Systems Programming

Devices

H. Turgut Uyar Şima Uyar

2001-2014

Topics

I/O Subsystem
- Introduction
- Device Types
- I/O Software
- Accessing Devices

Device Drivers
- Interface
- Implementation
- Device Access

Topics

I/O Subsystem
- Introduction
- Device Types
- I/O Software
- Accessing Devices

Device Drivers
- Interface
- Implementation
- Device Access

I/O Devices

- O/S controls all I/O devices
- issues commands to devices
- catches interrupts
- handles errors
- provides interface

Device Controllers

- devices consist of:
  - mechanical components
  - electronic components: *device controller*
- O/S deals with controller
  - connected through a standard interface
  - SCSI, USB, Firewire, ...

Controller Registers

- CPU communicates with the controller through registers
- data register: for sending/receiving data
- control register: for sending commands to device
- status register: for getting/setting the state of device

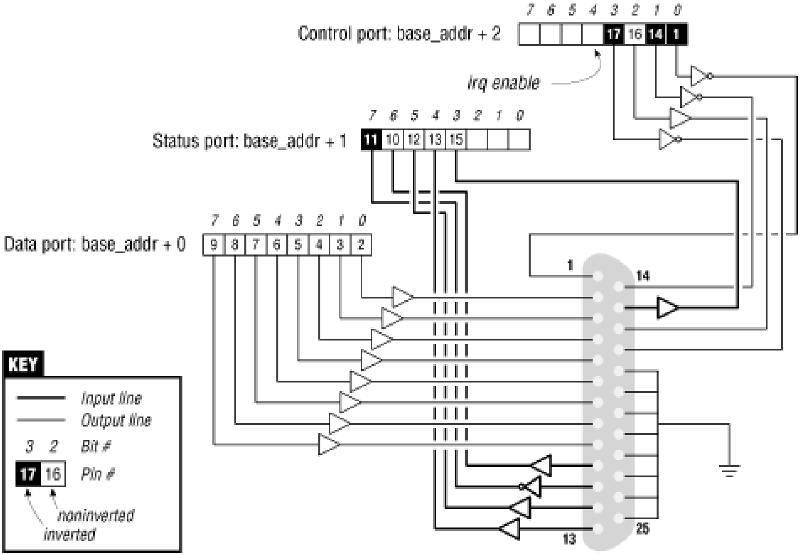
I/O Architecture

- ports: special address space for I/O
  - separate lines for I/O ports
  - special instructions for I/O
- memory-mapped: registers part of regular address space
  - directly-mapped: part of address space reserved for I/O
  - software-mapped: I/O space part of virtual memory

PC Parallel Interface

- parallel interface base addresses on a PC: 0x378, 0x278
- ports:
  - +0: bidirectional data register
  - +1: status register (read-only)
    online, out-of-paper, busy
  - +2: control register (write-only)
    enable/disable interrupts

Parallel Interface



Topics

I/O Subsystem

- Introduction
- Device Types
- I/O Software
- Accessing Devices

Device Drivers

- Interface
- Implementation
- Device Access

Device Types

- character devices
- block devices
- network interfaces
- clocks and timers

Character Devices

- a character device acts like a stream of characters
- arbitrary-sized data transfer
- not addressable: no seek operation

examples

- console, mouse
- sound card
- serial port, parallel port

Block Devices

- a block device can host a filesystem
- data transfer in fixed-size blocks
- each block has its own address
- read/write each block independently

example

- disks

Device Type

- the device type is more the characteristic of the driver rather than the device itself

example: disk

- usually a block device
- it can also be used as a character device: tar

Topics

I/O Subsystem
- Introduction
- Device Types
- I/O Software
- Accessing Devices

Device Drivers
- Interface
- Implementation
- Device Access

I/O Software

- blocking vs interrupt-driven
  - better for CPU to work interrupt-driven fashion
  - better for user-space programs to work in blocking fashion
  - easier to develop programs that work in blocking fashion
  - O/S makes interrupt-driven operations look blocking
- standardized interface
- uniform naming

