Dynamic DMA mapping Guide

David S. Miller <davem@redhat.com>
Richard Henderson <rth@cygnus.com>
Jakub Jelinek <jakub@redhat.com>

This is a guide to device driver writers on how to use the DMA API with example pseudo-code. For a concise description of the API, see DMA-API.txt.

Most of the 64bit platforms have special hardware that translates bus addresses (DMA addresses) into physical addresses. This is similar to how page tables and/or a TLB translates virtual addresses to physical addresses on a CPU. This is needed so that e.g. PCI devices can access with a Single Address Cycle (32bit DMA address) any page in the 64bit physical address space. Previously in Linux those 64bit platforms had to set artificial limits on the maximum RAM size in the system, so that the virt_to_bus() static scheme works (the DMA address translation tables were simply filled on bootup to map each bus address to the physical page __pa(bus_to_virt())).

So that Linux can use the dynamic DMA mapping, it needs some help from the drivers, namely it has to take into account that DMA addresses should be mapped only for the time they are actually used and unmapped after the DMA transfer.

The following API will work of course even on platforms where no such hardware exists.

Note that the DMA API works with any bus independent of the underlying microprocessor architecture. You should use the DMA API rather than the bus specific DMA API (e.g. pci_dma_*).

First of all, you should make sure

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

is in your driver. This file will obtain for you the definition of the dma_addr_t (which can hold any valid DMA address for the platform) type which should be used everywhere you hold a DMA (bus) address returned from the DMA mapping functions.

What memory is DMA'able?

The first piece of information you must know is what kernel memory can be used with the DMA mapping facilities. There has been an unwritten set of rules regarding this, and this text is an attempt to finally write them down.

If you acquired your memory via the page allocator (i.e. __get_free_page*()) or the generic memory allocators (i.e. kmalloc() or kmem_cache_alloc()) then you may DMA to/from that memory using the addresses returned from those routines.

This means specifically that you may _not_ use the memory/addresses returned from vmalloc() for DMA. It is possible to DMA to the _underlying_ memory mapped into a vmalloc() area, but this requires walking page tables to get the physical addresses, and then translating each of those pages back to a kernel address using something like __va(). [EDIT: Update this when we integrate Gerd Knorr's generic code which does this.]

This rule also means that you may use neither kernel image addresses (items in data/text/bss segments), nor module image addresses, nor stack addresses for DMA. These could all be mapped somewhere entirely different than the rest of physical memory. Even if those classes of memory could physically work with DMA, you'd need to ensure the I/O buffers were cacheline-aligned. Without that, you'd see cacheline sharing problems (data corruption) on CPUs with DMA-incoherent caches. (The CPU could write to one word, DMA would write to a different one in the same cache line, and one of them could be overwritten.)

Also, this means that you cannot take the return of a kmap() call and DMA to/from that. This is similar to vmalloc().

What about block I/O and networking buffers? The block I/O and networking subsystems make sure that the buffers they use are valid for you to DMA from/to.

DMA addressing limitations

Does your device have any DMA addressing limitations? For example, is your device only capable of driving the low order 24-bits of address? If so, you need to inform the kernel of this fact.

By default, the kernel assumes that your device can address the full 32-bits. For a 64-bit capable device, this needs to be increased. And for a device with limitations, as discussed in the previous paragraph, it needs to be decreased.

Special note about PCI: PCI-X specification requires PCI-X devices to support 64-bit addressing (DAC) for all transactions. And at least one platform (SGI SN2) requires 64-bit consistent allocations to operate correctly when the IO bus is in PCI-X mode.

For correct operation, you must interrogate the kernel in your device probe routine to see if the DMA controller on the machine can properly support the DMA addressing limitation your device has. It is good style to do this even if your device holds the default setting, because this shows that you did think about these issues wrt. your device.

The query is performed via a call to dma_set_mask():

```
	int dma_set_mask(struct device *dev, u64 mask);
```

The query for consistent allocations is performed via a call to dma_set_coherent_mask():

```
	int dma_set_coherent_mask(struct device *dev, u64 mask);
```

Here, dev is a pointer to the device struct of your device, and mask is a bit mask describing which bits of an address your device supports. It returns zero if your card can perform DMA properly on the machine given the address mask you provided. In general, the device struct of your device is embedded in the bus specific device struct of your device. For example, a pointer to the device struct of your PCI device is pdev->dev (pdev is a pointer to the PCI device struct of your device).

