Bus-Independent Device Accesses

Author: Matthew Wilcox
Author: Alan Cox

Introduction

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

Memory Mapped IO

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().

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;
}
```

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.

Port Space Accesses

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.

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.

__iomem pointer tokens

The data type for an MMIO address is an `__iomem` qualified pointer, such as `void __iomem *reg`. On most architectures it is a regular pointer that points to a virtual memory address and can be offset or dereferenced, but in portable code, it must only be passed from and to functions that explicitly operated on an `__iomem` token, in particular the ioremap() and readl()/writel() functions. The 'sparse' semantic code checker can be used to verify that this is done correctly.

While on most architectures, ioremap() creates a page table entry for an uncached virtual address pointing to the physical MMIO address, some architectures require special instructions for MMIO, and the `__iomem` pointer just encodes the physical address or an offsettable cookie that is interpreted by readl()/writel().

Differences between I/O access functions

readq(), readl(), readw(), readb(), writeq(), writel(), writew(), writeb()

> These are the most generic accessors, providing serialization against other MMIO accesses and DMA accesses as well as fixed endianness for accessing little-endian PCI devices and on-chip peripherals. Portable device drivers should generally use these for any access to `__iomem` pointers.
>
> Note that posted writes are not strictly ordered against a spinlock, see Documentation/driver-api/io_ordering.rst.

readq_relaxed(), readl_relaxed(), readw_relaxed(), readb_relaxed(), writeq_relaxed(), writel_relaxed(), writew_relaxed(), writeb_relaxed()

> On architectures that require an expensive barrier for serializing against DMA, these "relaxed" versions of the MMIO accessors only serialize against each other, but contain a less expensive barrier operation. A device driver might use these in a particularly performance sensitive fast path, with a comment that explains why the usage in a specific location is safe without the extra barriers.
>
> See memory-barriers.txt for a more detailed discussion on the precise ordering guarantees of the non-relaxed and relaxed versions.

ioread64(), ioread32(), ioread16(), ioread8(), iowrite64(), iowrite32(), iowrite16(), iowrite8()

> These are an alternative to the normal readl()/writel() functions, with almost identical behavior, but they can also operate on `__iomem` tokens returned for mapping PCI I/O space with pci_iomap() or ioport_map(). On architectures that require special instructions for I/O port access, this adds a small overhead for an indirect function call implemented in lib/iomap.c, while on other architectures, these are simply aliases.

ioread64be(), ioread32be(), ioread16be() iowrite64be(), iowrite32be(), iowrite16be()

> These behave in the same way as the ioread32()/iowrite32() family, but with reversed byte order, for accessing devices with big-endian MMIO registers. Device drivers that can operate on either big-endian or little-endian registers may have to implement a custom wrapper function that picks one or the other depending on which device was found.
>
> Note: On some architectures, the normal readl()/writel() functions traditionally assume that devices are the same

endianness as the CPU, while using a hardware byte-reverse on the PCI bus when running a big-endian kernel. Drivers that use readl()/writel() this way are generally not portable, but tend to be limited to a particular SoC.

hi_lo_readq(), lo_hi_readq(), hi_lo_readq_relaxed(), lo_hi_readq_relaxed(), ioread64_lo_hi(), ioread64_hi_lo(), ioread64be_lo_hi(), ioread64be_hi_lo(), hi_lo_writeq(), lo_hi_writeq(), hi_lo_writeq_relaxed(), lo_hi_writeq_relaxed(), iowrite64_lo_hi(), iowrite64_hi_lo(), iowrite64be_lo_hi(), iowrite64be_hi_lo()

Some device drivers have 64-bit registers that cannot be accessed atomically on 32-bit architectures but allow two consecutive 32-bit accesses instead. Since it depends on the particular device which of the two halves has to be accessed first, a helper is provided for each combination of 64-bit accessors with either low/high or high/low word ordering. A device driver must include either <linux/io-64-nonatomic-lo-hi.h> or <linux/io-64-nonatomic-hi-lo.h> to get the function definitions along with helpers that redirect the normal readq()/writeq() to them on architectures that do not provide 64-bit access natively.

__raw_readq(), __raw_readl(), __raw_readw(), __raw_readb(), __raw_writeq(), __raw_writel(), __raw_writew(), __raw_writeb()

These are low-level MMIO accessors without barriers or byteorder changes and architecture specific behavior. Accesses are usually atomic in the sense that a four-byte __raw_readl() does not get split into individual byte loads, but multiple consecutive accesses can be combined on the bus. In portable code, it is only safe to use these to access memory behind a device bus but not MMIO registers, as there are no ordering guarantees with regard to other MMIO accesses or even spinlocks. The byte order is generally the same as for normal memory, so unlike the other functions, these can be used to copy data between kernel memory and device memory.

inl(), inw(), inb(), outl(), outw(), outb()