I/O Software

- blocking vs interrupt-driven
  - better for CPU to work interrupt-driven fashion
  - better for user-space programs to work in blocking fashion
  - easier to develop programs that work in blocking fashion
  - O/S makes interrupt-driven operations look blocking
- standardized interface
- uniform naming

Unix Device Naming

- in Unix, every device has a device node
- under the `/dev` folder
- `/dev/sda`: first SCSI disk
- `/dev/sdb`: second SCSI disk
- `/dev/sdb1`: first partition of the second SCSI disk
- `/dev/sdb2`: second partition of the second SCSI disk
- `/dev/parport0`: first parallel port

Unix Device Naming

- in Unix, every device has a device node
- under the `/dev` folder

- `/dev/sda`: first SCSI disk
- `/dev/sdb`: second SCSI disk
- `/dev/sdb1`: first partition of the second SCSI disk
- `/dev/sdb2`: second partition of the second SCSI disk

- `/dev/parport0`: first parallel port

Unix Device Naming

- in Unix, every device has a device node
- under the `/dev` folder

- `/dev/sda`: first SCSI disk
- `/dev/sdb`: second SCSI disk
- `/dev/sdb1`: first partition of the second SCSI disk
- `/dev/sdb2`: second partition of the second SCSI disk

- `/dev/parport0`: first parallel port

Unix Device Naming

- device nodes have major and minor numbers
- major number identifies the driver
- minor number identifies the physical device

- all /dev/sd* devices have the same major number
- they all have different minor numbers

- (recently) major number alone doesn't identify driver
- major number + region of minor numbers

Unix Device Naming

- device nodes have major and minor numbers
- major number identifies the driver
- minor number identifies the physical device

- all `/dev/sd*` devices have the same major number
- they all have different minor numbers

- (recently) major number alone doesn't identify driver
- major number + region of minor numbers

Unix Device Naming

- device nodes have major and minor numbers
- major number identifies the driver
- minor number identifies the physical device

- all `/dev/sd*` devices have the same major number
- they all have different minor numbers

- (recently) major number alone doesn't identify driver
- major number + region of minor numbers

I/O Services

- copy semantics: transfer the snapshot of data at the time of the I/O request
- scheduling: issue order may not be the best execution order
- buffering: adapt between different data transfer sizes
- caching
- spooling: deal with dedicated devices (e.g. printers)
  - a daemon for controlling the device
  - a spooling directory
- error handling

I/O Services

- copy semantics: transfer the snapshot of data at the time of the I/O request
- scheduling: issue order may not be the best execution order
- buffering: adapt between different data transfer sizes
- caching
- spooling: deal with dedicated devices (e.g. printers)
  - a daemon for controlling the device
  - a spooling directory
- error handling

I/O Services

- copy semantics: transfer the snapshot of data
  at the time of the I/O request
- scheduling: issue order may not be the best execution order
- buffering: adapt between different data transfer sizes
- caching
- spooling: deal with dedicated devices (e.g. printers)
  - a daemon for controlling the device
  - a spooling directory
- error handling

I/O Services

- copy semantics: transfer the snapshot of data at the time of the I/O request
- scheduling: issue order may not be the best execution order
- buffering: adapt between different data transfer sizes
- caching
- spooling: deal with dedicated devices (e.g. printers)
  - a daemon for controlling the device
  - a spooling directory
- error handling

I/O Services

- copy semantics: transfer the snapshot of data
  at the time of the I/O request
- scheduling: issue order may not be the best execution order
- buffering: adapt between different data transfer sizes
- caching
- spooling: deal with dedicated devices (e.g. printers)
  - a daemon for controlling the device
  - a spooling directory
- error handling

I/O Services

- copy semantics: transfer the snapshot of data
  at the time of the I/O request
- scheduling: issue order may not be the best execution order
- buffering: adapt between different data transfer sizes
- caching
- spooling: deal with dedicated devices (e.g. printers)
  - a daemon for controlling the device
  - a spooling directory
- error handling

Software Layers

- top-down:

- user-space applications
- device-independent software
- device drivers
- interrupt handlers

Device-Independent Software

- functions common to all devices
- uniform interface to user-level software
- device naming
- device protection
- provide device-independent block sizes
- buffering
- allocating and releasing dedicated devices
- error reporting

