NAME

ddi_dmae_req – DMA engine request structure

SYNOPSIS

#include <sys/dma_engine.h>

AVAILABILITY

x86

INTERFACE LEVEL

Solaris x86 DDI specific (Solaris x86 DDI).

DESCRIPTION

A **ddi_dmae_req** structure is used by a device driver to describe the parameters for a DMA channel. This structure contains all the information necessary to set up the channel, except for the DMA memory address and transfer count. The defaults as specified below support most standard devices. Other modes may be desirable for some devices, or to increase performance. The DMA engine request structure is passed to **ddi_dmae_prog**(9F).

STRUCTURE MEMBERS

The **ddi_dmae_req** structure contains several members, each of which controls some aspect of DMA engine operation. The structure members associated with supported DMA engine options are described here.

```
uchar_t            der_command;      /* Read / Write */
uchar_t            der_bufprocess;   /* Standard / Chain */
uchar_t            der_path;         /* 8 / 16 / 32 */
u_short            der_ioadr;        /* MicroChannel I/O address */
uchar_t            der_cycles;       /* Compat / Type A / Type B / Burst */
uchar_t            der_trans;        /* Single / Demand / Block */
ddi_dma_cookie_t   *(*proc)( );      /* address of nextcookie routine */
void               *procparms;       /* parameter for nextcookie call */
```

der_command specifies what DMA operation is to be performed. The value **DMAE_CMD_WRITE** signifies that data is to be transferred from memory to the I/O device. The value **DMAE_CMD_READ** signifies that data is to be transferred from the I/O device to memory. This field must be set by the driver before calling **ddi_dmae_prog( )**.

der_bufprocess On some bus types, a driver may set **der_bufprocess** to the value **DMAE_BUF_CHAIN** to specify that multiple DMA cookies will be given to the DMA engine for a single I/O transfer, thus effecting a scatter/gather operation. In this mode of operation, the driver calls **ddi_dmae_prog( )** to give the DMA engine the DMA engine request structure and a pointer to the first cookie. The **proc** structure member must be set to the address of a driver **nextcookie** routine that takes one argument, specified by the **procparms** structure member, and returns a pointer to a structure of type **ddi_dma_cookie_t** that specifies the next cookie for the I/O transfer. When the DMA engine is ready to receive an additional cookie, the bus nexus driver controlling that DMA engine calls the routine specified by the **proc** structure member to obtain the next cookie from the driver. The driver's nextcookie routine must then return the address of the next cookie (in static storage) to the bus nexus routine that called it. If there are no more segments in the current DMA window, then **(*proc)( )** must return the NULL pointer.

A driver may only specify the **DMAE_BUF_CHAIN** flag if the particular bus architecture supports the use of multiple DMA cookies in a single I/O transfer. A bus DMA engine may support this feature either with a fixed-length scatter/gather list, or via an interrupt chaining feature such as the one implemented in the EISA architecture. A driver must ascertain whether its parent bus nexus supports this feature by examining the scatter/gather list size returned in the **dlim_sgllen** member of the DMA limit structure (see **ddi_dma_lim_x86**(9S)) returned by the driver's call to **ddi_dmae_getlim( )**. If the size of the scatter/gather list is 1, then no chaining is available, the driver *must not* specify the **DMAE_BUF_CHAIN** flag in the **ddi_dmae_req** structure it passes to **ddi_dmae_prog( ),** and the driver need not provide a nextcookie routine.

If the size of the scatter/gather list is greater than 1, then DMA chaining is available, and the driver has two options. Under the first option, the driver chooses not to use the chaining feature, in which case (a) the driver must *set* the size of the scatter/gather list to 1 *before* passing it to the DMA setup routine, and (b) the driver must *not* set the **DMAE_BUF_CHAIN** flag.

Under the second option, the driver chooses to use the chaining feature, in which case (a) it should leave the size of the scatter/gather list alone, and (b) it *must* set the **DMAE_BUF_CHAIN** flag in the **ddi_dmae_req** structure. Before calling **ddi_dmae_prog( )** the driver must *prefetch* cookies by repeatedly calling **ddi_dma_nextseg**(9F) and **ddi_dma_segtocookie**(9F) until either (1) the end of the DMA window is reached (**ddi_dma_nextseg**(9F) returns NULL), or (2) the size of the scatter/gather list is reached, whichever occurs first. These cookies must be saved by the driver until they are requested by the nexus driver calling the driver's nextcookie routine. The driver's nextcookie routine must return the prefetched cookies, in order, one cookie for each call to the nextcookie routine, until the list of prefetched cookies is exhausted. After the end of the list of cookies is reached, the nextcookie routine must return the NULL pointer.

