How to avoid writing device drivers for embedded Linux

Chris Simmonds
2net Limited

Winchester, UK
chris@2net.co.uk

***Abstract*—Modern Linux kernels provide interfaces that allow you to control hardware directly from an application, thus avoiding the need to write kernel device drivers. Beginning with the most basic interface of all, GPIO (General Purpose I/O), you can configure individual pins as inputs or outputs and then access them as special files. If the hardware is capable of generating interrupts on the GPIO inputs you can make use of that to write interrupt-driven functions.**

Likewise, the PWM (Pulse Width Modulation) interface allows you to make use of PWM hardware designed into most SoCs, allowing you to generate pulse trains to control lights, motors and more. Finally, when it comes to I2C devices, Linux provides a device node for each controller and a set of operations to read and write each slave device on the bus.

By using these interfaces you can control hardware and access a range of sensors from the safe and simple environment of your application, written in C, C++, Perl, Python or another language of your choice.

Keywords—Linux, User-space drivers, GPIO, PWM, i2c

I. Introduction

In Linux-based operating systems there is a distinction between device drivers and applications. Device drivers are part of the kernel and operate in at a high privilege level which allows them to access hardware registers, service interrupts and so on. They implement an interface that allows an application to call and interact with the device driver. A good example is the serial port driver. The device driver interfaces with a UART (Universal Asynchronous Receiver/Transmitter) and uses it to send and receive characters using RS-232 or a similar protocol. The driver implements an application-level interface in the form of a device node, which for a PC would have a name of the form `/dev/ttyS0`. An application can open this file and use the POSIX read(2) and write(2) functions to read characters from and send characters to the serial interface.

Following this model, each new piece of hardware requires a kernel device driver to control it and an application to make use of the basic I/O functions that the driver provides. Writing kernel code is complex and difficult to debug.

An alternative approach is to create general purpose device drivers that can handle a whole class of hardware and allow most of the logic required to control the hardware to be implemented in the application. These are often referred to as *user-space device drivers*.

There are several good guides to these interfaces available, including Mastering Embedded Linux Programming [1]. This paper focuses on three such interfaces: GPIO, PWM and I2C. They are generally easier to write and so allow for rapid prototyping of new hardware.

II. GPIO

Most embedded SoCs have a number of GPIO (General Purpose I/O) pins that can be used to control digital interfaces. Most SoC designs include several registers that control GPIO pins, usually in groups of 32. In addition, there are I/O extender chips, such as the MAX7313 from Maxim or the MPC23017 from Microchip. These particular devices are attached via the I2C bus, but that is a detail that is hidden in the interface described here. In most cases, GPIO pins can be configured as inputs or outputs or and in the former case, may be able to generate an interrupt when the input state changes.

GPIOs can be used to control digital outputs like LEDs and relays, and be be used to read digital inputs from push buttons, keypads and similar devices. It is also possible to use a group of GPIO pins to implement a more complex interface, such as a serial interface, a process that it known as *bit-banging*. The kernel driver that allows access to GPIO from applications is enabled by building the kernel with `CONFIG_GPIO_SYSFS`: almost all embedded platforms are build with this turned on.

The GPIO pins available from the registers and extender chips are numbered from 0 to N. Each register or chip is assigned a base GPIO number in that range. The allocations are visible through directories in `/sys/class/gpio`. This is a typical example:

```
# ls /sys/class/gpio
```

```
export gpiochip0 gpiochip32 gpiochip64
gpiochip96 unexport
```

In this case there are four chips with base GPIO numbers 0, 32, 64 and 96, providing a total of 128 GPIO pins Within each directory are these files:

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

There are three files that are important here:

- **base** – the base GPIO number, which is also reflected in the name of the directory
- **label** – a name for the register or chip
- **ngpio** – the number of GPIO pins in this register or chip

Initially all GPIO pins are controlled by the kernel. To gain access to a GPIO from user-space, it is necessary to write the number of the GPIO to the file `/sys/class/goio/export`. If the export succeeds, meaning that the pin is not being used by a kernel driver, then a new directory is created which contains files necessary to interact with it. For example, to export GPIO pin 48 you would:

```
# echo 48 > /sys/class/gpio/export
# ls /sys/class/gpio
export gpio48 gpiochip0 gpiochip32 gpiochip64
gpiochip96 unexport
```

A new directory named **gpio48** has been created. Within that are these files:

```
# ls /sys/class/gpio/gpio48
active_low direction edge power subsystem
uevent value
```

Reading the file **`direction`** returns the string “in” or “out”, indicating whether it is an input or an output. Initially all GPIOs are inputs. To change the direction, write “out” or “in” to **`direction`**.