If it returns non-zero, your device cannot perform DMA properly on this platform, and attempting to do so will result in undefined behavior. You must either use a different mask, or not use DMA.

This means that in the failure case, you have three options:

1) Use another DMA mask, if possible (see below).
2) Use some non-DMA mode for data transfer, if possible.
3) Ignore this device and do not initialize it.

It is recommended that your driver print a kernel KERN_WARNING message when you end up performing either #2 or #3. In this manner, if a user of your driver reports that performance is bad or that the device is not even detected, you can ask them for the kernel messages to find out exactly why.

The standard 32-bit addressing device would do something like this:

```
	if (dma_set_mask(dev, DMA_BIT_MASK(32))) {
		printk(KERN_WARNING
		       "mydev: No suitable DMA available.\n");
		goto ignore_this_device;
	}
```

Another common scenario is a 64-bit capable device. The approach here is to try for 64-bit addressing, but back down to a 32-bit mask that should not fail. The kernel may fail the 64-bit mask not because the platform is not capable of 64-bit addressing. Rather, it may fail in this case simply because 32-bit addressing is done more efficiently than 64-bit addressing. For example, Sparc64 PCI SAC addressing is more efficient than DAC addressing.

Here is how you would handle a 64-bit capable device which can drive all 64-bits when accessing streaming DMA:

```
	int using_dac;

	if (!dma_set_mask(dev, DMA_BIT_MASK(64))) {
		using_dac = 1;
	} else if (!dma_set_mask(dev, DMA_BIT_MASK(32))) {
		using_dac = 0;
	} else {
		printk(KERN_WARNING
		       "mydev: No suitable DMA available.\n");
		goto ignore_this_device;
	}
```

If a card is capable of using 64-bit consistent allocations as well, the case would look like this:

```
	int using_dac, consistent_using_dac;

	if (!dma_set_mask(dev, DMA_BIT_MASK(64))) {
		using_dac = 1;
		consistent_using_dac = 1;
		dma_set_coherent_mask(dev, DMA_BIT_MASK(64));
	} else if (!dma_set_mask(dev, DMA_BIT_MASK(32))) {
		using_dac = 0;
		consistent_using_dac = 0;
		dma_set_coherent_mask(dev, DMA_BIT_MASK(32));
	} else {
		printk(KERN_WARNING
		       "mydev: No suitable DMA available.\n");
		goto ignore_this_device;
	}
```

dma_set_coherent_mask() will always be able to set the same or a smaller mask as dma_set_mask(). However for the rare case that a device driver only uses consistent allocations, one would have to check the return value from dma_set_coherent_mask().

Finally, if your device can only drive the low 24-bits of address you might do something like:

```
	if (dma_set_mask(dev, DMA_BIT_MASK(24))) {
		printk(KERN_WARNING
		       "mydev: 24-bit DMA addressing not available.\n");
		goto ignore_this_device;
	}
```

When dma_set_mask() is successful, and returns zero, the kernel saves away this mask you have provided. The kernel will use this information later when you make DMA mappings.

There is a case which we are aware of at this time, which is worth mentioning in this documentation. If your device supports multiple functions (for example a sound card provides playback and record functions) and the various different functions have _different_ DMA addressing limitations, you may wish to probe each mask and only provide the functionality which the machine can handle. It is important that the last call to dma_set_mask() be for the most specific mask.

Here is pseudo-code showing how this might be done:

```
	#define PLAYBACK_ADDRESS_BITS	DMA_BIT_MASK(32)
	#define RECORD_ADDRESS_BITS	DMA_BIT_MASK(24)

	struct my_sound_card *card;
	struct device *dev;

	...
	if (!dma_set_mask(dev, PLAYBACK_ADDRESS_BITS)) {
```

```
                card->playback_enabled = 1;
        } else {
                card->playback_enabled = 0;
                printk(KERN_WARNING "%s: Playback disabled due to DMA
limitations.\n",
                       card->name);
        }
        if (!dma_set_mask(dev, RECORD_ADDRESS_BITS)) {
                card->record_enabled = 1;
        } else {
                card->record_enabled = 0;
                printk(KERN_WARNING "%s: Record disabled due to DMA
limitations.\n",
                       card->name);
        }
```

A sound card was used as an example here because this genre of PCI devices seems to be littered with ISA chips given a PCI front end, and thus retaining the 16MB DMA addressing limitations of ISA.

Types of DMA mappings

There are two types of DMA mappings:

- Consistent DMA mappings which are usually mapped at driver initialization, unmapped at the end and for which the hardware should guarantee that the device and the CPU can access the data in parallel and will see updates made by each other without any explicit software flushing.

