

# Device Drivers

Prof. Stephen A. Edwards

Columbia University

Spring 2025

# Linux Operating System Structure

## Applications

Function Calls ↓    Callbacks ↑

## Libraries

System Calls ↓    Signals ↑

## The Kernel

Processes    Scheduling    Networking  
Memory Management    File Systems  
Device Drivers

iowrite32(), etc. ↓    Interrupts ↑

## Hardware

Busses    Memory  
Peripherals

# User Space vs. Kernel Space

## User Space

Process abstraction central to most OSes

Independent PC, registers, and memory

Virtual memory hardware isolates processes, OS

Processes run in limited-resource “user mode”

Bug in a process only affects the process

---

## Kernel Space

Kernel runs in “supervisor mode” with no access limitations

Bugs in kernel code take down the whole system

# Unix Device Driver Model

“Everything is a file”

By convention, special “device” files stored in /dev

Created by the mknod command or dynamically

```
# ls -Ggl --time-style=+ \
/dev/sd{a,a1,a2,b} /dev/tty{,1,2} \
/dev/ttyUSB0
```

|                  |            |   |        |              |                                     |
|------------------|------------|---|--------|--------------|-------------------------------------|
| Block Device     | brw-rw---- | 1 | 8, 0   | /dev/sda     | First SCSI drive                    |
|                  | brw-rw---- | 1 | 8, 1   | /dev/sda1    | First partition of first SCSI drive |
|                  | brw-rw---- | 1 | 8, 2   | /dev/sda2    | Second SCSI drive                   |
| Character Device | brw-rw---- | 1 | 8, 16  | /dev/sdb     |                                     |
|                  | crw-rw-rw- | 1 | 5, 0   | /dev/tty     | Current terminal                    |
|                  | crw-rw---- | 1 | 4, 1   | /dev/tty1    | Second terminal                     |
|                  | crw-rw---- | 1 | 4, 2   | /dev/tty2    |                                     |
|                  | crw-rw---- | 1 | 188, 0 | /dev/ttyUSB0 | First USB terminal                  |

Owner Group    World permissions    Major Device Number    Minor Device Number

## /proc/devices

Virtual file with a list of device drivers by major number

```
# cat /proc/devices
```

Character devices:

```
4 /dev/vc/0
4 tty
4 ttyS
5 /dev/tty
10 misc
188 ttyUSB
```

Block devices:

```
8 sd
```

More virtual files and directories:

```
# ls /sys/bus
```

```
amba          cpu          hid  mdio_bus  platform  sdio  soc  usb
clocksource  event_source i2c  mmc       scsi      serio spi
```

```
# ls /sys/class/misc
```

```
cpu_dma_latency  network_latency  network_throughput  psaux  vga_ball
```

# Kernel Modules

Device drivers can be compiled into the kernel

Really annoying for, e.g., “hotplug” USB devices

Solution: dynamically linked kernel modules

Similar to shared libraries/DLLs

```
# lsmod
Module                Size  Used by
# insmod vga_ball.ko
# lsmod
Module                Size  Used by
vga_ball              16384  0
# rmmod vga_ball
```

---

4K stack limit (don't use recursion)

No standard library; many equivalent functions available

*init* and *exit* functions compulsory; called when loaded/unloaded

# Our First Driver

```
#include <linux/module.h>
#include <linux/version.h>
#include <linux/kernel.h>

static int __init ofd_init(void)
{
    pr_info("ofd registered");
    return 0;
}

static void __exit ofd_exit(void)
{
    pr_info("ofd unregistered");
}

module_init(ofd_init);
module_exit(ofd_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen Edwards <sedwards@cs.columbia.edu>");
MODULE_DESCRIPTION("Our First Driver: Nothing");
```

## Debugging: pr\_info and friends

In the kernel, there's no *printf* (no `stdio.h`)

*printk* the traditional replacement:

```
printk(KERN_ERR "something went wrong, return code: %d\n", ret);
```

KERN\_ERR just the string "<3>"

Now deprecated in favor of equivalent

```
pr_info("Information\n");  
pr_err("Error\n");  
pr_alert("Really big problem\n");  
pr_emerg("Life as we know it is over\n");
```



# Kernel Logging

How do you see the output of *printk* et al.?

