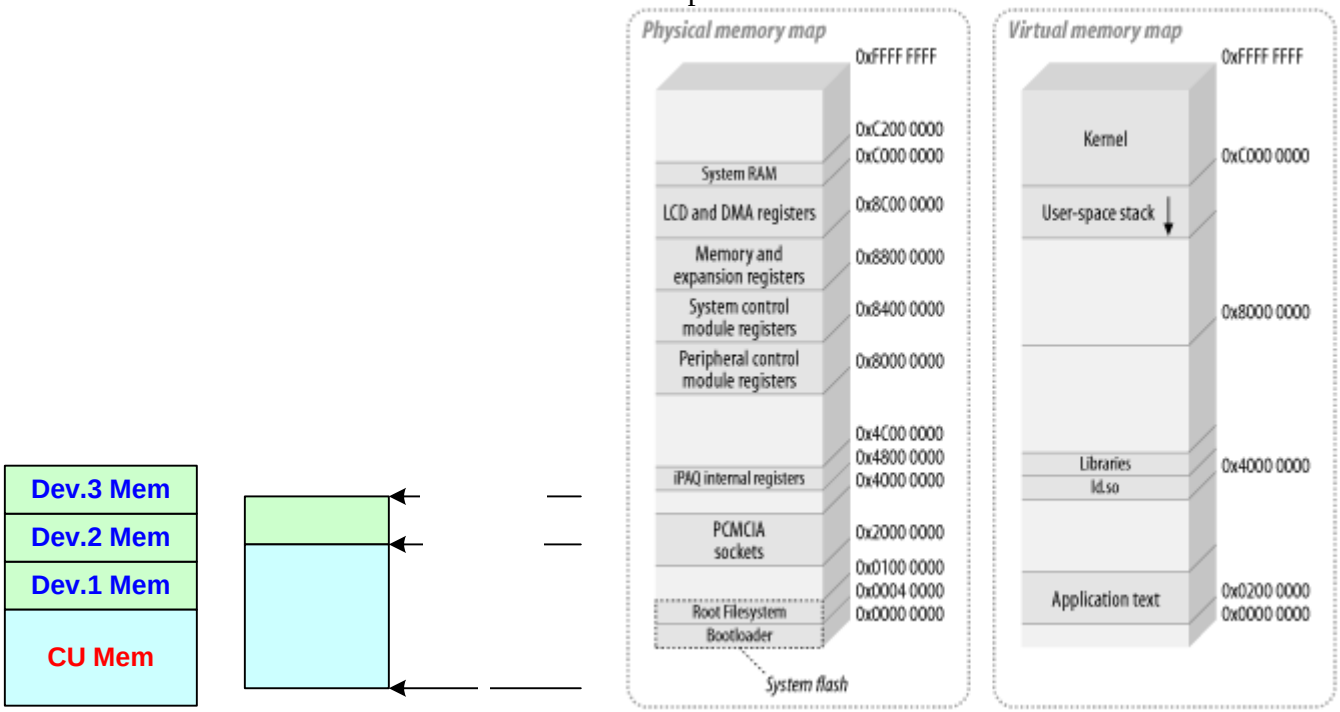
Any unit under user control must have a controller board (CB) with a controller unit (CU) and several devices (Dev.x) doing what user wants. Each device has its own memory map (mem-map), including CU, under user control, actual under SW control as user makes SW control the unit. CB is also a device, so it must have its own mem-map by combining those of CU and Dev.x. Each device now has its own **base address** and its own mem-map is **offset address** to that base address. Address in CB mem-map is known as **absolute address** or **physical address** of CB.

In a bare-metal unit with no operating system (OS), SW accesses device using physical address.

