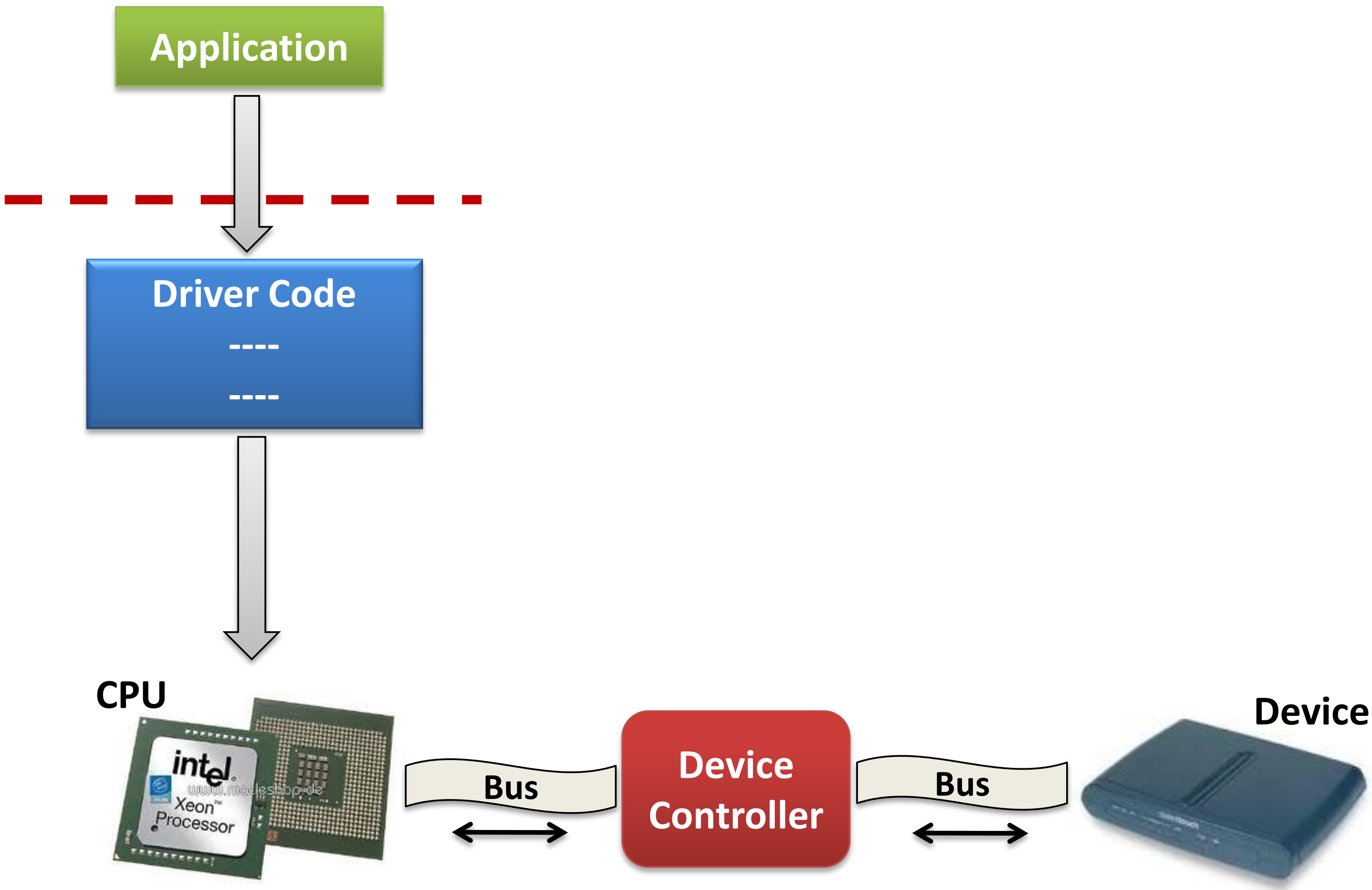
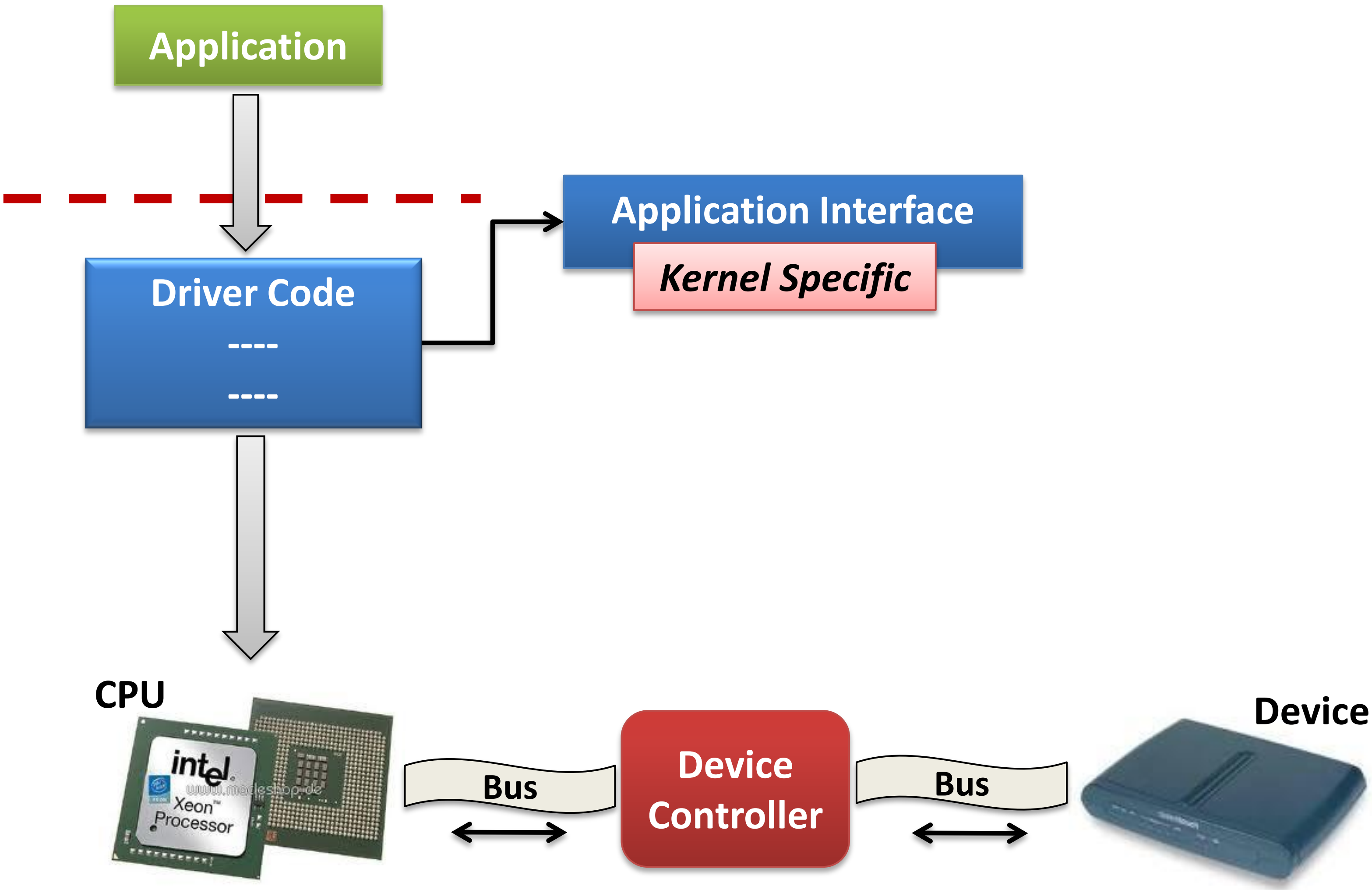
# Device Drivers

