```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.2//EN"
"http://www.oasis-open.org/docbook/xml/4.2/docbookx.dtd" []>

<book id="index">
<bookinfo>
<title>The Userspace I/O HOWTO</title>

<author>
      <firstname>Hans-Jürgen</firstname>
      <surname>Koch</surname>
      <authorblurb><para>Linux developer, Linutronix</para></authorblurb>
        <affiliation>
        <orgname>
                <ulink url="http://www.linutronix.de">Linutronix</ulink>
        </orgname>

        <address>
           <email>hjk@linutronix.de</email>
        </address>
    </affiliation>
</author>

<copyright>
        <year>2006-2008</year>
        <holder>Hans-Jürgen Koch.</holder>
</copyright>
<copyright>
        <year>2009</year>
        <holder>Red Hat Inc, Michael S. Tsirkin (mst@redhat.com)</holder>
</copyright>

<legalnotice>
<para>
This documentation is Free Software licensed under the terms of the
GPL version 2.
</para>
</legalnotice>

<pubdate>2006-12-11</pubdate>

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

<revhistory>
        <revision>
        <revnumber>0.9</revnumber>
        <date>2009-07-16</date>
        <authorinitials>mst</authorinitials>
        <revremark>Added generic pci driver
                </revremark>
        </revision>
        <revision>
        <revnumber>0.8</revnumber>
```

```
        <date>2008-12-24</date>
        <authorinitials>hjk</authorinitials>
        <revremark>Added name attributes in mem and portio sysfs directories.
                </revremark>
        </revision>
        <revision>
        <revnumber>0.7</revnumber>
        <date>2008-12-23</date>
        <authorinitials>hjk</authorinitials>
        <revremark>Added generic platform drivers and offset
attribute.</revremark>
        </revision>
        <revision>
        <revnumber>0.6</revnumber>
        <date>2008-12-05</date>
        <authorinitials>hjk</authorinitials>
        <revremark>Added description of portio sysfs attributes.</revremark>
        </revision>
        <revision>
        <revnumber>0.5</revnumber>
        <date>2008-05-22</date>
        <authorinitials>hjk</authorinitials>
        <revremark>Added description of write() function.</revremark>
        </revision>
        <revision>
        <revnumber>0.4</revnumber>
        <date>2007-11-26</date>
        <authorinitials>hjk</authorinitials>
        <revremark>Removed section about uio_dummy.</revremark>
        </revision>
        <revision>
        <revnumber>0.3</revnumber>
        <date>2007-04-29</date>
        <authorinitials>hjk</authorinitials>
        <revremark>Added section about userspace drivers.</revremark>
        </revision>
        <revision>
        <revnumber>0.2</revnumber>
        <date>2007-02-13</date>
        <authorinitials>hjk</authorinitials>
        <revremark>Update after multiple mappings were added.</revremark>
        </revision>
        <revision>
        <revnumber>0.1</revnumber>
        <date>2006-12-11</date>
        <authorinitials>hjk</authorinitials>
        <revremark>First draft.</revremark>
        </revision>
</revhistory>
</bookinfo>

<chapter id="aboutthisdoc">
<?dbhtml filename="aboutthis.html"?>
<title>About this document</title>

<sect1 id="translations">
```

```
<?dbhtml filename="translations.html"?>
<title>Translations</title>

<para>If you know of any translations for this document, or you are
interested in translating it, please email me
<email>hjk@linutronix.de</email>.
</para>
</sect1>

<sect1 id="preface">
<title>Preface</title>
        <para>
        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.
        </para>
        <para>
        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.
        </para>
        <para>
        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:
        </para>
<itemizedlist>
<listitem>
        <para>The device has memory that can be mapped. The device can be
        controlled completely by writing to this memory.</para>
</listitem>
<listitem>
        <para>The device usually generates interrupts.</para>
</listitem>
<listitem>
        <para>The device does not fit into one of the standard kernel
        subsystems.</para>
</listitem>
</itemizedlist>
</sect1>

<sect1 id="thanks">
<title>Acknowledgments</title>
        <para>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.</para>
```

```
</sect1>

<sect1 id="feedback">
<title>Feedback</title>
        <para>Find something wrong with this document? (Or perhaps something
        right?) I would love to hear from you. Please email me at
        <email>hjk@linutronix.de</email>.</para>
</sect1>
</chapter>

<chapter id="about">
<?dbhtml filename="about.html"?>
<title>About UIO</title>

<para>If you use UIO for your card's driver, here's what you get:</para>

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

<sect1 id="how_uio_works">
<title>How UIO works</title>
        <para>
        Each UIO device is accessed through a device file and several
        sysfs attribute files. The device file will be called
        <filename>/dev/uio0</filename> for the first device, and
        <filename>/dev/uio1</filename>, <filename>/dev/uio2</filename>
        and so on for subsequent devices.
        </para>

        <para><filename>/dev/uioX</filename> is used to access the
        address space of the card. Just use
        <function>mmap()</function> to access registers or RAM
        locations of your card.
        </para>

        <para>
        Interrupts are handled by reading from
        <filename>/dev/uioX</filename>. A blocking
        <function>read()</function> from
        <filename>/dev/uioX</filename> will return as soon as an
        interrupt occurs. You can also use
        <function>select()</function> on
```

