Creating an Ioctl command in linux

Ioctl which stand for Input Output control is a system call used in linux to implement system calls which are not be available in the kernel by default.

The major use of this is in case of handling some specific operations of a device for which the kernel does not have a system call by default. For eg: Ejecting the media from a "cd" drive.An ioctl command is implemented to give the eject system call to the cd drive.

Note: The following code is only valid for 2.6 kernels below 2.6.36.
For versions above 2.6.36 refer to the post "Implementing ioctl call for kernel versions above 2.6.36
To implement a new ioctl command we need to follow the following steps.
1. Define the ioctl code in a header file and include the same in the application as well as the module.
The definition is done as follows
#define "ioctl name" __IOX("magic number","command number","argument type")

where *IOX* can be :
"IO": If the command neither reads any data from the user nor writes any data to the userspace.
"IOW": If the commands needs to write some to the kernel space.
"IOR": If the command needs to read some thing from the kernel space.
"IOWR": If the command does both read as well as write from the user

The Magic Number is a unique number or character that will differentiate our set of ioctl calls from the other ioctl calls. some times the major number for the device is used here.

Command Number is the number that is assigned to the ioctl .It is this number that is used to differentiate the commands from one another .

The last is the type of data that will be written in case of __IOW or read in case of __IOR or both read as well as write in case of __IOWR. In the case of _IO we need not pass any thing.

2. Add the header file linux/ioctl.h to make use of the above mentioned calls.

Let us call the ioctl that we will create as "IOCTL_HELLO" , hence the header file , ioctl_basic.h, would be

#include <linux/ioctl.h> #define IOC_MAGIC k // defines the magic number #define IOCTL_HELLO _IO(IOC_MAGIC,0) // defines our ioctl call

3. The next step is to implement the ioctl call we defined in to the corresponding driver. First we will need to #include the header file *ioctl_basic.h*

Then we need to add the ioctl function which has the prototype

int ioctl(struct inode *inode,struct file *filp,unsigned int cmd,unsigned long arg)

Where
ionde :is the inode number of the file being worked on
filp : is the file pointer to the file that was passed by the application.
cmd : is the ioctl command that was called from the user space.
arg : are the arguments passed from the user space.

With in the function "ioctl" we need to implement all the commands that we define in the header file.As we saw each command is given a command number in the header file, the same number will be used in a switch case statement to implement the calls.
We will just add a print statement to see how the ioctl call works. so the function would look as follows

int ioctl_funcs(struct inode *inode,struct file *filp,unsigned int cmd,unsigned long arg)
{ switch(cmd) { case IOCTL_HELLO: printk(KERN_INFO "In the ioctl command");
 break; default: printk(KERN_INFO "Wrong command");
 return 1; } return 0; }

In the above function the ioctl command "IOCTL_HELLO" will have got the number 0,that we assigned in the header file *ioctl_basic.h*, The same number would be passed in the argument "cmd" when the ioctl call is made from the user space. Thus using "cmd" in the switch case makes sure that the correct command get executed for the ioctl call.

4. Next step is to inform the kernel that the ioctl calls are implmented in the function "our_ioctl". This is done by making the fops pointer "ioctl" to point to "our_ioctl" as shown below

Note: You can read about fops at "fops implementation"

struct file_operations fops = { open: open ioctl: ioctl_funcs, //Mapping the ioctl function release: release, };

Now we need to call the new ioctl command from a user application, to test its working.

The call to an ioctl in the user space i.e in an application looks as follows

int ioctl("file descriptor","ioctl command","Arguments")

file descriptor : This the open file on which the ioctl command needs to be executed, which would generally be device files
ioctl command : ioctl command which is implemented to achieve the desired functionality
arguments: The arguments that needs to be passed to the ioctl command.

Thus is our case the call would look as below int ioctl(fd,IOCTL_HELLO,NULL)

We need to "#include" the header file "ioctl_basic.h" in the user space program too.

The following are the full codes for the header file,the module and the user space application.

