Mastering the DMA and IOMMU APIs

Embedded Linux Conference Europe 2014
Düsseldorf

Laurent Pinchart
laurent.pinchart@ideasonboard.com

RENESAS

DMA != DMA

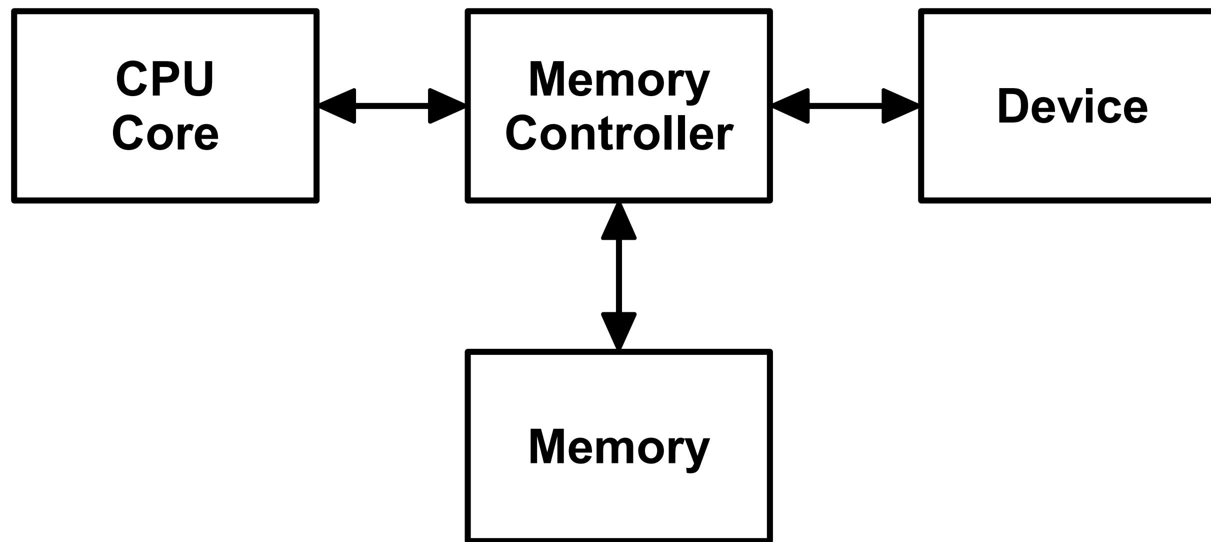
DMA != DMA

(mapping) (engine)

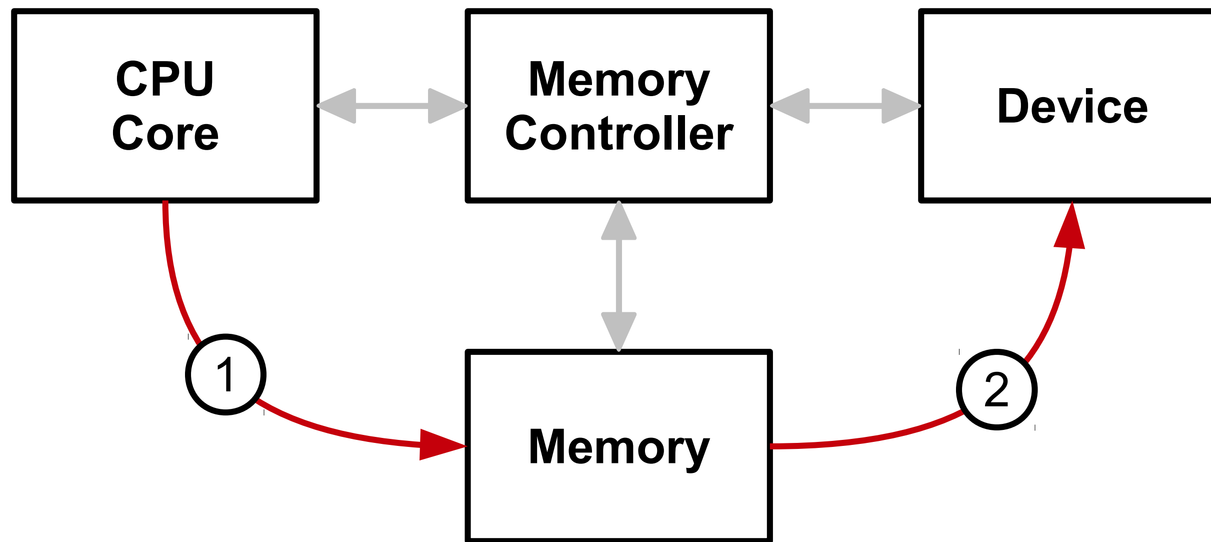
The topic we will focus on is how to manage system memory used for DMA.

This presentation will not discuss the DMA engine API, nor will it address how to control DMA operations from a device point of view.

