

**NAME**

**bus\_dma**, **bus\_dma\_tag\_create**, **bus\_dma\_tag\_destroy**, **bus\_dmamap\_create**,  
**bus\_dmamap\_destroy**, **bus\_dmamap\_load**, **bus\_dmamap\_load\_mbuf**,  
**bus\_dmamap\_load\_uio**, **bus\_dmamap\_unload**, **bus\_dmamap\_sync**, **bus\_dmamem\_alloc**,  
**bus\_dmamem\_free**, – Bus and Machine Independent DMA Mapping Interface

**SYNOPSIS**

```
#include <machine/bus.h>

int
bus_dma_tag_create(bus_dma_tag_t parent, bus_size_t alignment,
                  bus_size_t boundary, bus_addr_t lowaddr, bus_addr_t highaddr,
                  bus_dma_filter_t *filtfunc, void *filtfuncarg, bus_size_t maxsize,
                  int nsegments, bus_size_t maxsegsz, int flags,
                  bus_dma_tag_t *dmat);

int
bus_dma_tag_destroy(bus_dma_tag_t dmat);

int
bus_dmamap_create(bus_dma_tag_t dmat, int flags, bus_dmamap_t *mapp);

int
bus_dmamap_destroy(bus_dma_tag_t dmat, bus_dmamap_t map);

int
bus_dmamap_load(bus_dma_tag_t dmat, bus_dmamap_t map, void *buf,
                bus_size_t buflen, bus_dmamap_callback_t *callback,
                void *callback_arg, int flags);

int
bus_dmamap_load_mbuf(bus_dma_tag_t dmat, bus_dmamap_t map,
                    struct mbuf *mbuf, bus_dmamap_callback2_t *callback,
                    void *callback_arg, int flags);

int
bus_dmamap_load_uio(bus_dma_tag_t dmat, bus_dmamap_t map, struct uio *uio,
                   bus_dmamap_callback2_t *callback, void *callback_arg, int flags);

int
bus_dmamem_alloc(bus_dma_tag_t dmat, void **vaddr, int flags,
                bus_dmamap_t *mapp);

void
bus_dmamap_unload(bus_dma_tag_t dmat, bus_dmamap_t map);

void
bus_dmamap_sync(bus_dma_tag_t dmat, bus_dmamap_t map, op);

void
bus_dmamem_free(bus_dma_tag_t dmat, void *vaddr, bus_dmamap_t map);
```

**DESCRIPTION**

Direct Memory Access (DMA) is a method of transferring data without involving the CPU, thus providing higher performance. A DMA transaction can be achieved between device to memory, device to device, or memory to memory.

The **bus\_dma** API is a bus, device, and machine-independent (MI) interface to DMA mechanisms. It provides the client with flexibility and simplicity by abstracting machine dependent issues like setting up DMA mappings, handling cache issues, bus specific features and limitations.

## STRUCTURES AND TYPES

### *bus\_dma\_tag\_t*

A machine-dependent (MD) opaque type that describes the characteristics of DMA transactions. DMA tags are organized into a hierarchy, with each child tag inheriting the restrictions of its parent. This allows all devices along the path of DMA transactions to contribute to the constraints of those transactions.

### *bus\_dma\_filter\_t*

Client specified address filter having the format:

```
int      client_filter(void *filtarg, bus_addr_t testaddr)
```

Address filters can be specified during tag creation to allow for devices whose DMA address restrictions cannot be specified by a single window. The *filtarg* is client specified during tag creation to be passed to all invocations of the callback. The *testaddr* argument contains a potential starting address of a DMA mapping. The filter function operates on the set of addresses from *testaddr* to `trunc_page(testaddr) + PAGE_SIZE - 1`, inclusive. The filter function should return zero for any mapping in this range that can be accommodated by the device and non-zero otherwise.

### *bus\_dma\_segment\_t*

A machine-dependent type that describes individual DMA segments.

```
bus_addr_t    ds_addr;
bus_size_t    ds_len;
```

The *ds\_addr* field contains the device visible address of the DMA segment, and *ds\_len* contains the length of the DMA segment. Although the DMA segments returned by a mapping call will adhere to all restrictions necessary for a successful DMA operation, some conversion (e.g. a conversion from host byte order to the device's byte order) is almost always required when presenting segment information to the device.

### *bus\_dmamap\_t*

A machine-dependent opaque type describing an individual mapping. Multiple DMA maps can be associated with one DMA tag.

