**DRIVERS BASICS**

WHAT IS DRIVER? Why driver?

* The software that handles or manages a hardware controller is known as a device **driver**. The **Linux** kernel device **drivers** are, essentially, a shared library of privileged, memory resident, low level hardware handling routines.
* In computing, a **device driver** is a computer **program** that operates or controls a particular type of **device** that is attached to a computer. ... **Drivers** are **hardware** dependent and operating-system-specific. They usually provide the interrupt handling required for any necessary asynchronous time-dependent **hardware** interface.
* A **driver** is software that allows your computer to communicate with hardware or devices.
* A device driver (often referred to as driver’) is a piece of software that controls a particular type of device which is connected to the computer system. It provides a software interface to the hardware device, and enables access to the operating system and other applications. There are various types of drivers present in GNU/Linux such as Character, Block, Network and USB drivers.
* In this column, we will explore only character drivers.  
  Character drivers are the most common drivers. They provide unbuffered, direct access to hardware devices. One can think of character drivers as a long sequence of bytes — same as regular files but can be accessed only in sequential order. Character drivers support at least the open(), close(), read() and write() operations. The text console, i.e., /dev/console, serial consoles /dev/stty\*, and audio/video drivers fall under this category.  
  To make a device usable there must be a driver present for it. So let us understand how an application accesses data from a device with the help of a driver. We will discuss the following four major entities.
* User-space application: This can be any simple utility like echo, or any complex application.
* Device file: This is a special file that provides an interface for the driver. It is present in the file system as an ordinary file. The application can perform all supported operation on it, just like for an ordinary file. It can move, copy, delete, rename, read and write these device files.
* Device driver: This is the software interface for the device and resides in the kernel space.
* Device: This can be the actual device present at the hardware level, or a pseudo device.

Why clock used?

In synchronous communication used “synchronous” data bus, which means that it uses separate lines for data and a “clock” that keeps both sides in perfect sync. The clock is an oscillating signal that tells the receiver exactly when to sample the bits on the data line. This could be the rising (low to high) or falling (high to low) edge of the clock signal; the datasheet will specify which one to use. When the receiver detects that edge, it will immediately look at the data line to read the next bit (see the arrows in the below diagram). Because the clock is sent along with the data, specifying the speed isn’t important, although devices will have a top speed at which they can operate (We’ll discuss choosing the proper clock edge and speed in a bit).

# **IOCTL System call:**

IOCTL is referred as Input and Output Control, which is used to talking to device drivers. This system call, available in most driver categories.  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.

Some real time applications of ioctl is Ejecting the media from a “cd” drive, to change the Baud Rate of Serial port, Adjust the Volume, Reading or Writing device registers, etc. We already have write and read function in our device driver. But it is not enough for all cases.

# **Steps involved in IOCTL**

There are some steps involved to use IOCTL.

* Create IOCTL command in driver
* Write IOCTL function in driver
* Create IOCTL command in User space application
* Use IOCTL system call in User space

A **magic number** is a **number** embedded at or near the beginning of a file that indicates its file format (i.e., the type of file it is). It is also sometimes referred to as a file signature. **Magic numbers** are generally not visible to users.

## **Create IOCTL Command in Driver**

To implement a new ioctl command we need to follow the following steps.

1. Define the ioctl code

#define "ioctlname" \_\_IOX("magic number","command number","argument type")

where *IOX* can be :  
“IO”: an ioctl with no parameters  
“IOW”: an ioctl with write parameters (copy\_from\_user)  
“IOR”: an ioctl with read parameters (copy\_to\_user)  
“IOWR”: an ioctl with both write and read parameters

* 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 .This is used to differentiate the commands from one another.
* The last is the type of data.

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

Example:

#include <linux/ioctl.h> #define WR\_VALUE \_IOW('a','a',int32\_t\*) #define RD\_VALUE \_IOR('a','b',int32\_t\*)

|  |  |
| --- | --- |
| 1  2  3  4 | #include <linux/ioctl.h>    #define WR\_VALUE \_IOW('a','a',int32\_t\*)  #define RD\_VALUE \_IOR('a','b',int32\_t\*) |

## **Write IOCTL function in driver**

The next step is to implement the ioctl call we defined in to the corresponding driver. We need to add the ioctl function which has the prototype.

Where  
<file>   : 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 defined above. We need to use the same commands in switch statement which is defined above.

Then need to inform the kernel that the ioctl calls are implemented in the function “etx\_ioctl”. This is done by making the fops pointer “unlocked\_ioctl” to point to “etx\_ioctl” as shown below.

Example:

static long etx\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg) { switch(cmd) { case WR\_VALUE: copy\_from\_user(&value ,(int32\_t\*) arg, sizeof(value)); printk(KERN\_INFO "Value = %d\n", value); break; case RD\_VALUE: copy\_to\_user((int32\_t\*) arg, &value, sizeof(value)); break; } return 0; } static struct file\_operations fops = { .owner = THIS\_MODULE, .read = etx\_read, .write = etx\_write, .open = etx\_open, .unlocked\_ioctl = etx\_ioctl, .release = etx\_release, };

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24 | static long etx\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)  {  switch(cmd) {  case WR\_VALUE:                          copy\_from\_user(&value ,(int32\_t\*) arg, sizeof(value));                          printk(KERN\_INFO "Value = %d\n", value);                          break;                  case RD\_VALUE:                          copy\_to\_user((int32\_t\*) arg, &value, sizeof(value));                          break;          }          return 0;  }      static struct file\_operations fops =  {  .owner = THIS\_MODULE,  .read = etx\_read,  .write = etx\_write,  .open = etx\_open,  .unlocked\_ioctl = etx\_ioctl,  .release = etx\_release,  }; |

Now we need to call the new ioctl command from a user application.

## **Create IOCTL command in User space application**

Just define the ioctl command like how we defined in driver.

Example:

|  |  |
| --- | --- |
| 1  2 | #define WR\_VALUE \_IOW('a','a',int32\_t\*)  #define RD\_VALUE \_IOR('a','b',int32\_t\*) |

## **Use IOCTL system call in User space**

Include the header file <sys/ioctl.h>.Now we need to call the new ioctl command from a user application.

long 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.

Example:

|  |  |
| --- | --- |
| 1  2  3 | ioctl(fd, WR\_VALUE, (int32\_t\*) &number);    ioctl(fd, RD\_VALUE, (int32\_t\*) &value); |

Now we will see the complete driver and application.

# **Device Driver Source Code**

In this example we only implemented IOCTL. In this driver, i defined one variable (int32\_t value). Using ioctl command we can read or change the variable. So other functions like open, close, read, write, We simply left empty. Just go through the code below