



Linux DMA

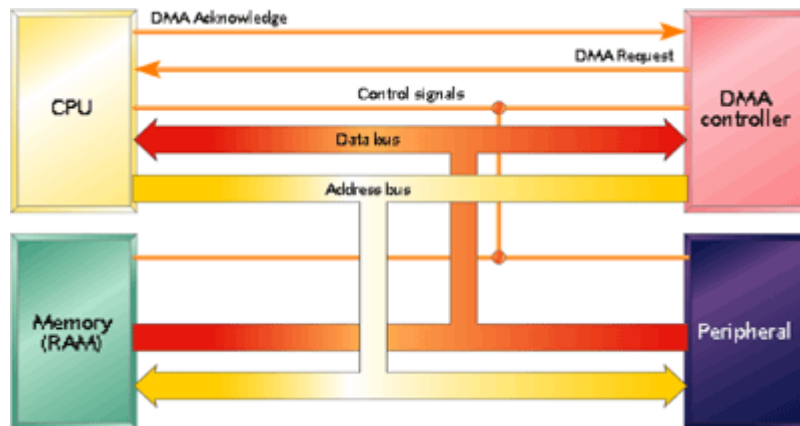
Table of Contents

DMA.....	1
DMA Transfer Overview.....	2
Allocating the DMA Buffer.....	3
The Generic DMA Layer.....	4
DMA Buffers.....	5
DMA Mappings.....	7
Types of DMA mappings.....	8
Some Details on Using Consistent and Streaming DMA APIs.....	10
1. Using Consistent (synchronous / cache coherent) DMA mappings.....	10
DMA Direction.....	11
2. Using Streaming (asynchronous / non-coherent) DMA mappings.....	12
Practical Usage of DMA in device drivers.....	19
1. Common case: how exactly to perform DMA from device directly to a userspace buffer	19
2. Block Drivers and DMA [Source: <i>Essential Linux Device Drivers</i> , Venkateshwaran]...22	22
- Efficient data transfer through zero copy www.ibm.com/developerworks/linux/library/j-zerocopy/	28
- Zero Copy I: User-Mode Perspective www.linuxjournal.com/article/6345	28
CMA.....	28
So how does it work?.....	28
CMA on ARM.....	30

[Source - LDD3](#)

DMA – Direct Memory Access - is the hardware mechanism that allows peripheral components to transfer their I/O data directly to and from main memory without the need to involve the system processor. Use of this mechanism **can greatly increase throughput** to and from a device, because a great deal of computational overhead is eliminated.

...



A DMA controller shares the processor's memory bus [\[Source\]](#)

DMA Transfer Overview

Data transfer can be triggered in two ways: **either the software (app) asks** for data (via a function such as read) **or the hardware asynchronously pushes** data to the system.

In the first case – the **synchronous** one - the steps involved can be summarized as follows:

1. When a process calls read, the driver method allocates a DMA buffer and instructs the hardware to transfer its data into that buffer. The process is put to sleep.
2. The hardware writes data to the DMA buffer and raises an interrupt when it's done.
3. The interrupt handler gets the input data, acknowledges the interrupt, and awakens the process, which is now able to read data.

The second case comes about when DMA is used **asynchronously**. This happens, for example, with data acquisition devices that go on pushing data even if nobody is reading them. In this case, the driver should maintain a buffer so that a subsequent read call will return all the accumulated data to user space. The steps involved in this kind of transfer are slightly different:

1. The hardware raises an interrupt to announce that new data has arrived.
2. The interrupt handler allocates a buffer and tells the hardware where to transfer its data.
3. The peripheral device writes the data to the buffer and raises another interrupt when it's done.
4. The handler dispatches the new data, wakes any relevant process, and takes care of housekeeping.

A variant of the asynchronous approach is often seen with network cards. These cards often expect to see a circular buffer (often called a DMA ring buffer) established in memory shared with the

processor; each incoming packet is placed in the next available buffer in the ring, and an interrupt is signaled. The driver then passes the network packets to the rest of the kernel and places a new DMA buffer in the ring.

The processing steps in all of these cases **emphasize that efficient DMA handling relies on interrupt reporting**. While it is possible to implement DMA with a **polling driver**, it wouldn't make sense, because a polling driver would waste the performance benefits that DMA offers over the easier processor-driven I/O. *

* There are, of course, **exceptions** to everything; see the section "Receive Interrupt Mitigation" in Chapter 17 for a demonstration of how high-performance network drivers are best implemented using polling << the so-called **NAPI** technique. >>

...

Allocating the DMA Buffer

The main issue that arises with DMA buffers is that, when they are bigger than one page, they **must occupy contiguous pages in physical memory** because the device transfers data using the ISA or PCI system bus, both of which carry physical addresses. ... Some architectures can also use virtual addresses on the PCI bus << those that boast an IOMMU >>, but a portable driver cannot count on that capability.

Although DMA buffers can be allocated either at system boot or at runtime, modules can allocate their buffers only at runtime. Driver writers **must take care to allocate the right kind of memory** when it is used for DMA operations; not all memory zones are suitable. In particular, high memory << *ZONE_HIGHMEM* – the 'zone' that's typically setup on IA-32 systems that have more RAM than kernel virtual address space; the 'unmapped' physical memory (above 896 MB) thus gets mapped into zone highmem >> may not work for DMA on some systems and with some devices — the peripherals simply cannot work with addresses that high.

Most devices on modern buses **can handle 32-bit** addresses, meaning that normal memory allocations work just fine for them. Some PCI devices, however, fail to implement the full PCI standard and cannot work with 32-bit addresses. And **ISA** devices, of course, are limited to **24-bit** addresses only << $2^{24} = 16 \text{ MB}$; *effectively giving them a memory range of 0-16 MB only; hence the requirement for ZONE_DMA spanning 0 – 16 MB on the PC architecture of course* >>.

For devices with this kind of limitation, memory should be allocated **from the DMA zone** by adding the GFP_DMA flag to the *kmalloc* or *get_free_pages* call. When this flag is present, only memory that can be addressed with 24 bits is allocated. Alternatively, you can use the *generic DMA layer* to allocate buffers that work around your device's limitations << *recommended* >>.

[...]

The Generic DMA Layer

DMA operations, in the end, come down to allocating a buffer and passing bus addresses to your device. However, the task of **writing portable drivers that perform DMA safely and correctly** on all architectures is harder than one might think. Different systems have different ideas of how cache coherency should work; if you do not handle this issue correctly, your driver may corrupt memory.

Some systems have complicated bus hardware that can make the DMA task easier—or harder. And not all systems can perform DMA out of all parts of memory. Fortunately, **the kernel provides a bus and architecture-independent DMA layer that hides most of these issues** from the driver author. We **strongly encourage you to use this layer for DMA operations in any driver you write.**

Many of the functions below require a pointer to a *struct device*. This structure is the low-level representation of a device within the Linux device model. It is not something that drivers often have to work with directly, but you do need it when using the generic DMA layer. Usually, you can find this structure buried **inside the bus specific** that describes your device. For example, it can be found as the *dev* field in *struct pci_device* or *struct usb_device* << or *struct platform_device* >>. ...

<< *FYI: (on kernel ver 4.4.21):*

For PCI:

include/linux/pci.h

```
...
255 /*
256  * The pci_dev structure is used to describe PCI devices.
257  */
258 struct pci_dev {
...
321     pci_channel_state_t error_state;    /* current connectivity state */
322     struct device dev;                  /* Generic device interface */
323
...

```

For USB:

include/linux/usb.h

```
...
545 struct usb_device {
...
560
561     struct device dev;
562
...
>>

```

Drivers that use the following functions should include *<linux/dma-mapping.h>*.

<<

Discussion follows on dealing with the memory range that the device can address; as it varies, drivers must inform the kernel about a valid memory range by using the *dma_set_mask()* API. See [Documentation/DMA-API-HOWTO.txt](#) (initial part) for more detail.

>>

<<

Notes from “[Essential Linux Device Drivers](#)” by Sreekrishnan Venkateshwaran, Prentice Hall.
Ch 10 “Peripheral Component Interconnect”.

...

The **issue of cache coherency** is synonymous with DMA. For optimum performance, processors cache recently accessed bytes, so data passing between the CPU and main memory streams through the processor cache.

[MMU/CPU] <-----> [CPU Caches] <-----> [RAM]

During DMA, however, data travels **directly** between the DMA controller and main memory and, hence, **bypasses the processor cache**. This **evasion has the potential to introduce inconsistencies** because the processor might work on **stale data** living in its cache. Some architectures automatically synchronize the cache with main memory using a technique called bus snooping << *Intel* >>. Many others **rely on software to achieve coherency**, however. We will learn how to perform (cache) coherent DMA operations after introducing a few more topics.

DMA can occur **synchronously or asynchronously**:

An example of the former is DMA from a system frame buffer to an LCD controller. A **user application writes** pixel data to a DMA-mapped frame buffer via `/dev/fbX`, while the LCD controller uses DMA to collect this data synchronously at timed intervals.

