**ioctl (input/output control)** is a system call that enables user-space applications to **communicate directly with device drivers** in the kernel space. It is commonly used for **device-specific operations** not covered by standard system calls like read, write, or open. Example: Turning on/off an LED, ejecting a CD-ROM, querying hardware status, etc.

In **user space**, the function prototype looks like:

int ioctl(int fd, unsigned long request, ...);

* fd: File descriptor obtained from open() system call.
* request: Request code that tells the driver which operation to perform.
* ...: Optional argument(s) passed to the driver.

**Step 1: User-space to Kernel-space Transition (System Call Entry):** When a user-space application calls ioctl(), the following happens:

int fd = open("/dev/mydevice", O\_RDWR);

ioctl(fd, MY\_IOCTL\_COMMAND, &data);

* The above ioctl() call in user-space triggers **software interrupt (SWI)** to switch from **user space to kernel space**.

**System call table entry**

* The system call for ioctl is defined in **syscall\_64.tbl**:

ioctl sys\_ioctl

* The function sys\_ioctl() in the kernel gets invoked.

**Step 2: sys\_ioctl() in Kernel**

The function prototype for sys\_ioctl() looks like this:

SYSCALL\_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)

{

struct file \*file;

int error;

file = fget(fd);

if (!file)

return -EBADF;

error = file->f\_op->unlocked\_ioctl(file, cmd, (unsigned long)arg);

fput(file);

return error;

}

**Explanation:** fget(fd) fetches the file structure from the file descriptor. It then invokes the function pointer file->f\_op->unlocked\_ioctl(). This function is provided by the driver and is responsible for executing the ioctl operation. The kernel uses the file structure linked to your device:

file -> inode -> struct cdev -> struct device -> driver

* **file** → User-space file descriptor.
* **inode** → Represents the device node (/dev/mydevice).
* **cdev** → Represents the character device (cdev\_add() in driver).
* **device** → Represents the physical device (struct device).
* **driver** → The actual device driver (mydriver).

The kernel directly jumps to your driver’s unlocked\_ioctl(). In PCIe device drivers, the file structure (struct file) is linked to the PCIe device (struct pci\_dev). The flow looks like this:

file -> inode -> struct cdev -> struct device -> struct pci\_dev

The PCIe driver declares:

static struct pci\_driver my\_pci\_driver = {

.name = "mydevice",

.id\_table = mydevice\_table,

.probe = my\_probe,

.remove = my\_remove,

};

The probe() is called during device insertion. The driver binds:

pci\_register\_driver(&my\_pci\_driver);

**Step 3: Flow from VFS to Driver (file\_operations):** The file\_operations structure in the driver must implement the unlocked\_ioctl callback:

static long my\_ioctl(struct file \*file, unsigned int cmd, unsigned long arg)

{

switch (cmd) {

case MY\_IOCTL\_COMMAND:

// Handle the IOCTL command

break;

default:

return -ENOTTY;

}

return 0;

}

**File Operations Binding:**

static const struct file\_operations my\_fops = {

.owner = THIS\_MODULE,

.unlocked\_ioctl = my\_ioctl,

};

The kernel calls my\_ioctl() directly when the user invokes ioctl() from user-space.

**Step 4: Decoding the ioctl Number (cmd)**

The cmd (request code) in ioctl is defined as:

#define MY\_IOCTL\_COMMAND \_IOR('M', 1, int)

The macro \_IOR() is defined in <linux/ioctl.h> as:

#define \_IOR(type,nr,size) \_IOC(\_IOC\_READ,(type),(nr),(\_IOC\_TYPECHECK(size)))

This macro generates a **32-bit ioctl number** containing:

* **Type:** Identifies the driver type (M for mydevice).
* **Number:** Command number (1).
* **Direction:** Read/Write operation.
* **Size:** Size of the data structure.

The driver can decode it using:

switch (\_IOC\_TYPE(cmd)) {

case 'M': // Handle our device commands

break;

}

**Step 5: Data Transfer Between User and Kernel Space**

**For Input (User → Kernel):** User space sends data to the kernel like:

ioctl(fd, MY\_IOCTL\_COMMAND, &data);

In driver:

copy\_from\_user(&kdata, (void \_\_user \*)arg, sizeof(data)); copy\_from\_user() ensures secure memory transfer from user-space to kernel-space.

