Dynamic DMA mapping using the generic device

Author: James E.J. Bottomley <James.Bottomley@HansenPartnership.com>

This document describes the DMA API. For a more gentle introduction of the API (and actual examples), see Documentation/core-api/dma-api-howto.rst.

This API is split into two pieces. Part I describes the basic API. Part II describes extensions for supporting non-consistent memory machines. Unless you know that your driver absolutely has to support non-consistent platforms (this is usually only legacy platforms) you should only use the API described in part I.

Part I - dma_API

To get the dma_API, you must #include <linux/dma-mapping.h>. This provides dma_addr_t and the interfaces described below.

A dma_addr_t can hold any valid DMA address for the platform. It can be given to a device to use as a DMA source or target. A CPU cannot reference a dma_addr_t directly because there may be translation between its physical address space and the DMA address space.

Part Ia - Using large DMA-coherent buffers

```
void *
dma_alloc_coherent(struct device *dev, size_t size,
                   dma_addr_t *dma_handle, gfp_t flag)
```

Consistent memory is memory for which a write by either the device or the processor can immediately be read by the processor or device without having to worry about caching effects. (You may however need to make sure to flush the processor's write buffers before telling devices to read that memory.)

This routine allocates a region of <size> bytes of consistent memory.

It returns a pointer to the allocated region (in the processor's virtual address space) or NULL if the allocation failed.

It also returns a <dma_handle> which may be cast to an unsigned integer the same width as the bus and given to the device as the DMA address base of the region.

Note: consistent memory can be expensive on some platforms, and the minimum allocation length may be as big as a page, so you should consolidate your requests for consistent memory as much as possible. The simplest way to do that is to use the dma_pool calls (see below).

The flag parameter (dma_alloc_coherent() only) allows the caller to specify the `GFP_` flags (see kmalloc()) for the allocation (the implementation may choose to ignore flags that affect the location of the returned memory, like GFP_DMA).

```
void
dma_free_coherent(struct device *dev, size_t size, void *cpu_addr,
                  dma_addr_t dma_handle)
```

Free a region of consistent memory you previously allocated. dev, size and dma_handle must all be the same as those passed into dma_alloc_coherent(). cpu_addr must be the virtual address returned by the dma_alloc_coherent().

Note that unlike their sibling allocation calls, these routines may only be called with IRQs enabled.

Part Ib - Using small DMA-coherent buffers

To get this part of the dma_API, you must #include <linux/dmapool.h>

Many drivers need lots of small DMA-coherent memory regions for DMA descriptors or I/O buffers. Rather than allocating in units of a page or more using dma_alloc_coherent(), you can use DMA pools. These work much like a struct kmem_cache, except that they use the DMA-coherent allocator, not __get_free_pages(). Also, they understand common hardware constraints for alignment, like queue heads needing to be aligned on N-byte boundaries.

```
struct dma_pool *
dma_pool_create(const char *name, struct device *dev,
                size_t size, size_t align, size_t alloc);
```

dma_pool_create() initializes a pool of DMA-coherent buffers for use with a given device. It must be called in a context which can sleep.

The "name" is for diagnostics (like a struct kmem_cache name); dev and size are like what you'd pass to dma_alloc_coherent(). 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.

```
void *
dma_pool_zalloc(struct dma_pool *pool, gfp_t mem_flags,
                dma_addr_t *handle)
```

Wraps dma_pool_alloc() and also zeroes the returned memory if the allocation attempt succeeded.

```
void *
dma_pool_alloc(struct dma_pool *pool, gfp_t gfp_flags,
               dma_addr_t *dma_handle);
```

This allocates memory from the pool; the returned memory will meet the size and alignment requirements specified at creation time. Pass GFP_ATOMIC to prevent blocking, or if it's permitted (not in_interrupt, not holding SMP locks), pass GFP_KERNEL to allow blocking. Like dma_alloc_coherent(), this returns two values: an address usable by the CPU, and the DMA address usable by the pool's device.

```
void
dma_pool_free(struct dma_pool *pool, void *vaddr,
              dma_addr_t addr);
```

This puts memory back into the pool. The pool is what was passed to dma_pool_alloc(); the CPU (vaddr) and DMA addresses are what were returned when that routine allocated the memory being freed.

```
void
dma_pool_destroy(struct dma_pool *pool);
```

dma_pool_destroy() frees the resources of the pool. It must be called in a context which can sleep. Make sure you've freed all allocated memory back to the pool before you destroy it.