# Device Driver Framework

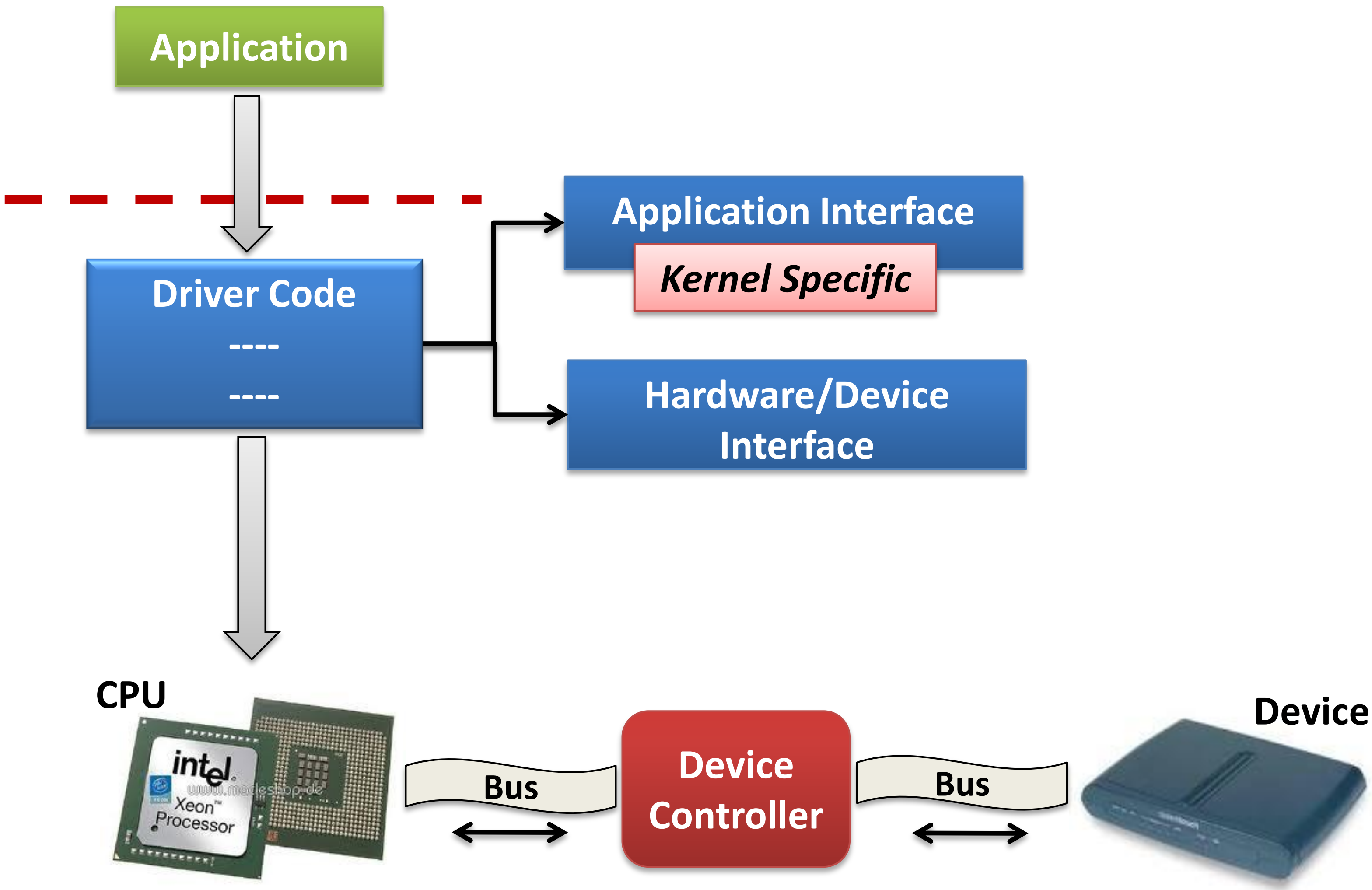
# Device Driver Framework



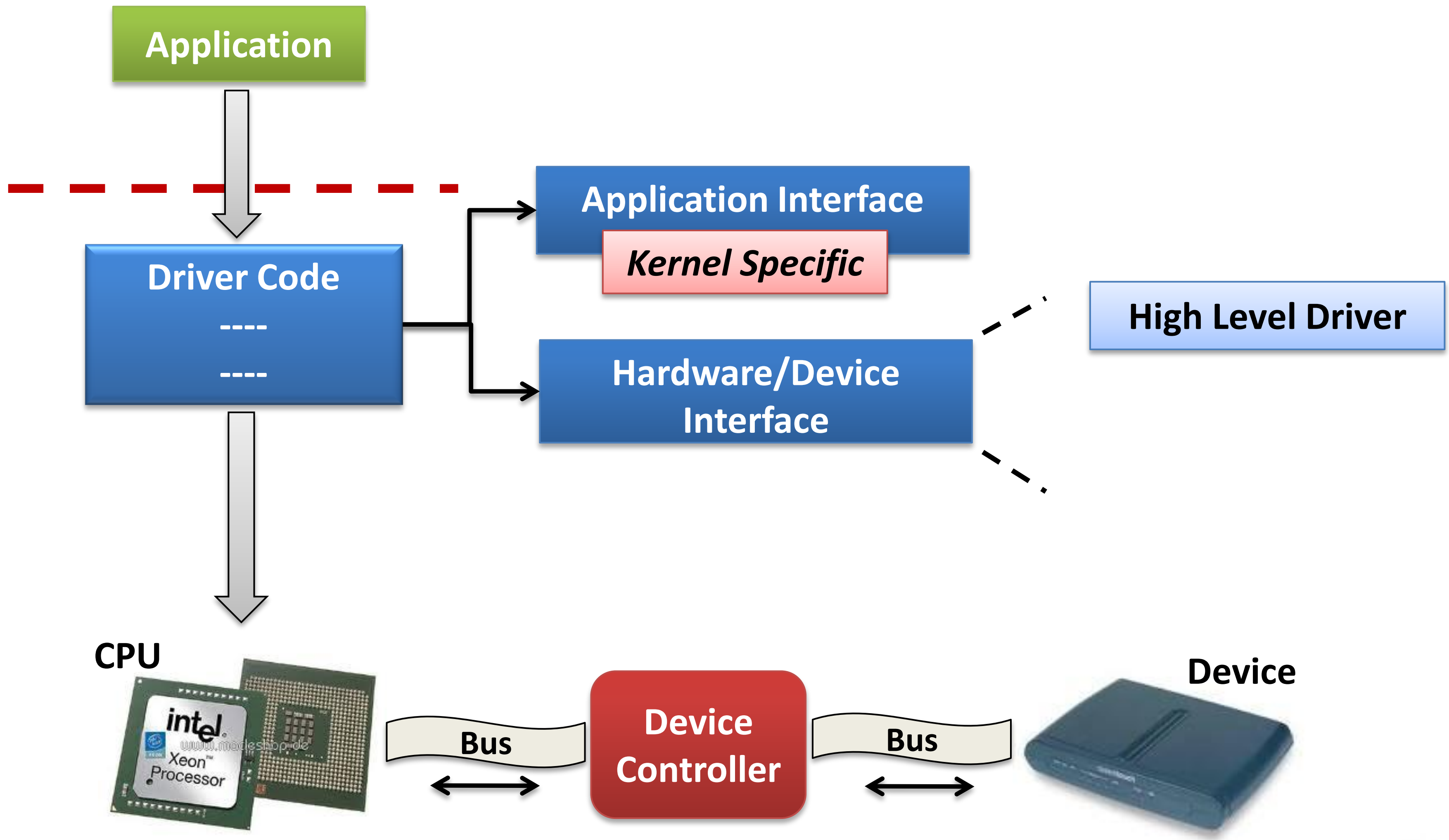
# Device Driver Framework



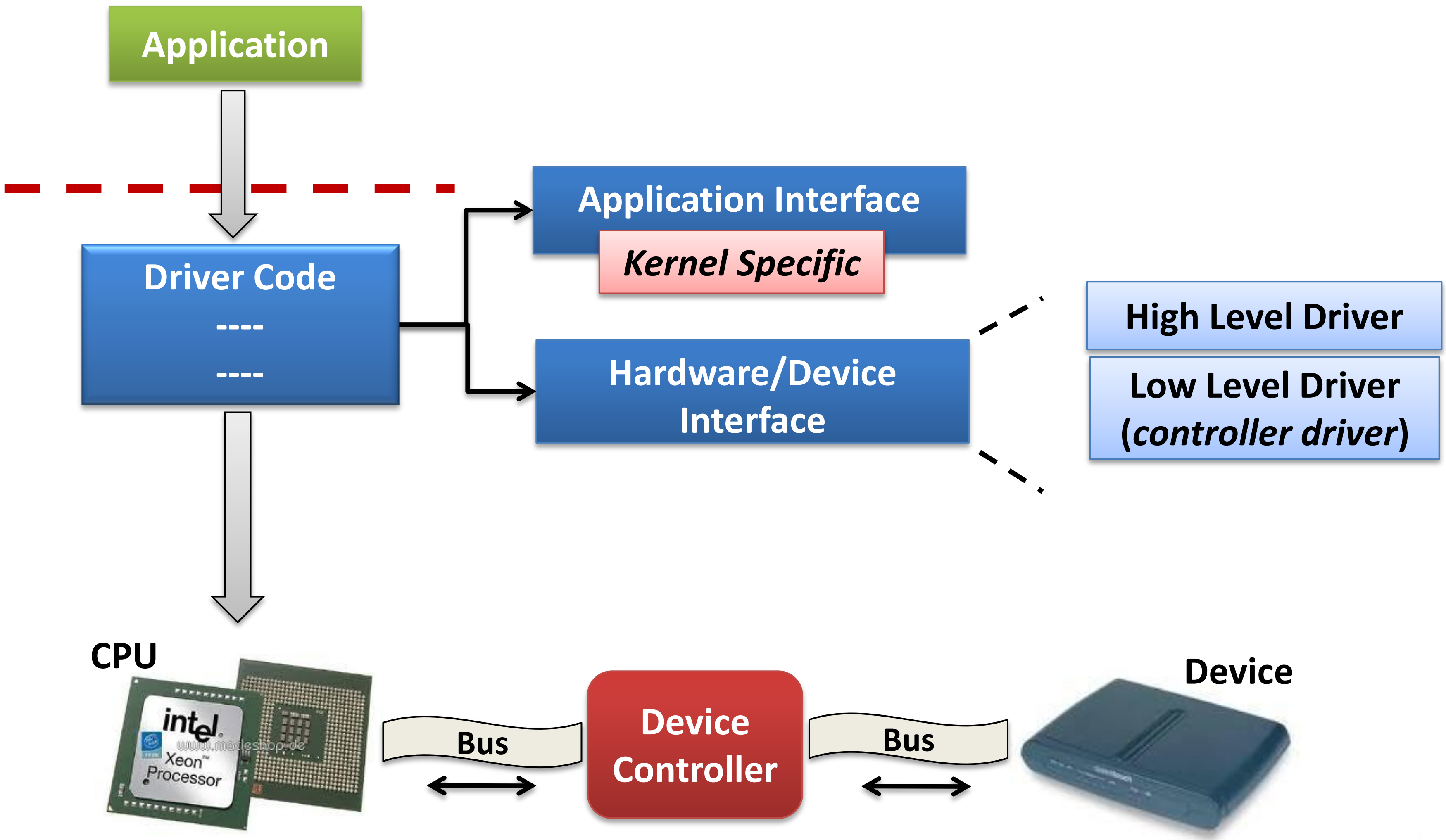
# Device Driver Framework



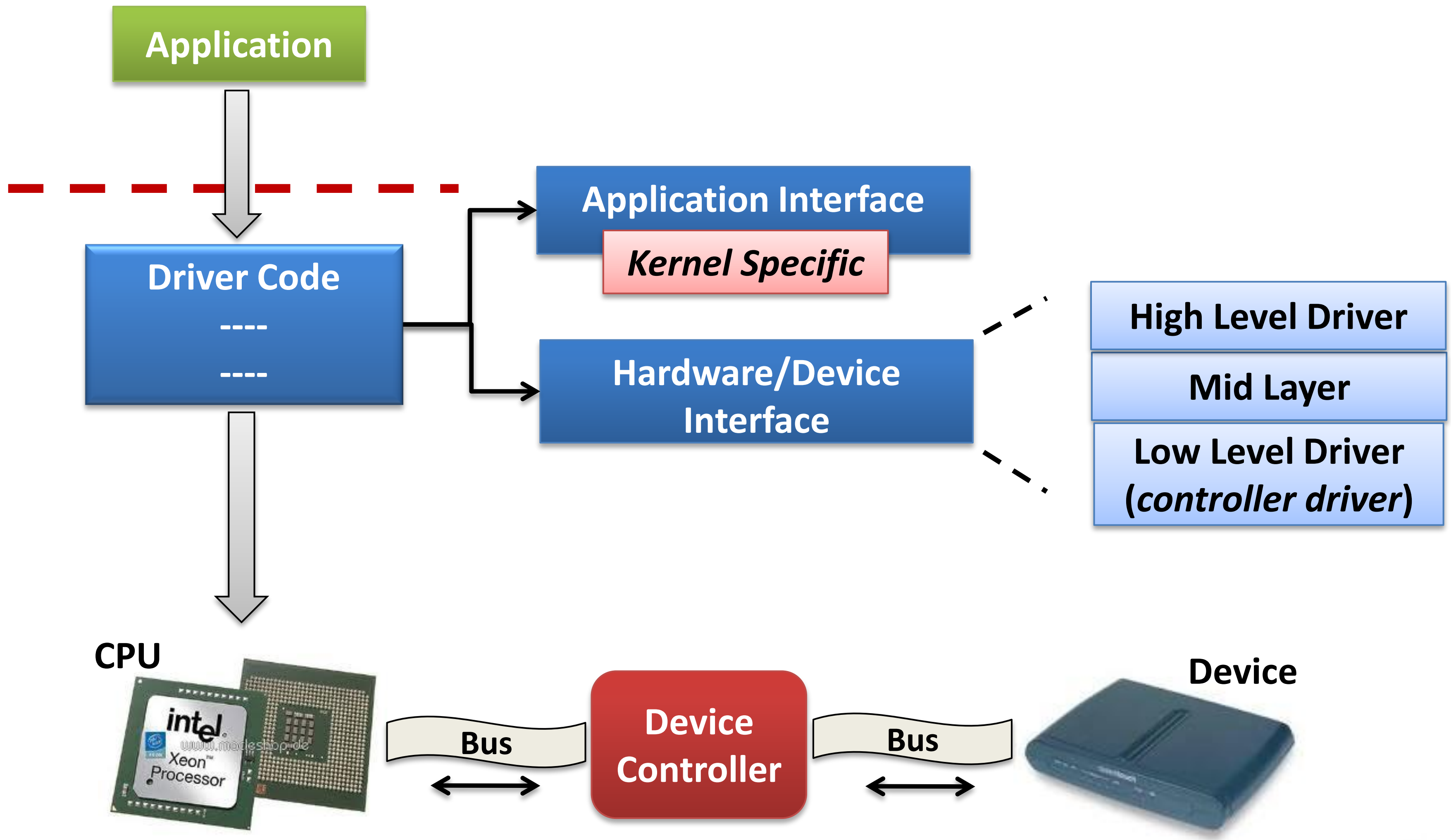
# Device Driver Framework



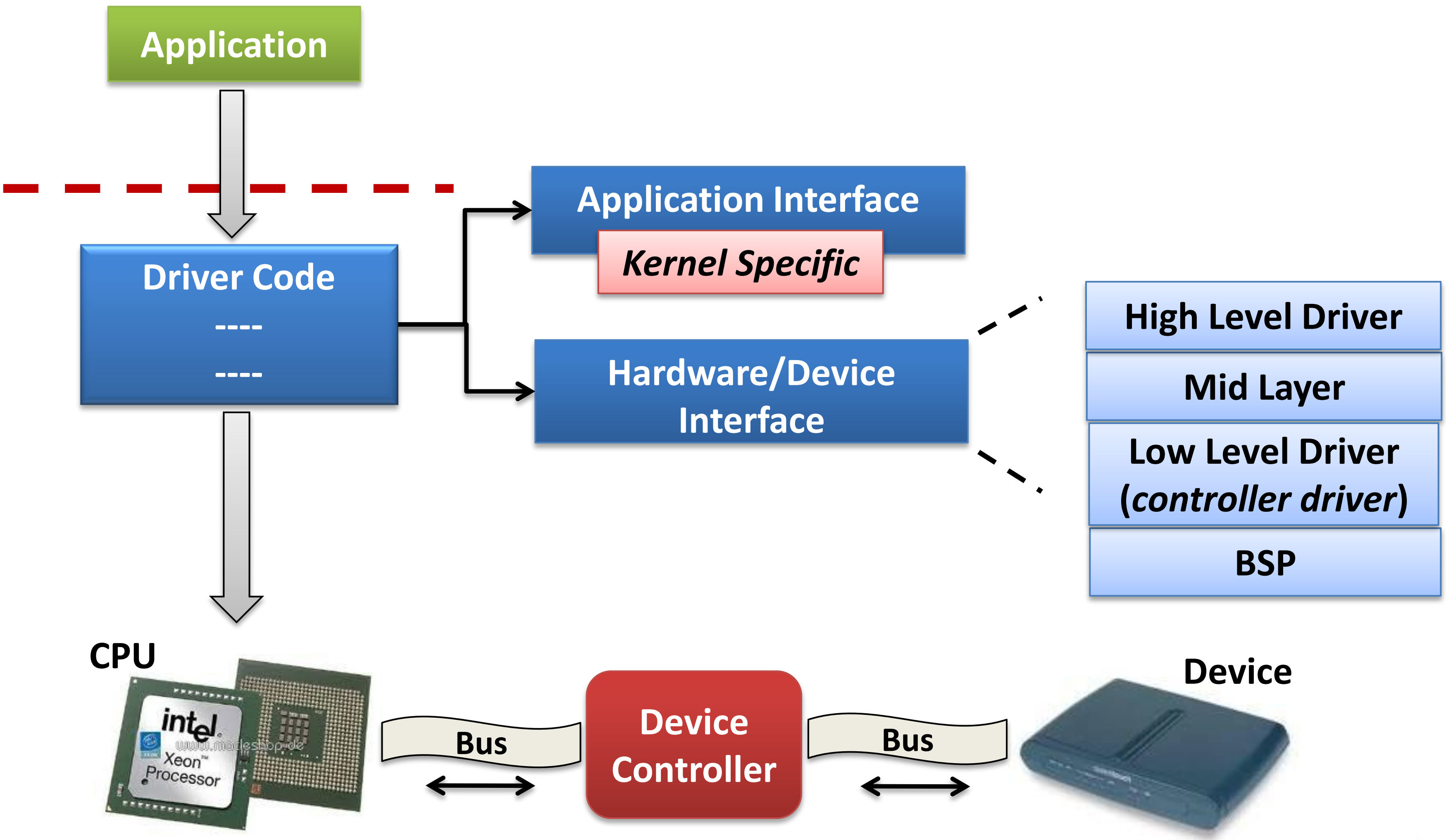
# Device Driver Framework



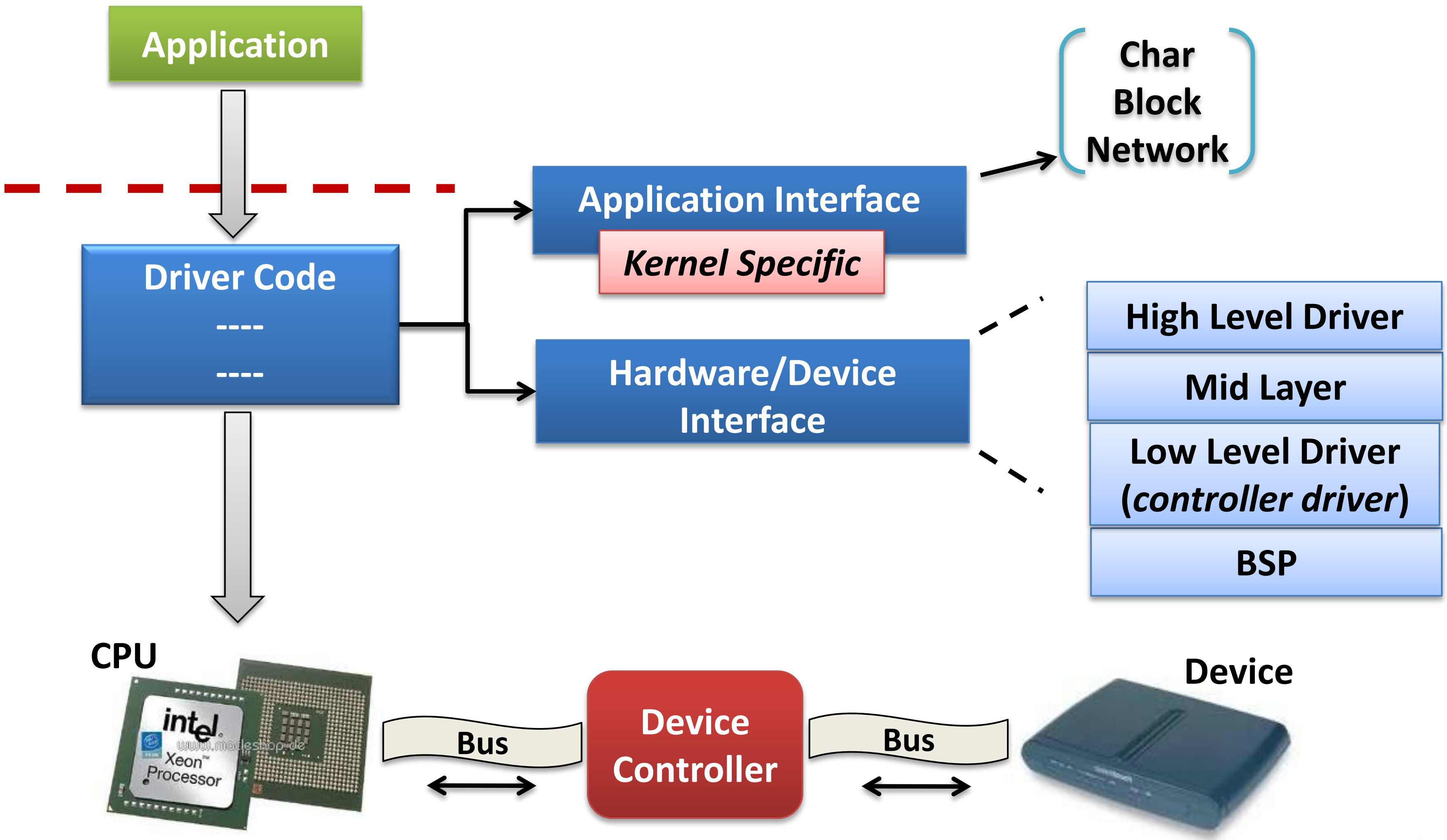
# Device Driver Framework



# Device Driver Framework



# Device Driver Framework



# Character Drivers

# Character Driver

- Except for storage device drivers, most drivers for devices with input and output flows are implemented as character drivers.
- So, most drivers you will face will be character drivers
  You will regret if you sleep during this part!

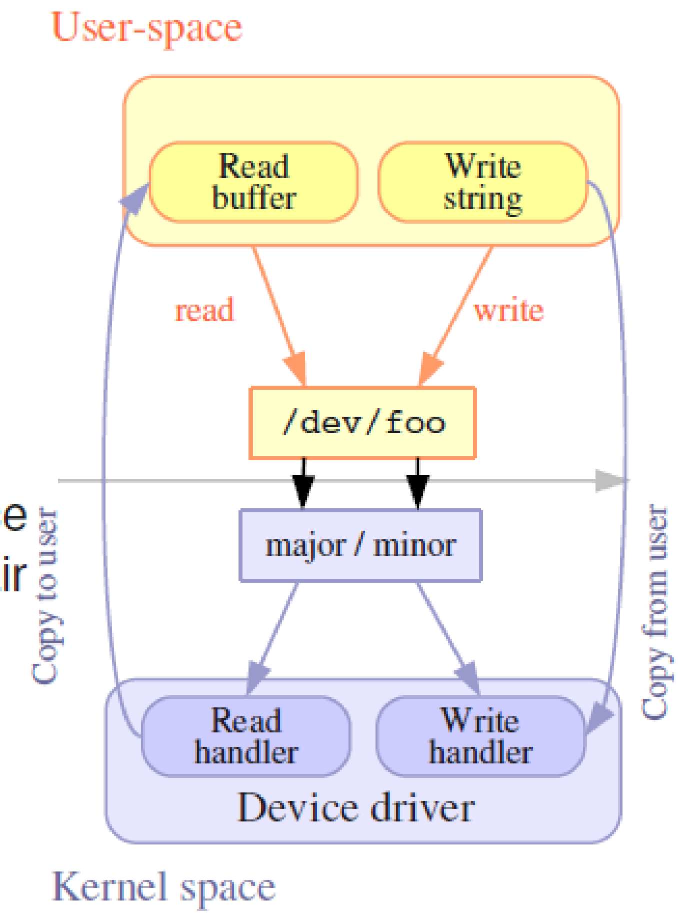