<filename>/dev/uioX</filename> to wait for an interrupt. The
integer value read from <filename>/dev/uioX</filename>
represents the total interrupt count. You can use this number
to figure out if you missed some interrupts.
</para>
<para>
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.
</para>
<para>
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 <filename>/dev/uioX</filename>
will call the <function>irqcontrol()</function> 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
<function>irqcontrol()</function>, <function>write()</function> will
return with <varname>-ENOSYS</varname>.
</para>

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

<para>
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
<function>uio_event_notify()</function>
from the timer's event handler.
</para>

<para>
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.
</para>

<para>
The following standard attributes are provided by the UIO

```
        framework:
        </para>
<itemizedlist>
<listitem>
        <para>
        <filename>name</filename>: The name of your device. It is
        recommended to use the name of your kernel module for this.
        </para>
</listitem>
<listitem>
        <para>
        <filename>version</filename>: 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.
        </para>
</listitem>
<listitem>
        <para>
        <filename>event</filename>: The total number of interrupts
        handled by the driver since the last time the device node was
        read.
        </para>
</listitem>
</itemizedlist>
<para>
        These attributes appear under the
        <filename>/sys/class/uio/uioX</filename> 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.
</para>
<para>
        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.
</para>
<para>
        Each mapping has its own directory in sysfs, the first mapping
        appears as <filename>/sys/class/uio/uioX/maps/map0/</filename>.
        Subsequent mappings create directories <filename>map1/</filename>,
        <filename>map2/</filename>, and so on. These directories will only
        appear if the size of the mapping is not 0.
</para>
<para>
        Each <filename>mapX/</filename> directory contains four read-only files
        that show attributes of the memory:
</para>
<itemizedlist>
<listitem>
        <para>
        <filename>name</filename>: 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.
        </para>
</listitem>
<listitem>
```

```
        <para>
        <filename>addr</filename>: The address of memory that can be mapped.
        </para>
</listitem>
<listitem>
        <para>
        <filename>size</filename>: The size, in bytes, of the memory
        pointed to by addr.
        </para>
</listitem>
<listitem>
        <para>
        <filename>offset</filename>: The offset, in bytes, that has to be
        added to the pointer returned by <function>mmap()</function> to get
        to the actual device memory. This is important if the device's memory
        is not page aligned. Remember that pointers returned by
        <function>mmap()</function> are always page aligned, so it is good
        style to always add this offset.
        </para>
</listitem>
</itemizedlist>

<para>
        From userspace, the different mappings are distinguished by adjusting
        the <varname>offset</varname> parameter of the
        <function>mmap()</function> call. To map the memory of mapping N, you
        have to use N times the page size as your offset:
</para>
<programlisting format="linespecific">
offset = N * getpagesize();
</programlisting>

<para>
        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
        <function>ioperm()</function>, <function>iopl()</function>,
        <function>inb()</function>, <function>outb()</function>, and similar
        functions.
</para>
<para>
        Since these ioport regions can not be mapped, they will not appear under
        <filename>/sys/class/uio/uioX/maps/</filename> 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.
