# Bus-Independent Device Accesses

**Matthew Wilcox**

**matthew@wil.cx**

**Alan Cox**

**alan@lxorguk.ukuu.org.uk**

**Bus-Independent Device Accesses**

by Matthew Wilcox

by Alan Cox

Copyright © 2001 Matthew Wilcox

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.

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.

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

For more details see the file COPYING in the source distribution of Linux.

# Table of Contents

# Chapter 1. Introduction

Linux provides an API which abstracts performing IO across all busses and devices, allowing device drivers to be written independently of bus type.

# Chapter 2. Known Bugs And Assumptions

None.

# Chapter 3. Memory Mapped IO

## 3.1. Getting Access to the Device

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.

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

After you've finished using the device (say, in your module's exit routine), call `iounmap` in order to return the address space to the kernel. Most architectures allocate new address space each time you call `ioremap`, and they can run out unless you call `iounmap`.

## 3.2. Accessing the device

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.

The functions are named `readb`, `readw`, `readl`, `readq`, `readb_relaxed`, `readw_relaxed`, `readl_relaxed`, `readq_relaxed`, `writeb`, `writew`, `writel` and `writeq`.

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

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 `__readb` and friends to indicate the relaxed ordering. Use this with care.

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.

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.

```
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(&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(&reg->ictrl);
 ha->flags.ints_enabled = 0;
}
```

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 `mmiowb` to make sure they arrive in the order intended. Issuing a regular `readX` 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 `readX` is a relatively expensive operation.

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

```
CPU A:  spin_lock_irqsave(&dev_lock, flags)
CPU A:  ...
CPU A:  writel(newval, ring_ptr);
CPU A:  spin_unlock_irqrestore(&dev_lock, flags)
        ...
CPU B:  spin_lock_irqsave(&dev_lock, flags)
CPU B:  writel(newval2, ring_ptr);
CPU B:  ...
CPU B:  spin_unlock_irqrestore(&dev_lock, flags)
```

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

```
CPU A:  spin_lock_irqsave(&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(&dev_lock, flags)
        ...
CPU B:  spin_lock_irqsave(&dev_lock, flags)
CPU B:  writel(newval2, ring_ptr);
CPU B:  ...
CPU B:  mmiowb();
CPU B:  spin_unlock_irqrestore(&dev_lock, flags)
```

See tg3.c for a real world example of how to use `mmiowb`

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 `readb` 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 `readb` call has no relation to any previous DMA writes performed by the device. The driver can use `readb_relaxed` 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 `readX_relaxed`. In many cases, a majority of the driver's `readX` calls can safely be converted to `readX_relaxed` calls, since only a few will indicate or depend on DMA completion.

# Chapter 4. Port Space Accesses

## 4.1. Port Space Explained

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.

Unlike memory mapped IO, no preparation is required to access port space.

## 4.2. Accessing Port Space

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 `inb`, `inw`, `inl`, `outb`, `outw` and `outl`.

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 `_p` to the end of the function. There are also equivalents to memcpy. The `ins` and `outs` functions copy bytes, words or longs to the given port.

# Chapter 5. Public Functions Provided

## virt_to_phys

**LINUX**

Kernel Hackers ManualApril 2009

### Name

`virt_to_phys` — map virtual addresses to physical

### Synopsis

```
unsigned long virt_to_phys (volatile void * address);
```

### Arguments

*`address`*

address to remap

### Description

The returned physical address is the physical (CPU) mapping for the memory address given. It is only valid to use this function on addresses directly mapped or allocated via kmalloc.

This function does not give bus mappings for DMA transfers. In almost all conceivable cases a device driver should not be using this function

## phys_to_virt

**LINUX**

## Name

`phys_to_virt` — map physical address to virtual

## Synopsis

```
void * phys_to_virt (unsigned long address);
```

## Arguments

*address*

address to remap

## Description

The returned virtual address is a current CPU mapping for the memory address given. It is only valid to use this function on addresses that have a kernel mapping

This function does not handle bus mappings for DMA transfers. In almost all conceivable cases a device driver should not be using this function

# ioremap_nocache

### LINUX



## Name

`ioremap_nocache` — map bus memory into CPU space

## Synopsis

```
void __iomem * ioremap_nocache (resource_size_t offset, unsigned long size);
```

## Arguments

*offset*

bus address of the memory

*size*

size of the resource to map

## Description

ioremap performs a platform specific sequence of operations to make bus memory CPU accessible via the readb/readw/readl/writeb/ writew/writel functions and the other mmio helpers. The returned address is not guaranteed to be usable directly as a virtual address.

If the area you are trying to map is a PCI BAR you should have a look at `pci_iomap`.

# pci_iomap

**LINUX**

Kernel Hackers ManualApril 2009

## Name

`pci_iomap` — create a virtual mapping cookie for a PCI BAR

## Synopsis

```
void __iomem * pci_iomap (struct pci_dev * dev, int bar, unsigned long
maxlen);
```

## Arguments

*dev*

PCI device that owns the BAR

*bar*

BAR number

*maxlen*

length of the memory to map

## Description

Using this function you will get a __iomem address to your device BAR. You can access it using ioread*() and iowrite*(). These functions hide the details if this is a MMIO or PIO address space and will just do what you expect from them in the correct way.

*maxlen* specifies the maximum length to map. If you want to get access to the complete BAR without checking for its length first, pass `0` here.