SW uses dynamic memory allocation to deal with memory shortage, and memory becomes fragmented after so many allocation/de-allocation to free up memory after used. "**Memory leak**" happens when memory de-allocation does not occur for some reason. The memory allocation cannot be done with some size when the memory is too fragmented! The technology Memory Management Unit (**MMU**) comes to rescues this problem in mapping fragmented memory into contiguous one using "**page table**" with "page per entry" (PTE) associated with "translation lookaside buffer" (TLB). The new mapped address by MMU is known as Intel jargon **logical address**.

SW uses **virtual address** to access devices, it could be physical or logical.

System, a big unit, could e shutdown due to error in setting device address. So Linux divides memory space into 2 areas (1) kernel space (2) user space, where only kernel space is allowed to access device memory. The maximum of 32-bit number is 4G and Linux reserves the top 1 GB for kernel and the bottom 3 GG for user.



To make SW even more efficient, Linux kernel has its own memory management where virtual memory size is bigger than physical (more memory allocation), hence there's the term "**high memory**" for this unreal extra memory, and "low memory" for real one. In Linux kernel, **logical address** is virtual address at "low memory"

created by kmalloc(), while virtual one created by vmalloc(). The bottom line is using kmalloc() or vmalloc() depends on what expected in function arguments.

2. Linux Device Driver (LDD)

User can access direct to HW devices in a single-task OS (bare-metal, DOS or upto Winws-98), but must go through check-point of kernel in multi-task OS like Linux or starting from Windows-NT, …, Windows-7.

Every HW device has its own driver to be accessible and Linux provides a unified approach where all HW device seen as a FILE, device file indeed, and a well-defined file operations to be implemented, regardless it is a LED, USB, Network, hard-drive, …

There are 4 file operations (1) open (2) close (3) read (4) write. These are the most popular ones, there are a lot more and could be found in kernel source code.

In Linux, drivers are known as **modules** with an option of loading by kernel at startup or by user later on, so there're 2 module operations : (1) load (2) remove.

Operation	Kernel function (driver)	User function (app)	User command
Open device	file_operations: open = dev_open	open()	
Close device	file_operations: release = dev_close	close()	
Read device	file_operations: read = dev_read	read()	cat
Write device	file_operations: write = dev_write	write()	echo
Device Control	file_operations: ioctl = dev_ioctl	ioctl()	
Load module	module_init(dev_init)		insmod
Remove module	module_exit(dev_exit)		rmmod
Check module			lsmod

NOTE

- dev_open, dev_close, dev_read, dev_write, dev_ioctl, dev_mmap are user-defined functions, can be any name
- open, close, read, write, ioctl, mmap are kernel-defined functions, cannot be different name
- Low-Level file operations open, clode, read, write must be used, NOT fopen, fclose, fwrite, fread.
- a FPU, Fake Processor Unit, not Floating Point ;-), just as a memory buffer is used to implement LDD with all operations above
- device file has 3 attributes (1) block[b] or char[c] type (2) major_num (3) minor_num; created by "mknod"

```
mknod console c 5 1
```

For DMA Device Memory, we need VMA ops

Operation	Kernel function (driver)	User function (app)
Device Memory	file_operations: mmap = dev_mmap	mmap()
Open VMA	vm_operations_struct: open = dev_vma_open	
Close VMA	vm_operations_struct: close = dev_vma_close	
NoPage VMA	vm_operations_struct: fault = dev_vma_nopage	

2.1. File Operations

For kernel to recognize our file operations, we need the following

New Syn tax

```
struct file_operations fops = {
       read: dev_read,
       write: dev_write,
       open: dev_open,
       release: dev_close
       ioctl: dev_ioctl,
       release: dev_close
};
```

Old syntax

```
struct file_operations fops = {
       .read = dev_read,
       .write = dev_write,
       .open = dev_open,
       .release = dev_close
       .ioctl = dev_ioctl,
       .mmap = dev_mmap
};
```

2.2. Module Operations

For kernel to recognize our module operations, we need the following

```
module_init(device_init);
module_exit(device_exit);
```