Send kernel logging to the console:

```
# echo 8 > /proc/sys/kernel/printk  
# insmod vga_ball.ko  
[ 1533.730421] vga_ball: init
```

Diagnostic messages from *dmesg*:

```
# dmesg | tail -4  
[ 990.780462] vga_ball: init  
[ 1530.230146] vga_ball: exit  
[ 1533.730421] vga_ball: init
```

## Copying to/from user memory

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

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

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

Checks that pointers are valid before copying memory between user and kernel space

Return number of bytes remaining to transfer (0 on success)

```
#include <linux/module.h>  // A Very Simple Character Device
#include <linux/printk.h>
#include <linux/fs.h>
#include <linux/cdev.h>
#include <linux/uaccess.h>

#define MY_MAJOR 60
#define MY_MINOR 0

static int schar_open(struct inode *inode, struct file *file)
{
    pr_info("schar open\n");
    return 0;
}

static int schar_release(struct inode *inode, struct file *f)
{
    pr_info("schar release\n");
    return 0;
}

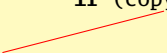
static ssize_t schar_write(struct file *f, const char __user *buf,
                           size_t count, loff_t *f_pos)
{
    pr_info("schar write %zu\n", count);
    return 0;
}
```

```
static char welcome_message[] = "Hello World!\n";
#define WELCOME_MESSAGE_LEN 13

static ssize_t schar_read(struct file *f, char __user *buf,
                          size_t count, loff_t *f_pos)
{
    pr_info("schar read %zu\n", count);
    if ((*f_pos == 0) && count > WELCOME_MESSAGE_LEN) {
        if (copy_to_user(buf, welcome_message,
                          WELCOME_MESSAGE_LEN)) {
            return -EFAULT;
        }
        *f_pos = WELCOME_MESSAGE_LEN;
        return WELCOME_MESSAGE_LEN;
    }
    return 0;
}

static long schar_ioctl(struct file *f, unsigned int cmd,
                        unsigned long arg)
{
    pr_info("schar ioctl %d %lu\n", cmd, arg);
    return 0;
}
```

Send data  
to userspace



```
static struct file_operations schar_fops = {
    .owner          = THIS_MODULE,
    .open           = schar_open,
    .release        = schar_release,
    .read           = schar_read,
    .write          = schar_write,
    .unlocked_ioctl = schar_ioctl };

```

Function pointer called by each operation

```
static struct cdev schar_cdev = { .owner = THIS_MODULE,
                                   .ops   = &schar_fops };

```

```
static int __init schar_init(void) {
    int result;
    dev_t dev = MKDEV(MY_MAJOR, 0);
    pr_info("schar init\n");
    result = register_chrdev_region(dev, 2, "schar");
    if (result < 0) {
        pr_warn("schar: unable to get major %d\n", MY_MAJOR);
        return result;
    }
    cdev_init(&schar_cdev, &schar_fops);
    result = cdev_add(&schar_cdev, dev, 1);
    if (result < 0) {
        unregister_chrdev_region(dev, 2);
        pr_notice("schar: unable to add cdev\n");
        return result;
    }
    return 0;
}

```

Request minor numbers 0-1

```
static void __exit schar_exit(void)
{
    cdev_del(&schar_cdev);
    unregister_chrdev_region(MKDEV(MY_MAJOR, 0), 2);
    pr_info("schar unregistered\n");
}

module_init(schar_init);
module_exit(schar_exit);

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Stephen Edwards <sedwards@cs.columbia.edu>");
MODULE_DESCRIPTION("Really Simple Character Driver");

// End of the Very Simple Character Device
```

## Simple Char Driver: Behavior

```
# echo 8 > /proc/sys/kernel/printk
# cd /dev
# mknod schar c 60 0
# ls -Ggl --time-style=+ schar
crw-r--r-- 1 60, 0  schar
# cd ~/schar
# insmod schar.ko
schar init
# cat /dev/schar > foo
schar open
schar read 65536
schar read 65536
schar release
# cat foo
Hello World!
# rmmod schar.ko
schar unregistered
```