# Creating a Character Driver

## User-space needs

- The name of a device file in `/dev` to interact with the device driver through regular file operations (open, read, write, close...)

## The kernel needs

- To know which driver is in charge of device files with a given major / minor number pair
- For a given driver, to have handlers ("*file operations*") to execute when user-space opens, reads, writes or closes the device file.



# Implementing Character Driver

- Four major steps
  - Implement operations corresponding to the system calls an application can apply to a file: file operations
  - Define a `file_operations` structure associating function pointers to their implementation in your driver
  - Reserve a set of major and minors for your driver
  - Tell the kernel to associate the reserved major and minor to your file operations
- This is a very common design scheme in the Linux kernel
  - A common kernel infrastructure defines a set of operations to be implemented by a driver and functions to register your driver
  - Your driver only needs to implement this set of well-defined operations

# File Operations

- Before registering character devices, you have to define `file_operations` (called *fops*) for the device files.
- The file_operations structure is generic to all files handled by the Linux kernel. It contains many operations that aren't needed for character drivers.
- Here are the most important operations for a character driver. All of them are optional.

```
struct file_operations {
  [...]
  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
  long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
  int (*mmap) (struct file *, struct vm_area_struct *);
  int (*open) (struct inode *, struct file *);
  int (*release) (struct inode *, struct file *);
  [...]
};
```

# open() and release()

- `int foo_open (struct inode *i, struct file *f)`
  - Called when user-space opens the device file.
  - `inode` is a structure that uniquely represent a file in the system (be it a regular file, a directory, a symbolic link, a character or block device)
  - `file` is a structure created every time a file is opened. Several file structures can point to the same `inode` structure.
    - Contains informations like the current position, the opening mode, etc.
    - Has a `void *private_data` pointer that one can freely use.
    - A pointer to the file structure is passed to all other operations
- `int foo_release(struct inode *i, struct file *f)`
  - Called when user-space closes the file.

