UIO_driver_and_app

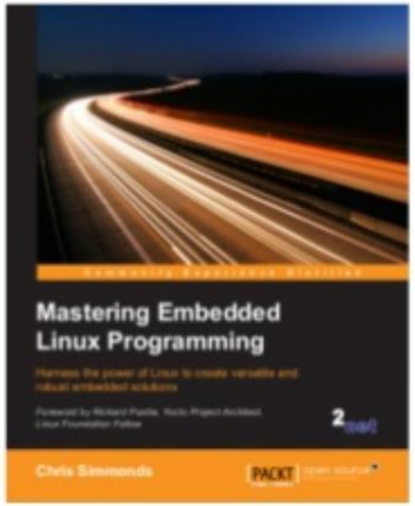
Quick and Easy Device Drivers for Embedded Linux Using UIO

Chris Simmonds

Embedded World 2017

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/`

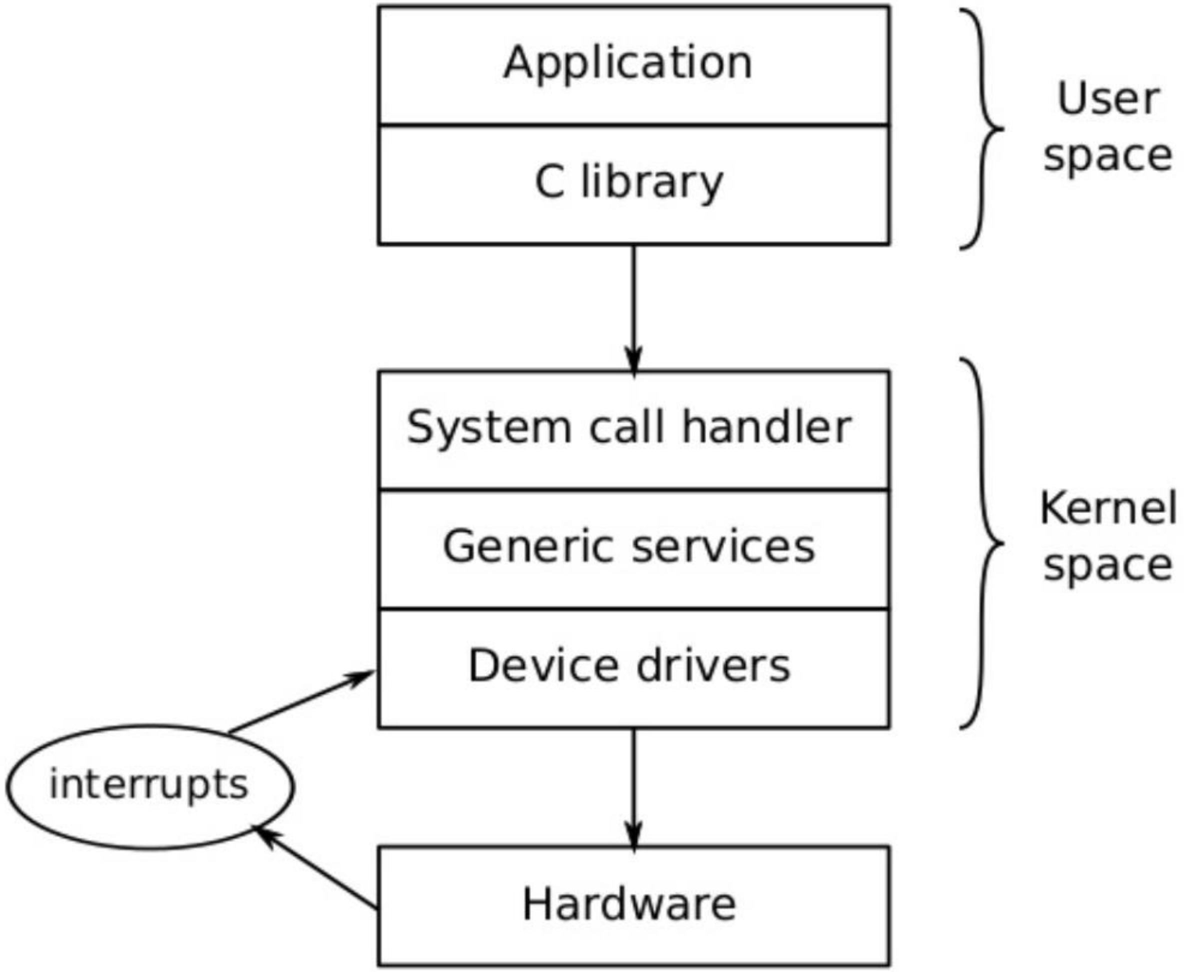
`https://uk.linkedin.com/in/chrisdsimmonds/`

