

The Userspace I/O HOWTO

Hans-Jürgen Koch

Linutronix

<hjk@linutronix.de>

Copyright © 2006-2008 Hans-Jürgen Koch.

This documentation is Free Software licensed under the terms of the GPL version 2.

2006-12-11

Revision History		
Revision 0.5	2008-05-22	hjk
Added description of write() function.		
Revision 0.4	2007-11-26	hjk
Removed section about uio_dummy.		
Revision 0.3	2007-04-29	hjk
Added section about userspace drivers.		
Revision 0.2	2007-02-13	hjk
Update after multiple mappings were added.		
Revision 0.1	2006-12-11	hjk
First draft.		

Abstract

This HOWTO describes concept and usage of Linux kernel's Userspace I/O system.

Table of Contents

1. About this document
 1. Translations
 2. Preface
 3. Acknowledgments
 4. Feedback
2. About UIO

1. How UIO works
 3. Writing your own kernel module
 1. struct uio_info
 2. Adding an interrupt handler
 4. Writing a driver in userspace
 1. Getting information about your UIO device
 2. mmap() device memory
 3. Waiting for interrupts
- A. Further information

Chapter 1. About this document

Table of Contents

1. Translations
2. Preface
3. Acknowledgments
4. Feedback

1. Translations

If you know of any translations for this document, or you are interested in translating it, please email me <hjk@linutronix.de>.

2. Preface

For many types of devices, creating a Linux kernel driver is overkill. All that is really needed is some way to handle an interrupt and provide access to the memory space of the device. The logic of controlling the device does not necessarily have to be within the kernel, as the device does not need to take advantage of any of other resources that the kernel provides. One such common class of devices that are like this are for industrial I/O cards.

To address this situation, the userspace I/O system (UIO) was designed. For typical industrial I/O cards, only a very small kernel module is needed. The main part of the driver will run in user space. This simplifies development and reduces the risk of serious bugs within a kernel module.

Please note that UIO is not an universal driver interface. Devices that are already handled well by other kernel subsystems (like networking or serial or USB) are no candidates for an UIO driver. Hardware that is ideally suited for an UIO driver fulfills all of the following:

- The device has memory that can be mapped. The device can be controlled completely by writing to this memory.
- The device usually generates interrupts.

- The device does not fit into one of the standard kernel subsystems.

3.Acknowledgments

I'd like to thank Thomas Gleixner and Benedikt Spranger of Linutronix, who have not only written most of the UIO code, but also helped greatly writing this HOWTO by giving me all kinds of background information.

4.Feedback

Find something wrong with this document? (Or perhaps something right?) I would love to hear from you. Please email me at <hjk@linutronix.de>.

Chapter2.About UIO

Table of Contents

1. How UIO works

If you use UIO for your card's driver, here's what you get:

- only one small kernel module to write and maintain.
- develop the main part of your driver in user space, with all the tools and libraries you're used to.
- bugs in your driver won't crash the kernel.
- updates of your driver can take place without recompiling the kernel.

1.How UIO works

Each UIO device is accessed through a device file and several sysfs attribute files. The device file will be called `/dev/uio0` for the first device, and `/dev/uio1`, `/dev/uio2` and so on for subsequent devices.

`/dev/uioX` is used to access the address space of the card. Just use `mmap()` to access registers or RAM locations of your card.

Interrupts are handled by reading from `/dev/uioX`. A blocking `read()` from `/dev/uioX` will return as soon as an interrupt occurs. You can also use `select()` on `/dev/uioX` to wait for an interrupt. The integer value read from `/dev/uioX` represents the total interrupt count. You can use this number to figure out if you missed some interrupts.

For some hardware that has more than one interrupt source internally, but not separate IRQ mask and status registers, there might be situations where userspace cannot determine what the interrupt source was if the kernel handler disables them by writing to the chip's IRQ register. In such a case, the kernel has to

disable the IRQ completely to leave the chip's register untouched. Now the userspace part can determine the cause of the interrupt, but it cannot re-enable interrupts. Another corner case is chips where re-enabling interrupts is a read-modify-write operation to a combined IRQ status/acknowledge register. This would be racy if a new interrupt occurred simultaneously.

To address these problems, UIO also implements a `write()` function. It is normally not used and can be ignored for hardware that has only a single interrupt source or has separate IRQ mask and status registers. If you need it, however, a write to `/dev/uioX` will call the `irqcontrol()` function implemented by the driver. You have to write a 32-bit value that is usually either 0 or 1 to disable or enable interrupts. If a driver does not implement `irqcontrol()`, `write()` will return with `-ENOSYS`.