Memory Access

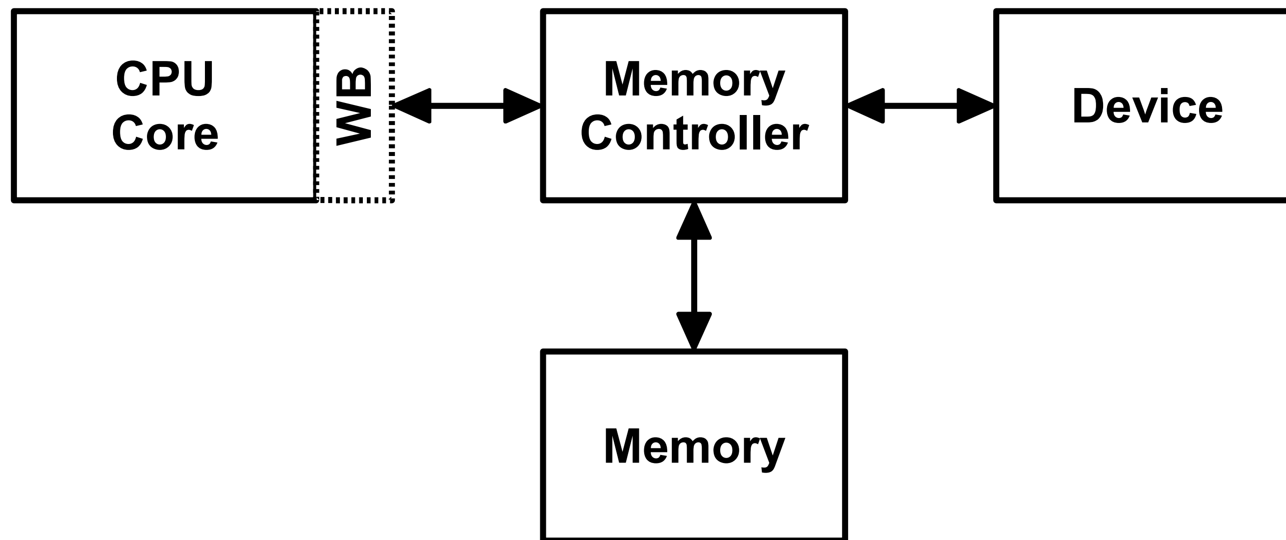


Simple Case

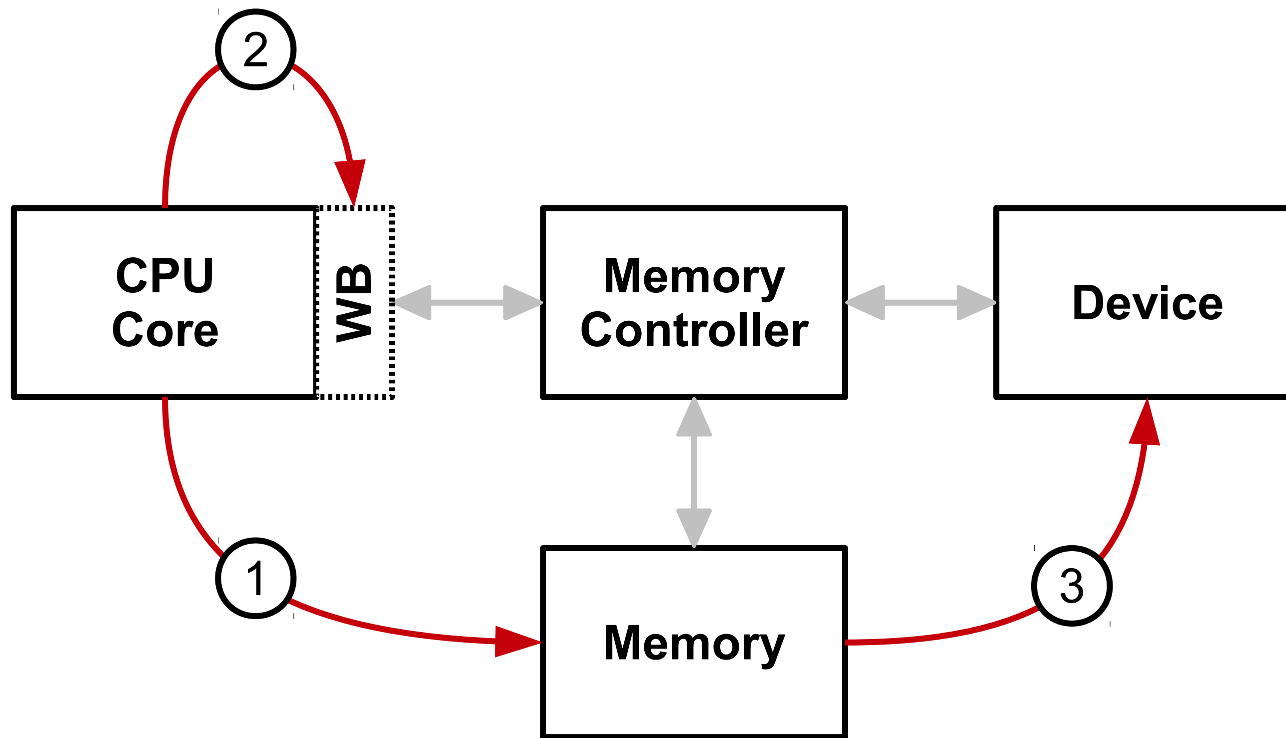


(1) CPU writes to memory
(2) Device reads from memory

Simple Case

