NAME

ddi_dmae, ddi_dmae_alloc, ddi_dmae_release, ddi_dmae_prog, ddi_dmae_disable, ddi_dmae_enable, ddi_dmae_stop, ddi_dmae_getcnt, ddi_dmae_1stparty, ddi_dmae_getlim, ddi_dmae_getattr – system DMA engine functions

SYNOPSIS

int ddi_dmae_alloc(dev_info_t **dip*, **int** *chnl*, **int (****dmae_waitfp***)( )**, **caddr_t** *arg***);**

int ddi_dmae_release(dev_info_t **dip*, **int** *chnl***);**

int ddi_dmae_prog(dev_info_t **dip*, **struct ddi_dmae_req** **dmaereqp*,
ddi_dma_cookie_t **cookiep*, **int** *chnl***);**

int ddi_dmae_disable(dev_info_t **dip*, **int** *chnl***);**

int ddi_dmae_enable(dev_info_t **dip*, **int** *chnl***);**

int ddi_dmae_stop(dev_info_t **dip*, **int** *chnl***);**

int ddi_dmae_getcnt(dev_info_t **dip*, **int** *chnl*, **int** **countp***);**

int ddi_dmae_1stparty(dev_info_t **dip*, **int** *chnl***);**

int ddi_dmae_getlim(dev_info_t **dip*, **ddi_dma_lim_t** **limitsp***);**

int ddi_dmae_getattr(dev_info_t **dip***, ddi_dma_attr_t** **attrp***);**

AVAILABILITY

x86

INTERFACE LEVEL

Solaris x86 DDI specific (Solaris x86 DDI).

ARGUMENTS

dip	A **dev_info** pointer, which identifies the device.
chnl	A DMA channel number, or an MCA bus arbitration level. On ISA or EISA buses this number must be **0**, **1**, **2**, **3**, **5**, **6**, or **7**. On MCA buses this number must be in the range **0** to **14**.
dmae_waitfp	A wait/callback function address.
arg	The argument to be passed to the callback function.
dmaereqp	A pointer to a DMA engine request (**ddi_dmae_req**(9S)) structure.
cookiep	A pointer to a **ddi_dma_cookie**(9S) object, obtained from **ddi_dma_segtocookie**(9F), which contains the address and count.
countp	A pointer to an integer that will receive the count of the number of bytes not yet transferred upon completion of a DMA operation.
limitsp	A pointer to a DMA limit structure. (see **ddi_dma_lim_x86**(9S)).
attrp	A pointer to a DMA attribute structure (see **ddi_dma_attr**(9S)).

DESCRIPTION

There are three possible ways that a device can perform DMA engine functions.

"Bus master DMA" If the device is capable of acting as a true bus master, then the driver should program the device's DMA registers directly and not make use of the DMA engine functions described here. The driver should obtain the DMA address and count from

ddi_dma_segtocookie( ). See **ddi_dma_cookie**(9S) for a description of a DMA cookie.

"Third-party DMA" This method uses the system DMA engine that is resident on the main system board. In this model, the device cooperates with the system's DMA engine to effect the data transfers between the device and memory. The driver uses the functions documented here, except **ddi_dmae_1stparty( )**, to initialize and program the DMA engine. For each DMA data transfer, the driver programs the DMA engine and then gives the device a command to initiate the transfer in cooperation with that engine.

"First-party DMA" Using this method, the device uses its own DMA bus cycles, but requires a channel from the system's DMA engine. After allocating the DMA channel, the **ddi_dmae_1stparty( )** function may be used to perform whatever configuration is necessary to enable this mode.

ddi_dmae_alloc( )

The **ddi_dmae_alloc( )** function is used to acquire a DMA channel of the system DMA engine. **ddi_dmae_alloc( )** allows only one device at a time to have a particular DMA channel allocated. It must be called prior to any other system DMA engine function on a channel. If the device allows the channel to be shared with other devices, it must be freed using **ddi_dmae_release( )** after completion of the DMA operation. In any case the channel must be released before the driver successfully detaches. See **detach**(9E). No other driver may acquire the DMA channel until it is released.

If the requested channel is not immediately available, the value of *dmae_waitfp* determines what action will be taken. If the value of *dmae_waitfp* is **DDI_DMA_DONTWAIT, ddi_dmae_alloc( )** will return immediately. The value **DDI_DMA_SLEEP** will cause the thread to sleep and not return until the channel has been acquired. Any other value is assumed to be a callback function address. In that case, **ddi_dmae_alloc( )** returns immediately, and when resources might have become available, the callback function is called (with the argument *arg*) from interrupt context.