2.3. Module Script “lldd” for Dynamic Major Device Number

It’s convenient to have a tool to deal with dynamic device major number. It detects major number and makes node accordingly.

```
#!/bin/sh

# NAME “lldd”

echo "Check and Load module ${1}.ko"
MOD=$(lsmod | grep ${1})

if [[ ! $MOD == "" ]]; then
       sudo rmmod ${1}.ko
fi

sudo insmod ${1}.ko

##########################################################################################

echo "Check Major and Make dev node $1 c major 0"

# No space aroung '= otherwise interpreted as command, not variable!
MAJ=$(dmesg | tail -1 | awk '{print $NF}')

echo "Major : $MAJ"

if [[ ! $MAJ = *[[:digit:]]* ]]; then
       echo "Major not found!"
```

```
	exit
fi

if [[ -e /dev/${1} ]]; then
	sudo rm -f /dev/${1}
fi

sudo mknod /dev/${1} c $MAJ 0
sudo chmod 777 /dev/${1}

ls -als /dev/ | grep ${1}
```

2.4. Makefile

```
# “make” to compile driver, load module & make device node
# “make tst” to test

TGT = fprw
obj-m += ${TGT}.o

MJR = 250

all:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules
	echo 'make -C /lib/modules/$(shell uname -r)/build M=$(PWD) modules'
	@rm ${TGT}.o ${TGT}.mod.c ${TGT}.mod.o modules.order Module.symvers .${TGT}.*
	@rm -r .tmp_versions

tst:
	@
	@echo ""
	@echo "->Devide READ"
	cat /dev/${TGT}
	@echo ""
	@echo "->Device WRITE"
	printf "Hello from User!\n" > /dev/${TGT}
	@echo ""
	@echo "->Device Read to check Device Write"
	cat /dev/${TGT}
	@echo ""
	@echo "->Devide READ"
	cat /dev/${TGT}
	@echo ""
	@echo ""

clean:
	make -C /lib/modules/$(shell uname -r)/build M=$(PWD) clean
```

2.5. Working with Modules

There’re 3 “raw” utilities to work with modules : (1)lsmod (2) insmod (3) rmmod.

There’re also “high-level” “depmod” and “modprobe” based on those above, but with additional checking dependency at a price of extra requirements. Modules must be compiled and installed at the right place “/lib/modules” with many supporting files for “modprobe” to check dependency.

I found raw utilities work best during driver development as they’re fully under user control, so they’re used here. The “modprobe” is more suitable to deal with modules come with the kernel.