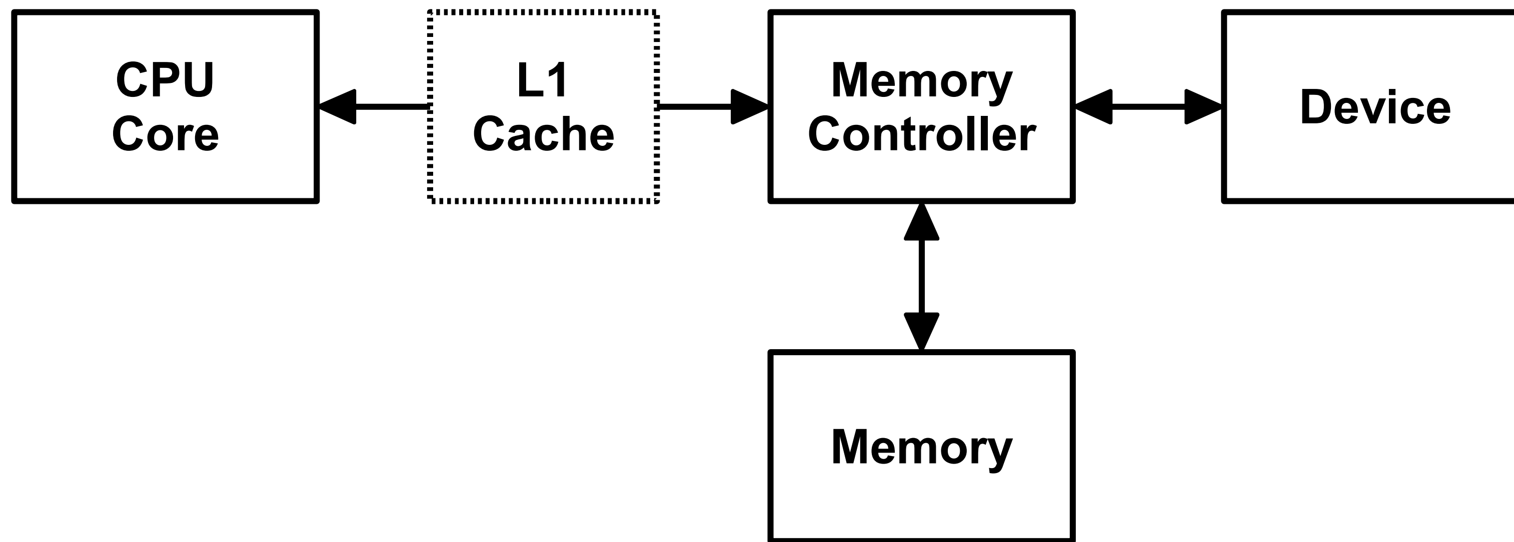


Write Buffer

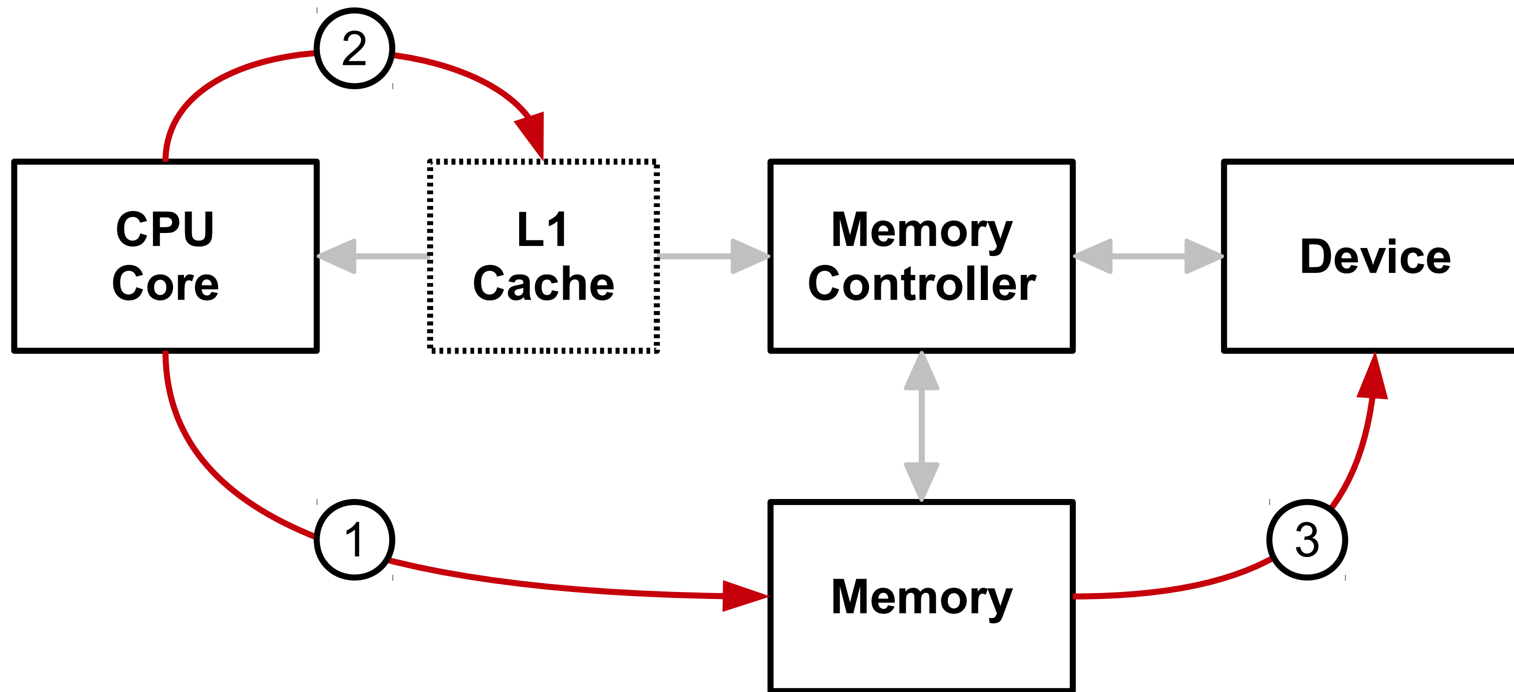


(1) CPU writes to memory
(2) CPU flushes its write buffers
(3) Device reads from memory

