Driver Development

Matt Genovese

Fall 2009

Outline

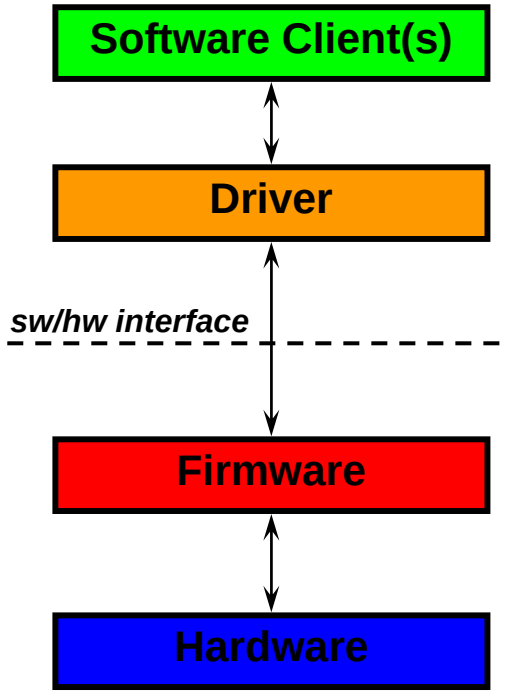
- **Driver basics**
  - What are drivers?
  - Where do they fit in embedded system design?
  - What are the general responsibilities of drivers?
  - What are application drivers versus OS drivers?
  - How can the driver's client interface be architected?
  - Discuss some basics of Linux drivers.

- **Details of Linux drivers**
  - Linux driver fundamentals
  - Char drivers
  - Memory allocation
  - Char driver read and write
  - Blocking
  - Hardware access
  - I/O ports
  - I/O memory

Driver basics

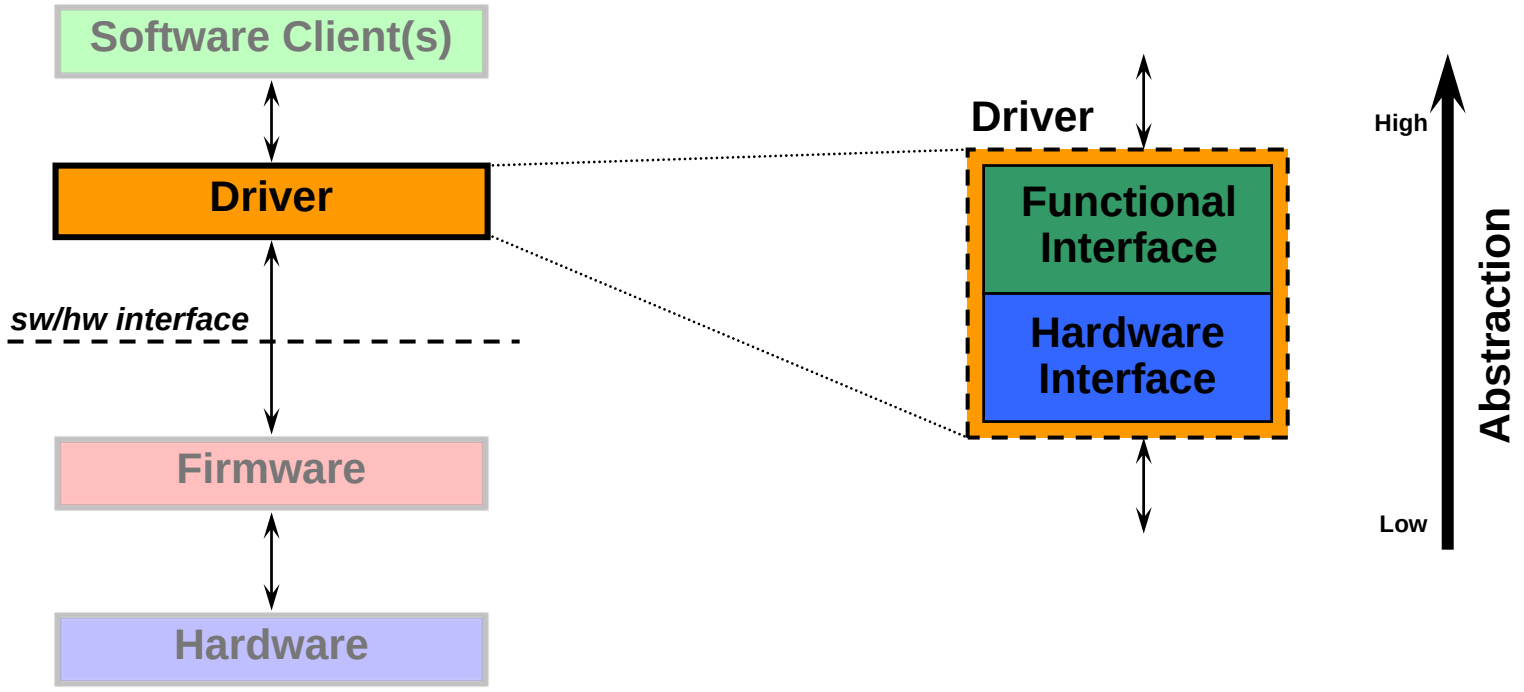
What is a driver?

- **Drivers are software…**
  - Create a clean separation between upper-level client software and the hardware
    - Provide an abstraction layer above direct hardware communication
    - Manage bandwidth across the communication channel to the hardware
  - Restrict the client interface state-space to only what is possible on the hardware
    - Control access to hardware from multiple clients
    - Ensure hardware errors translate into recoverable errors in the above software



- **Drivers are NOT applications.**
  - Applications are executable.
  - Drivers are part of an application (library), or a module that registers itself with the kernel (Linux).

What is a driver?



- **Drivers conceptually have two interfaces**
  - The *Functional Interface* interacts with the software application or OS kernel, receiving commands and returning responses.
  - The *Hardware Interface* interacts with the hardware device to execute commands and receive results and status.

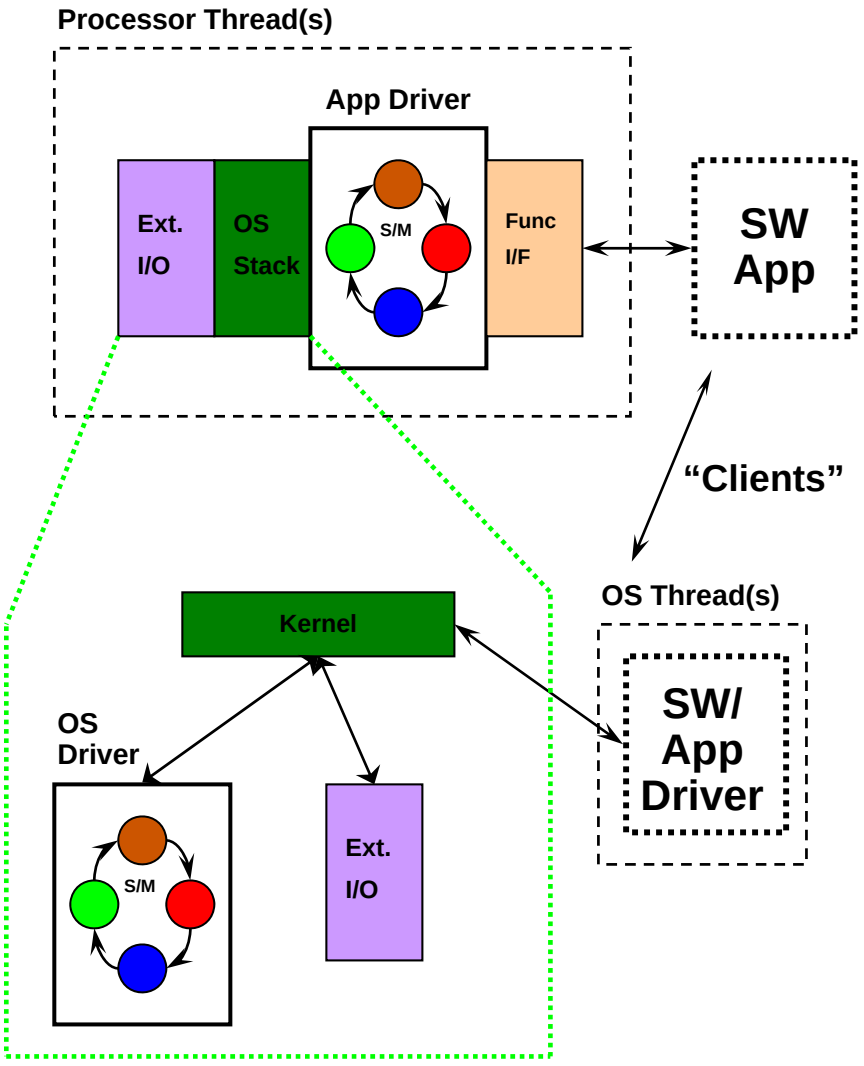
Drivers: Philosophy

- **Mechanism versus policy**
  - Mechanism: Make something at the lower level look like something else at the higher level. Emphasizes *capabilities*.
  - Policy: Who can access it, what can be done with it, etc. Emphasizes *usage*.

- **In general, drivers should implement the mechanism, and leave the policy to the OS or application.**
  - Drivers should implement the function-space of what is permissible and possible at the low-level.
  - A driver that implements a policy also implements an inherent restriction on all possible applications above it.
  - Very few instances when a driver should implement a policy.
    - When driver can affect global resources, multiple users, system stability, policy checks are present in the driver.

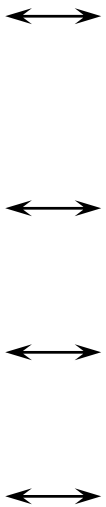
Drivers: Application versus OS

- **Application drivers cleanly separate and abstract device communication.**
  - Function at a higher level than OS drivers to further abstract communication based on an application's needs.
- **OS drivers enable safe and unified access to hardware.**
  - Accessed via kernel by software applications (e.g. application driver) to communicate with hardware.