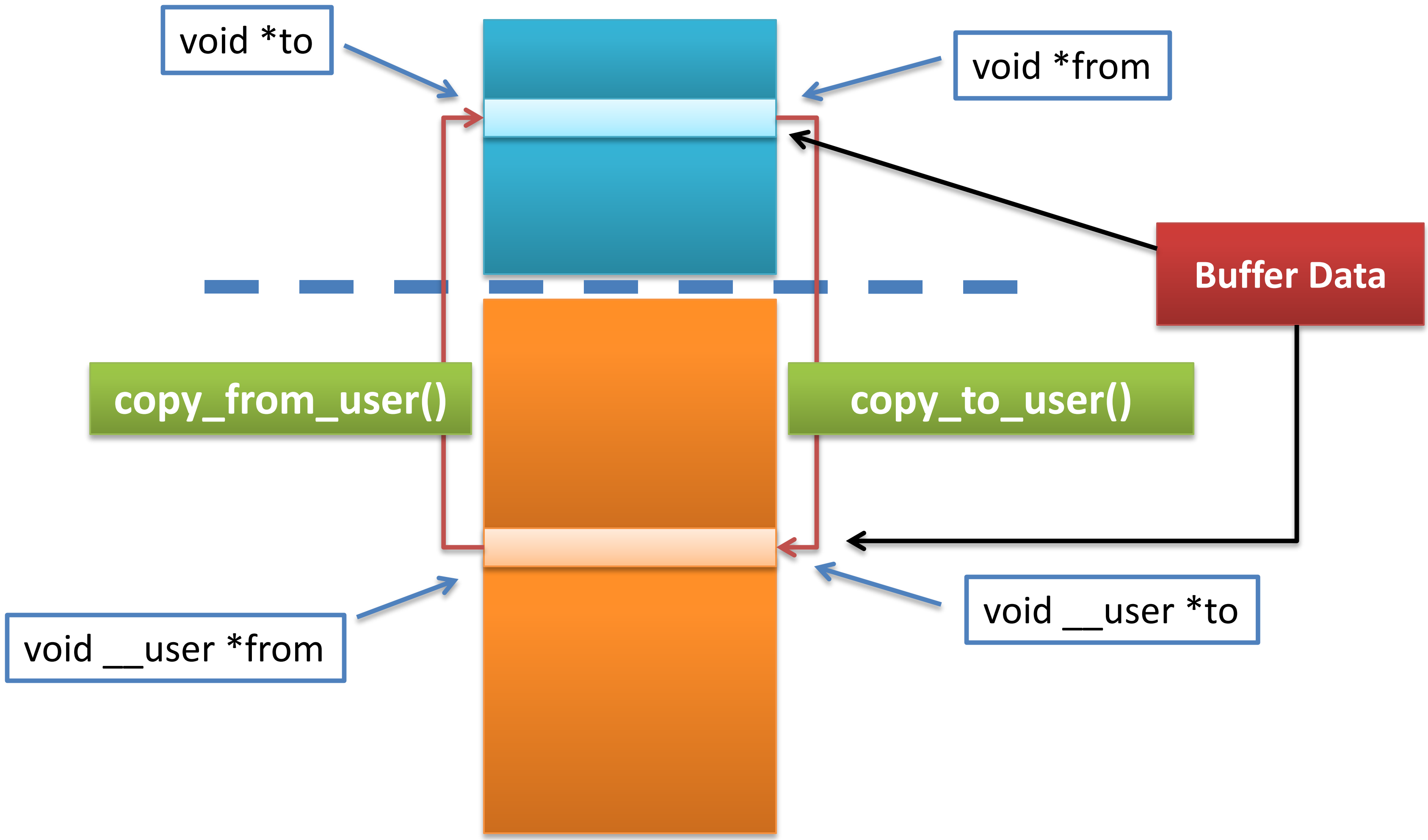
# Exchanging data with user-space

- Kernel code isn't allowed to directly access user-space memory, using `memcpy` or direct pointer dereferencing
  - Doing so does not work on some architectures
  - If the address passed by the application was invalid, the application would segfault
- To keep the kernel code portable and have proper error handling, your driver must use special kernel functions to exchange data with user-space

# Exchanging data with user-space

- A single value
  - `get_user(v, p);`
    The kernel variable v gets the value pointer by the user-space pointer p
  - `put_user(v, p);`
    The value pointed by the user-space pointer p is set to the contents of the kernel variable v.
- A buffer
  - ```
    unsigned long copy_to_user(void __user *to,
              const void *from, unsigned long n);
    ```
  - ```
    unsigned long copy_from_user(void *to,
              const void __user *from, unsigned long n);
    ```
- The return value must be checked. Zero on success, non-zero on failure. If non-zero, the convention is to return `-EFAULT`.

# Exchanging data with user-space



# read operation example

```
static ssize_t
acme_read(struct file *file, char __user *buf, size_t count, loff_t *ppos)
{
   /* The acme_buf address corresponds to a device I/O memory area */
   /* of size acme_bufsize, obtained with ioremap() */
   int remaining_size, transfer_size;

   remaining_size = acme_bufsize - (int) (*ppos); // bytes left to transfer
   if (remaining_size == 0) { /* All read, returning 0 (End Of File) */
      return 0;
   }

   /* Size of this transfer */
   transfer_size = min(remaining_size, (int) count);

   if (copy_to_user(buf /* to */, acme_buf + *ppos /* from */, transfer_size)) {
      return -EFAULT;
   } else { /* Increase the position in the open file */
      *ppos += transfer_size;
      return transfer_size;
   }
}
```

***Read method***

# write operation example

```
static ssize_t
acme_write(struct file *file, const char __user *buf, size_t count, loff_t *ppos)
{
   int remaining_bytes;

   /* Number of bytes not written yet in the device */
   remaining_bytes = acme_bufsize - (*ppos);

   if (count > remaining_bytes) {
      /* Can't write beyond the end of the device */
      return -EIO;
   }

   if (copy_from_user(acme_buf + *ppos /* to */, buf /* from */, count)) {
      return -EFAULT;
   } else {
      /* Increase the position in the open file */
      *ppos += count;
      return count;
   }
}
```

***write method***

# unlocked_ioctl()

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

- Associated to the `ioctl()` system call
  Called `unlocked` because it doesn't hold the Big Kernel Lock.
- Allows to extend the driver capabilities beyond the limited read/write API
- For example: changing the speed of a serial port, setting video output format, querying a device serial number...
- `cmd` is a number identifying the operation to perform
- `arg` is the optional argument passed as third argument of the `ioctl()` system call. Can be an integer, an address, etc.
- The semantic of `cmd` and `arg` is driver-specific.