Write Buffer

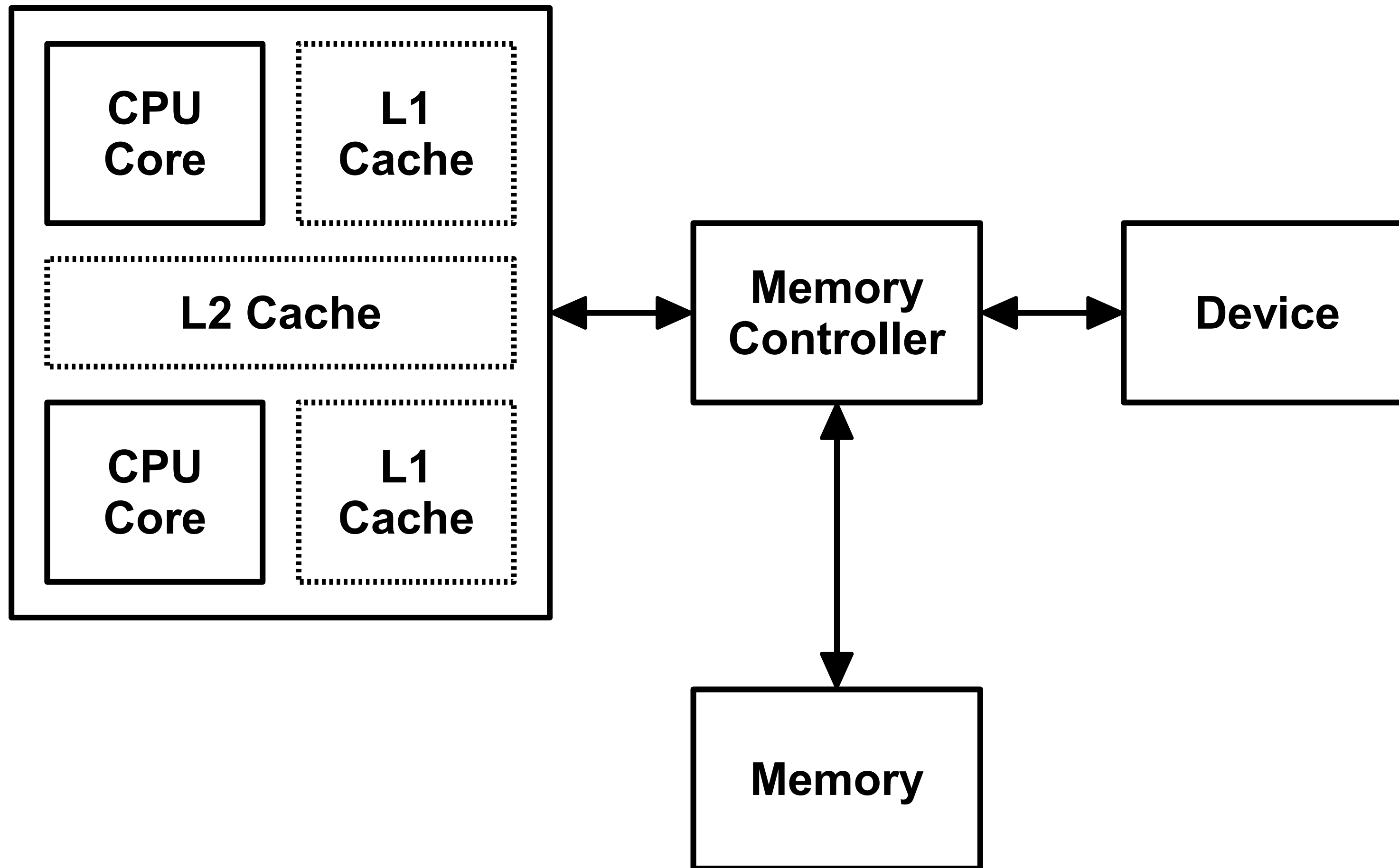


L1 Cache



(1) CPU writes to memory
(2) CPU cleans L1 cache
(3) Device reads from memory

L1 Cache



IDEAS ON BOARD

L2 Cache

L2 Cache

CPU Core
L1 Cache
L2 Cache
CPU Core
L1 Cache
Memory Controller
Device
Memory
1
2
3
4

(1) CPU writes to memory
(2) CPU cleans L1 cache
(3) CPU cleans L2 cache
(4) Device reads from memory

CPU Core

L1 Cache

L2 Cache

CPU Core

L1 Cache

Cache Coherent Interconnect

Device

Memory Controller

Memory

Cache Coherent Interconnect

CPU Core

L1 Cache

L2 Cache

CPU Core

L1 Cache

Cache Coherent Interconnect

Device

Memory Controller

Memory

2

1

(1) CPU writes to memory
(2) Device reads from memory

Cache Coherent Interconnect

CPU Core
L1 Cache
L2 Cache
CPU Core
L1 Cache
Cache Coherent Interconnect
IOMMU
Device
Memory Controller
Memory

IOMMU

CPU Core

L1 Cache

L2 Cache

CPU Core

L1 Cache

Cache Coherent Interconnect

IOMMU

Device

Memory Controller

Memory

(1) CPU writes to memory
(2) CPU programs the IOMMU
(3) Device reads from memory

IOMMU

Ethernet
SATA
USB
DSP
PCIe
core
core
L2 cache
core
core
Multi-layer reordering bus matrix
graphics accelerator
graphics controller
Southbridge
DMC
DDR3
DDR3

Even More Complex

Mali-T604

LCD

Video decode

DDR3/ LPDDR2

NIC-400

MMU-400

MMU-400

DMC-400

Cache Coherent Interconnect CCI-400

Cortex-A15 MPCore

Cortex-A15 MPCore

NIC-400

System

Outer-shareable

Non-shareable

Inner-shareable

Even More Complex

Memory Mappings

- Fully Coherent

Coherent (or 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.

Consistent memory can be expensive on some platforms, and the minimum allocation length may be as big as a page.

- Write Combining

Writes to the mapping may be buffered to improve performance. You need to make sure to flush the processor's write buffers before telling devices to read that memory. This memory type is typically used for (but not restricted to) graphics memory.

- Weakly Ordered

Reads and writes to the mapping may be weakly ordered, that is that reads and writes may pass each other. Not all architectures support non-cached weakly ordered mappings.

Memory Mapping Types

- Non-Coherent

This memory mapping type permits speculative reads, merging of accesses and (if interrupted by an exception) repeating of writes without side effects. Accesses to non-coherent memory can always be buffered, and in most situations they are also cached (but they can be configured to be uncached). There is no implicit ordering of non-coherent memory accesses. When not explicitly restricted, the only limit to how out-of-order non-dependent accesses can be is the processor's ability to hold multiple live transactions.

When using non-coherent memory mappings you are guaranteeing to the platform that you have all the correct and necessary sync points for this memory in the driver.

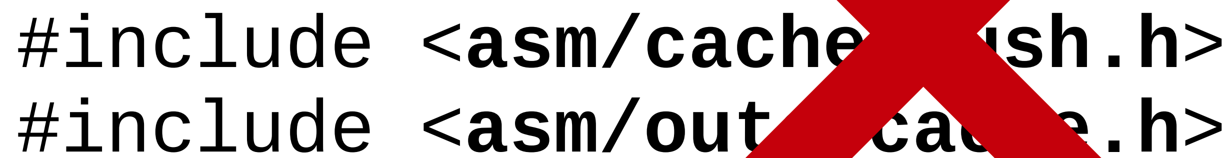
Cache Management

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

Cache Management API

```
#include <asm/cacheflush.h>
#include <asm/outercache.h>
```

Cache Management API



Cache Management API

Cache management operations are architecture and device specific.

To remain portable, device drivers must not use the cache handling API directly.

Conclusion

DMA Mapping API

- Allocate memory suitable for DMA operations
- Map DMA memory to devices
- Map DMA memory to userspace
- Synchronize memory between CPU and device domains