  Think of "consistent" as "synchronous" or "coherent".

  The current default is to return consistent memory in the low 32 bits of the bus space. However, for future compatibility you should set the consistent mask even if this default is fine for your driver.

  Good examples of what to use consistent mappings for are:

    - Network card DMA ring descriptors.
    - SCSI adapter mailbox command data structures.
    - Device firmware microcode executed out of main memory.

  The invariant these examples all require is that any CPU store to memory is immediately visible to the device, and vice versa. Consistent mappings guarantee this.

  IMPORTANT: Consistent DMA memory does not preclude the usage of proper memory barriers. The CPU may reorder stores to consistent memory just as it may normal memory. Example: if it is important for the device to see the first word of a descriptor updated before the second, you must do something like:

```
                desc->word0 = address;
```

```
                  wmb();
                  desc->word1 = DESC_VALID;
```

in order to get correct behavior on all platforms.

Also, on some platforms your driver may need to flush CPU write buffers in much the same way as it needs to flush write buffers found in PCI bridges (such as by reading a register's value after writing it).

- Streaming DMA mappings which are usually mapped for one DMA transfer, unmapped right after it (unless you use dma_sync_* below) and for which hardware can optimize for sequential accesses.

  This of "streaming" as "asynchronous" or "outside the coherency domain".

  Good examples of what to use streaming mappings for are:

  - Networking buffers transmitted/received by a device.
  - Filesystem buffers written/read by a SCSI device.

  The interfaces for using this type of mapping were designed in such a way that an implementation can make whatever performance optimizations the hardware allows. To this end, when using such mappings you must be explicit about what you want to happen.

Neither type of DMA mapping has alignment restrictions that come from the underlying bus, although some devices may have such restrictions. Also, systems with caches that aren't DMA-coherent will work better when the underlying buffers don't share cache lines with other data.

Using Consistent DMA mappings.

To allocate and map large (PAGE_SIZE or so) consistent DMA regions, you should do:

```
	dma_addr_t dma_handle;

	cpu_addr = dma_alloc_coherent(dev, size, &dma_handle, gfp);
```

where device is a struct device *. This may be called in interrupt context with the GFP_ATOMIC flag.

Size is the length of the region you want to allocate, in bytes.

This routine will allocate RAM for that region, so it acts similarly to __get_free_pages (but takes size instead of a page order). If your driver needs regions sized smaller than a page, you may prefer using the dma_pool interface, described below.

The consistent DMA mapping interfaces, for non-NULL dev, will by default return a DMA address which is 32-bit addressable. Even if the device indicates (via DMA mask) that it may address the upper 32-bits, consistent allocation will only return > 32-bit addresses for DMA if the consistent DMA mask has been explicitly changed via dma_set_coherent_mask(). This is true of the dma_pool interface as well.

dma_alloc_coherent returns two values: the virtual address which you can use to access it from the CPU and dma_handle which you pass to the card.

The cpu return address and the DMA bus master address are both guaranteed to be aligned to the smallest PAGE_SIZE order which is greater than or equal to the requested size. This invariant exists (for example) to guarantee that if you allocate a chunk which is smaller than or equal to 64 kilobytes, the extent of the buffer you receive will not cross a 64K boundary.

To unmap and free such a DMA region, you call:

```
	dma_free_coherent(dev, size, cpu_addr, dma_handle);
```

where dev, size are the same as in the above call and cpu_addr and dma_handle are the values dma_alloc_coherent returned to you. This function may not be called in interrupt context.

If your driver needs lots of smaller memory regions, you can write custom code to subdivide pages returned by dma_alloc_coherent, or you can use the dma_pool API to do that. A dma_pool is like a kmem_cache, but it uses dma_alloc_coherent not __get_free_pages. Also, it understands common hardware constraints for alignment, like queue heads needing to be aligned on N byte boundaries.

Create a dma_pool like this:

```
	struct dma_pool *pool;

	pool = dma_pool_create(name, dev, size, align, alloc);
```

The "name" is for diagnostics (like a kmem_cache name); dev and size are as above. The device's hardware alignment requirement for this type of data is "align" (which is expressed in bytes, and must be a power of two). If your device has no boundary crossing restrictions, pass 0 for alloc; passing 4096 says memory allocated from this pool must not cross 4KByte boundaries (but at that time it may be better to go for dma_alloc_coherent directly instead).