Types of drivers

- **Application-level driver libraries**
  - The software application(s) are the direct clients of the driver.
  - Applicable within a specific set of software applications.
  - Represents a software architecture decision.
  - Driver software interface and architecture is largely unrestricted.

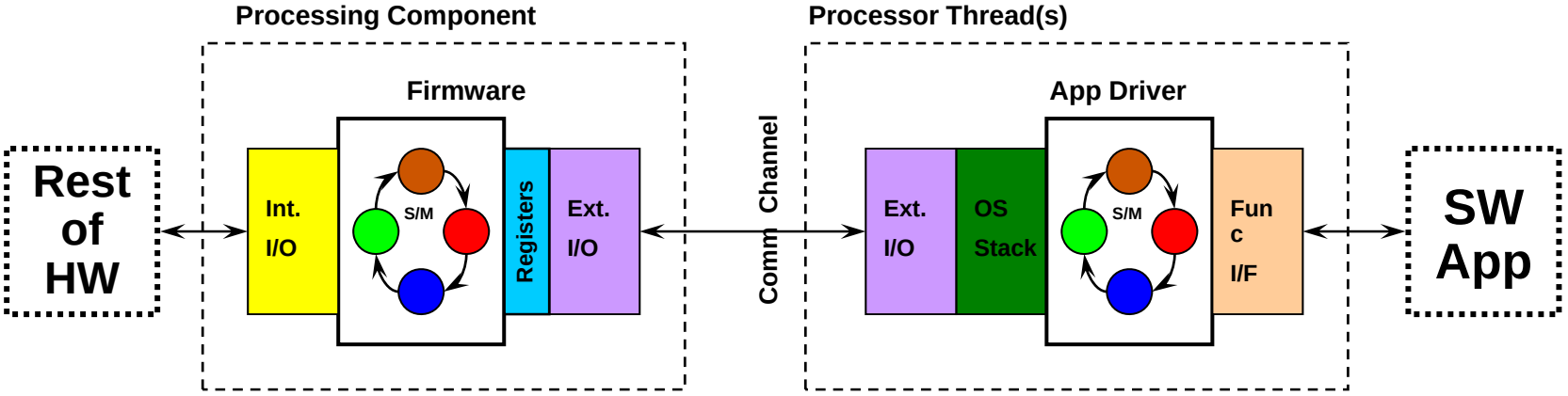


- **Operating system (OS) drivers (aka “stack”)**
  - The OS is the primary client of the driver, and applications access the driver via the kernel.
  - Applicable within the OS, and any possible application written.
  - Represents an OS stability and maintainability decision.
  - Driver software interface and architecture is quite restricted.

Software communication with the hardware

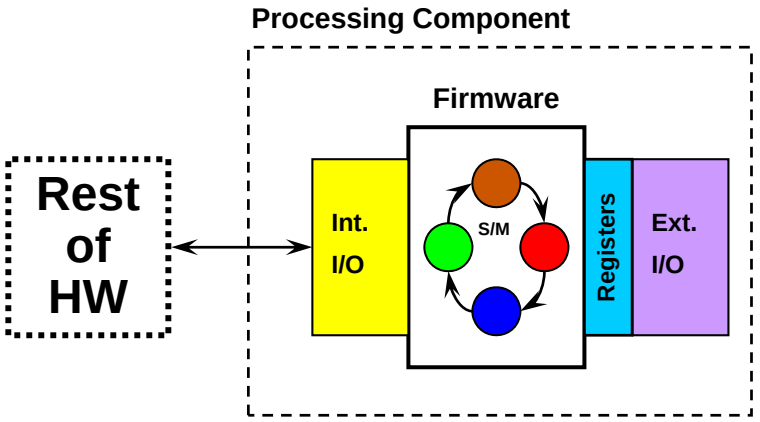
- Firmware manages basic hardware and time-sensitive tasks, and provides a low-level software interface.
  - Small embedded memory footprint can limit functionality to providing the basics.
- Communication to firmware occurs over a communication channel (e.g. USB, RS232, SRIO, SATA, etc.)

- Client software communicates with the driver via a high-level software interface.
- Application driver vets client requests and manages channel communication with firmware.
- The OS stack for the channel protocol is used for primitive communication with the hardware.



Firmware

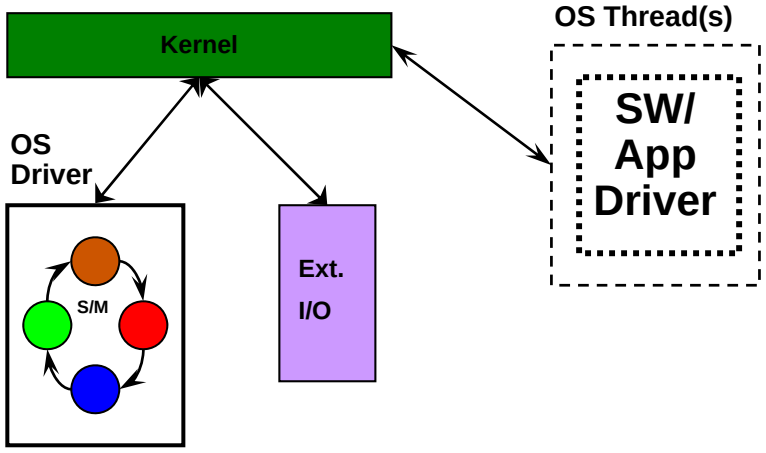
- **Needs to manage…**
  - Reset
  - Debug
  - Basic functional operation
  - Administrative tasks (e.g. upgrade)



- **Typically architected as a state machine**
  - Incoming commands from driver via the external I/O channel translate into state machine traversals, or message passing.
  - Command status reported back to driver.
  - Based on each device, firmware capabilities may be relatively basic and low-level, or could be more advanced.
  - Error detection and basic error recovery capabilities may exist in firmware, but advanced recovery can be pushed up to the driver.
  - If firmware is not employed on the hardware, then the driver must manage these responsibilities.

Linux OS drivers

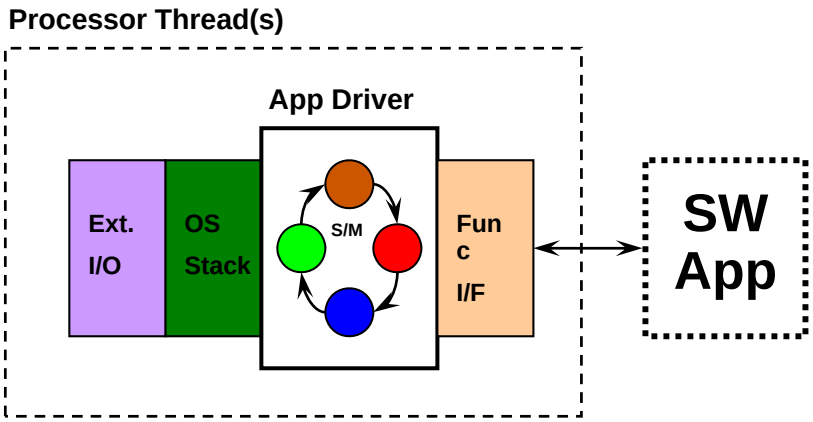
- **Needs to manage…**
  - Reset, debug, functional operation, administrative tasks in hardware
  - Error recovery
  - OS driver specifications



- **Linux drivers are modules that communicate with the hardware in a structured, safe way.**
- **Provide implementation of base methods for communicating with the hardware.**
  - Result: The device can be used by existing and new applications and other drivers seamlessly.
- **Drivers may manage access to a communication channel or to a specific device.**

Application drivers

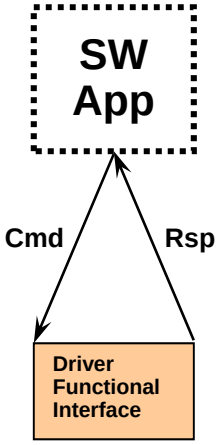
- **Needs to manage…**
  - Reset, debug, functional operation, administrative tasks in firmware (not already managed by OS driver - device specific).
  - Advanced debug and error recovery (for hardware, and client software)



- **Purpose: Provide an advanced feature set relevant to the specific device used by the client application.**
  - Functionality provided to application is higher-level than OS driver, combining appropriate low-level OS driver functions.
- **For application drivers, many architectures are possible.**
  - Dependent upon the needs of the intended client applications.

