Operating Systems And Applications For Embedded Systems

Device Drivers

Katedra Inżynierii Komputerowej

Plan

- Introducing Device Drivers
  - Main types of device driver
  - Character devices
  - Block devices
  - Network devices
  - GPIO
  - GPIO interrupt
- Wyniki
  - LEDs
  - I2C
  - SPI
  - Anatomy of a device driver
  - Loading kernel modules
  - Device trees
  - Platform data
  - Linking hardware with device drivers
  - Additional reading

Main types of device driver

- character: This is for unbuffered I/O with a rich range of functions and a thin layer between the application code and the driver. It is the first choice when implementing custom device drivers.
- block: This has an interface tailored for block I/O to and from mass storage devices. There is a thick layer of buffering designed to make disk reads and writes as fast as possible, which makes it unsuitable for anything else.
- network: This is similar to a block device but is used for transmitting and receiving network packets rather than disk blocks.

Character devices I

```
ls -l /dev/ttyAMA*
crw-rw—- 1 root root 204, 64 Jan 1 1970 /dev/ttyAMA0
crw-rw—- 1 root root 204, 65 Jan 1 1970 /dev/ttyAMA1
crw-rw—- 1 root root 204, 66 Jan 1 1970 /dev/ttyAMA2
crw-rw—- 1 root root 204, 67 Jan 1 1970 /dev/ttyAMA3
```

Listing 1: Listing

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
int main(void)
{
    int f;
```

Character devices II

```
    unsigned int rnd;
    int n;
    f = open("/dev/urandom", O_RDONLY);
    if (f < 0) {
        perror("Failed to open urandom");
        return 1;
    }
    n = read(f, &rnd, sizeof(rnd));
    if (n != sizeof(rnd)) {
        perror("Problem reading urandom");
        return 1;
    }
    printf("Random number = 0x%x\n", rnd);
    close(f);
    return 0;
```

Character devices III

```
}
```

Block devices

```
ls -l /dev/mmcblk*
brw——- 1 root root 179, 0 Jan 1 1970 /dev/mmcblk0
brw——- 1 root root 179, 1 Jan 1 1970 /dev/mmcblk0p1
brw——- 1 root root 179, 2 Jan 1 1970 /dev/mmcblk0p2
brw——- 1 root root 179, 8 Jan 1 1970 /dev/mmcblk1
brw——- 1 root root 179, 9 Jan 1 1970 /dev/mmcblk1p1
brw——- 1 root root 179, 10 Jan 1 1970 /dev/mmcblk1p2
```

Network devices I

```
my_netdev = alloc_netdev(0, ńetnetdev_setup);
ret = register_netdev(my_netdev);
```

Listing 2: Listing

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/ioctl.h>
#include <linux/sockios.h>
#include <net/if.h>
int main (int argc, char *argv[])
{
    int s;
    int ret;
```

Network devices II

```c
    struct ifreq ifr;
    int i;
    if (argc != 2) {
        printf("Usage %s [network interface]\n", argv[0]);
        return 1;
    }
    s = socket(PF_INET, SOCK_DGRAM, 0);
    if (s < 0) {
        perror("socket");
    return 1;
    }
    strcpy(ifr.ifr_name, argv[1]);
    ret = ioctl(s, SIOCGIFHWADDR, &ifr);
    if (ret < 0) {
        perror("ioctl");
```

Network devices III

```c
            return 1;
        }
        for (i = 0; i < 6; i++)
            printf("%02x:", (unsigned char)ifr.ifr_hwaddr.sa_data[i]);
        printf("\n");
        close(s);
        return 0;
    }
```

GPIO

There are details about the implementation of gpiolib in the kernel source in Documentation/gpio and the drivers themselves are in drivers/gpio. Applications can interact with gpiolib through files in the /sys/class/gpio directory. Here is an example of what you will see in there on a typical embedded board (a BeagleBone Black):

```
ls /sys/class/gpio
export gpiochip0 gpiochip32 gpiochip64 gpiochip96 unexport
```

The gpiochip0 to gpiochip96 directories represent four GPIO registers, each with 32 GPIO bits. If you look in one of the gpiochip directories, you will see the following:

```
ls /sys/class/gpio/gpiochip96/
base label ngpio power subsystem uevent
```

GPIO interrupt I

To enable interrupts, you can set it to one of these values:
rising: Interrupt on rising edge
falling: Interrupt on falling edge
both: Interrupt on both rising and falling edges
none: No interrupts (default)