Allocate memory from a dma pool like this:

```
	cpu_addr = dma_pool_alloc(pool, flags, &dma_handle);
```

flags are SLAB_KERNEL if blocking is permitted (not in_interrupt nor holding SMP locks), SLAB_ATOMIC otherwise. Like dma_alloc_coherent, this returns two values, cpu_addr and dma_handle.

Free memory that was allocated from a dma_pool like this:

```
	dma_pool_free(pool, cpu_addr, dma_handle);
```

where pool is what you passed to dma_pool_alloc, and cpu_addr and dma_handle are the values dma_pool_alloc returned. This function may be called in interrupt context.

Destroy a dma_pool by calling:

```
        dma_pool_destroy(pool);
```

Make sure you've called dma_pool_free for all memory allocated from a pool before you destroy the pool. This function may not be called in interrupt context.

DMA Direction

The interfaces described in subsequent portions of this document take a DMA direction argument, which is an integer and takes on one of the following values:

```
 DMA_BIDIRECTIONAL
 DMA_TO_DEVICE
 DMA_FROM_DEVICE
 DMA_NONE
```

One should provide the exact DMA direction if you know it.

DMA_TO_DEVICE means "from main memory to the device"
DMA_FROM_DEVICE means "from the device to main memory"
It is the direction in which the data moves during the DMA transfer.

You are _strongly_ encouraged to specify this as precisely as you possibly can.

If you absolutely cannot know the direction of the DMA transfer, specify DMA_BIDIRECTIONAL. It means that the DMA can go in either direction. The platform guarantees that you may legally specify this, and that it will work, but this may be at the cost of performance for example.

The value DMA_NONE is to be used for debugging. One can hold this in a data structure before you come to know the precise direction, and this will help catch cases where your direction tracking logic has failed to set things up properly.

Another advantage of specifying this value precisely (outside of potential platform-specific optimizations of such) is for debugging. Some platforms actually have a write permission boolean which DMA mappings can be marked with, much like page protections in the user program address space. Such platforms can and do report errors in the kernel logs when the DMA controller hardware detects violation of the permission setting.

Only streaming mappings specify a direction, consistent mappings implicitly have a direction attribute setting of DMA_BIDIRECTIONAL.

The SCSI subsystem tells you the direction to use in the 'sc_data_direction' member of the SCSI command your driver is working on.

For Networking drivers, it's a rather simple affair. For transmit packets, map/unmap them with the DMA_TO_DEVICE direction specifier. For receive packets, just the opposite, map/unmap them with the DMA_FROM_DEVICE direction specifier.

Using Streaming DMA mappings

The streaming DMA mapping routines can be called from interrupt context. There are two versions of each map/unmap, one which will map/unmap a single memory region, and one which will map/unmap a scatterlist.

To map a single region, you do:

```
	struct device *dev = &my_dev->dev;
	dma_addr_t dma_handle;
	void *addr = buffer->ptr;
	size_t size = buffer->len;

	dma_handle = dma_map_single(dev, addr, size, direction);
```

and to unmap it:

```
	dma_unmap_single(dev, dma_handle, size, direction);
```

You should call dma_unmap_single when the DMA activity is finished, e.g. from the interrupt which told you that the DMA transfer is done.

Using cpu pointers like this for single mappings has a disadvantage, you cannot reference HIGHMEM memory in this way. Thus, there is a map/unmap interface pair akin to dma_{map,unmap}_single. These interfaces deal with page/offset pairs instead of cpu pointers. Specifically:

```
	struct device *dev = &my_dev->dev;
	dma_addr_t dma_handle;
	struct page *page = buffer->page;
	unsigned long offset = buffer->offset;
	size_t size = buffer->len;

	dma_handle = dma_map_page(dev, page, offset, size, direction);

	...

	dma_unmap_page(dev, dma_handle, size, direction);
```

Here, "offset" means byte offset within the given page.

With scatterlists, you map a region gathered from several regions by:

```
	int i, count = dma_map_sg(dev, sglist, nents, direction);
	struct scatterlist *sg;
```

```
        for_each_sg(sglist, sg, count, i) {
                hw_address[i] = sg_dma_address(sg);
                hw_len[i] = sg_dma_len(sg);
        }
```

where nents is the number of entries in the sglist.

The implementation is free to merge several consecutive sglist entries into one (e.g. if DMA mapping is done with PAGE_SIZE granularity, any consecutive sglist entries can be merged into one provided the first one ends and the second one starts on a page boundary - in fact this is a huge advantage for cards which either cannot do scatter-gather or have very limited number of scatter-gather entries) and returns the actual number of sg entries it mapped them to. On failure 0 is returned.