Client-driver communication: Commands

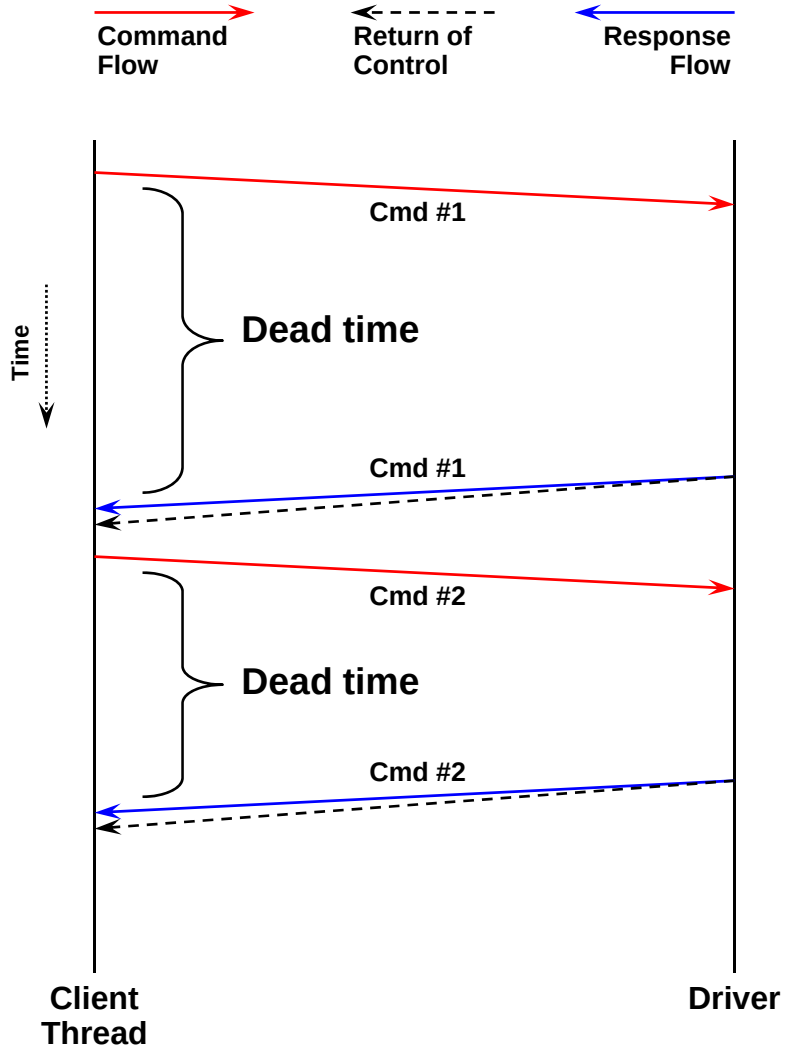
- **Definitions:**
  - A command holds the desired driver operation and inbound data.
  - A response holds the command status/result and return data.
- **How does the client issue commands to the driver?**
  - Application driver: Likely C-style global functions to driver library
  - Linux: C-style global functions to kernel
- **Application drivers: We have options for client interface:**
  - Command call: Client calls functions, each of which represents a command to the driver. Easiest for blocking drivers.
    - Pass in command as function's arguments, and the data + status are returned by that function.
  - Command object: Use a single function for passing command objects. Useful for non-blocking drivers.
    - The command object is filled with the desired operation and data, and given to the driver.
    - After the driver executes the command, the response is encapsulated in or otherwise directly linked to that command object.
    - Advantage: Upgrade driver without changing command interface.
- **Linux OS drivers: Kernel registration; client calls system functions:**
  - More secure because kernel vets all access to drivers
  - Enables categorization of drivers, and unified client interface to each
  - Ensures privileged access (i.e. policy enforcement)
  - This is the method Linux uses – discussed later.



Client-driver communication: Flow of control

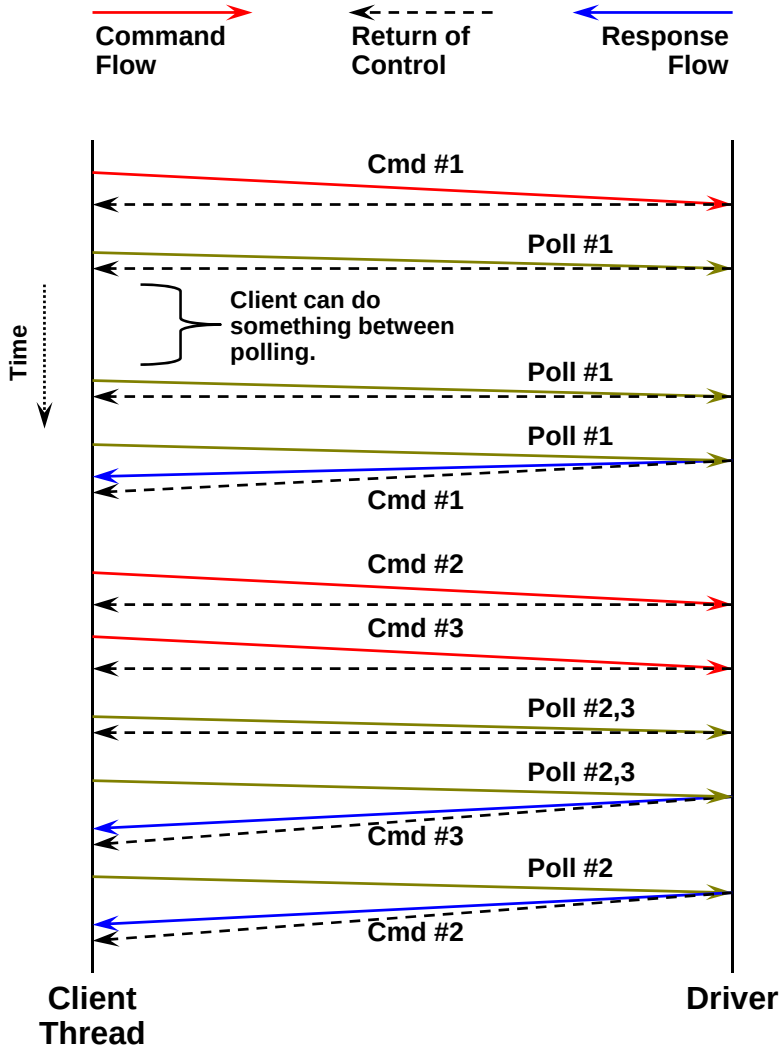
- **In general, there are a variety of ways a software client can interface with a driver in terms of program flow control. For example:**
  - Blocking interface
    - Call to driver blocks client thread execution until command is completed.
    - Easy to implement, but restrictive by forcing sequential operation.
  - Non-blocking (NB) interface, client polling
    - Call to driver returns immediately, even if command did not complete.
    - Client polls for command completion.
    - Allows for pipelining of commands, if supported by firmware.
    - Can create a blocking interface from a non-blocking interface.
  - Non-blocking (NB) interface, driver call-back or software interrupt to client
    - Same benefits as above (control returned immediately, pipelining).
    - Call-back: Requires client registration with the driver.
    - Driver alerts client when command has completed; no polling required.

Client-driver interface: Blocking



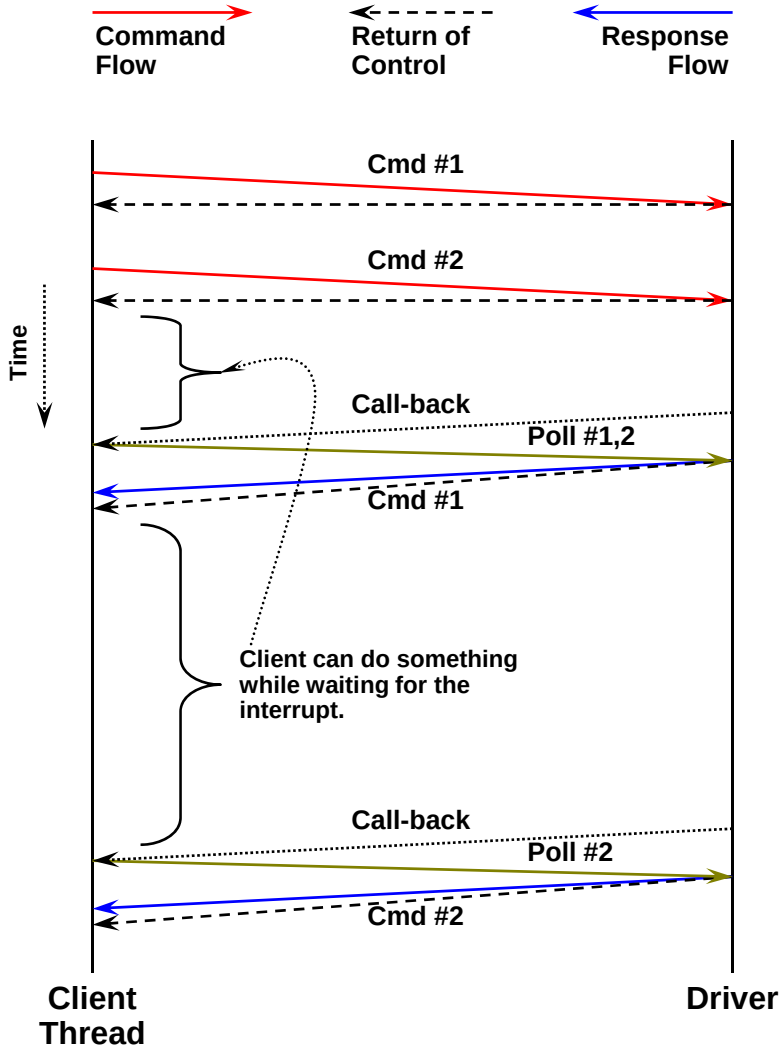
- Client commands to driver will block client thread until the driver fully executes the command.
- When the command is fully executed by the driver, the response is returned and the command has completed.
- Program control flow is coincident with command and response flow.

Client-driver interface: NB with polling



- Client sends command to driver, and receives control back immediately.
- Meanwhile driver executes command, and eventually queues up the result.
- Client polls driver to see if command completed.
  - Client has control: client can do other things while waiting.
- Client can pipeline commands, if driver allows.
  - Driver may enforce in-order execution, or all out-of-order for specific/all commands.

