NAME

bus_dma, **bus_dma_tag_create**, **bus_dma_tag_destroy**, **bus_dma_template_init**, **bus_dma_template_tag**, **bus_dma_template_clone**, **bus_dma_template_fill**, **BUS_DMA_TEMPLATE_FILL**, **bus_dmamap_create**, **bus_dmamap_destroy**, **bus_dmamap_load**, **bus_dmamap_load_bio**, **bus_dmamap_load_ccb**, **bus_dmamap_load_crp**, **bus_dmamap_load_crp_buffer**, **bus_dmamap_load_mbuf**, **bus_dmamap_load_mbuf_sg**, **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_addr_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_lock_t *lockfunc*, *void *lockfuncarg*, *bus_dma_tag_t *dmat*);

int
bus_dma_tag_destroy(*bus_dma_tag_t dmat*);

void
bus_dma_template_init(*bus_dma_template_t *template*, *bus_dma_tag_t parent*);

int
bus_dma_template_tag(*bus_dma_template_t *template*, *bus_dma_tag_t *dmat*);

void
bus_dma_template_clone(*bus_dma_template_t *template*, *bus_dma_tag_t dmat*);

void
bus_dma_template_fill(*bus_dma_template_t *template*, *bus_dma_param_t params[]*, *u_int count*);

BUS_DMA_TEMPLATE_FILL(*bus_dma_template_t *template*, *bus_dma_param_t param* ...);

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_bio(*bus_dma_tag_t dmat, bus_dmamap_t map, struct bio *bio, bus_dmamap_callback_t *callback, void *callback_arg, int flags*);

int
bus_dmamap_load_ccb(*bus_dma_tag_t dmat, bus_dmamap_t map, union ccb *ccb, bus_dmamap_callback_t *callback, void *callback_arg, int flags*);

int
bus_dmamap_load_crp(*bus_dma_tag_t dmat, bus_dmamap_t map, struct crypto *crp, bus_dmamap_callback_t *callback, void *callback_arg, int flags*);

int
bus_dmamap_load_crp_buffer(*bus_dma_tag_t dmat, bus_dmamap_t map, struct crypto_buffer *cb, 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_mbuf_sg(*bus_dma_tag_t dmat, bus_dmamap_t map, struct mbuf *mbuf, bus_dma_segment_t *segs, int *nsegs, 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*);

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*);

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

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.

OVERVIEW

A tag structure (*bus_dma_tag_t*) is used to describe the properties of a group of related DMA transactions. One way to view this is that a tag describes the limitations of a DMA engine. For example, if a DMA engine in a device is limited to 32-bit addresses, that limitation is specified by a parameter when creating the tag for that device. Similarly, a tag can be marked as requiring buffers whose addresses are aligned to a specific boundary.

Some devices may require multiple tags to describe DMA transactions with differing properties. For example, a device might require 16-byte alignment of its descriptor ring while permitting arbitrary alignment of I/O buffers. In this case, the driver must create one tag for the descriptor ring and a separate tag for I/O buffers. If a device has restrictions that are common to all DMA transactions in addition to restrictions that differ between unrelated groups of transactions, the driver can first create a "parent" tag that decribes the common restrictions. The per-group tags can then inherit these restrictions from this "parent" tag rather than having to list them explicitly when creating the per-group tags.

A mapping structure (*bus_dmamap_t*) represents a mapping of a memory region for DMA. On systems with I/O MMUs, the mapping structure tracks any I/O MMU entries used by a request. For DMA requests that require bounce pages, the mapping tracks the bounce pages used.

To prepare for one or more DMA transactions, a mapping must be bound to a memory region by calling one of the **bus_dmamap_load**() functions. These functions configure the mapping which can include programming entries in an I/O MMU and/or allocating bounce pages. An output of these functions (either directly or indirectly by invoking a callback routine) is the list of scatter/gather address ranges a consumer can pass to a DMA engine to access the memory region. When a mapping is no longer needed, the mapping must be unloaded via **bus_dmamap_unload**().