An example of asynchronous DMA is the transmit and receive of data frames between the CPU and a network card.

DMA Buffers

System memory regions that are the **source or destination of DMA transfers** are called **DMA buffers**. If a bus interface has **addressing limitations**, that'll affect the memory range that can hold DMA buffers. So, DMA buffers suitable for a 24-bit bus such as ISA can live only in the bottom 16 MB of system memory called **ZONE_DMA**. PCI buses are 32-bits wide by default, so you won't usually face such limitations on 32-bit platforms.

To inform the kernel about any special needs of DMA-able buffers, use the following:

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



If this function returns success, **you may DMA to any address that is mask bits in length**.

[In effect, we restrict the DMA space with this function].

For example, the (Intel) e1000 PCI-X Gigabit Ethernet driver ([Linux 3.10.24 : drivers/net/ethernet/intel/e1000/e1000_main.c](#)) does the following:

```
...
1015     /* there is a workaround being applied below that limits
1016     * 64-bit DMA addresses to 64-bit hardware. There are some
1017     * 32-bit adapters that Tx hang when given 64-bit DMA addresses
1018     */
1019     pci_using_dac = 0;
1020     if ((hw->bus_type == e1000_bus_type_pcix) &&
1021         !dma_set_mask(&pdev->dev, DMA_BIT_MASK(64))) {
1022         /* Update: now the function is dma_set_mask_and_coherent() : Set both the DMA
1023         mask and the coherent DMA mask to the same thing (see screenshot below) >>
1024         /* according to DMA-API-HOWTO, coherent calls will always
1025         * succeed if the set call did
1026         */
1027         dma_set_coherent_mask(&pdev->dev, DMA_BIT_MASK(64));
1028         pci_using_dac = 1;
1029     } else { << if 64-bit DMA addresses fail, fallback to using 32-bit >>
1030         err = dma_set_mask(&pdev->dev, DMA_BIT_MASK(32));
1031         if (err) {
1032             pr_err("No usable DMA config, aborting\n");
1033             goto err_dma;
1034         }
1035         dma_set_coherent_mask(&pdev->dev, DMA_BIT_MASK(32));
1036     }
1037     ...
```

[Via Linus's github repo](#) – “blame” view:

 e1000: fix whitespace issues and multi-...	4 years ago	1012	/* there is a workaround being applied below that limits
 e1000: fix Tx hangs by disabling 64-bit ...	7 years ago	1013	* 64-bit DMA addresses to 64-bit hardware. There are some
		1014	* 32-bit adapters that Tx hang when given 64-bit DMA addresses
		1015	*/
		1016	pci_using_dac = 0;
		1017	if ((hw->bus_type == e1000_bus_type_pcix) &&
		1018	!dma_set_mask_and_coherent(&pdev->dev, DMA_BIT_MASK(64))) {
		1019	pci_using_dac = 1;
		1020	} else {
		1021	err = dma_set_mask_and_coherent(&pdev->dev, DMA_BIT_MASK(32));
 e1000: fix return value not set on error	7 years ago	1022	if (err) {
		1023	pr_err("No usable DMA config, aborting\n");
		1024	goto err_dma;
		1025	}
 e1000: fix Tx hangs by disabling 64-bit ...	7 years ago	1026	}

>>

I/O devices view DMA buffers through the lens of the bus controller and any intervening I/O memory management unit (IOMMU). Because of this, **I/O devices work with bus addresses**, rather than physical or kernel virtual addresses. So, when you inform a PCI card about the location of a DMA buffer, **you have to let it know the buffer's bus address**.

DMA service routines map the kernel virtual address of DMA buffers to bus addresses so that both the device and the CPU can access the buffers. Bus addresses are of type ***dma_addr_t***, defined in *include/asm-**<your-arch>**/types.h*.

```
<< Now in include/linux/types.h :
145 #ifdef CONFIG_ARCH_DMA_ADDR_T_64BIT
146 typedef u64 dma_addr_t;
147 #else
148 typedef u32 dma_addr_t;
149 #endif /* dma_addr_t */
>>
```

There are a couple more concepts worth knowing about DMA. One is the idea of bounce buffers. **Bounce buffers** reside in DMA-able regions and are used as temporary memory when DMA is requested to/from non-DMA-able memory regions. An example is DMA to an address higher than 4GB from a 32-bit PCI peripheral when there is no intervening IOMMU. Data is first transferred to a bounce buffer and then copied to the final destination.

The second concept is a **flavor of DMA called scatter-gather**. When data to be DMA'ed is spread over discontinuous regions, scatter-gather capability enables the hardware to gather contents of the scattered buffers at one go. The reverse occurs when data is DMA'ed from the card to buffers scattered in memory. Scatter-gather capability boosts performance by eliminating the need to service multiple DMA requests.

The kernel features **a healthy API that masks many of the internal details** of configuring DMA. This API gets simpler if you are writing a driver for a PCI card that supports bus mastering. (Most PCI cards do.) PCI DMA routines are essentially wrappers around the generic DMA service routines and are defined in *include/asm-generic/pci-dma-compat.h*. ...

DMA Mappings

The kernel provides **two classes of DMA service routines** to PCI drivers:

1. **Consistent (or coherent) DMA access methods**. These routines **guarantee data coherency** in the face of DMA activity. **If both the PCI device and the CPU are likely to frequently operate** on a DMA buffer, consistency is crucial, so use the consistent API. The trade-off is a degree of performance penalty.

...

```
<< "Consistent" and/or "coherent" simply means "Cache coherency". The area allocated is always up-to date with the CPU caches. Using this API guarantees the allocated memory is reliably setup for DMA transfers. >>
```


2. **Streaming DMA access methods**. These routines **do not guarantee (cache) consistency** and are **faster** as a result. They are useful when there is **not much need for shared access** between the CPU and the I/O device. When a streamed buffer has been mapped for device access, the driver has to **explicitly unmap** (or sync*) it before the CPU can reliably operate on it.

<< * See this SO Q&A "[Where to start learning about linux DMA / device drivers / memory allocation](#)"; there is a brief explanation on sync'ing dma as well. >>

...

Let's summarize the characteristics of coherent and streaming DMA to help you decide their suitability for your usage scenario:

- Coherent mappings are simple to code but expensive to use. Streaming mappings have the reverse characteristic.
- Coherent mappings are preferred when both the CPU and the I/O device need to **frequently manipulate the DMA buffer**. This is usually the case for **synchronous DMA**. An example is the frame buffer driver mentioned previously, where each DMA operates on the same buffer. Because the CPU and the video controller are constantly accessing the frame buffer, it makes sense to use coherent mappings in this situation.
- Use **streaming mappings when the I/O device owns the buffer for long durations**. Streamed DMA is common for **asynchronous operation** when each DMA operates on a different buffer. An example is a **network** driver, where the buffers that carry transmit packets are mapped and unmapped on-the-fly.
- **DMA descriptors** are good candidates for **coherent** mapping. DMA descriptors **contain metadata about DMA buffers** such as their address and length and are frequently accessed by both the CPU and the device. Mapping descriptors on-the-fly is expensive because that entails frequent unmappings and remappings (or sync operations).

>>

<<

In order to better understand the DMA API, a note on it as used within the kernel and by device drivers:

Source : [Documentation/DMA-API-HOWTO.txt](#)
(formatting changed for readability).

>>

Types of DMA mappings

There are **two types** of DMA mappings:

1. - Consistent (or Coherent) 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**" and "**(cache) 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 << with `dma_set_mask_and_coherent()` >> 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 (to) normal memory.

...

2.- Streaming (or async) 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.

*Think of **streaming** as "**asynchronous**" and "**outside the (cache) 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.

[Note that for both the above methods of setting up DMA buffers and mapping them to a device, the underlying implementation will take care of setting up the device IOMMU if one exists.]

(Exception: In practice, there are still problems on ARM; the driver author needs to setup the IOMMU)].

Some Details on Using Consistent and Streaming DMA APIs

1. Using Consistent (synchronous / cache coherent) DMA mappings

<<

Remember:

- *Use the consistent / coherent DMA API when both CPU and the I/O device need to frequently use the DMA buffer*
- *used frequently for Network card DMA ring descriptors at driver init time.*

>>

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

- *dev* is a *struct device **.
- *size* is the length of the region you want to allocate, in bytes.
- *gfp* is the usual GFP_XXX mask passed to allocation routines. (This API may be called in interrupt context with the GFP_ATOMIC flag).

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

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:

- *cpu_addr* : the (kernel) virtual address which you can use to access the buffer from the CPU, and
- *dma_handle* : the (physical*) bus address of the buffer which you pass to the card (device).

[* **IMPORTANT!** While we refer to the *dma_handle* as a “physical” address, one is not to take this literally! It may or may not be a literal physical address in RAM; specifically, if the device connects to the system bus via an IOMMU, the IOMMU will be translating addresses to and from the device and memory. So do not treat *dma_handle* as a pure physical address; just think of it as a “hardware” address.]

The *cpu return* address and the DMA bus master address (*dma_handle*) 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.

...

... <*description of DMA pools interface*>

...