The file **value** represents the level of the pin, which can be “0” or “1”. For inputs, reading this file returns the level of the input; for outputs writing to this file to sets the level of the output. The file **active_low** changes the polarity of the pin so that a high level represents 0 and a low level represents 1.

If the GPIO can generate an interrupt when the input changes state, the file **edge** will be present. It can contain the following strings:

- **none** - no interrupts generated (the default)
- **rising** - Interrupt on rising edge
- **falling** - Interrupt on falling edge
- **both** - Interrupt on both edges

If interrupts are enabled an application can wait for a state change on the input using the POSIX poll(2) function to wait for a `POLLPRI` or `POLLERR` event. Here is a sample program, taken from [1]:

```
#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];
    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);
        if (ret > 0) {
            n = read(f, &value,
                     sizeof(value));
            printf("Button pressed\n");
        }
    }

    return 0;
}
```

Since the interface is implemented entirely using file reads writes, it can be used by any language that provides and API to access the file system.

III. PWM

PWM (Pulse-Width Modulation) is a mechanism used to drive a circuit at different levels using a digital output by rapidly pulsing it on and off. By varying the “on” time compared to the “off” time a range of drive levels can be achieved. PWM is often used to control the brightness of LCD backlights, LEDs and other light sources, to control the speed of small DC motors and other similar devices. It is possible to generate a PWM signal using a GPIO pin and software to turn it on and off, but this is very inefficient in CPU cycles and consequent power demand. Many SoCs implement PWM interfaces that can perform the switching entirely in hardware, and there are PWM interfaces on some I/O extender chips.

The design of the PWM application interface is similar to the GPIO driver [2]. Each PWM interface is exposed through subdirectories in `/sys/class/pwm`. For each PWM controller there is a directory named `pwmchipN`, where N is the numeric identifier of the interface, usually starting from 0. For example:

```
# ls /sys/class/pwm/

export pwmchip0 pwmchip2 pwmchip3 pwmchip5
pwmchip7 unexport
```

Ini this instance there are five PWM interfaces. In addition, there are files named **`export`** and **`unexport`**. As with GPIO, you write the interface number to which you want to gain access to `export`, and when you no longer need the interface you write the interface number to unexport to return it to the kernel. For example, to obtain access to interface pwm3, you would do this;

```
# echo 3 > /sys/class/pwm/export
# ls /sys/class/pwm/
export pwm3 pwmchip0 pwmchip2 pwmchip3
pwmchip5 pwmchip7 unexport
```

Note that directory pwm3 has been created. Within that are these files:

```
# ls /sys/class/pwm/pwm3/
device duty_ns period_ns polarity power run
subsystem uevent
```

The files that control the PWM are as follows:

- **duty_cycle_ns** - The active time of the PWM signal in nanoseconds. It must be less than the period.
- **period_ns** - The total period of the PWM signal in nanoseconds. This is the sum of the active and inactive time of the PWM.
- **polarity** – Sets the polarity of the PWM signal. Writes to this property only work if the PWM chip supports changing the polarity. The polarity can only be changed if the PWM is not enabled. It may be "normal" or "inversed".
- **run** – Enable the PWM by writing "1" and disable by writing "0"

As an example, assume that you want to program the PWM interface to a frequency of 1000Hz, and a duty cycle of 25%. The period is 1 millisecond, which is 1000000 nanoseconds, and the duty cycle is 0.25 milliseconds, or 250000 nanoseconds, so this sequence of commands would be necessary:

```
# echo 1000000 > \
/sys/class/pwm/pwm3/period_ns
# echo 250000 > \ /sys/class/pwm/pwm3/duty_ns
# echo 1 > /sys/class/pwm/pwm3/run
```

As with GPIO, the interface consists entirely of file reads and writes and so is easy to implement in most languages.