Part Ic - DMA addressing limitations

```
int
dma_set_mask_and_coherent(struct device *dev, u64 mask)
```

Checks to see if the mask is possible and updates the device streaming and coherent DMA mask parameters if it is.

Returns: 0 if successful and a negative error if not.

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

Checks to see if the mask is possible and updates the device parameters if it is.

Returns: 0 if successful and a negative error if not.

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

Checks to see if the mask is possible and updates the device parameters if it is.

Returns: 0 if successful and a negative error if not.

```
u64
dma_get_required_mask(struct device *dev)
```

This API returns the mask that the platform requires to operate efficiently. Usually this means the returned mask is the minimum required to cover all of memory. Examining the required mask gives drivers with variable descriptor sizes the opportunity to use smaller descriptors as necessary.

Requesting the required mask does not alter the current mask. If you wish to take advantage of it, you should issue a dma_set_mask() call to set the mask to the value returned.

```
size_t
dma_max_mapping_size(struct device *dev);
```

Returns the maximum size of a mapping for the device. The size parameter of the mapping functions like dma_map_single(), dma_map_page() and others should not be larger than the returned value.

```
bool
dma_need_sync(struct device *dev, dma_addr_t dma_addr);
```

Returns %true if dma_sync_single_for_{device,cpu} calls are required to transfer memory ownership. Returns %false if those calls can be skipped.

```
unsigned long
dma_get_merge_boundary(struct device *dev);
```

Returns the DMA merge boundary. If the device cannot merge any the DMA address segments, the function returns 0.

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 DMA address of the memory.

The direction for both APIs 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

> **Note**
>
> Not all memory regions in a machine can be mapped by this API. Further, contiguous 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, memory to be mapped by this API should be obtained from sources which guarantee it to be physically contiguous (like kmalloc).
>
> Further, the DMA address of the memory must be within the dma_mask of the device (the dma_mask is a bit mask of the addressable region for the device, i.e., if the DMA address of the memory ANDed with the dma_mask is still equal to the DMA address, then the device can perform DMA to the memory). To ensure that the memory allocated by kmalloc is within the dma_mask, the driver may specify various platform-dependent flags to restrict the DMA address range of the allocation (e.g., on x86, GFP_DMA guarantees to be within the first 16MB of available DMA addresses, 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 maps an I/O DMA address to a physical memory address). However, to be portable, device driver writers may *not* assume that such an IOMMU exists.

> **Warning**
>
> 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 device. 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.

```
dma_addr_t
dma_map_page(struct device *dev, struct page *page,
             unsigned long offset, size_t size,
             enum dma_data_direction direction)

void
dma_unmap_page(struct device *dev, dma_addr_t dma_address, size_t size,
               enum dma_data_direction direction)
```

API for mapping and unmapping for pages. All the notes and warnings for the other mapping APIs apply here. Also, although the

<offset> and <size> parameters are provided to do partial page mapping, it is recommended that you never use these unless you really know what the cache width is.

```
dma_addr_t
dma_map_resource(struct device *dev, phys_addr_t phys_addr, size_t size,
                 enum dma_data_direction dir, unsigned long attrs)

void
dma_unmap_resource(struct device *dev, dma_addr_t addr, size_t size,
                   enum dma_data_direction dir, unsigned long attrs)
```

API for mapping and unmapping for MMIO resources. All the notes and warnings for the other mapping APIs apply here. The API should only be used to map device MMIO resources, mapping of RAM is not permitted.

```
int
dma_mapping_error(struct device *dev, dma_addr_t dma_addr)
```

In some circumstances dma_map_single(), dma_map_page() and dma_map_resource() will fail to create a mapping. A driver can check for these errors by testing the returned DMA address with dma_mapping_error(). A non-zero return value means the mapping could not be created and the driver should take appropriate action (e.g. reduce current DMA mapping usage or delay and try again later).

```
int
dma_map_sg(struct device *dev, struct scatterlist *sg,
           int nents, enum dma_data_direction direction)
```

Returns: the number of DMA address segments mapped (this may be shorter than <nents> passed in if some elements of the scatter/gather list are physically or virtually adjacent and an IOMMU maps them with a single entry).

