Bus-Independent Device Accesses

Matthew Wilcox

matthew@wil.cx

Alan Cox

alan@redhat.com

Bus-Independent Device Accesses

by Matthew Wilcox

by Alan Cox

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`, `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. The `rmb` provides a read memory barrier. The `wmb` provides a write memory barrier.

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 asynchronocity. In paticular 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.

3.3. ISA legacy functions

On older kernels (2.2 and earlier) the ISA bus could be read or written with these functions and without ioremap being used. This is no longer true in Linux 2.4. A set of equivalent functions exist for easy legacy driver porting. The functions available are prefixed with 'isa_' and are `isa_readb`, `isa_writeb`, `isa_readw`, `isa_writew`, `isa_readl`, `isa_writel`, `isa_memcpy_fromio` and `isa_memcpy_toio`

These functions should not be used in new drivers, and will eventually be going away.

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

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

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

Name

`ioremap` — map bus memory into CPU space

Synopsis

```
void * ioremap (unsigned long 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.

ioremap_nocache

Name

`ioremap_nocache` — map bus memory into CPU space

Synopsis

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

Arguments

offset

bus address of the memory

size

size of the resource to map

Description

ioremap_nocache 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.

This version of ioremap ensures that the memory is marked uncachable on the CPU as well as honouring existing caching rules from things like the PCI bus. Note that there are other caches and buffers on many busses. In paticular driver authors should read up on PCI writes

It's useful if some control registers are in such an area and

check_signature

Name

`check_signature` — find BIOS signatures

Synopsis

```
int check_signature (unsigned long io_addr, const unsigned char * signature,
int length);
```

Arguments

io_addr

mmio address to check

signature

signature block

length

length of signature

Description

Perform a signature comparison with the mmio address io_addr. This address should have been obtained by ioremap. Returns 1 on a match.

isa_check_signature

Name

`isa_check_signature` — find BIOS signatures

Synopsis

```
int isa_check_signature (unsigned long io_addr, const unsigned char *
signature, int length);
```

Arguments

`io_addr`

mmio address to check

`signature`

signature block

`length`

length of signature

Description

Perform a signature comparison with the ISA mmio address io_addr. Returns 1 on a match.

This function is deprecated. New drivers should use ioremap and check_signature.