# ioctl() example - Kernel side

```
static long phantom_ioctl(struct file *file, unsigned int cmd,
                          unsigned long arg)
{
        struct phm_reg r;
        void __user *argp = (void __user *)arg;

        switch (cmd) {
        case PHN_SET_REG:
                if (copy_from_user(&r, argp, sizeof(r)))
                        return -EFAULT;
                /* Do something */
                break;
         case PHN_GET_REG:
                if (copy_to_user(argp, &r, sizeof(r)))
                        return -EFAULT;

                /* Do something */
                break;
         default:
                return -ENOTTY;
         }

         return 0;
}
```

# ioctl() example - application side

```
int main(void)
{
   int fd, ret;
   struct phm_reg reg;

   fd = open("/dev/phantom");
   assert(fd > 0);

   reg.field1 = 42;
   reg.field2 = 67;

   ret = ioctl(fd, PHN_SET_REG, & reg);
   assert(ret == 0);

   return 0;
}
```

# File operations definition example

Defining a `file_operations` structure:

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

static struct file_operations acme_fops =
{
    .owner = THIS_MODULE,
    .read = acme_read,
    .write = acme_write,
};
```

You just need to supply the functions you implemented! Defaults for other functions (such as `open`, `release`...) are fine if you do not implement anything special.

# dev_t data type

Kernel data type to represent a major / minor number pair

- Also called a *device number*.
- Defined in `<linux/kdev_t.h>`
  Linux 2.6: 32 bit size (major: 12 bits, minor: 20 bits)
- Macro to compose the device number:
  ```
  MKDEV(int major, int minor);
  ```
- Macro to extract the minor and major numbers:
  ```
  MAJOR(dev_t dev);
  MINOR(dev_t dev);
  ```

# Registering device numbers

```
#include <linux/fs.h>
int register_chrdev_region(
     dev_t from,              /* Starting device number */
     unsigned count,          /* Number of device numbers */
     const char *name);       /* Registered name */
```

Returns 0 if the allocation was successful.

Example

```
static dev_t acme_dev = MKDEV(202, 128);


if (register_chrdev_region(acme_dev, acme_count, "acme")) {
printk(KERN_ERR "Failed to allocate device number\n");
...
```

# Registering device numbers

If you don't have fixed device numbers assigned to your driver

- Better not to choose arbitrary ones.
  There could be conflicts with other drivers.
- The kernel API offers a `alloc_chrdev_region` function
  to have the kernel allocate free ones for you. You can find the allocated major number in `/proc/devices`.

# Information on registered numbers

Registered devices are visible in `/proc/devices`:

```
Character devices:        Block devices:
  1 mem                     1 ramdisk
  4 /dev/vc/0               3 ide0
  4 tty                     8 sd
  4 ttyS                    9 md
  5 /dev/tty               22 ide1
  5 /dev/console           65 sd
  5 /dev/ptmx              66 sd
  6 lp                     67 sd
 10 misc                   68 sd
 13 input
 14 sound
 ...
```

Major number

Registered name

# Character Driver Registration

- The kernel represents character drivers with a `cdev` structure
- Declare this structure globally (within your module):

```
#include <linux/cdev.h>
static struct cdev acme_cdev;
```

- In the init function, initialize the structure:

```
cdev_init(&acme_cdev, &acme_fops);
```

# Character Driver Registration

- Then, now that your structure is ready, add it to the system:

```
int cdev_add(
      struct cdev *p,      /* Character device structure */
      dev_t dev,           /* Starting device major / minor number */
      unsigned count);     /* Number of devices */
```

- After this function call, the kernel knows the association between the major/minor numbers and the file operations. Your device is ready to be used!
- Example (continued):

```
if (cdev_add(&acme_cdev, acme_dev, acme_count)) {
   printk (KERN_ERR "Char driver registration failed\n");
...
```

# Character Device Un-registration

- First delete your character device:

```
void cdev_del(struct cdev *p);
```

- Then, and only then, free the device number:

```
void unregister_chrdev_region(dev_t from,
unsigned count);
```

- Example (continued):

```
cdev_del(&acme_cdev);
unregister_chrdev_region(acme_dev, acme_count);
```

# Linux error codes

- The kernel convention for error management is
  - Return 0 on success
    `return 0;`
  - Return a negative error code on failure
    `return -EFAULT;`
- Error codes
  - `include/asm-generic/errno-base.h`
  - `include/asm-generic/errno.h`

# Char driver example summary

```
static void *acme_buf;
static int acme_bufsize=8192;

static int acme_count=1;
static dev_t acme_dev = MKDEV(202,128);

static struct cdev acme_cdev;

static ssize_t acme_write(...) {...}

static ssize_t acme_read(...) {...}

static struct file_operations acme_fops =
{
    .owner = THIS_MODULE,
    .read = acme_read,
    .write = acme_write
};
```

# Char driver example summary

Shows how to handle errors and deallocate resources in the right order!

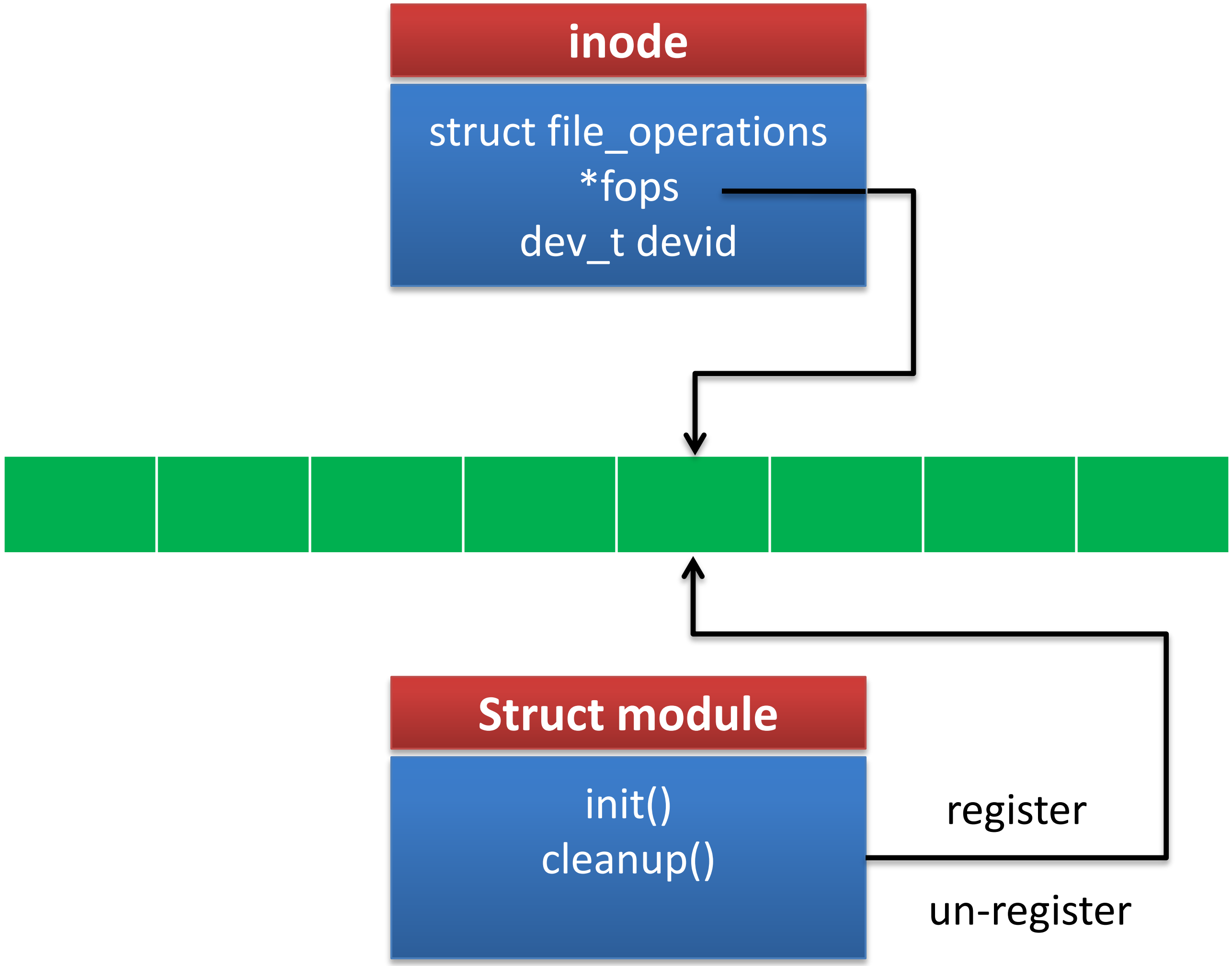
```
static int __init acme_init(void)
{
    int err;
    acme_buf = ioremap (ACME_PHYS,
                        acme_bufsize);