Please note that the sg cannot be mapped again if it has been mapped once. The mapping process is allowed to destroy information in the sg.

As with the other mapping interfaces, dma_map_sg() can fail. When it does, 0 is returned and a driver must take appropriate action. It is critical that the driver do something, in the case of a block driver aborting the request or even oopsing is better than doing nothing and corrupting the filesystem.

With scatterlists, you use the resulting mapping like this:

```
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. with an IOMMU, or if several pages just happen to be physically contiguous) 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.

```
void
dma_unmap_sg(struct device *dev, struct scatterlist *sg,
             int nents, enum dma_data_direction direction)
```

Unmap the previously mapped scatter/gather list. All the parameters must be the same as those and passed in to the scatter/gather mapping API.

Note: <nents> must be the number you passed in, *not* the number of DMA address entries returned.

```
void
dma_sync_single_for_cpu(struct device *dev, dma_addr_t dma_handle,
                        size_t size,
                        enum dma_data_direction direction)

void
dma_sync_single_for_device(struct device *dev, dma_addr_t dma_handle,
                           size_t size,
                           enum dma_data_direction direction)

void
dma_sync_sg_for_cpu(struct device *dev, struct scatterlist *sg,
                    int nents,
                    enum dma_data_direction direction)

void
dma_sync_sg_for_device(struct device *dev, struct scatterlist *sg,
```

```
                        int nents,
                        enum dma_data_direction direction)
```

Synchronise a single contiguous or scatter/gather mapping for the CPU and device. With the sync_sg API, all the parameters must be the same as those passed into the single mapping API. With the sync_single API, you can use dma_handle and size parameters that aren't identical to those passed into the single mapping API to do a partial sync.

> **Note**
>
> You must do this:
>
> - Before reading values that have been written by DMA from the device (use the DMA_FROM_DEVICE direction)
> - After writing values that will be written to the device using DMA (use the DMA_TO_DEVICE) direction
> - before *and* after handing memory to the device if the memory is DMA_BIDIRECTIONAL

See also dma_map_single().

```
dma_addr_t
dma_map_single_attrs(struct device *dev, void *cpu_addr, size_t size,
                     enum dma_data_direction dir,
                     unsigned long attrs)

void
dma_unmap_single_attrs(struct device *dev, dma_addr_t dma_addr,
                       size_t size, enum dma_data_direction dir,
                       unsigned long attrs)

int
dma_map_sg_attrs(struct device *dev, struct scatterlist *sgl,
                 int nents, enum dma_data_direction dir,
                 unsigned long attrs)

void
dma_unmap_sg_attrs(struct device *dev, struct scatterlist *sgl,
                   int nents, enum dma_data_direction dir,
                   unsigned long attrs)
```

The four functions above are just like the counterpart functions without the _attrs suffixes, except that they pass an optional dma_attrs.

The interpretation of DMA attributes is architecture-specific, and each attribute should be documented in Documentation/core-api/dma-attributes.rst.

If dma_attrs are 0, the semantics of each of these functions is identical to those of the corresponding function without the _attrs suffix. As a result dma_map_single_attrs() can generally replace dma_map_single(), etc.

As an example of the use of the `*_attrs` functions, here's how you could pass an attribute DMA_ATTR_FOO when mapping memory for DMA:

```
#include <linux/dma-mapping.h>
/* DMA_ATTR_FOO should be defined in linux/dma-mapping.h and
* documented in Documentation/core-api/dma-attributes.rst */
...

        unsigned long attr;
        attr |= DMA_ATTR_FOO;
        ....
        n = dma_map_sg_attrs(dev, sg, nents, DMA_TO_DEVICE, attr);
        ....
```

Architectures that care about DMA_ATTR_FOO would check for its presence in their implementations of the mapping and unmapping routines, e.g.::

