Userspace I/O drivers in a realtime context

Hans J. Koch
Linutronix GmbH
Bahnhofstr. 25, 88690 Uhldingen, Germany
hjk@linutronix.de

Abstract

Userspace I/O (UIO) drivers and realtime kernels often meet each other, since both are frequently used on embedded devices. Being designed to handle strange devices like FPGAs found on embedded boards, UIO is a simple and convenient way to implement a driver for such devices. However, the author frequently receives enquiries from people who are uncertain about the performance of a driver partly written in userspace, especially in a realtime system. This article shows how UIO works, and how it performs under realtime conditions.

1 Introduction

The Userspace I/O framework (UIO) was introduced in Linux 2.6.23 and allows device drivers to be written almost entirely in userspace. UIO is suitable for hardware that does not fit into other kernel subsystems, like fieldbus cards, industrial I/O cards, or A/D converters. Programmers in industry who work with such hardware are rarely kernel experts. Writing and maintaining a large inkernel driver for special hardware of this kind can be a nightmare.

UIO addresses this situation by allowing the programmer to write most of the driver in userspace using all standard application programming tools and libraries. This greatly simplifies development, maintenance, and distribution of device drivers for this kind of hardware.

2 Linux device drivers

Hardware devices can be grouped by function (network devices, block devices) or by the way they are connected to the processor (PCI devices, USB devices). These different groups (or classes) of hardware are supported by different kernel subsystems. For example, if somebody wants to develop a driver for a new PCI network card, he will make use of the already existing PCI and networking subsystems. He will only implement those functions that are really hardware specific. He will neither implement the many thousand lines of code that are needed for every PCI card nor the many thousand lines needed for every networking device.

The programmer in this example will also notice that there are already many drivers for other PCI networking cards in the kernel source tree. He can use them as examples or templates for his own driver.

As a result, drivers for standard hardware are relatively small and easy to implement. They can be easily included in the mainline kernel source tree, which simplifies future maintenance of the driver code.

Unfortunately, there is a lot of hardware out there that does not fit into one of these subsystems. Analog or digital I/O, fieldbus interfaces, or custom specific FPGAs are examples for such hardware.

3 A new driver model? Why?

Drivers for such non-standard hardware are often implemented as character devices. For very simple devices, the read() and write() methods of the device file are all that is needed to control the device from userspace.

But typically this sort of hardware tends to be more complex. Driver authors usually implement all the extra functionality using ioctl().

Figure 1 shows the architecture of such a conventional driver.

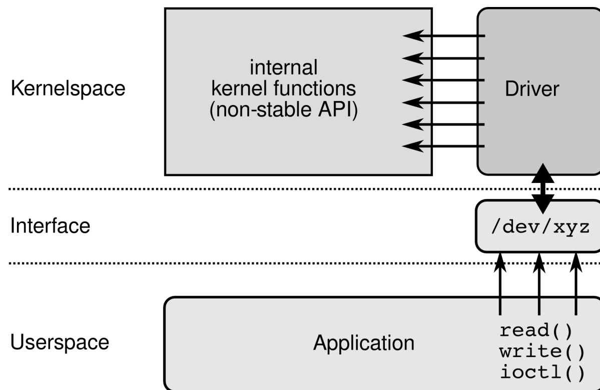

FIGURE 1: *Conventional driver*

Note that this sort of driver will have to use many internal kernel functions and macros. For several good reasons, the kernel developers refuse to keep the *internal* API stable. This means that a driver that works fine with today's kernel might neither work nor compile anymore in three months. Drivers included in the mainline kernel do not share this problem: A programmer who applies changes to the internal API is of course obliged to fix all affected drivers, so the community will actually maintain the driver. But the special driver for non-standard hardware will probably never make it into the mainline kernel, so the original author is fully responsible for keeping his driver up-to-date.

Summing it all up, a programmer who needs to develop a driver for such a device used to be in this situation: He must write a complete inkernel driver from scratch. The driver will become large because there is no support by existing subsystems. There are no examples in the kernel sources. He will face a steep learning curve since he has to deal with many internal kernel functions and macros. Debugging the driver will be very different from debugging an application. The driver will not go to mainline, so he will have considerable work maintaining it throughout the whole lifetime of the product.