Device Drivers

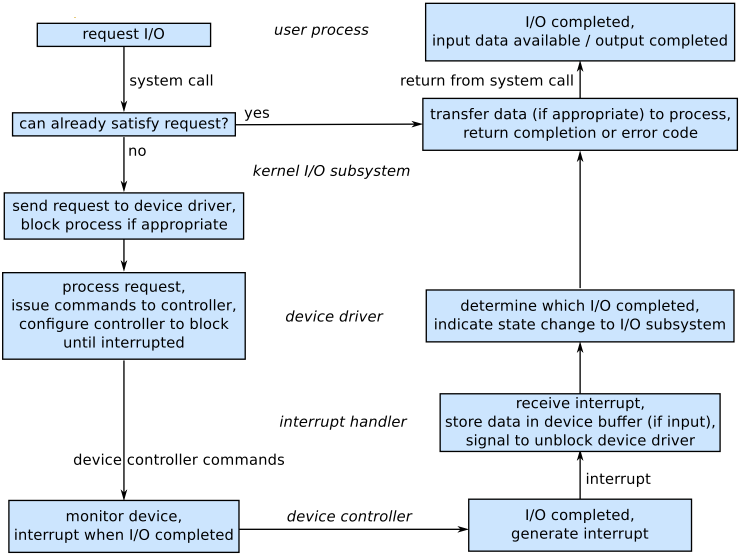
- device-dependent code
- a driver for each device type

- accept request from device-independent software
- decide on sequence of controller operations

Interrupt Handlers

- interrupts hidden from rest of system
  - requesting process is blocked until I/O is completed
- when I/O is completed, interrupt occurs
  - process is made to unblock

I/O Life Cycle



Topics

I/O Subsystem
- Introduction
- Device Types
- I/O Software
- Accessing Devices

Device Drivers
- Interface
- Implementation
- Device Access

Accessing Devices

- directly: using ports or memory
- through device drivers: using the device driver interface

Direct Access

- input: `inb`, `inw`, `inl`
- output: `outb`, `outw`, `outl`
- get permission from O/S: `ioperm` system call

```
int ioperm(unsigned long from,
           unsigned long num,
           int turn_on);
```

Direct Access

- input: `inb`, `inw`, `inl`
- output: `outb`, `outw`, `outl`

- get permission from O/S: `ioperm` system call

```
int ioperm(unsigned long from,
           unsigned long num,
           int turn_on);
```

Direct Access Example

output to parallel interface

```
ioperm(0x378, 1, 255);
outb(0xff, 0x378);
```

Reading Material

- Silberschatz, 8/e
  - Chapter 13: I/O Systems

Topics

I/O Subsystem
- Introduction
- Device Types
- I/O Software
- Accessing Devices

Device Drivers
- Interface
- Implementation
- Device Access

Unix Device Driver Interface

- in Unix, the device driver interface
  is similar to the file interface
- `open`, `close`
- `read`, `write`

Device Specific Operations

- some operations are neither `read` nor `write`
- `ioctl`: issue command specific to device

device-specific operation examples

- eject CDROM
- make the speaker beep
- set communication parameters for modem

System Calls

- open:

```
int open(const char *pathname,
         int flags,
         mode_t mode);
```

- flags:
  - O_RDONLY O_WRONLY O_RDWR
  - O_CREAT O_APPEND
- mode: permissions
- returns: file descriptor

System Calls

- `close`:

```c
int close(int fd);
```

- returns: success / failure status

System Calls

- `read`:

```
ssize_t read(int fd,
             void *buf,
             size_t count);
```

- returns: number of bytes read (x)
  - $x = count$: successful completion
  - $x = 0$: end-of-file
  - $x < 0$: error
  - $0 < x < count$: partial transfer, retry remaining part

System Calls

- `write`:

```
ssize_t write(int fd,
              const void *buf,
              size_t count);
```

- returns: number of bytes written (x)
  - $x = count$: successful completion
  - $x = 0$: end-of-file
  - $x < 0$: error
  - $0 < x < count$: partial transfer, retry remaining part

System Calls

- `ioctl`:

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

- parameter and return values depend on request

Device Access Example

output to parallel port

```
fd = open("/dev/parport0", O_WRONLY);
if (fd == -1)
{
    perror("cannot access device");
    exit(EXIT_FAILURE);
}
write(fd, buffer, len);
close(fd);
```

Device Specific Command Example

make the speaker beep

```
fd = open("/dev/console", O_RDWR);
status = ioctl(fd, KDMKTONE, 0x100011AA);
if (status == -1)
{
    perror("cannot generate beep");
    exit(EXIT_FAILURE);
}
close(fd);
```

Topics

I/O Subsystem

- Introduction
- Device Types
- I/O Software
- Accessing Devices

Device Drivers

- Interface
- Implementation
- Device Access

Implementing Device Drivers

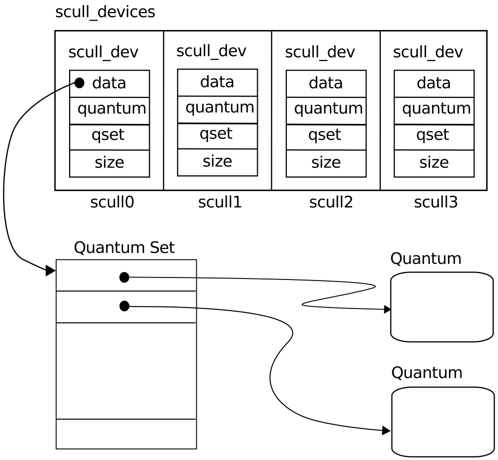
- implement system calls for device
- convert system calls to device specific I/O instructions

Device Driver Example

simplified scull

- use memory as device
  - `/dev/scull0`
  - `/dev/scull1`
- each device can hold data up to a limit
  - data persists during module's lifetime

Memory Layout



- each device has a quantum set
- each quantum contains the actual data
- memory is allocated as data is written

Global Definitions

scull.h

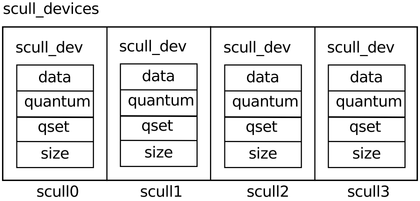
```
#define SCULL_MAJOR 0
#define SCULL_NR_DEVS 4
#define SCULL_QUANTUM 4000
#define SCULL_QSET 1000
```

Module Parameters

```c
int scull_major = SCULL_MAJOR;
int scull_minor = 0;
int scull_nr_devs = SCULL_NR_DEVS;
int scull_quantum = SCULL_QUANTUM;
int scull_qset = SCULL_QSET;

module_param(scull_major, int, S_IRUGO);
module_param(scull_minor, int, S_IRUGO);
module_param(scull_nr_devs, int, S_IRUGO);
module_param(scull_quantum, int, S_IRUGO);
module_param(scull_qset, int, S_IRUGO);
```

Data Structures




```c
struct scull_dev {
    char **data;
    int quantum;
    int qset;
    unsigned long size;
    struct semaphore sem;
    struct cdev cdev;
};

struct scull_dev *scull_devices;
```

Module Initialization

- allocate I/O region
  - base address
  - number of ports
- register driver with the kernel
  - major and minor numbers
  - capabilities: file operations

Module Initialization

driver registration: major and minor numbers

```
if (scull_major)
{
    devno = MKDEV(scull_major, scull_minor);
    result = register_chrdev_region(devno,
                scull_nr_devs, "scull");
}
else    /* dynamic */
{
    result = alloc_chrdev_region(&devno,
        scull_minor, scull_nr_devs, "scull");
    scull_major = MAJOR(devno);
}
```

Module Initialization

data structure allocation

```
scull_devices = kmalloc(scull_nr_devs *
       sizeof(struct scull_dev), GFP_KERNEL);
if (!scull_devices)
{
    result = -ENOMEM;
    goto fail;
}
memset(scull_devices, 0,
       scull_nr_devs * sizeof(struct scull_dev));
```

