```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book PUBLIC "-//OASIS//DTD DocBook XML V4.1.2//EN"
        "http://www.oasis-open.org/docbook/xml/4.1.2/docbookx.dtd" []>

<book id="DoingIO">
 <bookinfo>
  <title>Bus-Independent Device Accesses</title>

  <authorgroup>
   <author>
    <firstname>Matthew</firstname>
    <surname>Wilcox</surname>
    <affiliation>
     <address>
      <email>matthew@wil.cx</email>
     </address>
    </affiliation>
   </author>
  </authorgroup>

  <authorgroup>
   <author>
    <firstname>Alan</firstname>
    <surname>Cox</surname>
    <affiliation>
     <address>
      <email>alan@lxorguk.ukuu.org.uk</email>
     </address>
    </affiliation>
   </author>
  </authorgroup>

  <copyright>
   <year>2001</year>
   <holder>Matthew Wilcox</holder>
  </copyright>

  <legalnotice>
   <para>
     This documentation is free software; you can redistribute
     it and/or modify it under the terms of the GNU General Public
     License as published by the Free Software Foundation; either
     version 2 of the License, or (at your option) any later
     version.
   </para>

   <para>
     This program is distributed in the hope that it will be
     useful, but WITHOUT ANY WARRANTY; without even the implied
     warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.
     See the GNU General Public License for more details.
   </para>

   <para>
     You should have received a copy of the GNU General Public
     License along with this program; if not, write to the Free
```

Software Foundation, Inc., 59 Temple Place, Suite 330, Boston,
MA 02111-1307 USA
</para>

<para>
For more details see the file COPYING in the source
distribution of Linux.
</para>
</legalnotice>
</bookinfo>

<toc></toc>

<chapter id="intro">
<title>Introduction</title>
<para>
Linux provides an API which abstracts performing IO across all busses
and devices, allowing device drivers to be written independently of
bus type.
</para>
</chapter>

<chapter id="bugs">
<title>Known Bugs And Assumptions</title>
<para>
None.
</para>
</chapter>

<chapter id="mmio">
<title>Memory Mapped IO</title>
<sect1 id="getting_access_to_the_device">
<title>Getting Access to the Device</title>
<para>
The most widely supported form of IO is memory mapped IO.
That is, a part of the CPU's address space is interpreted
not as accesses to memory, but as accesses to a device. Some
architectures define devices to be at a fixed address, but most
have some method of discovering devices. The PCI bus walk is a
good example of such a scheme. This document does not cover how
to receive such an address, but assumes you are starting with one.
Physical addresses are of type unsigned long.
</para>

<para>
This address should not be used directly. Instead, to get an
address suitable for passing to the accessor functions described
below, you should call <function>ioremap</function>.
An address suitable for accessing the device will be returned to you.
</para>