### *bus\_dmamap\_callback\_t*

Client specified callback for receiving mapping information resulting from the load of a *bus\_dmamap\_t* via **bus\_dmamap\_load()**. Callbacks are of the format:

```
void      client_callback(void *callback_arg, bus_dma_segment_t
                          *segs, int nseg, int error)
```

The *callback\_arg* is the callback argument passed to dmamap load functions. The *segs* and *nseg* parameters describe an array of *bus\_dma\_segment\_t* structures that represent the mapping. This array is only valid within the scope of the callback function. The success or failure of the mapping is indicated by the *error* parameter. More information on the use of callbacks can be found in the description of the individual dmamap load functions.

### *bus\_dmamap\_callback2\_t*

Client specified callback for receiving mapping information resulting from the load of a *bus\_dmamap\_t* via **bus\_dmamap\_load\_uio()** or **bus\_dmamap\_load\_mbuf()**.

Callback2s are of the format:

```
void client_callback2(void *callback_arg, bus_dma_segment_t
    *segs, int nseg, bus_size_t mapsize, int error)
```

Callback2's behavior is the same as *bus\_dmamap\_callback\_t* with the addition that the length of the data mapped is provided via *mapsize*.

*bus\_dmasync\_op\_t*

Memory synchronization operation specifier. Bus DMA requires explicit synchronization of memory with its device visible mapping in order to guarantee memory coherency. The *bus\_dmasync\_op\_t* allows the type of DMA operation that will be or has been performed to be communicated to the system so that the correct coherency measures are taken. All operations specified below are performed from the DMA engine's point of view:

BUS\_DMASYNC\_PREREAD Perform any synchronization required after an update of memory by the CPU but prior to DMA read operations.

BUS\_DMASYNC\_PREWRITE Perform any synchronization required after an update of memory by the CPU but prior to DMA write operations.

BUS\_DMASYNC\_PREREAD | BUS\_DMASYNC\_PREWRITE  
Perform any synchronization required prior to a combination of DMA read and write operations.

BUS\_DMASYNC\_POSTREAD Perform any synchronization required after DMA read operations, but prior to CPU access of the memory.

BUS\_DMASYNC\_POSTWRITE Perform any synchronization required after DMA write operations, but prior to CPU access of the memory.

BUS\_DMASYNC\_POSTREAD | BUS\_DMASYNC\_POSTWRITE  
Perform any synchronization required after a combination of DMA read and write operations.

## FUNCTIONS

**bus\_dma\_tag\_create**(parent, alignment, boundary, lowaddr, highaddr, \*filtfunc, \*filtfuncarg, maxsize, nsegments, maxsegsz, flags, \*dmat)

Allocates a device specific DMA tag, and initializes it according to the arguments provided:

*parent* Indicates restrictions between the parent bridge, CPU memory, and the device. May be NULL, if no DMA restrictions are to be inherited.

*alignment* Alignment constraint, in bytes, of any mappings created using this tag. The alignment must be a power of 2. Hardware that can DMA starting at any address would specify 1 for byte alignment. Hardware requiring DMA transfers to start on a multiple of 4K would specify 4096.

*boundary* Boundary constraint, in bytes, of the target DMA memory region. The boundary indicates the set of addresses, all multiples of the boundary argument, that cannot be crossed by a single *bus\_dma\_segment\_t*. The boundary must be either a power of 2 or 0. '0' indicates that there are no boundary restrictions.

*lowaddr*

*highaddr* Bounds of the window of bus address space that *cannot* be directly accessed by the device. The window contains all address greater than lowaddr and less than or equal to highaddr. For example, a device incapable of DMA above 4GB, would specify a highaddr of BUS\_SPACE\_MAXADDR and a lowaddr of BUS\_SPACE\_MAXADDR\_32BIT. Similarly a device that can only dma to

addresses below 16MB would specify a `highaddr` of `BUS_SPACE_MAXADDR` and a `lowaddr` of `BUS_SPACE_MAXADDR_24BIT`. Some implementations requires that some region of device visible address space, overlapping available host memory, be outside the window. This area of *safe* memory is used to bounce requests that would otherwise conflict with the exclusion window.

*filtfunc* Optional filter function (may be NULL) to be called for any attempt to map memory into the window described by *lowaddr* and *highaddr*. A filter function is only required when the single window described by *lowaddr* and *highaddr* cannot adequately describe the constraints of the device. The filter function will be called for every machine page that overlaps the exclusion window.

*filtfuncarg* Argument passed to all calls to the filter function for this tag. May be NULL.

*maxsegsz* Maximum size, in bytes, of a segment in any DMA mapped region associated with *dmata*.

*maxsize* Maximum size, in bytes, of the sum of all segment lengths in a given DMA mapping associated with this tag.