File Operations

- map system calls to functions:
  `struct file_operations`

```
struct file_operations scull_fops = {
    .open    = scull_open,
    .release = scull_release,
    .read    = scull_read,
    .write   = scull_write,
    .llseek  = scull_llseek,
    .ioctl   = scull_ioctl,
};
```

Module Initialization

driver activation

```
for (i = 0; i < scull_nr_devs; i++)
{
    dev = &scull_devices[i];
    dev->quantum = scull_quantum;
    dev->qset = scull_qset;
    init_MUTEX(&dev->sem);

    devno = MKDEV(scull_major, scull_minor + i);
    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    cdev_add(&dev->cdev, devno, 1);
}
```

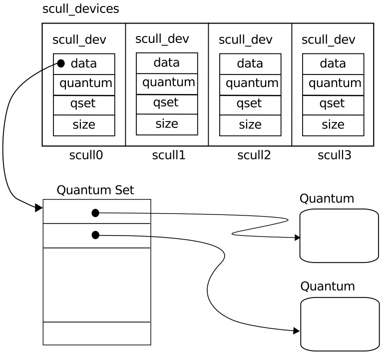
Module Cleanup

```c
dev_t devno = MKDEV(scull_major, scull_minor);

if (scull_devices)
{
    for (i = 0; i < scull_nr_devs; i++)
    {
        scull_trim(scull_devices + i);
        cdev_del(&scull_devices[i].cdev);
    }
    kfree(scull_devices);
}

unregister_chrdev_region(devno, scull_nr_devs);
```

Module Cleanup



data structure deallocation

```
if (dev->data)
{
    for (i = 0; i < dev->qset;
         i++)
    {
        if (dev->data[i])
            kfree(dev->data[i]);
    }
    kfree(dev->data);
}
dev->data = NULL;
dev->quantum = scull_quantum;
dev->qset = scull_qset;
dev->size = 0;
```

Kernel Data Structures

- a structure for each device node:
  `struct inode`
- a structure for each open file:
  `struct file`
  - `f_mode`: readable, writable, both
  - `f_pos`: current reading/writing position
  - `f_flags`
  - `f_op`: operations associated with the file
  - `private_data`: pointer to allocated data

Open

- identify actual device
- check for device-specific errors
- initialize device
- allocate and initialize data structures

Kernel System Call Interface

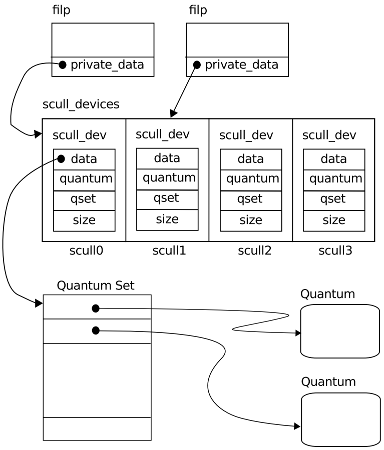
- open system call:

```
int open(const char *pathname,
         int flags,
         mode_t mode);
```

- kernel function to implement:

```
int foo_open(struct inode *inode,
             struct file *filp);
```

Open




```
struct scull_dev *dev;

dev = container_of(
        inode->i_cdev,
        struct scull_dev,
        cdev
);
filp->private_data = dev;
```

Kernel System Call Interface

- `close` system call:

```
int close(int fd);
```

- kernel function to implement:

```
int foo_release(struct inode *inode,
                struct file *filp);
```

Kernel System Call Interface

- `write` system call:

```
ssize_t write(int fd,
              const void *buf,
              size_t count);
```

- kernel function to implement:

```
ssize_t foo_write(struct file *filp,
                  const char __user *buf,
                  size_t count,
                  loff_t *f_pos);
```

Write

```
  struct scull_dev *dev = filp->private_data;
  ssize_t retval = -ENOMEM;

  if (down_interruptible(&dev->sem))
      return -ERESTARTSYS;

  /* determine position */
  /* allocate quantum if necessary */
  /* copy from user space */
  /* update size */

out:
  up(&dev->sem);
  return retval;
```