To address this situation, the Userspace I/O framework was introduced.

4 How does UIO work?

A device driver has basically two tasks:

1. accessing device memory
2. dealing with interrupts generated by the device

The first task is easy to accomplish. Linux is able to map physical device memory to an address accessible from userspace. This has already been possible by using /dev/mem, and a lot of people used it for similar purposes, introducing some security leaks or stability problems. UIO improves this by preventing userspace from mapping memory that does not belong to the device. The UIO core already has an mmap() implementation capable of doing this for all kinds of memory (physical, logical, and virtual memory). A UIO driver author does not have to deal with the non-trivial internals of Linux kernel memory management in any way.

The second task is more difficult. Interrupts need to be serviced within the kernel because it is not possible to do that in userspace. With today's level-triggered interrupts, the machine will hang if the interrupt is still active at the end of the interrupt service routine (ISR). Therefore, we still need a small kernel module containing a minimal ISR that only needs to acknowledge or disable the interrupt. All the other work is done in userspace.

If the userspace part of the driver wants to wait for an interrupt, it simply does a blocking read() from /dev/uioX. This call will return immediately as soon as an interrupt occurs. UIO also implements the poll() system call, so you can (and should) use select() to wait for an interrupt. The select() function has a timeout parameter that can be used to prevent the task from hanging if there are no more interrupts.

Figure 2 shows a very small kernel driver that calls only a few kernel functions. Most of the functionality is handled in a generic way by the UIO framework, effectively protecting the driver author from the dirty sides of the kernel. As a bonus, the UIO framework generates a set of directories and attribute files in sysfs. These sysfs attributes allow searching for devices, finding the correct size of the memory mappings, version of the kernel driver, and so on.

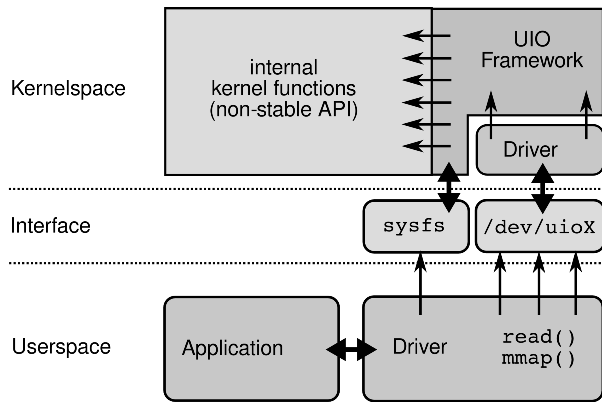

FIGURE 2: *UIO driver*

5 Example kernel driver

Handling interrupts is the main reason why a kernel module is necessary. However, the ISR can be very small. All it needs to do is ensure that the interrupt line becomes inactive again. It is also good style to handle shared interrupts properly, even if the current hardware does not make use of it. A typical ISR for a UIO driver will usually look like this (pseudo code):

```
static irqreturn_t my_handler(int irq,
                  struct uio_info *dev_info)
{
    if (IRQ is not caused by my hardware)
        return IRQ_NONE;

    /* Disable interrupt */
    (Perform some register access
     to silence the IRQ line)

    return IRQ_HANDLED;
}
```

Next, we need to implement the driver's probe function. Here we fill a structure of type struct uio_info with information about the device. There is a name and a version string that should be set to a value that is meaningful for the userspace part of the driver. The UIO core will report an error if one of these strings is a NULL pointer, so we might as well fill in something useful.

We also need to fill in the device resources. How exactly we get this information depends on what kind of device we have. For a PCI card, we will call pci_resource_start() and pci_resource_len() to get information about the card's memory resources. A PCI card is physical memory, so we set the memtype field to UIO_MEM_PHYS. The IRQ was already detected by the PCI subsystem and can be found in struct pci_dev-¿irq. We also pass in the flags we want the UIO core to use for request_irq().

