How to avoid writing kernel drivers

Chris Simmonds

Embedded Linux Conference Europe 2018
These slides are available under a Creative Commons Attribution-ShareAlike 3.0 license. You can read the full text of the license here http://creativecommons.org/licenses/by-sa/3.0/legalcode

You are free to

- copy, distribute, display, and perform the work
- make derivative works
- make commercial use of the work

Under the following conditions

- Attribution: you must give the original author credit
- Share Alike: if you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one (i.e. include this page exactly as it is)
- For any reuse or distribution, you must make clear to others the license terms of this work
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/](http://2net.co.uk/)

@2net_software

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

- Device drivers in kernel space
- Device drivers in user space
- Some examples:
  - GPIO
  - PWM
  - I2C
Conventional device driver model

- Application
- C library
- System call handler
- Generic services
- Device drivers
- Hardware

User space

Linux kernel

interrupts
How applications interact 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

• Two main types of interface:

1. Device nodes in /dev
   • For example, the serial driver, ttyS. 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 allow you to write most of the code in userspace
• We will look at three
  • GPIO
  • PWM
  • I2C
A note about device trees

• Even though you are writing userspace drivers, you still need to make sure that the hardware is accessible to the kernel.

• On ARM based systems, this may mean changing the device tree or adding a device tree overlay (which is outside the scope of this talk).
GPIO: General Purpose Input/Output

- Pins that can be configured as inputs or outputs
- As outputs:
  - used to control LEDs, relays, control chip selects, etc.
- As inputs:
  - used to read a switch or button state, etc.
  - some GPIO hardware can generate an interrupt when the input changes
Two userspace drivers!

• **gpiolib**: old, but scriptable interface using sysfs
• **gpio-cdev**: new, higher performance method using character device nodes /dev/gpiochip*

\(^1\) it’s not a library
The gpiolib sysfs interface

- GPIO pins grouped into registers, named `gpiochipNN`
- Each pin is assigned a number from 0 to XXX

```
# 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
Inside a gpiochip

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

The number of GPIO pins (32)

A label to identify the chip (gpiochip0)

The starting GPIO number (0)
Exporting a GPIO pin

```bash
# 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

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

- Set to 1 to invert input and output
- 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 a falling edge:
  
  • `echo falling > /sys/class/gpio/gpio60/edge`

• To wait for an interrupt, use the `poll(2)` function
The gpio-cdev interface

• One device node per GPIO register named /dev/gpiochip*
• Access the GPIO pins using ioctl(2)
• Advantages
  • Naming scheme gpiochip/pin rather than uniform but opaque name space from 0 to XXX
  • Multiple pin transitions in single function call without glitches
  • More robust handling of interrupts
/* 
* Demonstrate using gpio cdev to output a single bit 
* On a BeagleBone Black, GPIO1_21 is user LED 1 
*/

#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <linux/gpio.h>

int main(void)
{
    int f;
    int ret;
    struct gpiohandle_request req;
    struct gpiohandle_data data;
```c
f = open("/dev/gpiochip1", O_RDONLY);
req.lineoffsets[0] = 21;
req.flags = GPIOHANDLE_REQUEST_OUTPUT; /* Request as output */
req.default_values[0] = 0;
strncpy(req.consumer_label, "gpio-output"); /* up to 31 characters */
req.lines = 1;

ret = ioctl(f, GPIO_GET_LINEHANDLE_IOCTL, &req);

/* Note that there is a new file descriptor in req.fd to handle the
GPIO lines */
data.values[0] = 1;
ret = ioctl(req.fd, GPIOHANDLE_SET_LINE_VALUES_IOCTL, &data);
close(f);
return 0;
```
PWM: Pulse-Width Modulation

Most SoCs have dedicated circuits that can produce a wave with period and duty cycle.

Two main use cases:

- Dimmable LEDs and backlights
- Servo motors: deflection is proportional to duty cycle
The PWM sysfs interface

```
# ls /sys/class/pwm/pwmchip0
device  export  npwm  power  subsystem  uevent  unexport
```

Write to this file to export a PWM to user space

Write to this file to unexport a PWM to user space
Exporting a PWM

# echo 0 > /sys/class/pwm/pwmchip0/export
# ls /sys/class/pwm/pwmchip0
device  export  npwm  power  pwm0  subsystem  uevent  unexport

If the export is successful, a new directory is created

# ls /sys/class/pwm/pwmchip0/pwm0
capture  duty_cycle  export  period  power  uevent
device   enable    npwm    polarity  subsystem  unexport
PWM example

• For example, set period to 1 ms (1,000,000 ns) ...
• and duty to 0.5 ms (500,000 ns) ...
• then enable it

```bash
echo 1000000 > /sys/class/pwm/pwmchip0/pwm0/period
echo 500000 > /sys/class/pwm/pwmchip0/pwm0/duty_cycle
echo 1 > /sys/class/pwm/pwmchip0/pwm0/enable
```
I2C: the Inter-IC bus

- Simple 2-wire serial bus, commonly used to connect sensor devices
- Each I2C device has a 7-bit address, usually hard wired
- 16 bus addresses are reserved, giving a maximum of 112 nodes per bus
- The master controller manages read/write transfers with slave nodes
The i2c-dev driver

- **i2c-dev** exposes I2C master controllers
- Need to load/configure the i2c-dev driver (**CONFIG_I2C_CHARDEV**)
- There is one device node per i2c master 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
```

- You access I2C slave nodes using read(2), write(2) and ioctl(2)
- Structures defined in *usr/include/linux/i2c-dev.h*
**Detecting i2c slaves using i2cdetect**

- **i2cdetect**, from i2c-tools package, lists i2c adapters and probes devices.

  - Example: detect devices on bus 1 (/dev/i2c-1)

```bash
# 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 |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 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>`: write data to an I2C device
  - Example: Write 0x03 to register 0x80:
  
  ```
  # i2cset -y 1 0x39 0x80 3
  ```
```c
#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
What are you missing?

- User-space drivers are not always the best solution
  - User-space programs can be killed; kernel drivers cannot
  - Kernel drivers can use advanced locking techniques - spinlocks, rwlocks, rcu, etc
  - Kernel drivers have direct access to DMA channels and interrupts
  - A kernel driver can fit in to a subsystem
    - Example: controlling an LCD backlight is better done as a kernel PWM driver so that it can use the common backlight infrastructure
• Questions?