IV. I2C

I2C stands for Inter-Integrated Circuit [3]. It is a simple low speed 2-wire bus that is common on embedded boards, typically used to access peripherals which are not on the SoC itself. Example usage includes display controllers, camera sensors, GPIO extenders, and temperature sensors. There is a related standard known as SMBus (System Management Bus) that is found on PCs. SMBus is a subset of I2C.

I2C is a master-slave protocol, with the master being a host controller on the SoC. Slaves have a 7-bit address assigned by the manufacturer, which you can discover by reading the data sheet. This allows for up to 128 nodes per bus, but 16 are reserved, so only 112 nodes are allowed in practice. The bus speed is 100 KHz in standard mode, or up to 400 KHz in fast mode. The protocol allows read and write transactions between the master and slave of up to 32 bytes. Frequently, the first byte is used to specify a register on the peripheral and the remaining bytes are the data read from or written to that register.

There is a kernel driver that allows you to access i2c devices from user-space. You have to build the kernel with option `CONFIG_I2C_CHARDEV` enabled. If it is built as a kernel module, you will have to load module i2c-dev.ko. That will create a device node for each i2c host adapter. For example:

```
   # ls -l /dev/i2c*

   crw-rw---- 1 root i2c 89, 0 Jan  1 00:18
/dev/i2c-0

   crw-rw---- 1 root i2c 89, 1 Jan  1 00:18
/dev/i2c-1

   crw-rw---- 1 root i2c 89, 2 Jan  1 00:18
/dev/i2c-2

   crw-rw---- 1 root i2c 89, 3 Jan  1 00:18
/dev/i2c-3
```

You can perform some simple actions using the command-line tools in the package i2ctools. The tools are:

- **i2cdetect** - lists i2c adapters and probe bus
- **i2cdump** - dump data from all registers of an I2C peripheral (warning: dangerous!!)
- **i2cget** - read data from an I2C device
- **i2cget** <bus> <chip> <register>
- **i2cset** - write data to an I2C device
- **i2cset** <bus> <chip> <register> <value>

The programming interface for I2C is based on the POSIX ioctl(2) function. Unfortunately, ioctl represents input and output parameters as structures and passes a pointer to the structure. This makes it difficult to write code to access the I2C peripherals in any language other than C or C++. Here is an example program, once again taken from [1]:

```
#include <i2c-dev.h>
#include <sys/ioctl.h>

#define I2C_ADDRESS 0x5d
#define CHIP_REVISION_REG 0x10

main ()
{
       int f_i2c;
       int val;

       /* Open the adapter and set the address
          of the I2C device */
       f_i2c = open ("/dev/i2c-1", O_RDWR);
       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);
}
```

To access I2C and SMBus from Python you can use smbus-cffi [4].

V. Conclusion

This paper has presented three general purpose Linux kernel drivers that allow a considerable portion of the interface logic to be implementing in user-space, thus avoiding the need to write complex kernel device drivers for a wide range of real-world cases. In addition to these interfaces, the interested reader could also explore accessing USB peripherals using libusb [5].

I will finish by mentioning that there is a hybrid approach using a Linux sub-system called User I/O, or UIO. This requires a simple device driver to be written, but retains most of the logic in user space. Using it, you can write more complex drivers that can handle interrupts and even DMA transfers. The details are too complex to describe here. It is described in the Linux kernel documentation [6].

References

[1] Mastering Embedded Linux Programming by Chris Simmonds, Packt Publishing, ISBN-13: 978-1784392536

[2] https://www.kernel.org/doc/Documentation/pwm.txt

[3] https://en.wikipedia.org/wiki/I%C2%B2C

[4] https://pypi.python.org/pypi/smbus-cffi/0.4.1

[5] http://www.libusb.org/

[6] https://www.kernel.org/doc/htmldocs/uio-howto/about.html