To handle interrupts properly, your custom kernel module can provide its own interrupt handler. It will automatically be called by the built-in handler.

For cards that don't generate interrupts but need to be polled, there is the possibility to set up a timer that triggers the interrupt handler at configurable time intervals. This interrupt simulation is done by calling `uio_event_notify()` from the timer's event handler.

Each driver provides attributes that are used to read or write variables. These attributes are accessible through sysfs files. A custom kernel driver module can add its own attributes to the device owned by the uio driver, but not added to the UIO device itself at this time. This might change in the future if it would be found to be useful.

The following standard attributes are provided by the UIO framework:

- `name`: The name of your device. It is recommended to use the name of your kernel module for this.
- `version`: A version string defined by your driver. This allows the user space part of your driver to deal with different versions of the kernel module.
- `event`: The total number of interrupts handled by the driver since the last time the device node was read.

These attributes appear under the `/sys/class/uio/uioX` directory. Please note that this directory might be a symlink, and not a real directory. Any userspace code that accesses it must be able to handle this.

Each UIO device can make one or more memory regions available for memory mapping. This is necessary because some industrial I/O cards require access to more than one PCI memory region in a driver.

Each mapping has its own directory in sysfs, the first mapping appears as `/sys/class/uio/uioX/maps/map0/`. Subsequent mappings create directories `map1/`, `map2/`, and so on. These directories will only appear if the size of the mapping is not 0.

Each `mapX/` directory contains two read-only files that show start address and size of the memory:

- `addr`: The address of memory that can be mapped.
- `size`: The size, in bytes, of the memory pointed to by `addr`.

From userspace, the different mappings are distinguished by adjusting the `offset` parameter of the `mmap()` call. To map the memory of mapping `N`, you have to use `N` times the page size as your offset:

```
offset = N * getpagesize();
```

Chapter 3. Writing your own kernel module

Table of Contents

1. `struct uio_info`
2. Adding an interrupt handler

Please have a look at `uio_cif.c` as an example. The following paragraphs explain the different sections of this file.

1. `struct uio_info`

This structure tells the framework the details of your driver; Some of the members are required, others are optional.

- `char *name`: Required. The name of your driver as it will appear in `sysfs`. I recommend using the name of your module for this.
- `char *version`: Required. This string appears in `/sys/class/uio/uioX/version`.
- `struct uio_mem mem[MAX_UIO_MAPS]`: Required if you have memory that can be mapped with `mmap()`. For each mapping you need to fill one of the `uio_mem` structures. See the description below for details.
- `long irq`: Required. If your hardware generates an interrupt, it's your module's task to determine the `irq` number during initialization. If you don't have a hardware generated interrupt but want to trigger the interrupt handler in some other way, set `irq` to `UIO_IRQ_CUSTOM`. If you had no interrupt at all, you could set `irq` to `UIO_IRQ_NONE`, though this rarely makes sense.
- `unsigned long irq_flags`: Required if you've set `irq` to a hardware interrupt number. The flags given here will be used in the call to `request_irq()`.
- `int (*mmap)(struct uio_info *info, struct vm_area_struct *vma)`: Optional. If you need a special `mmap()` function, you can set it here. If this pointer is not `NULL`, your `mmap()` will be called instead of the built-in one.
- `int (*open)(struct uio_info *info, struct inode *inode)`: Optional. You might want to have your own `open()`, e.g. to enable interrupts only when your device is actually used.

- `int (*release)(struct uio_info *info, struct inode *inode)`: Optional. If you define your own `open()`, you will probably also want a custom `release()` function.
- `int (*irqcontrol)(struct uio_info *info, s32 irq_on)`: Optional. If you need to be able to enable or disable interrupts from userspace by writing to `/dev/uioX`, you can implement this function. The parameter `irq_on` will be 0 to disable interrupts and 1 to enable them.