**For Output (Kernel → User)**

If the driver wants to send data back:

copy\_to\_user((void \_\_user \*)arg, &kdata, sizeof(data)); copy\_to\_user() securely copies data from kernel-space to user-space.

**Step 6: Memory Mapping (if applicable)**

If the ioctl involves large continuous memory (like framebuffer, PCIe BAR space, etc.), the driver can **map physical memory to user space**:

int my\_mmap(struct file \*file, struct vm\_area\_struct \*vma)

{

return remap\_pfn\_range(vma,

vma->vm\_start,

phys\_addr >> PAGE\_SHIFT,

vma->vm\_end - vma->vm\_start,

vma->vm\_page\_prot);

}

This bypasses copy\_to\_user() and maps hardware memory directly to user-space for high-speed access.

**Step 7: Unlocking and Error Handling**

The ioctl handler typically returns:

* **0:** Success.
* **-EINVAL:** Invalid argument.
* **-ENOTTY:** Invalid ioctl command.
* **-EFAULT:** Invalid memory access.

Example:

if (copy\_from\_user(&kdata, (void \_\_user \*)arg, sizeof(data)))

return -EFAULT;

**Step 8: Closing the File Descriptor**

Once done, the user closes the device:

close(fd); This triggers release() callback in driver:

static int my\_release(struct inode \*inode, struct file \*file)

{

// Free resources

return 0;

}

**Step 9: ioctl Flow Summary**

Here’s a complete high-level flow:

|  |  |  |
| --- | --- | --- |
| **Step** | **Flow** | **Responsible Code** |
| 1 | ioctl(fd, cmd, arg) | User-space application |
| 2 | sys\_ioctl() | Kernel system call |
| 3 | file->f\_op->unlocked\_ioctl() | File operations |
| 4 | my\_ioctl() in driver | Driver-specific logic |
| 5 | copy\_from\_user() | Move data from user to kernel |
| 6 | copy\_to\_user() | Move data from kernel to user |
| 7 | close(fd) | Trigger release() in driver |

**Bonus: ioctl in PCIe, Platform, and Character Drivers**

* **PCIe Driver:**
  + ioctl is mostly used for **BAR memory access, DMA setup, and hot-plug events**.
* **Platform Driver:**
  + ioctl can configure **platform-specific hardware** like GPIO, I2C, or SPI.
* **Character Driver:**
  + ioctl directly controls devices like **UART, touch screen, LEDs, etc.**

**Common Mistakes in ioctl**

1. **Directly dereferencing user-space pointers:** Always use copy\_from\_user() and copy\_to\_user().
2. **Ignoring alignment issues:** Misaligned memory in ioctl leads to crashes.
3. **Improper ioctl number:** Use \_IOR(), \_IOW(), \_IORW() macros correctly.

**Hotplug ioctls Example** Suppose you want to trigger **device reset** during hotplug:User-space:

ioctl(fd, MY\_IOCTL\_RESET\_DEVICE);

Driver:

case MY\_IOCTL\_RESET\_DEVICE:

pci\_reset\_function(pdev);

This completely resets the PCIe device.

**1. mmap with ioctl for PCIe BAR** Suppose you want to map the PCIe device’s memory (BAR) to user space: User-space:

mmap(fd, 0, len, PROT\_READ | PROT\_WRITE, MAP\_SHARED, 0);

Driver:

static int my\_mmap(struct file \*file, struct vm\_area\_struct \*vma)

{

struct pci\_dev \*pdev = file->private\_data;

unsigned long start = pci\_resource\_start(pdev, 0);

unsigned long len = pci\_resource\_len(pdev, 0);

return remap\_pfn\_range(vma,

vma->vm\_start,

start >> PAGE\_SHIFT,

len,

vma->vm\_page\_prot);

}

This directly maps BAR memory to user-space.

**2. DMA Transfer with ioctl**

Suppose you want the device to write data to host memory using DMA: User-space:

ioctl(fd, MY\_IOCTL\_START\_DMA);

Driver:

case MY\_IOCTL\_START\_DMA:

dma\_addr\_t dma\_addr = dma\_map\_single(&pdev->dev, buf, size, DMA\_FROM\_DEVICE);

// Initiate DMA from device to host memory

**3. Hotplug Reset with ioctl:** User-space:

ioctl(fd, MY\_IOCTL\_RESET\_DEVICE);

Driver:

case MY\_IOCTL\_RESET\_DEVICE:

pci\_reset\_function(pdev);

break;

The PCIe device will physically reset.