</para>
<para>
        To address this situation, the new directory
        <filename>/sys/class/uio/uioX/portio/</filename> 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
        <filename>port0</filename>, <filename>port1</filename>, and so on,
        will appear underneath
        <filename>/sys/class/uio/uioX/portio/</filename>.
```

```
</para>
<para>
        Each <filename>portX/</filename> directory contains four read-only
        files that show name, start, size, and type of the port region:
</para>
<itemizedlist>
<listitem>
        <para>
        <filename>name</filename>: 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.
        </para>
</listitem>
<listitem>
        <para>
        <filename>start</filename>: The first port of this region.
        </para>
</listitem>
<listitem>
        <para>
        <filename>size</filename>: The number of ports in this region.
        </para>
</listitem>
<listitem>
        <para>
        <filename>porttype</filename>: A string describing the type of port.
        </para>
</listitem>
</itemizedlist>


</sect1>
</chapter>

<chapter id="custom_kernel_module" xreflabel="Writing your own kernel module">
<?dbhtml filename="custom_kernel_module.html"?>
<title>Writing your own kernel module</title>
        <para>
        Please have a look at <filename>uio_cif.c</filename> as an
        example. The following paragraphs explain the different
        sections of this file.
        </para>

<sect1 id="uio_info">
<title>struct uio_info</title>
        <para>
        This structure tells the framework the details of your driver,
        Some of the members are required, others are optional.
        </para>

<itemizedlist>
<listitem><para>
<varname>const char *name</varname>: Required. The name of your driver as
it will appear in sysfs. I recommend using the name of your module for this.
</para></listitem>
```

<listitem><para>
<varname>const char *version</varname>: Required. This string appears in
<filename>/sys/class/uio/uioX/version</filename>.
</para></listitem>

<listitem><para>
<varname>struct uio_mem mem[MAX_UIO_MAPS]</varname>: Required if you
have memory that can be mapped with <function>mmap()</function>. For each
mapping you need to fill one of the <varname>uio_mem</varname> structures.
See the description below for details.
</para></listitem>

<listitem><para>
<varname>struct uio_port port[MAX_UIO_PORTS_REGIONS]</varname>: Required
if you want to pass information about ioports to userspace. For each port
region you need to fill one of the <varname>uio_port</varname> structures.
See the description below for details.
</para></listitem>

<listitem><para>
<varname>long irq</varname>: 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
<varname>irq</varname> to <varname>UIO_IRQ_CUSTOM</varname>.
If you had no interrupt at all, you could set
<varname>irq</varname> to <varname>UIO_IRQ_NONE</varname>, though this
rarely makes sense.
</para></listitem>

<listitem><para>
<varname>unsigned long irq_flags</varname>: Required if you've set
<varname>irq</varname> to a hardware interrupt number. The flags given
here will be used in the call to <function>request_irq()</function>.
</para></listitem>

<listitem><para>
<varname>int (*mmap)(struct uio_info *info, struct vm_area_struct
*vma)</varname>: Optional. If you need a special
<function>mmap()</function> function, you can set it here. If this
pointer is not NULL, your <function>mmap()</function> will be called
instead of the built-in one.
</para></listitem>

<listitem><para>
<varname>int (*open)(struct uio_info *info, struct inode *inode)
</varname>: Optional. You might want to have your own
<function>open()</function>, e.g. to enable interrupts only when your
device is actually used.
</para></listitem>

<listitem><para>
<varname>int (*release)(struct uio_info *info, struct inode *inode)
</varname>: Optional. If you define your own
<function>open()</function>, you will probably also want a custom
<function>release()</function> function.