3. FPU Driver

3.1. FPU Driver : RW

3.1.1. Basic RW Driver

```
// fprw.c : kernel driver
// RUN : make; make tst

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.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 <linux/seq_file.h>
#include <linux/cdev.h>

////////////////////////////////////////////////////////////////////////////////////////

#include <asm/system.h>         /* cli(), *_flags */
#include <asm/uaccess.h>        /* copy_*_user */

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("Duy-Ky Nguyen");

#define      MEM_SIZE     0X1000
#define      READ_SIZE    0x20

#define      MEM_ID       0x12345678
#define      MEM_REV            0x01
#define      MEM_DATE     0x05062001

#define      MEM_DFT "0x12345678, 0x01, 0x05062001\0"
char FPU_MEM[MEM_SIZE] = MEM_DFT;

////////////////////////////////////////////////////////////////////////////////////////

#define FPU_DBG

#ifdef FPU_DEBUG
   #define DMSG(string, args...) printk(string, ##args)
#else
   #define DMSG(string, args...)
#endif

////////////////////////////////////////////////////////////////////////////////////////
// Start of device struct
static int   fpu_open (struct inode *, struct file *);
static int   fpu_close (struct inode *, struct file *);
static ssize_t fpu_read (struct file *, char *, size_t, loff_t *);
static ssize_t fpu_write (struct file *, const char *, size_t, loff_t *);

static int   fpu_ioctl (struct inode *, struct file *, uint, ulong);    // just place holder
static int   fpu_mmap(struct file *filp, struct vm_area_struct *vma);   // just place holder
```

```
static struct file_operations fpu_fops = {
   open:     fpu_open,
   release:  fpu_close,
   read:     fpu_read,    // copy_to_user(void __user * to, const void * from, unsigned long n);
   write:    fpu_write,   // copy_from_user(void * to, const void __user * from, unsigned long n);
};
// End of device struct

static int fpu_major_id = 0;

static const char fpu_name_id[] = "Dev_FPU";

////////////////////////////////////////////////////////////////////////////////////////////////

static ssize_t fpu_read(struct file *file_ptr, char __user *ubuf, size_t count, loff_t *pos)
{
   int len = strlen(FPU_MEM);
   if( *pos )
      return 0;

   // the whole buf used, update pos for next run to stop.
   *pos += len;     // count = 32K, by default

   // make sure fresh read; otherwise run forever
   DMSG("<< READ >> %s : count.%d pos.%d\n", fpu_name_id, count, (int)*pos);

   // transfer the whole buf from kernel to user for reading
   // put_user(val, ubuf) : val --> ubuf
   // copy_to_user(ubuf, buf, count) : buf --> ubuf; count=1 ==> put_ser()
   if( copy_to_user(ubuf, FPU_MEM, len) != 0 )
      return -EFAULT;

   // restore default
   memcpy(FPU_MEM, MEM_DFT, strlen(MEM_DFT));

   return count;
}

////////////////////////////////////////////////////////////////////////////////////////////////
static ssize_t fpu_write (struct file *filp, const char __user *ubuf, size_t count, loff_t *pos)
{
   uint8_t *kbuf;

   // make sure fresh write; otherwise run forever
   if( *pos )
      return 0;

   *pos += count;  // set by user

   // Create buffer for kernel to get data for writing
   kbuf = kmalloc(count, GFP_KERNEL);  // buffer in kernel

   if (!kbuf)
      return -ENOMEM;

   // transfer data from user to kernel for writing
   // get_user(val, ubuf) : ubuf --> val
   // copy_from_user(kbuf, ubuf, count) : ubuf --> kbuf; count=1 ==> get_user()
   if (copy_from_user(kbuf, ubuf, count))
   {
```

```
      kfree(kbuf);
      return -EFAULT;
   }
   FPU_MEM[count] = '\0'; // terminate string for using strlen()
   memcpy((char*)FPU_MEM, kbuf, count);
   DMSG("<< WRITE >> %s : count.%d, pos.%d, mem[%s]\n", fpu_name_id, count, (int)*pos, FPU_MEM);

   return count;
}

////////////////////////////////////////////////////////////////////////////////////////////////
static int   fpu_open (struct inode *inode, struct file *filp)
{
   DMSG("<< OPEN >> %s\n", fpu_name_id);

   return 0;
}

////////////////////////////////////////////////////////////////////////////////////////////////
static int   fpu_close (struct inode *inode, struct file *filp)
{
   DMSG("<< RLS >> %s \n", fpu_name_id);

   return 0;
}


////////////////////////////////////////////////////////////////////////////////////////////////

static int   fpu_ioctl (struct inode *inode, struct file *filp, uint cmd, ulong arg)
{
   DMSG("<< IOCTL >> %s \n", fpu_name_id);

   return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////////

static int   fpu_mmap(struct file *filp, struct vm_area_struct *vma)
{
   DMSG("<< MMAP >> %s \n", fpu_name_id);

   return 0;
}

////////////////////////////////////////////////////////////////////////////////////////////////
int register_device(void)
{
      int result = 0;

   result = register_chrdev( 0, fpu_name_id, &fpu_fops );
   if( result < 0 )
   {
      DMSG("<< INIT >> %s : ERR failed to get registered\n", fpu_name_id);
      return result;
   }

   fpu_major_id = result;
   DMSG("<< INIT >> %s : OK to get registered with major id %d\n", fpu_name_id, fpu_major_id);

   return 0;
}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////
void unregister_device(void)
{
   DMSG("<< EXIT >> %s : unregister_device\n", fpu_name_id);

   if(fpu_major_id)
   {
      unregister_chrdev(fpu_major_id, fpu_name_id);
      DMSG("<< EXIT >> %s : unregister_chrdev\n", fpu_name_id);
   }
}

////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////
static int fpu_init(void)
{
   DMSG("<< INIT >> %s \n", fpu_name_id);

    return register_device();
}
////////////////////////////////////////////////////////////////////////////////////////////////
static void fpu_exit(void)
{
   DMSG("<< EXIT >> %s \n", fpu_name_id);

    unregister_device();
}

module_init(fpu_init);
module_exit(fpu_exit);
```