*nsegments* Number of discontinuities (scatter/gather segments) allowed in a DMA mapped region. If there is no restriction, `BUS_SPACE_UNRESTRICTED` may be specified.

*flags* Are as follows:  
`BUS_DMA_ALLOCNOW` Allocate the resources necessary to guarantee that all map load operations associated with this tag will not block. If sufficient resources are not available, `ENOMEM` is returned.

*dmata* Pointer to a `bus_dma_tag_t` where the resulting DMA tag will be stored.

Returns `ENOMEM` if sufficient memory is not available for tag creation or allocating mapping resources.

**bus\_dma\_tag\_destroy**(*dmata*)

Deallocate the DMA tag *dmata* that was created by `bus_dma_tag_create()`.

Returns `EBUSY` if any DMA maps remain associated with *dmata* or '0' on success.

**bus\_dmamap\_create**(*dmata*, *flags*, \**mapp*)

Allocates and initializes a DMA map. Arguments are as follows:

*dmata* DMA tag.  
*flags* The value of this argument is currently undefined and should be specified as '0'.  
*mapp* Pointer to a `bus_dmamap_t` where the resulting DMA map will be stored.

Returns `ENOMEM` if sufficient memory is not available for creating the map or allocating mapping resources.

**bus\_dmamap\_destroy**(*dmata*, *map*)

Frees all resources associated with a given DMA map. Arguments are as follows:

*dmata* DMA tag used to allocate *map*.  
*map* The DMA map to destroy.

Returns `EBUSY` if a mapping is still active for *map*.

**bus\_dmamap\_load**(*dmata*, *map*, *buf*, *buflen*, \**callback*, . . .)

Creates a mapping in device visible address space of *buflen* bytes of *buf*, associated with the DMA map *map*. Arguments are as follows:

*dmata* DMA tag used to allocate *map*.

*map* A DMA map without a currently active mapping.

*buf* A kernel virtual address pointer to a contiguous (in KVA) buffer, to be mapped into device visible address space.

*buflen* The size of the buffer.

*callback callback\_arg*  
The callback function, and its argument.

*flags* The value of this argument is currently undefined, and should be specified as '0'.

Return values to the caller are as follows:

0 The callback has been called and completed. The status of the mapping has been delivered to the callback.

EINPROGRESS The mapping has been deferred for lack of resources. The callback will be called as soon as resources are available. Callbacks are serviced in FIFO order. DMA maps created from DMA tags that are allocated with the BUS\_DMA\_ALLOCNOW flag will never return this status for a load operation.

EINVAL The load request was invalid. The callback has not, and will not be called. This error value may indicate that *dmata*, *map*, *buf*, or *callback* were invalid, or *buflen* was larger than the *maxsize* argument used to create the dma tag *dmata*.

When the callback is called, it is presented with an error value indicating the disposition of the mapping. Error may be one of the following:

0 The mapping was successful and the *dm\_segs* callback argument contains an array of *bus\_dma\_segment\_t* elements describing the mapping. This array is only valid during the scope of the callback function.

EFBIG A mapping could not be achieved within the segment constraints provided in the tag even though the requested allocation size was less than *maxsize*.

**bus\_dmamap\_load\_mbuf**(*dmata*, *map*, *mbuf*, *callback2*, *callback\_arg*, *flags*)

This is a variation of **bus\_dmamap\_load**() which maps mbuf chains for DMA transfers. A *bus\_size\_t* argument is also passed to the callback routine, which contains the mbuf chain's packet header length.

Mbuf chains are assumed to be in kernel virtual address space.

Returns EINVAL if the size of the mbuf chain exceeds the maximum limit of the DMA tag.

**bus\_dmamap\_load\_uio**(*dmata*, *map*, *uio*, *callback2*, *callback\_arg*, *flags*)

This is a variation of **bus\_dmamap\_load**() which maps buffers pointed to by *uio* for DMA transfers. A *bus\_size\_t* argument is also passed to the callback routine, which contains the size of *uio*, i.e. *uio->uio\_resid*.

If *uio->uio\_segflg* is UIO\_USERSPACE, then it is assumed that the buffer, *uio* is in *uio->uio\_td->td\_proc*'s address space. User space memory must be in-core and wired prior to attempting a map load operation.

**bus\_dmamap\_unload**(*dmata*, *map*)

Unloads a DMA map. Arguments are as follows:

*dmata* DMA tag used to allocate *map*.

*map* The DMA map that is to be unloaded.

**bus\_dmamap\_unload**() will not perform any implicit synchronization of DMA buffers. This must be done explicitly by a call to **bus\_dmamap\_sync**() prior to unloading the map.