Header file: ioctl_basic.h

#include <linux/ioctl.h> #define IOC_MAGIC 'k' #define IOCTL_HELLO _IO(IOC_MAGIC,0)

Module: ioctl_basic.c

```
#include <linux/module.h>
#include <linux/kernel.h>
 #include <linux/fs.h> // required for various structures related to files liked fops.
#include <linux/semaphore.h>
#include <linux/cdev.h>
#include "ioctl_basic.h" //ioctl header file
#include <linux/version.h>
static int Major; int open(struct inode *inode, struct file *filp)
{
printk(KERN_INFO "Inside open \n");
 return 0;
}
int release(struct inode *inode, struct file *filp)
{
 printk (KERN_INFO "Inside close \n");
 return 0;
}
int ioctl_funcs(struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg)
 {
int data=10,ret;
switch(cmd)
```

```
{
case IOCTL_HELLO: printk(KERN_INFO "Hello ioctl world");
break;
}
return ret;
 }
struct file_operations fops = {
                open: open,
                ioctl: ioctl_funcs,
                release: release
                };
struct cdev *kernel_cdev;
int char_arr_init (void)
{
        int ret;
        dev_t dev_no,dev;
        kernel_cdev = cdev_alloc();
        kernel_cdev->ops = &fops;
        kernel_cdev->owner = THIS_MODULE;
        printk (" Inside init module\n");
        ret = alloc_chrdev_region( &dev_no , 0, 1,"char_arr_dev");
        if (ret < 0)
        {
                printk("Major number allocation is failed\n");
                return ret;
        }
         Major = MAJOR(dev_no);
        dev = MKDEV(Major,0);
        printk (" The major number for your device is %d\n", Major);
        ret = cdev_add( kernel_cdev,dev,1);
        if(ret < 0 )
        {
                printk(KERN_INFO "Unable to allocate cdev");
                return ret;
        }
        return 0;
}
 void char_arr_cleanup(void)
{
        printk(KERN_INFO " Inside cleanup_module\n");
        cdev_del(kernel_cdev);
        unregister_chrdev_region(Major, 1);
}
MODULE_LICENSE("GPL");
```

```
module_init(char_arr_init);
module_exit(char_arr_cleanup);
```

User space appliction: user_basic_ioctl.c

```
#include <stdio.h>
#include <fcntl.h>
#include "ioctl_basic.h" //ioctl header file
main ( )
{
        int fd;
        fd = open("/dev/temp", O_RDWR);
        if (fd == -1)
        {
                printf("Error in opening file \n");
                exit(-1);
        }
        ioctl(fd,IOCTL_HELLO); //ioctl call close(fd);
}
```

Now compile the module using the following makefile

```
ifneq ($(KERNELRELEASE),)
obj-m := ioctl_basic.o
else
        KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
        $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
endif
```

Note : To know more about compiling a module refer to "Inserting into the kernel" and "user access"

$ make $ sudo insmod ioctl_basic.ko $ dmesg (To find the major number) $ sudo mknod /dev/temp c <major number got by dmesg> 0 $ cc user_app.c -o user_app $ ./user_app $ dmesg {Other logs} Hello ioctl world

The message that was printed in the output of *dmesg* command is the same message that we have incorporated in the *ioctl* call implementation in the driver, hence proving that the call from the user space was passed on to the kernel space successfully.

Q. Exporting symbols from module ?

Linux kernel allows modules stacking, which basically means one module can use the symbols

defined in other modules. But this is possible only when the symbols are exported.

- *let us see how can a module export a symbol and how another module can make use of it.*

A symbol can be exported by a module using the macro EXPORT_SYMBOL().

Let us make use of the very basic hello world module. In this module we have added function called "hello_export" and used the EXPORT_SYMBOL macro to export this function.

hello_export.c

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
static int hello_init(void)
{
 printk(KERN_INFO "Hello,world");
 return 0;
}
static hello_export(void) {
printk(KERN_INFO "Hello from another module");
return 0;
}
static void hello_exit(void)
{
 printk(KERN_INFO "Goodbye cruel world");
}
EXPORT_SYMBOL(hello_export);
module_init(hello_init);
module_exit(hello_exit);
```

Makefile required to compile this module is

Makefile

```
ifneq ($(KERNELRELEASE),)
  obj-m := hello_export.o
else
KERNELDIR ?= /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
   $(MAKE) -C $(KERNELDIR) M=$(PWD) modules
clean:
   $(MAKE) -C $(KERNELDIR) M=$(PWD) clean
endif
```

.....

Compile it using the "make" command and then insert it into the kernel using insmod

$make

$insmod hello_export.ko

All the symbols that the kernel is aware of is listed in /proc/kallsyms. Let us search for our symbol in this file.

$ cat /proc/kallsyms | grep hello_export d09c4000 T hello_export[hello_export]

From the output we can see that the symbol we exported is listed in the symbols recognized by the kernel. The first column gives the address of the column and the last column the module to which the symbol belongs to.

If you look into the Module.symvers file in directory where the module was compiled you will see a line similar to the following .

0x06577ca7 hello_export /home/user/hello EXPORT_SYMBOL

Now let us write another module which will make use of the exported symbol.

To make use of the symbol we just have to inform the module that the symbols being used is defined externally by using the "extern" keyword. Thus to use the hello_export function we will have to use

extern hello_export();

Once this is done we can use the function any where in the module, as shown in the module below.

hello_use_export.c