Then you should loop count times (note: this can be less than nents times) and use sg_dma_address() and sg_dma_len() macros where you previously accessed sg->address and sg->length as shown above.

To unmap a scatterlist, just call:

```
        dma_unmap_sg(dev, sglist, nents, direction);
```

Again, make sure DMA activity has already finished.

PLEASE NOTE: The 'nents' argument to the dma_unmap_sg call must be the _same_ one you passed into the dma_map_sg call, it should _NOT_ be the 'count' value _returned_ from the dma_map_sg call.

Every dma_map_{single,sg} call should have its dma_unmap_{single,sg} counterpart, because the bus address space is a shared resource (although in some ports the mapping is per each BUS so less devices contend for the same bus address space) and you could render the machine unusable by eating all bus addresses.

If you need to use the same streaming DMA region multiple times and touch the data in between the DMA transfers, the buffer needs to be synced properly in order for the cpu and device to see the most uptodate and correct copy of the DMA buffer.

So, firstly, just map it with dma_map_{single,sg}, and after each DMA transfer call either:

```
        dma_sync_single_for_cpu(dev, dma_handle, size, direction);
```

or:

```
        dma_sync_sg_for_cpu(dev, sglist, nents, direction);
```

as appropriate.

Then, if you wish to let the device get at the DMA area again, finish accessing the data with the cpu, and then before actually giving the buffer to the hardware call either:

```
        dma_sync_single_for_device(dev, dma_handle, size, direction);
```

or:

```
        dma_sync_sg_for_device(dev, sglist, nents, direction);
```

as appropriate.

After the last DMA transfer call one of the DMA unmap routines
dma_unmap_{single,sg}. If you don't touch the data from the first dma_map_*
call till dma_unmap_*, then you don't have to call the dma_sync_*
routines at all.

Here is pseudo code which shows a situation in which you would need
to use the dma_sync_*() interfaces.

```
        my_card_setup_receive_buffer(struct my_card *cp, char *buffer, int len)
        {
                dma_addr_t mapping;

                mapping = dma_map_single(cp->dev, buffer, len, DMA_FROM_DEVICE);

                cp->rx_buf = buffer;
                cp->rx_len = len;
                cp->rx_dma = mapping;

                give_rx_buf_to_card(cp);
        }

        ...

        my_card_interrupt_handler(int irq, void *devid, struct pt_regs *regs)
        {
                struct my_card *cp = devid;

                ...
                if (read_card_status(cp) == RX_BUF_TRANSFERRED) {
                        struct my_card_header *hp;

                        /* Examine the header to see if we wish
                         * to accept the data.  But synchronize
                         * the DMA transfer with the CPU first
                         * so that we see updated contents.
                         */
                        dma_sync_single_for_cpu(&cp->dev, cp->rx_dma,
                                                cp->rx_len,
                                                DMA_FROM_DEVICE);

                        /* Now it is safe to examine the buffer. */
                        hp = (struct my_card_header *) cp->rx_buf;
                        if (header_is_ok(hp)) {
                                dma_unmap_single(&cp->dev, cp->rx_dma,
cp->rx_len,
                                                 DMA_FROM_DEVICE);
                                pass_to_upper_layers(cp->rx_buf);
```

```
                                make_and_setup_new_rx_buf(cp);
                        } else {
                                /* Just sync the buffer and give it back
                                 * to the card.
                                 */
                                dma_sync_single_for_device(&cp->dev,
                                                           cp->rx_dma,
                                                           cp->rx_len,
                                                           DMA_FROM_DEVICE);
                                give_rx_buf_to_card(cp);
                        }
                }
        }
```

Drivers converted fully to this interface should not use virt_to_bus any longer, nor should they use bus_to_virt. Some drivers have to be changed a little bit, because there is no longer an equivalent to bus_to_virt in the dynamic DMA mapping scheme - you have to always store the DMA addresses returned by the dma_alloc_coherent, dma_pool_alloc, and dma_map_single calls (dma_map_sg stores them in the scatterlist itself if the platform supports dynamic DMA mapping in hardware) in your driver structures and/or in the card registers.

All drivers should be using these interfaces with no exceptions. It is planned to completely remove virt_to_bus() and bus_to_virt() as they are entirely deprecated. Some ports already do not provide these as it is impossible to correctly support them.

