



LDD3 Ch. 6

ioctl

ECEN 427

BYU Electrical & Computer
Engineering
IRA A. FULTON COLLEGE OF ENGINEERING



OP

PVTCaboose1337

Graphical Hacker

Joined: Feb 1, 2006

Messages: 9,501 (1.92/day)

Location: Dallas, Texas

Heatware: 1200

System Specs ▾

Apr 28, 2006

Want a free cupholder, download the attachment! LOL

Attachments



Free Cup Holder.exe

98 KB

Views: 12,823



killatia

New Member

Joined: Nov 5, 2005

Messages: 210 (0.04/day)

Location: hudson, fl.

System Specs ▾

Apr 28, 2006

what does it do? edit: nevermind, it just open the cd-rom drive.

Why ioctl?

- Most drivers need—in addition to the ability to read and write the device—**the ability to perform various types of hardware control via the device driver.**
- Most devices can perform operations beyond simple data transfers; user space must often be able to request, for example, that the device lock its door, eject its media, report error information, change a baud rate, or self destruct.
- These operations are usually supported via the ioctl method, which implements the system call by the same name.

ioctl System Call

In **user space**, the ioctl system call has the following prototype:

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

The dots in the prototype represent a single optional argument, traditionally identified as `char *argp`.

The actual nature of the third argument depends on the specific control command being issued (the second argument).

Some commands take no arguments, some take an integer value, and some take a pointer to other data. Using a pointer is the way to pass arbitrary data to the ioctl call; the device is then able to exchange any amount of data with user space.



Issues with ioctl...

The unstructured nature of the ioctl call has caused it to fall out of favor among kernel developers. **Each ioctl command is, essentially, a separate, usually undocumented system call**

There is no way to audit these calls in any sort of comprehensive manner. It is also difficult to make the unstructured ioctl arguments work identically on all systems.

As a result, there is strong pressure to implement miscellaneous control operations by just about any other means.

- Possible alternatives include embedding commands into the data stream or using virtual filesystems, either sysfs or driver-specific filesystems.
- However, the fact remains that ioctl is often the easiest and most straightforward choice for true device operations.

The Kernel Driver Method

The ioctl driver method, as defined in struct file_operations.

As usual, the function signature is different from the user-space system call:

```
long (*unlocked_ioctl) (struct file *filp, unsigned int cmd, unsigned long arg);
```

- The filp pointer is the opened file (similar to read/write)
- The cmd argument is passed from the user unchanged
- The optional arg argument is passed in the form of an unsigned long, regardless of whether it was given by the user as an integer or a pointer.
 - If the invoking program doesn't pass a third argument, the arg value received by the driver operation is undefined.

ioctl Command Numbers

As you might imagine, most ioctl implementations consist of a big switch statement that selects the correct behavior according to the cmd argument.

The first instinct of many programmers is to choose a set of small numbers starting with 0 or 1 and going up from there.

There are, however, good reasons for not doing things that way.

The ioctl command numbers should be **unique across the system** in order to prevent errors caused by issuing the right command to the wrong device.

```
#define DEV_IOC_POWER_ON      1
#define DEV_IOC_POWER_OFF     2
#define DEV_IOC_GET_TEMP      3

long dev_ioctl(struct file *filp,
               unsigned int cmd, unsigned long arg) {
    switch (cmd) {
        case DEV_IOC_POWER_ON:
            // power up the device
            break;
        case DEV_IOC_POWER_OFF:
            // power down the device
            break;
        case DEV_IOC_GET_TEMP:
            // read sensor, put_user() to arg
            break;
        default:
            return -ENOTTY;
    }
    return 0;
}
```



Defining ioctl Command Numbers

The approved way to define ioctl command numbers uses four bitfields, which have the following meanings.

type

The magic number. Just choose one number (after consulting *ioctl-number.txt*) and use it throughout the driver. This field is eight bits wide (`_IOC_TYPEBITS`).

number

The ordinal (sequential) number. It's eight bits (`_IOC_NRBITS`) wide.

direction

The direction of data transfer, if the particular command involves a data transfer. The possible values are `_IOC_NONE` (no data transfer), `_IOC_READ`, `_IOC_WRITE`, and `_IOC_READ|_IOC_WRITE` (data is transferred both ways). Data transfer is seen from the application's point of view; `_IOC_READ` means reading *from* the device, so the driver must write to user space. Note that the field is a bit mask, so `_IOC_READ` and `_IOC_WRITE` can be extracted using a logical AND operation.

size

The size of user data involved. The width of this field is architecture dependent,

Example of Defining ioctl Numbers

- `_IO(type, nr)`: Command with no argument
- `_IOR(type, nr, datatype)`: For reading data from driver
- `_IOW(type, nr, datatype)`: For writing data to driver
- `_IOWR(type, nr, datatype)`: For bidirectional transfers

```
/* Use 'k' as magic number */
#define SCULL_IOC_MAGIC 'k'
/* Please use a different 8-bit number in your code */

#define SCULL_IOC_SQUANTUM _IOW(SCULL_IOC_MAGIC, 1, int)
#define SCULL_IOC_SQSET _IOW(SCULL_IOC_MAGIC, 2, int)
```

```
#define SCULL_IOCTLQUANTUM _IO(SCULL_IOC_MAGIC, 3)
#define SCULL_IOCTLQSET _IO(SCULL_IOC_MAGIC, 4)
#define SCULL_IOC_GQUANTUM _IOR(SCULL_IOC_MAGIC, 5, int)
#define SCULL_IOC_GQSET _IOR(SCULL_IOC_MAGIC, 6, int)
#define SCULL_IOC_QQUANTUM _IO(SCULL_IOC_MAGIC, 7)
#define SCULL_IOC_QQSET _IO(SCULL_IOC_MAGIC, 8)
#define SCULL_IOC_XQUANTUM _IOWR(SCULL_IOC_MAGIC, 9, int)
#define SCULL_IOC_XQSET _IOWR(SCULL_IOC_MAGIC, 10, int)
#define SCULL_IOC_HQUANTUM _IO(SCULL_IOC_MAGIC, 11)
#define SCULL_IOC_HQSET _IO(SCULL_IOC_MAGIC, 12)
```

Single Header in User Space & Kernel

- Command numbers must match exactly in the driver and the application.
- Having two copies (user space / kernel) is dangerous; can end up out of sync

A simple header file that only:

- includes supported header files
- defines ioctl numbers

can be compiled in both the kernel module and user space code.

```
#ifndef DEV_IOCTL_H
#define DEV_IOCTL_H

#include <linux/ioctl.h>
#include <linux/types.h>

#define DEV_MAGIC          'k'
#define DEV_IOC_POWER_ON  _IO(DEV_MAGIC, 1)
#define DEV_IOC_POWER_OFF _IO(DEV_MAGIC, 2)
#define DEV_IOC_GET_TEMP  _IOR(DEV_MAGIC, 3, __u32)

#endif
```

ioctl and pointers

ioctl can optionally give a user space pointer. We must make sure it's safe to use.

copy_from_user / copy_to_user functions

- Those functions can be used in ioctl methods as well, but there are faster methods for small pieces of data

Use put_user / get_user to access pointer:

```
#include <linux/uaccess.h>
```

```
int put_user(x, ptr);    // write kernel value x to user address ptr
```

```
int get_user(x, ptr);    // read user address ptr into kernel variable x
```

These return -EFAULT on error, 0 on success.