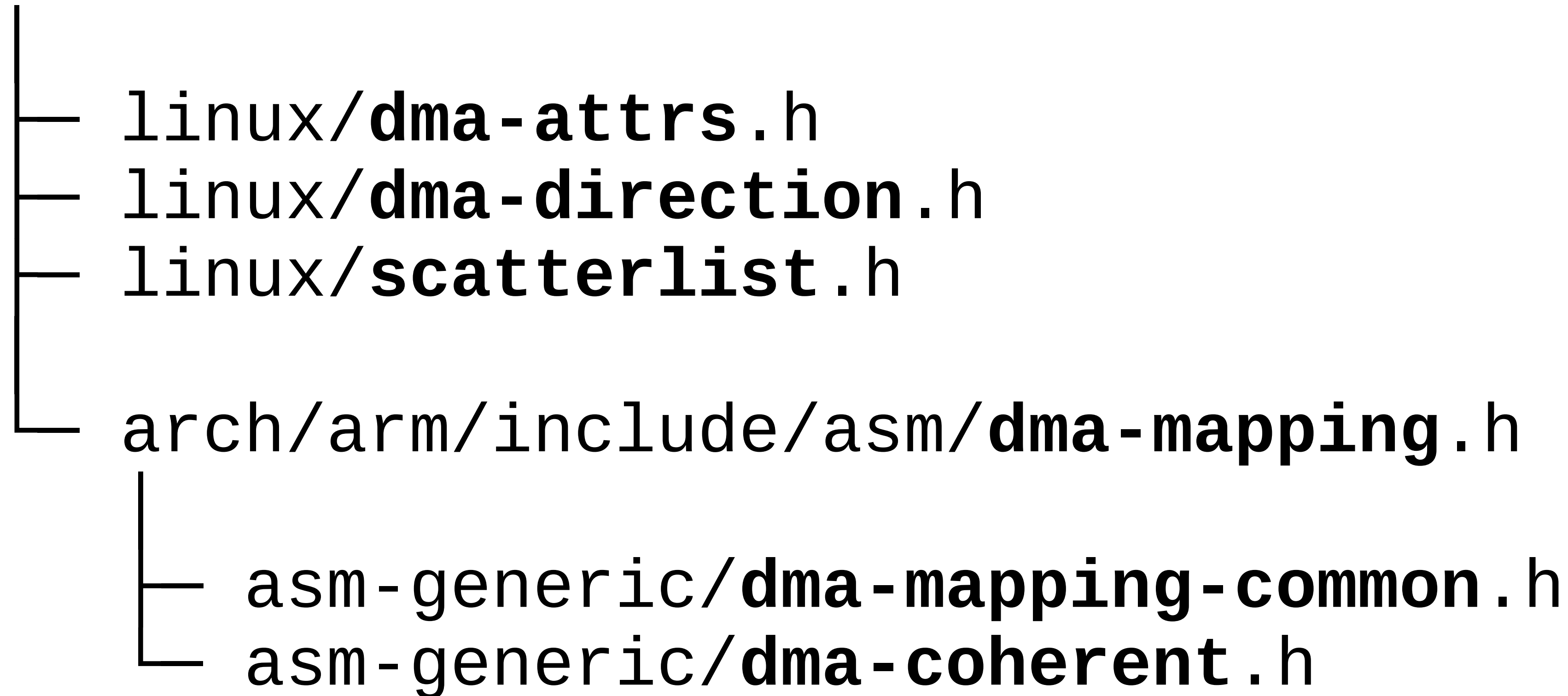
DMA Mapping API

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

DMA Mapping API

```
linux/dma-mapping.h
│
├── linux/dma-attrs.h
├── linux/dma-direction.h
├── linux/scatterlist.h
│
#ifdef CONFIG_HAS_DMA
└── asm/dma-mapping.h
#else
└── asm-generic/dma-mapping-broken.h
#endif
```

DMA Mapping API



IDEAS ON BOARD

DMA Mapping API (ARM)

DMA Coherent Mapping

```
/* asm-generic/dma-mapping.h */

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

This routine allocates a region of @size bytes of coherent memory. It also returns a @dma_handle which may be cast to an unsigned integer the same width as the bus and used as the device address base of the region.

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

Note: coherent 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.

Coherent Allocation

```
/* asm-generic/dma-mapping.h */

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

Free memory previously allocated by dma_free_coherent(). Unlike with CPU memory allocators, calling this function with a NULL cpu_addr is not safe.

Coherent Allocation

```
/* asm-generic/dma-mapping.h */

void *
dma_alloc_attrs(struct device *dev, size_t size,
                dma_addr_t *dma_handle, gfp_t flag,
                struct dma_attrs *attrs);

void
dma_free_attrs(struct device *dev, size_t size,
               void *cpu_addr, dma_addr_t dma_handle,
               struct dma_attrs *attrs);
```

Those two functions extend the coherent memory allocation API by allowing the caller to specify attributes for the allocated memory. When @attrs is NULL the behaviour is identical to the dma_*_coherent() functions.

Attribute-Based Allocation

- Allocation Attributes
  - DMA_ATTR_WRITE_COMBINE
  - DMA_ATTR_WEAK_ORDERING
  - DMA_ATTR_NON_CONSISTENT
  - DMA_ATTR_WRITE_BARRIER
  - DMA_ATTR_FORCE_CONTIGUOUS
- Allocation and mmap Attributes
  - DMA_ATTR_NO_KERNEL_MAPPING
- Map Attributes
  - DMA_ATTR_SKIP_CPU_SYNC

All attributes are optional. An architecture that doesn't implement an attribute ignores it and exhibit default behaviour.

(See Documentation/DMA-attributes.txt)

DMA Mapping Attributes

- DMA_ATTR_WRITE_COMBINE

DMA_ATTR_WRITE_COMBINE specifies that writes to the mapping may be buffered to improve performance.

This attribute is only supported by the ARM and ARM64 architectures.

Additionally, the AVR32 architecture doesn't implement the attribute-based allocation API but supports write combine allocation with the dma_alloc_writecombine() and dma_free_writecombine() functions.

- DMA_ATTR_WEAK_ORDERING

DMA_ATTR_WEAK_ORDERING specifies that reads and writes to the mapping may be weakly ordered, that is that reads and writes may pass each other.

This attribute is only supported by the CELL architecture (and isn't used by any driver).

- DMA_ATTR_NON_CONSISTENT

DMA_ATTR_NON_CONSISTENT lets the platform to choose to return either consistent or non-consistent memory as it sees fit. By using this API, you are guaranteeing to the platform that you have all the correct and necessary sync points for this memory in the driver.

Only the OpenRISC architecture returns non-consistent memory in response to this attribute. The ARC, MIPS and PARISC architectures don't support this attribute but offer dedicated dma_alloc_noncoherent() and dma_free_noncoherent() functions for the same purpose.

- DMA_ATTR_WRITE_BARRIER

DMA_ATTR_WRITE_BARRIER is a (write) barrier attribute for DMA. DMA to a memory region with the DMA_ATTR_WRITE_BARRIER attribute forces all pending DMA writes to complete, and thus provides a mechanism to strictly order DMA from a device across all intervening buses and bridges. This barrier is not specific to a particular type of interconnect, it applies to the system as a whole, and so its implementation must account for the idiosyncrasies of the system all the way from the DMA device to memory.

