DMA with ISA and LPC devices

Pierre Ossman <drzeus@drzeus.cx>

This document describes how to do DMA transfers using the old ISA DMA controller. Even though ISA is more or less dead today the LPC bus uses the same DMA system so it will be around for quite some time.

Part I - Headers and dependencies

To do ISA style DMA you need to include two headers:

```
#include <linux/dma-mapping.h>
#include <asm/dma.h>
```

The first is the generic DMA API used to convert virtual addresses to physical addresses (see Documentation/DMA-API.txt for details).

The second contains the routines specific to ISA DMA transfers. Since this is not present on all platforms make sure you construct your Kconfig to be dependent on ISA_DMA_API (not ISA) so that nobody tries to build your driver on unsupported platforms.

Part II - Buffer allocation

The ISA DMA controller has some very strict requirements on which memory it can access so extra care must be taken when allocating buffers.

(You usually need a special buffer for DMA transfers instead of transferring directly to and from your normal data structures.)

The DMA-able address space is the lowest 16 MB of _physical_ memory. Also the transfer block may not cross page boundaries (which are 64 or 128 KiB depending on which channel you use).

In order to allocate a piece of memory that satisfies all these requirements you pass the flag GFP_DMA to kmalloc.

Unfortunately the memory available for ISA DMA is scarce so unless you allocate the memory during boot-up it's a good idea to also pass __GFP_REPEAT and __GFP_NOWARN to make the allocater try a bit harder.

(This scarcity also means that you should allocate the buffer as early as possible and not release it until the driver is unloaded.)

Part III - Address translation

To translate the virtual address to a physical use the normal DMA API. Do _not_ use isa_virt_to_phys() even though it does the same thing. The reason for this is that the function isa_virt_to_phys() will require a Kconfig dependency to ISA, not just ISA_DMA_API which is really all you need. Remember that even though the DMA controller has its origins in ISA it is used elsewhere.

Note: x86_64 had a broken DMA API when it came to ISA but has since been fixed. If your arch has problems then fix the DMA API instead of reverting to the ISA functions.

Part IV - Channels

A normal ISA DMA controller has 8 channels. The lower four are for 8-bit transfers and the upper four are for 16-bit transfers.

(Actually the DMA controller is really two separate controllers where channel 4 is used to give DMA access for the second controller (0-3). This means that of the four 16-bits channels only three are usable.)

You allocate these in a similar fashion as all basic resources:

```
extern int request_dma(unsigned int dmanr, const char * device_id);
extern void free_dma(unsigned int dmanr);
```

The ability to use 16-bit or 8-bit transfers is _not_ up to you as a driver author but depends on what the hardware supports. Check your specs or test different channels.

Part V - Transfer data

Now for the good stuff, the actual DMA transfer. :)

Before you use any ISA DMA routines you need to claim the DMA lock using claim_dma_lock(). The reason is that some DMA operations are not atomic so only one driver may fiddle with the registers at a time.

The first time you use the DMA controller you should call clear_dma_ff(). This clears an internal register in the DMA controller that is used for the non-atomic operations. As long as you (and everyone else) uses the locking functions then you only need to reset this once.

Next, you tell the controller in which direction you intend to do the transfer using set_dma_mode(). Currently you have the options DMA_MODE_READ and DMA_MODE_WRITE.

Set the address from where the transfer should start (this needs to be 16-bit aligned for 16-bit transfers) and how many bytes to transfer. Note that it's _bytes_. The DMA routines will do all the required translation to values that the DMA controller understands.

The final step is enabling the DMA channel and releasing the DMA lock.

Once the DMA transfer is finished (or timed out) you should disable the channel again. You should also check get_dma_residue() to make sure that all data has been transferred.

Example:

```
int flags, residue;

flags = claim_dma_lock();

clear_dma_ff();

set_dma_mode(channel, DMA_MODE_WRITE);
set_dma_addr(channel, phys_addr);
set_dma_count(channel, num_bytes);

dma_enable(channel);

release_dma_lock(flags);

while (!device_done());

flags = claim_dma_lock();

dma_disable(channel);

residue = dma_get_residue(channel);
if (residue != 0)
        printk(KERN_ERR "driver: Incomplete DMA transfer!"
                " %d bytes left!\n", residue);

release_dma_lock(flags);
```

Part VI - Suspend/resume

It is the driver's responsibility to make sure that the machine isn't suspended while a DMA transfer is in progress. Also, all DMA settings are lost when the system suspends so if your driver relies on the DMA controller being in a certain state then you have to restore these registers upon resume.