

The Userspace I/O HOWTO

Hans-Jürgen Koch

The Userspace I/O HOWTO

Hans-Jürgen Koch

Publication date 2006-12-11

Copyright © 2006-2008 Hans-Jürgen Koch.

Copyright © 2009 Red Hat Inc, Michael S. Tsirkin (mst@redhat.com)

Abstract

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

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

Table of Contents

1. About this document	1
Translations	1
Preface	1
Acknowledgments	1
Feedback	1
2. About UIO	2
How UIO works	2
3. Writing your own kernel module	5
struct uio_info	5
Adding an interrupt handler	6
Using uio_pdrv for platform devices	7
Using uio_pdrv_genirq for platform devices	7
Using uio_dmem_genirq for platform devices	7
4. Writing a driver in userspace	9
Getting information about your UIO device	9
mmap() device memory	9
Waiting for interrupts	9
5. Generic PCI UIO driver	11
Making the driver recognize the device	11
Things to know about uio_pci_generic	11
Writing userspace driver using uio_pci_generic	12
Example code using uio_pci_generic	12
A. Further information	14

Chapter 1. About this document

Translations

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

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.

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.

Feedback

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

- `size`: The number of ports in this region.
- `porttype`: A string describing the type of port.

Chapter 3. Writing your own kernel module

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

struct uio_info

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

- `const char *name`: Required. The name of your driver as it will appear in sysfs. I recommend using the name of your module for this.
- `const 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.
- `struct uio_port port[MAX_UIO_PORTS_REGIONS]`: Required if you want to pass information about iports to userspace. For each port region you need to fill one of the `uio_port` structures. See the description below for details.
- `long irq`: Required. If your hardware generates an interrupt, it's your modules 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`:

- `const char *name`: Optional. Set this to help identify the memory region, it will show up in the corresponding sysfs node.

Set the `.name` element of `struct platform_device` to `"uio_dmem_genirq"` to use this driver.

When using this driver, fill in the `.platform_data` element of `struct platform_device`, which is of type `struct uio_dmem_genirq_pdata` and which contains the following elements:

- `struct uio_info uiainfo`: The same structure used as the `uio_pdrv_genirq` platform data
- `unsigned int *dynamic_region_sizes`: Pointer to list of sizes of dynamic memory regions to be mapped into user space.
- `unsigned int num_dynamic_regions`: Number of elements in `dynamic_region_sizes` array.

The dynamic regions defined in the platform data will be appended to the `mem[]` array after the platform device resources, which implies that the total number of static and dynamic memory regions cannot exceed `MAX_UIO_MAPS`.

The dynamic memory regions will be allocated when the UIO device file, `/dev/uioX` is opened. Similar to static memory resources, the memory region information for dynamic regions is then visible via sysfs at `/sys/class/uio/uioX/maps/mapY/*`. The dynamic memory regions will be freed when the UIO device file is closed. When no processes are holding the device file open, the address returned to userspace is `~0`.

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`.

Chapter 5. Generic PCI UIO driver

The generic driver is a kernel module named `uio_pci_generic`. It can work with any device compliant to PCI 2.3 (circa 2002) and any compliant PCI Express device. Using this, you only need to write the userspace driver, removing the need to write a hardware-specific kernel module.

Making the driver recognize the device

Since the driver does not declare any device ids, it will not get loaded automatically and will not automatically bind to any devices, you must load it and allocate id to the driver yourself. For example:

```
modprobe uio_pci_generic
echo "8086 10f5" > /sys/bus/pci/drivers/uio_pci_generic/new_id
```

If there already is a hardware specific kernel driver for your device, the generic driver still won't bind to it, in this case if you want to use the generic driver (why would you?) you'll have to manually unbind the hardware specific driver and bind the generic driver, like this:

```
echo -n 0000:00:19.0 > /sys/bus/pci/drivers/e1000e/unbind
echo -n 0000:00:19.0 > /sys/bus/pci/drivers/uio_pci_generic/bind
```

You can verify that the device has been bound to the driver by looking for it in `sysfs`, for example like the following:

```
ls -l /sys/bus/pci/devices/0000:00:19.0/driver
```

Which if successful should print

```
.../0000:00:19.0/driver -> ../../../../bus/pci/drivers/uio_pci_generic
```

Note that the generic driver will not bind to old PCI 2.2 devices. If binding the device failed, run the following command:

```
dmesg
```

and look in the output for failure reasons

Things to know about `uio_pci_generic`

Interrupts are handled using the Interrupt Disable bit in the PCI command register and Interrupt Status bit in the PCI status register. All devices compliant to PCI 2.3 (circa 2002) and all compliant PCI Express devices should support these bits. `uio_pci_generic` detects this support, and won't bind to devices which do not support the Interrupt Disable Bit in the command register.

On each interrupt, `uio_pci_generic` sets the Interrupt Disable bit. This prevents the device from generating further interrupts until the bit is cleared. The userspace driver should clear this bit before blocking and waiting for more interrupts.

Writing userspace driver using `uio_pci_generic`

Userspace driver can use `pci sysfs` interface, or the `libpci` library that wraps it, to talk to the device and to re-enable interrupts by writing to the command register.

Example code using `uio_pci_generic`

Here is some sample userspace driver code using `uio_pci_generic`:

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>

int main()
{
    int uiofd;
    int configfd;
    int err;
    int i;
    unsigned icount;
    unsigned char command_high;

    uiofd = open("/dev/uio0", O_RDONLY);
    if (uiofd < 0) {
        perror("uio open:");
        return errno;
    }
    configfd = open("/sys/class/uio/uio0/device/config", O_RDWR);
    if (configfd < 0) {
        perror("config open:");
        return errno;
    }

    /* Read and cache command value */
    err = pread(configfd, &command_high, 1, 5);
    if (err != 1) {
        perror("command config read:");
        return errno;
    }
    command_high &= ~0x4;

    for(i = 0;; ++i) {
        /* Print out a message, for debugging. */
        if (i == 0)
```

```
    fprintf(stderr, "Started uio test driver.\n");
else
    fprintf(stderr, "Interrupts: %d\n", icount);

/*****
/* Here we got an interrupt from the
   device. Do something to it. */
*****/

/* Re-enable interrupts. */
err = pwrite(configfd, &command_high, 1, 5);
if (err != 1) {
    perror("config write:");
    break;
}

/* Wait for next interrupt. */
err = read(uiofd, &icount, 4);
if (err != 4) {
    perror("uio read:");
    break;
}
}
return errno;
}
```

Appendix A. Further information

- OSADL homepage. [<http://www.osadl.org>]
- Linutronix homepage. [<http://www.linutronix.de>]