Write

determine position

```
int quantum = dev->quantum, qset = dev->qset;
int s_pos, q_pos;

if (*f_pos >= quantum * qset)
{
    retval = 0;
    goto out;
}

s_pos = (long) *f_pos / quantum;
q_pos = (long) *f_pos % quantum;
```

Write

allocate quantum if necessary

```
if (!dev->data)
{
    dev->data = kmalloc(qset * sizeof(char *),
                        GFP_KERNEL);
    if (!dev->data)
        goto out;
    memset(dev->data, 0, qset * sizeof(char *));
}
if (!dev->data[s_pos])
{
    dev->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
    if (!dev->data[s_pos])
        goto out;
}
```

Write

copy from user space

```
/* adjust write amount */
if (count > quantum - q_pos)
    count = quantum - q_pos;

if (copy_from_user(dev->data[s_pos] + q_pos,
                   buf, count))
{
    retval = -EFAULT;
    goto out;
}
```

Write

update size

```
*f_pos += count;
retval = count;

if (dev->size < *f_pos)
    dev->size = *f_pos;
```

Kernel System Call Interface

- read system call:

```
ssize_t read(int fd,
             void *buf,
             size_t count);
```

- kernel function to implement:

```
ssize_t foo_read(struct file *filp,
                 char __user *buf,
                 size_t count,
                 loff_t *f_pos);
```

Read

```
  struct scull_dev *dev = filp->private_data;
  ssize_t retval = 0;

  if (down_interruptible(&dev->sem))
      return -ERESTARTSYS;

  /* determine position */
  /* copy to user space */

out:
  up(&dev->sem);
  return retval;
```

Read

determine position

```c
int quantum = dev->quantum;
int s_pos, q_pos;

if (*f_pos >= dev->size)
    goto out;
if (*f_pos + count > dev->size)
    count = dev->size - *f_pos;

s_pos = (long) *f_pos / quantum;
q_pos = (long) *f_pos % quantum;

if (dev->data == NULL || ! dev->data[s_pos])
    goto out;
```

Reading from the Device

copy to user space

```
/* adjust read amount */
if (count > quantum - q_pos)
    count = quantum - q_pos;

if (copy_to_user(buf, dev->data[s_pos] + q_pos, count))
{
    retval = -EFAULT;
    goto out;
}
*f_pos += count;
retval = count;
```

Kernel System Call Interface

- `lseek` system call:

```
off_t lseek(int fd,
            off_t offset,
            int whence);
```

- kernel function to implement:

```
loff_t foo_llseek(struct file *filp,
                  loff_t off,
                  int whence);
```

Seek

calculate new position

```
switch(whence)
{
    case 0: /* SEEK_SET */
        newpos = off;
        break;
    case 1: /* SEEK_CUR */
        newpos = filp->f_pos + off;
        break;
    case 2: /* SEEK_END */
        newpos = dev->size + off;
        break;
    default: /* can't happen */
        return -EINVAL;
}
```

Seek

set new position

```
if (newpos < 0)
    return -EINVAL;
filp->f_pos = newpos;
return newpos;
```

Kernel System Call Interface

- `ioctl` system call:

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

- kernel function to implement:

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

Device-Specific Commands

- SCULL_IOCRESET:
  assign default values to quantum set size and quantum size
- SCULL_IOCSQUANTUM: set quantum size from pointer
- SCULL_IOCTQUANTUM: (tell) set quantum size from value
- SCULL_IOCGQUANTUM: get quantum size to pointer
- SCULL_IOCQQUANTUM: (query) return quantum size
- SCULL_IOCXQUANTUM: (exchange) set + get
- SCULL_IOCHQUANTUM: (shift) tell + query
- similar operations for quantum set size

Device-Specific Commands

- `SCULL_IOCRESET`:
  assign default values to quantum set size and quantum size
- `SCULL_IOCSQUANTUM`: set quantum size from pointer
- `SCULL_IOCTQUANTUM`: (tell) set quantum size from value
- `SCULL_IOCGQUANTUM`: get quantum size to pointer
- `SCULL_IOCQQUANTUM`: (query) return quantum size
- `SCULL_IOCXQUANTUM`: (exchange) set + get
- `SCULL_IOCHQUANTUM`: (shift) tell + query
- similar operations for quantum set size

