

Device Drivers

Prof. Stephen A. Edwards

Columbia University

Spring 2016

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

Buses Memory
Peripherals

User Space vs. Kernel 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 takes down the process only

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
```

Owner	Group	World permissions	Major Device Number	Minor Device Number	Device Name	Description
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
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

/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
```

```
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_led
```

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_led.ko
# lsmod
Module                Size  Used by
vga_led               1814  0
# rmmmod vga_led
```

4K stack limit (don't use recursion)

No standard library; many replacements 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
```

Diagnostic messages from *dmesg*:

```
# dmesg | tail -4  
init: tty1 main process (933) killed by TERM signal  
vga_led: init  
vga_led: exit  
vga_led: init
```

/var/log/syslog

```
# tail -3 /var/log/syslog  
Jan  1 07:28:11 linaro-nano kernel: vga_led: init  
Jan  1 07:49:57 linaro-nano kernel: vga_led: exit  
Jan  1 07:51:06 linaro-nano kernel: vga_led: 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 left to transfer (0 on success)

A Very Simple Character Device

```
#include <linux/module.h>
#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;
}
```

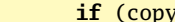
A Very Simple Character Device: Read

```
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



A Very Simple Character Device: Init

```
static struct file_operations schar_fops = {
    .owner          = THIS_MODULE,           Function
    .open           = schar_open,          pointer
    .release        = schar_release,       called
    .read           = schar_read,         by each
    .write          = schar_write,        operation
    .unlocked_ioctl = schar_ioctl };

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);        Request
                                           minor numbers 0-1
    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;
}
```

A Very Simple Character Device: Exit

```
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");
```

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!
# rmmmod 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_led */
    struct file_operations *fops;
};

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

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

The Platform Bus

Modern busses can discover their devices (lsusb, lspci, etc.); subsystems exist to deal with these
"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://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

The Raspberry Pi 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-cells = <1>;      address  address  size
        ranges = <0x7e000000 0x20000000 0x02000000>;

        uart@20201000 {
            compatible = "brcm,bcm2835-pl011",
                "arm,pl011", "arm,primecell";
            base
            address
            reg = <0x7e201000 0x1000>;
            interrupts = <2 25>;
            clock-frequency = <3000000>;
        };
    };
};
}
```

DTS for the VGA_LED

Connected through the “lightweight AXI bridge”
Avalon bus address 0 appears to the ARM at 0xff200000

```
lightweight_bridge: bridge@0xff200000 {
    compatible = "simple-bus";

    #address-cells = <1>;
    #size-cells = <1>;
    ranges = < 0x0 0xff200000 0x200000 >;

    vga_led: vga_led@0 {
        compatible = "altr,vga_led";
        reg = <0x0 0x8>;
    };
};
```

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_led_of_match[] = {
    { .compatible = "altr,vga_led" },
    {}
};
MODULE_DEVICE_TABLE(of, vga_led_of_match);

/* Platform device info */
static struct platform_driver vga_led_driver = {
    .driver = {
        .name      = "vga_led",
        .owner     = THIS_MODULE,
        .of_match_table = of_match_ptr(vga_led_of_match),
    },
    .remove = __exit_p(vga_led_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_led.ko
vga_led: init
# cat /proc/iomem
00000000-3fffffff : System RAM
    00008000-0052262f : Kernel code
    00552000-005bd72b : Kernel data
ff200000-ff200007 : vga_led
ff702000-ff703fff : /soc/ethernet@ff702000
ff704000-ff704fff : /soc/dwmmc0@ff704000
ff705000-ff705fff : ff705000.spi
ffa00000-ffa00fff : ff705000.spi
ffb40000-ffb4ffff : dwc_otg
ffc02000-ffc0201f : serial
ffc03000-ffc0301f : serial
ffc04000-ffc04fff : ffc04000.i2c
fff00000-fff00fff : fff00000.spi
fff01000-fff01fff : fff01000.spi
```

The VGA_LED Driver: Header File

```
#ifndef _VGA_LED_H
#define _VGA_LED_H

#include <linux/ioctl.h>

#define VGA_LED_DIGITS 8

typedef struct {
    unsigned char digit;    /* 0, 1, .. , VGA_LED_DIGITS - 1 */
    unsigned char segments; /* LSB: segment a; MSB: decimal point */
} vga_led_arg_t;

#define VGA_LED_MAGIC 'q'

/* ioctls and their arguments */
#define VGA_LED_WRITE_DIGIT _IOW(VGA_LED_MAGIC, 1, vga_led_arg_t *)
#define VGA_LED_READ_DIGIT  _IOWR(VGA_LED_MAGIC, 2, vga_led_arg_t *)

#endif
```

