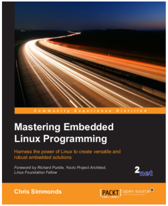
How to avoid writing device drivers for embedded Linux

License

About Chris Simmonds



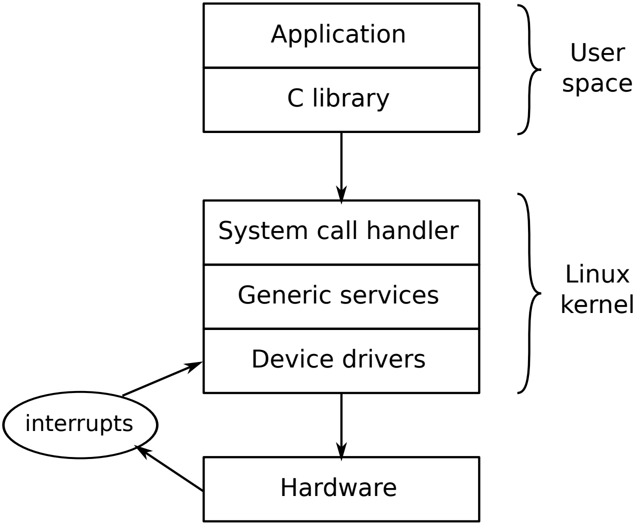
- Consultant and trainer
- Author of *Mastering Embedded Linux Programming*
- Working with embedded Linux since 1999
- Android since 2009
- Speaker at many conferences and workshops

"Looking after the Inner Penguin" blog at `http://2net.co.uk/`

`https://uk.linkedin.com/in/chrisdsimmonds/`

`https://google.com/+chrissimmonds`

Conventional device driver model



How applications call device drivers

- In Linux, everything is a file [1]
- Applications interact with drivers via POSIX functions open(2), read(2), write(2), ioctl(2), etc
- There are two types of interface
- 1. Device nodes in `/dev`
  - The serial driver, ttyS is an example
  - Device nodes are named `/dev/ttyS0`, `/dev/ttyS1` ...
- 2. Driver attributes, exported via *sysfs*
  - For example `/sys/class/gpio`

[1] Except network interfaces, which are sockets

Userspace drivers

- Writing kernel device drivers can be difficult
- Luckily, there are generic drivers that that allow you to write most of the code in userspace
- We will look at three
  - GPIO
  - PWM
  - I2C
- Note: applications will need read/write permissions for the files. Consequently, they usually have to run as user root

/sys/class/gpio

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

This device has 4 gpio chips
each with 32 pins

Write to this
file to export
a GPIO pin
to user space

Write to this
file to unexport
a GPIO pin
to user space

gpiochip

```
# /sys/class/gpio/gpiochip0
base  device   label  ngpio  power   subsystem  uevent
```

ngpio: The number of GPIO pins (32)

label: A lable to identify the chip (gpiochip0)

base: The starting GPIO number (0)

Exporting a GPIO pin

```
# echo 42 > /sys/class/gpio/export
# ls /sys/class/gpio
export   gpio42  gpiochip0   gpiochip32   gpiochip64   gpiochip96   unexport
```

If the export is successful, a new directory is created

Inputs and outputs

```
# ls /sys/class/gpio/gpio42
active_low  device  direction  edge  power  subsystem  uevent  value
```

Set to 1 to invert input and ouput

Set direction by writing "out" or "in". Default "in"

The logic level of the pin. Change the level of outputs by writing "0" or "1"

Interrupts

- If the GPIO can generate interrupts, the file `edge` can be used to control interrupt handling
- edge = ["none", "rising", "falling","both]
- For example, to make GPIO60 interrupt on falling edge:
  - `echo falling > /sys/class/gpio/gpio60/edge`
- To wait for an interrupt, use the poll(2) function
- Example on next slide

GPIO interrupt code example

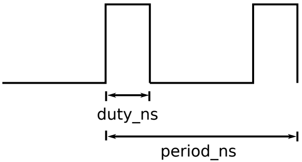
```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <poll.h>
int main (int argc, char *argv[])
{
    int f;
    struct pollfd poll_fds [1];
    int ret;
    char value[4];
    f = open("/sys/class/gpio/gpio60/value", O_RDONLY);
    poll_fds[0].fd = f;
    poll_fds[0].events = POLLPRI | POLLERR;
    while (1) {
        if (poll(poll_fds, 1, -1) > 0) {
            read(f, &value, sizeof(value));
            printf("Interrupt! value=%c\n", value[0]);
        }
    }
}
```


PWM

```
# echo 6 > /sys/class/pwm/export
# ls /sys/class/pwm
export  pwm6  pwmchip0  pwmchip2  pwmchip3  pwmchip5  pwmchip7  unexport
```

If the export is successful, a new directory is created

```
# ls /sys/class/pwm/pwm6/
device  duty_ns  period_ns  polarity  power  run  subsystem  uevent
```



I2C

- Device nodes, one per I2C bus controller:

```
# ls -l /dev/i2c*
crw-rw---T 1 root i2c 89, 0 Jan  1  2000 /dev/i2c-0
crw-rw---T 1 root i2c 89, 1 Jan  1  2000 /dev/i2c-1
```

- Some functions are implemented using ioctl(2), using commands and structures defined in
  `usr/include/linux/i2c-dev.h`

i2c-utils

- Command-line tools for interacting with I2C devices
- i2cdetect - list I2C adapters and probe bus
- i2cget - read data from an I2C device
- i2cset - write data to an I2C device

i2cdetect

- i2cdetect - list i2c adapters and probe bus
  - Example: detect devices on bus 1 (`/dev/i2c-1`)

```
# i2cdetect -y -r 1
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:          -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- 39 -- -- -- -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: -- -- -- -- UU UU UU UU -- -- -- -- -- -- -- --
60: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
70: -- -- -- -- -- -- -- --
```

UU = device already handled by kernel driver

0x39 = device discovered at address 0x39

i2cget/i2cset

- `i2cget <bus> <chip> <register>`: read data from an I2C device
    - Example: read register 0x8a from device at 0x39

```
# i2cget -y 1 0x39 0x8a
0x50
```

- `i2cset <bus> <chip> <register>`: writedata to an I2C device
    - Example: Write 0x03 to register 0x80:

```
# i2cset -y 1 0x39 0x80 3
```


I2C code example

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/i2c-dev.h>

int main(int argc, char **argv)
    int f;
    char buf[4];

    f = open("/dev/i2c-1", O_RDWR);
    ioctl(f, I2C_SLAVE, 0x39) < 0) {

    buf[0] = 0x8a;            /* Chip ID register */
    write(f, buf, 1);
    read(f, buf, 1);
    printf("ID 0x%x\n", buf [0]);
}
```


Other examples

- SPI: access SPI devices via device nodes `/dev/spidev*`
- USB: access USB devices via libusb
- User defined I/O: UIO
  - Generic kernel driver that allows you to write userspace drivers
  - access device registers and handle interrupts from userspace