</para></listitem>

<listitem><para>
<varname>int (*irqcontrol)(struct uio_info *info, s32 irq_on)
</varname>: Optional. If you need to be able to enable or disable
interrupts from userspace by writing to <filename>/dev/uioX</filename>,
you can implement this function. The parameter <varname>irq_on</varname>
will be 0 to disable interrupts and 1 to enable them.
</para></listitem>
</itemizedlist>

<para>
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
<varname>struct uio_mem</varname> in the <varname>mem[]</varname> array.
Here's a description of the fields of <varname>struct uio_mem</varname>:
</para>

<itemizedlist>
<listitem><para>
<varname>int memtype</varname>: Required if the mapping is used. Set this to
<varname>UIO_MEM_PHYS</varname> if you you have physical memory on your
card to be mapped. Use <varname>UIO_MEM_LOGICAL</varname> for logical
memory (e.g. allocated with <function>kmalloc()</function>). There's also
<varname>UIO_MEM_VIRTUAL</varname> for virtual memory.
</para></listitem>

<listitem><para>
<varname>unsigned long addr</varname>: Required if the mapping is used.
Fill in the address of your memory block. This address is the one that
appears in sysfs.
</para></listitem>

<listitem><para>
<varname>unsigned long size</varname>: Fill in the size of the
memory block that <varname>addr</varname> points to. If <varname>size</varname>
is zero, the mapping is considered unused. Note that you
<emphasis>must</emphasis> initialize <varname>size</varname> with zero for
all unused mappings.
</para></listitem>

<listitem><para>
<varname>void *internal_addr</varname>: If you have to access this memory
region from within your kernel module, you will want to map it internally by
using something like <function>ioremap()</function>. Addresses
returned by this function cannot be mapped to user space, so you must not
store it in <varname>addr</varname>. Use <varname>internal_addr</varname>
instead to remember such an address.
</para></listitem>
</itemizedlist>

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

<para>
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 <varname>struct uio_port</varname> in the <varname>port[]</varname> array. Here's a description of the fields of <varname>struct uio_port</varname>:
</para>

<itemizedlist>
<listitem><para>
<varname>char *porttype</varname>: Required. Set this to one of the predefined constants. Use <varname>UIO_PORT_X86</varname> for the ioports found in x86 architectures.
</para></listitem>

<listitem><para>
<varname>unsigned long start</varname>: Required if the port region is used. Fill in the number of the first port of this region.
</para></listitem>

<listitem><para>
<varname>unsigned long size</varname>: Fill in the number of ports in this region. If <varname>size</varname> is zero, the region is considered unused. Note that you <emphasis>must</emphasis> initialize <varname>size</varname> with zero for all unused regions.
</para></listitem>
</itemizedlist>