Before and after each DMA transaction, **bus_dmamap_sync**() must be used to ensure that the correct data is used by the DMA engine and the CPU. If a mapping uses bounce pages, the sync operations

copy data between the bounce pages and the memory region bound to the mapping. Sync operations also handle architecture-specific details such as CPU cache flushing and CPU memory operation ordering.

STATIC VS DYNAMIC

bus_dma handles two types of DMA transactions: static and dynamic. Static transactions are used with a long-lived memory region that is reused for many transactions such as a descriptor ring. Dynamic transactions are used for transfers to or from transient buffers such as I/O buffers holding a network packet or disk block. Each transaction type uses a different subset of the **bus_dma** API.

Static Transactions

Static transactions use memory regions allocated by **bus_dma**. Each static memory region is allocated by calling **bus_dmamem_alloc**(). This function requires a valid tag describing the properties of the DMA transactions to this region such as alignment or address restrictions. Multiple regions can share a single tag if they share the same restrictions.

bus_dmamem_alloc() allocates a memory region along with a mapping object. The associated tag, memory region, and mapping object must then be passed to **bus_dmamap_load**() to bind the mapping to the allocated region and obtain the scatter/gather list.

It is expected that **bus_dmamem_alloc**() will attempt to allocate memory requiring less expensive sync operations (for example, implementations should not allocate regions requiring bounce pages), but sync operations should still be used. For example, a driver should use **bus_dmamap_sync**() in an interrupt handler before reading descriptor ring entries written by the device prior to the interrupt.

When a consumer is finished with a memory region, it should unload the mapping via **bus_dmamap_unload**() and then release the memory region and mapping object via **bus_dmamem_free**().

Dynamic Transactions

Dynamic transactions map memory regions provided by other parts of the system. A tag must be created via **bus_dma_tag_create**() to describe the DMA transactions to and from these memory regions, and a pool of mapping objects must be allocated via **bus_dmamap_create**() to track the mappings of any in-flight transactions.

When a consumer wishes to schedule a transaction for a memory region, the consumer must first obtain an unused mapping object from its pool of mapping objects. The memory region must be bound to the mapping object via one of the **bus_dmamap_load**() functions. Before scheduling the transaction, the consumer should sync the memory region via **bus_dmamap_sync**() with one or more of the "PRE" flags. After the transaction has completed, the consumer should sync the memory region via

bus_dmamap_sync() with one or more of the "POST" flags. The mapping can then be unloaded via **bus_dmamap_unload**(), and the mapping object can be returned to the pool of unused mapping objects.

When a consumer is no longer scheduling DMA transactions, the mapping objects should be freed via **bus_dmamap_destroy**(), and the tag should be freed via **bus_dma_tag_destroy**().

STRUCTURES AND TYPES

bus_dma_tag_t

A machine-dependent (MD) opaque type that describes the characteristics of a group 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_template_t

A template is a structure for creating a *bus_dma_tag_t* from a set of defaults. Once initialized with **bus_dma_template_init**(), a driver can over-ride individual fields to suit its needs. The following fields start with the indicated default values:

alignment	1
boundary	0
lowaddr	BUS_SPACE_MAXADDR
highaddr	BUS_SPACE_MAXADDR
maxsize	BUS_SPACE_MAXSIZE
nsegments	BUS_SPACE_UNRESTRICTED
maxsegsize	BUS_SPACE_MAXSIZE
flags	0
lockfunc	NULL
lockfuncarg	NULL

Descriptions of each field are documented with **bus_dma_tag_create**(). Note that the *filtfunc* and *filtfuncarg* attributes of the DMA tag are not supported with templates.

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* argument is specified by the client 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 if any mapping in this range can be accommodated by the device and non-zero otherwise.