Having set up struct uio_info, we then call uio_register_device(). If this function returns any non-zero value, something went wrong and we let the whole probe function fail, returning -ENODEV. Note that we can also perform our own tests here to find out if the hardware is really in a usable state and let probe() fail if it is not. Actually, we should at least check the more obvious errors, e.g. the presence of the physical memory.

Putting it all together, a probe function for a PCI card could look like this:

```
static int __devinit my_pci_probe(struct pci_dev *dev,
                        const struct pci_device_id *id)
{
    struct uio_info *info;

    info = kzalloc(sizeof(struct uio_info),
                   GFP_KERNEL);
    if (!info)
        return -ENOMEM;

    if (pci_enable_device(dev))
        goto out_free;
    if (pci_request_regions(dev, "my_card"))
        goto out_disable;

    info->name = "my_card";
    info->version = "0.0.1";
    info->mem[0].addr = pci_resource_start(dev, 0);
    if (!info->mem[0].addr)
        goto out_release;
    info->mem[0].internal_addr =
        ioremap(pci_resource_start(dev, 0),
                pci_resource_len(dev, 0));
    if (!info->mem[0].internal_addr)
        goto out_release;
    info->mem[0].size = pci_resource_len(dev, 0);
    info->mem[0].memtype = UIO_MEM_PHYS;

    info->irq = dev->irq;
    info->irq_flags = IRQF_SHARED;
    info->handler = my_handler;

    if (uio_register_device(&dev->dev, info))
        goto out_unmap;
    pci_set_drvdata(dev, info);
    return 0;
out_unmap:
    iounmap(info->mem[0].internal_addr);
```

```
out_release:
    pci_release_regions(dev);
out_disable:
    pci_disable_device(dev);
out_free:
    kfree (info);
    return -ENODEV;
}
```

If the probe function is successful, we remember the pointer to the uio_info structure by calling pci_set_drvdata(). This is necessary because we need that pointer at module unload time to unregister our UIO device.

You can find working examples of UIO drivers for PCI cards in drivers/uio/, e.g. uio_cif.c, in the kernel source tree.

6 Example userspace driver

Let's have a look at a minimal UIO userspace driver. The following code simply opens a UIO device, maps its memory, and waits for interrupts. I omitted all error checking, and the code does not do anything with the mapped memory:

```
int32_t irq_count;
int fd = open("/dev/uio0", O_RDWR);
void *map_addr = mmap(NULL, 4096,
                  PROT_READ | PROT_WRITE,
                  MAP_SHARED, fd, 0);

/* Always read exactly 4 bytes! */
while (read(fd, &irq_count, 4) == 4) {
    printf(Interrupt number %d\n,
           irq_count);
}
```

For a real driver, this code has to be improved, of course. The values returned by open() and mmap() need to be checked for errors, and we should call munmap() before we leave the program. The name of the device file (/dev/uio0) and the size of the memory mapping (4096) should never be hardcoded. The values we set up in the kernel module in the last chapter are available in sysfs and should be used here. Instead of blindly assuming that our device is /dev/uio0, we should check if the string in /sys/class/uio/uio0/name is really the name we set in our kernel part. If it is not, we try uio1, uio2, and so on. In the mmap() call, we should use the value found in /sys/class/uio/uio0/maps/map0/size instead of hardcoding the size value.

Note that UIO implements mmap() in an unusual way. The last parameter ("offset", 0 in the example above) is used to tell UIO which mapping we want to use. It has to be set to N*getpagesize() for mapping number N. An offset cannot be given, UIO always maps from the beginning of the memory. This strange behaviour of mmap() was necessary to allow devices to have more than one address range that can be mapped. If mmap() was successful, map_addr is a pointer to the beginning of the device's memory.

The read() call in our example needs some explanation. It is important to read exactly four bytes. The internal counter in the UIO core is a signed 32-bit integer *on all architectures*, so you should use a suitable type (like int32_t in the example) for your counter variable. The read() call will block, until an interrupt occurs, there is no way around this. You cannot specify O_NONBLOCK in your open() call.