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

</sect1>

<sect1 id="adding_irq_handler">
<title>Adding an interrupt handler</title>
	<para>
	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 <emphasis>have</emphasis> to perform after each interrupt, then your handler can be empty.</para> <para>If, on the other hand, your hardware <emphasis>needs</emphasis> some action to be performed after each interrupt, then you <emphasis>must</emphasis> 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.
	</para>

```
        <para>
        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.
        </para>

        <para>
        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.
        </para>

        <para>
        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.
        </para>
</sect1>

<sect1 id="using_uio_pdrv">
<title>Using uio_pdrv for platform devices</title>
        <para>
        In many cases, UIO drivers for platform devices can be handled in a
        generic way. In the same place where you define your
        <varname>struct platform_device</varname>, you simply also implement
        your interrupt handler and fill your
        <varname>struct uio_info</varname>. A pointer to this
        <varname>struct uio_info</varname> is then used as
        <varname>platform_data</varname> for your platform device.
        </para>
        <para>
        You also need to set up an array of <varname>struct resource</varname>
        containing addresses and sizes of your memory mappings. This
        information is passed to the driver using the
        <varname>.resource</varname> and <varname>.num_resources</varname>
        elements of <varname>struct platform_device</varname>.
        </para>
        <para>
        You now have to set the <varname>.name</varname> element of
        <varname>struct platform_device</varname> to
        <varname>"uio_pdrv"</varname> to use the generic UIO platform device
        driver. This driver will fill the <varname>mem[]</varname> array
        according to the resources given, and register the device.
        </para>
        <para>
        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.
</para>
</sect1>

<sect1 id="using_uio_pdrv_genirq">
<title>Using uio_pdrv_genirq for platform devices</title>
<para>
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 <varname>uio_pdrv</varname> one step further and use a generic interrupt handler. That's what <varname>uio_pdrv_genirq</varname> does.
</para>
<para>
The setup for this driver is the same as described above for <varname>uio_pdrv</varname>, except that you do not implement an interrupt handler. The <varname>.handler</varname> element of <varname>struct uio_info</varname> must remain <varname>NULL</varname>. The <varname>.irq_flags</varname> element must not contain <varname>IRQF_SHARED</varname>.
</para>
<para>
You will set the <varname>.name</varname> element of <varname>struct platform_device</varname> to <varname>"uio_pdrv_genirq"</varname> to use this driver.
</para>
<para>
The generic interrupt handler of <varname>uio_pdrv_genirq</varname> will simply disable the interrupt line using <function>disable_irq_nosync()</function>. After doing its work, userspace can reenable the interrupt by writing 0x00000001 to the UIO device file. The driver already implements an <function>irq_control()</function> to make this possible, you must not implement your own.
</para>
<para>
Using <varname>uio_pdrv_genirq</varname> 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.
</para>
</sect1>

</chapter>

<chapter id="userspace_driver" xreflabel="Writing a driver in user space">
<?dbhtml filename="userspace_driver.html"?>
<title>Writing a driver in userspace</title>
<para>
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.
        </para>

<sect1 id="getting_uio_information">
<title>Getting information about your UIO device</title>
        <para>
        Information about all UIO devices is available in sysfs. The
        first thing you should do in your driver is check
        <varname>name</varname> and <varname>version</varname> to
        make sure your talking to the right device and that its kernel
        driver has the version you expect.
        </para>
        <para>
        You should also make sure that the memory mapping you need
        exists and has the size you expect.
        </para>
        <para>
        There is a tool called <varname>lsuio</varname> that lists
        UIO devices and their attributes. It is available here:
        </para>
        <para>
        <ulink url="http://www.osadl.org/projects/downloads/UIO/user/">
                http://www.osadl.org/projects/downloads/UIO/user/</ulink>
        </para>
        <para>
        With <varname>lsuio</varname> you can quickly check if your
        kernel module is loaded and which attributes it exports.
        Have a look at the manpage for details.
        </para>
        <para>
        The source code of <varname>lsuio</varname> can serve as an
        example for getting information about an UIO device.
        The file <filename>uio_helper.c</filename> contains a lot of
        functions you could use in your userspace driver code.
        </para>
</sect1>

<sect1 id="mmap_device_memory">
<title>mmap() device memory</title>
        <para>
