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@hansjkoch.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@hansjkoch.de>.

Chapter 2. About UIO

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.

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 cornercase 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 four read-only files that show attributes of the memory:

- `name`: A string identifier for this mapping. This is optional, the string can be empty. Drivers can set this to make it easier for userspace to find the correct mapping.

- `addr`: The address of memory that can be mapped.

- `size`: The size, in bytes, of the memory pointed to by addr.

- `offset`: The offset, in bytes, that has to be added to the pointer returned by `mmap()` to get to the actual device memory. This is important if the device's memory is not page aligned. Remember that pointers returned by `mmap()` are always page aligned, so it is good style to always add this offset.

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();
```

Sometimes there is hardware with memory-like regions that can not be mapped with the technique described here, but there are still ways to access them from userspace. The most common example are x86 ioports. On x86 systems, userspace can access these ioports using `ioperm()`, `iopl()`, `inb()`, `outb()`, and similar functions.

Since these ioport regions can not be mapped, they will not appear under `/sys/class/uio/uioX/maps/` like the normal memory described above. Without information about the port regions a hardware has to offer, it becomes difficult for the userspace part of the driver to find out which ports belong to which UIO device.

To address this situation, the new directory `/sys/class/uio/uioX/portio/` was added. It only exists if the driver wants to pass information about one or more port regions to userspace. If that is the case, subdirectories named `port0`, `port1`, and so on, will appear underneath `/sys/class/uio/uioX/portio/`.

Each `portX/` directory contains four read-only files that show name, start, size, and type of the port region:

- `name`: A string identifier for this port region. The string is optional and can be empty. Drivers can set it to make it easier for userspace to find a certain port region.

- `start`: The first port of this region.

- `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 ioports 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.

- `int memtype`: Required if the mapping is used. Set this to `UIO_MEM_PHYS` if you 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.
- `phys_addr_t addr`: Required if the mapping is used. Fill in the address of your memory block. This address is the one that appears in sysfs.
- `resource_size_t 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 `map` element of `struct uio_mem`! It is used by the UIO framework to set up sysfs files for this mapping. Simply leave it alone.

Sometimes, your device can have one or more port regions which can not be mapped to userspace. But if there are other possibilities for userspace to access these ports, it makes sense to make information about the ports available in sysfs. For each region, you have to set up a `struct uio_port` in the `port[]` array. Here's a description of the fields of `struct uio_port`:

- `char *porttype`: Required. Set this to one of the predefined constants. Use `UIO_PORT_X86` for the ioports found in x86 architectures.
- `unsigned long start`: Required if the port region is used. Fill in the number of the first port of this region.
- `unsigned long size`: Fill in the number of ports in this region. If `size` is zero, the region is considered unused. Note that you *must* initialize `size` with zero for all unused regions.

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

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.

Using uio_pdrv for platform devices

In many cases, UIO drivers for platform devices can be handled in a generic way. In the same place where you define your `struct platform_device`, you simply also implement your interrupt handler and fill your `struct uio_info`. A pointer to this `struct uio_info` is then used as `platform_data` for your platform device.

You also need to set up an array of `struct resource` containing addresses and sizes of your memory mappings. This information is passed to the driver using the `.resource` and `.num_resources` elements of `struct platform_device`.

You now have to set the `.name` element of `struct platform_device` to `"uio_pdrv"` to use the generic UIO platform device driver. This driver will fill the `mem[]` array according to the resources given, and register the device.

The advantage of this approach is that you only have to edit a file you need to edit anyway. You do not have to create an extra driver.

Using uio_pdrv_genirq for platform devices

Especially in embedded devices, you frequently find chips where the irq pin is tied to its own dedicated interrupt line. In such cases, where you can be really sure the interrupt is not shared, we can take the concept of `uio_pdrv` one step further and use a generic interrupt handler. That's what `uio_pdrv_genirq` does.

The setup for this driver is the same as described above for `uio_pdrv`, except that you do not implement an interrupt handler. The `.handler` element of `struct uio_info` must remain `NULL`. The `.irq_flags` element must not contain `IRQF_SHARED`.

You will set the `.name` element of `struct platform_device` to `"uio_pdrv_genirq"` to use this driver.

The generic interrupt handler of `uio_pdrv_genirq` will simply disable the interrupt line using `disable_irq_nosync()`. After doing its work, userspace can reenable the interrupt by writing 0x00000001 to the UIO device file. The driver already implements an `irq_control()` to make this possible, you must not implement your own.

Using `uio_pdrv_genirq` not only saves a few lines of interrupt handler code. You also do not need to know anything about the chip's internal registers to create the kernel part of the driver. All you need to know is the irq number of the pin the chip is connected to.

Using uio_dmem_genirq for platform devices

In addition to statically allocated memory ranges, they may also be a desire to use dynamically allocated regions in a user space driver. In particular, being able to access memory made available through the dma-mapping API, may be particularly useful. The `uio_dmem_genirq` driver provides a way to accomplish this.

This driver is used in a similar manner to the `"uio_pdrv_genirq"` driver with respect to interrupt configuration and handling.

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 uioinfo`: 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.

Chapter 4. Writing a driver in userspace

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.

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 your 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:

http://www.osadl.org/projects/downloads/UIO/user/ [http://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.

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.

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`.

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 libray 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]