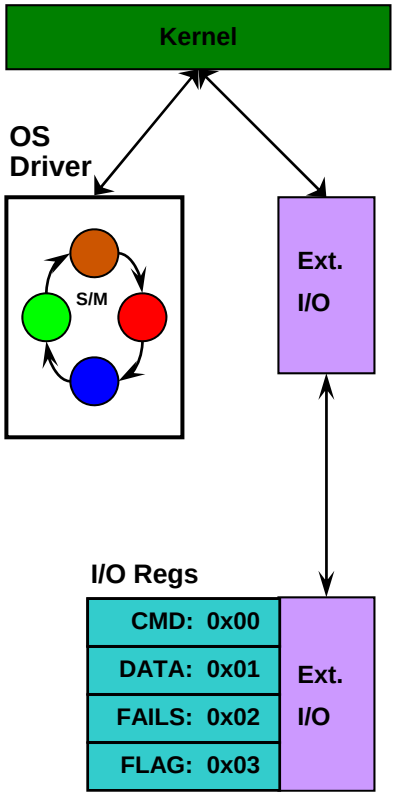
Client-driver interface: NB w/ interrupt or call-back



- Same benefits as the NB with polling scenario.
- If software interrupt, driver hits interrupt when command completes, and client polls once.
- If call-back, after command completes, driver calls client directly. Then…
  - Client polls for result, or
  - Driver passed back results w/ call-back.
- Relieves client from polling after commands are issued.

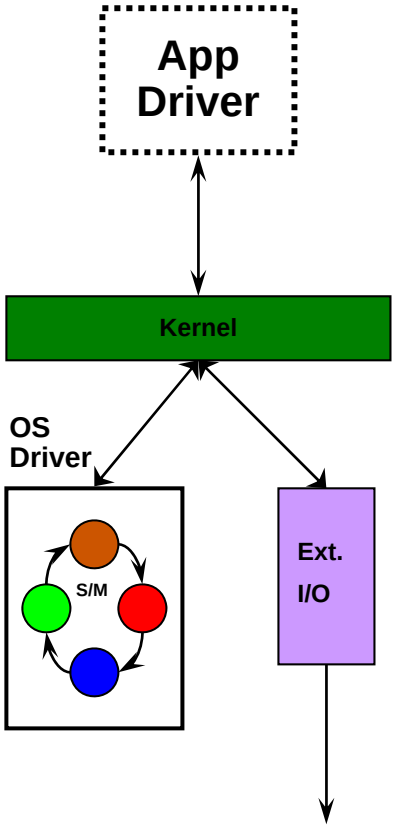
OS driver ⇒ direct hardware communication

- **The OS driver communicates with registers via the I/O channel provided by the OS.**
  - **Devices may be memory mapped, or port accessed (depending upon processor architecture).**
  - **8/16/32-bit hardware registers are accessed to configure and monitor state, and initiate activity.**
  - **Responsibility of OS driver is to translate basic commands into individual I/O register accesses that execute them on the hardware.**
  - **OS stability should never be compromised.**



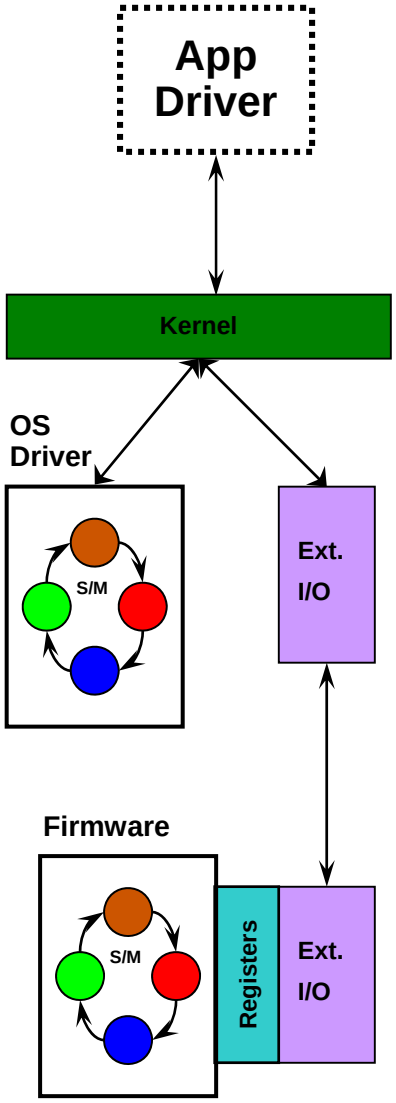
Application driver ⇒ OS driver communication

- The OS driver has taken care of basic communication with the hardware.
  - OS driver interface is basic and standardized for all Linux devices (of a given driver category)
- Application driver now uses OS driver to…
  - Build higher-level commands relevant to the upstream client (e.g. using command-response methodology).
  - Create program control interface(s) applicable to the upstream client (blocking, NB varieties).
  - Provide "nice" error recovery via timeouts, etc.
    - A crash of the hardware or firmware should not crash the client application.



Application driver ⇒ firmware communication

- **For hardware that employs firmware, the application driver communicates with the firmware to execute commands.**
  - Driver may communicate with the device's firmware via messages (higher-level than register accesses).
  - Device firmware may be written using a simple state machine approach. Driver takes firmware through states to execute commands.
  - Driver manages unique functionality of the device that is not directly managed or understood by the OS driver.
  - Prevent clients from "getting into trouble" by restricting scenarios that cause bad states in firmware.



Linux drivers

Linux drivers

- **Linux drivers are modules.**
  - Extension to kernel functionality that can be enabled/disabled at run-time (i.e. without reboot).
- **Drivers are event-driven, and registered with the kernel for call-backs.**
- ***User space* versus *kernel space***
  - *User space* is the land of processes. Every application executes within the context of a process. No process has direct access to hardware I/O.
  - *Kernel space* is separate, and can execute on behalf of user space processes (when a system call is made), or execute outside of any user space process (whenever a hardware interrupt occurs).
- **Drivers must be *reentrant*: Handle multiple non-interfering contexts.**
- **Drivers can *stack*: Drivers can be used by other drivers.**
  - Subsystem drivers used for common protocols, reducing bugs for other drivers.
- **Drivers have programming restrictions and guidelines**
  - Drivers cannot make use of *libc*. e.g *printf*() is not available; must use *printk*().
  - Floating point arithmetic not available.
  - Kernel stack space limited; dynamic variables preferred over automatic.
  - Handle concurrency, never assuming continuous access to CPU.

Classes of Linux drivers

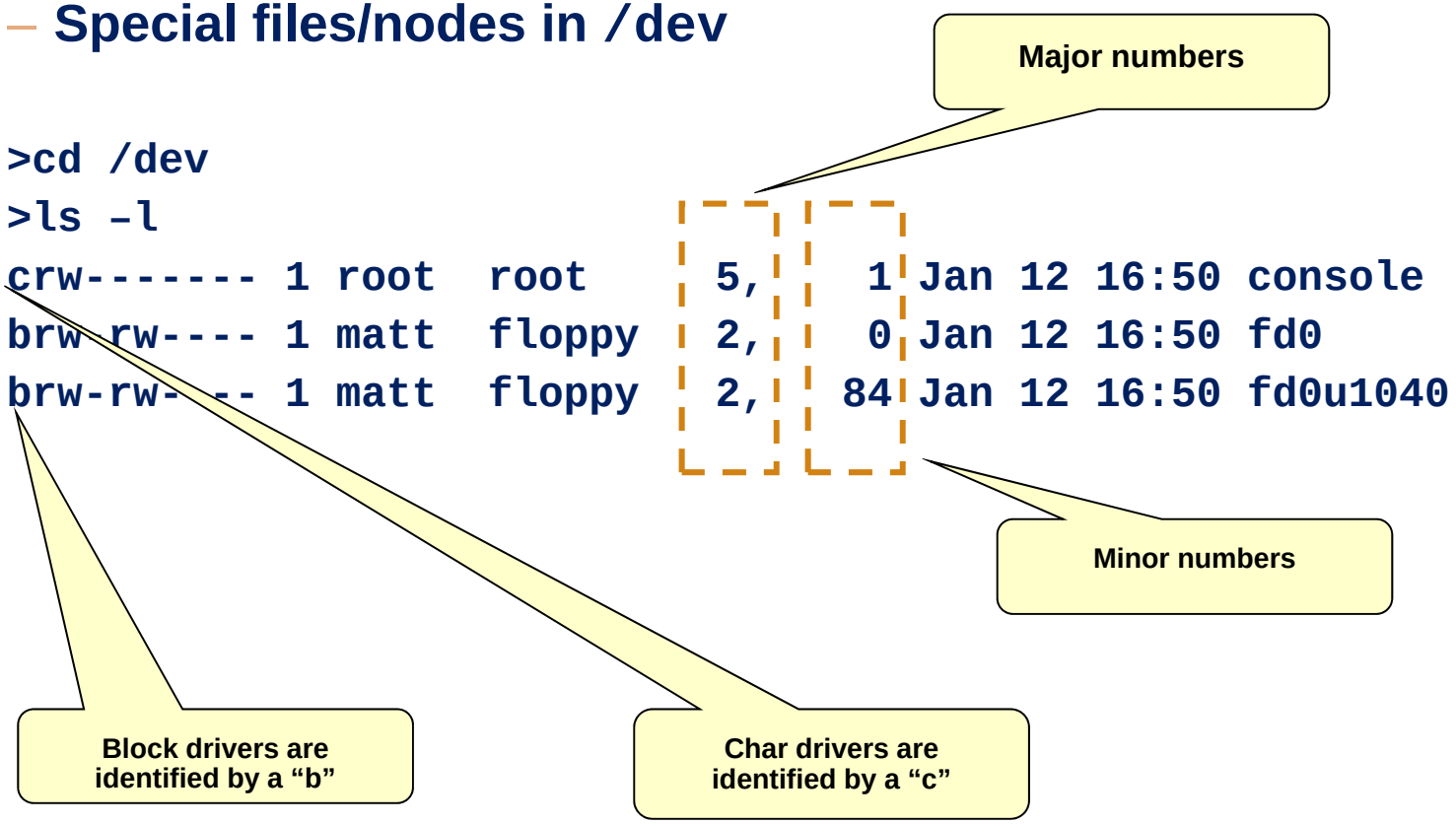
- Linux categorizes drivers by *classes*, which become Linux *devices* when implemented:
  - Char - Stream of bytes, generally sequential
  - Block - Holds a filesystem, generally random-access
  - Network - Communication with other hosts, packet access

- This categorization provides a common software interface for all drivers of each class.