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

Overview

- Conventional Linux drivers
- The UIO framework
- An example UIO driver
- Scheduling and interrupt latencies

Conventional device driver model



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
- For example
  - USB (via libusb)
  - GPIO
  - I2C

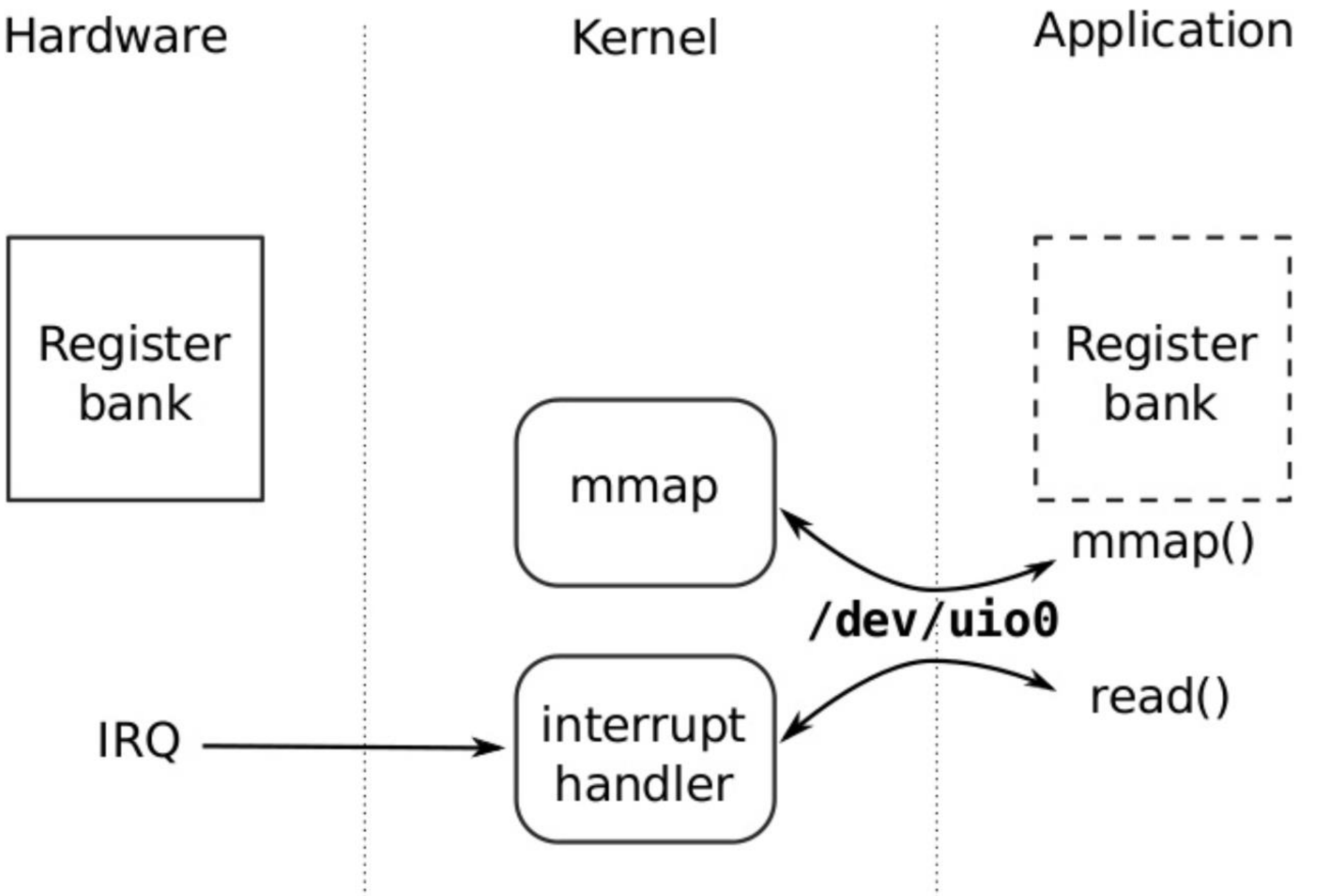
Reference

www.slideshare.net/chrissimmonds/userspace-drivers2016

UIO drivers

- **Userspace I/O (UIO)** is a framework for userspace drivers that do not fit into the standard patterns
- Typical use-cases include interfaces to FPGAs and custom PCI functions
- UIO may be appropriate for your hardware interface if:
  - it has registers and/or buffers that are memory mapped
  - it generates interrupts

The UIO way


Hardware
Kernel
Application
Register bank
Register bank
mmap
mmap()
/dev/uio0
read()
IRQ
interrupt handler

Kernel and userspace components

- UIO drivers are in two parts
  - A simple kernel stub driver, which creates device node /dev/uioX
  - A user-space driver that implements the majority of the code
- Device node /dev/uioX links the two together

The UIO kernel driver

- Kernel driver needs to
  - point to one (or more) memory regions
  - assign the interrupt number (IRQ)
  - implement interrupt handler
  - register as a UIO driver

Example driver

This device has 8 KiB (0x2000) of memory-mapped registers at 0x4804C000 and is attached to hardware interrupt IRQ 85

```
static int demo_probe(struct platform_device *pdev)
{