The VGA_LED Driver: write_digit

```
#include <linux/module.h>
#include <linux/init.h>
#include <linux/errno.h>
#include <linux/version.h>
#include <linux/platform_device.h>
#include <linux/miscdevice.h>
#include <linux/io.h>
#include <linux/of.h>
#include <linux/of_address.h>
#include <linux/fs.h>
#include <linux/uaccess.h>
#include "vga_led.h"
#define DRIVER_NAME "vga_led"

struct vga_led_dev {
    struct resource res;      /* Resource: our registers */
    void __iomem *virtbase; /* Pointer to registers */
    u8 segments[VGA_LED_DIGITS];
} dev;

static void write_digit(int digit, u8 segments)
{
    iowrite8(segments, dev.virtbase + digit);
    dev.segments[digit] = segments;
}
```

The VGA_LED Driver: ioctl

```
static long vga_led_ioctl(struct file *f, unsigned int cmd,
                          unsigned long arg)
{
    vga_led_arg_t vla;
    switch (cmd) {
    case VGA_LED_WRITE_DIGIT:
        if (copy_from_user(&vla, (vga_led_arg_t *) arg,
                          sizeof(vga_led_arg_t)))
            return -EACCES;
        if (vla.digit > 8)
            return -EINVAL;
        write_digit(vla.digit, vla.segments);
        break;
    case VGA_LED_READ_DIGIT:
        if (copy_from_user(&vla, (vga_led_arg_t *) arg,
                          sizeof(vga_led_arg_t)))
            return -EACCES;
        if (vla.digit > 8)
            return -EINVAL;
        vla.segments = dev.segments[vla.digit];
        if (copy_to_user((vga_led_arg_t *) arg, &vla,
                        sizeof(vga_led_arg_t)))
            return -EACCES;
        break;
    default: return -EINVAL;
    }
}
```

The VGA_LED Driver: file_operations

```
static const struct file_operations vga_led_fops = {
    .owner          = THIS_MODULE,
    .unlocked_ioctl = vga_led_ioctl,
};

static struct miscdevice vga_led_misc_device = {
    .minor          = MISC_DYNAMIC_MINOR,
    .name           = DRIVER_NAME,
    .fops           = &vga_led_fops,
};
```

The VGA_LED Driver: vga_led_probe

```
static int __init vga_led_probe(struct platform_device *pdev)
{
    static unsigned char welcome_message[VGA_LED_DIGITS] = {
        0x3E, 0x7D, 0x77, 0x08, 0x38, 0x79, 0x5E, 0x00};
    int i, ret;

    /* Register ourselves as a misc device: creates /dev/vga_led */
    ret = misc_register(&vga_led_misc_device);
    /* Find our registers in device tree; verify availability */
    ret = of_address_to_resource(pdev->dev.of_node, 0, &dev.res);
    if (ret) {
        ret = -ENOENT;
        goto out_deregister;
    }
    if (request_mem_region(dev.res.start, resource_size(&dev.res)
        DRIVER_NAME) == NULL) {
        ret = -EBUSY;
        goto out_deregister;
    }

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

The VGA_LED Driver: probe (cont) & remove

```
/* Display a welcome message */
for (i = 0; i < VGA_LED_DIGITS; i++)
    write_digit(i, welcome_message[i]);

return 0;

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

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

The VGA_LED Driver: init and exit

```
static const struct of_device_id vga_led_of_match[] = {
    { .compatible = "altr,vga_led" },
    {}
};
MODULE_DEVICE_TABLE(of, vga_led_of_match);

static struct platform_driver vga_led_driver = {
    .driver = {
        .name      = DRIVER_NAME,
        .owner     = THIS_MODULE,
        .of_match_table = of_match_ptr(vga_led_of_match),
    },
    .remove = __exit_p(vga_led_remove),
};

static int __init vga_led_init(void)
{
    pr_info(DRIVER_NAME " :_init\n");
    return platform_driver_probe(&vga_led_driver, vga_led_probe);
}

static void __exit vga_led_exit(void)
{
    platform_driver_unregister(&vga_led_driver);
    pr_info(DRIVER_NAME " :_exit\n");
}
```


The VGA_LED Driver

```
module_init(vga_led_init);  
module_exit(vga_led_exit);  
  
MODULE_LICENSE("GPL");  
MODULE_AUTHOR("Stephen_A._Edwards,_Columbia_University");  
MODULE_DESCRIPTION("VGA_7-segment_LED_Emulator");
```

References

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

[http://www.opersys.com/training/
linux-device-drivers](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.