- Benefit: After developing a new char driver module, it can be used by any existing software application that communicates with char drivers.

Major and Minor Numbers

- **Char devices are accessed through names in the file system**
  - **Special files/nodes in /dev**



Major and Minor Number Assignments

1 char Memory devices

Major number

Minor numbers

1 = /dev/mem Physical memory access
2 = /dev/kmem Kernel virtual memory access
3 = /dev/null Null device
4 = /dev/port I/O port access
5 = /dev/zero Null byte source
6 = /dev/core OBSOLETE - replaced by /proc/kcore
7 = /dev/full Returns ENOSPC on write 8
8 = /dev/random Nondeterministic random number gen.
9 = /dev/urandom Faster, less secure random number gen.
10 = /dev/aio Asyncronous I/O notification interface
11 = /dev/kmsg Writes to this come out as printk's

1 block RAM disk

0 = /dev/ram0 First RAM disk
1 = /dev/ram1 Second RAM disk ...
250 = /dev/initrd Initial RAM disk {2.6}

FROM http://www.kernel.org/pub/linux/docs/device-list/devices.txt

Char drivers

- Char devices are accessed through */dev* filesystem.
- Drivers need an init and exit function, called by the kernel.
  - *module_init* called via *insmod, module_exit* called via *rmmod.*
  - Registers driver with kernel, and performs all setup or cleanup.
- Device numbers (major, minor)
  - Major number: ID of the driver.
  - Minor number: ID of devices implemented by the driver.
- Structure *file_operations* links char driver methods to the implementations for this driver.
- Structure *file* represents an open file (not to be confused with a C-language *FILE*).
  - A *struct file* object is created by the kernel and passed to all *file_operations* functions that operate on that file.
  - Created when device is opened, and deleted when device is released.
- Structure *inode* is used by the kernel to represent files.
  - Different than the *file* structure which is an open file descriptor.
  - Most used: the *i_cdev* pointer to the char device structure.

I²C driver: Module init and exit

```
static int __init i2c_dev_init(void)
{
        int res;

        printk(KERN_INFO "i2c /dev entries driver\n");

        res = register_chrdev(I2C_MAJOR, "i2c", &i2cdev_fops);
        if (res)
                goto out;

        i2c_dev_class = class_create(THIS_MODULE, "i2c-dev");
        if (IS_ERR(i2c_dev_class))
                goto out_unreg_chrdev;

        res = i2c_add_driver(&i2cdev_driver);
        if (res)
                goto out_unreg_class;

        return 0;

out_unreg_class:
        class_destroy(i2c_dev_class);
out_unreg_chrdev:
        unregister_chrdev(I2C_MAJOR, "i2c");
out:
        printk(KERN_ERR "%s: Driver Initialisation failed\n", __FILE__);
        return res;
}
```

```
static void __exit i2c_dev_exit(void)
{
        i2c_del_driver(&i2cdev_driver);
        class_destroy(i2c_dev_class);
        unregister_chrdev(I2C_MAJOR,"i2c");
}

module_exit(i2c_dev_exit);
module_init(i2c_dev_init);

MODULE_AUTHOR("Frodo Looijaard <frodol@dds.nl> and "
              "Simon G. Vogl <simon@tk.uni-linz.ac.at>");
MODULE_DESCRIPTION("I2C /dev entries driver");
MODULE_LICENSE("GPL");
```

```
static const struct file_operations i2cdev_fops = {
        .owner      = THIS_MODULE,
        .llseek     = no_llseek,
        .read       = i2cdev_read,
        .write      = i2cdev_write,
        .ioctl      = i2cdev_ioctl,
        .open       = i2cdev_open,
        .release    = i2cdev_release,
};
```

These are the functions that implement the driver open, close, control, reads, and writes to be called by the user application via the kernel.

Reference: Linux 2.6 source (i2c-dev.c)

I²C driver : Obtaining major and minor numbers

```
static int i2cdev_attach_adapter(struct i2c_adapter *adap)
{
        struct i2c_dev *i2c_dev;
        int res;

        i2c_dev = get_free_i2c_dev(adap);
        if (IS_ERR(i2c_dev))
                return PTR_ERR(i2c_dev);

        /* register this i2c device with the driver core */
        i2c_dev->dev = device_create(i2c_dev_class, &adap->dev,
                                     MKDEV(I2C_MAJOR, adap->nr),
                                     "i2c-%d", adap->nr);
        if (IS_ERR(i2c_dev->dev)) {
                res = PTR_ERR(i2c_dev->dev);
                goto error;
        }
        res = device_create_file(i2c_dev->dev, &dev_attr_name);
        if (res)
                goto error_destroy;

        pr_debug("i2c-dev: adapter [%s] registered as minor %d\n",
                 adap->name, adap->nr);
        return 0;
error_destroy:
        device_destroy(i2c_dev_class, MKDEV(I2C_MAJOR, adap->nr));
error:
        return_i2c_dev(i2c_dev);
        return res;
}
```

- *MKDEV* creates the major and minor device numbers.
- In this example, since this is called for every new adapter attached, a new device number is created for each new I2C device.
- Notice the use of the often avoided *goto* statement.
  - Useful for backing out of registration process when an error is encountered.

Reference: Linux 2.6 source (i2c-dev.c)

I²C driver : *Struct file* usage and fields

```
static ssize_t i2cdev_read (struct file *file, char __user *buf, size_t count,
                            loff_t *offset)
{
        char *tmp;
        int ret;

        struct i2c_client *client = (struct i2c_client *)file->private_data;

        if (count > 8192)
                count = 8192;

        tmp = kmalloc(count,GFP_KERNEL);
        if (tmp==NULL)
                return -ENOMEM;

        pr_debug("i2c-dev: i2c-%d reading %zd bytes.\n",
                iminor(file->f_path.dentry->d_inode), count);

        ret = i2c_master_recv(client,tmp,count);
        if (ret >= 0)
                ret = copy_to_user(buf,tmp,count)?-EFAULT:ret;
        kfree(tmp);
        return ret;
}
```

Reference: Linux 2.6 source (i2c-dev.c)

- mode_t *f_mode*
  - Is the file readable and/or writable?
- loff_t *f_pos*
  - Current reading or writing position in the file
- unsigned int *f_flags*
  - Non-blocking? Sync?
- struct *file_operations* *f_op
  - File operations associated with this call.
- void **private_data*
  - Often used to preserve data between system calls.
- struct dentry **f_dentry*
  - Directory entry (access to inode structure).

Scull driver: A more recent example

- The Linux 2.6 driver interface was upgraded, so there's a new way to register drivers with the kernel.
- To create the character device, use:
  void *cdev_init* (struct cdev *cdev, struct file_operations *fops);
  - The *cdev* structure states the owner of the driver (use "THIS_MODULE"), and the operations that are supported by the driver.
- Then, to add the device to the kernel, use:
  int *cdev_add* (struct cdev *dev, dev_t num, unsigned int count);
- Later, to remove the device from the kernel, use:
  void *cdev_del* (struct cdev *dev);
- Next example is *scull*: Driver that just writes to system memory, and does not use hardware I/O.

Scull driver: Registration

```
struct file_operations scull_fops = {
    .owner = THIS_MODULE,
    .llseek = scull_llseek,
    .read = scull_read,
    .write = scull_write,
    .ioctl = scull_ioctl,
    .open = scull_open,
    .release = scull_release,
};
struct scull_dev {
    struct scull_qset *data;          /* Pointer to first quantum set */
    int quantum;                      /* the current quantum size */
    int qset;                         /* the current array size */
    unsigned long size;               /* amount of data stored here */
    unsigned int access_key;          /* used by sculluid and scullpriv */
    struct semaphore sem;             /* mutual exclusion semaphore */
    struct cdev cdev;                 /* Char device structure */
};
static void scull_setup_cdev (struct scull_dev *dev, int index) {
    int err, devno = MKDEV(scull_major, scull_minor + index);
    cdev_init(&dev->cdev, &scull_fops);
    dev->cdev.owner = THIS_MODULE;
    dev->cdev.ops = &scull_fops;
    err = cdev_add (&dev->cdev, devno, 1);
    /* Fail gracefully if need be */
    if (err)
        printk(KERN_NOTICE "Error %d adding scull%d", err, index);
}
```

- The *scull_fops* object contains pointers to functions that implement all the char device operations.
- One *scull_dev* object will exist per device served, containing all the data relevant to that device.
  - Single object provides easy encapsulation “package” for a given device.
- The device number *devno* is derived via *MKDEV*.
- The *cdev* structure is populated in *scull_setup_cdev*(), and finally the device is added to the kernel.

Char device operations: Open

```
int scull_open (struct inode *inode,
    struct file *filp)
{
    /* device information */
    struct scull_dev *dev;
    dev = container_of (inode->i_cdev,
    struct scull_dev, cdev);
    /* save pointer for later */
    filp->private_data = dev;
    /* now trim to 0 the length of the
    device if open was write-only */
    if ( (filp->f_flags & O_ACCMODE) ==
    O_WRONLY) {
        /* ignore errors */
        scull_trim(dev);
    }
    return 0; /* success */
}
```

- **Prototype:**
  int (**open*) (struct inode *inode, struct file *filp);
- **Implementation should initialize the driver for any subsequent operations.**
  - Check for device-specific errors
  - Initialize the device if necessary
  - Update the *file_operations* pointer, as necessary
  - Allocate and initialize any private data in *filp->private_data*
- **Use the *inode* to get to the *cdev* structure, via:**
  container_of (pointer, container_type, container_field);
  - If using *register_chrdev*() method (old), must look up via major/minor device numbers.