As an example of a situation where DMA_ATTR_WRITE_BARRIER would be useful, suppose that a device does a DMA write to indicate that data is ready and available in memory. The DMA of the "completion indication" could race with data DMA. Mapping the memory used for completion indications with DMA_ATTR_WRITE_BARRIER would prevent the race.

This attribute is only implemented by the SGI SN2 (IA64) subarchitecture.

- DMA_ATTR_FORCE_CONTIGUOUS

By default the DMA-mapping subsystem is allowed to assemble the buffer allocated by the dma_alloc_attrs() function from individual pages if it can be mapped contiguously into device DMA address space. By specifying this attribute the allocated buffer is forced to be contiguous also in physical memory.

This attribute is only supported by the ARM architecture.

- DMA_ATTR_NO_KERNEL_MAPPING

DMA_ATTR_NO_KERNEL_MAPPING lets the platform to avoid creating a kernel virtual mapping for the allocated buffer. On some architectures creating such mapping is non-trivial task and consumes very limited resources (like kernel virtual address space or dma consistent address space). Buffers allocated with this attribute can be only passed to user space by calling dma_mmap_attrs(). By using this API, you are guaranteeing that you won't dereference the pointer returned by dma_alloc_attr(). You can treat it as a cookie that must be passed to dma_mmap_attrs() and dma_free_attrs(). Make sure that both of these also get this attribute set on each call.

This attribute is only supported by the ARM architecture.

- DMA_ATTR_SKIP_CPU_SYNC

When a buffer is shared between multiple devices one mapping must be created separately for each device. This is usually performed by calling the DMA mapping functions more than once for the given buffer. The first call transfers buffer ownership from CPU domain to device domain, which synchronizes CPU caches for the given region. However, subsequent calls to dma_map_*() for other devices will perform exactly the same potentially expensive synchronization operation on the CPU cache.

DMA_ATTR_SKIP_CPU_SYNC allows platform code to skip synchronization of the CPU cache for the given buffer assuming that it has been already transferred to “device” domain. This is highly recommended but must be used with care. This attribute can be also used for the DMA mapping functions to force buffer to stay in device domain.

This attribute is only supported by the ARM architecture.

DMA Mask

```
/* asm/dma-mapping.h */

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

/* linux/dma-mapping.h */

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

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

DMA Mask

```
/* linux/device.h */

struct device {
        ...
        u64                 *dma_mask;
        u64                 coherent_dma_mask;
        ...
};

/* linux/dma-mapping.h */

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

DMA Mask

Userspace Mapping

```
/* asm-generic/dma-mapping.h */

/* Implemented on arm, arm64 and powerpc */

int dma_mmap_attrs(struct device *dev,
                   struct vm_area_struct *vma,
                   void *cpu_addr,
                   dma_addr_t dma_addr, size_t size,
                   struct dma_attrs *attrs);

/* Wrappers */

int dma_mmap_coherent(...);
int dma_mmap_writecombine(...);
```

Userspace Mapping

```
int dma_mmap_attrs(struct device *dev,
                   struct vm_area_struct *vma,
                   void *cpu_addr,
                   dma_addr_t dma_addr, size_t size,
                   struct dma_attrs *attrs);
```

Map coherent or write-combine DMA memory previously allocated by dma_alloc_attrs() into user space. The DMA memory must not be freed by the driver until the user space mapping has been released.

Creating multiple mappings with different types (coherent, write-combined, weakly ordered or non-coherent) produces undefined results on some architectures. Care must be taken to specify the same type attributes for all calls to the dma_alloc_attrs() and dma_mmap_attrs() functions for the same memory.

If the memory has been allocated with the NO_KERNEL_MAPPING attribute the same attribute must be passed to all calls to dma_mmap_attrs().

Userspace Mapping

```
/*
 * Implemented on arc, avr32, blackfin, cris, m68k and
 * metag
 */
int dma_mmap_coherent(struct device *dev,
                      struct vm_area_struct *vma,
                      void *cpu_addr,
                      dma_addr_t dma_addr, size_t size);

/* Implemented on metag */
int dma_mmap_writecombine(struct device *dev,
                          struct vm_area_struct *vma,
                          void *cpu_addr,
                          dma_addr_t dma_addr,
                          size_t size);
```

Userspace Mapping

DMA Streaming Mapping

```
/* linux/dma-direction.h */

enum dma_data_direction {
        DMA_BIDIRECTIONAL = 0,
        DMA_TO_DEVICE = 1,
        DMA_FROM_DEVICE = 2,
        DMA_NONE = 3,
};
```

DMA Direction

```
/* asm-generic/dma-mapping.h */

dma_addr_t
dma_map_single_attrs(struct device *dev, void *ptr,
                     size_t size,
                     enum dma_data_direction dir,
                     struct dma_attrs *attrs);

void
dma_unmap_single_attrs(struct device *dev,
                       dma_addr_t addr, size_t size,
                       enum dma_data_direction dir,
                       struct dma_attrs *attrs);

dma_addr_t dma_map_single(...);
void dma_unmap_single(...);
```

Device Mapping

```
/* asm-generic/dma-mapping.h */

dma_addr_t
dma_map_page(struct device *dev, struct page *page,
             size_t offset, size_t size,
             enum dma_data_direction dir);

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

Device Mapping

```
/* asm-generic/dma-mapping.h */

int
dma_map_sg_attrs(struct device *dev,
                 struct scatterlist *sg, int nents,
                 enum dma_data_direction dir,
                 struct dma_attrs *attrs);

void
dma_unmap_sg_attrs(struct device *dev,
                   struct scatterlist *sg,
                   int nents,
                   enum dma_data_direction dir,
                   struct dma_attrs *attrs);

int dma_map_sg(...);
void dma_unmap_sg(...);
```

Device Mapping

```
/* asm/dma-mapping.h */

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

In some circumstances dma_map_*() 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).