Listing 3: Listing

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <poll.h>
 int main (int argc, char *argv[])
 {
```

GPIO interrupt II

```c
    int f;
    struct pollfd poll_fds [1];
    int ret;
    char value[4];
    int n;
    f = open("/sys/class/gpio/gpio48", O_RDONLY);
    if (f == -1) {
        perror("Can't open gpio48");
    return 1;
    }
    poll_fds[0].fd = f;
    poll_fds[0].events = POLLPRI | POLLERR;
    while (1) {
        printf("Waiting\n");
        ret = poll(poll_fds, 1, -1);
```

GPIO interrupt III

```c
        if (ret > 0) {
            n = read(f, &value, sizeof(value));
            printf("Button pressed: read %d bytes, value=%c\n",
            n, value[0]);
        }
    }
    return 0;
}
```

LEDs

LEDs are often controlled though a GPIO pin, but there is another kernel subsystem that offers more specialized control specifc to the purpose. The leds kernel subsystem adds the ability to set brightness, should the LED have that ability, and can handle LEDs connected in other ways than a simple GPIO pin. It can be confgured to trigger the LED on an event such as block device access or just a heartbeat to show that the device is working. There is more information in Documentation/leds/ and the drivers are in drivers/leds/.

As with GPIOs, LEDs are controlled through an interface in sysfs, in /sys/class/leds. The LEDs have names in the form devicename:colour:function, as shown here:

```
ls /sys/class/leds
beaglebone:green:heartbeat beaglebone:green:usr2
beaglebone:green:mmc0 beaglebone:green:usr3
```

This shows one individual LED:

```
ls /sys/class/leds/beaglebone:green:usr2
brightness max_brightness subsystem uevent
device power trigger
```

I2C I

Listing 4: Listing

```c
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <i2c-dev.h>
#include <sys/ioctl.h>
#define I2C_ADDRESS 0x5d
#define CHIP_REVISION_REG 0x10
void main (void)
{
    int f_i2c;
    int val;
    /* Open the adapter and set the address of the I2C device */
    f_i2c = open ("/dev/i2c-1", O_RDWR);
```

I2C II

```
    ioctl (f_i2c, I2C_SLAVE, I2C_ADDRESS);
    /* Read 16-bits of data from a register */
    val = i2c_smbus_read_word_data(f, CHIP_REVISION_REG);
    printf ("Sensor chip revision %d\n", val);
    close (f_i2c);
}
```

SPI I

The serial peripheral interface bus is similar to I2C, but is a lot faster, up to the low MHz. The interface uses four wires with separate send and receive lines which allows it to operate in full duplex. Each chip on the bus is selected with a dedicated chip select line. It is commonly used to connect to touchscreen sensors, display controllers, and serial NOR flash devices.

As with I2C, it is a master-slave protocol, with most SoCs implementing one or more master host controllers. There is a generic SPI device driver which you can enable through the kernel configuration CONFIG_SPI_SPIDEV. It creates a device node for each SPI controller which allows you to access SPI chips from user space. The device nodes are named spidev[bus].[chip select]:

```
ls -l /dev/spi*
crw-rw—– 1 root root 153, 0 Jan 1 00:29 /dev/spidev1.0
```

For examples of using the spidev interface, refer to the example code in Documentation/spi.

Anatomy of a device driver I

Listing 5: Listing

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/device.h>
#define DEVICE_NAME "dummy"
#define MAJOR_NUM 42
#define NUM_DEVICES 4
static struct class *dummy_class;
static int dummy_open(struct inode *inode, struct file *file)
{
    pr_info("%s\n", __func__);
    return 0;
```

Anatomy of a device driver II

```c
}
static int dummy_release(struct inode *inode, struct file *file)
{
    pr_info("%s\n", __func__);
    return 0;
}
static ssize_t dummy_read(struct file *file,
char *buffer, size_t length, loff_t * offset)
{
    pr_info("%s %u\n", __func__, length);
    return 0;
}
static ssize_t dummy_write(struct file *file,
const char *buffer, size_t length, loff_t * offset)
{
```

Anatomy of a device driver III

```
    pr_info("%s %u\n", __func__, length);
    return length;
}
struct file_operations dummy_fops = {
    .owner = THIS_MODULE,
    .open = dummy_open,
    .release = dummy_release,
    .read = dummy_read,
    .write = dummy_write,
};
int __init dummy_init(void)
{
    int ret;
    int i;
    printk("Dummy loaded\n");
```