Char device operations: Release

```
int scull_release (struct inode *inode,
struct file *filp)
{
    /* No memory allocs to clean up */
    /* No hardware to close */
    return 0;
}
```

- **Prototype:**

  int (**release*) (struct inode *inode, struct file *filp)

- **Implementation should:**
  - Deallocate anything allocated in *filp->private_data*
  - Shutdown the device on the last close
- **Not every *close* system call results in a subsequent *release*.**
  - The kernel keeps track of how many times a *file* structure is being used.
  - Only when the *file* structure is to be destroyed will the *release* function be called.

Dynamic memory allocation

```
struct scull_qset {
    void **data;
    struct scull_qset *next;
};

int scull_trim(struct scull_dev *dev)
{
    struct scull_qset *next, *dptr;
    int qset = dev->qset; /* "dev" is not-null */
    int i;
    for (dptr = dev->data; dptr; dptr = next)
    {
        /* all the list items */
        if (dptr->data) {
            for (i = 0; i < qset; i++)
            kfree(dptr->data[i]);
            kfree(dptr->data);
            dptr->data = NULL;
        }
        next = dptr->next;
        kfree(dptr);
    }
    dev->size = 0;
    dev->quantum = scull_quantum;
    dev->qset = scull_qset;
    dev->data = NULL;
    return 0;
}
```

- Analogous calls for *malloc* and *free* in C:
  void **kmalloc* (size_t size, int flags);
  void *kfree* (void *ptr);
- Recall guideline to avoid automatic variables due to kernel stack limitations.
- In practice, should not implement limits on amount of data that can be managed.
  - Resolution: Use a linked list of pointers to dynamically allocated memory regions.

Char device operations: Read and write

- **Prototypes:**

  ssize_t *read* (struct file *filp, char __user *buff, size_t count, loff_t *offp);

  ssize_t *write* (struct file *filp, const char __user *buff, size_t count, loff_t *offp);

  - *filp* is the file structure pointer
  - *buff* is the buffer where data should be read from or written to in user space.
  - *count* is the size of the requested transfer (bytes)
  - *offp* is the file position the user program is accessing

- **The *buff* pointer cannot be dereferenced the standard "C way". Instead, you must use kernel functions:**

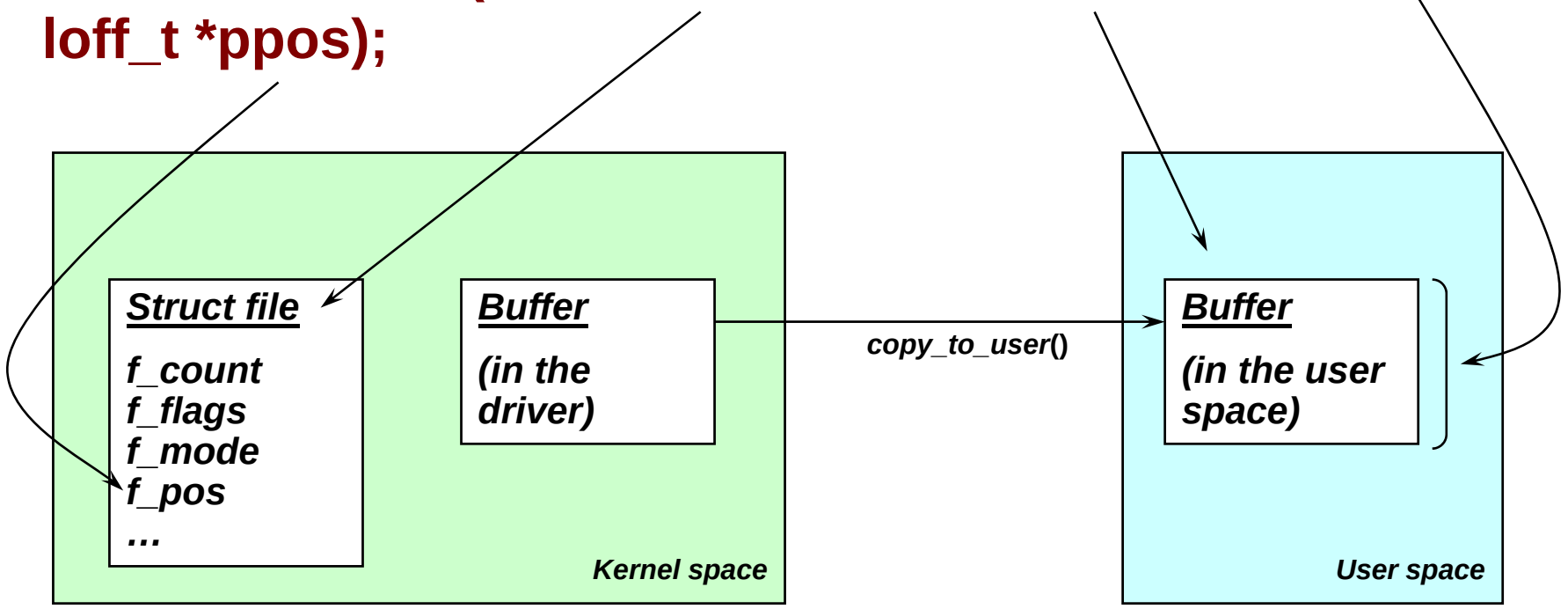
  unsigned long *copy_to_user* (void __user *to, const void *from, unsigned long count);

  unsigned long *copy_from_user* (void *to, const void __user *from, unsigned long count);

  - These functions check validity of user space pointer, and make sure pages are loaded so memory is available (i.e. perform safety checks).

Read argument usage

ssize_t *dev_read* (struct file *file, char *buf, size_t count, loff_t *ppos);



- **Return value less than 0 means an error occurred. Value >= 0 reports the number of bytes successfully transferred.**

Scull driver: Read implementation

```
ssize_t scull_read (struct file *filp, char __user *buf, size_t count, loff_t *f_pos)
{
        struct scull_dev *dev = filp->private_data;
        struct scull_qset *dptr; /* the first listitem */
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset; /* how many bytes in the listitem */
        int item, s_pos, q_pos, rest;
        ssize_t retval = 0;
        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;
        if (*f_pos >= dev->size)
                goto out;
        if (*f_pos + count > dev->size)
                count = dev->size - *f_pos;
        /* find listitem, qset index, and offset in the quantum */
        item = (long)*f_pos / itemsize;
        rest = (long)*f_pos % itemsize;
        s_pos = rest / quantum; q_pos = rest % quantum;
        /* follow the list up to the right position (defined elsewhere) */
        dptr = scull_follow(dev, item);
        if (dptr = = NULL || !dptr->data || ! dptr->data[s_pos])
                goto out; /* don't fill holes */
        /* read only up to the end of this quantum */
        if (count > quantum - q_pos)
                count = quantum - q_pos;
        if (copy_to_user(buf, dptr->data[s_pos] + q_pos, count)) {
                retval = -EFAULT;
                goto out;
        }
        *f_pos += count;
        retval = count;
        out:
        up(&dev->sem);
        return retval;
}
```

- **Prototype:**

  ssize_t *read* (struct file *filp, char __user *buff, size_t count, loff_t *offp);

- **Implementation notes:**
  - If return value…
    - == count, requested # of bytes were transferred.
    - >0 && < count, part of the data was transferred.
    - == 0, EOF was reached.
    - < 0, error occurred. See *<linux/errono.h>*
  - If data is not available right now, then read should block.

Scull driver: Write implementation

```
ssize_t scull_write(struct file *filp, const char __user *buf, size_t count,
loff_t *f_pos) {
        struct scull_dev *dev = filp->private_data;
        struct scull_qset *dptr;
        int quantum = dev->quantum, qset = dev->qset;
        int itemsize = quantum * qset;
        int item, s_pos, q_pos, rest;
        ssize_t retval = -ENOMEM; /* value used in "goto out" statements */
        if (down_interruptible(&dev->sem))
                return -ERESTARTSYS;
        /* find listitem, qset index and offset in the quantum */
        item = (long)*f_pos / itemsize;
        rest = (long)*f_pos % itemsize;
        s_pos = rest / quantum; q_pos = rest % quantum;
        /* follow the list up to the right position */
        dptr = scull_follow(dev, item);
        if (dptr = = NULL)
                goto out;
        if (!dptr->data) {
                dptr->data = kmalloc(qset * sizeof(char *), GFP_KERNEL);
                if (!dptr->data)
                        goto out;
                memset(dptr->data, 0, qset * sizeof(char *));
        }
        if (!dptr->data[s_pos]) {
                dptr->data[s_pos] = kmalloc(quantum, GFP_KERNEL);
                if (!dptr->data[s_pos])
                        goto out;
        }
        /* write only up to the end of this quantum */
        if (count > quantum - q_pos)
                count = quantum - q_pos;
        if (copy_from_user(dptr->data[s_pos]+q_pos, buf, count)) {
                retval = -EFAULT;
                goto out;
        }
        *f_pos += count;
        retval = count;
        /* update the size */
        if (dev->size < *f_pos)
                dev->size = *f_pos;
        out:
        up(&dev->sem);
        return retval;
}
```

- **Prototype:**

  ssize_t *write* (struct file *filp, const char __user *buff, size_t count, loff_t *offp);

- **Implementation notes:**
  - If return value…
    - == count, requested # of bytes were transferred.
    - >0 && < count, part of the data was transferred.
    - == 0, nothing was written.
    - < 0, error occurred. See *<linux/errono.h>*

Optional implementations: *writev* and *readv*

```
struct iovec
{
   void _ _user *iov_base;
   __kernel_size_t iov_len;
};
```

- In some cases, it's desirable to read or write vectors of data. The implementation of *read* and *write* may not be optimized for such operations.
- Solution: Implement *writev* and *readv* (vector operations).

  ssize_t (**readv*) (struct file *filp, const struct iovec *iov, unsigned long count, loff_t *ppos);

  ssize_t (**writev*) (struct file *filp, const struct iovec *iov, unsigned long count, loff_t *ppos);
- Each *iovec* describes one set of data to be transferred.
  - *iov_base*: user space pointer to data
  - *iov_len*: total length in bytes
- The *counter* indicates the number of *iovec* structures.
- If *writev()* or *readv()* are not implemented, kernel iterates over the *write* and *read* functions to accomplish the same task.