In most applications, it is not desirable to have a read() call that blocks forever if no interrupt occurs. You can use select() to find out if a subsequent read() will block. It is also possible to pass a timeout value to select(), making it return if nothing happens within the given time. Read the manpage of select() for details. Another advantage of select() is that it can wait for multiple file descriptors. Applications often not only wait for interrupts but also e.g. for data to arrive from a socket. Using select() can eliminate the need for several threads in many cases.

One last detail: The open() call in our example opens the UIO device file with O_RDWR. We do not want to write to the UIO device file, so why did I use O_RDWR? The answer is simple: If we opened the UIO device with O_RDONLY, the subsequent mmap() with PROT_WRITE set would fail. As we want to have read *and* write access to our device memory, we have to use O_RDWR.

6.1 libuio

Userspace parts of UIO drivers generally contain code segments that are independent of the hardware, e.g. finding the UIO device by name, opening it, mapping memory using the information given in sysfs, and so on.

We created a library called libuio that contains these frequently needed functions. It is licensed under the terms of the LGPL. A link to the git repository is given at the end of this document [1].

7 Performance considerations

There are two important things to look at: How fast can we access the card's memory (registers)? How long does it take until the userspace part of the driver can respond to an interrupt?

The following two chapters deal with these aspects.

7.1 Performance of memory access

In real-world drivers, ioctl() is often used to write a single value to a hardware register. As can be seen in figure 3, this is not as straight-forward as one might think. In that one syscall, the Virtual File System (VFS) first needs to find the ioctl implementation for that device and call it. Then the ioctl function will copy the value from userspace to kernelspace. And finally, the value is written to the card. If the call is supposed to return a result, that same path (reversed) is walked again.

In a UIO driver, the device's memory is directly mapped into userspace. Writing to a register is as simple as an access to a normal array of integers. This makes UIO userspace driver code faster and easier to read.

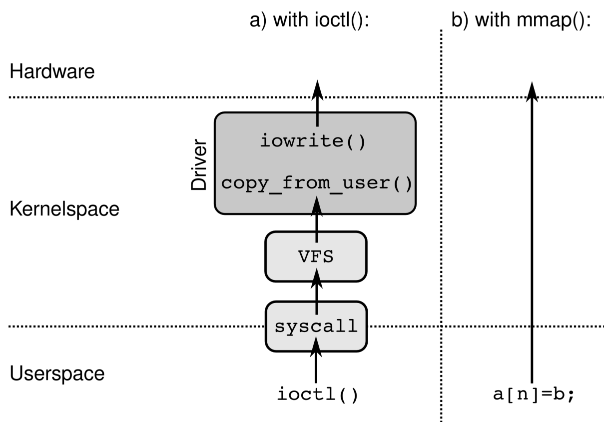

FIGURE 3: *ioctl() vs. direct memory access*

7.2 Interrupt latency

The interrupt latency of a UIO driver is obviously slightly longer than the one of an in-kernel driver, because the userspace process needs to be woken up. But how large is that latency overhead, and how can it be measured?

Most UIO drivers depend on a certain hardware. That makes it difficult to compare different boards or architectures. To work around that problem, I use a driver called uio_dummy for testing. I wrote this driver a few years ago to test the UIO core. The kernel part of uio_dummy uses a timer that regularly calls uio_event_notify(). This function is also called when real hardware interrupts occur in a normal driver. That means that uio_dummy can very well be used for testing interrupt behaviour without the need for real hardware.

The userspace part of the uio_dummy driver can be very small, too. All it needs to do is call select() to wait for an interrupt. After select() returns, it has to call read() to allow the next interrupt to trigger the UIO core handler.

The time when uio_event_notify() is called is the time when an in-kernel driver could respond to an interrupt. The time when select() returns is the time when the userspace part of the UIO driver can respond to the interrupt. Consequently, the difference between these times is the UIO latency overhead we are looking for.

Both times can be measured using a kernel with tracing enabled. Make sure you have CONFIG_FUNCTION_TRACER and CONFIG_DYNAMIC_FTRACE enabled.