# The ioctl() System Call

```
#include <sys/ioctl.h>

int ioctl(int fd, int request, void *argp);
```

A catch-all for “out-of-band” communication with a device

E.g., setting the baud rate of a serial port, reading and setting a real-time clock

Ultimately passes a number and a userspace pointer to a device driver

ioctl requests include some “magic numbers” to prevent accidental invocation. Macros do the encoding:

```
_IO(magic, number)      /* No argument */
_IOW(magic, number, type) /* Data sent to driver */
_IOR(magic, number, type) /* Data returned by driver */
_IOWR(magic, number, type) /* Data sent and returned */
```



# The Misc Class

Thin layer around character devices

Major number 10; minor numbers assigned dynamically

Subsystem automatically creates special file in */dev* directory

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

```
struct miscdevice {  
    int minor; /* MISC_DYNAMIC_MINOR assigns it dynamically */  
    const char name; /* e.g., vga_ball */  
    struct file_operations *fops;  
};
```

```
int misc_register(struct miscdevice *misc);  
int misc_deregister(struct miscdevice *misc);
```

```
# ls -Ggl --time-style=+ /dev/vga_ball  
crw----- 1 10, 60 /dev/vga_ball  
# cat /proc/misc  
60 vga_ball  
61 network_throughput  
62 network_latency  
63 cpu_dma_latency  
1 psaux
```

# The Platform Bus

Linux has subsystems for busses that know their devices (lsusb, lspci, etc.)  
The "Platform Bus" is for everything else

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

struct platform_driver {
    int (*probe)(struct platform_device *);
    int (*remove)(struct platform_device *);
    void (*shutdown)(struct platform_device *);
    int (*suspend)(struct platform_device *, pm_message_t state);
    int (*resume)(struct platform_device *);
    struct device_driver driver;
    const struct platform_device_id *id_table;
};

int platform_driver_register(struct platform_driver *driver);
/* Or, for non hot-pluggable devices */
int platform_driver_probe(struct platform_driver *driver,
                          int (*probe)(struct platform_device *));

void platform_driver_unregister(struct platform_driver *driver);
```

# Device Tree

Where are our device's registers?

```
#define PARPORT_BASE 0x378
```

Compiling this into the kernel is too fragile: different kernel for each system?

Alternative: a standard data structure holding a description of the hardware platform.

Device Tree: Standard derived from Open Firmware, originally from Sun

<http://devicetree.org/>

[http://devicetree.org/Device\\_Tree\\_Usage](http://devicetree.org/Device_Tree_Usage)

<http://elinux.org/images/a/a3/Elce2013-petazzoni-devicetree-for-dummies.pdf>

<http://lwn.net/Articles/572692/>

<http://xillybus.com/tutorials/device-tree-zynq-1>

# Raspberry Pi DTS Excerpt

Uses a Broadcom BCM2835 SoC with a 700 MHz ARM processor

```
/ {
    compatible = "brcm,bcm2835";
    model = "BCM2835";
    interrupt-parent = <&intc>;

    soc {
        compatible = "simple-bus";
        #address-cells = <1>; from      to      size
        #size-cells = <1>; address  address
        ranges = <0x7e000000 0x20000000 0x02000000>;

        uart@20191000 {
            compatible = "brcm,bcm2835-pl011",
            base        "arm,pl011", "arm,primecell";
            address
            reg = <0x7e201000 0x1000>;

            interrupts = <2 25>; size
            clock-frequency = <3000000>;
        };
    };
};
}
```

## Vga\_ball in the soc\_system DTS

Connected through the "lightweight AXI bridge"

Avalon bus address 0 appears to the ARM at 0xff200000