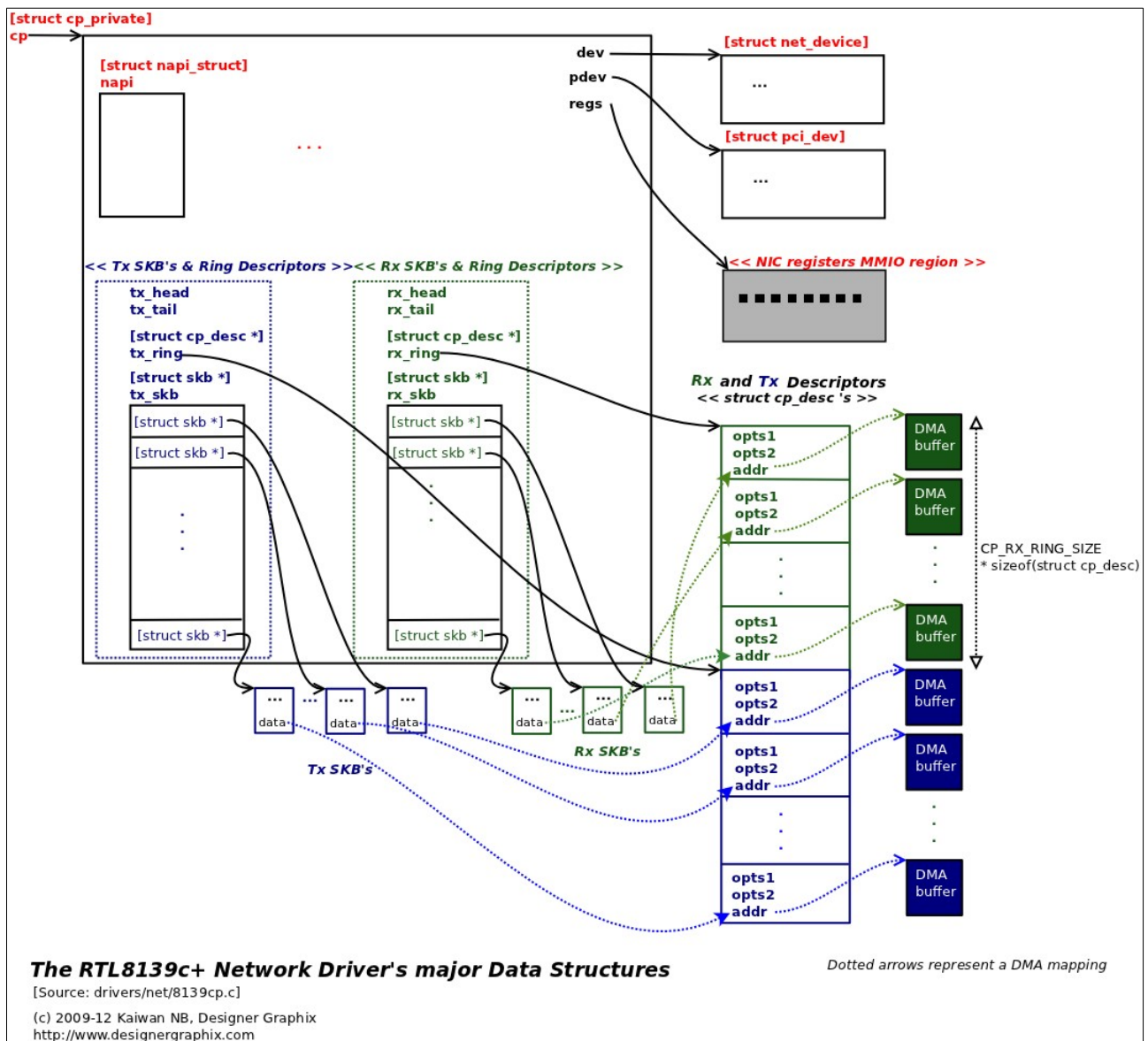
Note

If **CMA (Contiguous Memory Allocation)** is enabled (CONFIG_CMA), even large memory buffers (> 4MB) allocated with the *dma_alloc_coherent* function is *guaranteed to be physically contiguous*!

[Resource: [A deep dive into CMA](#), lwn]

We shall use the Realtek 8139cp NIC device driver as a typical use case

The “big picture” with focus on the DMA buffer management:



The DMA Descriptor object:

```
struct cp_desc {
    __le32      opts1;
    __le32      opts2;
    __le64      addr;
};
```

To perform DMA, the following steps must be taken:

1. Allocation of

- 1.1 the DMA Rx/Tx Descriptor objects; as coherent DMA memory (`dma_alloc_coherent`)
- 1.2 the Rx/Tx SKB arrays (normal kernel allocation, `netdev_alloc_skb_ip_align`)

1.3 the Rx/Tx DMA buffer arrays must be allocated as a streaming DMA allocation (*dma_map_single*).

2. the Rx / Tx *skb* → *data* pointers must be *mapped* to the streaming DMA.

1.1 Allocate the DMA Rx/Tx Descriptor objects; as coherent DMA memory (*dma_alloc_coherent*)

Code: the array of **DMA Tx and Rx Descriptors** being allocated as coherent DMA memory by the network driver:

Code below is from the 8139cp NIC driver [\[drivers/net/ethernet/realtek/8139cp.c\]](#)

```
static int cp_alloc_rings (struct cp_private *cp)    << called from cp_open(),
                                                    which is the NIC "up" routine hook >>
{
    struct device *d = &cp->pdev->dev;
    void *mem;
    int rc;

    << cpu_addr=dma_alloc_coherent(dev, size, &dma_handle, gfp); >>
    mem = dma_alloc_coherent(d, CP_RING_BYTES, &cp->ring_dma, GFP_KERNEL);
    if (!mem)    << alloc sufficient DMA-addressable mem for the entire Tx and Rx DMA ring
    buffers:
        << #define CP_RING_BYTES
                |
                ((sizeof(struct cp_desc) * CP_RX_RING_SIZE) + |
                (sizeof(struct cp_desc) * CP_TX_RING_SIZE) + |
                CP_STATS_SIZE) >>
    return -ENOMEM;

    cp->rx_ring = mem;
    cp->tx_ring = &cp->rx_ring[CP_RX_RING_SIZE];    << 64 bytes >>
    ...
}
```

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.

...

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.

...

2. Using Streaming (asynchronous / non-coherent) DMA mappings

<<

Remember:

- *Use the streaming / non-coherent DMA API when the I/O device works for long-ish times with the DMA buffer*

- *used frequently for networking buffers transmitted/received by a device.*

>>

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:

<<

```
dma_addr_t dma_map_single(struct device *dev, void *cpu_addr, size_t size,
                          enum dma_data_direction direction)
```

dma_map_single() maps a piece of processor virtual memory << *meaning, it's already been allocated and mapped by the kernel* >> **so it can be accessed by the device** and **returns the physical handle** of the memory.

It's interesting to contrast this API with the *ioremap()*; *ioremap* takes a physical (bus) address and returns a kernel ("cpu-accessible") virtual address, allowing the driver to access the physical memory using a kernel virtual address. *dma_map_single()* is **kind of the "opposite"**: it takes a kernel/cpu virtual address and returns a hardware DMA address, **allowing the device to access the hardware memory (the page frame in RAM) referenced by the kernel virtual address.**

>>

```
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.

...

<Notes on using *dma_[un]map_sg* – for scatterlists>

>>

1.2 Allocate the Rx/Tx SKB arrays (normal kernel allocation, `netdev_alloc_skb_ip_align`), and
 1.3 the Rx/Tx DMA buffer arrays must be allocated as a streaming DMA allocation (`dma_map_single`).

2. the Rx / Tx `skb` → `data` pointers must be *mapped* to the streaming DMA.

Sample code below from the 8139cp NIC driver [\[drivers/net/ethernet/realtek/8139cp.c\]](#)

Rx SKB allocation and DMA setup

The Rx SKB's are allocated (kernel) memory (min of 1536 bytes (PKT_BUF_SZ; also, it's the typical MTU)) via the `netdev_alloc_skb_ip_align()` function (in `cp_refill_rx`). They are then *mapped* for DMA (in effect, the hardware NIC can now “see” the kernel memory) via the `dma_map_single` API.