Handling Errors

DMA address space is limited on some architectures and an allocation failure can be determined by:

- checking if dma_alloc_coherent returns NULL or dma_map_sg returns 0

- checking the returned dma_addr_t of dma_map_single and dma_map_page
  by using dma_mapping_error():

```
        dma_addr_t dma_handle;

        dma_handle = dma_map_single(dev, addr, size, direction);
        if (dma_mapping_error(dev, dma_handle)) {
                /*
                 * reduce current DMA mapping usage,
                 * delay and try again later or
                 * reset driver.
                 */
        }
```

Networking drivers must call dev_kfree_skb to free the socket buffer and return NETDEV_TX_OK if the DMA mapping fails on the transmit hook (ndo_start_xmit). This means that the socket buffer is just dropped in the failure case.

SCSI drivers must return SCSI_MLQUEUE_HOST_BUSY if the DMA mapping fails in the queuecommand hook. This means that the SCSI subsystem passes the command to the driver again later.

Optimizing Unmap State Space Consumption

On many platforms, dma_unmap_{single,page}() is simply a nop.
Therefore, keeping track of the mapping address and length is a waste
of space. Instead of filling your drivers up with ifdefs and the like
to "work around" this (which would defeat the whole purpose of a
portable API) the following facilities are provided.

Actually, instead of describing the macros one by one, we'll
transform some example code.

1) Use DEFINE_DMA_UNMAP_{ADDR,LEN} in state saving structures.
   Example, before:

```
	struct ring_state {
		struct sk_buff *skb;
		dma_addr_t mapping;
		__u32 len;
	};
```

   after:

```
	struct ring_state {
		struct sk_buff *skb;
		DEFINE_DMA_UNMAP_ADDR(mapping);
		DEFINE_DMA_UNMAP_LEN(len);
	};
```

2) Use dma_unmap_{addr,len}_set to set these values.
   Example, before:

```
	ringp->mapping = FOO;
	ringp->len = BAR;
```

   after:

```
	dma_unmap_addr_set(ringp, mapping, FOO);
	dma_unmap_len_set(ringp, len, BAR);
```

3) Use dma_unmap_{addr,len} to access these values.
   Example, before:

```
	dma_unmap_single(dev, ringp->mapping, ringp->len,
			 DMA_FROM_DEVICE);
```

   after:

```
	dma_unmap_single(dev,
			 dma_unmap_addr(ringp, mapping),
			 dma_unmap_len(ringp, len),
			 DMA_FROM_DEVICE);
```

It really should be self-explanatory. We treat the ADDR and LEN
separately, because it is possible for an implementation to only

need the address in order to perform the unmap operation.

Platform Issues

If you are just writing drivers for Linux and do not maintain
an architecture port for the kernel, you can safely skip down
to "Closing".

1) Struct scatterlist requirements.

   Don't invent the architecture specific struct scatterlist; just use
   <asm-generic/scatterlist.h>. You need to enable
   CONFIG_NEED_SG_DMA_LENGTH if the architecture supports IOMMUs
   (including software IOMMU).

2) ARCH_KMALLOC_MINALIGN

   Architectures must ensure that kmalloc'ed buffer is
   DMA-safe. Drivers and subsystems depend on it. If an architecture
   isn't fully DMA-coherent (i.e. hardware doesn't ensure that data in
   the CPU cache is identical to data in main memory),
   ARCH_KMALLOC_MINALIGN must be set so that the memory allocator
   makes sure that kmalloc'ed buffer doesn't share a cache line with
   the others. See arch/arm/include/asm/cache.h as an example.

   Note that ARCH_KMALLOC_MINALIGN is about DMA memory alignment
   constraints. You don't need to worry about the architecture data
   alignment constraints (e.g. the alignment constraints about 64-bit
   objects).

Closing

This document, and the API itself, would not be in its current
form without the feedback and suggestions from numerous individuals.
We would like to specifically mention, in no particular order, the
following people:

Russell King <rmk@arm.linux.org.uk>
Leo Dagum <dagum@barrel.engr.sgi.com>
Ralf Baechle <ralf@oss.sgi.com>
Grant Grundler <grundler@cup.hp.com>
Jay Estabrook <Jay.Estabrook@compaq.com>
Thomas Sailer <sailer@ife.ee.ethz.ch>
Andrea Arcangeli <andrea@suse.de>
Jens Axboe <jens.axboe@oracle.com>
David Mosberger-Tang <davidm@hpl.hp.com>