```
sopc0: socp@0 {  
    device_type = "soc";  
  
    hps_0_bridges: bridge@0xc0000000 {  
        compatible = "altr,bridge-18.1", "simple-bus";  
        reg = <0xc0000000 0x20000000>,  
            <0xff200000 0x00200000>;  
        reg-names = "axi_h2f", "axi_h2f_lw";  
        clocks = <&clk_0 &clk_0>;  
        clock-names = "h2f_axi_clock", "h2f_lw_axi_clock";  
        #address-cells = <2>;  
        #size-cells = <1>;  
        ranges = <0x00000001 0x00000000 0xff200000 0x00000008>;  
  
        vga_ball_0: vga@0x100000000 {  
            compatible = "csee4840,vga_ball-1.0";  
            reg = <0x00000001 0x00000000 0x00000008>;  
            clocks = <&clk_0>;  
        };  
    };  
};
```

# Accessing the Device Tree

```
#include <linux/of.h>  /* "Open Firmware" */
#include <linux/of_address.h>

/* Table of "compatible" values to search for */
static const struct of_device_id vga_ball_of_match[] = {
    { .compatible = "csee4840,vga_ball-1.0" },
    {}
};

MODULE_DEVICE_TABLE(of, vga_ball_of_match);

/* Platform device info */
static struct platform_driver vga_ball_driver = {
    .driver = {
        .name    = "vga_ball",
        .owner    = THIS_MODULE,
        .of_match_table = of_match_ptr(vga_ball_of_match),
    },
    .remove = __exit_p(vga_ball_remove),
};

/* Locate a device's registers, return a pointer to their base */
void __iomem *of_iomap(struct device_node *node, int index);
```

# I/O Memory Management

Resource allocation a central OS facility

Interface for requesting/releasing memory regions:

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

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

void release_mem_region(unsigned long start, unsigned long extent);
```

# I/O Memory Access

Mapping I/O regions in memory; accessing them:

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

void *ioremap(unsigned long offset, unsigned long size);
void iounmap(void *addr);

u8 ioread8( const __iomem *addr);
u16 ioread16(const __iomem *addr);
u32 ioread32(const __iomem *addr);

void iowrite8( u8 val, void __iomem *addr);
void iowrite16(u16 val, void __iomem *addr);
void iowrite32(u32 val, void __iomem *addr);
```



## /proc/iomem

```
# insmod vga_ball.ko
vga_ball: init
# cat /proc/iomem
00000000-3fffffff : System RAM
    00008000-00bfffff : Kernel code
    00d00000-00da24ff : Kernel data
ff200000-ff200007 : vga_ball
ff702000-ff703fff : ethernet@0xff702000
ff704000-ff704fff : flash@0xff704000
ff706000-ff706fff : axi_slave0
ff708000-ff7080ff : gpio@0xff708000
ff709000-ff7090ff : gpio@0xff709000
ff70a000-ff70a0ff : gpio@0xff70a000
ffb40000-ffb7ffff : usb@0xffb40000
ffb90000-ffb900ff : axi_slave1
ffc02000-ffc0201f : serial
ffc04000-ffc040ff : i2c@0xffc04000
```

# The Vga\_ball Driver: Header File

```
#ifndef _VGA BALL_H
#define _VGA BALL_H

#include <linux/ioctl.h>

typedef struct {
    unsigned char red, green, blue;
} vga_ball_color_t;

typedef struct {
    vga_ball_color_t background;
} vga_ball_arg_t;

#define VGA BALL_MAGIC 'q'

/* ioctls and their arguments */
#define VGA BALL_WRITE_BACKGROUND \
    _IOW(VGA BALL_MAGIC, 1, vga_ball_arg_t *)
#define VGA BALL_READ_BACKGROUND \
    _IOR(VGA BALL_MAGIC, 2, vga_ball_arg_t *)

#endif
```

## The Vga\_ball Driver: write\_background

```
#include <linux/module.h>
/* ... many more #includes ... */
#include <linux/uaccess.h>
#include "vga_ball.h"
#define DRIVER_NAME "vga_ball"

/* Device registers */
#define BG_RED(x) (x)
#define BG_GREEN(x) ((x)+1)
#define BG_BLUE(x) ((x)+2)

struct vga_ball_dev {
    struct resource res; /* Resource: our registers */
    void __iomem *virtbase; /* Where registers are in memory */
    vga_ball_color_t background;
} dev;

static void write_background(vga_ball_color_t *background)
{
    iowrite8(background->red, BG_RED(dev.virtbase) );
    iowrite8(background->green, BG_GREEN(dev.virtbase) );
    iowrite8(background->blue, BG_BLUE(dev.virtbase) );
    dev.background = *background;
}
```