3.1.2. Basic RW Script Test Result

```
# “make tst”

lldd fprw
Check and Load module fprw.ko
Check Major and Make dev node fprw c major 0
Major : 250
0 crwxrwxrwx   1 root root    250,   0 2015-06-21 21:23 fprw

->Devide READ
cat /dev/fprw
0x12345678, 0x01, 0x05062001
->Device WRITE
printf "Hello from User!\n" > /dev/fprw

->Device Read to check Device Write
cat /dev/fprw
Hello from User!

->Devide READ
cat /dev/fprw
0x12345678, 0x01, 0x05062001
```

The read back message is correct after write “`Hello from User!`”, and the following read gets back to default message correctly “`0x12345678, 0x01, 0x05062001`”

3.1.3. Basic RW Test

```
// t_fprw.c : user tst
// gcc fprw.c -o t_fprw

#include <stdio.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <string.h>

#define      DEV_ID "/dev/fprw"

int main()
{
       char buf[200];

       int fd = -1, len=0;
//     FILE *fd = NULL;

       char msg[] = "Hello from user space !";

       if ((fd = open(DEV_ID, O_RDWR)) < 0) {
              printf("<< ERR >> open %s\r", DEV_ID);
              return -1;
       }

       ////////////////////////////////////////////////////////////////////////////////////////
///

       len = read(fd, buf, 50);
       printf("<<READ 1 : Check Device Read>> len[%d] msg[%s]\n", len, buf);

       // back to head of file; otherwise failure in operation
       lseek(fd, 0, SEEK_SET);
       len = write(fd, msg, sizeof(msg));
       printf("<<WRITE>> len[%d] msg[%s]\n", len, msg);

       // back to head of file; otherwise failure in operation
       lseek(fd, 0, SEEK_SET);
       len = read(fd, buf, 50);
       printf("<<READ 2 : Check Device Write>> len[%d] msg[%s]\n", len, buf);

       // back to head of file; otherwise failure in operation
       lseek(fd, 0, SEEK_SET);
       len = read(fd, buf, 50);
       printf("<<READ 3 : Check Device Read>> len[%d] msg[%s]\n", len, buf);

       return 0;
}
```

3.1.4. Basic RW Test Result

```
<<READ 1>> len[50] msg[0x12345678, 0x01, 0x05062001]
<<WRITE>> len[24] msg[Hello from user space !]
<<READ 2>> len[50] msg[Hello from user space !]
<<READ 3>> len[50] msg[0x12345678, 0x01, 0x05062001]
```

The read back message is correct after write "`Hello from user space!`", and the following read gets back to default message correctly "`0x12345678, 0x01, 0x05062001`"

