NAME

ddi_dma_req – DMA Request structure

SYNOPSIS

#include <sys/ddidmareq.h>

INTERFACE LEVEL

Solaris DDI specific (Solaris DDI).

DESCRIPTION

A **ddi_dma_req** structure describes a request for DMA resources. A driver may use it to describe forms of and ways to allocate DMA resources for a DMA request.

STRUCTURE MEMBERS

```
ddi_dma_lim_t   *dmar_limits;        /* Caller's dma engine's */
                                     /* constraints */
u_int           dmar_flags;          /* Contains information for */
                                     /* mapping routines */
int             (*dmar_fp)(caddr_t); /* Callback function */
caddr_t         dmar_arg;            /* Callback function's argument */
ddi_dma_obj_t   dmar_object;         /* Description of the object */
                                     /* to be mapped */
```

For the definition of the DMA limits structure, which **dmar_limits** points to, see **ddi_dma_lim_sparc**(9S) or **ddi_dma_lim_x86**(9S).

Valid values for **dmar_flags** are:

```
DDI_DMA_WRITE        /* Direction memory --> IO */
DDI_DMA_READ         /* Direction IO --> memory */
DDI_DMA_RDWR         /* Both read and write */
DDI_DMA_REDZONE      /* Establish an MMU redzone at end of mapping */
DDI_DMA_PARTIAL      /* Partial mapping is allowed */
DDI_DMA_CONSISTENT   /* Byte consistent access wanted */
DDI_DMA_SBUS_64BIT   /* Use 64 bit capability on SBus */
```

DDI_DMA_WRITE, **DDI_DMA_READ** and **DDI_DMA_RDWR** describe the intended direction of the DMA transfer. Some implementations may explicitly disallow **DDI_DMA_RDWR.**

DDI_DMA_REDZONE asks the system to establish a protected *red zone* after the object. The DMA resource allocation functions do not guarantee the success of this request as some implementations may not have the hardware ability to support it.

DDI_DMA_PARTIAL tells the system that the caller can accept a partial mapping. That is, if the size of the object exceeds the resources available, only allocate a portion of the object and return status indicating so. At a later point, the caller can use **ddi_dma_curwin**(9F) and **ddi_dma_movwin**(9F) to change the valid portion of the object that has resources allocated.

DDI_DMA_CONSISTENT gives a hint to the system that the object should be mapped for *byte consistent* access. Normal data transfers usually use a *streaming* mode of operation. They start at a specific point, transfer a fairly large amount of data sequentially, and then stop usually on a aligned boundary. Control mode data transfers for memory resident device control blocks (for example ethernet message descriptors) do not access memory in such a sequential fashion. Instead, they tend to modify a few words or bytes, move around and maybe modify a few more. There are many machine implementations that make this difficult to control in a generic and seamless fashion. Therefore, explicit synchronization steps using **ddi_dma_sync**(9F) or **ddi_dma_free**(9F) are required in order to make the view of a memory object shared between a CPU and a DMA device consistent. However, proper use of the **DDI_DMA_CONSISTENT** flag gives a *hint* to the system so that it will attempt to pick resources such that these synchronization steps are as efficient as possible.

DDI_DMA_SBUS_64BIT tells the system that the device can do 64 bit transfers on a 64 bit SBus. If the SBus does not support 64 bit data transfers, data will be transferred in 32 mode.

The callback function specified by the member **dmar_fp** indicates how a caller to one of the DMA resource allocation functions (see **ddi_dma_setup**(9F)) wants to deal with the possibility of resources not being available. If **dmar_fp** is set to **DDI_DMA_DONTWAIT**, then the caller does not care if the allocation fails, and can deal with an allocation failure appropriately. If **dmar_fp** is set to **DDI_DMA_SLEEP**, then the caller wishes to have the the allocation routines wait for resources to become available. If any other value is set, and a DMA resource allocation fails, this value is assumed to be a function to call at a later time when resources may become available. When the specified function is called, it is passed the value set in the structure member **dmar_arg**. The specified callback function *must* return either 0 (indicating that it attempted to allocate a DMA resources but failed to do so, again), in which case the callback function will be put back on a list to be called again later, or the callback function must return 1 indicating either success at allocating DMA resources or that it no longer wishes to retry.

The callback function will be called in interrupt context. Therefore, only system functions and contexts that are accessible from interrupt context will be available. The callback function must take whatever steps necessary to protect its critical resources, data structures, queues, so forth.

Note that it is possible that a call to **ddi_dma_free**(9F), which frees DMA resources, may cause a callback function to be called, and unless some care is taken an undesired recursion may occur. Unless care is taken, this may cause an undesired recursive **mutex_enter**(9F), which will cause a system panic.

dmar_object Structure

The **dmar_object** member of the **ddi_dma_req** structure is itself a complex and extensible structure:

```
u_int            dmao_size;    /* size, in bytes, of the object */
ddi_dma_atyp_t   dmao_type;    /* type of object */
ddi_dma_aobj_t   dmao_obj;     /* the object described */
```

The **dmao_size** element is the size, in bytes, of the object resources are allocated for DMA.

The **dmao_type** element selects the *kind* of object described by **dmao_obj**. It may be set to **DMA_OTYP_VADDR** indicating virtual addresses.

The last element, **dmao_obj**, consists of the virtual address type:

struct v_address virt_obj;

It is specified as:

```
struct v_address {
        caddr_t v_addr;     /* base virtual address */
        struct as *v_as;    /* pointer to address space */
};
```

SEE ALSO

ddi_dma_addr_setup(9F), **ddi_dma_buf_setup**(9F), **ddi_dma_curwin**(9F), **ddi_dma_free**(9F), **ddi_dma_movwin**(9F), **ddi_dma_setup**(9F), **ddi_dma_sync**(9F), **mutex**(9F)

Writing Device Drivers