Note: The use of filters is deprecated. Proper operation is not guaranteed.

bus_dma_segment_t

A machine-dependent type that describes individual DMA segments. It contains the following fields:

```
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. One map is used for each memory allocation that will be loaded. Maps can be reused once they have been unloaded. Multiple maps can be associated with one DMA tag. While the value of the map may evaluate to NULL on some platforms under certain conditions, it should never be assumed that it will be NULL in all cases.

bus_dmamap_callback_t

Client specified callback for receiving mapping information resulting from the load of a *bus_dmamap_t* via **bus_dmamap_load**(), **bus_dmamap_load_bio**(), **bus_dmamap_load_ccb**(), **bus_dmamap_load_crp**(), or **bus_dmamap_load_crp_buffer**(). 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* arguments 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* argument. 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. The operations are represented as bitfield flags that can be combined together, though it only makes sense to combine PRE flags or POST flags, not both. See the **bus_dmamap_sync**() description below for more details on how to use these operations.

All operations specified below are performed from the host memory point of view, where a read implies data coming from the device to the host memory, and a write implies data going from the host memory to the device. Alternatively, the operations can be thought of in terms of driver operations, where reading a network packet or storage sector corresponds to a read operation in **bus_dma**.

BUS_DMASYNC_PREREAD	Perform any synchronization required prior to an update of host memory by the device.
BUS_DMASYNC_PREWRITE	Perform any synchronization required after an update of host memory by the CPU and prior to device access to host memory.
BUS_DMASYNC_POSTREAD	Perform any synchronization required after an update of host memory by the device and prior to CPU access to host memory.
BUS_DMASYNC_POSTWRITE	Perform any synchronization required after device access to host memory.

bus_dma_lock_t

Client specified lock/mutex manipulation method. This will be called from within busdma whenever a client lock needs to be manipulated. In its current form, the function will be called immediately before the callback for a DMA load operation that has been deferred with BUS_DMA_LOCK and immediately after with BUS_DMA_UNLOCK. If the load operation does not need to be deferred, then it will not be called since the function loading the map should be holding the appropriate locks. This method is of the format:

void **lockfunc**(*void *lockfunc_arg, bus_dma_lock_op_t op*)

The *lockfuncarg* argument is specified by the client during tag creation to be passed to all invocations of the callback. The *op* argument specifies the lock operation to perform.

Two *lockfunc* implementations are provided for convenience. **busdma_lock_mutex**() performs standard mutex operations on the sleep mutex provided via *lockfuncarg*. **dflt_lock**() will generate a system panic if it is called. It is substituted into the tag when *lockfunc* is passed as NULL to **bus_dma_tag_create**() and is useful for tags that should not be used with deferred load operations.

bus_dma_lock_op_t

Operations to be performed by the client-specified **lockfunc**().

BUS_DMA_LOCK Acquires and/or locks the client locking primitive.

BUS_DMA_UNLOCK Releases and/or unlocks the client locking primitive.

FUNCTIONS

bus_dma_tag_create(*parent, alignment, boundary, lowaddr, highaddr, *filtfunc, *filtfuncarg, maxsize, nsegments, maxsegsz, flags, lockfunc, lockfuncarg, *dmat*)

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

parent A parent tag from which to inherit restrictions. The restrictions passed in other arguments can only further tighten the restrictions inherited from the parent tag.

All tags created by a device driver must inherit from the tag returned by **bus_get_dma_tag**() to honor restrictions between the parent bridge, CPU memory, and the device.

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 a power of 2 and must be no smaller than the maximum segment size. '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 addresses 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 perform DMA to addresses below 16MB would specify a *highaddr* of BUS_SPACE_MAXADDR and a *lowaddr* of BUS_SPACE_MAXADDR_24BIT. Some implementations require 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.