Device control via IOCTL

- Reading and writing are typically for data only, and not for device control.
  - Example device control actions: Eject media, lock device, etc.
  - Example exception: Escape codes for console display. Requires special parsing of data for control information.
- IOCTL = I/O control
- User space IOCTL call to kernel is:

  int *ioctl* (int fd, unsigned long cmd, ...);
  - Notice the "…", meaning that a variable number of arguments are allowed. In reality, the argument is *char *argp.*
  - This is because a generic control command can conceivably have 0..N arguments, depending on the operation.
- Driver method prototype:

  int (**ioctl*) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);
  - The ellipsis arguments from the kernel system call translate into the *unsigned long arg* argument.
- For more info, see supplemental slides, and Linux reference.

Blocking and sleeping

- **Accessing hardware is generally asynchronous to the OS.**
  - Read data is not always immediately available from hardware, and write data cannot always be written immediately.
  - When the action cannot take place now, but it is anticipated to eventually take place (due to a hardware latency), driver needs to *block* the calling process by putting it to *sleep*.
- **Sleeping the process**
  - Process is removed from the process scheduler queue.
- **Rules**
  - Never sleep an atomic operation.
  - No sleeping unless assured task will be woken up.
  - Never sleep if interrupts are disabled. After all, who will wake it up?
  - Can sleep while holding a semaphore.
    - All threads waiting for the semaphore will also block.
    - Be careful that the process to sleep isn't required to wake you up.
  - Make no assumptions about system state or time passed when woken up.

Waiting and awaking

- A Linux *wait queue* is used to contain a list of processes all waiting for an event to wake them up.
- Drivers can wait on conditions using *wait_event*() and variants:
  - *wait_event* (queue, condition)
  - *wait_event_interruptible* (queue, condition)
  - *wait_event_timeout* (queue, condition, timeout)
  - *wait_event_interruptible_timeout* (queue, condition, timeout)
  - Typically *wait_event_interruptible*() is used since the sleeping process should be interruptible by a signal.
- Then, another process or interrupt handler can wake up the driver:
  - void *wake_up* (wait_queue_head_t *queue);
  - void *wake_up_interruptible* (wait_queue_head_t *queue);
- Possible issues here that could lead to corner cases, especially in multiprocesor systems. Consult Linux reference for more details.

Hardware access

- The previous slides discussed the kernel and software application interface to the driver. What about the driver's interface to the hardware?
- In hardware, all peripherals are manipulated through memory-mapped registers.
  - Some processors define separate address spaces between regular memory and I/O (peripheral).
    - x86 has separate R/W lines for conventional memory and I/O, with special CPU instructions to access I/O ports.
  - Other processors may use a unified address space, and certain regions are mapped for I/O versus memory.
    - Regions are defined as conventional memory and I/O memory.

Issues with hardware access

- **CPU can reorder instructions.**
  - Reads and writes may be reordered on the assumption that the addresses accessed are conventional memory and well-behaved.
  - Conventional memory is generally cacheable. Possible for data to never reach the system bus.
- **Compilers can optimize source code.**
  - Heavily used variables may be stored in CPU general-purpose registers instead of written to memory.
- **Two solutions (need to do both):**
  - Solution #1: Disable hardware caching in I/O regions.
    - E.g. Use Linux init code to set this up.
  - Solution #2: Prevent reordering by use of memory barriers.
    - Linux provides kernel functions to do this.

Memory barriers

- See *<linux/kernel.h>*:
  - *barrier*()
    - Software barrier that affects compiler optimization by instructing all memory in GP registers must be flushed to memory.
    - No affect on hardware reordering.
  - *rmb*(), *read_barrier_depends*(), *wmb*(), *mb*()
    - Hardware memory barriers
    - *rmb*() - Reads prior to *rmb*() will be completed before any subsequent read is executed.
    - *wmb*() - Writes prior to *wmb*() will be completed before any subsequent write is executed.
    - *mb*() - Same as above, for both reads and writes.
    - *read_barrier_depends*() - Same as *rmb*(), but blocks reordering of reads dependent upon previous read data. Better to just use *rmb*() for safety.
  - *smp_rmb*(), *smp_read_barrier_depends*(), *smp_wmb*(), *smp_mb*()
    - Same as above, but for SMP-compiled kernels.
- Benefit: Functions available for all Linux architectures, and already account for hardware implementations.
  - i.e. Portable code across architectures.

I/O port allocation

- **So we can just start using the I/O port, right?**
  - **WRONG.**
  - **Need to make sure your driver doesn't clobber another driver's access to the same peripheral.**
  - **For Linux, the process is always: *request*, *access*, *release*.**
- **Use kernel functions to gain port access:**

  struct resource **request_region* (unsigned long first, unsigned long n, const char *name);
  - **Request a set of ports (see */proc/ioports*) from *first* until *first+n* for your driver named *name*.**
  - **Non-NULL return value means success.**
- **Similarly, release ports:**

  void *release_region* (unsigned long start, unsigned long n);

I/O port access

- **Once access gained, use kernel functions to communicate with hardware ports.**
  - unsigned *inb* (unsigned port)
    void *outb* (unsigned char byte, unsigned port)
    - 8-bit read or write byte access
  - unsigned *inw* (unsigned port)
    void *outw* (unsigned short word, unsigned port)
    - 16-bit read or write byte access
  - unsigned *inl* (unsigned port)
    void *outl* (unsigned long longword, unsigned port)
    - 32-bit read or write byte access
  - 64-bit access not supported. All port register accesses are max 32-bits wide.
- **There are also versions for R/W strings of bytes, words, longs to a single port**
  - E.g. unsigned *insb* (unsigned port, void *addr, long count)
  - Others functions use same nomenclature.
  - Beware of byte ordering for string word and long versions.

I/O memory allocation

- In contrast to ports, I/O memory is an area that a peripheral memory-maps for access by the CPU.
  - Area may include registers or RAM - both looks the same to the CPU.
- Even though the I/O memory is accessible via normal memory references, the kernel functions must be used for safety.
- Like I/O ports, we must request and later release I/O memory.
  - First step is to request the memory region (see /proc/iomem).

  struct resource **request_mem_region* (unsigned long start, unsigned long len, char *name);

  void *release_mem_region* (unsigned long start, unsigned long len);
- Next, must make sure physical memory is available to kernel:

  #include <asm/io.h>

  void **ioremap* (unsigned long phys_addr, unsigned long size);

  void **ioremap_nocache* (unsigned long phys_addr, unsigned long size);

  void *iounmap* (void * addr);

I/O memory access

- **To access an address in I/O memory, use the built-in kernel functions to read/write 8/16/32-bit data.**
  - **Read/write one value to one address:**

unsigned int *ioread8* (void *addr);
unsigned int *ioread16* (void *addr);
unsigned int *ioread32* (void *addr);
void *iowrite8* (u8 value, void *addr);
void *iowrite16* (u16 value, void *addr);
void *iowrite32* (u32 value, void *addr);

- **The *addr* (address) comes from the *ioremap*() return value, with a possible offset.**

I/O memory access

- **To repeatedly access an address in I/O memory, use the built-in kernel functions to read/write 8/16/32-bit data.**
  - **Read/write multiple values to one address:**

void *ioread8_rep* (void *addr, void *buf, unsigned long count);
void *ioread16_rep* (void *addr, void *buf, unsigned long count);
void *ioread32_rep* (void *addr, void *buf, unsigned long count);
void *iowrite8_rep* (void *addr, const void *buf, unsigned long count);
void *iowrite16_rep* (void *addr, const void *buf, unsigned long count);
void *iowrite32_rep* (void *addr, const void *buf, unsigned long count);

I/O memory access

- **To access a block of I/O memory area, use these built-in kernel functions.**
  - **Fill a single value into I/O memory space:**

void *memset_io* (void *addr, u8 value, unsigned int count);

  - **Copy data from or to I/O memory space:**

void *memcpy_fromio* (void *dest, void *source, unsigned int count);

void *memcpy_toio* (void *dest, void *source, unsigned int count);

I/O ports as I/O memory

- **Some versions of hardware may use I/O ports, while others use I/O memory.**
  - **This would cause duplicate driver code that needs to handle both cases.**
  - **Problem address in Linux kernel 2.6**
- **For hardware that uses I/O ports, the appropriate port regions would be requested using *request_region*().**
- **Then, ports are mapped to memory:**

  void **ioport_map* (unsigned long port, unsigned int count);
  - **Address returned is now start of ports.**
  - **Now, *ioread8*(), etc. can be used on these addresses.**
- **Afterwards, ports are unmapped:**

  void *ioport_unmap* (void *addr);

Parallel port driver: I/O port allocation

```
static void __devinit winbond_check(int io, int key) {
      int devid,devrev,oldid,x_devid,x_devrev,x_oldid;

      if (!request_region(io, 3, __func__))
            return;

      /* First probe without key */
      outb(0x20,io);
      x_devid=inb(io+1);
      outb(0x21,io);
      x_devrev=inb(io+1);
      outb(0x09,io);
      x_oldid=inb(io+1);

      outb(key,io);
      outb(key,io);     /* Write Magic Sequence to EFER, extended function
      enable register */
      outb(0x20,io);    /* Write EFIR, extended function index register */
      devid=inb(io+1);  /* Read EFDR, extended function data register */
      outb(0x21,io);
      devrev=inb(io+1);
      outb(0x09,io);
      oldid=inb(io+1);
      outb(0xaa,io);    /* Magic Seal */

      if ((x_devid == devid) && (x_devrev == devrev) && (x_oldid == oldid))
            goto out; /* protection against false positives */
      decode_winbond(io,key,devid,devrev,oldid);
out:
      release_region(io, 3);
}
```

- Code is used in parallel port driver to probe for Winbond-compatible hardware.
- Ports are first requested, and if not available, exit.
- Configuration attempted, and then finally the ports are released.