    info.name = "demo";
    info.version = "1.0";
    info.mem[0].addr = 0x4804C000;
    info.mem[0].size = 0x2000;
    info.mem[0].memtype = UIO_MEM_PHYS;
    info.irq = 85;
    info.irq_flags = 0;
    info.handler = demo_handler;

    return uio_register_device(&pdev->dev, &info);
}
```


Kernel interrupt handler

- Usually very simple
- For example, disable interrupt source, which will be enabled again in userspace

drivers/uio/uio_aec.c:

```
static irqreturn_t aectc_irq(int irq, struct uio_info *dev_info)
{
        void __iomem *int_flag = dev_info->priv + INTA_DRVR_ADDR;
        unsigned char status = ioread8(int_flag);

        if ((status & INTA_ENABLED_FLAG) && (status & INTA_FLAG)) {
                /* application writes 0x00 to 0x2F to get next interrupt */
                status = ioread8(dev_info->priv + MAILBOX);
                return IRQ_HANDLED;
        }
        return IRQ_NONE;
}
```


The user-space driver

- Each UIO driver represented by device node `/dev/uioX`
- X = 0 for first, 1 for second, etc.
- User space application uses it to mmap the memory and receive notification of interrupts

Mapping memory

- mmap(2) maps memory associated with a file descriptor
- Example: map 0x2000 bytes from file /dev/uio0:

```
int main(int argc, char **argv)
{
    int f;
    char *ptr;

    f = open("/dev/uio0", O_RDWR);
    if (f == -1) {
        return 1;
    }
    ptr = mmap(0, 0x2000, PROT_READ | PROT_WRITE, MAP_SHARED, f, 0);
    if (ptr == MAP_FAILED) {
        return 1;
    }
```