PCI I/O port resources traditionally require separate helpers as they are implemented using special instructions on the x86 architecture. On most other architectures, these are mapped to readl()/writel() style accessors internally, usually pointing to a fixed area in virtual memory. Instead of an `__iomem` pointer, the address is a 32-bit integer token to identify a port number. PCI requires I/O port access to be non-posted, meaning that an outb() must complete before the following code executes, while a normal writeb() may still be in progress. On architectures that correctly implement this, I/O port access is therefore ordered against spinlocks. Many non-x86 PCI host bridge implementations and CPU architectures however fail to implement non-posted I/O space on PCI, so they can end up being posted on such hardware.

In some architectures, the I/O port number space has a 1:1 mapping to `__iomem` pointers, but this is not recommended and device drivers should not rely on that for portability. Similarly, an I/O port number as described in a PCI base address register may not correspond to the port number as seen by a device driver. Portable drivers need to read the port number for the resource provided by the kernel.

There are no direct 64-bit I/O port accessors, but pci_iomap() in combination with ioread64/iowrite64 can be used instead.

inl_p(), inw_p(), inb_p(), outl_p(), outw_p(), outb_p()

On ISA devices that require specific timing, the _p versions of the I/O accessors add a small delay. On architectures that do not have ISA buses, these are aliases to the normal inb/outb helpers.

readsq, readsl, readsw, readsb writesq, writesl, writesw, writesb ioread64_rep, ioread32_rep, ioread16_rep, ioread8_rep iowrite64_rep, iowrite32_rep, iowrite16_rep, iowrite8_rep insl, insw, insb, outsl, outsw, outsb

These are helpers that access the same address multiple times, usually to copy data between kernel memory byte stream and a FIFO buffer. Unlike the normal MMIO accessors, these do not perform a byteswap on big-endian kernels, so the first byte in the FIFO register corresponds to the first byte in the memory buffer regardless of the architecture.

Device memory mapping modes

Some architectures support multiple modes for mapping device memory. ioremap_*() variants provide a common abstraction around these architecture-specific modes, with a shared set of semantics.

ioremap() is the most common mapping type, and is applicable to typical device memory (e.g. I/O registers). Other modes can offer weaker or stronger guarantees, if supported by the architecture. From most to least common, they are as follows:

ioremap()

The default mode, suitable for most memory-mapped devices, e.g. control registers. Memory mapped using ioremap() has the following characteristics:

- Uncached - CPU-side caches are bypassed, and all reads and writes are handled directly by the device
- No speculative operations - the CPU may not issue a read or write to this memory, unless the instruction that does so has been reached in committed program flow.
- No reordering - The CPU may not reorder accesses to this memory mapping with respect to each other. On some

architectures, this relies on barriers in readl_relaxed()/writel_relaxed().
- No repetition - The CPU may not issue multiple reads or writes for a single program instruction.
- No write-combining - Each I/O operation results in one discrete read or write being issued to the device, and multiple writes are not combined into larger writes. This may or may not be enforced when using __raw I/O accessors or pointer dereferences.
- Non-executable - The CPU is not allowed to speculate instruction execution from this memory (it probably goes without saying, but you're also not allowed to jump into device memory).

On many platforms and buses (e.g. PCI), writes issued through ioremap() mappings are posted, which means that the CPU does not wait for the write to actually reach the target device before retiring the write instruction.

On many platforms, I/O accesses must be aligned with respect to the access size; failure to do so will result in an exception or unpredictable results.

ioremap_wc()

Maps I/O memory as normal memory with write combining. Unlike ioremap(),

- The CPU may speculatively issue reads from the device that the program didn't actually execute, and may choose to basically read whatever it wants.
- The CPU may reorder operations as long as the result is consistent from the program's point of view.
- The CPU may write to the same location multiple times, even when the program issued a single write.
- The CPU may combine several writes into a single larger write.

This mode is typically used for video framebuffers, where it can increase performance of writes. It can also be used for other blocks of memory in devices (e.g. buffers or shared memory), but care must be taken as accesses are not guaranteed to be ordered with respect to normal ioremap() MMIO register accesses without explicit barriers.

On a PCI bus, it is usually safe to use ioremap_wc() on MMIO areas marked as `IORESOURCE_PREFETCH`, but it may not be used on those without the flag. For on-chip devices, there is no corresponding flag, but a driver can use ioremap_wc() on a device that is known to be safe.

ioremap_wt()

Maps I/O memory as normal memory with write-through caching. Like ioremap_wc(), but also,

- The CPU may cache writes issued to and reads from the device, and serve reads from that cache.

This mode is sometimes used for video framebuffers, where drivers still expect writes to reach the device in a timely manner (and not be stuck in the CPU cache), but reads may be served from the cache for efficiency. However, it is rarely useful these days, as framebuffer drivers usually perform writes only, for which ioremap_wc() is more efficient (as it doesn't needlessly trash the cache). Most drivers should not use this.

ioremap_np()