        After you made sure you've got the right device with the
        memory mappings you need, all you have to do is to call
        <function>mmap()</function> to map the device's memory
        to userspace.
        </para>
        <para>
        The parameter <varname>offset</varname> of the
        <function>mmap()</function> 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:
        </para>
<programlisting format="linespecific">
        offset = N * getpagesize();
</programlisting>
```

```
        <para>
        N starts from zero, so if you've got only one memory
        range to map, set <varname>offset = 0</varname>.
        A drawback of this technique is that memory is always
        mapped beginning with its start address.
        </para>
</sect1>

<sect1 id="wait_for_interrupts">
<title>Waiting for interrupts</title>
        <para>
        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 occured.
        </para>
        <para>
        <filename>/dev/uioX</filename> is a read-only file. A
        <function>read()</function> will always block until an
        interrupt occurs. There is only one legal value for the
        <varname>count</varname> parameter of
        <function>read()</function>, and that is the size of a
        signed 32 bit integer (4). Any other value for
        <varname>count</varname> causes <function>read()</function>
        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.
        </para>
        <para>
        You can also use <function>select()</function> on
        <filename>/dev/uioX</filename>.
        </para>
</sect1>

</chapter>

<chapter id="uio_pci_generic" xreflabel="Using Generic driver for PCI cards">
<?dbhtml filename="uio_pci_generic.html"?>
<title>Generic PCI UIO driver</title>
        <para>
        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.
        </para>

<sect1 id="uio_pci_generic_binding">
<title>Making the driver recognize the device</title>
        <para>
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:
```

```
        <programlisting>
 modprobe uio_pci_generic
 echo &quot;8086 10f5&quot; &gt; /sys/bus/pci/drivers/uio_pci_generic/new_id
        </programlisting>
        </para>
        <para>
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:
        <programlisting>
    echo -n 0000:00:19.0 &gt; /sys/bus/pci/drivers/e1000e/unbind
    echo -n 0000:00:19.0 &gt; /sys/bus/pci/drivers/uio_pci_generic/bind
        </programlisting>
        </para>
        <para>
You can verify that the device has been bound to the driver
by looking for it in sysfs, for example like the following:
        <programlisting>
    ls -l /sys/bus/pci/devices/0000:00:19.0/driver
        </programlisting>
Which if successful should print
        <programlisting>
  .../0000:00:19.0/driver -&gt; ../../../bus/pci/drivers/uio_pci_generic
        </programlisting>
Note that the generic driver will not bind to old PCI 2.2 devices.
If binding the device failed, run the following command:
        <programlisting>
  dmesg
        </programlisting>
and look in the output for failure reasons
        </para>
</sect1>

<sect1 id="uio_pci_generic_internals">
<title>Things to know about uio_pci_generic</title>
        <para>
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.
        </para>
        <para>
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.
        </para>
</sect1>
<sect1 id="uio_pci_generic_userspace">
<title>Writing userspace driver using uio_pci_generic</title>
        <para>
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.
```

```
        </para>
</sect1>
<sect1 id="uio_pci_generic_example">
<title>Example code using uio_pci_generic</title>
        <para>
Here is some sample userspace driver code using uio_pci_generic:
<programlisting>
#include &lt;stdlib.h&gt;
#include &lt;stdio.h&gt;
#include &lt;unistd.h&gt;
#include &lt;sys/types.h&gt;
#include &lt;sys/stat.h&gt;
#include &lt;fcntl.h&gt;
#include &lt;errno.h&gt;

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

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

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

        for(i = 0;; ++i) {
                /* Print out a message, for debugging. */
                if (i == 0)
                        fprintf(stderr, &quot;Started uio test driver.\n&quot;);
                else
                        fprintf(stderr, &quot;Interrupts: %d\n&quot;, icount);

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

                /* Re-enable interrupts. */
```

```
                err = pwrite(configfd, &amp;command_high, 1, 5);
                if (err != 1) {
                        perror(&quot;config write:&quot;);
                        break;
                }

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

        }
        return errno;
}

</programlisting>
        </para>
</sect1>

</chapter>

<appendix id="app1">
<title>Further information</title>
<itemizedlist>
        <listitem><para>
                        <ulink url="http://www.osadl.org">
                                OSADL homepage.</ulink>
                </para></listitem>
        <listitem><para>
                <ulink url="http://www.linutronix.de">
                 Linutronix homepage.</ulink>
                </para></listitem>
</itemizedlist>
</appendix>

</book>
```