Note: The use of filters is deprecated. Proper operation is not guaranteed.

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

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.

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

flags Are as follows:

BUS_DMA_ALLOCNOW Pre-allocate enough resources to handle at least one map load operation on this tag. If sufficient resources are not available, ENOMEM is returned. This should not be used for tags that only describe buffers that will be allocated with **bus_dmamem_alloc**(). Also, due to resource sharing with other tags, this flag does not guarantee that resources will be allocated or reserved exclusively for this tag. It should be treated only as a minor optimization.

BUS_DMA_COHERENT Indicate that the DMA engine and CPU are cache-coherent. Cached memory may be used to back allocations created by **bus_dmamem_alloc**(). For **bus_dma_tag_create**(), the BUS_DMA_COHERENT flag is currently implemented on arm64.

lockfunc Optional lock manipulation function (may be NULL) to be called when busdma needs to manipulate a lock on behalf of the client. If NULL is specified, **dflt_lock**() is used.

lockfuncarg
Optional argument to be passed to the function specified by *lockfunc*.

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

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

bus_dma_template_init(**template*, *parent*)
Initializes a *bus_dma_template_t* structure. If the *parent* argument is non-NULL, this parent tag is associated with the template and will be compiled into the dma tag that is later created. The values of the parent are not copied into the template. During tag creation in **bus_dma_tag_template**(), any parameters from the parent tag that are more restrictive than what

is in the provided template will overwrite what goes into the new tag.

bus_dma_template_tag(**template*, **dmat*)

Unpacks a template into a tag, and returns the tag via the *dmat*. All return values are identical to **bus_dma_tag_create**(). The template is not modified by this function, and can be reused and/or freed upon return.

bus_dma_template_clone(**template*, *dmat*)

Copies the fields from an existing tag to a template. The template does not need to be initialized first. All of its fields will be overwritten by the values contained in the tag. When paired with **bus_dma_template_tag**(), this function is useful for creating copies of tags.

bus_dma_template_fill(**template*, *params[]*, *count*)

Fills in the selected fields of the template with the keyed values from the *params* array. This is not meant to be called directly, use **BUS_DMA_TEMPLATE_FILL**() instead.

BUS_DMA_TEMPLATE_FILL(**template*, *param* ...)

Fills in the selected fields of the template with a variable number of key-value parameters. The macros listed below take an argument of the specified type and encapsulate it into a key-value structure that is directly usable as a parameter argument. Muliple parameters may be provided at once.

BD_PARENT()	void *
BD_ALIGNMENT()	uintmax_t
BD_BOUNDARY()	uintmax_t
BD_LOWADDR()	vm_paddr_t
BD_HIGHADDR()	vm_paddr_t
BD_MAXSIZE()	uintmax_t
BD_NSEGMENTS()	uintmax_t
BD_MAXSEGSIZE()	uintmax_t
BD_FLAGS()	uintmax_t
BD_LOCKFUNC()	void *
BD_LOCKFUNCARG()	void *

bus_dmamap_create(*dmat*, *flags*, **mapp*)

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

dmat DMA tag.

flags Are as follows:

BUS_DMA_COHERENT Attempt to map the memory loaded with this map such that cache sync operations are as cheap as possible. This flag is typically set on maps when the memory loaded with these will be accessed by both a CPU and a DMA engine, frequently such as control data and as opposed to streamable data such as receive and transmit buffers. 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 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(*dmat*, *map*)

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

dmat 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(*dmat*, *map*, *buf*, *buflen*, **callback*, *callback_arg*, *flags*)

Creates a mapping in device visible address space of *buflen* bytes of *buf*, associated with the DMA map *map*. This call will always return immediately and will not block for any reason. Arguments are as follows:

dmat 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. This function is called once sufficient mapping

resources are available for the DMA operation. If resources are temporarily unavailable, this function will be deferred until later, but the load operation will still return immediately to the caller. Thus, callers should not assume that the callback will be called before the load returns, and code should be structured appropriately to handle this. See below for specific flags and error codes that control this behavior.