Device-Specific Commands

- `SCULL_IOCRESET`:
  assign default values to quantum set size and quantum size
- `SCULL_IOCSQUANTUM`: set quantum size from pointer
- `SCULL_IOCTQUANTUM`: (tell) set quantum size from value
- `SCULL_IOCGQUANTUM`: get quantum size to pointer
- `SCULL_IOCQQUANTUM`: (query) return quantum size
- `SCULL_IOCXQUANTUM`: (exchange) set + get
- `SCULL_IOCHQUANTUM`: (shift) tell + query
- similar operations for quantum set size

Device-Specific Commands

- `SCULL_IOCRESET`:
  assign default values to quantum set size and quantum size
- `SCULL_IOCSQUANTUM`: set quantum size from pointer
- `SCULL_IOCTQUANTUM`: (tell) set quantum size from value
- `SCULL_IOCGQUANTUM`: get quantum size to pointer
- `SCULL_IOCQQUANTUM`: (query) return quantum size
- `SCULL_IOCXQUANTUM`: (exchange) set + get
- `SCULL_IOCHQUANTUM`: (shift) tell + query
- similar operations for quantum set size

Device-Specific Commands

- `SCULL_IOCRESET`:
  assign default values to quantum set size and quantum size
- `SCULL_IOCSQUANTUM`: set quantum size from pointer
- `SCULL_IOCTQUANTUM`: (tell) set quantum size from value
- `SCULL_IOCGQUANTUM`: get quantum size to pointer
- `SCULL_IOCQQUANTUM`: (query) return quantum size
- `SCULL_IOCXQUANTUM`: (exchange) set + get
- `SCULL_IOCHQUANTUM`: (shift) tell + query
- similar operations for quantum set size

Device Operations

```
switch(cmd)
{
    case SCULL_IOCRESET:
        scull_quantum = SCULL_QUANTUM;
        scull_qset = SCULL_QSET;
        break;

    /* other cases */
}
```

Device Operations

setting quantum size

```
case SCULL_IOCSQUANTUM:
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    retval = __get_user(scull_quantum,
                        (int __user *) arg);
    break;

case SCULL_IOCTQUANTUM:
    if (! capable (CAP_SYS_ADMIN))
        return -EPERM;
    scull_quantum = arg;
    break;
```

Device Operations

getting quantum size

```
case SCULL_IOCGQUANTUM:
    retval = __put_user(scull_quantum,
                        (int __user *) arg);
    break;

case SCULL_IOCQQUANTUM:
    return scull_quantum;
```

Topics

I/O Subsystem

- Introduction
- Device Types
- I/O Software
- Accessing Devices

Device Drivers

- Interface
- Implementation
- Device Access

Device Driver Example

short: read/write I/O ports

- each device node accesses a different port:
  - `/dev/short0`: port at base
  - `/dev/short1`: port at base+1
- module parameters:
  - major number (default dynamic)
  - base address (default 0x378)

Region Allocation

module initializion

```
if (!request_region(short_base, SHORT_NR_PORTS,
                    "short"))
{
    return -ENODEV;
}
```

module cleanup

```
release_region(short_base, SHORT_NR_PORTS);
```

Read

```c
int retval = count;
int minor = iminor(filp->f_dentry->d_inode);
unsigned long port = short_base + (minor & 0x0f);
unsigned char *kbuf, *ptr;

kbuf = kmalloc(count, GFP_KERNEL);
if (!kbuf)
    return -ENOMEM;

/* do the I/O */

kfree(kbuf);
return retval;
```

Read

do the I/O

```
ptr = kbuf;
while (count--)
{
    *(ptr++) = inb(port);
    rmb();
}
if ((retval > 0) && copy_to_user(buf, kbuf, retval))
    retval = -EFAULT;
```

Write

```
if (copy_from_user(kbuf, buf, count))
    return -EFAULT;
ptr = kbuf;
while (count--)
{
    outb(*(ptr++), port);
    wmb();
}
```

Reading Material

- Corbet-Rubini-Hartman, 3/e
  - Chapter 3: Char Drivers
  - Chapter 9: Communicating with Hardware