```
...
static int cp_refill_rx(struct cp_private *cp)
{
    ...
    for (i = 0; i < CP_RX_RING_SIZE; i++) {
        struct sk_buff *skb;
        dma_addr_t mapping;

        skb = netdev_alloc_skb_ip_align(dev, cp->rx_buf_sz);
        << alloc 1536 bytes >>

        if (!skb)
            goto err_out;

<<
#define dma_map_single(d, a, s, r) dma_map_single_attrs(d, a, s, r, NULL)
...
static inline dma_addr_t dma_map_single_attrs(struct device *dev, void *ptr,
        size_t size,
        enum dma_data_direction dir,
        struct dma_attrs *attrs) { ... }
```

FYI, more details on this API follow below...

>>

```
        mapping = dma_map_single(&cp->pdev->dev, skb->data, << skb->data: this is the “piece
of processor virtual memory so it can be accessed by the device” - the pre-allocated Rx buffers
setup to receive incoming network packets >>
        cp->rx_buf_sz, PCI_DMA_FROMDEVICE);
        << the return value is the DMA-mapped hardware address;
        useful to see the diagram above >>
        if (dma_mapping_error(&cp->pdev->dev, mapping)) { << TIP: DO / MUST use
        the dma_mapping_error() API to check for errors! >>
            kfree_skb(skb);
            goto err_out;
        }
        cp->rx_skb[i] = skb;

        << Setup the Rx DMA descriptors (desc format is from the NIC datasheet)
        for receiving incoming network packets:
        struct cp_desc {
```



```

        __le32      opts1;
        __le32      opts2;
        __le64      addr;
    };
>>
    cp->rx_ring[i].opts2 = 0;
    cp->rx_ring[i].addr = cpu_to_le64(mapping); << Rx ring DMA address is
        set to the hardware DMA address just returned >>

    if (i == (CP_RX_RING_SIZE - 1))
        cp->rx_ring[i].opts1 =
            cpu_to_le32(DescOwn | RingEnd | cp->rx_buf_sz);
    else
        cp->rx_ring[i].opts1 =
            cpu_to_le32(DescOwn | cp->rx_buf_sz);
    ...
        << /* Tx and Rx status descriptors */
        DescOwn      = (1 << 31), /* Descriptor is owned by NIC */
    >>
}
...

```

Tx SKB allocation and DMA setup

```

static netdev_tx_t cp_start_xmit(struct sk_buff *skb, << driver transmit method: skb already
populated by protocol stack on Tx path >>
    struct net_device *dev)
{
    ...
    len = skb->len;
    mapping = dma_map_single(&cp->pdev->dev, skb->data, << skb->data: this is the "piece of
processor virtual memory so it can be accessed by the device" - the pre-allocated and populated
network packet to be transmitted >>
        len, PCI_DMA_TODEVICE);
    << return value is the DMA-mapped hardware address >>
    if (dma_mapping_error(&cp->pdev->dev, mapping)) << ALWAYS check! >>
        goto out_dma_error;

    << Setup the Tx DMA descriptors (desc format is from the NIC datasheet)
    for transmitting outgoing network packets:
        struct cp_desc {
            __le32      opts1;
            __le32      opts2;
            __le64      addr;
        };
    >>

    txd->opts2 = opts2;
    txd->addr = cpu_to_le64(mapping); << DMA descriptor is set to the
        hardware DMA address >>

    wmb();
    opts1 |= eor | len | FirstFrag | LastFrag;

    txd->opts1 = cpu_to_le32(opts1);
    wmb();

    cp->tx_skb[entry] = skb;
    cp->tx_opts[entry] = opts1;
    netif_dbg(cp, tx_queued, cp->dev, "tx queued, slot %d, skblen %d\n",
        entry, skb->len);
    ...
}

```

Okay, so as we've seen above, the following have been allocated and initialized:

- the DMA Rx/Tx Descriptor objects; as coherent DMA memory (*dma_alloc_coherent*)
- the Rx / Tx SKB arrays (normal kernel allocation, *netdev_alloc_skb_ip_align*)
- the Rx / Tx DMA buffer arrays are allocated as a streaming allocation (*dma_map_single*)
- the Rx / Tx *skb* → *data* pointers have been *mapped* to the streaming DMA.

Now all that is required for the NIC hardware and driver to be able to perform DMA is this: the *start address of the DMA Rx/Tx Descriptor array must be made known to the NIC*. How? The data sheet for the Realtek 8139cp chip prescribes that the DMA address of the start of the Rx and Tx descriptor arrays be written into the *RxRingAddr* and *TxRingAddr* registers respectively. The driver performs this here:

cp_open → *cp_init_hw* → *cp_start_hw*

```
static inline void cp_start_hw (struct cp_private *cp)
{
    dma_addr_t ring_dma;

    cpw16(CpCmd, cp->cpcmd);

    /*
     * These (at least TxRingAddr) need to be configured after the
     * corresponding bits in CpCmd are enabled. Datasheet v1.6 §6.33
     * (C+ Command Register) recommends that these and more be configured
     * *after* the [RT]xEnable bits in CpCmd are set. And on some hardware
     * it's been observed that the TxRingAddr is actually reset to garbage
     * when C+ mode Tx is enabled in CpCmd.
     */
    cpw32_f(HiTxRingAddr, 0);
    cpw32_f(HiTxRingAddr + 4, 0);

<<
enum {
    /* NIC register offsets */
    ...
    TxRingAddr    = 0x20, /* 64-bit start addr of Tx ring */
    Cmd           = 0x37, /* Command register */
    RxRingAddr    = 0xE4, /* 64-bit start addr of Rx ring */
>>

    ring_dma = cp->ring_dma;
    cpw32_f(RxRingAddr, ring_dma & 0xffffffff);
    cpw32_f(RxRingAddr + 4, (ring_dma >> 16) >> 16);

    ring_dma += sizeof(struct cp_desc) * CP_RX_RING_SIZE;
    cpw32_f(TxRingAddr, ring_dma & 0xffffffff);
    cpw32_f(TxRingAddr + 4, (ring_dma >> 16) >> 16);

    /*
     * Strictly speaking, the datasheet says this should be enabled
     * *before* setting the descriptor addresses. But what, then, would
     * prevent it from doing DMA to random unconfigured addresses?
     * This variant appears to work fine.
     */
    cpw8(Cmd, Rx0n | Tx0n);

    netdev_reset_queue(cp->dev);
}
```

Synchronization

When performing streaming DMA, the DMA buffers are frequently mapped and unmapped.

Process:

allocate kernel memory ->

map to the device -> perform DMA I/O (r/w) -> unmap it ->

deallocate memory

However, bear in mind that frequently mapping and unmapping memory to and from the device is *expensive*. So, to mitigate this, **high performance drivers will keep the mapping around and just change the “ownership” of the mapping.**

So, how do we tell the system when it is okay for the kernel to access the memory and when the DMA controller is using it? This can be done with the

`dma_sync_single_for_cpu`, and

`dma_sync_single_for_device`

APIs.

Let kernel (cpu) access – have ownership over the DMA buffer (specified by ‘`addr`’):

```
static inline void dma_sync_single_for_cpu(struct device *dev, dma_addr_t addr,
                                         size_t size,
                                         enum dma_data_direction dir);
```

Let device access – have ownership over the DMA buffer (specified by ‘`addr`’):

```
static inline void dma_sync_single_for_device(struct device *dev,
                                             dma_addr_t addr, size_t size,
                                             enum dma_data_direction dir)
```

Note that this synchronization is not required at all for coherent DMA mappings – only for ‘streaming’ mappings.

Eg. Many. Just use **cscope** to look up ‘*functions calling xxx*’; where `xxx`= `dma_alloc_coherent` and `dma_map_single`.

Eg. In the SPI-based SD/MMC host driver-
`drivers/mmc/host/mmc_spi.c` :

```
...
/* underlying DMA-aware controller, or null */
struct device      *dma_dev;
...
mmc_spi_probe()
...
    host->dma_dev = dev;
    host->ones_dma = dma_map_single(dev, ones,
                                   MMC_SPI_BLOCKSIZE, DMA_TO_DEVICE);
    host->data_dma = dma_map_single(dev, host->data,
                                   sizeof(*host->data), DMA_BIDIRECTIONAL);

    /* REVISIT in theory those map operations can fail... */

    dma_sync_single_for_cpu(host->dma_dev,
                            host->data_dma, sizeof(*host->data),
```

```
DMA_BIDIRECTIONAL);
```

```
...
```

In Summary, DMA (De)Allocation for a (typical) NIC driver:

- The Rx and Tx **Ring Descriptors** are allocated DMA memory (at the network interface “open” time) by `dma_alloc_coherent`, i.e., via a **coherent** (synchronous, *cache-coherent but slower*) DMA allocation
 - and conversely freed via the `dma_free_coherent` API
 - The **Tx/Rx buffers**, i.e., the SKB (pointed-to) data regions which will hold **packet data (the payload)**, are allocated and mapped (and subsequently refilled) DMA memory by the combination of `netdev_alloc_skb_ip_align` and `dma_map_single` APIs respectively, i.e., via a **streaming** (asynchronous; *cache-coherency not guaranteed but faster*) DMA mapping
 - and conversely freed and unmapped via the `dev_free_skb` and `dma_unmap_single` APIs.
-

[OPTIONAL / FYI]

- A quite simple and interesting example of a “DMA test application” device driver is from Xilinx (for it's Zynq PL330 DMA controller). [See the wiki page here](#).
 - See this Q&A on SO: [Linux DMA: Using the DMAengine for scatter-gather transactions](#) [May '16]
 - *FAQ : Can one perform DMA directly to user pages?*
 - [Linux kernel device driver to DMA from a device into user-space memory](#)
 - [Mapping DMA buffers to userspace](#)
 - Refer to this question / answer on StackOverflow: [From the kernel to the user space \(DMA\)](#).
 - Has links to using the [PF-RING](#) and [Netmap frameworks](#), which bypass the kernel completely in order to perform very fast data transfer (DMA) from the NIC to user-space directly.
 - See netmap poster: [netmap: memory-mapped access to network devices](#)
 - Hardware layer DMA Management - [“Implementing DMA on ARM SMP Systems”](#).
-

[FYI / OPTIONAL]

More Details on DMA API

Source : [Documentation/DMA-API.txt](#)