Parallel port driver: I/O port access

```
static size_t parport_pc_epp_write_data (struct parport *port, const
    void *buf, size_t length, int flags)
{
    size_t written = 0;

    if ((flags & PARPORT_EPP_FAST) && (length > 1)) {
        if (!(((long)buf | length) & 0x03)) {
            outsl (EPPDATA (port), buf, (length >> 2));
        } else {
            outsb (EPPDATA (port), buf, length);
        }
        if (inb (STATUS (port)) & 0x01) {
            clear_epp_timeout (port);
            return -EIO;
        }
        return length;
    }
    for (; written < length; written++) {
        outb (*((char*)buf), EPPDATA(port));
        buf++;
        if (inb (STATUS(port)) & 0x01) {
            clear_epp_timeout (port);
            break;
        }
    }

    return written;
}
```

- Parallel port write implementation supports both EPP (hardware handshaking) and standard (software handshaking).
  - Evident from the two implementations in the function.
  - First portion executes if EPP, outputting a string of data, and finally checking status at the end. Hardware manages flow control.
  - Second portion is standard mode: output each byte, and wait for confirmation.

PCI driver: I/O memory or port allocation

```
int pci_request_region(struct pci_dev *pdev, int bar, const char *res_name)
{
        struct pci_devres *dr;

        if (pci_resource_len(pdev, bar) == 0)
                return 0;

        if (pci_resource_flags(pdev, bar) & IORESOURCE_IO) {
                if (!request_region(pci_resource_start(pdev, bar),
                                pci_resource_len(pdev, bar), res_name))
                        goto err_out;
        }
        else if (pci_resource_flags(pdev, bar) & IORESOURCE_MEM) {
                if (!request_mem_region(pci_resource_start(pdev, bar),
                                        pci_resource_len(pdev, bar), res_name))
                        goto err_out;
        }

        dr = find_pci_dr(pdev);
        if (dr)
                dr->region_mask |= 1 << bar;

        return 0;

err_out:
        printk (KERN_WARNING "PCI: Unable to reserve %s region #%d:%llx@%llx "
                "for device %s\n",
                pci_resource_flags(pdev, bar) & IORESOURCE_IO ? "I/O" : "mem",
                bar + 1, /* PCI BAR # */
                (unsigned long long)pci_resource_len(pdev, bar),
                (unsigned long long)pci_resource_start(pdev, bar),
                pci_name(pdev));
        return -EBUSY;
}
```

- **PCI method to request a region can access either I/O memory or I/O ports.**
  - **Flag *IORESOURCE_IO* instructs use of *request_region*().**
  - **Flag *IORESOURCE_MEM* instructs use of *request_mem_region*().**

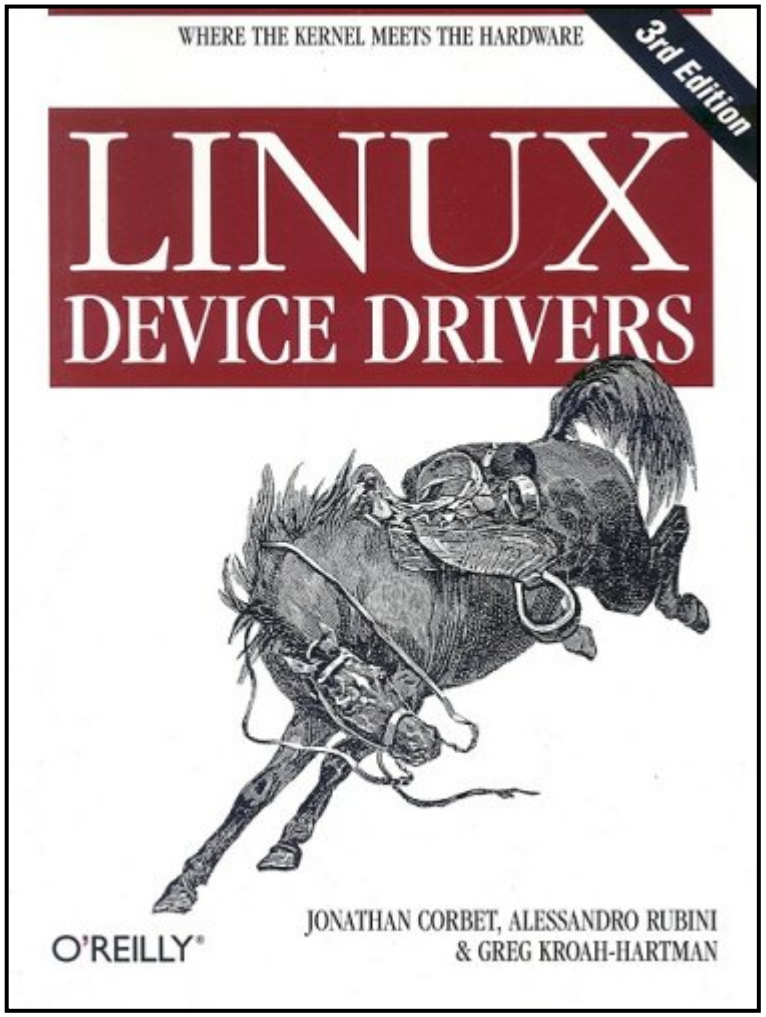
Resources

- Linux Device Drivers, Third Edition (O'Reilly).

  Johnathan Corbet, Alessandro Rubini, Greg Kroah-Hartman.

  Available for free in electronic form:

  http://lwn.net/Kernel/LDD3/



Supplemental slides

IOCTL commands

- **Driver method prototype**

  int (*ioctl) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);

  - Generally a big *switch* statement will be used to check for each possible command in *cmd* and then check for its valid arguments.
    - Thus, any errors will have to be debugged at run-time. No compile-time check possible on IOCTL arguments passed in.
    - If command is invalid, should return error code *-ENOTTY*.
  - All commands are unique in a system to prevent unreported mix-ups (command inadvertently sent to wrong driver).
    - Command is 32-bits: type(8), number(8), direction(2), size(14)
    - Field *type* is a driver-specific value.
    - Field *number* is the command selection.
    - Field *direction* is: _IOC_NONE, _IOC_READ, _IOC_WRITE, or _IOC_READ | _IOC_WRITE (bi-directional)
    - Field *size* is the size of user data.
  - All of these are arbitrary for the command, and not all fields have to be used. It's just an attempt at standardizing IOCTL commands.
  - FYI: Some commands are predefined and recognized by the kernel (before your driver receives it).

IOCTL command arguments

- **Driver method prototype**

  int (ioctl*) (struct inode *inode, struct file *filp, unsigned int cmd, unsigned long arg);**

- **Handling arguments:**
  - **If no argument was used from the user space call, *arg* is undefined.**
  - **If a single integer argument, it can be accessed directly.**
  - **If a pointer, then we must be careful about access to user space.**
    - **Can use *copy_to_user*() and *copy_from_user*() kernel methods, but that's overkill for such a small memory footprint for the arguments. Use:**

    int *access_ok* (int type, const void *addr, unsigned long size);
    - *type*: *VERIFY_READ* or *VERIFY_WRITE*, depending if command will read or write to user space.
    - *addr*: Address in user space.
    - *size*: Use *sizeof*(<type>). E.g. if accessing an integer, use *sizeof(int)*.
    - Return value is boolean: 1 if ok, 0 if not.
  - **Then, use *put_user*() and *get_user*() to transfer data.**
    - **See <*asm/uaccess.h*>.**

Scull driver: IOCTL example

```
switch (cmd) {
    case SCULL_IOCRESET:
        scull_quantum = SCULL_QUANTUM;
        scull_qset = SCULL_QSET;
        break;
    case SCULL_IOCSQUANTUM: /* Set: arg points to the value */
        if (! capable (CAP_SYS_ADMIN))
            return -EPERM;
        retval = __get_user(scull_quantum, (int __user *)arg);
        break;
    case SCULL_IOCTQUANTUM: /* Tell: arg is the value */
        if (! capable (CAP_SYS_ADMIN))
            return -EPERM;
        scull_quantum = arg;
        break;
    case SCULL_IOCGQUANTUM: /* Get: arg is pointer to result */
        retval = __put_user(scull_quantum, (int __user *)arg);
        break;
    case SCULL_IOCQQUANTUM: /* Query: return it (it's positive) */
        return scull_quantum;
    default:  /* redundant, as cmd was checked against MAXNR */
        return -ENOTTY;
}
return retval;
```

- ***capable*() kernel function is a means of adding permissions to the driver.**
  - **Isn't this an implementation of policy? Yes.**
  - **However, sometimes read/write access needs to be differentiated from some device control access.**
  - ***CAP_SYS_ADMIN* and other permission grades are listed in *<linux/capability.h>*.**

Glossary

- scull - Simple Character Utility for Loading Localities
- HAL – Hardware Abstraction Layer
- IOCTL – I/O Control