```
void whizco_dma_map_sg_attrs(struct device *dev, dma_addr_t dma_addr,
                             size_t size, enum dma_data_direction dir,
                             unsigned long attrs)
{
        ....
        if (attrs & DMA_ATTR_FOO)
                /* twizzle the frobnozzle */
        ....
}
```

Part II - Non-coherent DMA allocations

These APIs allow to allocate pages that are guaranteed to be DMA addressable by the passed in device, but which need explicit

management of memory ownership for the kernel vs the device.

If you don't understand how cache line coherency works between a processor and an I/O device, you should not be using this part of the API.

```
struct page *
dma_alloc_pages(struct device *dev, size_t size, dma_addr_t *dma_handle,
                enum dma_data_direction dir, gfp_t gfp)
```

This routine allocates a region of <size> bytes of non-coherent memory. It returns a pointer to first struct page for the region, or NULL if the allocation failed. The resulting struct page can be used for everything a struct page is suitable for.

It also returns a <dma_handle> which may be cast to an unsigned integer the same width as the bus and given to the device as the DMA address base of the region.

The dir parameter specified if data is read and/or written by the device, see dma_map_single() for details.

The gfp parameter allows the caller to specify the `GFP_` flags (see kmalloc()) for the allocation, but rejects flags used to specify a memory zone such as GFP_DMA or GFP_HIGHMEM.

Before giving the memory to the device, dma_sync_single_for_device() needs to be called, and before reading memory written by the device, dma_sync_single_for_cpu(), just like for streaming DMA mappings that are reused.

```
void
dma_free_pages(struct device *dev, size_t size, struct page *page,
                dma_addr_t dma_handle, enum dma_data_direction dir)
```

Free a region of memory previously allocated using dma_alloc_pages(). dev, size, dma_handle and dir must all be the same as those passed into dma_alloc_pages(). page must be the pointer returned by dma_alloc_pages().

```
int
dma_mmap_pages(struct device *dev, struct vm_area_struct *vma,
               size_t size, struct page *page)
```

Map an allocation returned from dma_alloc_pages() into a user address space. dev and size must be the same as those passed into dma_alloc_pages(). page must be the pointer returned by dma_alloc_pages().

```
void *
dma_alloc_noncoherent(struct device *dev, size_t size,
                dma_addr_t *dma_handle, enum dma_data_direction dir,
                gfp_t gfp)
```

This routine is a convenient wrapper around dma_alloc_pages that returns the kernel virtual address for the allocated memory instead of the page structure.

```
void
dma_free_noncoherent(struct device *dev, size_t size, void *cpu_addr,
                dma_addr_t dma_handle, enum dma_data_direction dir)
```

Free a region of memory previously allocated using dma_alloc_noncoherent(). dev, size, dma_handle and dir must all be the same as those passed into dma_alloc_noncoherent(). cpu_addr must be the virtual address returned by dma_alloc_noncoherent().

```
struct sg_table *
dma_alloc_noncontiguous(struct device *dev, size_t size,
                        enum dma_data_direction dir, gfp_t gfp,
                        unsigned long attrs);
```

This routine allocates <size> bytes of non-coherent and possibly non-contiguous memory. It returns a pointer to struct sg_table that describes the allocated and DMA mapped memory, or NULL if the allocation failed. The resulting memory can be used for struct page mapped into a scatterlist are suitable for.

The return sg_table is guaranteed to have 1 single DMA mapped segment as indicated by sgt->nents, but it might have multiple CPU side segments as indicated by sgt->orig_nents.

The dir parameter specified if data is read and/or written by the device, see dma_map_single() for details.

The gfp parameter allows the caller to specify the `GFP_` flags (see kmalloc()) for the allocation, but rejects flags used to specify a memory zone such as GFP_DMA or GFP_HIGHMEM.

The attrs argument must be either 0 or DMA_ATTR_ALLOC_SINGLE_PAGES.

Before giving the memory to the device, dma_sync_sgtable_for_device() needs to be called, and before reading memory written by the device, dma_sync_sgtable_for_cpu(), just like for streaming DMA mappings that are reused.