Like ioremap(), but explicitly requests non-posted write semantics. On some architectures and buses, ioremap() mappings have posted write semantics, which means that writes can appear to "complete" from the point of view of the CPU before the written data actually arrives at the target device. Writes are still ordered with respect to other writes and reads from the same device, but due to the posted write semantics, this is not the case with respect to other devices. ioremap_np() explicitly requests non-posted semantics, which means that the write instruction will not appear to complete until the device has received (and to some platform-specific extent acknowledged) the written data.

This mapping mode primarily exists to cater for platforms with bus fabrics that require this particular mapping mode to work correctly. These platforms set the `IORESOURCE_MEM_NONPOSTED` flag for a resource that requires ioremap_np() semantics and portable drivers should use an abstraction that automatically selects it where appropriate (see the Higher-level ioremap abstractions section below).

The bare ioremap_np() is only available on some architectures; on others, it always returns NULL. Drivers should not normally use it, unless they are platform-specific or they derive benefit from non-posted writes where supported, and can fall back to ioremap() otherwise. The normal approach to ensure posted write completion is to do a dummy read after a write as explained in Accessing the device, which works with ioremap() on all platforms.

ioremap_np() should never be used for PCI drivers. PCI memory space writes are always posted, even on architectures that otherwise implement ioremap_np(). Using ioremap_np() for PCI BARs will at best result in posted write semantics, and at worst result in complete breakage.

Note that non-posted write semantics are orthogonal to CPU-side ordering guarantees. A CPU may still choose to issue other reads or writes before a non-posted write instruction retires. See the previous section on MMIO access functions for details on the CPU side of things.

ioremap_uc()

ioremap_uc() behaves like ioremap() except that on the x86 architecture without 'PAT' mode, it marks memory as uncached even when the MTRR has designated it as cacheable, see Documentation/x86/pat.rst.

Portable drivers should avoid the use of ioremap_uc().

ioremap_cache()

ioremap_cache() effectively maps I/O memory as normal RAM. CPU write-back caches can be used, and the CPU is free to treat the device as if it were a block of RAM. This should never be used for device memory which has side effects of any kind, or which does not return the data previously written on read.

It should also not be used for actual RAM, as the returned pointer is an `__iomem` token. memremap() can be used for mapping normal RAM that is outside of the linear kernel memory area to a regular pointer.

Portable drivers should avoid the use of ioremap_cache().

Architecture example

Here is how the above modes map to memory attribute settings on the ARM64 architecture:

API	Memory region type and cacheability
ioremap_np()	Device-nGnRnE
ioremap()	Device-nGnRE
ioremap_uc()	(not implemented)
ioremap_wc()	Normal-Non Cacheable
ioremap_wt()	(not implemented; fallback to ioremap)
ioremap_cache()	Normal-Write-Back Cacheable

Higher-level ioremap abstractions

Instead of using the above raw ioremap() modes, drivers are encouraged to use higher-level APIs. These APIs may implement platform-specific logic to automatically choose an appropriate ioremap mode on any given bus, allowing for a platform-agnostic driver to work on those platforms without any special cases. At the time of this writing, the following ioremap() wrappers have such logic:

devm_ioremap_resource()

> Can automatically select ioremap_np() over ioremap() according to platform requirements, if the `IORESOURCE_MEM_NONPOSTED` flag is set on the struct resource. Uses devres to automatically unmap the resource when the driver probe() function fails or a device in unbound from its driver.
>
> Documented in Documentation/driver-api/driver-model/devres.rst.

of_address_to_resource()

> Automatically sets the `IORESOURCE_MEM_NONPOSTED` flag for platforms that require non-posted writes for certain buses (see the nonposted-mmio and posted-mmio device tree properties).

of_iomap()

> Maps the resource described in a `reg` property in the device tree, doing all required translations. Automatically selects ioremap_np() according to platform requirements, as above.

pci_ioremap_bar(), pci_ioremap_wc_bar()

> Maps the resource described in a PCI base address without having to extract the physical address first.

pci_iomap(), pci_iomap_wc()

> Like pci_ioremap_bar()/pci_ioremap_bar(), but also works on I/O space when used together with ioread32()/iowrite32() and similar accessors

pcim_iomap()

> Like pci_iomap(), but uses devres to automatically unmap the resource when the driver probe() function fails or a device in unbound from its driver
>
> Documented in Documentation/driver-api/driver-model/devres.rst.

Not using these wrappers may make drivers unusable on certain platforms with stricter rules for mapping I/O memory.

Generalizing Access to System and I/O Memory

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\(linux-master) (Documentation) (driver-api)device-io.rst`, line 508)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/iosys-map.h
   :doc: overview
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\(linux-master) (Documentation) (driver-api)device-io.rst`, line 511)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: include/linux/iosys-map.h
   :internal:
```

Public Functions Provided

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\(linux-master) (Documentation) (driver-api)device-io.rst`, line 517)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: arch/x86/include/asm/io.h
   :internal:
```

System Message: ERROR/3 (`D:\onboarding-resources\sample-onboarding-resources\linux-master\Documentation\driver-api\(linux-master) (Documentation) (driver-api)device-io.rst`, line 520)

Unknown directive type "kernel-doc".

```
.. kernel-doc:: lib/pci_iomap.c
   :export:
```