Anatomy of a device driver IV

```c
    ret = register_chrdev(MAJOR_NUM, DEVICE_NAME, &dummy_fops);
    if (ret != 0)
        return ret;
    dummy_class = class_create(THIS_MODULE, DEVICE_NAME);
    for (i = 0; i < NUM_DEVICES; i++) {
        device_create(dummy_class, NULL,
        MKDEV(MAJOR_NUM, i), NULL, "dummy%d", i);
    }
    return 0;
}
void __exit dummy_exit(void)
{
    int i;
    for (i = 0; i < NUM_DEVICES; i++) {
        device_destroy(dummy_class, MKDEV(MAJOR_NUM, i));
```

Anatomy of a device driver V

```
        }
        class_destroy(dummy_class);
        unregister_chrdev(MAJOR_NUM, DEVICE_NAME);
        printk("Dummy unloaded\n");
    }
    module_init(dummy_init);
    module_exit(dummy_exit);
    MODULE_LICENSE("GPL");
    MODULE_AUTHOR("Chris Simmonds");
    MODULE_DESCRIPTION("A dummy driver");
```

Loading kernel modules I

```
insmod /lib/modules/4.1.10/kernel/drivers/dummy.ko
lsmod
dummy 1248 0 - Live 0xbf009000 (O)
rmmod dummy
```

Device trees I

```
net@10010000
compatible = śmsc,lan91c111";
reg = <0x10010000 0x10000>;
interrupts = <25>;
;
```

Platform data I

Listing 6: Listing

```
#define VERSATILE_ETH_BASE 0x10010000
#define IRQ_ETH 25
static struct resource smc91x_resources[] = {
[0] = {
.start = VERSATILE_ETH_BASE,
.end = VERSATILE_ETH_BASE + SZ_64K - 1,
.flags = IORESOURCE_MEM,
},
[1] = {
.start = IRQ_ETH,
.end = IRQ_ETH,
.flags = IORESOURCE_IRQ,
},
```

Platform data II

```c
};
static struct platform_device smc91x_device = {
.name = "smc91x",
.id = 0,
.num_resources = ARRAY_SIZE(smc91x_resources),
.resource = smc91x_resources,
};
```

Linking hardware with device drivers I

Listing 7: Listing

```
 static const struct of_device_id smc91x_match[] = {
{ .compatible = "smsc,lan91c94", },
{ .compatible = "smsc,lan91c111", },
{},
};
MODULE_DEVICE_TABLE(of, smc91x_match);
static struct platform_driver smc_driver = {
.probe = smc_drv_probe,
.remove = smc_drv_remove,
.driver = {
.name = "smc91x",
.of_match_table = of_match_ptr(smc91x_match),
},
```

Linking hardware with device drivers II

```
};
static int __init smc_driver_init(void)
{
return platform_driver_register(&smc_driver);
}
static void __exit smc_driver_exit(void) \
{
platform_driver_unregister(&smc_driver);
}
module_init(smc_driver_init);
module_exit(smc_driver_exit);

static int smc_drv_probe(struct platform_device *pdev)
{
struct smc91x_platdata *pd = dev_get_platdata(&pdev->dev);
```

Linking hardware with device drivers III

```
const struct of_device_id *match = NULL;
struct resource *res, *ires;
int irq;
res = platform_get_resource(pdev, IORESOURCE_MEM, 0);
ires = platform_get_resource(pdev, IORESOURCE_IRQ, 0);
[...]
addr = ioremap(res->start, SMC_IO_EXTENT);
irq = ires->start;
[...]
}
match = of_match_device(of_match_ptr(smc91x_match), &pdev->dev);
if (match) {
struct device_node *np = pdev->dev.of_node;
u32 val;
[...]
```

Linking hardware with device drivers IV

```
of_property_read_u32(np, "reg-io-width", &val);
[...]
}
```

Additional reading

- Linux Device Drivers, 4th edition, by Jessica McKellar, Alessandro Rubini, Jonathan Corbet, and Greg Kroah-Hartman. This is not published at the time of writing, but if it is as good as the predecessor, it will be a good choice. However, the 3rd edition is too out of date to recommend.
- Linux Kernel Development, 3rd edition by Robert Love, Addison-Wesley Professional; (July 2, 2010) ISBN-10: 0672329468
- Linux Weekly News, lwn.net.

References

C. Simmonds.
Mastering Embedded Linux Programming.
Packt Publishing, 2015.

The End