flags Are as follows:

BUS_DMA_NOWAIT The load should not be deferred in case of insufficient mapping resources, and instead should return immediately with an appropriate error.

BUS_DMA_NOCACHE
The generated transactions to and from the virtual page are non-cacheable.

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.

Note that subsequent load operations for the same tag that do not require extra resources will still succeed. This may result in out-of-order processing of requests. If the caller requires the order of requests to be preserved, then the caller is required to stall subsequent requests until a pending request's callback is invoked.

ENOMEM The load request has failed due to insufficient resources, and the caller specifically used the BUS_DMA_NOWAIT flag.

EINVAL The load request was invalid. The callback has been called and has been provided the same error. This error value may indicate that *dmat*, *map*, *buf*, or *callback* were invalid, or *buflen* was larger than the *maxsize* argument used to create the dma tag *dmat*.

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_bio(*dmat*, *map*, *bio*, *callback*, *callback_arg*, *flags*)
This is a variation of **bus_dmamap_load**() which maps buffers pointed to by *bio* for DMA transfers. *bio* may point to either a mapped or unmapped buffer.

bus_dmamap_load_ccb(*dmat*, *map*, *ccb*, *callback*, *callback_arg*, *flags*)
This is a variation of **bus_dmamap_load**() which maps data pointed to by *ccb* for DMA transfers. The data for *ccb* may be any of the following types:

CAM_DATA_VADDR The data is a single KVA buffer.

CAM_DATA_PADDR The data is a single bus address range.

CAM_DATA_SG The data is a scatter/gather list of KVA buffers.

CAM_DATA_SG_PADDR The data is a scatter/gather list of bus address ranges.

CAM_DATA_BIO The data is contained in a *struct bio* attached to the CCB.

bus_dmamap_load_ccb() supports the following CCB XPT function codes:

XPT_ATA_IO
XPT_CONT_TARGET_IO
XPT_SCSI_IO

bus_dmamap_load_crp(*dmat*, *map*, *crp*, *callback*, *callback_arg*, *flags*)
This is a variation of **bus_dmamap_load**() which maps the input buffer pointed to by *crp* for DMA transfers. The BUS_DMA_NOWAIT flag is implied, thus no callback deferral will happen.

bus_dmamap_load_crp_buffer(*dmat*, *map*, *cb*, *callback*, *callback_arg*, *flags*)
This is a variation of **bus_dmamap_load**() which maps the crypto data buffer pointed to by *cb* for

DMA transfers. The BUS_DMA_NOWAIT flag is implied, thus no callback deferral will happen.

bus_dmamap_load_mbuf(*dmat*, *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. The BUS_DMA_NOWAIT flag is implied, thus no callback deferral will happen.

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

Beside the error values listed for **bus_dmamap_load**(), EINVAL will be returned if the size of the mbuf chain exceeds the maximum limit of the DMA tag.

bus_dmamap_load_mbuf_sg(*dmat*, *map*, *mbuf*, *segs*, *nsegs*, *flags*)

This is just like **bus_dmamap_load_mbuf**() except that it returns immediately without calling a callback function. It is provided for efficiency. The scatter/gather segment array *segs* is provided by the caller and filled in directly by the function. The *nsegs* argument is returned with the number of segments filled in. Returns the same errors as **bus_dmamap_load_mbuf**().

bus_dmamap_load_uio(*dmat*, *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*. The BUS_DMA_NOWAIT flag is implied, thus no callback deferral will happen. Returns the same errors as **bus_dmamap_load**().

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. Pages may be locked using vslock(9).

bus_dmamap_unload(*dmat*, *map*)

Unloads a DMA map. Arguments are as follows:

dmat 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(*dmat*, *map*, *op*)

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

dmat 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*.

The **bus_dmamap_sync**() function is the method used to ensure that CPU's and device's direct memory access (DMA) to shared memory is coherent. For example, the CPU might be used to set up the contents of a buffer that is to be made available to 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_PREWRITE must be performed after the CPU has updated the buffer and before the device access is initiated. If the CPU modifies this buffer again later, another BUS_DMASYNC_PREWRITE sync operation must be performed before an additional device access. Conversely, suppose a device updates memory that is to be read by a CPU. In this case, the buffer must be loaded, and a DMA sync operation of BUS_DMASYNC_PREREAD must be performed before the device access is initiated. The CPU will only be able to see the results of this memory update once the DMA operation has completed and a BUS_DMASYNC_POSTREAD sync operation has been performed.