The size of the scatter/gather list determines how many discontiguous segments of physical memory may participate in a single DMA transfer. ISA and MCA bus DMA engines have no scatter/gather capability, so their scatter/gather list sizes are 1. EISA bus DMA engines have a DMA chaining interrupt facility that allows very large scatter/gather operations. Other finite scatter/gather list sizes would also be possible. For performance reasons, it is recommended that drivers use the chaining capability if it is available on their parent bus.

As described above, a driver making use of DMA chaining must pre-fetch DMA cookies before calling **ddi_dmae_prog( )**. There are two reasons why the driver must do this. First, the driver must have some way to know the total I/O count with which to program the I/O device. This I/O count must match the total size of all the DMA segments that will be chained together into one DMA operation. Depending on the size of the scatter/gather list and the memory position and alignment of the DMA object, all or just part of the current DMA window may be able to participate in a single I/O operation. The driver must compute the I/O count by adding up the sizes of the prefetched DMA cookies. The number of cookies whose sizes are to be summed is the lesser of (a) the size of the scatter/gather list, or (b) the number of segments remaining in the window. Second, on some bus architectures, the driver's nextcookie routine may be called from a high-level interrupt routine. If the cookies were not prefetched, the nextcookie routine would have to call **ddi_dma_nextseg( )** and **ddi_dma_segtocookie( )** from a high-level interrupt routine, which is not recommended.

When breaking a DMA window into segments, the system arranges that the end of every segment whose number is an integral multiple of the scatter/gather list size will fall on a device-granularity boundary (as specified in the **dlim_granular** field in the **ddi_dma_lim_x86**(9S) structure).

If the scatter/gather list size is 1 (either because no chaining is available or because the driver does not wish to use the chaining feature), then the total I/O count for a single DMA operation is simply the size of DMA segment denoted by the single DMA cookie that is passed in the call to **ddi_dmae_prog( ) .** In this case, the system arranges that each DMA segment is a multiple of the device-granularity size.

der_path specifies the DMA transfer size. The default of zero (**DMAE_PATH_DEF**) specifies ISA compatibility mode. In that mode, channels 0, 1, 2, and 3 are programmed in 8-bit mode (**DMAE_PATH_8**), and channels 5, 6, and 7 are programmed in 16-bit, count-by-word mode (**DMAE_PATH_16**). On the EISA bus, other sizes may be specified: **DMAE_PATH_32** specifies 32-bit mode, and **DMAE_PATH_16B** specifies a 16-bit, count-by-byte mode. MCA channel 4 must be explicitly programmed with **DMAE_PATH_8** or **DMAE_PATH_16**.

der_ioadr only applicable to devices using MicroChannel DMA services, and if non-zero, specifies the MicroChannel DMA I/O address register value. This register causes the MicroChannel DMA controller to present the I/O address on the bus during DMA cycles; thus a DMA slave device can be made to respond to the I/O request by decoding the address and control buses rather than the bus arbitration level. Set **der_ioadr** to the I/O address of the device being accessed through DMA if the device operates in this way.

der_cycles specifies the timing mode to be used during DMA data transfers. The default of zero (**DMAE_CYCLES_1**) specifies ISA compatible timing. Drivers using this mode must also specify **DMAE_TRANS_SNGL** in the **der_trans** structure member. On EISA buses, these other timing modes are available:

DMAE_CYCLES_2 specifies type ''A'' timing;

DMAE_CYCLES_3 specifies type ''B'' timing;

DMAE_CYCLES_4 specifies ''Burst'' timing.

der_trans specifies the bus transfer mode that the DMA engine should expect from the device. The default value of zero (**DMAE_TRANS_SNGL**) specifies that the device will perform one transfer for each bus arbitration cycle. Devices that use ISA compatible timing (specified by a value of zero, which is the default, in the **der_cycles** structure member) should use the **DMAE_TRANS_SNGL** mode.

On EISA buses, a **der_trans** value of **DMAE_TRANS_BLCK** specifies that the device will perform a block of transfers for each arbitration cycle. A value of **DMAE_TRANS_DMND** specifies that the device will perform the Demand Transfer Mode protocol.

SEE ALSO

eisa(4), **isa**(4), **mca**(4), **ddi_dma_segtocookie**(9F), **ddi_dmae**(9F), **ddi_dma_lim_x86**(9S), **ddi_dma_req**(9S)