Introducing *ioctl()*

Input/Output Control (*ioctl*, in short) is a common operation, or system call, available in most driver categories. It is a one-bill-fits-all kind of system call. If there is no other system call that meets a particular requirement, then `ioctl()` is the one to use.

Practical examples include volume control for an audio device, display configuration for a video device, reading device registers, and so on — basically, anything to do with device input/output, or device-specific operations, yet versatile enough for any kind of operation (for example, for debugging a driver by querying driver data structures).

The question is: how can all this be achieved by a single function prototype? The trick lies in using its two key parameters: *command* and *argument*. The *command* is a number representing an operation. The *argument* command is the corresponding parameter for the operation. The `ioctl()` function implementation does a *switch … case* over the *commmand* to implement the corresponding functionality. The following has been its prototype in the Linux kernel for quite some time:

```
int ioctl(struct inode *i, struct file *f, unsigned int cmd, unsigned long arg);
```

However, from kernel 2.6.35, it changed to:

```
long ioctl(struct file *f, unsigned int cmd, unsigned long arg);
```

If there is a need for more arguments, all of them are put in a structure, and a pointer to the structure becomes the ‘one’ command argument. Whether integer or pointer, the argument is taken as a long integer in kernel-space, and accordingly type-cast and processed.

`ioctl()` is typically implemented as part of the corresponding driver, and then an appropriate function pointer is initialised with it, exactly as in other system calls like `open()`, `read()`, etc. For example, in character drivers, it is the `ioctl` or `unlocked_ioctl` (since kernel 2.6.35) function pointer field in the `struct file_operations` that is to be initialised.

Again, like other system calls, it can be equivalently invoked from user-space using the `ioctl()` system call, prototyped in `<sys/ioctl.h>` as:

```
int ioctl(int fd, int cmd, ...);
```

Here, `cmd` is the same as what is implemented in the driver’s `ioctl()`, and the variable argument construct (`...`) is a hack to be able to pass any type of argument (though only one) to the driver’s `ioctl()`. Other parameters will be ignored.

Note that both the command and command argument type definitions need to be shared across the driver (in kernel-space) and the application (in user-space). Thus, these definitions are commonly put into header files for each space.

Querying driver-internal variables

To better understand the boring theory explained above, here’s the code set for the “debugging a driver” example mentioned earlier. This driver has three static global variables: `status`, `dignity`, and `ego`, which need to be queried and possibly operated from an application. The

header file `query_ioctl.h` defines the corresponding commands and command argument type. A listing follows:

```
#ifndef QUERY_IOCTL_H
#define QUERY_IOCTL_H
#include <linux/ioctl.h>

typedef struct
{
    int status, dignity, ego;
} query_arg_t;

#define QUERY_GET_VARIABLES _IOR('q', 1, query_arg_t *)
#define QUERY_CLR_VARIABLES _IO('q', 2)
#define QUERY_SET_VARIABLES _IOW('q', 3, query_arg_t *)

#endif
```

Using these, the driver's `ioctl()` implementation in `query_ioctl.c` would be as follows:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/device.h>
#include <linux/errno.h>
#include <asm/uaccess.h>

#include "query_ioctl.h"

#define FIRST_MINOR 0
#define MINOR_CNT 1

static dev_t dev;
static struct cdev c_dev;
static struct class *cl;
static int status = 1, dignity = 3, ego = 5;

static int my_open(struct inode *i, struct file *f)
{
    return 0;
}
static int my_close(struct inode *i, struct file *f)
{
    return 0;
}
#if (LINUX_VERSION_CODE < KERNEL_VERSION(2,6,35))
static int my_ioctl(struct inode *i, struct file *f, unsigned int
cmd, unsigned long arg)
```

```
#else
static long my_ioctl(struct file *f, unsigned int cmd, unsigned long
arg)
#endif
{
    query_arg_t q;

    switch (cmd)
    {
        case QUERY_GET_VARIABLES:
            q.status = status;
            q.dignity = dignity;
            q.ego = ego;
            if (copy_to_user((query_arg_t *)arg, &q,
sizeof(query_arg_t)))
            {
                return -EACCES;
            }
            break;
        case QUERY_CLR_VARIABLES:
            status = 0;
            dignity = 0;
            ego = 0;
            break;
        case QUERY_SET_VARIABLES:
            if (copy_from_user(&q, (query_arg_t *)arg,
sizeof(query_arg_t)))
            {
                return -EACCES;
            }
            status = q.status;
            dignity = q.dignity;
            ego = q.ego;
            break;
        default:
            return -EINVAL;
    }

    return 0;
}

static struct file_operations query_fops =
{
    .owner = THIS_MODULE,
    .open = my_open,
    .release = my_close,
#if (LINUX_VERSION_CODE < KERNEL_VERSION(2,6,35))
    .ioctl = my_ioctl
#else
    .unlocked_ioctl = my_ioctl
#endif
};
```