    if (!acme_buf) {
       err = -ENOMEM;
       goto err_exit;
    }

    if (register_chrdev_region(acme_dev,
                     acme_count, "acme")) {
       err=-ENODEV;
       goto err_free_buf;
    }

    cdev_init(&acme_cdev, &acme_fops);

    if (cdev_add(&acme_cdev, acme_dev,
                 acme_count)) {
       err=-ENODEV;
       goto err_dev_unregister;
    }
```

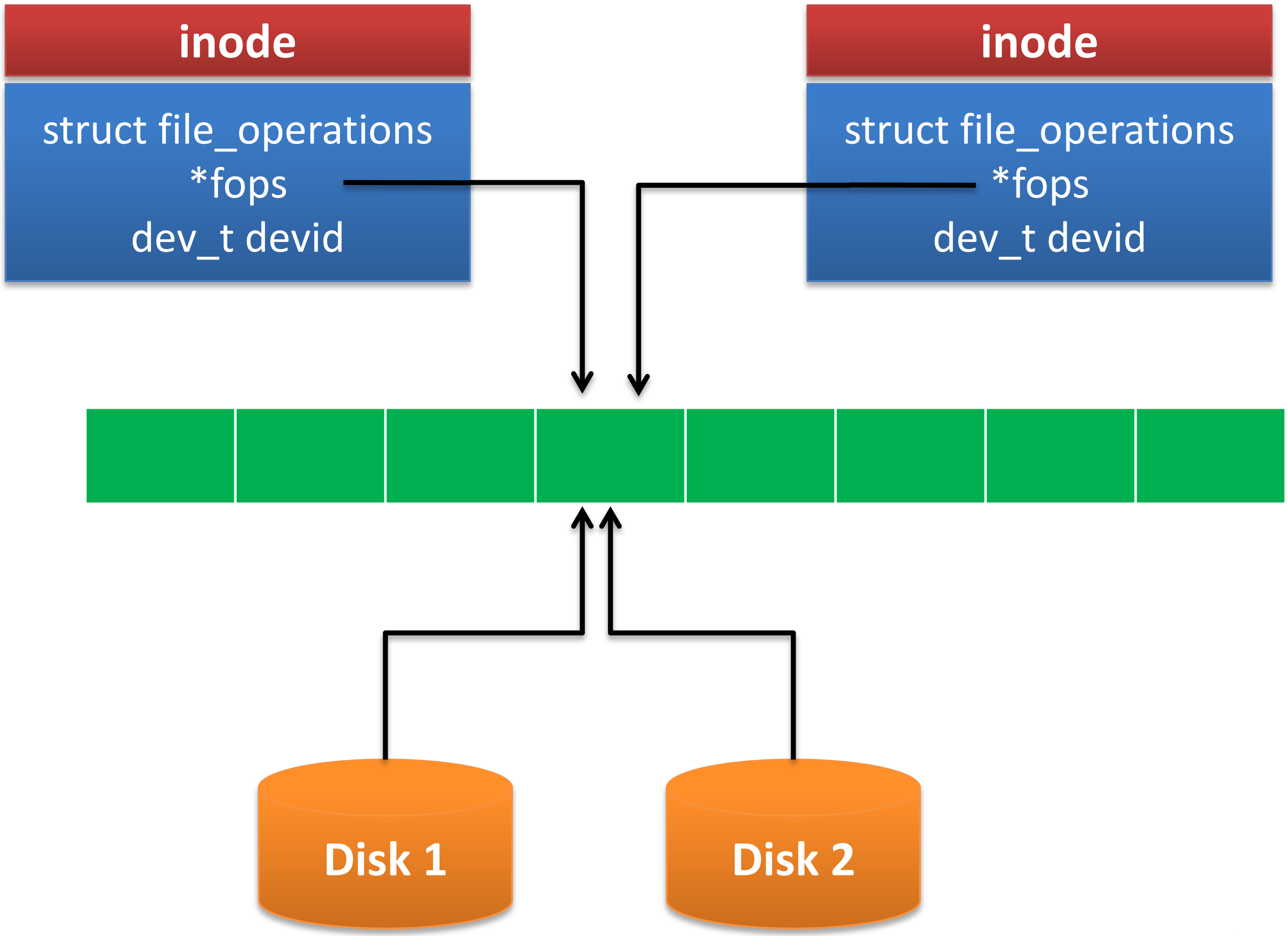
```
    return 0;

    err_dev_unregister:
        unregister_chrdev_region(
           acme_dev, acme_count);
    err_free_buf:
        iounmap(acme_buf);
    err_exit:
        return err;
}

static void __exit acme_exit(void)
{
    cdev_del(&acme_cdev);
    unregister_chrdev_region(acme_dev,
                   acme_count);
    iounmap(acme_buf);
}
```

VEDA Solutions
Inspired by open Source

# Char driver summary

| | |
|---|---|
| Character driver writer<br>- Define the file operations callbacks for the device file: `read`, `write`, `ioctl`...<br>- In the module init function, reserve major and minor numbers with `register_chrdev_region()`, init a `cdev` structure with your file operations and add it to the system with `cdev_add()`.<br>- In the module exit function, call `cdev_del()` and `unregister_chrdev_region()` | Kernel |
| System administration<br>- Load the character driver module<br>- Create device files with matching major and minor numbers if needed<br>The device file is ready to use! | User-space |
| System user<br>- Open the device file, read, write, or send ioctl's to it. | User-space |
| Kernel<br>- Executes the corresponding file operations | Kernel |


inode
struct file_operations
*fops
dev_t devid
Struct module
init()
cleanup()
register
un-register


inode
struct file_operations
*fops
dev_t devid
inode
struct file_operations
*fops
dev_t devid
Disk 1
Disk 2

# Thank You