`ptr` points to the base of the register bank. UIO framework maps the memory without processor cache, so writes and reads force memory cycles on the system bus

Handling interrupts

- Wait for interrupt by reading from `/dev/uioX`
- The read() blocks until the next interrupt arrives
- Data returned by read contains the count of interrupts since the UIO kernel driver was started
    - Can detect missed interrupts by comparing with previous interrupt count

Interrupt example 1

- Simple example, using blocking read call

```
static void wait_for_int(int f)
{
    int n;
    unsigned int irc_count;

    printf("Waiting\n");
    n = read(f, &irc_count, sizeof(irc_count));
    if (n == -1) {
        printf("read error\n");
        return;
    }
    printf("irc_count = %d\n", irc_count);
}
```


Blocking behaviour

- Blocking in read() means that the application cannot do any work between interrupts
- Solution 1: use poll() or select() to wait with a timeout
  - possibly on several data sources at once
- Solution 2: use a thread to wait

Interrupt example 2

- Using poll(2)

```
static void wait_for_int_poll(int f)
{
    struct pollfd poll_fds [1];
    int ret;
    unsigned int irc_count;

    printf("Waiting\n");
    poll_fds[0].fd = f;
    poll_fds[0].events = POLLIN;

    ret = poll(poll_fds, 1, 100); /* timeout = 100 ms */
    if (ret > 0) {
        if (poll_fds[0].revents && POLLIN) {
            read(f, &irc_count, sizeof(irc_count));
            printf("irc_count = %d\n", irc_count);
        }
    }
}
```

Mapping more than one memory area

- Example: a device with two address ranges:

```
info.name = "demo";
info.version = "1.0";
info.mem[0].addr = 0x4804C000;
info.mem[0].size = 0x2000;
info.mem[0].memtype = UIO_MEM_PHYS;
info.mem[1].addr = 0x48060000;
info.mem[1].size = 0x4000;
info.mem[1].memtype = UIO_MEM_PHYS;
return uio_register_device(&pdev->dev, &info);
```

UIO allows up to 5 address ranges

Mapping address ranges

- The range to map is specified as mmap offset (last parameter)
- Because behind the scenes mmap works in pages instead of bytes, the index has to be given in pages
- To mmap address range in `info.mem[1]`, use offset = 1 page:

```
ptr2 = mmap(0, 0x4000, PROT_READ | PROT_WRITE,
            MAP_SHARED, f, 1 * getpagesize());
```

Scheduling

- Where interrupts are concerned, the process or thread should be real-time
  - scheduling policy = `SCHED_FIFO`
  - Memory locked using `mlockall`

```
struct sched_param param;

mlockall (MCL_CURRENT | MCL_FUTURE);
param.sched_priority = 10;

if (sched_setscheduler (getpid (), SCHED_FIFO, &param) != 0)
        printf ("Failed to change policy and priority\n");
```


Interrupt latency

- Even a real-time thread may have high jitter on interrupt latency if the kernel is not preemptive
- Configure kernel with `CONFIG_PREEMPT`
- Or, implement Real-time kernel patche and configure with `CONFIG_PREEMPT_RT`
- Measures latencies will vary from platform to platform, but should be less than a few hundred microseconds

Further reading

- Kernel source documentation: `Documentation/DocBook/uio-howto`
  - on-line at https://www.osadl.org/fileadmin/dam/interface/docbook/howtos/uio-howto.pdf
- LWN article: https://lwn.net/Articles/232575/

Summary

- UIO provides a convenient way to implement drivers for FPGA interfaces and hardware for which there is no existing Linux driver
- Applicable to hardware that provides mappable memory and generates interrupts

- Any questions?

This and other topics associated with building robust embedded systems are covered in my training courses `http://www.2net.co.uk/training.html`