```
...
```

Part Id - Streaming DMA mappings

```
dma_addr_t
dma_map_single(struct device *dev, void *cpu_addr, size_t size,
               enum dma_data_direction direction)
```

Maps a piece of processor virtual memory so it can be accessed by the device and returns the physical handle of the memory.

The direction for both api's may be converted freely by casting. However the dma_ API uses a strongly typed enumerator for its direction:

DMA_NONE	no direction (used for debugging)
DMA_TO_DEVICE	data is going from the memory to the device
DMA_FROM_DEVICE	data is coming from the device to the memory
DMA_BIDIRECTIONAL	direction isn't known

Notes: Not all memory regions in a machine can be mapped by this API. Further, regions that appear to be physically contiguous in kernel virtual space may not be contiguous as physical memory. Since this API does not provide any scatter/gather capability, it will fail if the user tries to map a non-physically contiguous piece of memory. For this reason, it is recommended that memory mapped by this API be obtained only from sources which guarantee it to be physically contiguous (like kmalloc).

Further, the physical address of the memory must be within the dma_mask of the device (the dma_mask represents a bit mask of the addressable region for the device. I.e., if the physical address of the memory anded with the dma_mask is still equal to the physical address, then the device can perform DMA to the memory). In order to ensure that the memory allocated by kmalloc is within the dma_mask, the driver may specify various platform-dependent flags to restrict the physical memory range of the allocation (e.g. on x86, GFP_DMA guarantees to be within the first 16Mb of available physical memory, as required by ISA devices).

Note also that the above constraints on physical contiguity and dma_mask may not apply if the platform has an IOMMU (a device which supplies a physical to virtual mapping between the I/O memory bus and the device). However, to be portable, device driver writers may **not** assume that such an IOMMU exists.

Warnings: Memory coherency operates at a granularity called the cache line width. In order for memory mapped by this API to operate correctly, **the mapped region must begin exactly on a cache line boundary and end exactly on one** (to prevent two separately mapped regions from sharing a single cache line). Since the cache line size may not be known at compile time, the API will not enforce this requirement. Therefore, it is recommended that driver writers who don't take special care to determine the cache line size at run time **only map virtual regions that begin and end on page boundaries (which are guaranteed also to be cache line boundaries)**.

DMA_TO_DEVICE synchronisation must be done after the last modification of the memory region by the software and before it is handed off to the driver. Once this primitive is used, memory covered by this primitive should be treated as read-only by the device. If the device may write to it at any point, it should be DMA_BIDIRECTIONAL (see below).

DMA_FROM_DEVICE synchronisation must be done before the driver accesses data that may be changed by the device. This memory should be treated as read-only by the driver. If the driver needs to write to it at any point, it should be DMA_BIDIRECTIONAL (see below).

DMA_BIDIRECTIONAL requires special handling: it means that the driver isn't sure if the memory was

modified before being handed off to the device and also isn't sure if the device will also modify it. Thus, you must always sync bidirectional memory twice: once before the memory is handed off to the device (to make sure all memory changes are flushed from the processor) and once before the data may be accessed after being used by the device (to make sure any processor cache lines are updated with data that the device may have changed).

```
void
dma_unmap_single(struct device *dev, dma_addr_t dma_addr, size_t size,
                 enum dma_data_direction direction)
```

Unmaps the region previously mapped. All the parameters passed in must be identical to those passed in (and returned) by the mapping API.

...

[Source](#)

Scatter-gather mappings

This is a special case of streaming DMA mappings. You may need to transfer several buffers all at the same time.

Scatter-Gather (SG) mappings allow a set of physically non-contiguous memory locations (on the kernel side), to be directly DMA mapped to the device. This way, the device can read or write multiple memory chunks in “one shot” rather than looping over each of them. Of course, for the device to be able to map the physically non-contiguous memory locations, **it must have an IOMMU**.

[See page 409-410 of device driver book << LDD3 >> for details on how this may occur and why scatter-gather has advantages over mapping each buffer by themselves.]

<< *Easier method to setup sg-DMA seen below, after this section* >>

To use scatter-gather mappings, you need to:

- create and fill in an array of *struct scatterlist* for buffs to be transfered
 - found in <linux/asm-generic/scatterlist.h>
 - has: `char *address` - address of buffer
 - `unsigned int length` - length of that buffer.
- call:


```
int pci_map_sg(struct pci_dev *pdev, struct scatterlist *list,
               int nents, int direction);
```

This returns the number of DMA buffers to transfer, NOTE: this may be less than nents which is the number of scatterlist entries passed in.

- transfer each buffer returned by *pci_map_sg()*. use the following calls to make your code portable:


```
dma_addr_t sg_dma_address(struct scatterlist *sg);
```

```
unsigned int sg_dma_len(struct scatterlist *sg);
```

Use these to get the fields of the scatterlist entries because the location in the structure varies from arch to arch.

NOTE: the address and length of the buffers may be different from what was passed into *pci_map_sg()*.

- after the transfer is complete, you need to unmap the scatter gather mapping:

```
void pci_unmap_sg(struct pci_dev *pdev,  
                  struct scatterlist *list,  
                  int nents, int direction);
```

NOTE: nents must be the the same that you passed into *pci_map_sg* (not what it returned).

The same rules apply to scatter-gather mappings as the regular streaming mappings. Use *pci_dma_sync_sg(<same args as pci_dma_sync_single>)* to sync the buffer before access to the mapped buffer.

See the device driver book (LDD3) page 411 to see how different arch suport PCI DMA because they are very hardware dependant.

...

Practical Usage of DMA in device drivers

Sidebar :: SG-DMA / CMA

Hardware DMA Engine on device (camera/disk/NIC/etc):
- supports SG-DMA? (Scatter-Gather)

If Y, then we don't absolutely require a large physically contiguous chunk of RAM for the DMA buffer; can setup an sg-list and initiate data transfer. (See eg.

`drivers/block/cpqarray.c:do_ida_request()` and SO:[Linux kernel device driver to DMA from a device into user-space memory](#)).

If N, then unfortunately you cannot transfer in small pieces via SG-DMA and will require a large physically contiguous chunk of RAM for the DMA buffer. Could be a problem. Unless you're on Linux kernel ver ≥ 3.18 and can therefore use the Contiguous Memory Allocator (CMA). CMA's API is abstracted via the 'usual' DMA generic layer API; just use the usual APIs (`dma_alloc_coherent/etc`) requesting whatever large-sized DMA buffer you require.

1. Common case: how exactly to perform DMA from device directly to a userspace buffer

The answer below is directly copied from here [stackoverflow]:

[Linux kernel device driver to DMA from a device into user-space memory](#)

...

["Correct Answer"]

I'm actually working on exactly the same thing right now and I'm going the `ioctl()` route. The general idea is for user space to allocate the buffer which will be used for the DMA transfer and an `ioctl()` will be used to pass the size and address of this buffer to the device driver. The driver will then use scatter-gather lists along with the streaming DMA API to transfer data directly to and from the device and user-space buffer.

The implementation strategy I'm using is that the `ioctl()` in the driver enters a loop that DMA's the userspace buffer in chunks of 256k (which is the hardware imposed limit for how many scatter/gather entries it can handle). This is isolated inside a function that blocks until each transfer is complete (see below). When all bytes are transferred or the incremental transfer function returns an error the `ioctl()` exits and returns to userspace.

Pseudo code for the `ioctl()`

```
/*serialize all DMA transfers to/from the device*/
if (mutex_lock_interruptible( &device_ptr->mtx ) )
    return -EINTR;

chunk_data = (unsigned long) user_space_addr;
while( *transferred < total_bytes && !ret ) {
    chunk_bytes = total_bytes - *transferred;
```

```

    if (chunk_bytes > HW_DMA_MAX)
        chunk_bytes = HW_DMA_MAX; /* 256kb limit imposed by my device */
    ret = transfer_chunk(device_ptr, chunk_data, chunk_bytes, transferred);
    chunk_data += chunk_bytes;
    chunk_offset += chunk_bytes;
}

mutex_unlock(&device_ptr->mtx);

```

Pseudo code for incremental transfer function [`transfer_chunk`]:

```

/*Assuming the userspace pointer is passed as an unsigned long, */
/*calculate the first,last, and number of pages being transferred via*/

first_page = (udata & PAGE_MASK) >> PAGE_SHIFT;
last_page = ((udata+nbytes-1) & PAGE_MASK) >> PAGE_SHIFT;
first_page_offset = udata & PAGE_MASK;
npages = last_page - first_page + 1;