**bus\_dmamap\_sync**(*dmata*, *map*, *op*)

Performs synchronization of a device visible mapping with the CPU visible memory referenced by that mapping. Arguments are as follows:

*dmata* DMA tag used to allocate *map*.

*map* The DMA mapping to be synchronized.

*op* Type of synchronization operation to perform. See the definition of *bus\_dmasync\_op\_t* for a description of the acceptable values for *op*.

**bus\_dmamap\_sync**() is the method used to ensure that CPU and device DMA access to shared memory is coherent. For example, the CPU might be used to setup the contents of a buffer that is to be DMA'ed into a device. To ensure that the data are visible via the device's mapping of that memory, the buffer must be loaded and a dma sync operation of `BUS_DMASYNC_PREREAD` must be performed. Additional sync operations must be performed after every CPU write to this memory if additional DMA reads are to be performed. Conversely, for the DMA write case, the buffer must be loaded, and a dma sync operation of `BUS_DMASYNC_PREWRITE` must be performed. The CPU will only be able to see the results of this DMA write once the DMA has completed and a `BUS_DMASYNC_POSTWRITE` operation has been performed.

If DMA read and write operations are not preceded and followed by the appropriate synchronization operations, behavior is undefined.

**bus\_dmamem\_alloc**(*dmata*, *\*\*vaddr*, *flags*, *mapp*)

Allocates memory that is mapped into KVA at the address returned in *vaddr* that is permanently loaded into the newly created *bus\_dmamap\_t* returned via *mapp*. Arguments are as follows:

*dmata* DMA tag describing the constraints of the DMA mapping.

*vaddr* Pointer to a pointer that will hold the returned KVA mapping of the allocated region.

*flags* Flags are defined as follows:

`BUS_DMA_WAITOK` The routine can safely wait (sleep) for resources.

`BUS_DMA_NOWAIT` The routine is not allowed to wait for resources. If resources are not available, `ENOMEM` is returned.

`BUS_DMA_COHERENT`

Attempt to map this memory such that cache sync operations are as cheap as possible. This flag is typically set on memory that will be accessed by both a CPU and a DMA engine, frequently. Use of this flag does not remove the requirement of using `bus_dmamap_sync`, but it may reduce the cost of performing these operations.

*mapp* Pointer to storage for the returned DMA map.

The size of memory to be allocated is *maxsize* as specified in *dmata*.

The current implementation of **bus\_dmamem\_alloc**() will allocate all requests as a single segment.

Although no explicit loading is required to access the memory referenced by the returned map, the synchronization requirements as described in the **bus\_dmamap\_sync**() section still apply.

Returns `ENOMEM` if sufficient memory is not available for completing the operation.

**bus\_dmamem\_free**(*dmata*, *\*vaddr*, *map*)

Frees memory previously allocated by **bus\_dmamem\_alloc**(). Any mappings will be invalidated. Arguments are as follows:

*dma\_t* DMA tag.  
*vaddr* Kernel virtual address of the memory.  
*map* DMA map to be invalidated.

## RETURN VALUES

Behavior is undefined if invalid arguments are passed to any of the above functions. If sufficient resources cannot be allocated for a given transaction, ENOMEM is returned. All routines that are not of type, *void*, will return 0 on success or an error code, as discussed above.

All *void* routines will succeed if provided with valid arguments.

## SEE ALSO

`devclass(9)`, `device(9)`, `driver(9)`, `rman(9)`

Jason R. Thorpe, "A Machine-Independent DMA Framework for NetBSD", *Proceedings of the Summer 1998 USENIX Technical Conference*, USENIX Association, June 1998.

## HISTORY

The **bus\_dma** interface first appeared in NetBSD 1.3.

The **bus\_dma** API was adopted from NetBSD for use in the CAM SCSI subsystem. The alterations to the original API were aimed to remove the need for a *bus\_dma\_segment\_t* array stored in each *bus\_dmamap\_t* while allowing callers to queue up on scarce resources.

## AUTHORS

The **bus\_dma** interface was designed and implemented by Jason R. Thorpe of the Numerical Aerospace Simulation Facility, NASA Ames Research Center. Additional input on the **bus\_dma** design was provided by Chris Demetriou, Charles Hannum, Ross Harvey, Matthew Jacob, Jonathan Stone, and Matt Thomas.

The **bus\_dma** interface in FreeBSD benefits from the contributions of Justin T. Gibbs, Peter Wemm, Doug Rabson, Matthew N. Dodd, Sam Leffler, Maxime Henrion, Jake Burkholder, Takahashi Yoshihiro, Scott Long and many others.

This manual page was written by Hiten M. Pandya and Justin T. Gibbs.