Usually, your device will have one or more memory regions that can be mapped to user space. For each region, you have to set up a `struct uio_mem` in the `mem[]` array. Here's a description of the fields of `struct uio_mem`:

- `int memtype`: Required if the mapping is used. Set this to `UIO_MEM_PHYS` if you have physical memory on your card to be mapped. Use `UIO_MEM_LOGICAL` for logical memory (e.g. allocated with `kmalloc()`). There's also `UIO_MEM_VIRTUAL` for virtual memory.
- `unsigned long addr`: Required if the mapping is used. Fill in the address of your memory block. This address is the one that appears in `sysfs`.
- `unsigned long size`: Fill in the size of the memory block that `addr` points to. If `size` is zero, the mapping is considered unused. Note that you *must* initialize `size` with zero for all unused mappings.
- `void *internal_addr`: If you have to access this memory region from within your kernel module, you will want to map it internally by using something like `ioremap()`. Addresses returned by this function cannot be mapped to user space, so you must not store it in `addr`. Use `internal_addr` instead to remember such an address.

Please do not touch the `kobj` element of `struct uio_mem`! It is used by the UIO framework to set up `sysfs` files for this mapping. Simply leave it alone.

2. Adding an interrupt handler

What you need to do in your interrupt handler depends on your hardware and on how you want to handle it. You should try to keep the amount of code in your kernel interrupt handler low. If your hardware requires no action that you *have* to perform after each interrupt, then your handler can be empty.

If, on the other hand, your hardware *needs* some action to be performed after each interrupt, then you *must* do it in your kernel module. Note that you cannot rely on the userspace part of your driver. Your userspace program can terminate at any time, possibly leaving your hardware in a state where proper interrupt handling is still required.

There might also be applications where you want to read data from your hardware at each interrupt and buffer it in a piece of kernel memory you've allocated for that purpose. With this technique you could avoid loss of data if your userspace program misses an interrupt.

A note on shared interrupts: Your driver should support interrupt sharing whenever this is possible. It is possible if and only if your driver can detect whether your hardware has triggered the interrupt or not. This is usually done by looking at an interrupt status register. If your driver sees that the IRQ bit is actually set, it will perform its actions, and the handler returns `IRQ_HANDLED`. If the driver detects that it was not your hardware that caused the interrupt, it will do nothing and return `IRQ_NONE`, allowing the kernel to call the next possible interrupt handler.

If you decide not to support shared interrupts, your card won't work in computers with no free interrupts. As this frequently happens on the PC platform, you can save yourself a lot of trouble by supporting interrupt sharing.

Chapter 4. Writing a driver in userspace

Table of Contents

1. Getting information about your UIO device
2. `mmap()` device memory
3. Waiting for interrupts

Once you have a working kernel module for your hardware, you can write the userspace part of your driver. You don't need any special libraries, your driver can be written in any reasonable language, you can use floating point numbers and so on. In short, you can use all the tools and libraries you'd normally use for writing a userspace application.

1. Getting information about your UIO device

Information about all UIO devices is available in `sysfs`. The first thing you should do in your driver is check `name` and `version` to make sure you're talking to the right device and that its kernel driver has the version you expect.

You should also make sure that the memory mapping you need exists and has the size you expect.

There is a tool called `lsuio` that lists UIO devices and their attributes. It is available here:

<https://www.osadl.org/projects/downloads/UIO/user/>

With `lsuio` you can quickly check if your kernel module is loaded and which attributes it exports. Have a look at the manpage for details.

The source code of `lsuio` can serve as an example for getting information about an UIO device. The file `uio_helper.c` contains a lot of functions you could use in your userspace driver code.

2.mmap() device memory

After you made sure you've got the right device with the memory mappings you need, all you have to do is to call `mmap()` to map the device's memory to userspace.

The parameter `offset` of the `mmap()` call has a special meaning for UIO devices: It is used to select which mapping of your device you want to map. To map the memory of mapping `N`, you have to use `N` times the page size as your offset:

```
offset = N * getpagesize();
```

`N` starts from zero, so if you've got only one memory range to map, set `offset = 0`. A drawback of this technique is that memory is always mapped beginning with its start address.

3.Waiting for interrupts

After you successfully mapped your devices memory, you can access it like an ordinary array. Usually, you will perform some initialization. After that, your hardware starts working and will generate an interrupt as soon as it's finished, has some data available, or needs your attention because an error occurred.

`/dev/uioX` is a read-only file. A `read()` will always block until an interrupt occurs. There is only one legal value for the `count` parameter of `read()`, and that is the size of a signed 32 bit integer (4). Any other value for `count` causes `read()` to fail. The signed 32 bit integer read is the interrupt count of your device. If the value is one more than the value you read the last time, everything is OK. If the difference is greater than one, you missed interrupts.

You can also use `select()` on `/dev/uioX`.

AppendixA.Further information

- OSADL homepage.
- Linutronix homepage.