/* Ensure that all userspace pages are locked in memory for the */
/* duration of the DMA transfer */

down_read(&current->mm->mmap_sem);
ret = get_user_pages(current, << get_user_pages() pins down user pages in memory >>
                    current->mm,
                    udata,
                    npages,
                    is_writing_to_userspace,
                    0,
                    &pages_array,
                    NULL);
up_read(&current->mm->mmap_sem);

/* Map a scatter-gather list to point at the userspace pages */

/*first*/
sg_set_page(&sglist[0], pages_array[0], PAGE_SIZE - fp_offset, fp_offset);

/*middle*/
for(i=1; i < npages-1; i++)
    sg_set_page(&sglist[i], pages_array[i], PAGE_SIZE, 0);

/*last*/
if (npages > 1) {
    sg_set_page(&sglist[npages-1], pages_array[npages-1],
                nbytes - (PAGE_SIZE - fp_offset) - ((npages-2)*PAGE_SIZE), 0);
}

/* Do the hardware specific thing to give it the scatter-gather list
   and tell it to start the DMA transfer */
[DMA descriptor setup, DMA buffer alloc/mapping/etc, trigger DMA]

/* Wait for the DMA transfer to complete */
ret = wait_event_interruptible_timeout( &device_ptr->dma_wait,
                                       &device_ptr->flag_dma_done, HZ*2 );

if (ret == 0)

```

```

    /* DMA operation timed out */
else if (ret == -ERESTARTSYS )
    /* DMA operation interrupted by signal */
else {
    /* DMA success */
    *transferred += nbytes;
    return 0;
}

```

The interrupt handler is exceptionally brief:


```

/* Do hardware specific thing to make the device happy */
[DMA buffer unmapping, ...]

/* Wake the thread waiting for this DMA operation to complete */
device_ptr->flag_dma_done = 1;
wake_up_interruptible(device_ptr->dma_wait);

```

Please note that this is just a general approach, I've been working on this driver for the last few weeks and have yet to actually test it... So please, don't treat this pseudo code as gospel and be sure to double check all logic and parameters ;-).

answered Apr 4 '11 at 14:38
 Rakis
 5,7741323

...

[Another answer]

You basically have the right idea: in 2.1, you can just have userspace allocate any old memory. You do want it page-aligned, so `posix_memalign()` is a handy API to use.

Then have userspace pass in the userspace virtual address and size of this buffer somehow; `ioctl()` is a good quick and dirty way to do this. In the kernel, allocate an appropriately sized buffer array of `struct page*` -- `user_buf_size/PAGE_SIZE` entries -- and use `get_user_pages()` to get a list of `struct page*` for the userspace buffer.

Once you have that, you can allocate an array of `struct scatterlist` that is the same size as your page array and loop through the list of pages doing `sg_set_page()`. After the sg list is set up, you do `dma_map_sg()` on the array of scatterlist and then you can get the `sg_dma_address` and `sg_dma_len` for each entry in the scatterlist (note you have to use the return value of `dma_map_sg()` because you may end up with fewer mapped entries because things might get merged by the DMA mapping code).

That gives you all the bus addresses to pass to your device, and then you can trigger the DMA and wait for it however you want. The read()-based scheme you have is probably fine. You can refer to `drivers/infiniband/core/umem.c`, specifically `ib_umem_get()`, for some code that builds up this mapping, although the generality that that code needs to deal with may make it a bit confusing.

Alternatively, if your device doesn't handle scatter/gather lists too well and you want contiguous memory, you could use `get_free_pages()` to allocate a physically contiguous buffer and use `dma_map_page()` on that. To give userspace access to that memory, your driver just needs to implement an `mmap` method instead of the `ioctl` as described above.



answered May 21 '11 at 7:16

[Roland](#)

4,604

2. Block Drivers and DMA

[Source: *Essential Linux Device Drivers*, Venkateshwaran]

If you are working on a high-performance block driver, chances are you will be **using DMA** for the actual data transfers. A block driver can certainly step through the *bio* structures, create a DMA mapping for each one, and pass the result to the device. There is an easier way, however, if your device can do scatter/gather I/O. The function:

```
int blk_rq_map_sg(request_queue_t *queue,
    struct request *req, struct scatterlist *list);
```

fills in the given list with the full set of segments from the given request. Segments that are adjacent in memory are coalesced prior to insertion into the scatterlist, so you need not try to detect them yourself. The return value is the number of entries in the list. The function also passes back, in its third argument, a scatterlist suitable for passing to `dma_map_sg`. (See the section “Scatter-gather mappings” in Chapter 15 for more information on `dma_map_sg`.)

Your driver must allocate the storage for the scatterlist before calling `blk_rq_map_sg`. The list must be able to hold at least as many entries as the request has physical segments; the `struct request` field `nr_phys_segments` holds that count, which will not exceed the maximum number of physical segments specified with `blk_queue_max_phys_segments`.

```
<<
File : [kernel 3.10.24 ] : block/blk-merge.c
...
156 /*
157  * map a request to scatterlist, return number of sg entries setup. Caller
158  * must make sure sg can hold rq->nr_phys_segments entries
159  */
160 int blk_rq_map_sg(struct request_queue *q, struct request *rq,
161     struct scatterlist *sglist)
162 {
163     struct bio_vec *bvec, *bvprv;
164     struct req_iterator iter;
165     ...
171     /*
172      * for each bio in rq
173      */
174     bvprv = NULL;
175     sg = NULL;
176     rq_for_each_segment(bvec, rq, iter) {
```

```

177     __blk_segment_map_sg(q, bvec, sglist, &bvprv, &sg,
178                          &nsegs, &cluster);
179 } /* segments in rq */
...

```

>>

If you do not want `blk_rq_map_sg` to coalesce adjacent segments, you can change the default behavior with a call such as:

```
clear_bit(Queue_FLAG_CLUSTER, &queue->queue_flags);
```

Some SCSI disk drivers mark their request queue in this way, since they do not benefit from the coalescing of requests.

<<

SG-DMA:

A real driver that does scatter-gather (SG-)DMA : the Disk Array driver for Compaq SMART2 Controllers.

Code: `drivers/block/cpqarray.c:do_ida_request()` function.

For context, see the excellent text "Essential Linux Device Drivers", Venkateswaran, Ch 14 "Block Drivers" sec "Advanced Topics" pg 484.

Code View

File : [kernel 3.10.24] : `drivers/block/cpqarray.c`

```

...
<< In interrupt context here ... >>
897 /*
898  * Get a request and submit it to the controller.
899  * This routine needs to grab all the requests it possibly can from the
900  * req Q and submit them. Interrupts are off (and need to be off) when you
901  * are in here (either via the dummy do_ida_request functions or by being
902  * called from the interrupt handler
903  */
904 static void do_ida_request(struct request_queue *q)
905 {
906     ctlr_info_t *h = q->queuedata;
907     cmdlist_t *c;
908     struct request *creq;
909     struct scatterlist tmp_sg[SG_MAX]; << here, SG_MAX = 32 >>
910     int i, dir, seg;
911
912 queue_next:
913     creq = blk_peek_request(q);
914     if (!creq) << no more request queued? Start IO... >>
915         goto startio;
916
917     BUG_ON(creq->nr_phys_segments > SG_MAX);
918
919     if ((c = cmd_alloc(h,1)) == NULL)
920         goto startio;
921
922     blk_start_request(creq); << Start request processing on the driver
                             Dequeue @req and start timeout timer on it. This hands off the
                             request to the driver. >>

```

```

923
924     c->ctrlr = h->ctrlr;
925     c->hdr.unit = (drv_info_t *) (creq->rq_disk->private_data) - h->drv;
926     c->hdr.size = sizeof(rblk_t) >> 2;
927     c->size += sizeof(rblk_t);
928
929     c->req.hdr.blk = blk_rq_pos(creq); << the current sector >>
930     c->rq = creq;
931     DBGKX(
932         printk("sector=%d, nr_sectors=%u\n",
933             blk_rq_pos(creq), blk_rq_sectors(creq));
934     );
935     sg_init_table(tmp_sg, SG_MAX);
936     seg = blk_rq_map_sg(q, creq, tmp_sg); << map a request to scatterlist,
        return number of sg entries setup. Caller must make sure sg can hold
        rq->nr_phys_segments entries >>
937
938     /* Now do all the DMA Mappings */
939     if (rq_data_dir(creq) == READ)
940         dir = PCI_DMA_FROMDEVICE;
941     else
942         dir = PCI_DMA_TODEVICE;
943     for( i=0; i < seg; i++) << loop around the scatterlist, setting up the
        DMA descriptors and mapping each page in the list >>
944     {
945         c->req.sg[i].size = tmp_sg[i].length;
946         c->req.sg[i].addr = (__u32) pci_map_page(h->pci_dev,
            << pci_map_page is a wrapper over dma_map_page >>
            sg_page(&tmp_sg[i]),
            tmp_sg[i].offset,
            tmp_sg[i].length, dir);
947     }
948
949     DBGKX( printk("Submitting %u sectors in %d segments\n", blk_rq_sectors(creq), seg); );
950
951     c->req.hdr.sg_cnt = seg;
952     c->req.hdr.blk_cnt = blk_rq_sectors(creq); << sectors left in the
        entire request >>
953
954     c->req.hdr.cmd = (rq_data_dir(creq) == READ) ? IDA_READ : IDA_WRITE;
955     c->type = CMD_RWREQ;
956
957     /* Put the request on the tail of the request queue */
958     addQ(&h->reqQ, c);
959     h->Qdepth++;
960     if (h->Qdepth > h->maxQsinceinit)
961         h->maxQsinceinit = h->Qdepth;
962
963     goto queue_next;
964
965     startio:
966     start_io(h);
967 }

```

<< The comment in the driver's start_io() function says:

```

970 * start_io submits everything on a controller's request queue
971 * and moves it to the completion queue.
972 *
973 * Interrupts had better be off if you're in here
>>

```

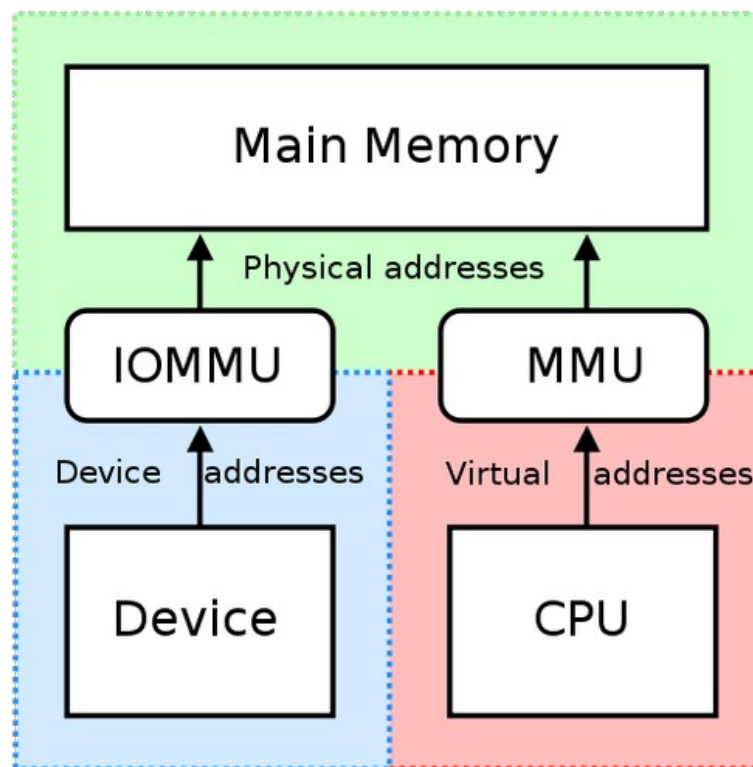
>>

[ARM, DMA, and memory management, Jon Corbet, LWN](#)

As the effort to bring proper abstractions to the ARM architecture and remove duplicated code continues, one clear problem area that has arisen is in the area of DMA memory management. The ARM architecture brings some unique challenges to this area, but the problems are not all ARM-specific. We are also seeing an interesting view into a future where more complex hardware requires new mechanisms within the kernel to operate properly.

One development in the ARM sphere is the somewhat belated addition of I/O memory management units (IOMMUs) to the architecture.

<<



Source: [Quora: Memory Management: Could you explain IOMMU in plain English?](#)

>>

An IOMMU sits between a device and main memory, translating addresses between the two. One obvious application of an IOMMU is to make physically scattered memory look contiguous to the device, simplifying large DMA transfers. An IOMMU can also restrict DMA access to a specific range of memory, adding a layer of protection to the system. Even in the absence of security worries, a device which can scribble on random memory can cause no end of hard-to-debug problems.

As this feature has come to ARM systems, developers have, in the classic ARM fashion, created special interfaces for the management of IOMMUs. The only problem is that the kernel already has an interface for the management of IOMMUs - it's the DMA API. Drivers which use this API should work on just about any architecture; all of the related problems, including cache coherency, IOMMU programming, and bounce buffering, are nicely hidden. So it seems clear that the DMA API is the mechanism by which ARM-based drivers, too, should work with IOMMUs;

ARM maintainer Russell King recently [made this point](#) in no uncertain terms.

That said, there are some interesting difficulties which arise when using the DMA API on the ARM architecture. Most of these **problems have their roots in the architecture's inability to deal with multiple mappings to a page if those mappings do not all share the same attributes**. This is a problem which has come up before; see [this article](#) for more information. In the DMA context, it is quite easy to create mappings with conflicting attributes, and performance concerns are likely to make such conflicts more common.

...

Note-

It appears that drivers on ARM still do need to take care of IOMMU for the device they're programming. They typically use these APIs:

```
arm_iommu_create_mapping
arm_iommu_release_mapping

arm_iommu_attach_device
arm_iommu_detach_device
```

Ref: See [this video tutorial](#) [time: ~ 40min into the presentation].

ARM Hardware layer DMA Management

Source: "[Implementing DMA on ARM SMP Systems](#)".

...

Considerations on using DMA

Most modern operating systems, including Linux, provide a DMA API capable of handling coherency between CPUs and external devices accessing the same physical memory.

The following cases are possible in the context of Linux device drivers:

- Use uncached memory mapping from kernel space, usually allocated using `dma_alloc_coherent()`
- Use uncached memory mapping from user space, usually created with `dma_mmap_coherent()` (in the kernel driver)
- Use cached mapping and clean or invalidate it according to the operation needed (`dma_map_single()` and `dma_unmap_single()`)

When opting for un-cached memory mapping, the memory utilized can be configured either as strongly ordered or normal uncached. Strongly ordered memory is memory configured to be uncached and un-buffered: the changes made by a CPU on a shared strongly order locations of memory are immediately visible to all cores in the system (and do not require barriers). Normal uncached is buffered memory, so memory barriers will be required. The cache is cleaned and invalidated when the mapping is created so there can be no stale data.

When using cached memory, the driver specifies the **direction** of the DMA operation (FROM_DEVICE or TO_DEVICE) so that the **cache is cleaned or invalidated accordingly**. Using cached memory mapping may be **preferred** in many circumstances (even where it gives rise to memory coherency issues requiring extra software clean/invalidate operations) as it is likely to result in a much more efficient set of bus accesses and faster overall performance.

Zero-copy DMA should be used wherever possible: User pages are mapped for DMA and the transfer takes place from device memory directly into user space. A traditional (non-zero-copy) solution would copy the data across several intermediate buffers between user and kernel spaces, adding considerable overheads. Linux 2.6 supports zero-copy DMA functionality for all block devices.

...

<<

Zero-copy Networking Techniques

Refer to this question / answer on StackOverflow: [From the kernel to the user space \(DMA\)](#)

Has links to using the [PF-RING](#) and [Netmap frameworks](#), which bypass the kernel completely in order to perform very fast data transfer (DMA) from the NIC to user-space directly.

[netmap: memory-mapped access to network devices](#)

Also see:

- In kernel documentation:

[Documentation/networking/packet_mmap.txt](#)

- [Efficient data transfer through zero copy](#)

www.ibm.com/developerworks/linux/library/j-j-zerocopy/

- [Zero Copy I: User-Mode Perspective](#)

www.linuxjournal.com/article/6345

>>

CMA

[Source: A Deep Dive into CMA, lwn](#)

The contiguous memory allocation (CMA) patches are an attempt to put together a flexible solution which can be used in all drivers. The basic technique will be familiar: CMA grabs a chunk of contiguous physical memory at boot time (when it's plentiful), then doles it out to drivers in response to allocation requests. Where it differs is mainly in an elaborate mechanism for defining the memory region(s) to reserve and the policies for handing them out.

...

So how does it work?

To understand how CMA works, one needs to know a little about migrate types and pageblocks.

When requesting memory from the buddy allocator, one provides a `gfp_mask`. Among other things, it specifies the "migrate type" of the requested page(s). One of the migrate types is `MIGRATE_MOVABLE`. The idea behind it is that data from a movable page can be migrated (or moved, hence the name), which works well for disk caches, process pages, etc.

To keep pages with the same migrate type together, **the buddy allocator groups pages into "pageblocks," each having a migrate type assigned to it.** The allocator then tries to allocate pages from pageblocks with a type corresponding to the request.

If that's not possible, however, it will take pages from different pageblocks and may even change a pageblock's migrate type. This means that a non-movable page can be allocated from a `MIGRATE_MOVABLE` pageblock which can also result in that pageblock changing its migrate type. This is undesirable for CMA, so it **introduces a `MIGRATE_CMA` type** which has one important property: **only movable pages** can be allocated from a `MIGRATE_CMA` pageblock.

...

Important:

“... **From a device driver author's point of view, nothing should change.** CMA is integrated with the DMA subsystem, so the usual calls to the DMA API (such as `dma_alloc_coherent()`) should work as usual. In fact, device drivers should never need to call the CMA API directly, since instead of bus addresses and kernel mappings it operates on pages and page frame numbers (PFNs), and provides no mechanism for maintaining cache coherency. ...”

Kernel command-line parameters:

...