When the callback function **(****dmae_waitfp***)( )** is called, it should attempt to allocate the DMA channel again. If it succeeds or does not need the channel any more, it must return the value **0**. If it tries to allocate the channel, but fails to do so, it must return the value **0**.

ddi_dmae_prog( )

The **ddi_dmae_prog( )** function programs the DMA channel for a DMA transfer. The **ddi_dmae_req** structure contains all the information necessary to set up the channel, except for the memory address and count. Once the channel has been programmed, subsequent calls to **ddi_dmae_prog( )** may specify a value of **NULL** for *dmaereqp* if no changes to the programming are required other than the address and count values. It disables the channel prior to setup, and enables the channel before returning. The DMA address and count are specified by passing **ddi_dmae_prog( )** a cookie obtained from **ddi_dma_segtocookie( )**. Other DMA engine parameters are specified by the DMA engine request structure passed in through *dmaereqp*. The fields of that structure are documented in **ddi_dmae_req**(9S).

Before using **ddi_dmae_prog( )**, you must allocate system DMA resources using DMA setup functions such as **ddi_dma_buf_setup**(9F). **ddi_dma_segtocookie( )** can then be used to retrieve a cookie which contains the address and count. Then this cookie is passed to **ddi_dmae_prog( )**.

ddi_dmae_disable( ) The **ddi_dmae_disable( )** function disables the DMA channel so that it no longer responds to a device's DMA service requests.

ddi_dmae_enable( ) The **ddi_dmae_enable( )** function enables the DMA channel for operation. This may be used to re-enable the channel after a call to **ddi_dmae_disable( )**. The channel is automatically enabled after successful programming by **ddi_dmae_prog( ).**

ddi_dmae_stop( ) The **ddi_dmae_stop( )** function disables the channel and terminates any active operation.

ddi_dmae_getcnt( ) The **ddi_dmae_getcnt( )** function examines the count register of the DMA channel and sets **countp* to the number of bytes remaining to be transferred. The channel is assumed to be stopped.

ddi_dmae_1stparty( ) In the case of ISA and EISA buses, **ddi_dmae_1stparty( )** configures a channel in the system's DMA engine to operate in a ''slave'' (''cascade'') mode.

In the case of the MCA bus, a call to **ddi_dmae_1stparty( )** should still be made, regardless of whether the channel number specifies one of the DMA arbitration levels or a non-DMA arbitration level.

When operating in **ddi_dmae_1stparty( )** mode, the DMA channel must first be allocated using **ddi_dmae_alloc( )** and then configured using **ddi_dmae_1stparty( )**. The driver then programs the device to perform the I/O, including the necessary DMA address and count values obtained from **ddi_dma_segtocookie( )**.

ddi_dmae_getlim( ) The **ddi_dmae_getlim( )** function fills in the DMA limit structure, pointed to by *limitsp*, with the DMA limits of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA limit structures; they should not use **ddi_dmae_getlim( )**. The DMA limit structure must be passed to the DMA setup routines so that they will know how to break the DMA request into windows and segments (see **ddi_dma_nextseg**(9F) and **ddi_dma_nextwin**(9F)). If the device has any particular restrictions on transfer size or granularity (such as the size of disk sector), the driver should further restrict the values in the structure members before passing them to the DMA setup routines. The driver must not relax any of the restrictions embodied in the structure after it is filled in by **ddi_dmae_getlim( )**. After calling **ddi_dmae_getlim( )**, a driver must examine, and possibly set, the size of the DMA engine's scatter/gather list to determine whether DMA chaining will be used. See **ddi_dma_lim_x86**(9S) and **ddi_dmae_req**(9S) for additional information on scatter/gather DMA.

ddi_dmae_getattr The **ddi_dmae_getattr( )** function fills in the DMA attribute structure, pointed to by *attrp*, with the DMA attributes of the system DMA engine. Drivers for devices that perform their own bus mastering or use first-party DMA must create and initialize their own DMA attribute structures; they should not use **ddi_dmae_getattr( )**. The DMA attribute structure must be passed to the DMA resource allocation functions to provide the information necessary to break the DMA request into DMA windows and DMA cookies (see **ddi_dma_nextcookie**(9F) and **ddi_dma_getwin**(9F)).

RETURN VALUES

DDI_SUCCESS Upon success, for all of these routines.

DDI_FAILURE May be returned due to invalid arguments.

DDI_DMA_NORESOURCES
May be returned by **ddi_dmae_alloc( )** if the requested resources are not available and the value of *dmae_waitfp* is not **DDI_DMA_SLEEP**.

CONTEXT

If **ddi_dmae_alloc( )** is called from interrupt context, then its *dmae_waitfp* argument and the callback function must not have the value **DDI_DMA_SLEEP**. Otherwise, all these routines may be called from user or interrupt context.

SEE ALSO

eisa(4), **isa**(4), **mca**(4), **ddi_dma_buf_setup**(9F), **ddi_dma_nextseg**(9F), **ddi_dma_nextwin**(9F), **ddi_dma_req**(9S), **ddi_dma_segtocookie**(9F), **ddi_dma_setup**(9F), **ddi_dma_attr**(9S), **ddi_dma_cookie**(9S), **ddi_dma_lim_x86**(9S), **ddi_dmae_req**(9S)