3.2.1. IOCTL Driver

```
// fpioctl.c : kernel driver
// RUN : make; make tst

#include <linux/module.h>
#include <linux/moduleparam.h>
#include <linux/init.h>

#include <linux/kernel.h> /* DMSG() */
#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 <linux/seq_file.h>
#include <linux/cdev.h>

#include <asm/system.h>   /* cli(), *_flags */
#include <asm/uaccess.h>  /* copy_*_user */

#include "fpu_ioctl.h"

////////////////////////////////////////////////////////////////////////////////////////////////

MODULE_LICENSE("Dual BSD/GPL");
MODULE_AUTHOR("Duy-Ky Nguyen");

#define      MEM_SIZE     0X1000
#define      READ_SIZE    0x20

#define      MEM_ID       0x12345678
#define      MEM_REV            0x01
#define      MEM_DATE     0x05062001

#define      MEM_DFT "0x12345678, 0x01, 0x05062001\0"
char FPU_MEM[MEM_SIZE] = MEM_DFT;

////////////////////////////////////////////////////////////////////////////////////////////////

#define FPU_DEBUG

#ifdef FPU_DEBUG
      #define DMSG(string, args...) printk(string, ##args)
#else
      #define DMSG(string, args...)
#endif

////////////////////////////////////////////////////////////////////////////////////////////////
// Start of device struct
static int   fpu_open (struct inode *, struct file *);
static int   fpu_close (struct inode *, struct file *);
static ssize_t fpu_read (struct file *, char *, size_t, loff_t *);
static ssize_t fpu_write (struct file *, const char *, size_t, loff_t *);

static int   fpu_ioctl (struct inode *, struct file *, uint, ulong);    // implemented
static int   fpu_mmap(struct file *filp, struct vm_area_struct *vma);   // just place holder
```

```
static struct file_operations fpu_fops = {
       open:  fpu_open,
       release: fpu_close,

       read:  fpu_read,    // copy_to_user(void __user * to, const void * from, unsigned long n);
       write: fpu_write,   // copy_from_user(void * to, const void __user * from, unsigned long n);

       ioctl: fpu_ioctl,
       mmap:  fpu_mmap,
};
// End of device struct

static int fpu_major_id = 0;

static const char fpu_name_id[] = "Dev_FPU";

////////////////////////////////////////////////////////////////////////////////////////////////

static ssize_t fpu_read(struct file *file_ptr, char __user *ubuf, size_t count, loff_t *pos)
{
       int len = strlen(FPU_MEM) + 1;   // extra 1 for '\0'
   if( *pos )
      return 0;

       // the whole buf used, update pos for next run to stop.
   *pos += len;     // count = 32K, by default

   // make sure fresh read; otherwise run forever
       DMSG("<< READ >> %s : count.%d pos.%d\n", fpu_name_id, count, (int)*pos);

   // transfer the whole buf from kernel to user for reading
   // put_user(val, ubuf) : val --> ubuf
   // copy_to_user(ubuf, buf, count) : buf --> ubuf; count=1 ==> put_ser()
   if( copy_to_user(ubuf, FPU_MEM, len) != 0 )
      return -EFAULT;

       // restore default
       memcpy(FPU_MEM, MEM_DFT, strlen(MEM_DFT));

   return count;
}

////////////////////////////////////////////////////////////////////////////////////////////////
static ssize_t fpu_write (struct file *filp, const char __user *ubuf, size_t count, loff_t *pos)
{
       uint8_t *kbuf;

       // make sure fresh write; otherwise run forever
       if( *pos )
              return 0;

       *pos += count;      // set by user

       // Create buffer for kernel to get data for writing
       kbuf = kmalloc(count, GFP_KERNEL);     // buffer in kernel

       if (!kbuf)
              return -ENOMEM;

       // transfer data from user to kernel for writing
```

```
	// get_user(val, ubuf) : ubuf --> val
	// copy_from_user(kbuf, ubuf, count) : ubuf --> kbuf; count=1 ==> get_user()
	if (copy_from_user(kbuf, ubuf, count))
	{
		kfree(kbuf);
		return -EFAULT;
	}
	FPU_MEM[count] = '\0';    // terminate string for using strlen()//
	memset((char*)FPU_MEM, 0, MEM_SIZE);
	memcpy((char*)FPU_MEM, kbuf, count);
	DMSG("<< WRITE >> %s : count.%d, pos.%d, mem[%s]\n", fpu_name_id, count,
		(int)*pos, FPU_MEM);

	return count;
}

////////////////////////////////////////////////////////////////////////////////////////////