```
cma=nn[MG]@[start[MG] [-end[MG]]]
[ARM,X86,KNL]
```

Sets the size of kernel global memory area for contiguous memory allocations and optionally the placement constraint by the physical address range of memory allocations. A value of 0 disables CMA altogether. For more information, see `include/linux/dma-contiguous.h`.

...

From here: <https://www.kernel.org/doc/Documentation/cma/debugfs.txt>

The CMA debugfs interface is useful to retrieve basic information out of the different CMA areas and to test allocation/release in each of the areas.

Each CMA zone represents a directory under `<debugfs>/cma/`, indexed by the

kernel's CMA index. So the first CMA zone would be:

```
<debugfs>/cma/cma-0
```

The structure of the files created under that directory is as follows:

- [RO] base_pfn: The base PFN (Page Frame Number) of the zone.
- [RO] count: Amount of memory in the CMA area.
- [RO] order_per_bit: Order of pages represented by one bit.
- [RO] bitmap: The bitmap of page states in the zone.
- [WO] alloc: Allocate N pages from that CMA area. For example:

```
echo 5 > <debugfs>/cma/cma-2/alloc
```

would try to allocate 5 pages from the cma-2 area.

- [WO] free: Free N pages from that CMA area, similar to the above.

CMA on ARM

Implementation of consistent mapping allocation, dma_alloc_coherent() on ARM : Code View

Kernel ver: 3.10.24

File : < Linux 3.10.24 > arch/arm/include/asm/dma-mapping.h

```
...
131 #define dma_alloc_coherent(d, s, h, f) dma_alloc_attrs(d, s, h, f, NULL)
132
133 static inline void *dma_alloc_attrs(struct device *dev, size_t size,
134                                   dma_addr_t *dma_handle, gfp_t flag,
135                                   struct dma_attrs *attrs)
136 {
137     struct dma_map_ops *ops = get_dma_ops(dev);
138     void *cpu_addr;
139     BUG_ON(!ops);
140
141     cpu_addr = ops->alloc(dev, size, dma_handle, flag, attrs);
142     debug_dma_alloc_coherent(dev, size, *dma_handle, cpu_addr);
143     return cpu_addr;
144 }
...
```

File : arch/arm/include/asm/dma-mapping.h

```
...
18 static inline struct dma_map_ops *get_dma_ops(struct device *dev)
19 {
20     if (dev && dev->archdata.dma_ops)
21         return dev->archdata.dma_ops;
22     return &arm_dma_ops;
23 }
...
```

So, lets look at *arm_dma_ops* :

File : arch/arm/mm/dma-mapping.c

```
...
127 struct dma_map_ops arm_dma_ops = {
128     .alloc      = arm_dma_alloc,
129     .free       = arm_dma_free,
130     .mmap       = arm_dma_mmap,

```

```

131     .get_sgtable      = arm_dma_get_sgtable,
132     .map_page         = arm_dma_map_page,
133     .unmap_page       = arm_dma_unmap_page,
134     .map_sg           = arm_dma_map_sg,
135     .unmap_sg         = arm_dma_unmap_sg,
136     .sync_single_for_cpu = arm_dma_sync_single_for_cpu,
137     .sync_single_for_device = arm_dma_sync_single_for_device,
138     .sync_sg_for_cpu   = arm_dma_sync_sg_for_cpu,
139     .sync_sg_for_device = arm_dma_sync_sg_for_device,
140     .set_dma_mask      = arm_dma_set_mask,
141 };
142 EXPORT_SYMBOL(arm_dma_ops);
...

```

File : [3.10.24] : arch/arm/mm/dma-mapping.c

```

...
684 /*
685  * Allocate DMA-coherent memory space and return both the kernel remapped
686  * virtual and bus address for that space.
687  */
688 void *arm_dma_alloc(struct device *dev, size_t size, dma_addr_t *handle,
689                    gfp_t gfp, struct dma_attrs *attrs)
690 {
691     pgprot_t prot = __get_dma_pgprot(attrs, pgprot_kernel);
692     void *memory;
693
694     if (dma_alloc_from_coherent(dev, size, handle, &memory))
695         return memory;
696
697     return __dma_alloc(dev, size, handle, gfp, prot, false,
698                      __builtin_return_address(0));
699 }
...
635 static void *__dma_alloc(struct device *dev, size_t size, dma_addr_t *handle,
636                          gfp_t gfp, pgprot_t prot, bool is_coherent, const void *caller)
637 {
...
669     if (is_coherent || nommu())
670         addr = __alloc_simple_buffer(dev, size, gfp, &page);
671     else if (!(gfp & __GFP_WAIT))
672         addr = __alloc_from_pool(size, &page);
673     else if (!IS_ENABLED(CONFIG_CMA))
674         addr = __alloc_remap_buffer(dev, size, gfp, prot, &page, caller);
675     else
676         addr = __alloc_from_contiguous(dev, size, prot, &page, caller);
677                                     << use CMA* ! >>
678
679     if (addr)
680         *handle = pfn_to_dma(dev, page_to_pfn(page));
681     return addr;
682 }

```

* Note: in order for CMA (Contiguous Memory Allocator) allocation to be performed, we require the architecture-specific `dev_get_cma_area(struct device *dev)` routine to be implemented.

Linux DMA: Additional Resources

Linux Kernel Doc

<https://www.kernel.org/doc/Documentation/DMA-API-HOWTO.txt>

<https://www.kernel.org/doc/Documentation/DMA-API.txt>

<https://www.kernel.org/doc/Documentation/DMA-attributes.txt>

Other

[Laurent Pinchart - mastering the dma and iommu apis | ELC 2014 \[YouTube video 52 min\]](#)

[Introduction to direct memory access](#) – (OS-agnostic; hardware-biased)

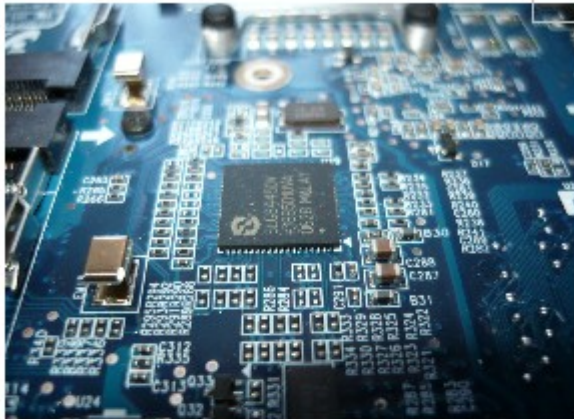
[IOMMU](#)

[Where to start learning about linux DMA / device drivers / memory allocation](#)

[Memory access ordering - an introduction](#)

<< *End Document* >>

Linux Operating System Specialized

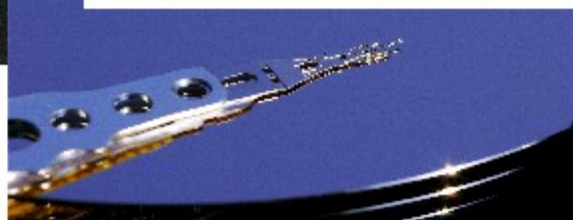
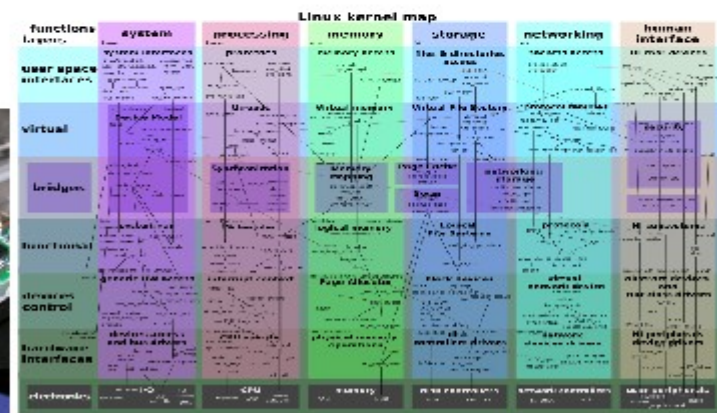
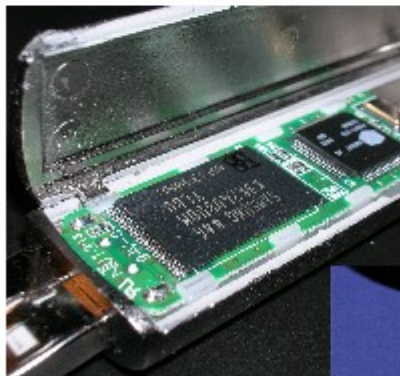


The highest quality Training on:

Linux Fundamentals, CLI and Scripting
Linux Systems Programming
Linux Kernel Internals
Linux Device Drivers
Embedded Linux
Linux Debugging Techniques
New! Linux OS for Technical Managers

Please do visit our website for details:

<http://kaiwantech.in>



<http://kaiwantech.in>