<para>
After you've finished using the device (say, in your module's
exit routine), call <function>iounmap</function> in order to return
the address space to the kernel. Most architectures allocate new
address space each time you call <function>ioremap</function>, and

```
    they can run out unless you call <function>iounmap</function>.
  </para>
</sect1>

<sect1 id="accessing_the_device">
  <title>Accessing the device</title>
  <para>
    The part of the interface most used by drivers is reading and
    writing memory-mapped registers on the device.  Linux provides
    interfaces to read and write 8-bit, 16-bit, 32-bit and 64-bit
    quantities.  Due to a historical accident, these are named byte,
    word, long and quad accesses.  Both read and write accesses are
    supported; there is no prefetch support at this time.
  </para>

  <para>
    The functions are named <function>readb</function>,
    <function>readw</function>, <function>readl</function>,
    <function>readq</function>, <function>readb_relaxed</function>,
    <function>readw_relaxed</function>, <function>readl_relaxed</function>,
    <function>readq_relaxed</function>, <function>writeb</function>,
    <function>writew</function>, <function>writel</function> and
    <function>writeq</function>.
  </para>

  <para>
    Some devices (such as framebuffers) would like to use larger
    transfers than 8 bytes at a time.  For these devices, the
    <function>memcpy_toio</function>, <function>memcpy_fromio</function>
    and <function>memset_io</function> functions are provided.
    Do not use memset or memcpy on IO addresses; they
    are not guaranteed to copy data in order.
  </para>

  <para>
    The read and write functions are defined to be ordered. That is the
    compiler is not permitted to reorder the I/O sequence. When the
    ordering can be compiler optimised, you can use <function>
    __readb</function> and friends to indicate the relaxed ordering. Use
    this with care.
  </para>

  <para>
    While the basic functions are defined to be synchronous with respect
    to each other and ordered with respect to each other the busses the
    devices sit on may themselves have asynchronicity. In particular many
    authors are burned by the fact that PCI bus writes are posted
    asynchronously. A driver author must issue a read from the same
    device to ensure that writes have occurred in the specific cases the
    author cares. This kind of property cannot be hidden from driver
    writers in the API.  In some cases, the read used to flush the device
    may be expected to fail (if the card is resetting, for example).  In
    that case, the read should be done from config space, which is
    guaranteed to soft-fail if the card doesn't respond.
  </para>
```

```
      <para>
        The following is an example of flushing a write to a device when
        the driver would like to ensure the write's effects are visible prior
        to continuing execution.
      </para>

<programlisting>
static inline void
qla1280_disable_intrs(struct scsi_qla_host *ha)
{
        struct device_reg *reg;

        reg = ha->iobase;
        /* disable risc and host interrupts */
        WRT_REG_WORD(&amp;reg->ictrl, 0);
        /*
         * The following read will ensure that the above write
         * has been received by the device before we return from this
         * function.
         */
        RD_REG_WORD(&amp;reg->ictrl);
        ha->flags.ints_enabled = 0;
}
</programlisting>

      <para>
        In addition to write posting, on some large multiprocessing systems
        (e.g. SGI Challenge, Origin and Altix machines) posted writes won't
        be strongly ordered coming from different CPUs.  Thus it's important
        to properly protect parts of your driver that do memory-mapped writes
        with locks and use the <function>mmiowb</function> to make sure they
        arrive in the order intended.  Issuing a regular <function>readX
        </function> will also ensure write ordering, but should only be used
        when the driver has to be sure that the write has actually arrived
        at the device (not that it's simply ordered with respect to other
        writes), since a full <function>readX</function> is a relatively
        expensive operation.
      </para>

      <para>
        Generally, one should use <function>mmiowb</function> prior to
        releasing a spinlock that protects regions using <function>writeb
        </function> or similar functions that aren't surrounded by <function>
        readb</function> calls, which will ensure ordering and flushing.  The
        following pseudocode illustrates what might occur if write ordering
        isn't guaranteed via <function>mmiowb</function> or one of the
        <function>readX</function> functions.
      </para>

<programlisting>
CPU A:  spin_lock_irqsave(&amp;dev_lock, flags)
CPU A:  ...
CPU A:  writel(newval, ring_ptr);
CPU A:  spin_unlock_irqrestore(&amp;dev_lock, flags)
        ...
CPU B:  spin_lock_irqsave(&amp;dev_lock, flags)
```

```
CPU B:  writel(newval2, ring_ptr);
CPU B:  ...
CPU B:  spin_unlock_irqrestore(&amp;dev_lock, flags)
</programlisting>

      <para>
        In the case above, newval2 could be written to ring_ptr before
        newval.  Fixing it is easy though:
      </para>

<programlisting>
CPU A:  spin_lock_irqsave(&amp;dev_lock, flags)
CPU A:  ...
CPU A:  writel(newval, ring_ptr);
CPU A:  mmiowb(); /* ensure no other writes beat us to the device */
CPU A:  spin_unlock_irqrestore(&amp;dev_lock, flags)
        ...
CPU B:  spin_lock_irqsave(&amp;dev_lock, flags)
CPU B:  writel(newval2, ring_ptr);
CPU B:  ...
CPU B:  mmiowb();
CPU B:  spin_unlock_irqrestore(&amp;dev_lock, flags)
</programlisting>

      <para>
        See tg3.c for a real world example of how to use <function>mmiowb
        </function>
      </para>

      <para>
        PCI ordering rules also guarantee that PIO read responses arrive
        after any outstanding DMA writes from that bus, since for some devices
        the result of a <function>readb</function> call may signal to the
        driver that a DMA transaction is complete.  In many cases, however,
        the driver may want to indicate that the next
        <function>readb</function> call has no relation to any previous DMA
        writes performed by the device.  The driver can use
        <function>readb_relaxed</function> for these cases, although only
        some platforms will honor the relaxed semantics.  Using the relaxed
        read functions will provide significant performance benefits on
        platforms that support it.  The qla2xxx driver provides examples
        of how to use <function>readX_relaxed</function>.  In many cases,
        a majority of the driver's <function>readX</function> calls can
        safely be converted to <function>readX_relaxed</function> calls, since
        only a few will indicate or depend on DMA completion.
      </para>
    </sect1>

  </chapter>

  <chapter id="port_space_accesses">
    <title>Port Space Accesses</title>
    <sect1 id="port_space_explained">
      <title>Port Space Explained</title>

      <para>
```

```
        Another form of IO commonly supported is Port Space.  This is a
        range of addresses separate to the normal memory address space.
        Access to these addresses is generally not as fast as accesses
        to the memory mapped addresses, and it also has a potentially
        smaller address space.
      </para>

      <para>
        Unlike memory mapped IO, no preparation is required
        to access port space.
      </para>

    </sect1>
    <sect1 id="accessing_port_space">
      <title>Accessing Port Space</title>
      <para>
        Accesses to this space are provided through a set of functions
        which allow 8-bit, 16-bit and 32-bit accesses; also
        known as byte, word and long.  These functions are
        <function>inb</function>, <function>inw</function>,
        <function>inl</function>, <function>outb</function>,
        <function>outw</function> and <function>outl</function>.
      </para>

      <para>
        Some variants are provided for these functions.  Some devices
        require that accesses to their ports are slowed down.  This
        functionality is provided by appending a <function>_p</function>
        to the end of the function.  There are also equivalents to memcpy.
        The <function>ins</function> and <function>outs</function>
        functions copy bytes, words or longs to the given port.
      </para>
    </sect1>

  </chapter>

  <chapter id="pubfunctions">
     <title>Public Functions Provided</title>
!Iarch/x86/include/asm/io.h
!Elib/iomap.c
  </chapter>

</book>
```