With such a kernel, and a userspace program called uio_events, you can do something like this:

```
#!/bin/sh

if ! lsmod | grep uio_dummy
then
    echo Loading uio_dummy...
    modprobe uio_dummy
fi

./uio_events 2>&1 > /dev/null &
PID=‘pidof -s uio_events‘

echo uio_events started, pid = $PID

if ! mount | grep debugfs
then
    echo Mounting debugfs on /sys/kernel/debug/...
    mount -t debugfs nodev /sys/kernel/debug/
fi

cd /sys/kernel/debug/tracing
echo uio_event_notify > set_ftrace_filter
echo function > current_tracer
echo global > trace_clock
echo 1 > events/syscalls/sys_exit_select/enable
cd events/syscalls/sys_exit_select/
echo "common_pid==$PID" > filter
```

```
cd /sys/kernel/debug/tracing
cat trace_pipe
```

This gave me the following output on a laptop with Core i3 processor, running 2.6.33.7-rt29, uio_dummy produced one interrupt per second:

```
...
sirq-timer/0-5     [000]   377.437654: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   377.437680: sys_select -> 0x1
sirq-timer/0-5     [000]   378.437251: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   378.437278: sys_select -> 0x1
sirq-timer/0-5     [000]   379.440866: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   379.440893: sys_select -> 0x1
sirq-timer/0-5     [000]   380.438907: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   380.438923: sys_select -> 0x1
sirq-timer/0-5     [000]   381.436042: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   381.436069: sys_select -> 0x1
sirq-timer/0-5     [000]   382.435631: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   382.435654: sys_select -> 0x1
sirq-timer/0-5     [000]   383.439290: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   383.439309: sys_select -> 0x1
sirq-timer/0-5     [000]   384.438831: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   384.438855: sys_select -> 0x1
sirq-timer/0-5     [000]   385.438425: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   385.438452: sys_select -> 0x1
sirq-timer/0-5     [000]   386.442043: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   386.442067: sys_select -> 0x1
sirq-timer/0-5     [000]   387.437709: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   387.437741: sys_select -> 0x1
sirq-timer/0-5     [000]   388.437237: uio_event_notify <-uio_dummy_timer
  uio_events-1971  [000]   388.437258: sys_select -> 0x1
...
```

The differences are 26, 27, 27, 16, 27, 23, 19, 24, 27, 24, 32, and 21 microseconds between the call to uio_event_notify and the return of userspace's select() call.

For comparison, cyclictest on the same computer delivered the following results:

```
# cyclictest -n -p80
...
... Min: 5 Act: 11 Avg: 52 Max: 134
```

This is just a short example to show that UIO latency overhead does not stand out when compared to other latencies in the system. More detailed measurement results will be made available, a link can be found at the end of this article [2].

7.3 Experiences with real systems

The author developed several UIO drivers for customer's embedded products. One example is a board equipped with an ARM11 CPU (Freescale i.MX31) where a custom specific chip caused 1000 interrupts/sec. These were easily handled by the UIO driver, even at 90% CPU load.

UIO drivers for fieldbus cards that are in mainline, like uio_cif.c and uio_sercos3.c can produce a similar interrupt load and are in commercial production use for quite a while.

It should be obvious that in real world systems, with high CPU loads and many RT tasks, optimizing the system, e.g. by tuning thread priorities, can consume a considerable amount of development time.

8 Conclusion

UIO is the easiest way to write a driver for devices that are not supported by other subsystems. It allows programmers to write the major part of the driver in userspace, using all the tools, programming languages, and libraries they are aquainted with.

The kernel part of the driver is very small and easy to write and to maintain. After passing the re-

view process, it is usually no problem to include it into the mainline kernel.

The interrupt latency overhead of UIO usually causes no problems and is often negligible. Compared to in-kernel character device drivers, it is often more than compensated by the fast register access through mapped memory.

References

[1] *http://git.linutronix.de/gitweb.cgi?p=projects/libUIO.git;a=summary*

[2] *http://hansjkoch.de/~hjk/uio*