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

bus_dmamem_alloc(*dmat*, ***vaddr*, *flags*, **mapp*)

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

dmat 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 in a coherent fashion. See **bus_dmamap_create**() above for a description of this flag. For **bus_dmamem_alloc**(), the BUS_DMA_COHERENT flag is currently implemented on arm and arm64.

BUS_DMA_ZERO Causes the allocated memory to be set to all zeros.

BUS_DMA_NOCACHE
The allocated memory will not be cached in the processor caches. All memory accesses appear on the bus and are executed without reordering. For **bus_dmamem_alloc**(), the BUS_DMA_NOCACHE flag is currently implemented on amd64 and i386 where it results in the Strong Uncacheable PAT to be set for the allocated virtual address range.

mapp Pointer to a *bus_dmamap_t* where the resulting DMA map will be stored.

The size of memory to be allocated is *maxsize* as specified in the call to **bus_dma_tag_create**() for *dmat*.

The current implementation of **bus_dmamem_alloc**() will allocate all requests as a single segment.

An initial load operation is required to obtain the bus address of the allocated memory, and an unload operation is required before freeing the memory, as described below in **bus_dmamem_free**(). Maps are automatically handled by this function and should not be explicitly allocated or destroyed.

Although an explicit load is not required for each access to the memory referenced by the returned map, the synchronization requirements as described in the **bus_dmamap_sync**() section still apply and should be used to achieve portability on architectures without coherent buses.

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

bus_dmamem_free(*dmat*, **vaddr*, *map*)
Frees memory previously allocated by **bus_dmamem_alloc**(). Any mappings will be invalidated.

Arguments are as follows:

dmat 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 on failure as discussed above.

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

LOCKING

Two locking protocols are used by **bus_dma**. The first is a private global lock that is used to synchronize access to the bounce buffer pool on the architectures that make use of them. This lock is strictly a leaf lock that is only used internally to **bus_dma** and is not exposed to clients of the API.

The second protocol involves protecting various resources stored in the tag. Since almost all **bus_dma** operations are done through requests from the driver that created the tag, the most efficient way to protect the tag resources is through the lock that the driver uses. In cases where **bus_dma** acts on its own without being called by the driver, the lock primitive specified in the tag is acquired and released automatically. An example of this is when the **bus_dmamap_load**() callback function is called from a deferred context instead of the driver context. This means that certain **bus_dma** functions must always be called with the same lock held that is specified in the tag. These functions include:

bus_dmamap_load()
bus_dmamap_load_bio()
bus_dmamap_load_ccb()
bus_dmamap_load_mbuf()
bus_dmamap_load_mbuf_sg()
bus_dmamap_load_uio()
bus_dmamap_unload()
bus_dmamap_sync()

There is one exception to this rule. It is common practice to call some of these functions during driver start-up without any locks held. So long as there is a guarantee of no possible concurrent use of the tag by different threads during this operation, it is safe to not hold a lock for these functions.

Certain **bus_dma** operations should not be called with the driver lock held, either because they are already protected by an internal lock, or because they might sleep due to memory or resource allocation. The following functions must not be called with any non-sleepable locks held:

bus_dma_tag_create()
bus_dmamap_create()
bus_dmamem_alloc()

All other functions do not have a locking protocol and can thus be called with or without any system or driver locks held.

SEE ALSO

devclass(9), device(9), driver(9), rman(9), vslock(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.