```
static long vga_ball_ioctl(struct file *f, unsigned int cmd,
                           unsigned long arg)
{
    vga_ball_arg_t vla;

    switch (cmd) {
    case VGA BALL_WRITE_BACKGROUND:
        if (copy_from_user(&vla, (vga_ball_arg_t *) arg,
                           sizeof(vga_ball_arg_t)))
            return -EACCES;
        write_background(&vla.background);
        break;

    case VGA BALL_READ_BACKGROUND:
        vla.background = dev.background;
        if (copy_to_user((vga_ball_arg_t *) arg, &vla,
                           sizeof(vga_ball_arg_t)))
            return -EACCES;
        break;

    default:
        return -EINVAL;
    }

    return 0;
}
```

# The Vga\_ball Driver: file\_operations

```
static const struct file_operations vga_ball_fops = {  
    .owner          = THIS_MODULE,  
    .unlocked_ioctl = vga_ball_ioctl,  
};  
  
static struct miscdevice vga_ball_misc_device = {  
    .minor          = MISC_DYNAMIC_MINOR,  
    .name           = DRIVER_NAME,  
    .fops           = &vga_ball_fops,  
};
```

## The Vga\_ball Driver: vga\_ball\_probe

```
static int __init vga_ball_probe(struct platform_device *pdev)
{
    vga_ball_color_t beige = { 0xf9, 0xe4, 0xb7 };
    int ret;

    /* Register ourselves as a misc device: creates /dev/vga_ball */
    ret = misc_register(&vga_ball_misc_device);

    /* Get the address of our registers from the device tree */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }

    /* Make sure we can use these registers */
    if (request_mem_region(dev.res.start, resource_size(&dev.res),
                           DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

    // continued on next slide...
}
```

```
/* Arrange access to our registers */
dev.virtbase = of_iomap(pdev->dev.of_node, 0);
if (dev.virtbase == NULL) {
    ret = -ENOMEM;
    goto out_release_mem_region;
}
```

```
/* Set an initial color */
write_background(&beige);

return 0;
```

```
out_release_mem_region:
    release_mem_region(dev.res.start, resource_size(&dev.res));
out_deregister:
    misc_deregister(&vga_ball_misc_device);
    return ret;
}
```

```
static int vga_ball_remove(struct platform_device *pdev)
{
    iounmap(dev.virtbase);
    release_mem_region(dev.res.start, resource_size(&dev.res));
    misc_deregister(&vga_ball_misc_device);
    return 0;
}
```

```
static const struct of_device_id vga_ball_of_match[] = {
    { .compatible = "csee4840,vga_ball-1.0" },
    {}},
};
MODULE_DEVICE_TABLE(of, vga_ball_of_match);

static struct platform_driver vga_ball_driver = {
    .driver = {
        .name      = DRIVER_NAME,
        .owner      = THIS_MODULE,
        .of_match_table = of_match_ptr(vga_ball_of_match),
    },
    .remove = __exit_p(vga_ball_remove),
};

static int __init vga_ball_init(void)
{
    pr_info(DRIVER_NAME ": init\n");
    return platform_driver_probe(&vga_ball_driver, vga_ball_probe);
}

static void __exit vga_ball_exit(void)
{
    platform_driver_unregister(&vga_ball_driver);
    pr_info(DRIVER_NAME ": exit\n");
}
```



# The Vga\_ball Driver

```
module_init(vga_ball_init);  
module_exit(vga_ball_exit);  
  
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Stephen A. Edwards, Columbia University");  
MODULE_DESCRIPTION("VGA ball driver");
```

## References

<http://free-electrons.com/>

<http://www.opersys.com/training/linux-device-drivers>

Rubini, Corbet, and Kroah-Hartman, *Linux Device Drivers*, 3ed, O'Reilly

<https://lwn.net/Kernel/LDD3/>

The Linux Kernel Source, and its Documentation/driver-model directory.