static int   fpu_open (struct inode *inode, struct file *filp)
{
	DMSG("<< OPEN >> %s\n", fpu_name_id);

	return 0;
}

////////////////////////////////////////////////////////////////////////////////////////////
static int   fpu_close (struct inode *inode, struct file *filp)
{
	DMSG("<< RLS >> %s \n", fpu_name_id);

	return 0;
}


////////////////////////////////////////////////////////////////////////////////////////////

static int   fpu_ioctl (struct inode *inode, struct file *filp, uint cmd, ulong arg)
{
	int err = 0, len = strlen(FPU_MEM) + 1;// extra 1 for '\0'
	fpu_ioc_data ioc_data;

	DMSG("<< IOCTL 1 >> %s \n", fpu_name_id);

	/*
	* extract the type and number bitfields, and don't decode
	* wrong cmds: return ENOTTY (inappropriate ioctl) before access_ok()
	*/
	if (_IOC_TYPE(cmd) != FPU_IOC_MAGIC)
		return -ENOTTY;
	if (_IOC_NR(cmd) > FPU_IOC_MAGIC)
		return -ENOTTY;

	DMSG("<< IOCTL 2 >> type and number\n");
	/*
	* the direction is a bitmask, and VERIFY_WRITE catches R/W
	* transfers. `Type' is user-oriented, while
	* access_ok is kernel-oriented, so the concept of "read" and
	* "write" is reversed
	*/
	if (_IOC_DIR(cmd) & _IOC_READ)
		err = !access_ok(VERIFY_WRITE, (void *)arg, _IOC_SIZE(cmd));
	else if (_IOC_DIR(cmd) & _IOC_WRITE)
```

```
            err = !access_ok(VERIFY_READ,  (void *)arg, _IOC_SIZE(cmd));
      if (err)
            return -EFAULT;

      DMSG("<< IOCTL 3 >> size and access\n");

      switch(cmd) {

      case FPU_IOC_RD:
            // previous ioc_data content from FPU_IOC_WR is over-written FPU_IOC_RD
            // use copy_from_user() to get new ioc_data from FPU_IOC_RD, if so required
            DMSG("<<IOC_RD>> len.%x msg[%s]\n", len, FPU_MEM);
            if( copy_to_user((char *)arg, FPU_MEM, len) != 0 )
                  return -EFAULT;

            // restore default
            memcpy(FPU_MEM, MEM_DFT, strlen(MEM_DFT));

            break;
            ////////////////////////////////////////////////////////////////////

      case FPU_IOC_WR:
//            copy_from_user(&ioc_data, (char *)arg, sizeof(ioc_data));
            copy_from_user(&ioc_data, (char *)arg, sizeof(ioc_data));
            FPU_MEM[len] = '\0';      // terminate string for using strlen()
            memcpy((char*)FPU_MEM, ioc_data.msg, len);
            DMSG("<<IOC_WR>> len.%d msg[%s]\n", ioc_data.len, ioc_data.msg);
            break;
            ////////////////////////////////////////////////////////////////////

      default:  /* redundant, as cmd was checked against MAXNR */
            return -ENOTTY;
      }

      return 0;
}
////////////////////////////////////////////////////////////////////////////////////////////

static int   fpu_mmap(struct file *filp, struct vm_area_struct *vma)
{
      DMSG("<< MMAP >> %s \n", fpu_name_id);

      return 0;
}

////////////////////////////////////////////////////////////////////////////////////////////
int register_device(void)
{
     int result = 0;

      result = register_chrdev( 0, fpu_name_id, &fpu_fops );
      if( result < 0 )
      {
            DMSG("<< INIT >> %s : ERR failed to get registered\n", fpu_name_id);
            return result;
      }

      fpu_major_id = result;
      DMSG("<< INIT >> %s : OK to get registered with major id %d\n", fpu_name_id, fpu_major_id);

      return 0;
```

```
}
////////////////////////////////////////////////////////////////////////////////////////////////
void unregister_device(void)
{
	DMSG("<< EXIT >> %s : unregister_device\n", fpu_name_id);

	if(fpu_major_id)
	{
		unregister_chrdev(fpu_major_id, fpu_name_id);
		DMSG("<< EXIT >> %s : unregister_chrdev\n", fpu_name_id);
	}
}

////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////
static int fpu_init(void)
{
	DMSG("<< INIT >> %s \n", fpu_name_id);

	return register_device();
}
////////////////////////////////////////////////////////////////////////////////////////////////
static void fpu_exit(void)
{
	DMSG("<< EXIT >> %s \n", fpu_name_id);

	unregister_device();
}

module_init(fpu_init);
module_exit(fpu_exit);
```