Error Checking

```
/* asm-generic/dma-mapping.h */

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

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

Synchronization

```
/* asm-generic/dma-mapping.h */

void
dma_sync_single_for_*(struct device *dev,
                      dma_addr_t addr, size_t size,
                      enum dma_data_direction dir);

void
dma_sync_single_range_for_*(struct device *dev,
                            dma_addr_t addr,
                            unsigned long offset,
                            size_t size,
                            enum dma_data_direction dir);

void
dma_sync_sg_for_*(struct device *dev,
                  struct scatterlist *sg, int nelems,
                  enum dma_data_direction dir);

(* = cpu or device)
```

Synchronization

Contiguous Memory Allocation

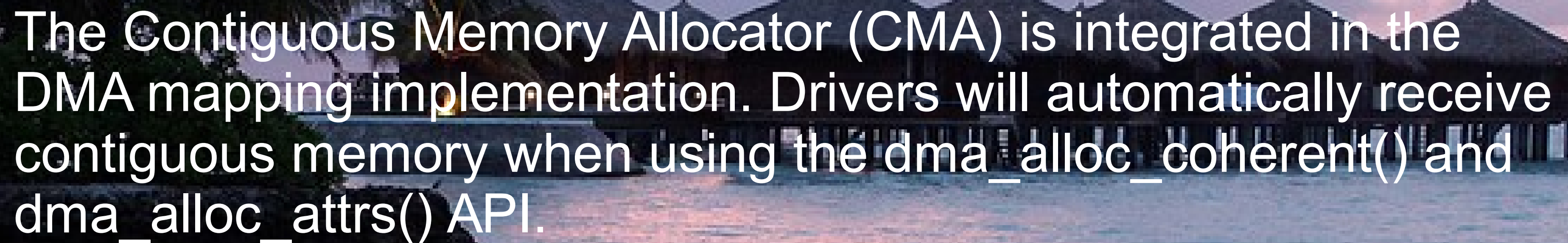
```
#include <linux/dma-contiguous.h>

drivers/base/dma-contiguous.h
```

CMA

From a Driver Point of View

From a Driver Point of View



The Contiguous Memory Allocator (CMA) is integrated in the DMA mapping implementation. Drivers will automatically receive contiguous memory when using the dma_alloc_coherent() and dma_alloc_attrs() API.

IDEAS ON BOARD

```
/* linux/dma-contiguous.h */

void dma_contiguous_reserve(phys_addr_t addr_limit);

int dma_declare_contiguous(struct device *dev,
                           phys_addr_t size,
                           phys_addr_t base,
                           phys_addr_t limit);
```

From a System Point of View

```
/* linux/dma-contiguous.h */

void dma_contiguous_reserve(phys_addr_t addr_limit);
```

This function reserves memory from early allocator. It should be called by arch specific code once the early allocator (memblock or bootmem) has been activated and all other subsystems have already allocated/reserved memory.

The size of the reserved memory area is specified through the kernel configuration and can be overridden on the kernel command line. An area of the given size is reserved from the early allocator for contiguous allocation.

```
int dma_declare_contiguous(struct device *dev,
                           phys_addr_t size,
                           phys_addr_t base,
                           phys_addr_t limit);
```

This function reserves memory for the specified device. It should be called by board specific code when early allocator (memblock or bootmem) has been activated.

From a System Point of View

IOMMU Integration

```
#include <linux/iommu.h>
```

IOMMU API

```
/* linux/iommu.h */

struct iommu_domain *
iommu_domain_alloc(struct bus_type *bus);
void iommu_domain_free(struct iommu_domain *domain);

int iommu_attach_device(struct iommu_domain *domain,
                        struct device *dev);
void iommu_detach_device(struct iommu_domain *domain,
                         struct device *dev);

int iommu_map(struct iommu_domain *domain,
              unsigned long iova, phys_addr_t paddr,
              size_t size, int prot);
size_t iommu_unmap(struct iommu_domain *domain,
                   unsigned long iova, size_t size);
```

IOMMU API

```
/* asm/dma-mapping.h */

struct dma_iommu_mapping *
arm_iommu_create_mapping(struct bus_type *bus,
                         dma_addr_t base, size_t size);
void arm_iommu_release_mapping(
                     struct dma_iommu_mapping *mapping);

int arm_iommu_attach_device(struct device *dev,
                     struct dma_iommu_mapping *mapping);
void arm_iommu_detach_device(struct device *dev);
```

IOMMU Integration (ARM)

"Someone" must create the ARM mapping and attach devices.

To achieve transparent IOMMU integration the calls must be moved from device drivers to IOMMU drivers. This creates new challenges:

- Devices might need fine-grained control over the IOMMU (such as mapping memory at a fixed device address). They would then need to manage the IOMMU in cooperation with the DMA mapping API.
- Devices might have several bus master ports connected to different IOMMUs, while the DMA mapping API operates at the device level.
- Power management needs to be taken care of.

IOMMU Integration (ARM)

Device Tree Bindings

[PATCH v7 0/4] Device Tree support for CMA (Contiguous Memory Allocator)

http://lwn.net/Articles/564830/

Device Tree Bindings – CMA

Documentation/devicetree/bindings/iommu

Device Tree Bindings – IOMMU

Tips & Tricks

- Use the correct API, choose wisely between coherent and streaming mappings.
- Don't try to manage the cache manually, it's bound to fail.
- Set your DMA masks.
- Use dma_mapping_error().

Coherent Mappings

- Set the DMA_ATTR_SKIP_CPU_SYNC when calling dma_map_*().
- Don't call dma_sync_*().

Tips & Tricks

Problems & Issues

Generic Problems

- Coherent mappings and streaming mappings exhibit different performances depending on the use case, which should be configurable from userspace.
- Lack of standard DT bindings for IOMMUs.
- Coherent and non-coherent masks are confusing and badly implemented.
- Headers hierarchy is confusing.
- The dma_sync_*() API has no attributes and thus can't skip CPU cache synchronization for coherent mappings.

ARM-Specific Problems

- Lack of non-coherent allocation.
- Flushing a cache range can be less efficient than flushing the whole D-cache.
- The DMA mask is not taken into account when creating IOMMU mappings.

Problems & Issues

Resources

- Documentation/DMA-API-HOWTO.txt
- Documentation/DMA-API.txt
- Documentation/DMA-attributes.txt

- http://community.arm.com/groups/processors/blog/2011/03/22/memory-access-ordering-an-introduction
- http://elinux.org/images/7/73/Deacon-weak-to-weedy.pdf
- https://lwn.net/Articles/486301/

Documentation

- **linux-kernel@vger.kernel.org**
- **linux-arm-kernel@lists.infradead.org**

- laurent.pinchart@ideasonboard.com

Contact

? !

Thx.

Advanced Topics

DMA Coherent Memory Pool

```
/* linux/dmapool.h */
```

The DMA mapping API allocates buffers in at least page size chunks. If your driver needs lots of smaller memory regions you can use the DMA pool API to subdivide pages returned by dma_alloc_coherent().

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

This function creates a DMA allocation pool to allocate buffers of the given @size and alignment characteristics (@ must be a power of two and can be set to zero). If @boundary is nonzero, objects returned from dma_pool_alloc() won't cross that size boundary. This is useful for devices which have addressing restrictions on individual DMA transfers.

Given one of these pools, dma_pool_alloc() may be used to allocate memory. Such memory will all have “consistent” DMA mappings, accessible by the device and its driver without using cache flushing primitives.

DMA Pool

```
/* linux/dmapool.h */

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

Destroy a DMA pool. The caller guarantees that no more memory from the pool is in use,and that nothing will try to use the pool after this call. A DMA pool can't be destroyed in interrupt context.