```
#include <linux/init.h>
#include <linux/module.h>
MODULE_LICENSE("Dual BSD/GPL");
extern hello_export();
static int hello_init(void)
{
        printk(KERN_INFO "Hello,world");
        hello_export();
        return 0;
}
static void hello_exit(void)
{
        printk(KERN_INFO "Goodbye cruel world");
}
module_init(hello_init);
module_exit(hello_exit);
```

Use the same Makefile as shown above only change the module name to hello_use_export.o

Compile and load as done before.

$ make

$ insmod hello_use_export.ko

If the function has got called successfully we should see the message "Hello from another module" in the kernel logs.

$dmesg | tail -5 [22706.056990] Hello,world [23662.239681] Hello,world [23662.239689] Hello from another module

Thus we could call the function defined in another module successfully once it was exported by the module.

The dependecy of hello_use_export on hello_export can be found using the lsmod command

$ lsmod |grep hello hello_use_export 590 0 hello_export 683 1 hello_use_export

We know the modprobe command can load the module that is passed to it as well as the dependent modules, hence we can also see the working of modprobe using these modules.

Copy the two .ko files, hello_export.ko and hello_use_export.ko to /lib/modules/`uname -r`/

$ cp hello_export.ko /lib/modules/`uname -r`/. $ cp hello_use_export.ko /lib/modules/`uname -r`/.

To use modprobe we have to update the dependencies in the modules.dep file in /lib/modules/`uname -r`/modules.dep . This can be done using the

command depmod.

$ depmod -a $ cat /lib/modules/2.6.32-5-686/modules.dep | grep hello hello_use_export.ko: hello_export.ko hello_export.ko:

The output "hello_use_export.ko: hello_export.ko" signifies that hello_use_export is

dependent on hello_export.

Now remove the two modules we inserted before

$ rmmod hello_use_export $ rmmod hello_export

Note: You will not be able to remove the modules in the reverse order cause hello_use_export will be using hello_export. Hence unless hello_use_export is not removed, hello_export can not be removed.

Now use the modprobe command to load hello_use_export, and we will see that hello_export gets loaded automatically.

$ modprobe hello_use_export

$ lsmod |grep hello hello_use_export 590 0 hello_export 683 1 hello_use_export

We can see that even though we did not load the module "hello_export" it got loaded automatically because "hello_use_export" was dependent on it.

EXPORT_SYMBOL(name);

EXPORT_SYMBOL_GPL(name);

“Either of the above macros makes the given symbol available outside the module.”

“The _GPL version makes the symbol available to GPL-licensed modules only. Symbols must be exported in the global part of the module’s file, outside of any function, because the macros expand to the declaration of a special-purpose variable that is expected to be accessible globally.”

“This variable is stored in a special part of the module executible (an “ELF section”) that is used by the kernel at load time to find the variables exported by the module.”

Probe function

Probe is called *when your device is recognized by the platform*.Generally registering the device requires the d*evice name* and this device name is compared

with the driver name(when you are registering the driver you would be specifying *the driver name,which should match with the device name*).

> Once the device gets detected by the kernel it will call the corresponding driver probe for that device.

Kernel-assisted probing

The Linux kernel offers a low-level facility for probing the interrupt number. It works for only nonshared interrupts, but most hardware that is capable of working in a shared interrupt mode provides better ways of finding the configured interrupt number anyway. The facility consists of two functions, declared in <linux/interrupt.h> (which also describes the probing machinery):

unsigned long probe_irq_on(void);

This function returns a bit mask of unassigned interrupts. The driver must preserve the returned bit mask, and pass it to probe_irq_off later. After this call, the driver should arrange for its device to generate at least one interrupt.

int probe_irq_off(unsigned long);

After the device has requested an interrupt, the driver calls this function, passing as its argument the bit mask previously returned by probe_irq_on. probe_irq_off *returns the number of the interrupt that was issued after "probe_on*." If no interrupts occurred, **0** is returned (therefore, IRQ 0 can't be probed for, but no custom device can use it on any of the supported architectures anyway). If more than one interrupt occurred (***ambiguous detection***), probe_irq_off returns a ***negative*** **value.**