3.2.2. IOCTL Test

There's no Linux command for IOCTL

```
// t_fpioctl.c : user tst
// gcc t_fpioctl.c -o t_fpioctl

#include <stdio.h>
#include <sys/ioctl.h>
#include <fcntl.h>
#include <string.h>

#include "fpu_ioctl.h"

#define	DEV_ID "/dev/fpioctl"

int main()
{
	char buf[200];
	int fd = -1, len=0;
	char msg[] = "Hello from User !";
	fpu_ioc_data ioc_data;

	if ((fd = open(DEV_ID, O_RDWR)) < 0) {
		printf("<< ERR >> open %s\r", DEV_ID);
		return -1;
	}

	////////////////////////////////////////////////////////////////////////////////////////
```

```
	if(ioctl(fd, FPU_IOC_RD, buf) < 0)
		printf("<< IOC_READ  1>> ERR\n");

	len = strlen(buf);
	printf("<<IOC_READ 1 >> Check Device Read : len[%d] msg[%s]\n", len, buf);

	////////////////////////////////////////////////////////////////////////////////

	// Setup IOCTL
	ioc_data.len = strlen(msg);
	ioc_data.msg = msg;

	if(ioctl(fd, FPU_IOC_WR, &ioc_data) < 0)
		printf("<< IOC_WRITE >> ERR\n");

	printf("<<IOC_WRITE >> USR : len.%d msg[%s]\n", ioc_data.len, ioc_data.msg);

	////////////////////////////////////////////////////////////////////////////////

	if(ioctl(fd, FPU_IOC_RD, buf) < 0)
		printf("<< IOC_READ  2>> ERR\n");

	len = strlen(buf);
	printf("<<IOC_READ 2 >> Check Device Write : len[%d] msg[%s]\n", len, buf);

	////////////////////////////////////////////////////////////////////////////////

	if(ioctl(fd, FPU_IOC_RD, buf) < 0)
		printf("<< IOC_READ  3>> ERR\n");

	len = strlen(buf);
	printf("<<IOC_READ 3 >> Check Device Read : len[%d] msg[%s]\n", len, buf);

	return 0;
}
```

3.2.3. IOCTL Test Result

```
<<IOC_READ 1 >> Check Device Read : len[28] msg[0x12345678, 0x01, 0x05062001]
<<IOC_WRITE >> USR : len.17 msg[Hello from User !]
<<IOC_READ 2 >> Check Device Write : len[17] msg[Hello from User !]
<<IOC_READ 3 >> Check Device Read : len[28] msg[0x12345678, 0x01, 0x05062001]
```

The read back message is correct after write “`Hello from User!`”, and the following read gets back to default message correctly “`0x12345678, 0x01, 0x05062001`”