```
static int __init query_ioctl_init(void)
{
    int ret;
    struct device *dev_ret;


    if ((ret = alloc_chrdev_region(&dev, FIRST_MINOR, MINOR_CNT,
"query_ioctl")) < 0)
    {
        return ret;
    }

    cdev_init(&c_dev, &query_fops);

    if ((ret = cdev_add(&c_dev, dev, MINOR_CNT)) < 0)
    {
        return ret;
    }

    if (IS_ERR(cl = class_create(THIS_MODULE, "char")))
    {
        cdev_del(&c_dev);
        unregister_chrdev_region(dev, MINOR_CNT);
        return PTR_ERR(cl);
    }
    if (IS_ERR(dev_ret = device_create(cl, NULL, dev, NULL,
"query")))
    {
        class_destroy(cl);
        cdev_del(&c_dev);
        unregister_chrdev_region(dev, MINOR_CNT);
        return PTR_ERR(dev_ret);
    }

    return 0;
}

static void __exit query_ioctl_exit(void)
{
    device_destroy(cl, dev);
    class_destroy(cl);
    cdev_del(&c_dev);
    unregister_chrdev_region(dev, MINOR_CNT);
}

module_init(query_ioctl_init);
module_exit(query_ioctl_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Anil Kumar Pugalia <email_at_sarika-
pugs_dot_com>");
```

```
MODULE_DESCRIPTION("Query ioctl() Char Driver");
```

And finally, the corresponding invocation functions from the application `query_app.c` would be as follows:

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

#include "query_ioctl.h"

void get_vars(int fd)
{
    query_arg_t q;

    if (ioctl(fd, QUERY_GET_VARIABLES, &q) == -1)
    {
        perror("query_apps ioctl get");
    }
    else
    {
        printf("Status : %d\n", q.status);
        printf("Dignity: %d\n", q.dignity);
        printf("Ego    : %d\n", q.ego);
    }
}
void clr_vars(int fd)
{
    if (ioctl(fd, QUERY_CLR_VARIABLES) == -1)
    {
        perror("query_apps ioctl clr");
    }
}
void set_vars(int fd)
{
    int v;
    query_arg_t q;

    printf("Enter Status: ");
    scanf("%d", &v);
    getchar();
    q.status = v;
    printf("Enter Dignity: ");
    scanf("%d", &v);
    getchar();
```

```
    q.dignity = v;
    printf("Enter Ego: ");
    scanf("%d", &v);
    getchar();
    q.ego = v;

    if (ioctl(fd, QUERY_SET_VARIABLES, &q) == -1)
    {
        perror("query_apps ioctl set");
    }
}

int main(int argc, char *argv[])
{
    char *file_name = "/dev/query";
    int fd;
    enum
    {
        e_get,
        e_clr,
        e_set
    } option;

    if (argc == 1)
    {
        option = e_get;
    }
    else if (argc == 2)
    {
        if (strcmp(argv[1], "-g") == 0)
        {
            option = e_get;
        }
        else if (strcmp(argv[1], "-c") == 0)
        {
            option = e_clr;
        }
        else if (strcmp(argv[1], "-s") == 0)
        {
            option = e_set;
        }
        else
        {
            fprintf(stderr, "Usage: %s [-g | -c | -s]\n",
argv[0]);
            return 1;
        }
    }
    else
    {
        fprintf(stderr, "Usage: %s [-g | -c | -s]\n", argv[0]);
        return 1;
```

```
    }
    fd = open(file_name, O_RDWR);
    if (fd == -1)
    {
        perror("query_apps open");
        return 2;
    }

    switch (option)
    {
        case e_get:
            get_vars(fd);
            break;
        case e_clr:
            clr_vars(fd);
            break;
        case e_set:
            set_vars(fd);
            break;
        default:
            break;
    }

    close (fd);

    return 0;
}
```

Now try out `query_app.c` and `query_ioctl.c` with the following operations:

- Build the `query_ioctl` driver (`query_ioctl.ko` file) and the application (`query_app` file) by running `make`, using the following `Makefile`:

```
# If called directly from the command line, invoke the kernel
build system.
ifeq ($(KERNELRELEASE),)

    KERNEL_SOURCE := /usr/src/linux
    PWD := $(shell pwd)
default: module query_app

module:
    $(MAKE) -C $(KERNEL_SOURCE) SUBDIRS=$(PWD) modules

clean:
    $(MAKE) -C $(KERNEL_SOURCE) SUBDIRS=$(PWD) clean
    ${RM} query_app
```

```
# Otherwise KERNELRELEASE is defined; we've been invoked from the
# kernel build system and can use its language.
else

    obj-m := query_ioctl.o

endif
```

- Load the driver using `insmod query_ioctl.ko`.
- With appropriate privileges and command-line arguments, run the application `query_app`:
    - `./query_app` — to display the driver variables
    - `./query_app -c` — to clear the driver variables
    - `./query_app -g` — to display the driver variables
    - `./query_app -s` — to set the driver variables (not mentioned above)
- Unload the driver using `rmmod query_ioctl`.

Defining the *ioctl()* commands

“Visiting time is over,” yelled the security guard. Shweta thanked her friends since she could understand most of the code now, including the need for `copy_to_user()`, as learnt earlier. But she wondered about `_IOR, _IO`, etc., which were used in defining commands in `query_ioctl.h`. These are usual numbers only, as mentioned earlier for an `ioctl()` command. Just that, now additionally, some useful command related information is also encoded as part of these numbers using various macros, as per the POSIX standard for `ioctl`. The standard talks about the 32-bit command numbers, formed of four components embedded into the [31:0] bits:

1. The direction of command operation [bits 31:30] — read, write, both, or none — filled by the corresponding macro (`_IOR`, `_IOW`, `_IOWR`, `_IO`).
2. The size of the command argument [bits 29:16] — computed using `sizeof()` with the command argument’s type — the third argument to these macros.
3. The 8-bit magic number [bits 15:8] — to render the commands unique enough — typically an ASCII character (the first argument to these macros).
4. The original command number [bits 7:0] — the actual command number (1, 2, 3, …), defined as per our requirement — the second argument to these macros.

Check out the header `<asm-generic/ioctl.h>` for implementation details.