How to access I/O mapped memory from within device drivers

Author: Linus

> **Warning**
>
> The virt_to_bus() and bus_to_virt() functions have been superseded by the functionality provided by the PCI DMA interface (see Documentation/core-api/dma-api-howto.rst). They continue to be documented below for historical purposes, but new code must not use them. --davidm 00/12/12

```
[ This is a mail message in response to a query on IO mapping, thus the
  strange format for a "document" ]
```

The AHA-1542 is a bus-master device, and your patch makes the driver give the controller the physical address of the buffers, which is correct on x86 (because all bus master devices see the physical memory mappings directly).

However, on many setups, there are actually **three** different ways of looking at memory addresses, and in this case we actually want the third, the so-called "bus address".

Essentially, the three ways of addressing memory are (this is "real memory", that is, normal RAM--see later about other details):

- CPU untranslated. This is the "physical" address. Physical address 0 is what the CPU sees when it drives zeroes on the memory bus.
- CPU translated address. This is the "virtual" address, and is completely internal to the CPU itself with the CPU doing the appropriate translations into "CPU untranslated".
- bus address. This is the address of memory as seen by OTHER devices, not the CPU. Now, in theory there could be many different bus addresses, with each device seeing memory in some device-specific way, but happily most hardware designers aren't actually actively trying to make things any more complex than necessary, so you can assume that all external hardware sees the memory the same way.

Now, on normal PCs the bus address is exactly the same as the physical address, and things are very simple indeed. However, they are that simple because the memory and the devices share the same address space, and that is not generally necessarily true on other PCI/ISA setups.

Now, just as an example, on the PReP (PowerPC Reference Platform), the CPU sees a memory map something like this (this is from memory):

```
0-2 GB          "real memory"
2 GB-3 GB       "system IO" (inb/out and similar accesses on x86)
3 GB-4 GB       "IO memory" (shared memory over the IO bus)
```

Now, that looks simple enough. However, when you look at the same thing from the viewpoint of the devices, you have the reverse, and the physical memory address 0 actually shows up as address 2 GB for any IO master.

So when the CPU wants any bus master to write to physical memory 0, it has to give the master address 0x80000000 as the memory address.

So, for example, depending on how the kernel is actually mapped on the PPC, you can end up with a setup like this:

```
physical address:      0
virtual address:       0xC0000000
bus address:           0x80000000
```

where all the addresses actually point to the same thing. It's just seen through different translations..

Similarly, on the Alpha, the normal translation is:

```
physical address:      0
virtual address:       0xfffffc0000000000
bus address:           0x40000000
```

(but there are also Alphas where the physical address and the bus address are the same).

Anyway, the way to look up all these translations, you do:

```
#include <asm/io.h>

phys_addr = virt_to_phys(virt_addr);
virt_addr = phys_to_virt(phys_addr);
 bus_addr = virt_to_bus(virt_addr);
virt_addr = bus_to_virt(bus_addr);
```

Now, when do you need these?

You want the **virtual** address when you are actually going to access that pointer from the kernel. So you can have something like this:

```
/*
 * this is the hardware "mailbox" we use to communicate with
 * the controller. The controller sees this directly.
 */
struct mailbox {
        __u32 status;
        __u32 bufstart;
        __u32 buflen;
        ..
} mbox;

        unsigned char * retbuffer;

        /* get the address from the controller */
        retbuffer = bus_to_virt(mbox.bufstart);
        switch (retbuffer[0]) {
                case STATUS_OK:
                        ...
```

on the other hand, you want the bus address when you have a buffer that you want to give to the controller:

```
/* ask the controller to read the sense status into "sense_buffer" */
mbox.bufstart = virt_to_bus(&sense_buffer);
mbox.buflen = sizeof(sense_buffer);
mbox.status = 0;
notify_controller(&mbox);
```

And you generally **never** want to use the physical address, because you can't use that from the CPU (the CPU only uses translated virtual addresses), and you can't use it from the bus master.

So why do we care about the physical address at all? We do need the physical address in some cases, it's just not very often in normal code. The physical address is needed if you use memory mappings, for example, because the "remap_pfn_range()" mm function wants the physical address of the memory to be remapped as measured in units of pages, a.k.a. the pfn (the memory management layer doesn't know about devices outside the CPU, so it shouldn't need to know about "bus addresses" etc).

> **Note**
>
> The above is only one part of the whole equation. The above only talks about "real memory", that is, CPU memory (RAM).

There is a completely different type of memory too, and that's the "shared memory" on the PCI or ISA bus. That's generally not RAM (although in the case of a video graphics card it can be normal DRAM that is just used for a frame buffer), but can be things like a packet buffer in a network card etc.

This memory is called "PCI memory" or "shared memory" or "IO memory" or whatever, and there is only one way to access it: the readb/writeb and related functions. You should never take the address of such memory, because there is really nothing you can do with such an address: it's not conceptually in the same memory space as "real memory" at all, so you cannot just dereference a pointer. (Sadly, on x86 it **is** in the same memory space, so on x86 it actually works to just deference a pointer, but it's not portable).

For such memory, you can do things like:

- reading:

  ```
  /*
   * read first 32 bits from ISA memory at 0xC0000, aka
   * C000:0000 in DOS terms
   */
  unsigned int signature = isa_readl(0xC0000);
  ```

- remapping and writing:

  ```
  /*
   * remap framebuffer PCI memory area at 0xFC000000,
   * size 1MB, so that we can access it: We can directly
   * access only the 640k-1MB area, so anything else
   * has to be remapped.
   */
  void __iomem *baseptr = ioremap(0xFC000000, 1024*1024);

  /* write a 'A' to the offset 10 of the area */
  writeb('A',baseptr+10);

  /* unmap when we unload the driver */
  iounmap(baseptr);
  ```

- copying and clearing:

```
/* get the 6-byte Ethernet address at ISA address E000:0040 */
memcpy_fromio(kernel_buffer, 0xE0040, 6);
/* write a packet to the driver */
memcpy_toio(0xE1000, skb->data, skb->len);
/* clear the frame buffer */
memset_io(0xA0000, 0, 0x10000);
```

OK, that just about covers the basics of accessing IO portably. Questions? Comments? You may think that all the above is overly complex, but one day you might find yourself with a 500 MHz Alpha in front of you, and then you'll be happy that your driver works ;)

Note that kernel versions 2.0.x (and earlier) mistakenly called the ioremap() function "vremap()". ioremap() is the proper name, but I didn't think straight when I wrote it originally. People who have to support both can do something like:

```
/* support old naming silliness */
#if LINUX_VERSION_CODE < 0x020100
#define ioremap vremap
#define iounmap vfree
#endif
```

at the top of their source files, and then they can use the right names even on 2.0.x systems.

And the above sounds worse than it really is. Most real drivers really don't do all that complex things (or rather: the complexity is not so much in the actual IO accesses as in error handling and timeouts etc). It's generally not hard to fix drivers, and in many cases the code actually looks better afterwards:

```
unsigned long signature = *(unsigned int *) 0xC0000;
        vs
unsigned long signature = readl(0xC0000);
```

I think the second version actually is more readable, no?