```
void
dma_free_noncontiguous(struct device *dev, size_t size,
                       struct sg_table *sgt,
                       enum dma_data_direction dir)
```

Free memory previously allocated using dma_alloc_noncontiguous(). dev, size, and dir must all be the same as those passed into dma_alloc_noncontiguous(). sgt must be the pointer returned by dma_alloc_noncontiguous().

```
void *
dma_vmap_noncontiguous(struct device *dev, size_t size,
        struct sg_table *sgt)
```

Return a contiguous kernel mapping for an allocation returned from dma_alloc_noncontiguous(). dev and size must be the same as those passed into dma_alloc_noncontiguous(). sgt must be the pointer returned by dma_alloc_noncontiguous().

Once a non-contiguous allocation is mapped using this function, the flush_kernel_vmap_range() and invalidate_kernel_vmap_range() APIs must be used to manage the coherency between the kernel mapping, the device and user space mappings (if any).

```
void
dma_vunmap_noncontiguous(struct device *dev, void *vaddr)
```

Unmap a kernel mapping returned by dma_vmap_noncontiguous(). dev must be the same the one passed into dma_alloc_noncontiguous(). vaddr must be the pointer returned by dma_vmap_noncontiguous().

```
int
dma_mmap_noncontiguous(struct device *dev, struct vm_area_struct *vma,
                       size_t size, struct sg_table *sgt)
```

Map an allocation returned from dma_alloc_noncontiguous() into a user address space. dev and size must be the same as those passed into dma_alloc_noncontiguous(). sgt must be the pointer returned by dma_alloc_noncontiguous().

```
int
dma_get_cache_alignment(void)
```

Returns the processor cache alignment. This is the absolute minimum alignment *and* width that you must observe when either mapping memory or doing partial flushes.

> **Note**
>
> This API may return a number *larger* than the actual cache line, but it will guarantee that one or more cache lines fit exactly into the width returned by this call. It will also always be a power of two for easy alignment.

Part III - Debug drivers use of the DMA-API

The DMA-API as described above has some constraints. DMA addresses must be released with the corresponding function with the same size for example. With the advent of hardware IOMMUs it becomes more and more important that drivers do not violate those constraints. In the worst case such a violation can result in data corruption up to destroyed filesystems.

To debug drivers and find bugs in the usage of the DMA-API checking code can be compiled into the kernel which will tell the developer about those violations. If your architecture supports it you can select the "Enable debugging of DMA-API usage" option in your kernel configuration. Enabling this option has a performance impact. Do not enable it in production kernels.

If you boot the resulting kernel will contain code which does some bookkeeping about what DMA memory was allocated for which device. If this code detects an error it prints a warning message with some details into your kernel log. An example warning message may look like this:

```
WARNING: at /data2/repos/linux-2.6-iommu/lib/dma-debug.c:448
        check_unmap+0x203/0x490()
Hardware name:
forcedeth 0000:00:08.0: DMA-API: device driver frees DMA memory with wrong
        function [device address=0x00000000640444be] [size=66 bytes] [mapped as
single] [unmapped as page]
Modules linked in: nfsd exportfs bridge stp llc r8169
Pid: 0, comm: swapper Tainted: G        W  2.6.28-dmatest-09289-g8bb99c0 #1
Call Trace:
<IRQ>  [<ffffffff80240b22>] warn_slowpath+0xf2/0x130
[<ffffffff80647b70>] _spin_unlock+0x10/0x30
[<ffffffff80537e75>] usb_hcd_link_urb_to_ep+0x75/0xc0
[<ffffffff80647c22>] _spin_unlock_irqrestore+0x12/0x40
[<ffffffff8055347f>] ohci_urb_enqueue+0x19f/0x7c0
[<ffffffff80252f96>] queue_work+0x56/0x60
[<ffffffff80237e10>] enqueue_task_fair+0x20/0x50
[<ffffffff80539279>] usb_hcd_submit_urb+0x379/0xbc0
[<ffffffff803b78c3>] cpumask_next_and+0x23/0x40
[<ffffffff80235177>] find_busiest_group+0x207/0x8a0
[<ffffffff8064784f>] _spin_lock_irqsave+0x1f/0x50
[<ffffffff803c7ea3>] check_unmap+0x203/0x490
[<ffffffff803c8259>] debug_dma_unmap_page+0x49/0x50
[<ffffffff80485f26>] nv_tx_done_optimized+0xc6/0x2c0
[<ffffffff80486c13>] nv_nic_irq_optimized+0x73/0x2b0
[<ffffffff8026df84>] handle_IRQ_event+0x34/0x70
[<ffffffff8026ffe9>] handle_edge_irq+0xc9/0x150
[<ffffffff8020e3ab>] do_IRQ+0xcb/0x1c0
[<ffffffff8020c093>] ret_from_intr+0x0/0xa
<EOI> <4>---[ end trace f6435a98e2a38c0e ]---
```

The driver developer can find the driver and the device including a stacktrace of the DMA-API call which caused this warning.

Per default only the first error will result in a warning message. All other errors will only silently counted. This limitation exist to prevent the code from flooding your kernel log. To support debugging a device driver this can be disabled via debugfs. See the debugfs interface documentation below for details.

The debugfs directory for the DMA-API debugging code is called dma-api/. In this directory the following files can currently be found:

dma-api/all_errors	This file contains a numeric value. If this value is not equal to zero the debugging code will print a warning for every error it finds into the kernel log. Be careful with this option, as it can easily flood your logs.
dma-api/disabled	This read-only file contains the character 'Y' if the debugging code is disabled. This can happen when it runs out of memory or if it was disabled at boot time
dma-api/dump	This read-only file contains current DMA mappings.
dma-api/error_count	This file is read-only and shows the total numbers of errors found.
dma-api/num_errors	The number in this file shows how many warnings will be printed to the kernel log before it stops. This number is initialized to one at system boot and be set by writing into this file
dma-api/min_free_entries	This read-only file can be read to get the minimum number of free dma_debug_entries the allocator has ever seen. If this value goes down to zero the code will attempt to increase nr_total_entries to compensate.
dma-api/num_free_entries	The current number of free dma_debug_entries in the allocator.
dma-api/nr_total_entries	The total number of dma_debug_entries in the allocator, both free and used.
dma-api/driver_filter	You can write a name of a driver into this file to limit the debug output to requests from that particular driver. Write an empty string to that file to disable the filter and see all errors again.

If you have this code compiled into your kernel it will be enabled by default. If you want to boot without the bookkeeping anyway you can provide 'dma_debug=off' as a boot parameter. This will disable DMA-API debugging. Notice that you can not enable it again at runtime. You have to reboot to do so.

If you want to see debug messages only for a special device driver you can specify the dma_debug_driver=<drivername> parameter. This will enable the driver filter at boot time. The debug code will only print errors for that driver afterwards. This filter can be disabled or changed later using debugfs.

When the code disables itself at runtime this is most likely because it ran out of dma_debug_entries and was unable to allocate more on-demand. 65536 entries are preallocated at boot - if this is too low for you boot with 'dma_debug_entries=<your_desired_number>' to overwrite the default. Note that the code allocates entries in batches, so the exact number of preallocated entries may be greater than the actual number requested. The code will print to the kernel log each time it has dynamically allocated as many entries as were initially preallocated. This is to indicate that a larger preallocation size may be appropriate, or if it happens continually that a driver may be leaking mappings.

```
void
debug_dma_mapping_error(struct device *dev, dma_addr_t dma_addr);
```

dma-debug interface debug_dma_mapping_error() to debug drivers that fail to check DMA mapping errors on addresses returned by dma_map_single() and dma_map_page() interfaces. This interface clears a flag set by debug_dma_map_page() to indicate that dma_mapping_error() has been called by the driver. When driver does unmap, debug_dma_unmap() checks the flag and if this flag is still set, prints warning message that includes call trace that leads up to the unmap. This interface can be called from dma_mapping_error() routines to enable DMA mapping error check debugging.