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

This returns the kernel virtual address of a currently unused block, and reports its DMA address through the handle. Return NULL when allocation fails.

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

Puts memory back into the pool. The CPU (vaddr) and DMA addresses are what were returned when dma_pool_alloc() allocated the memory being freed.

DMA Pool

Non-Coherent Mapping

```
/* asm-generic/dma-mapping.h */

void *
dma_alloc_noncoherent(struct device *dev, size_t size,
                      dma_addr_t *dma_handle,
                      gfp_t flag);

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

Non-Coherent Allocation

The non-coherent memory allocation is architecture-dependent. The following list summarizes the behaviour of supported architectures.

- **Allocates Normal Cacheable Memory**

  arc, mips, openrisc, parisc

- **Allocates Coherent Memory**

  alpha, avr32, blackfin, c6x, cris, frv, hexagon, ia64, m68k, metag, microblaze, mn10300, powerpc, s390, sh, sparc, tile, unicore32, x86, xtensa

  Note that some of those architectures can be fully coherent, in which case the concept of non-coherent memory doesn't apply and memory mappings are always coherent.

- **Returns NULL**

  arm, arm64

Non-Coherent Allocation

Generic DMA Coherent Memory Allocator

```
/* asm-generic/dma-coherent.h */

/*
 * Standard interface
 */
#define ARCH_HAS_DMA_DECLARE_COHERENT_MEMORY
extern int
dma_declare_coherent_memory(struct device *dev,
                            dma_addr_t bus_addr,
                            dma_addr_t device_addr,
                            size_t size, int flags);

extern void
dma_release_declared_memory(struct device *dev);

extern void *
dma_mark_declared_memory_occupied(struct device *dev,
                                  dma_addr_t device_addr,
                                  size_t size);
```

Device API

```
/* asm-generic/dma-coherent.h */

extern int
dma_declare_coherent_memory(struct device *dev,
                            dma_addr_t bus_addr,
                            dma_addr_t device_addr,
                            size_t size, int flags);
```

Declare a coherent memory area for a device. The area is specified by its (CPU) bus address, device bus address and size. The following flags can be specified:

- DMA_MEMORY_MAP – allocated memory is directly writable (always set).
- DMA_MEMORY_IO – allocated memory accessed as I/O mem (unused).
- DMA_MEMORY_INCLUDES_CHILDREN – declared memory available to all child devices (unsupported).
- DMA_MEMORY_EXCLUSIVE – force allocation to be made exclusively from the coherent area for this device without any fallback method.

Only a single coherent memory area can be declared per device.

Device API

```
/* asm-generic/dma-coherent.h */

extern void
dma_release_declared_memory(struct device *dev);
```

Release the coherent memory previously declared for the device. All DMA coherent memory allocated for the device must be freed before calling this function.

Device API

```
/* asm-generic/dma-coherent.h */

extern void *
dma_mark_declared_memory_occupied(struct device *dev,
                                  dma_addr_t device_addr,
                                  size_t size);
```

Mark part of the coherent memory area as unusable for DMA coherent memory allocation. Multiple ranges can be marked as occupied.

This function is used by the NCR_Q720 SCSI driver only to reserve the first kB. In this specific case this could be handled by declaring a coherent region that skips the first page.

Device API

```
/* asm-generic/dma-coherent.h */

/*
 * These three functions are only for dma allocator.
 * Don't use them in device drivers.
 */
```

Allocator Private API

```
/* asm-generic/dma-coherent.h */

/*
 * These three functions are only for dma allocator.
 * Don't use them in device drivers.
 */
int dma_alloc_from_coherent(struct device *dev,
                            ssize_t size,
                            dma_addr_t *dma_handle,
                            void **ret);
int dma_release_from_coherent(struct device *dev,
                              int order, void *vaddr);
int dma_mmap_from_coherent(struct device *dev,
                           struct vm_area_struct *vma,
                           void *cpu_addr, size_t size,
                           int *ret);
```

Allocator Private API

```
/* asm-generic/dma-coherent.h */

int dma_alloc_from_coherent(struct device *dev,
                            ssize_t size,
                            dma_addr_t *dma_handle,
                            void **ret);
```

Try to allocate memory from the per-device coherent area.

Returns 0 if dma_alloc_coherent should continue with allocating from generic memory areas, or !0 if dma_alloc_coherent should return @ret.

This function can only be called from per-arch dma_alloc_coherent (and dma_alloc_attrs) to support allocation from per-device coherent memory pools.

Allocator Private API

```
/* asm-generic/dma-coherent.h */

int dma_release_from_coherent(struct device *dev,
                              int order, void *vaddr);
```

Try to free the memory allocated from per-device coherent memory pool.

This checks whether the memory was allocated from the per-device coherent memory pool and if so, releases that memory and returns 1. Otherwise it returns 0 to signal that the caller should proceed with releasing memory from generic pools.

This function can only be called from within the architecture's dma_free_coherent (and dma_free_attrs) implementation.

Allocator Private API

```
/* asm-generic/dma-coherent.h */

int dma_mmap_from_coherent(struct device *dev,
                           struct vm_area_struct *vma,
                           void *cpu_addr, size_t size,
                           int *ret);
```

Try to mmap the memory allocated from per-device coherent memory pool to userspace.

This checks whether the memory was allocated from the per-device coherent memory pool and if so, maps that memory to the provided vma and returns 1. Otherwise it returns 0 to signal that the caller should proceed with mapping memory from generic pools.

This function can only be called from within the architecture's dma_alloc_coherent (and dma_alloc_attrs) implementation.

Allocator Private API